Communication Timestamps for File System Synchronization

Russ Cox
William Josephson

rsc,wkj@eecs.harvard.edu

ABSTRACT

The problem of detecting various kinds of update conflicts in file system synchronization following a network partition is well-known. All systems of which we are aware use the version vectors of Parker *et al.* These require $O(R \cdot F)$ storage space for F files shared among R replicas. We propose a number of different methods, the most space-efficient of which uses $O(R \cdot F)$ space in the worst case, but $O(R + F)$ in the expected case.

To gain experience with the various methods, we implemented a file synchronization tool called Tra. Based on this experience, we discuss the advantages and disadvantages of each particular method.

Tra itself turns out to be useful for a variety of tasks, including home directory maintenance, operating system installation, and managing offline work. We discuss some of these uses.

Introduction

Anyone who uses more than one computer system is aware of the data management problem posed by doing so: sharing files between systems requires propagation of changes to the other systems by some synchronization method. One solution is to avoid the need for synchronization, keeping all files on one system and accessing them via a network file service or remote login sessions. This is unsatisfactory, because it assumes continuous connectivity to the central system. The ability to operate in the face of network partition, whether intentional (as in the case of mobile computing) or unintentional (as in the case of failures), is highly desirable. Another solution is to synchronize manually, moving files by hand as necessary. This solution too is unsatisfactory, despite its apparently widespread usage: manual data synchronization is tedious, time-consuming, and error-prone.

In response to these problems, network file systems such as AFS [4] and Coda [3] provide support for disconnected operation, in which a local cache provides file service and then is synchronized with the central server when the network connection is reestablished. This works well for the case of members of a workgroup taking files with them on a laptop when they go home or on business trips, but the overhead of maintaining a central server makes the approach unappealing for personal use. It is also sometimes awkward or impossible to provide all machines with connectivity to a central server. For example, consider someone whose only connection between his desktop machines at home and at work is a laptop carried back and forth. While in this case the laptop could conceivably be made the central server, in more complicated scenarios there is no candidate at all for a central server.

The Rumor system [2] is one attempt to address such scenarios. In the Rumor model, files are replicated among a network of systems, with no one system being the central point of truth for any given file. Rumor does not assume that each machine can directly communicate with every other machine, but rather that between any pair of machines, there is some perhaps indirect path along which changes can propagate. Systems such as Unison [1] and Microsoft's Briefcase use a peer-to-peer model but only allow a single pair of replicas. All three systems have the added benefit of being implemented without additional support from the operating system, making them more portable. In particular, Rumor is available for Linux and FreeBSD; a Windows port is rumored to exist.

Unfortunately, the difficult part of synchronization is the detection of independent changes to copies of the same file on different replicas and the subsequent resolution of these conflicts. The frequency of conflicts has been measured to be quite small [3], so we take the attitude that most resolution can be left to the user, concerning ourselves with the problem of detecting conflicts. The most common detection method for a system of F files shared among R replicas requires $O(R \cdot F)$ space per replica. We propose a new method that requires $O(R \cdot F)$ space in the worst case, but only $O(R + F)$ space in the expected case. Our method also enables a novel approach to the detection of conflicts involving deleted files. We then examine a number of hybrids of the two methods.

To test the various methods, we wrote a file synchronization tool called Tra. We hope to employ Tra in our day-to-day use of a handful of systems running a handful of operating systems.

In what follows, we present two related formalizations of the conflict detection problem and prove their equivalence. We present both formalisms because we expect that one of the two will be significantly more intuitive depending on the reader's background. We also give a summary of the most common method, due to Parker *et al.*, along with its proof of correctness. Then we present our method and its proof of correctness. Finally, we describe the architecture of Tra and reflect on the various methods' pros and cons, closing with a discussion of future applications of both the theory and the software.

A Theory of Conflicts

Conflict detection reduces to answering the question, ''here are files from two replicas; is it safe to synchronize by simply copying one onto the other?''. We summarize the theory of conflicts used by Parker *et al.* [5]; our own modifications to the theory are noted as such. It is assumed that nothing is known about the semantics of the file contents. In some cases, such knowledge will enable simpler approaches, but this theory is interested in handling the general case.

