Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2005-014
MIT-LCS-TM-650

February 28, 2005

File Synchronization with Vector Time Pairs

Russ Cox and William Josephson

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

File Synchronization with Vector Time Pairs

Russ Cox, MIT CSAIL
William Josephson, Princeton CS
rsc@mit.edu, wkj@cs.princeton.edu

Abstract

Vector time pairs are a new method for tracking synchronization metadata. A vector time pair consists of two vector times: one tracking file modification history and one tracking file synchronization history. Because the vector times are maintained separately and used for different purposes, different algorithms and optimizations can be applied to each. As a result, vector time pairs impose no restriction on synchronization patterns, never falsely detect conflicts, require no space to store deletion notices, require network bandwidth proportional only to the number of files changed, and support partial synchronizations. No other current synchronization method has all these properties. Results from an implementation of vector time pairs in a new user-level file synchronizer called Tra confirm the benefits of vector time pairs.

KEYWORDS: vector time pairs, file synchronization, version vectors, logical clocks, distributed systems

1 Introduction

Anyone who uses more than one computer is aware of the data management problem posed by doing so: having multiple copies of files requires *synchronization* of files to bring all copies up to date after some copies have changed.

The simplest and perhaps most widespread method is manual synchronization, in which users remember which files they have changed on which computers and manually copy those files to the other computers. Since users must manually keep track of which files are up-to-date and which are out-of-date, this method is highly error-prone.

Most people augment manual synchronization, shifting some of the bookkeeping burden onto computers. For example, to avoid writing old files over new ones, the user could refer to the modification times on the files, or use a program such as Rsync [19] with appropriate options to automate this process.

This approach still depends on users to be careful. The user must remember to synchronize the computers whenever he switches from one to another. If he does not, he might end up with different "new" versions of the same file on different computers. For example, he makes one change to one computer's copy of a file and then, without any synchronizations, makes a different change to the other computer's copy. Now neither copy is really newer than the other; using modification times to declare one copy "newer" will lose one change or the other.

A better method is to use a file synchronizer, such as Unison [1] or Rumor [6]. File synchronizers track changes to files and propagate those changes from computer to computer. Unlike file copy programs such as `tar` or Rsync, file synchronizers also keep track of enough information about a file's history in order to determine whether it is safe to replace one version of a file with another. When it is not safe to replace either version with the other, as in the example above, a file synchronizer reports a *conflict*, to be resolved by some external method, usually by asking the user.

An ideal file synchronizer would meet all of the following goals:

1. Impose no restrictions or requirements on the synchronization patterns between computers. (For example, if there are three computers A, B, and C, any pair should be allowed to synchronize at any time, and one computer should not be necessary for the other two to synchronize.)

2. Detect all conflicts without any false positives. (False positives increase the amount of manual work required from the user and reduce the user's confidence in the synchronizer's judgments.)

3. Propagate file deletions without wasting space remembering files that once existed.

4. Identify the set of files differing between two computers using network bandwidth proportional to the size of the set.

5. Support partial synchronizations restricted to subtrees of the file system. (A user might, for example, want to synchronize his home directory frequently but only synchronize system software occasionally.)

As discussed in section 5, current file synchronization methods do not meet all of the goals. This paper introduces

a new file synchronization method, called *vector time pairs*, which does meet all of the goals.

Section 2 defines the problem of synchronization and explains the above goals in detail. Section 3 presents vector time pairs, their algorithms, and their time and space requirements. Section 4 evaluates an implementation of vector time pairs in a user-level file synchronizer called Tra, empirically demonstrating the theoretical claims. We end with a discussion of related work and conclusion (sections 5 and 6).

2 Problem Statement: Synchronization

Before discussing synchronization algorithms, we must define what a correct synchronization must accomplish. A strictly formal definition of correctness is an ongoing research topic (see, for example, Balasubramanian and Pierce [1], Ramsey and Csirmaz [16], and Pierce and Voilloun [15]), so we will make do with a less formal but still precise definition. We believe our characterization is reasonable because it is similar to the characterization given by Parker *et al.* in the original paper on version vectors [9] and because, when restricted to a pair of replicas, it is equivalent to the definition of synchronization used by Unison [1].

