CORE
Metadata, citation and similar papers at core.ac.uk
Provided by DSpace@MIT

Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2013-024 October 8, 2013

Distributed Shared State with History Maintenance

Pavel Panchekha and Micah Brodsky

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Distributed Shared State with History Maintenance

Pavel Panchekha
University of Washington
pavpan@cs.washington.edu

Micah Brodsky
MIT CSAIL
micahbro@csail.mit.edu

October 7, 2013

Abstract

Shared mutable state is challenging to maintain in a distributed environment. We develop a technique, based on the Operational Transform, that guides independent agents into producing consistent states through inconsistent but equivalent histories of operations. Our technique, *history maintenance*, extends and streamlines the Operational Transform for general distributed systems. We describe how to use history maintenance to create eventually-consistent, strongly-consistent, and hybrid systems whose correctness is easy to reason about.

1 Introduction

Shared state in distributed systems is a persistent challenge. Traditional approaches are built mainly around preventing concurrency, preemptively (locks) or after the fact (rollback). This guarantees global consistency, at the cost of availability and latency. Recently, eventually consistent systems that trade off consistency for availability have become popular in industry [2, 5, 13, 24]. However, guaranteeing correctness in these frameworks is often a challenge.

We describe a way of structuring eventually-consistent systems, called *history maintenance*. Our method uses simple tools from the Operational Transform [9, 21] to manage concurrency and allow strong correctness guarantees despite a lack of consistency.

As a motivating example, consider a distributed file system. Each agent in the system can apply file system operations locally at any time. If agents apply operations concurrently, however, each agent will have a different history of operations; our intent is that the histories will eventually become "equivalent". For example, let one agent create a copy of some file, while another renames it. The first agent applies the operation `cp a.dat proc.input` while the second applies `mv a.dat b.dat`. After synchronization, each agent will apply a variant of the other's operation, dictated by a function called the "cross-merge". In this example, the first agent will apply `mv a.dat b.dat` while the second will apply `cp b.dat proc.input`. The two histories are different, but they lead to the same result, so the two agents still reach a common state.

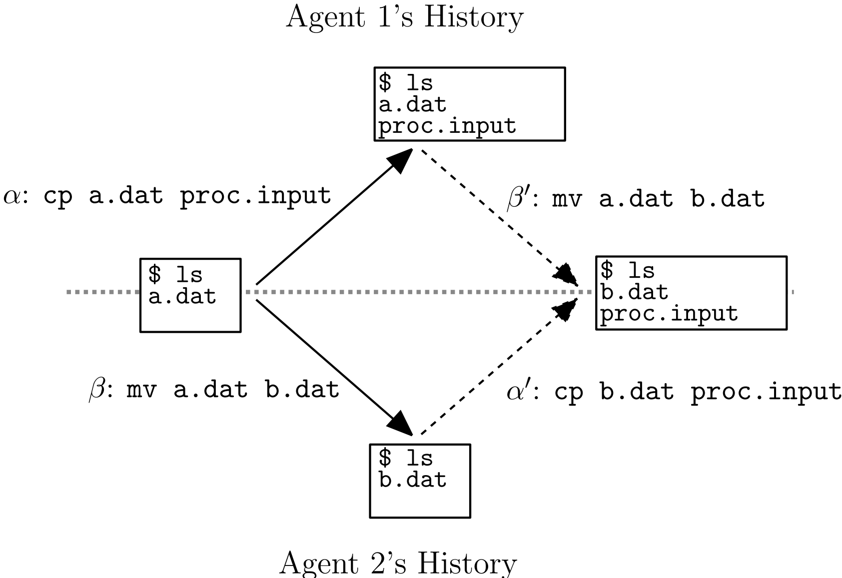


Figure 1: An example of two agents having different histories, but ending up in the same state

Conceptually, one can view these variant histories as paths in a directed, acyclic graph of operations, as in Figure 1. The first agent has a history along the top branch of the DAG, while the second agent has a history along the bottom branch. Either path results in the same state, though the intermediate states are different.

A variety of interesting data structures and semantics can be represented in our framework. Besides the distributed file system, we've encoded group membership, a variant of transactions, and several variations on common abstract data types including sets, counters, and associative arrays. Prior work on concurrent text editing using the Operational Transform [9] also fits naturally.

History maintenance is based on a framework called the Operational Transform [9, 21], a technique used in groupware and collaboration systems to synchronize simultaneous changes to documents. Operational Transforms have fostered a lot of recent industry interest, with systems like Google Docs [4,28] using them for collaborative text editing. We show how to apply these techniques to general distributed systems.

In our framework, as in the Operational Transform, we explicitly enumerate operations that may be applied to shared state. To capture the behavior of these operations in the face of concurrency, we rely on the definition of the "cross-merge", a function on pairs of operations. The cross-merge produces "variant", re-ordered versions of operations applied by other agents. In the example, the "variant" operations are derived by computing the cross-merge of the original move and copy operations. It must ensure that any two histories produce equivalent results, though the precise choice of conflict resolution behavior is up to the system designer. This allows our approach to achieve consistent state via inconsistent histories. The choice of variant operations generated in the example seemed useful, but a different cross-merge could be designed to produce different behavior.

Resolving conflicts by following distinct but equivalent paths allows each agent to act as though in isolation, shielded from concurrent operations and network latency. Operations are applied locally by every agent, independent of network latency, and synchronization happens in the background, using the cross-merge to fold remote histories into the local history.

We demonstrate a simple and efficient distributed algorithm for synchronizing shared state by using the cross-merge. It provides a strong form of eventual consistency, not only converging to a common state but also selecting a global "consensus history". As systems often have separate components requiring different levels of consistency [2, 27], we then demonstrate how to combine our system with a consensus protocol. Components that need only eventual consistency retain high availability, while other components maintain strong consistency.

In Section 2, we survey prior work on the Operational Transform and eventual consistency. In Section 3, we formalize operations and outline the desiderata for our system. In Section 4 we formalize the cross-merge, our generalization of the Operational Transform. In Section 5, we provide examples of how to encode data structures for our framework. In Section 6, we define the properties of a core abstraction, the history maintenance structure. Our replication protocol for eventually-consistent history maintenance structures is then described in Section 7. We build upon the eventual consistency in Section 8 to achieve strong consistency. We outline a file system built upon these techniques in Section 9. Finally, we outline future research in Section 10.