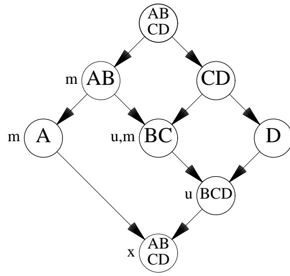
The first problem lies in naming among the distributed replicas. Specifically, the simultaneous renaming or creation of files in multiple replicas can create situations in which files on different replicas share a name but are unrelated. As a solution, Parker proposes the use of *origin points*, which serve as unique identifiers for files and last for the entire lifetime of the system. A candidate for such an origin point might be the name of the creating replica along with the time of creation (assuming the granularity of time is fine enough that no two files will have the same creation time). Origin points are unique in the sense that two perhaps differing files are derived from a common earlier version if and only if they have a common origin point. A *name conflict* occurs when two replicas contain files with the same name but different origin points: these files have been independently created, so it is not safe simply to copy one over the other.

The second problem is that of identifying simultaneous updates to a file by multiple replicas. For example, if A and B change their copies of the file independently, a conflict should be detected when they try to synchronize. On the other hand, if A changes its copy, A and B synchronize, B changes its copy, and then A and B synchronize again, there should be no conflict even though the file contents at the time of the second synchronization may be identical to those in the synchronization in the last example. Precise definition of these conflicts requires more terminology.

A *modification id* is a unique identifier for a modification to a file at some point during its life. The *modification history* associated with a version of a file is the ordered list of modification ids for all modifications made to the file. It is safe to replace one version of a file with another if the first version's modification history is a prefix of the second's since the second must necessarily be the result of further modifications to the first.

Partition Graphs

Parker *et al.* present conflicts in terms of characteristic graphs called *partition graphs;* such a graph, denoted $G(f)$ for a file f, is a directed acyclic graph in which each node represents a synchronized partition at a given time. The arrows, or edges of the graph, connect partitions sharing participants. Since nodes represent partitions, each replica listed in a node must be in exactly one parent and one child node. The exceptions, of course, are that the source has no parent and the sink no children. The represented series of communications must be serializable, in the sense that there exists an equivalent sequence of sequential synchronizations.



Remember that the graph $G(f)$ is for a particular file, not the entire shared set of files. A small ''m'' next to a node indicates the file was modified in that partition, a ''u'' indicates an update without conflict, and an ''x'' indicates that a conflict needs to be resolved as part of the synchronization that created the partition.[1] Parker *et al.*'s system assumes synchronized partitions of arbitrary size which break and recombine. In this example drawn from their paper, the file is modified in the *AB*-only partition, the new *AB*-only version is propagated by *B* to the *BC*-only partition, and then modified again. Meanwhile, *A* has taken the *AB*-only version and modified it independently. The *BC* change propagates to *D* in the *BCD* partition, and when *A* and *BCD* attempt to re-synchronize, a conflict is detected due to the earlier independent modifications.

As the example suggests, in a partition graph, a conflict must be reconciled at node P if and only if (a) P has two distinct ancestor nodes P_1 and P_2 at which modifications were made,[2] and (b) P_1 and P_2 have no common descendant that is also an ancestor of P. The proof is simple.

1. If there exist no distinct ancestors P_1 and P_2 with modifications, the histories of the parents would be identical and there would be no conflict. That is, if (a) does not hold, there is no conflict.
2. If for every pair of P's ancestors P_1 and P_2 with modifications, there is a common descendant of theirs that is an ancestor of P, then one of P's parents would also be a descendant of both P_1 and P_2, but that parent's modification history would contain both of the earlier modifications, so those modifications would not cause a conflict at P. That is, if (b) does not hold, there is no conflict.
3. Since P_1 and P_2 are distinct, they have incompatible modification histories. In particular, the history at P_1 contains the modification (P_1, t_1) but not (P_2, t_2), while the history at P_2 contains (P_2, t_2) but not (P_1, t_1). Since P_1 and P_2 have no common descendant that is also an ancestor of P, neither of P's parents' histories can contain both (P_1, t_1) and (P_2, t_2): each contains one, and thus the parent's histories are incompatible, so there is a conflict at P. That is, if (a) and (b) hold, then there is a conflict.

Thus, there is a conflict if and only if both (a) and (b) hold.

Partial Orders

Another way to think about conflicts is as a problem of partially ordered sets. The ''is a prefix of'' relation on modification histories imposes a partial order (and hence a lattice structure) on the set of modification histories of files in the system. We will write $P \leqq Q$ when P's modification history[3] is a prefix of Q's and will write $P = Q$ to denote equality. For a node P with two parents P_1 and P_2, there is a conflict if neither $P_1 \leqq P_2$ nor $P_2 \leqq P_1$. That is, if the two partitions are incomparable under the relation, neither associated history can be a prefix of the other. When $P_1 \leqq P_2$, the P_2 copy should be chosen, and vice versa.