We consider a network of replicas storing a common file tree. Changes are propagated through the network by pairwise unidirectional synchronizations between replicas. In a unidirectional synchronization, changes from one replica are propagated to a second, but changes on the second do not propagate back to the first. Bidirectional synchronization is easily built from two unidirectional synchronizations.

We will call a unidirectional synchronization that propagates changes from A to B, leaving A unchanged, a "sync from A to B."

This section presents the task of synchronization from two points of view. We first consider the synchronization of a single file and then consider the synchronization of an entire file hierarchy.

2.1 Synchronizing Files

To decide how to synchronize individual files, file synchronizers follow the "no lost updates" rule. We present the rule and then consider a sequence of examples to build intuition for how the rule guides synchronization.

2.1.1 No Lost Updates

Correct synchronization provides a "no lost updates" guarantee. Specifically, suppose each file is represented by a history of modifications made over the course of its lifetime, beginning with its initial creation. If two replicas have different copies of a file (call the copies X and Y), it is safe to replace X with Y only if X's history is a prefix of Y's.

If X's history is a prefix of Y's history, then all of the updates present in X are also present in Y: replacing X with Y will not lose updates. In this case we will say that Y is derived from X.

If neither history is a prefix of the other, then replacing either copy with the other will lose updates. In this situation, the synchronizer reports a *conflict*, to be resolved by external means.

F_A and F_B denote the version of the file on A and B.
H_A and H_B denote their histories.

```
sync(A → B, file) ≡
    if H_A = H_B
        // The histories are identical,
        // so the files must be identical.
        do nothing
    else if H_A is a prefix of H_B
        // F_B is derived from F_A.
        // (A bidirectional sync would copy F_B to A.)
        do nothing
    else if H_B is a prefix of H_A
        // F_A is derived from F_B.
        copy F_A to B
    else
        // Neither history is a prefix of the other.
        // Neither file is derived from the other.
        report a conflict
```

Figure 1: An algorithmic specification of single-file synchronization in terms of file histories.

During a sync, the "no lost updates" rule completely determines the outcome. Figure 1 shows the resulting algorithm.

2.1.2 Synchronizing Modifications

Figure 2 shows a set of examples illustrating the "no lost updates" rule. Each picture plots the state of one file on multiple replicas over time. Different shapes denote different versions of the file. We consider two versions different if they have different modification histories. Two files with identical versions have identical histories and therefore identical contents, but two files with identical contents do not necessarily have identical versions.

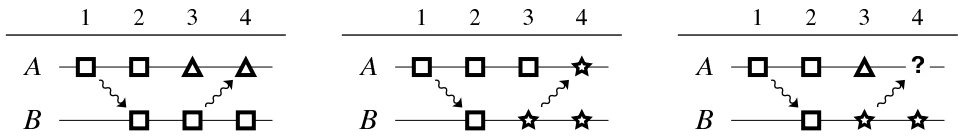
A wavy arrow between the two replicas' timelines denotes a unidirectional synchronization that might change the state of the file on the destination replica. We call a sync whose arrowhead points at time t a "sync at time t." It uses the file versions on the source and destination replicas at time $t - 1$ as inputs, deciding the version that will be stored on the destination replica at time t.

The text below uses modification histories (written like ⟨□, △⟩) as shorthand for "the file version with modification history ⟨□, △⟩."

The three scenarios begin the same way. At time 1, A creates a new file. At time 2, a sync from A to B copies the new file to B, so that the file now exists on B. (We have not yet explained why the synchronizer chose to copy the file from A to B. We will examine file creation shortly.)

At time 3, one or both replicas change the file. At time 4, a sync from B to A has varying results.