2 Related Work

This paper provides a method of constructing eventually consistent systems. Eventually consistent systems [22] provide at best weak guarantees on the ordering of operations, in exchange allowing an agent to respond to queries regardless of the status of other agents. Such systems sacrifice consistency for availability, a tradeoff that can be valuable in practice [2, 5, 13, 24]. A spectrum of possibilities have been explored [22]. Systems like Amazon's Dynamo [5] and Yahoo's PNUTS [2] provide eventually-consistent key-value stores; other systems, like Xerox's Bayou [20] provide a more general framework, describing how to replicate arbitrary data structures in an eventually-consistent fashion. Our work provides a general framework, with a stronger model of correctness (Strong Eventual Consistency [23]) than prior systems.

Typical eventually-consistent systems (e.g. Dynamo [5] and Bayou [20]) are built on the concept of merging two conflicting states. However, we are not aware of any consistent theoretical framework for state-merge functions. For example [1], consider a text editor. Two users start with a document

containing the string “AB”. One removes the initial “A”, while another adds characters to the original “AB” to produce “ABGAB”. A state-merge, such as `diff3` [12, 25], cannot distinguish between the second user adding “GAB” to the end (so that a merge into “BGAB” would be correct) or “ABG” to the beginning (“ABGB” would be correct). It is not clear how state-merge should approach such ambiguity. Our approach avoids this problem by looking at the history of edits, an idea we borrow from the Operational Transform.

The Operational Transform is a technique for implementing collaborative editing software [9,21]. It defines a mechanism for commuting any two editing operations. In a client-server environment, changes can be immediately applied locally. If the server orders another agent's changes first, an agent can commute the remote changes with its local changes and recover an equivalent history. Similar ideas are used in the Darcs version control system [11]. This shares many parallels with our approach, in particular our definition of the cross-merge function.

Early literature on the Operation Transform suggested it as a general technique not limited to collaborative text editing [9]. However, little work has provided guidance on how to generalize the technique beyond user documents. Early literature also attempted to provide algorithms for using Operation Transforms in peer-to-peer systems. However, these algorithms failed in some cases (see [3]); many deployments of the Operational Transform thus restrict themselves to client-server models [19, 28].

Some modern work (e.g. [18]) discusses generalizations beyond user documents, but the algorithms provided aren't appropriate for general distributed systems. Other work has focused on better formalizing the guarantees provided the Operational Transform [14, 16], in particular the property of intention-preservation [15, 26]. In this paper we build on the simpler formalism developed by Ressel et. al. [21]. Intention-preservation could be incorporated into history maintenance, but it seems less applicable to infrastructure systems.

Ressel et. al. [21] offer a formal specification of Operational Transforms and the adOPTed algorithm for distributed synchronization. However, adOPTed is insufficient for general-purpose distributed systems for several reasons. Chief among them is that adOPTed does not have a simple method to allow both eventually-consistent and strongly-consistent operations. adOPTed also relies on global broadcast of all operations, uses a version labeling scheme that assumes an unchanging set of replicas, and has less-than-ideal runtime performance properties. Furthermore, adOPTed requires modifications to support failure and recovery, and natural modifications yield particularly poor recovery performance. History maintenance corrects all of these flaws while also providing a simple way to combine eventual and strong consistency in one system.

3 Formalism

We treat our distributed system as composed of a collection of *agents*, with each agent containing some state we intend to replicate. Agents communicate on an asynchronous network guaranteeing eventual delivery but not order or timeliness[1].

An agent may freely read its copy of the shared state but can modify the state only through a designated set of *operations*. Each operation is a triple, written $a \xrightarrow{\delta} b$; δ is a function that, when applied to version a of the shared state, produces version[2] b. Initially, each agent begins with

[1]To guarantee that all agents converge, partitions must be assumed to eventually heal; otherwise, all agents on each side of a partition converge, but possibly to different states. Similarly, agents that fail permanently do not necessarily converge. We assume failures are not Byzantine.

[2]Changes may leave the shared state bitwise-identical to an earlier version. This is still considered a new version, because the intervening operations may have important semantics for concurrency.

identical shared state, at some universal initial version. a and b are encoded as *version identifiers*, as described in Section 4. A chain of operations forms a *history*, the ending version of each operation being the starting version of the next.

Since each agent may modify their shared state only by applying operations, each agent has a linear *local history* of applied operations, ordered as they were applied. Some of these operations are local operations, which an agent decides to apply because of external inputs (for example, user input). Others are applied in order to mirror changes made by other agents. Naturally, the local history is append-only, since it represents the actual temporal order of applying operations.

3.1 Properties of History Maintenance

Our goal is to produce an eventually-consistent system that provides complete availability and partition tolerance. We cannot guarantee that all agents will execute the same operations, since the agents will execute them in different orders and since the operations are not commutative. However, we can define a notion of "equivalent" histories (as we will do in Section 4) and require that all agents execute equivalent histories and converge to the same version of the state. In summary, we aim to achieve the following guarantees:

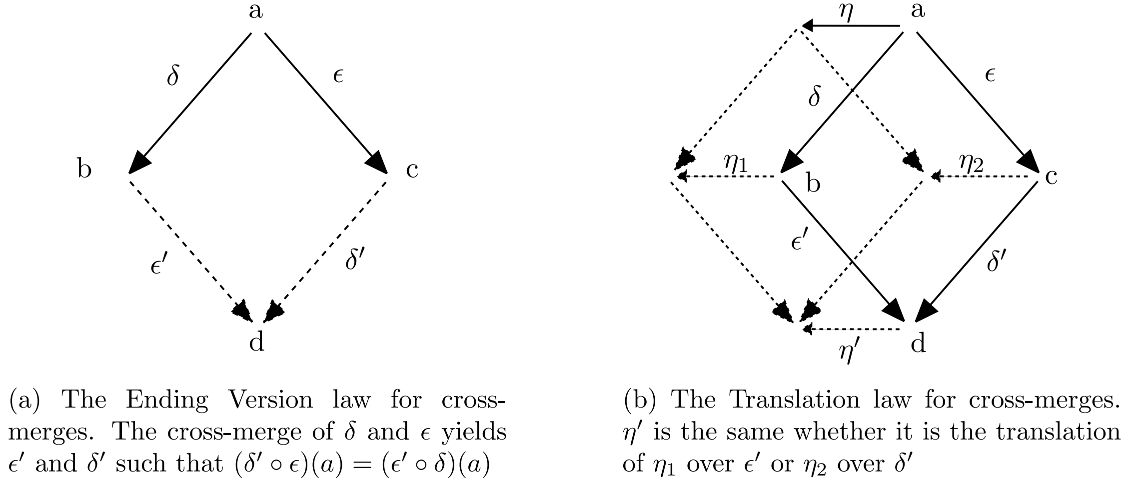
- *Availability:* An agent may apply an operation to its shared state at any time, extending its local history; the system is non-blocking. The time it takes an agent to apply an operation is independent of the latency of the network.
- *Eventual Consistency:* After a sufficiently long time with no new local operations being applied, the system will reach a state of quiescence. In this state of quiescence, all agents will have the same version of the state, and all agents' local histories will be equivalent.
- *Efficiency:* Synchronizing two replicas should take time independent of the total number of operations; instead, it should take time polynomial in the number of operations applied since the last sync.