Moreover, the relation defining the partial order can be derived from the partition graph of the preceding section: whenever one node in that graph is a parent of another node, it means the file propagated from child to parent, so the parent's modification history must be a prefix of the child's and so it follows that $P \leqq Q$. If a file passed from node P to Q and was not changed at Q, then the modification history at P is the same as the modification history at Q and so we must have $P = Q$. Conversely, if a file passes from P to Q and Q modifies it, $P \leqq Q$, but the converse fails to hold.

If P has parents Q_1 and Q_2, and a conflict occurs at P, this means that neither $Q_1 \leqq Q_2$ nor $Q_2 \leqq Q_1$, *i.e.,* Q_1 and Q_2 are incomparable under the partial order. Each history must contain a modification not in the other. More formally, there must exist modifying nodes P_1 and P_2 such that $P_1 \leqq Q_1$ and $P_2 \leqq Q_2$. but neither $P_1 \leqq Q_2$ nor $P_2 \leqq Q_1$. These conditions are exactly those laid out in the partition graph definition of conflict.

---

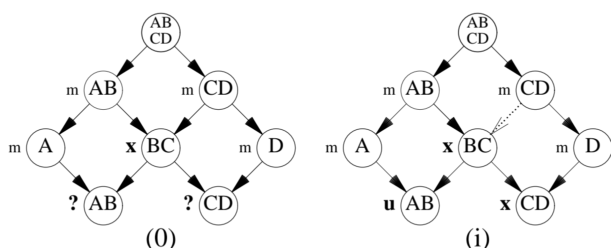
[1] Parker *et al.* use a small ''+'' for both ''m'' and ''x''; we feel our notation is clearer, and distinguishing these cases has advantages noted later.

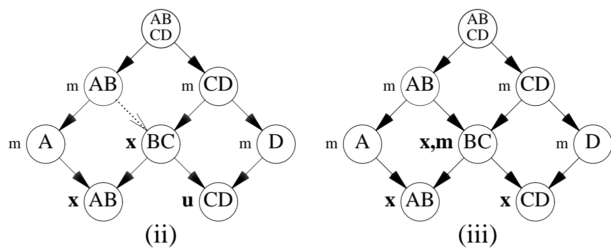
[2] The reader familiar with Parker *et al.*'s paper will at this moment be wondering why we have written ''modifications'' rather than ''modifications and/or reconciliations,'' as is used therein. To that reader, we say ''Hold that thought''.

[3] More precisely, the modification history of the version of the file at node *P*, but that's a bit long-winded.

A Digression on Conflict Resolution

The graphs we have been using readily capture a time-ordered sequence of modifications and conflict-free synchronizations, with the result that we can determine whether a conflict occurs at a given later node. If we introduce resolved conflicts into the Parker graphs, the meaning of the graphs becomes ambiguous. For example, consider the partition graph labeled (0):





Is there a conflict at each of the bottommost nodes? The answers depend on the resolution at *BC*. If the conflict at *BC* is resolved simply by using the copy from the upper *AB*, then there should not be a conflict at the lower *AB*, since in effect the upper *CD* modification has been ignored by *BC*, as in the graph labeled (i). There should, of course, still be a conflict at the lower *CD*. If we choose the *CD* copy, the answer is reversed, as in the graph labeled (ii). If we merge the *AB* and *CD* copies at *BC*, replacing them with a new copy, then there are conflicts in both bottommost nodes, as in the graph labeled (iii). By conflating ''m'' with ''x'', Parker's graphs always assume case (iii).

This example illustrates the subtleties that must be taken into account when dealing with synchronization scenarios. Part of the problem here is that the modification history definition of the partial order breaks down when conflicts are resolved: if *BC* chooses to merge the *AB* and *CD* version somehow, we would like to arrange that the resulting version of the file has the properties that $AB \leqq BC$ and $CD \leqq BC$, but this would mean that *BC*'s modification history needs to have both *AB*'s and *CD*'s as prefixes, which is impossible since neither *AB*'s nor *CD*'s history is a prefix of the other's. Parker *et al.*'s version vector method and our communication vector method differ in how they approach this problem.

Version Vectors

As we have presented it, the crux of the filesystems synchronization problem is the detection of synchronization conflicts, which we have defined in terms of the comparison of the files' modification histories. Thus to detect conflicts, it suffices to store along with each file its entire modification history. Such a method grows unwieldy very quickly. Hopefully a more compact representation would be less costly to store and easier to manipulate. In fact, all the methods we shall present, including Parker *et al.*'s version vector approach, begin with the problem of storing entire modification histories and then apply various assumptions to reduce the necessary storage. Let us begin by considering the version vector solution.