(a) A changes its copy of the file at time 3. The sync at time 4 chooses A's ⟨□, △⟩ over B's ⟨□⟩, so it does nothing



All three cases begin the same way. At time 1, *A* creates □.
At time 2, the sync from *A* to *B* copies □ to *B*.

At time 3, *A* modifies □, producing △. At time 4, the sync from *B* to *A* *does nothing*, since △ is derived from □.

(a)

At time 3, *B* modifies □, producing ☆. At time 4, the sync from *B* to *A* *copies* ☆ *to A*, since ☆ is derived from □.

(b)

At time 3, both *A* and *B* change the file. At time 4, the sync *reports a conflict*, since neither △ nor ☆ is derived from the other.

(c)

Figure 2: The possible outcomes of a sync involving two different versions of a file: do nothing, copy the file, or report a conflict.

to *A*. (A sync in the opposite direction would copy △ to *B*.)

(b) *B* changes its copy of the file at time 3. The sync at time 4 chooses *B*'s ⟨□, ☆⟩ over *A*'s ⟨□⟩, so it copies ☆ to *A*.

(c) Both *A* and *B* change their copies of the file at time 3. The sync at time 4 cannot choose between ⟨□, △⟩ and ⟨□, ☆⟩, since neither history is a prefix of the other. Instead, it reports a conflict, to be resolved by external means, either automatic or manual.

These three examples illustrate a synchronizer's choice when presented with different versions of a file: do nothing, copy the file, or report a conflict.

It is important to note that although we have shown a global clock marking time on all replicas, the replicas need not share a common clock. Synchronization decisions depend only on the past modification history, not on the exact times of synchronization. Two changes are considered *independent* if neither appears in the modification history of the other. When a synchronization discovers independent changes, it reports a conflict between the two.

2.1.3 Recording Conflict Resolutions

Once conflicts have been detected and resolved, a file synchronizer should record the resolutions so that if the same situation reoccurs (because the files involved have also propagated to other replicas), the user needn't be bothered again.

Figure 3 illustrates such a situation. The conflict identified between □ and ⬠ at time 4 in (a) might be resolved (manually or by an automated resolver) in favor of □ (as in (b)), in favor of ⬠ (as in (c)) or by creating a composite version ◇ (as in (d)). No matter what the decision, the syncs at times 5 and 6 should be no-ops: there's nothing new from *A* or from *C*. If the conflict at time 4 is resolved in favor of □, then according to the no lost updates rule, the sync at time 7 should choose △ over □. On the other hand, if the conflict at time 4 is resolved in favor of ⬠ or ◇, the sync at time 7 should report a conflict: the history of △ is incompatible with both the history of ⬠ and the history of ◇.

We will see later that existing synchronization techniques have no way to recording conflict resolutions, while vector time pairs do.

2.1.4 Synchronizing Creations and Deletions

A file synchronizer must also be able to synchronize both newly created and deleted files. Figure 4 shows a third set of examples, illustrating possible outcomes when the file exists on only one of the two replicas. Gray shapes denote deleted files.

The scenarios begin as in Figure 2. At time 1, *A* creates a new file. At time 2, a sync from *A* to *B* copies the new file to *B*, so that the file now exists on *B*. At time 3, one of the replicas deletes the file, and the other may change it. At time 4, a sync from *B* to *A* has varying results.

(a) *A* deletes its copy while *B* does nothing. The second sync chooses *A*'s ⟨□, □⟩ over *B*'s ⟨□⟩, so it does nothing to *A*. (A sync in the opposite direction would delete □ from *B*.)

(b) *B* deletes its copy while *A* does nothing. The second sync chooses *B*'s ⟨□, □⟩ over *A*'s ⟨□⟩, so it deletes □ from *A*.

(c) *A* deletes the file while *B* changes the file. The sync cannot choose between ⟨□, □⟩ and ⟨□, ☆⟩, since neither history is a prefix of the other. Instead, it reports a conflict.