4 The Cross-Merge

History maintenance, and the Operational Transform framework it builds upon, relies on the construction of different but equivalent histories. These are constructed by a function on operation pairs which we name the *cross-merge* (Ressel [21] calls them L-transformations). The cross-merge is defined on any two operations $a \xrightarrow{\delta} c$ and $a \xrightarrow{\epsilon} d$ (note that starting versions are the same), and must produce two operations ϵ' and δ' (see Figure 2a) with the following properties:

- *Starting Version:* ϵ' has starting version c and δ' has starting version d.
- *Ending Version:* Both ϵ' and δ' have the same ending version, f.
- *Commutativity:* The cross-merge of ϵ and δ must be δ' and ϵ'.
- *Translation:* Call ϵ' the *translation* of ϵ over δ, and likewise δ' the translation of δ over ϵ. Then for any arbitrary operation $a \xrightarrow{\eta} e$, if the translation of η over δ is η_1 and the translation of η over ϵ is η_2, then the translation of η_1 over ϵ' is the same as the translation of η_2 over δ'.

The translation requirement can be better visualized in Figure 2b. Informally, the cross-merge of two operations produces two equivalent histories, and the translation of an operation over either history should yield the same result[3].



(a) The Ending Version law for cross-merges. The cross-merge of δ and ϵ yields ϵ' and δ' such that $(\delta' \circ \epsilon)(a) = (\epsilon' \circ \delta)(a)$

(b) The Translation law for cross-merges. η' is the same whether it is the translation of η_1 over ϵ' or η_2 over δ'

Figure 2: Two of the laws for cross-merges

In our approach, the cross-merge function will encode the conflict-resolution behavior for operations on the shared state. We have defined cross-merges for many non-trivial domains, encoding interesting concurrent behavior. For example, in our distributed file system, operations include modifying, creating, deleting, and renaming files, and the cross-merge function encodes answers to questions like "What happens if one user renames a file while another moves it to a different directory?" The goal is to ensure that every agent in our distributed system eventually has a history containing equivalent operations, though the operations may differ to accommodate different orders of execution.

To formalize this notion of "equivalent" operations, we define a *family* of operations. Informally, an operation's family is the set of all operations it might turn into after a series of cross-merges. Formally, given an operation $a \xrightarrow{\delta} b$, consider the set of all operations $a \xrightarrow{\epsilon} c$; if the two cross-merge to $b \xrightarrow{\epsilon'} d$ and $c \xrightarrow{\delta'} d$, then $c \xrightarrow{\delta'} d$ is in the same family as $a \xrightarrow{\delta} b$. The family of an operation is the largest set connected only by this property. Two histories are then *equivalent* if the families of operations each contains are the same for both histories, up to order. We want every agent to have an equivalent history at quiescence.

The Ending Version property of the cross-merge shows that different histories may end at the same version. Indeed, each version will correspond to an equivalence class of histories beginning at the initial version. The cross-merge must construct new version identifiers such that equivalent histories meet at a common ending version, i.e. consistent with the Translation property. An implementation could thus identify a version by the set of families between it and the initial version. A more efficient implementation would name families by random binary strings and represent versions by the XOR of the family identifiers[4]. This would allow versions to have size logarithmic, not linear, in the number of families.

[3]This requirement is sufficient to ensure that translation over any two equivalent histories produces the same result, proven in Lemma B.1 in the Appendix.

[4]This can be implemented by naming the ending version of the cross-merge of $a \xrightarrow{\delta} b$ and $a \xrightarrow{\epsilon} c$ with $b \oplus a \oplus c$.

5 Example Cross-Mergeable Data Structures

In the previous sections, we described the motivation and formalism for the history maintenance approach to shared mutable state. In this section, we provide some examples of how to apply the cross-merge formalism to real-world problems.

5.1 Sets

Often, an application wants to track a set of items without duplicates. Our sets provide two mutating operations, put(x) and delete(x) (read operations, as usual, simply inspect the local state). The cross-merge of two put operations, or two delete operations, is easy: operations on two different items commute, and two simultaneous puts (or deletes) of the same item are idempotent. The difficult case might appear to be the cross-merge of putting and removing the same element. But, this situation cannot arise, since at the starting version, the element is either present or not, and so one of the two operations would never be issued.

We use sets like these to implement replication groups – the set stores addresses for all replicas with a copy of some data. A host that wants to join this replication group can obtain a copy of the set of members from some existing replica, establish replication relationships with the other members, and add itself to the set.

5.2 Associative Arrays

A hash table or associative array is a core data structure for many applications; it is a core component, for example, of the file system described in Section 9. Associative arrays are superficially similar to sets in providing put and delete operations, but with the added complications that concurrent puts may assign the same key conflicting values and that overwriting puts may conflict with deletes. One approach is to break ties by a total order on hosts, operations, or values: the winning operation cross-merges to itself, and the loser to a no-op. This may be appropriate for some applications; for others, destructive overwrites may be undesirable.

Another solution is to add a third type of operation: forking an entry in two. Two writes of different values to the same key then produce two copies of the key holding the two different values. For example, the cross-merge of $\delta = \mathrm{put}(k, v_1)$ and $\epsilon = \mathrm{put}(k, v_2)$ is $\delta' = \mathrm{put}(k^2, v_2) \circ \mathrm{fork}(k, k^2, k^1)$ and $\epsilon' = \mathrm{put}(k^1, v_1) \circ \mathrm{fork}(k, k^1, k^2)$. For the cross-merge of $\mathrm{fork}(k, k^1, k^2)$ and $\mathrm{delete}(k)$ operations, two subtly different choices are possible. In one, delete ensures no such key exists, cross-merging into deleting both k^1 and k^2; we call this delete-by-name. In the other, the cross-merge of delete deletes only k^1, the local copy; we call this delete-by-handle. Our file system implementation offers both.