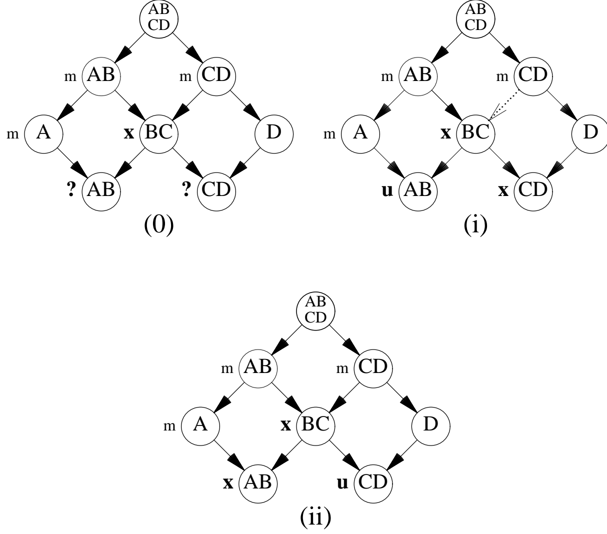
Suppose we use as our modification ids the replica name along with a per-file sequence number as the modification id, so the fifth change to a particular file by replica *A* is denoted by A^5. Note first that modification histories are never reordered: if a modification history of a file on one replica is the list $[A^1, B^1, A^2, A^3, B^2]$, another version of the same file will never be encountered that has a modification history that begins $[B^1, A^1, A^2]$. Thus, it is permissible to treat the list as a set. Now the partial ordering is given by the ''is a subset of'' relation rather than the ''is a prefix of'' relation. Next note that modification histories are ''locally backwards closed'': if A^3 is in a history, the history is sure to contain A^1 and A^2 as well. Parker *et al.*'s version vectors take advantage of these two properties to condense the history into a vector containing a single time for each replica. For instance, the example history above would be represented by the vector (A^3, B^2).

Two version vectors are compatible if one dominates the other: that is, if every entry in one vector is greater than or equal to the corresponding entry in the other vector. This vector domination relation exactly encodes our partial ordering from before. Two version vectors are incompatible, causing a conflict, if neither dominates the other.

Notice that the change to a set-based modification history solves the synchronization representation problem mentioned in the previous section: to indicate that a version at node *P* is the resolution of two conflicting versions at nodes Q_1 and Q_2 we set *P*'s modification history to the union of the histories of Q_1 and Q_2. In terms of vectors, we set *P*'s version vector to the elementwise maximum of the two conflicting

version vectors. This produces the desired effect that $Q_1 \leqq P$ and $Q_2 \leqq P$.

This solution is not completely general. In terms of the three possible resolutions described in the previous section, the union of the histories corresponds to the last solution, in which a new version is created that merges both previous versions. There is no way, using version vectors, to express both that we chose one version over the other and that we resolved the conflict. Consider the graph labeled (0):



(0) (i) (ii)

(This is the same example as above except that node D no longer modifies the file.) Note that in all cases, the modification histories of the upper AB and A and the upper CD and D are the same: that is, $AB = A$ and $CD = D$. Suppose the upper AB version is chosen over the upper CD version. If to express this we set the modification history $BC = AB$ (which, thinking strictly in terms of file versions, is clearly the case), then we correctly avoid the conflict at the lower AB, but we repeat the just-resolved conflict at the lower CD (see (i)). If, on the other hand, we choose as our history $BC = AB \cup CD$, we correctly avoid the conflict at the lower CD but now induce a false conflict at the lower AB (see (ii)).

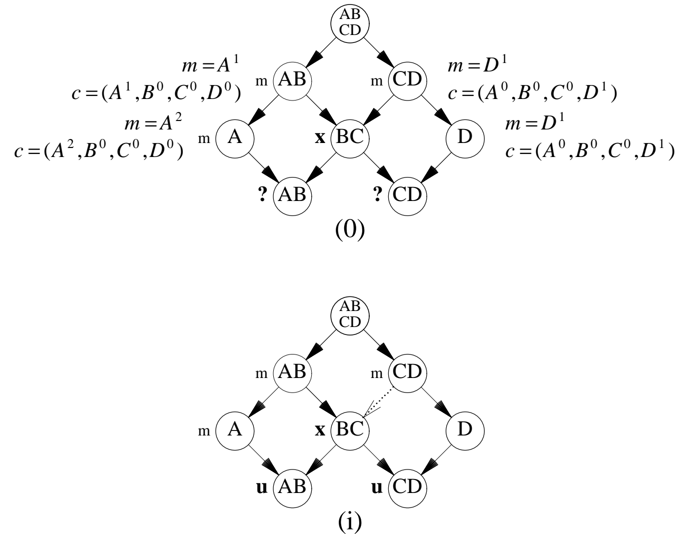
Put more succinctly, if we choose the upper AB's history as the modification history for BC, we will ''re-conflict'' at the lower CD. If we add anything to the upper AB's history to create BC's history, we will falsely conflict at the lower AB. Modification histories alone are not sufficient to keep track both of conflicts and of previous resolutions.