(d) *A* deletes the file while *B* changes the file. The sync cannot choose between ⟨□, ☆⟩ and ⟨□, □⟩, since neither history is a prefix of the other. Instead, it reports a conflict.

When a file exists on one replica but not on another, the sync has one more choice: it can create the file on the replica

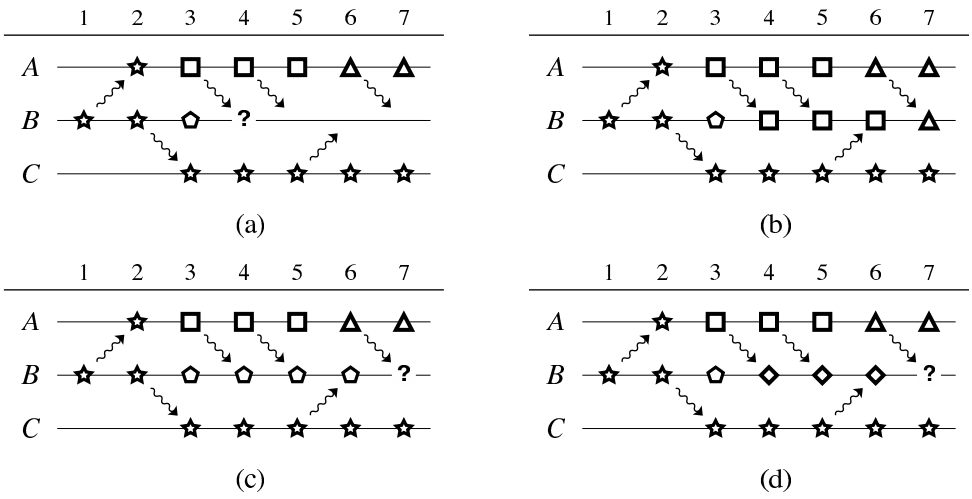


Figure 3: Conflict resolutions should "stick" in a synchronizer.

After the sync in (a) at time 4, the sync at time 5 (involving □) should be a no-op — it should not make any changes and it should not report any conflicts. The sync at time 6 (involving ☆) is sending ancient data, so it too should be a no-op. Whether there is a conflict in the sync at time 7 (involving △) depends on how the conflict at time 4 was resolved. The three possibilities are shown in (b), (c), and (d).

that doesn't have it. The sync from *A* to *B* at time 2 makes this choice in all the examples: the sync chooses *A*'s ⟨□⟩ over *B*'s ⟨⟩, creating □ on *B*.

Note the parallels between Figure 2 and Figure 4. In these examples, deletion is treated as just another simple change in the file's modification history. However, always treating deletions like changes can result in unnecessary conflicts, as depicted in Figure 5. The outcome of the deletion scenario (a) is the same as the outcome of the modification analogue (b), but in cases such as (c) and (e), it makes sense to treat deletions as different from changes.

The intuition behind Figure 5 is that *a deleted file should not conflict with an independently created file*. Even though the two operations happened independently, an order can be imposed that does not lose updates. The life of the deleted file version is treated as happening *before* the other version was created.

According to this observation, the outcome of (c) is to choose the new file, while the outcome of the modification analogue (d) is a conflict. Similarly, the outcome of (e) is a deletion notice representing both deleted files, while if we treated the deletions in (e) as simple changes, we would report a conflict as in (f). Reporting a conflict between two deleted files is unnecessary — no matter what, the result is going to be a deleted file.

When a file is deleted, we've assumed that the metadata remains available for use in future synchronizations. In a system where many files are short-lived, the cost of storing metadata for deleted files might be significant. We will examine the storage costs in detail later; using vector time pairs, it is possible to avoid storing any per-deleted-file metadata but still be able to make correct synchronization decisions.

2.2 Synchronizing File Trees

Although synchronizing a single file is certainly subtle, it is not the whole picture. File synchronizers operate on file trees, not just individual files. We want synchronization to require time and effort not worse than manual copying of files. This has two important implications. First, the amount of network bandwidth consumed should be proportional to the amount of changed data, not the entire file tree. Second, the synchronizer should support synchronizations of subtrees and individual files.