5.3 Faux Transactions

In transactional formalisms, conflicts typically cause transactions to be aborted, rolled back, and retried. A similar mechanism can be implemented with cross-merges. Each transaction is represented by the pair (f, s), where f is the action taken on the underlying data, and s is a copy of the affected data prior to applying f. We also add a special operation, rollback(s), which sets the state to s, effectively inverting f[5]. The cross-merge of two conflicting transactions (f, s) and (g, s) then chooses a "winner" transaction, e.g. f, and produces the cross-merges $(g, f(s))$ and

[5]Embedded copies of the state can be omitted if inverses exist for all actions; rollback(s) could then be replaced by f^{-1}.

$(g, f(s)) \circ (f, s) \circ \text{rollback}(s)$ (the state is rolled back to s, then the (f, s) and $(g, f(s))$ transactions are executed). Both paths lead to the same result – $(g \circ f, s)$ – f ran first, followed by g. The choice of winner must be deterministic (e.g. embedded timestamps or unique, random identifiers) but can otherwise be arbitrary.

The key limitation of such pseudo-transactions is the lack of any guarantee that a transaction has committed: another replica may yet submit a conflicting operation that causes an automatic rollback and retry. Any history will include every transaction, but possibly with several aborts and retries before success. To provide full ACID semantics [10], an additional mechanism must be used to ensure there will be no more conflicting operations. Such a mechanism is described in Section 8.

6 The History Maintenance Structure

The evolution of the shared state is tracked by an abstraction called a *history maintenance structure*, which we abbreviate to *HMS*. A history maintenance structure stores a finite, directed, acyclic graph where the edges are operations and the vertices are versions (such as in Figures 1, 2, and 3). This DAG is monotonic (edges and vertices are only added, never removed), and the HMS maintains several invariants:

Invariant 6.1 (Equivalent Histories in an HMS)**.** *For any two versions a and b in the HMS, all paths between them must represent equivalent histories.*

Invariant 6.2 (Unique Versions in an HMS)**.** *At any time, exactly one version in an HMS has no outgoing operations (it is termed the head) and one version has no incoming operations (it is termed the initial version).*

The DAG is a subset of the lattice formed by constructing every possible cross-merge. However, we do not construct the entire lattice.

To access and manipulate this DAG, an HMS provides the following methods:

- For any version a in the DAG except the head, the HMS can produce a *future pointer* $a \xrightarrow{\delta} b$ – that is, it produces an edge leading from the given version.

  Per Invariant 6.2, the head (and only the head) has no future pointer.

- For any version a in the DAG and version b not in the DAG, the HMS can *push* a new operation $a \xrightarrow{\delta} b$. This edge is now guaranteed to exist in the DAG that the HMS represents.

  Adding only this edge would cause the DAG to have two terminal nodes, so due to Invariant 6.2 the HMS must construct additional operations via cross-merges (see Figure 3) to make both of these nodes have paths to some new, singular head. Invariant 6.1 guarantees that all paths from the initial version to the new head will contain an operation in the same family as δ (see Proposition B.1 in the Appendix for a proof).

- At any time, the path formed by following future pointers from the initial version to the current head forms a distinguished history. We call this history the *spine*; the HMS exposes this history as a random-access list.

A simple implementation of the history maintenance structure, satisfying all the above properties, is detailed in Section A in the Appendix. *push* requires time proportional to the number of edges between the starting version of the operation pushed and the current head of the HMS.

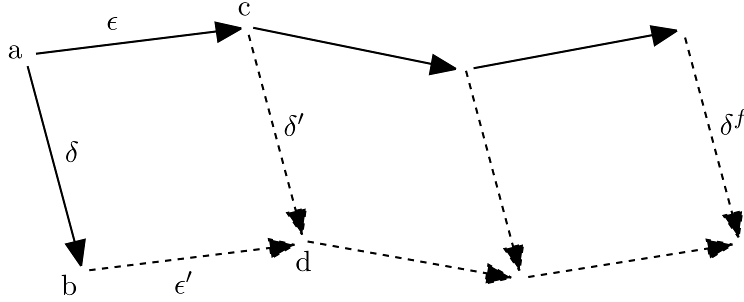


Figure 3: Pushing an operation δ onto an HMS. Dashed arrows represent operations constructed via cross-merge during the push.

A version in an HMS may have multiple operations leaving it; any of these could legally be the future pointer. While history maintenance would converge no matter the choice of future pointer, replication is made more efficient if each agent adopts a consistent choice of future pointer. We thus impose an arbitrary but global order on operation families and require that the future pointer of a version in any HMS is the outgoing operation whose family sorts first. The ordering of operation families both improves the efficiency of replication and causes agents to converge to a common spine, which lays the groundwork for mixing eventual and strong consistency.

7 Replicating a History Maintenance Structure

In distributed history maintenance, each agent has its own HMS, called its *replica*. Each agent relays new operations to others using a synchronization protocol. Our synchronization algorithm is unidirectional: one replica, the *offering replica*, sends new operations to another replica, the *receiving replica*. Agents will typically have reciprocal synchronization relationships, but this is not required (e.g. master/slave configurations). These agreements to synchronize form a graph on the agents, called the network of replicas, assumed for simplicity to be strongly connected[6]. Replicas are created by initializing a fresh HMS to the initial state and version, after which synchronization relationships may be established.

Algorithm 1 One-phase synchronization

```
▷ O is the offering replica
▷ R is the receiving replica

function OFFERING(O, R)
    s ← O.spine.copy()
    send s to R

function RECEIVING(O, R)
    receive s from O
    for a -δ-> b ∈ s do
        if b ∉ R then
            R.push(δ)
```

[6]In master/slave configurations, the network will not be strongly connected, so new operations may only be issued from the strongly-connected master component.