Communication Vectors

The first step in our proposed solution is the use of *communication vectors* to address the problems with version vectors and modification histories in general. A communication vector is like a version vector but notes the currency of the *information flow* from a given replica rather than the currency of the file modifications.

Note that in the absence of reconciliations, modification histories are ''globally backwards closed'': if the modification history of one version of a file is $[A^1, B^1, A^2, A^3, B^2]$ and another history contains B^2, that other history will also contain A^1, B^1, A^2, and A^3. As a result, to test whether one history is a prefix of another, it suffices to test whether the most recent entry in the one history exists in the other. This does not mean that we can store only the last entry of each history, since the larger history must be able to answer to containing any of its modification ids.

For the moment, consider our communication vectors as condensing the history in the same manner as version vectors. In addition to this vector, we store the id of the most recent modification to our version. To check whether a file with last modification id m_1 and communication vector c_1 is an older version of a file with last modification id m_2 and communication vector c_2, we need only check whether m_1 is contained in the history represented by the communication vector c_2. Consider the example that so troubled the version vectors:



(0) (i)

(Here, we assume that although A and B were continuously synchronized in the upper AB pair, it was A that made the modification; similarly, D made the modification in the upper CD pair.) Suppose again that we want to resolve the conflict at BC by using AB's

version as it is. We can set the communication vector to (A^1, B^0, C^0, D^1) but leave the last modification id as A^1. The addition of D^1 to the communication vector indicates that *BC* is aware of that modification, although *BC* has chosen not to apply the modification to its version of the file. It is a ''non-modification'' id in the history. Now $BC \leqq A$ since *BC*'s $m \leqq A$'s c: A^1 is in the communication history described by (A^1, B^0, C^0, D^0). So there is no conflict at the lower *AB* and the *A* version of the file wins, as it should. Similarly, $D \leqq BC$, so the *BC* version of the file wins without conflict at the lower *CD*. If we think of *D*'s modification counter as a sort of local clock, storing D^1 in all the vectors is equivalent to noting, ''I know all about this file as it existed on *D* at *D*'s time 1.'' This allows us to conclude, when we encounter at the lower *CD* a version of the file last modified by *D* at time 1, that it contains nothing but old news and can be ignored. We have successfully encoded the conflict resolutions without introducing false conflicts.

Communication vectors are attractive for another reason: they lend themselves to very high compression rates when used for a set of files. Suppose that on a replica with *F* files, we use a per-replica modification id counter rather than a per-file counter. That is, if replica *A* changes one file, then another, then the first again, the corresponding modification ids would be A^1, A^2, and A^3 where before they were A^1, A^1, and A^2. After *A* synchronizes its full set of files with *B*, the *B* element of all of *A*'s communication vector entries can safely be set to *B*'s largest modification id. The file last modified by *B* will already have that id in its vector. Changing the other vectors effectively have a sequence of non-modification ids to the histories. This does not pose problems because *B* uses a single counter for all modification ids, so we will never encounter these as real modification ids. Now the ''local time'' interpretation of the vector makes even more sense: if we think of each change to a file on *B* as a local *B* time step, updating all the *B* elements of *A*'s communication vectors records that for all the files, ''I know all about this file as it existed on *B* at *B*'s time *t*.'' This propagation is transitive: if *B*'s set of files is up-to-date with respect to machine *C* at time t_C, then after *A* synchronizes with *B*, *A*'s set of files is also up-to-date with respect to *C* at time t_C.

In a system with *R* replicas, each vector is of length $O(R)$. In a version vector method, most vectors are unique, so storing a set of *F* files requires $O(R \cdot F)$ space for the vectors. In our method, most of the time there will only be a small number of unique vectors. (In the case where the entire tree is always synchronized, there is only one unique vector.) We can store a list of vectors and use indices into this list. If there are *d* distinct vectors, the list requires $O(R \cdot d)$ space and the indices for a set of *F* files requires $O(F \log d)$ space. We also need to store the last modification id for each file, $O(F)$ space. Thus we have a $O(R \cdot d + F \cdot (1 + \log d))$ total space requirement. When $d = 1$, this reduces to $O(R + F)$.