2.2.1 Network Bandwidth

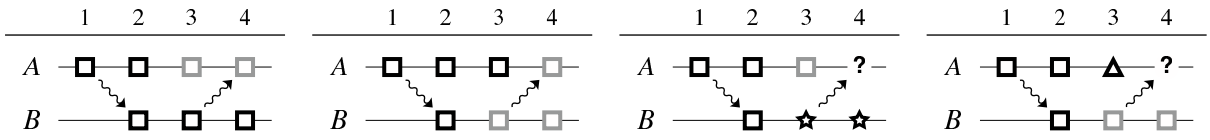
One approach to synchronizing a file tree is to run the per-file synchronization described above for every directory and file in the tree. To do this, metadata must be exchanged for each file in the tree, a waste of bandwidth for the files that have not changed.

In manual synchronization, users select the files to copy, spending network bandwidth only on the files that have changed. An ideal file synchronizer would match this bandwidth goal.

We will see later that syncs using vector time pairs need only exchange metadata for directories and files that have changed, achieving the goal.

2.2.2 Partial Synchronizations

A user may wish to sync subtrees or single files. For example, maybe the user is not currently interested in certain subtrees. Or maybe the user only wants to copy a single changed file. An ideal file synchronizer would support such cases. In



All four cases begin the same way. At time 1, *A* creates □. At time 2, the sync from *A* to *B* copies □ to *B*.

(a) At time 3, *A* deletes □, producing □. At time 4, the sync from *B* to *A* *does nothing*, since *A*'s □ is derived from *B*'s □.

(b) At time 3, *B* deletes □, producing □. At time 4, the sync from *B* to *A* *deletes* □ *from A*, since *B*'s □ is derived from *A*'s □.

(c) At time 3, *A* deletes □ while *B* modifies it. At time 4, the sync *reports a conflict*, since neither version is derived from the other.

(d) At time 3, *A* modifies the file, while *B* deletes it. At time 4, the sync *reports a conflict*, since neither version is derived from the other.

Figure 4: Three possible outcomes of a sync in which the file exists only on one computer: do nothing, delete the file, or report a conflict. Note the similarity to Figure 2. As far as synchronization is concerned, deletion is just another kind of change. The fourth possible outcome, create the file, is illustrated by the sync at time 2 in all the examples.

the case of an explicit list of files, it could be used in the same way that `rcp` or `scp` is used today.

This goal interacts in unexpected ways with the goal of using a minimal amount of network bandwidth. Some current synchronizers use heuristics to reduce the network bandwidth costs, but the heuristics fail to accommodate partial synchronizations.

2.3 Goals Revisited

This section has expanded on the goals for an ideal file synchronizer set forth in the introduction.

1. *Impose no restrictions or requirements on the synchronization patterns between computers.*
   We have implicitly assumed that the communication pattern between replicas is unrestricted. As discussed in section 5, some systems must restrict the communication pattern in order to achieve the next goal.

2. *Detect all conflicts without any false positives.*
   Detecting conflicts is useful only if there are few false positives. False positives decrease the user's confidence in the reported conflicts, making him more likely to ignore a real conflict and lose data. An ideal synchronizer must record conflict resolutions in order to avoid false positives.

3. *Propagate file deletions without wasting space remembering files that once existed.*
   A synchronizer must be able to decide whether a file version on one replica has been deleted from or never existed on another replica. For some usage patterns, keeping information about every file that has ever existed on a replica is impractical. An ideal synchronizer must be able to make these decisions but also limit the metadata storage devoted to deleted files.

4. *Identify the set of files differing between two computers using network bandwidth proportional to the size of the set.*
   A naive synchronizer could run a synchronization for every file on the two computers, but this would require bandwidth proportional to the size of the entire file system. An ideal synchronizer would quickly determine the set of files that differ between the two computers, using network bandwidth proportional to the size of that set rather than the entire file system.