7.1 One-phase Synchronization Protocol

Our protocol is illustrated in Algorithm 1. The offering replica sends the receiving replica a copy of its spine. The receiving replica then pushes each operation in turn (unless already present), guaranteeing that the receiving replica has every family that the offering replica has. This allows families of operations to spread from agent to agent, forming the basis of eventually consistent replication, discussed in Section 7.3.

Theorem 7.1. *Every family represented in the offering replica's HMS at the start of the sync is represented in the receiving replica's spine at the end of the sync.*

Proof. Every family in the offering replica's HMS is on its spine by Invariant 6.1; a copy of this spine is sent to the receiving replica. Every operation in this copy is then pushed onto the receiving replica, unless the ending version of the operation already exists. If the ending version does not exist, the operation's family is pushed onto the receiving replica and is thus on the receiving replica's spine (see Proposition B.1 in the Appendix). If the ending version does exist, then the operation's family already exists in the receving replica, since a version corresponds to a fixed set of families between itself and the initial version. Thus all families present in spine copy are guaranteed to be in the receiving replica upon the completion of the synchronization. □

Theorem 7.2. *Families appear in the same relative order on any replica's spine.*

Proof Sketch. We show that one family preceding another on a given replica's spine is equivalent to a property that is independent of which replica we are discussing. Then the relative order of any two families must be the same for all replicas. We claim that one family δ precedes another ϵ if a) the user operation with family ϵ starts at a version that incorporates the family δ, or otherwise b) if δ precedes ϵ in the global family order. The proof is by induction over the number of replicas containing families δ and ϵ. If there are none, the claim is vacuously true. If there is only one, it must be true by the construction of the spine. Finally, to prove the inductive case we show that this property is unchanged by synchronization.

The case (a) cannot be changed by cross-merging, so the relative order of two such families is unchanged by synchronization. The case (b) won't be changed by cross-merging, since the future pointer of an operation is always the operation whose family comes first the global order, and only the chain of future pointers is shared with other replicas. So, this case is also unchanged by synchronization. Thus, the relative order of any two families is defined by cases (a) and (b) no matter how many replicas have a copies of δ and ϵ. Then their relative order is equivalent to a replica-independent property and thus must be the same for all replicas in the system. □

7.2 Improved Synchronization Protocol

The protocol presented above requires the offering replica to send the receiving replica an entire history from initial version to head; as a result, syncs take progressively longer. To remedy this, we take advantage of consistent ordering on families to send only the part of the spine that is not shared. Our improved protocol can be seen in Algorithm 2. We break the protocol into two phases. In the first phase, the replicas locate (by expanding binary search) a version on the offering replica's spine that is present in the receiving replica. The receiving replica's history from the initial version to this common version must be equivalent to the offering replica's, so it is not necessary to send the receiving replica this prefix. Sending the remaining partial history is then the second phase of the protocol.

Theorem 7.3. *The result of this improved synchronization protocol is the same as that of the one-phase synchronization protocol.*

Proof. All operations whose pushes are omitted by the new protocol lead to a version present in the receiving replica. These operations form a history identical to one that already exists in the receiving replica, due to Theorem 7.2. The check that prevents existing operations from being pushed would already prevent any of these operations from being pushed. Thus both protocols push the same sequence of operations and so their effects must be the same. □

We'd like to bound the runtime of our synchronization protocol. To do this, we define the *staleness* of a version at a given HMS to be the number of operations in the HMS between that version and the head of the HMS. Pushing an operation $a \xrightarrow{\delta} b$ to an HMS takes time linear in the staleness of a (see Proposition A.2 in the Appendix). We can use this to bound the time required to run one instance of the synchronization protocol in terms of the staleness of the common version found in the first phase of the protocol.

Theorem 7.4. *Let c be the least-stale common version on the offering replica's spine, s_0 its staleness in the offering replica, and s_1 its staleness in the receiving replica. Then, one instance of the synchronization protocol between these replicas takes $O(s_0 s_1)$ time, if network requests are assumed to take bounded time.*

Proof. Finding the common version is done in $O(\log s_0)$ time (including $O(\log s_0)$ network requests). $O(s_0)$ operations are pushed, and each push takes $O(s_1)$ time by Proposition A.2. Thus the total time is $O(s_0 s_1)$. □

Theorem 7.5. *Let c, s_0, and s_1 be as in Theorem 7.4. Synchronizing two replicas uses $O(\log s_0)$ network round trips and $O(s_0)$ total data sent over the network, assuming operations are bounded in size.*

Proof. The common version is found by iteratively testing versions with staleness 2^i (requiring $O(\log s_0)$ round trips), followed by a binary search between this version and the head version to find the least-stale common version (also $O(\log s_0)$ round trips). Each of these network requests is constant-size. After a common version is found, the offering replica sends the receiving replica s_0 operations, which takes $O(s_0)$ total data and one round trip. Thus the total is $O(\log s_0)$ round trips and $O(s_0)$ total data. □

Algorithm 2 Two-phase synchronization

```
▷ O is the offering replica
▷ R is the receiving replica

function OFFERING-PHASE1(O, R)
    i ← O.spine.length, j ← 1
    ▷ i is spine length at start
    ▷ j is the staleness tested
    found ← false
    while ¬found do
        send O.spine[i − j] to R
        receive found from R
        j ← 2j          ▷ Exponentially larger steps
    Binary search for a k between i − j and i,
    testing indices by sending O.spine[k] to R.
    send break loop to R
    Move to OFFERING-PHASE2

function RECEIVING-PHASE1(O, R)
    loop
        receive version s or break loop from O
        send [s ∈ R] to O
    Move to RECEIVING-PHASE2

function OFFERING-PHASE2(O, R, k)
    ▷ O.spine[k] is the common version.
    s ← O.spine.copySlice(k, O.spine.length)
    send s to R

function RECEIVING-PHASE2(O, R)
    receive s from O
    for a –δ→ b ∈ s do
        if b ∉ R then
            R.push(δ)
```

7.3 Eventually Consistent Replication

Algorithm 1 describes how to synchronize two replicas. To synchronize an entire network of replicas, we can use a simple gossip protocol [6, 17]. By Theorem 7.1, we can consider each family individually, spreading promiscuously through the network. Each replica regularly synchronizes with its neighbors, thus passing on all families it knows about. If the network of replicas stays connected, the usual properties of gossip algorithms hold. In particular, after a sufficiently long period with no new families produced, every existing family will spread to all replicas, and thus all replicas eventually converge to the same version. In a synchronous model, this ensures convergence within d rounds of communication, where d is the diameter of the network (see Theorem B.1). Furthermore, due to Theorem 7.2, all replicas' spines will be identical.