Deletion Conflicts (Whom ya gonna call?)

If one replica deletes its version of the file, that deletion should propagate to other replicas so that eventually there is no record of the file remaining: it really is deleted. This is complicated by the possibility that one replica could delete a file while another independently modifies it. When those two replicas synchronize, it is not correct to delete the updated file, nor is it correct to recreate the deleted file: a *deletion conflict* conflict must be reported. If we consider deletion as a final modification, then the methods applicable to modification histories continue to apply.[4] In order to detect deletion conflicts, we need to keep the modification history after the file is deleted. Data left over even after the deletion of a file is commonly called a *ghost*. The central problem in handling deletion is ''ghostbusting.'' If we leave ghosts in our system even after all replicas have deleted the associated files, eventually we will unnecessarily run out of storage space. On the other hand, if we bust ghosts too early, we may not detect some deletion conflicts or may, instead of propagating a deletion, imagine a creation, resulting in files coming back to life without explanation.

Rumor, which uses version vectors, employs a distributed two-phase garbage collection for ghostbusting. Using communication vectors enables a much simpler approach to ghostbusting. When a file is deleted, we leave its communication vector as a ghost. If we need to determine whether a given file is a version of the one we deleted, we can compare its creation time to the ghost communication vector: a time older than the vector indicates that we deleted that file, while a time newer than the vector indicates a different file. Similarly, comparing the last modification id with the communication vector distinguishes between a simple delete propagation and a deletion conflict. Let us define that the communication vector associated with a directory is the elementwise minimum of the communication vectors of its children. Once the communication vector of a parent directory dominates the vector of a ghost, that ghost can be removed: if it is needed, the parent vector can be used

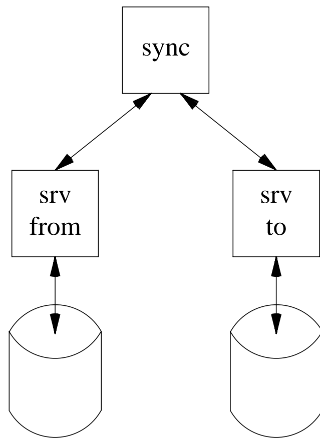
[4] Similarly, origin points can be eliminated by treating creation as an initial modification.

equivalently. The progress of ghostbusting depends only on the rest of the directory being eventually synchronized. If the whole tree is checked at each synchronization, ghosts never exist at all: since all communication vectors are the same and thus all directory vectors dominate all their children's vectors, and ghosts are busted immediately.

Implementation

We have implemented the ideas discussed here in a system called Tra. The system only synchronizes in one direction at a time, providing a convenient way to bootstrap new machines, do backups, or insulate one server from changes on another. In order accommodate the asymmetric synchronizations, the partition graph model changes somewhat: an edge need not have a common replica on its endpoints anymore. The partial order and information flow models apply without change.

The implementation of Tra is quite simple. A central synchronization program (`sync`) coordinates the synchronization between a ''from'' replica server and a ''to'' replica server (`srvs`).



The `srv` programs are charged with maintaining a *database* of communication and modification vectors, which the `sync` program queries and modifies throughout the synchronization. The database also typically contains signatures of the files, used locally by the `srv` programs to detect when a file changes. The signatures are system-dependent: Unix and Windows systems confident in the monotonicity of system time can use `mtime`, while Plan 9 systems can use the file system-provided qids. More paranoid systems might use MD5 hashes of the file contents.

Because of this architecture, the `sync` program and the two `srv` programs may all run on different systems: keeping the `srv` file system sweeps local is a large win, as is being able to run `sync` on either the ''from'' or the ''to'' system. (Indeed, `sync` could be run on a third system, but the utility of such an arrangement is questionable.) In fact, the replica names provided to `sync` are expected to be executables, usually shell scripts, that take care of establishing a connection to the desired machine and invoking `srv`. Thus the connection protocol is left unspecified, and could be `ssh`, local execution, or something else entirely.

`Sync` walks both trees simultaneously, using the synchronization and modification times to decide when files are out of date and when conflicts arise, as previously described.

Conflicts are reported by printing the names two files for the user to compare. Once the user has resolved the conflict, the resolution choice will be automatically detected at the next sweep. Signatures of the two choices are sufficient to distinguish between choosing one, choosing the other, and merging the two.