5. *Support partial synchronizations restricted to subtrees of the file system.*
   A naive synchronizer can easily do this: it would run a synchronization for every file in the specified subtree. As synchronizers work to achieve the previous goal, they sometimes lose the ability to support this one.

This is not an exhaustive list of good synchronizer features, but it does cover the important basics. Other desirable features might include the ability to support read-only replicas or partial replicas.

3 Vector Time Pair Algorithm

Existing synchronizers cannot deliver all of the goals we have laid out. We present a new synchronization method, *vector time pairs*, which can. In this section we introduce vector time pairs as a refinement of the current de facto synchronization method, version vectors, and then examine the synchronization algorithms using them.

3.1 Version Vectors

The main problem a file synchronizer must address is how to keep enough information about file modification histories in order to compare two histories and apply the "no lost updates" decision rule.

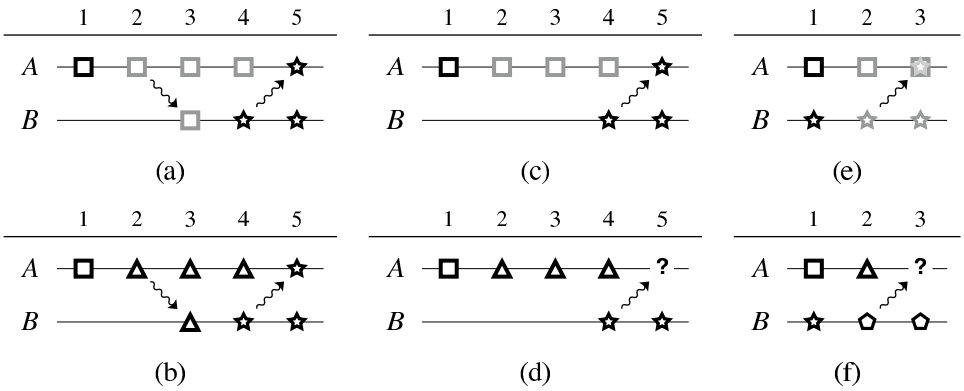


Figure 5: Corner cases when it makes sense to treat deletion differently from changes. Usually deletions (as in (a)) can be treated as changes (as in (b)). When the two file versions were created independently and one or both of them is deleted, the synchronization can be carried out without reporting a conflict and without losing updates. Memory of the deleted versions lives on in the local replica's per-file metadata. The result is successful synchronization in cases (such as (c) or (e)) where treating deletion as changes would have reported conflicts (such as (d) and (f)). The wisdom of the new rule is most clear in case (e). Treating deletions as changes would report a conflict as in (f), but there is little sense to reporting a conflict between two deleted files: for most purposes, one deleted file is the same as another. Instead, the synchronization remembers both versions in the replica's metadata for that file.

Figure 6: Naming of file versions using the place and time they they were created. (a) shows the scenario in the usual notation; (b) shows the naming. Since any version at time t on a given computer necessarily includes knowledge of versions at previous times on that same computer, the modification history $\langle A_1, B_3, A_5 \rangle$ can be summarized by the version vector $\{A \mapsto 5, B \mapsto 3\}$.

F_A and F_B denote the version of the file on A and B.
$\mathbf{m}_A$ and $\mathbf{m}_B$ denote their version vectors (vector modification times).

```
sync(A → B, file) ≡
    if mA ≤ mB
        // FB is the same as or derived from FA.
        do nothing
    else if mB ≤ mA
        // FA is derived from FB.
        copy FA to B
    else
        // Neither file is derived from the other.
        report a conflict
```

Figure 7: Single-file synchronization using version vectors.