However, few production systems have periods of total quiescence. To show that our system is robust to these environments, we consider a more limited form of quiescence we term *prefix quiescence*. Informally, prefix quiescence occurs when sufficiently old operations are not disturbed by new updates; our system then guarantees that the prefix of the spine containing those operations converges. Formally, we call a set of families a *prefix* of a given version s if histories between the initial version and s contain representatives of each family in the set. We then say the system has reached prefix quiescence on some set of families P if for all local operations $a \xrightarrow{\delta} b$ executed by any replica, P is a prefix of a. Our protocol guarantees that if prefix quiescence is reached on some prefix P, all replicas will eventually have representatives of all families in P, in the same relative order, regardless of what other operations are executed. This can be seen by considering only the families in the prefix and relying on the ordering of families and Theorem 7.2.

8 Strong Consistency

Applications often require a mix of strong and weak consistency semantics [27]. For example, creating a new post in a social network can be propagated with eventual consistency, but changing the visibility of a post, or deleting one, needs strong consistency to provide useful guarantees [2]. In history maintenance, prefix quiescence guarantees the eventual existence of a common serializable history. We can apply standard consensus protocols to ratify this history, transforming the eventually consistent history into a durable and strongly consistent one.

To commit an operation $a \xrightarrow{\delta} b$, each replica must guarantee that a) this operation is the first uncommitted operation on its spine, and b) the replica will not push any more operations starting at a. When satisfied globally, these two requirements guarantee prefix quiescence for that operation. We can keep track of currently-available replicas with a view-change protocol [7, 8]; commits use a standard two-phase commit protocol. Blocks of operations can be committed together for efficiency.

At any time, any replica may request that the next uncommitted operation on its spine be committed. Any replica that cannot provide the guarantees (a) and (b) blocks the commit; if all replicas agree, the replica that proposed the commit sends a final message actually committing the operation. After receiving this second message, all replicas mark the operation committed.

We need to specify the behavior of replicas partitioned away from the main quorum, however. Such replicas can still operate the HMS replication protocol to guarantee eventual consistency among themselves but cannot commit operations with strong consistency. When the partition dissolves, they need to be re-integrated into the replication group, and the operations executed while partitioned need to be incorporated into the common history.

To integrate a replica back into the main replication group takes three steps. First, the replica rejoins the main replication group when the main replication group executes a view change. Next,

the new replica requests the sequence of committed operations from any replica in the main replication group. It then replays its uncommitted operations onto this spine (so that already-committed operations sort first). The new replica's spine now begins with the sequence of committed operations, and can start allowing operations to be committed. Finally, the new replica begins to synchronize normally with members of the main replication group. Eventually, the operations it had made while partitioned will be committed.

Committing prefixes of the spine has several applications. One is to selectively execute certain operations with strong consistency. To execute with strong consistency, a replica pushes the operation and then repeatedly tries to commit it, returning to the application only when the commit is successful. For full ACID semantics, the faux transactions of Section 5.3 could be executed with strong consistency. Another way to make use of the strong consistency subsystem is to automatically commit blocks of old operations. These operations can then be exported for logging and backup, and eventually discarded to free space. One might term this mode of operation "eventual strong consistency".

In discarding old operations, two different cases must be considered. Once an operation is committed, all other operations from the same family are guaranteed never to appear on the spine of any replica. These side branches can be discarded immediately. Discarding operations on the spine, however, must be done with caution. If operations on the spine are discarded, replicas that rejoin after a sufficiently long partition may be unable to replay all their local operations. If reasonable bounds exist on how long replicas may be partitioned, operations older than this can safely be discarded.

9 HMS-FS

As a larger-scale example of a service implemented in the history maintenance framework, we've implemented a rudimentary peer-to-peer file system, called HMS-FS, using history maintenance.

HMS-FS supports the standard set of file system operations: creating and deleting files, renaming and moving them, and traversing a directory tree. HMS-FS focuses on the directory tree itself, not file contents; operations on file contents (and their cross-merges) are defined by user programs. However, as a peer-to-peer distributed file system, HMS-FS needs to make decisions on how to handle concurrent changes. What ought to happen, for example, if a file is simultaneously renamed and moved to a different directory? If a file is simultaneously renamed and changed? Or renamed to two different names by two different users?

HMS-FS presents a view of the world to each user wherein every operation completes sequentially and successfully. When a user refreshes their view of the file system, they see others' changes as coming *after* their own, even if they happened earlier. How to apply others' changes is determined by the design of the cross-merge function.

HMS-FS tries to answer the above questions about concurrent changes with inspiration from Unix file system semantics. For example, if one user modifies a file that another renames, the result ought to be a modified file under the new name. Generalizing beyond Unix semantics, if one user renames a file that another moves, we also resolve this to a renamed file in the new directory. Overall, we support seven file-system operations: create and delete files, create directories, move and rename files, create hard links, and perform user operations on file contents. However, we were loath to code 49 individual cross-merges.

Instead, we encode all operations in terms of three fundamental operations: link, unlink, and "do", which performs operations on file or directory contents. This simplifies the number of cross-merges but removes semantic information about why operations are happening. For example,

suppose one user moves `a.txt` to a new folder[7], while another renames it to `b.txt`. The rename is represented by linking `b.txt` and unlinking `a.txt`, so we begin resolving the conflict by cross-merging the linking of `b.txt` and the move of `a.txt` to another folder. But without knowing that the link operation is part of a rename (as opposed to creating a second entry for the same file), how is the move operation to know that the target's name is to be changed to `b.txt`?

We resolve this issue by annotating each link, unlink, and "do" operation with a description of the higher-level operation it is a part of. Renaming a file, for example, is still a link and unlink pair, but each operation describes the full rename operation. This allows us to specify default conflict resolution for link, unlink, and do, and then to special-case behaviors as we see fit. Instead of 49 cross-merges to write, we have nine cross-merges and approximately a dozen special cases. We think that combining simple primitive operations with semantic annotations makes a good, general design strategy for HMS operations.