A Review of Assumptions and Implications

The implementation and debugging of Tra pinpointed a number of assumptions latent in the analysis thus far, as well as some shortcomings in the communication vector method. We present a sequence of various assumptions that can be made, examine the effects of each, and discuss whether each is reasonable in practice. Understanding these tradeoffs is helpful in understanding the hybrid method adopted for Tra's current implementation.

Assumption: Local Backwards Closure of Modification Histories. *If we have a modification from replica R at local time t for a given file, then we also have all the modifications made to that file on replica R before local time t.*

Effects: It is precisely from this assumption that the method of Parker *et al.* falls out. The modification history for a file can be compressed by storing only the last modification from each replica. In effect, Parker is storing the last modification vector time and vector domination corresponds directly to one history being the prefix of another (just storing the last modification would not be sufficient). However, Parker's vectors do not compress especially well. In particular, each file will typically have a different modification time, so one must store a vector for each file.

Practicality: This assumption is entirely reasonable: since there is only one copy of file (or directory) on a given replica, changes must ultimately be ordered by local time.

Assumption: Total Synchronization. *The entire file*

tree is synchronized each time. Partial subtree synchronizations do not happen.

Effects: Total synchronization enables the next assumption, which helps speed up synchronizations considerably.

Practicality: The total synchronization assumption is, in practice, not reasonable. It is conceivable that a user might want only to synchronize a subtree, perhaps not wanting to deal with changes made in other subtrees yet. Further, the total synchronization assumption disallows the storage of anything less than the entire replica. It becomes impossible to have, say, a laptop with a stripped-down replica that contains binaries but no source tree.

Assumption: Local Backwards Closure of *Replica* Histories. *If we have a modification from replica R at local time t for a given file, then we also have all the modifications made to any file on replica R before local time t. (Follows from total synchronization and local backwards closure for files).*

Effects: Under this assumption, vector modification times provide enough information to prune the search for modified files. Let the modification time of a directory be the ''union'' of the modification times of its children (*i.e.*, the element-wise maximum). If the modification time of a directory on replica R_1 dominates that for the directory on replica R_2, then R_1 has all of the changes from R_2. Therefore, synchronization from R_2 to R_1 for the subtree rooted at the directory is complete. It is important to note, however, that without the total synchronization assumption, this scenario will break down. As an example, suppose that we have already synchronized `/usr/bob/quux`, but have not yet synchronized `/usr/ken/quux`, and both have modifications from time $t = 42$ on the same replica. If we then do a full sync, we might think that we do not need to worry about `/usr` on the basis of having a recent copy of `/usr/bob/quux`, when in fact we have an out of date copy of `/usr/ken/quux`.

Practicality: Since total synchronization is not a reasonable assumption, local backwards closure of replica histories is also not a reasonable assumption.

Assumption: Global Backwards Closure of Individual File Histories. *For any file, at any time a total order can be imposed on the union of modification histories held by the replica. Further, each history in the system is backwards closed with respect to this order. Put another way, each history in the system has an unambiguous last modification.*

Effects: Under this assumption, we can keep just the replica and local time of the last modification to a file, rather than storing the entire modification vector (as distinct from communication vectors). This information is just as strong as the entire vector under the assumption. That is, it suffices to detect all conflicts and to encode past synchronization decisions. This is the assumption that allowed us to reduce the modification vectors to a single element in the communication vector scheme.

The assumption makes ghostbusting easier, as described earlier. Once the synchronization time of a directory dominates the deletion time of a file, that file's ghost may safely be removed. If the deletion time is needed later, the synchronization time on the directory will suffice in its stead. If applied naively, however, it does not provide support for pruning the search for out-of-date files, as Parker's did. We might try to reconstruct the per-directory modification vectors by taking the maximum of all the modification times for all the directory's descendants. Since the ghostbusting procedure destroys modification times, this maximum of the modification histories for a tree cannot be calculated accurately: it will miss times for busted ghosts. This shortcoming means that synchronizations must walk the entire tree even if there are no changes, while in Parker's method we were able to detect this condition without walking past the root.

Practicality: This assumption does not hold for directories, and one can imagine situations where it would not hold for ordinary files. In the case of directories, consider the deletion of two files *A* and *B* from a directory at the same vector time on the same host. If we wish either ordering of the deletions to be treated in the same manner, then there is no unambiguous ''last deletion''. A similar situation can arise in the case of plain files. Consider, for instance, a Unix mail spool file. The entire file is analogous to a directory and the individual messages to plain files; the same problem arises as before.

Implementation, II

For the purposes of making Tra a useful tool, we chose to assume global backwards closure of file histories but not total synchronization nor its consequent local backwards closure of replica histories.