Version vectors [9] are one solution to this problem. Suppose we name a version by when and where it was created. For example, in Figure 6(a), ▲ has version A_5, and the modification history of ▲ is $\langle A_1, B_3, A_5 \rangle$. We can simplify the history by noting that if A_n is present in the history, then all distinct versions of the form A_m for $m < n$ must also be present in the history. In this example, the presence of A_5 in the history implies the presence of A_1. (It would also imply the presence of A_2, A_3, or A_4 if a new version had been created at one of those times.) We can then summarize the history as a vector with one entry for each replica; the entry contains the local time (local event counter) of the last modification made on that replica. Rather than define that (say) A's version is the first entry in the vector and B's version is the second, it is convenient to write version vectors as maps from replicas to their entries. For example, $\langle A_1, B_3, A_5 \rangle$ is summarized as $\{A \mapsto 5, B \mapsto 3\}$. To compare histories represented by version vectors $\mathbf{u}$ and $\mathbf{v}$:

1. If the vectors are identical, so are the histories.
2. If all elements in $\mathbf{u}$ are less than or equal to the corre-

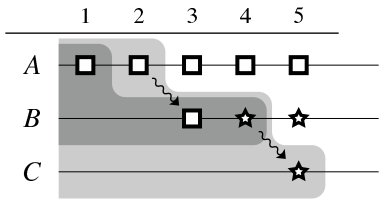


Figure 8: Graphical depiction of the vector modification time (dark gray) and the vector synchronization time (dark and light gray) of the file version on replica C at time 5. The modification time (dark gray area) is the minimal vector time containing C_5's modification history. The synchronization time (dark and light gray area combined) is the set of times and places in the system such that if a modification had occurred then and there, that modification would be present in C_5's modification history. The modification time is always contained in the synchronization time. Their difference (the light gray area) is known to be modification-free: the file was stored at those times on those replicas on its way to C_5, but it was passed along without modification.

F_A and F_B denote the version of the file on A and B.
$\mathbf{m}_A$ and $\mathbf{m}_B$ denote their vector modification times.
$\mathbf{s}_A$ and $\mathbf{s}_B$ denote their vector synchronization times.

```
sync(A → B, file) ≡
    if mA ≤ sB
        // FB is the same as or derived from FA.
        do nothing
    else if mB ≤ sA
        // FA is derived from FB.
        copy FA to B
    else
        // Neither file is derived from the other.
        report a conflict
```

Figure 9: Single-file synchronization using vector time pairs.

sponding elements in **v**, then the history represented by **u** is a prefix of the history represented by **v**.

3. If all elements in **v** are less than or equal to the corresponding elements in **u**, then the history represented by **v** is a prefix of the history represented by **u**.

4. Otherwise, neither history is a prefix of the other.

Inserting the version vector comparisons into the prototype sync algorithm (Figure 1) yields the algorithm in Figure 7.

A version vector is a vector time $[3, 11, 12, 18]$ tracking the file's modification history, a vector modification time.

3.2 Vector Time Pairs

A *vector time pair* augments the vector modification time with a vector synchronization time. Just as the vector modification time summarizes the modification events in the life of a file, the vector synchronization time summarizes the synchronization events in the life of a file. Informally, the modification time tracks "which version we have," while the synchronization time tracks "how much we know" about the file. For example, in the scenario depicted in Figure 8, the version stored on replica C at time 5 has modification time $\{A \mapsto 1, B \mapsto 4\}$ and synchronization time $\{A \mapsto 2, B \mapsto 4, C \mapsto 5\}$. From the two times, we can deduce that there was no change to the file at B_3, or anywhere else in the light gray area in the figure.