We want users to have control over the visibility of other users' concurrent changes. Thus, we only apply others' changes when requested: users explicitly update their view of a file or directory (or script this to happen on, say, `ls`). To implement this, the shell keeps track of the version the user has last synchronized to; we call this the user's "read token". User operations are applied at the read token, which is advanced after every operation. Since files and directories are stored in HMSs, we can walk along future pointers starting at the read token in order to incorporate concurrent changes.

The simplest way to implement HMS-FS would be to store the entire file system in a single HMS. However, this has limited scalability: each host must replicate the entire file system and resolve cross-merges over the entire tree, even for widely-separated files with no relationship. To improve scalability, one can partition the file system into multiple HMSs. Many partitioning schemes are possible: allow administrators to assign static partitions, assign one global HMS to the directory tree structure and one to each file, or separate every directory and every file. In our implementation of HMS-FS, we chose the last and most aggressive scheme. As a result, we cannot make atomicity guarantees about cross-directory operations or coordinated file changes. Atomicity guarantees for cross-HMS operations remain a subject of ongoing research.

To make use of the partition scheme, we want different hosts to be able to replicate different subsets of the file system. The root can be located by fiat, but how is a host to begin replicating a fragment it does not already replicate? We pair a unique replication group with each file and directory, implemented as in Section 5.1. Each replication group identifies who has a live copy of that item. To replicate a file a host does not yet have a copy of, we make use of the file system's hierachical structure. For every item a host replicates, we also require that host to replicate the directories leading to that item from the root. Then, to find a host with a copy of an item, it simply queries all hosts (by brute force or more cleverly) replicating the containing directory. Once one host with a copy is found, that item's replication group can be joined in the usual way to replicate with all interested hosts. This strategy can be generalized straightforwardly to other hierarchical partitioning schemes.

10 Future Work

We have described a novel approach to managing shared state in a distributed system. There remain several interesting avenues for future research. One can build dynamically extensible cross-merge functions, by specifying the semantics of an operation in the operation itself; we'd like to

[7] In our implementation, moving a file a (with id i) to directory d is implemented with $\mathrm{do}(d, \mathrm{link}(a, i)), \mathrm{unlink}(a)$, but this does not affect the example.

explore this idea. We'd also like to better understand the theory of multiple independent HMSs managing interacting shared state (as in HMS-FS) and to support joint operations across multiple HMSs. Incorporate stateless computation on the data in an HMS, similar to database views and "functional reactive programming", marks a third avenue. However, we believe the approach as described is already practical for real-world systems, and we plan to explore this space next.

Acknowledgments

We'd like to acknowledge G. J. Sussman for his guidance throughout the development of history maintenance, as well as A. Radul, I. Jacobi, K. Censor-Hillel, S. Sastry, and N. Shavit for their helpful feedback on the design and presentation.

References

[1] Conner, R. Git is inconsistent. `http://r6.ca/blog/20110416T204742Z.html`, 2011.

[2] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (Aug. 2008), 1277–1288.

[3] Cormack, G. V. A calculus for concurrent update (abstract). In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 269–.

[4] Day-Richter, J. Whats different about the new google docs: Making collaboration fast. `http://googledocs.blogspot.com/2010/09/whats-different-about-new-google-docs_23.html`, 2010.

[5] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[6] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing* (New York, NY, USA, 1987), PODC '87, ACM, pp. 1–12.

[7] El Abbadi, A., Skeen, D., and Cristian, F. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1985), PODS '85, ACM, pp. 215–229.

[8] El Abbadi, A., and Toueg, S. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems 14* (1989), 240–251.

[9] Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. *SIGMOD Rec. 18*, 2 (June 1989), 399–407.

[10] Haerder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (Dec. 1983), 287–317.

[11] Jacobson, J. A formalization of darcs patch theory using inverse semigroups. *CAM report 09-83, UCLA* (2009).

[12] Khanna, S., Kunal, K., and Pierce, B. A formal investigation of diff3. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, V. Arvind and S. Prasad, Eds., vol. 4855 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2007, pp. 485–496.

[13] Lakshman, A., and Malik, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[14] Li, D., and Li, R. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (New York, NY, USA, 2004), CSCW '04, ACM, pp. 457–466.

[15] Li, D., and Li, R. An admissibility-based operational transformation framework for collaborative editing systems. *Comput. Supported Coop. Work 19*, 1 (Feb. 2010), 1–43.

[16] Li, R., and Li, D. A new operational transformation framework for real-time group editors. *Parallel and Distributed Systems, IEEE Transactions on 18*, 3 (2007), 307–319.

[17] Lin, M., and Marzullo, K. Directional gossip: Gossip in a wide area network. Tech. rep., La Jolla, CA, USA, 1999.

[18] Molli, P., Oster, G., Skaf-Molli, H., and Imine, A. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work* (New York, NY, USA, 2003), GROUP '03, ACM, pp. 212–220.

[19] Nichols, D. A., Curtis, P., Dixon, M., and Lamping, J. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology* (New York, NY, USA, 1995), UIST '95, ACM, pp. 111–120.

[20] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), SOSP '97, ACM, pp. 288–301.

[21] Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work* (New York, NY, USA, 1996), CSCW '96, ACM, pp. 288–297.

[22] Saito, Y., and Shapiro, M. Optimistic replication. *ACM Comput. Surv. 37*, 1 (Mar. 2005), 42–81.

[23] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[24] Shoup, R., and Travostino, F. ebay's scaling odyssey: Growing and evolving a large ecommerce site. LADIS '08.

[25] Smith, R. *DIFF3(1) Manual Page*, diffutils 3.3 ed., March 2013.

[26] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact. 5*, 1 (Mar. 1998), 63–108.

[27] Terry, D. Replicated data consistency explained through baseball. Tech. rep., Microsoft Research, 2011.

[28] Wang, D., Mah, A., and Lassen, S. Google wave operational transformation. Tech. rep., Google, 2010.

A Implementation of the History Maintenance Structure

In abstract, an HMS represents a directed, acyclic graph, with operations as edges and versions as vertices. In practice, rather than storing the graph explicitly, we maintain only a mapping from versions to future pointers. Future pointer queries are a simple lookup.