To address the shortcoming of the global backwards closure of file histories assumption, we keep full vector modification times for each directory. While this negates some of the asymptotic storage benefits of the communication vector scheme, it lets directories hold modification times when ghosts are busted. This in turn enables the pleasant property that we can prune the search for modified files, avoiding a

walk of the entire tree.

The Rumor system, because it keeps only modification times, can only prune the search under the total synchronization assumption. We believe that Rumor does make this assumption, explaining the difficulties the Rumor team encountered trying to add support for partial synchronizations. We have examined the various design documents that come with the Rumor distribution, but they are fragmentary at best, not providing a definitive account of the methods used. We have been hesitant to examine the fairly large seven megabyte source tree.

Applications

We have used Tra to initialize a Plan 9 notebook from scratch. The machine is booted from a network file server or a CD-ROM, the file system is reamed, and then a single Tra command populates the file system from another replica. This replaces a complex installation program and the personalization that usually follows. At this point, the notebook provides an identical interface and set of files as the copied replica, and changes made on the notebook can be propagated to the original replica (or other replicas) as desired.

One issue in initializing a new system is marking files that should be copied at the beginning of time but then left alone. For example, one might want to start with a copy of the replica's `/etc/passwd` file, but then not propagate changes to it. This can be achieved by setting the communication time of the file to be the infinity vector. If the file does not exist on the target, it will be copied, but once copied, the synchronization algorithm will interpret the infinity vector as evidence that the file does not need any modifications other systems might have to offer.

The convenience of having a single home directory shared among multiple computers cannot be overstated. Spring cleaning of files on one replica propagates automatically to the others, and the set of replicas provides mutual backup for each other.

Future Additions

We have considered allowing parts of the replica tree to be synthesized by user-level ''file servers,'' allowing structured files to be presented as directory trees so that Tra can handle them without change. For example, a Unix mail spool file could be presented as a directory of messages, so that mailboxes on multiple systems could be synchronized despite the arrival of mail on one or both. Here the user-level file server allows us to counter the restrictions of the global backwards closure assumption, because the assumption applies only to files. We sidestep the assumption by presenting the file as a directory.

Such user-level file servers could also provide special semantics for ordinary files. For example, when initializing a notebook from a replica, one wants to create empty log files on the notebook rather than copying the replica's logs in full. Further, when synchronizing, one does not want log file modifications to propagate. A user-level file server that always presented the appearance of zero-length logs elegantly solves both these problems.

The Plan 9 local file system requires that after the `/adm/users` file (a combination of the Unix `passwd` and `group` files) is rewritten, the file server be notified with a control message before the newly added users or groups can be mentioned in operations such as *chown* or *chgrp*. A user-level file server could handle writes to just that file, making sure to notify the local file system of changes.

Conclusions

Version vectors, introduced by Parker *et al.* in 1983, have been the *de facto* standard for addressing file system synchronization problems. However, they bring with them a number of restrictions. The most notable are the inability to encode past conflict resolutions and the need for the total synchronization assumption in order to prune synchronizations. Version vectors in file synchronization have been compared to vector clocks, which find a wide variety of applications in distributed systems [6].

Communication vectors allow the lifting of the restrictions associated with version vectors. We believe that communication vectors are a truer analogue to vector clocks. At the least, we believe those using or considering the use of version vectors should be more aware of the various implicit assumptions and tradeoffs regarding version vectors, communication vectors, and hybrids of the two.

On a more practical note, the file system synchronization capabilities provided by Tra should not be ignored. They make the sharing of data between multiple computer systems bearable and even pleasant.

References

[1] S. Balasubramaniam and Benjamin C. Pierce. ''What is a file synchronizer?'', *Fourth Annual ACM/IEEE International Conference on* Mobile Computing and Networking (MobiCom '98) (October 1998).

[2] Richard Guy, Peter Reicher, Davd Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. ''Rumor:

mobile data access through optimistic peer-to-peer replication'', *Proceedings: ER'98 Workshop on Mobile Data Access*, 1998.

[3] James J. Kistler and M. Satyanarayanan. ''Disconnected Operation in the Coda file system'' , *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 1−25 (February 1992).

[4] L. B. Huston and P. Honeyman. ''Disconnected operation for AFS'', *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, August 1993.

[5] D. 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*, Vol. 9, No. 3 (May 1983), pp. 240−247.

[6] Reinhard Schwarz and Friedemann Mattern. ''Detecting causal relationships in distributed computations: In search of the holy grail'', Distributed Computing, Vol. 7, No. 3 (1994), pp. 149−174.