To implement the push operation, we have two cases. Suppose we push $a \xrightarrow{\delta} b$ onto an HMS. If a is the head, we simply set a's future pointer to be $a \xrightarrow{\delta} b$. If, on the other hand, a is not the head, it must have a future pointer. We resolve this case recursively. Let the current future pointer be $a \xrightarrow{\epsilon} c$ (as in Figure 3). Then we can cross-merge $a \xrightarrow{\delta} b$ and $a \xrightarrow{\epsilon} c$ to produce $b \xrightarrow{\epsilon'} d$ and $c \xrightarrow{\delta'} d$. Since the path from c to the head is shorter than that from a to the head, we can recursively push $c \xrightarrow{\delta'} d$ onto the HMS. Informally, to push δ onto the HMS, we simply add δ and ϵ' to the map and recursively push δ'. To maintain the correct choice of future pointer, adding δ to the mapping may or may not update the future pointer of a.

To show that this algorithm satisfies the definition of an HMS, we must prove that invariants 6.1 and 6.2 hold.

Proposition A.1. *If invariants 6.1 and 6.2 hold for a given HMS before a push, they hold after the push as well.*

Proof. We prove the two invariants separately.

Invariant 6.1: The push algorithm uses only cross-merges to construct new operations. So, this invariant follows from the properties of versions, which prevent versions arrived at via non-equivalent histories from being identical.

Invariant 6.2: In the non-recursive case, pushing $a \xrightarrow{\delta} b$ onto an HMS where a is the head makes b the head. In the recursive case, all newly added versions except the head have an outgoing edge, and all have an incoming edge; the head is added by the last recursive push, which falls back to the first case. □

From the recursive nature of the algorithm, the run time is straightforward:

Proposition A.2. *Each push onto an HMS takes time proportional to the starting version's staleness.*

The spine can not have changed more elements than the staleness of the pushed operation. The spine vector can thus be maintained without increasing the runtime of the algorithm as a whole.

B Additional Proofs

Let two histories be *similar* if both their first and last operations are the same. Invariant 6.1 then guarantees that similar histories in an HMS are equivalent. We can now prove an important proposition abstracting the action of an HMS:

Proposition B.1. *Once an operation $a \xrightarrow{\delta} b$ is pushed onto an HMS which contains the version a, any history in that HMS between the initial version and the head contains an operation in the family of δ.*

Proof. By Invariant 6.1, all histories between two fixed points are equivalent. Consider the history formed by the concatenation of any history between the initial version and a, then the operation $a \xrightarrow{\delta} b$, and then any history between b and the HMS's (new) head. The first of these must exist

since the HMS contains the version a, since the graph is finite and acyclic, and since there is a single node with no incoming edges (by Invariant 6.2). The second must exist as it was just pushed onto the HMS. The last must exist, as it can be formed by following future pointers from b until the head is reached. This produces a finite history ending at the head because the DAG is finite and acyclic, and since the head is, by Invariant 6.2, the only terminal node.

Since the given history exists and contains an operation in the family of δ, so must any similar history. Any history between the initial version and the head is similar to this one, so all contain an operation in the family of δ. □

B.1 A Generalized Translation Law

Lemma B.1. *The translation of an operation $a \xrightarrow{\delta} b$ over any history between a and c in any HMS is independent of the history chosen.*

Before proving this lemma, we need to define a simple concept. Two operations are termed *concurrent* if neither's family is in a prefix of the other's starting state. By the proof of Theorem 7.2, the relative order of the two on the spine is decided by the global order of families; thus, the two must be cross-merged at some point.

Proof Sketch. Any two histories H and G between a and c must be equivalent, by Invariant 6.1. Any two families whose relative order differs between H and G must be concurrent and thus a result of a cross-merge. If they occur sequentially in G, then following the other branch of the cross-merge will effectively reverse their relative order; by the Translation law for cross-merges, translation over this path is the same as translation over G. By a sequence of such swaps, we can produce a new history, translation over which is identical to translation over G, but with all families in the same order as H (such a sequence of swaps exists by the properties of permutations). We need only show that these two histories, with the same families in the same order, are indeed the same history.

We prove this inductively on subsets of HMSes. Note that any family is generated by the translations of a single user operation. If the two histories are not the same, there is a first version at which the two include differing operations. Since these two operations are in the same family, they must be generated by different translations of one operation over similar histories within the sub-HMS ending at that version. That sub-HMS must not obey translation, in order to produce differing operations. In order to violate translation, the sub-HMS must include within it some pair of histories that, when permuted to the same order, are non-identical. This is impossible by induction. Thus, any two all translations of an operation over similar histories must yield the same operation. □

B.2 Eventual Consistency Theorems

It is clear that after sufficiently many synchronizations without new operations being executed by any replica, the system will reach a state of consistency with every family being present in every HMS (and thus every HMS storing an equivalent history). However, we would like a bound on how much synchronization is necessary – that is, how "eventual" our eventual consistency is. To better analyze this situation, we temporarily adopt a synchronous model of communication, where agents communicate in rounds. During any round, an agent synchronizes with all agents adjacent to it in the network.

Theorem B.1. *Let d be the diameter of the network. If d rounds of communication proceed during which no replica executes new operations, all replicas will reach the same shared version.*

Proof Sketch. Pick a canonical source replica for each family. After each round, by Lemma 7.1, every family present in a replica at the start of the round is guaranteed to be present in all its neighbors. Since d is the greatest distance between two nodes, every family will be present in every replica after d rounds. Thus they will all have equivalent histories. Since replicas' spines always have the families sorted in the same order (by Theorem 7.2), all replicas' spines are in fact identical (there can only be one operation in a given family starting at a given version by Lemma B.1). These spines must terminate in the same version, so all replicas' heads are the same version. □

Of course, it is possible that our protocol would enforce eventual consistency in a synchronous model, but not an asynchronous one. To prove the eventual consistency of our protocol in an asynchronous model, we need to assume that any two neighboring replicas will eventually communicate.

Lemma B.2. *In an asynchronous network, any family will eventually be present in every replica.*

Proof Sketch. The set of replicas that contain a given family only expands. If there are any replicas that do not contain a given family, there must be some such replica A that is adjacent to a replica B that does contain this family. B will eventually synchronize to A, after which A will contain this family. Inductively, there will eventually be no replicas not containing this family. □

Theorem B.2. *In an aysnchronous network, all replicas will eventually have equivalent histories and a common version after a sufficiently long period of quiescence.*

Proof Sketch. All replicas will eventually contain all families. After the system enters quiescence, no new families will be created, so eventually all families will be present in all replicas. Then they must have equivalent histories from the initial version to their heads. By a similar argument to the one in Theorem B.1, all replicas must thus share a common version at their head. □