Because we know that nothing happened to a file between its modification time and the synchronization time, $\mathbf{m}_A \leq \mathbf{m}_B$ if and only if $\mathbf{m}_A \leq \mathbf{s}_B$. (One direction follows from the fact that $\mathbf{m}_B \leq \mathbf{s}_B$. The other direction follows from the fact that all modification events in $\mathbf{s}_B$ are contained in $\mathbf{m}_B$, as discussed in the previous section.) Because the two comparisons are equivalent, we can replace $\mathbf{m}_A \leq \mathbf{m}_B$ in the version vector algorithm with the $\mathbf{m}_A \leq \mathbf{s}_B$, yielding the algorithm shown in Figure 9. Intuitively, the comparison $\mathbf{m}_A \leq \mathbf{m}_B$ asks "does B's version have all the changes present in A's version?"; the comparison $\mathbf{m}_A \leq \mathbf{s}_B$ asks "is B aware of all the changes present in A's version?" or more simply "is B up-to-date with respect to A?".

Comparing the synchronization time from one replica against the modification time from a second checks whether the first replica knows about the modification events present on the second. This reasoning process — comparing what one replica would know against what another replica has — is the important operation enabled by keeping both vectors. This operation is sufficient for determining synchronization outcomes and is the reason that vector time pairs can achieve the synchronization goals that version vectors alone cannot.

We will see later in this section that keeping the extra vector time enables storage optimizations that make the metadata storage cost for vector time pairs *less* than the cost for version vectors.

3.3 Single File Synchronization

We presented the basic single file synchronization algorithm for vector time pairs above. We must still consider how the algorithm records conflict resolutions and handles deletions and file creations.

3.3.1 Recording Conflict Resolutions

Figure 3 showed three cases in which the choice of conflict resolution determines the result of future synchronizations. Vector time pairs make it easy to get each of the results:

- To choose □, use □'s modification time $\{A \mapsto 3\}$ ($\{A \mapsto 3, B \mapsto 1\}$, compressed).
- To choose ⬠, use ⬠'s modification time $\{B \mapsto 3\}$.
- To create a new version ◇ derived from both, set the modification time to $\{B \mapsto 4\}$ ($\{A \mapsto 1, B \mapsto 4\}$, compressed).

In all three cases, set the sync time of the new version to be the element-wise maximum of the sync times of the versions involved in the sync: $\{A \mapsto 3, B \mapsto 4\}$. The syncs at time 5 and

F_A denotes the version of the file on A.
The corresponding file F_B does not exist on B.
$\mathbf{m}_A$ denotes the vector modification time of F_A.
c_A denotes the (scalar) creation time of F_A.
$\mathbf{s}_A$ and $\mathbf{s}_B$ denote the vector synchronization times of F_A and F_B.

```
sync(A → B, file) ≡
    if mA ≤ sB
        // The deleted FB was derived from FA.
        // (sync(B → A, file) would delete FA.)
        do nothing
    else if cA ≰ sB
        // FB and FA were created independently.
        // (sync(B → A, file) would do nothing.)
        copy FA to B
    else
        // FB and FA were derived from the same
        // initial file, but they diverged.
        report a conflict
```

Figure 10: Synchronization of deleted files using vector time pairs. The synchronization does not report conflicts between independently created files when at least one of them has already been deleted.

time 6 will be no-ops because both □ and ☆ have modification times ($\{A\mapsto 3\}$ and $\{B\mapsto 1\}$) less than this sync time. The sync at time 7 will proceed without conflict only if the sync at time 4 chooses □: □'s modification time ($\{A\mapsto 3\}$) is less than or equal to the sync time of △ at A_6 ($\{A\mapsto 6, B\mapsto 1\}$), but ⬠'s and ◇'s modification times ($\{B\mapsto 3\}$ and $\{A\mapsto 3, B\mapsto 4\}$) are not.

Setting the sync time this way records that replica B is aware of the updates present in the conflicting versions, even if the conflict resolution has explicitly chosen to discard some of them (as in (a) and (b)). A single version vector per file is not enough state to provide this property.

3.3.2 Synchronizing Deletions

Recall from section 2.1.4 that deletions are treated as similar but not identical to modifications. It is easy to change the single-file sync algorithm to accommodate the differences. We track each existing file's creation time in addition to its vector time pair. (The creation time is the first element in the file's modification history.) Figure 10 shows the details. Having the creation time c_A lets the algorithm decide whether B has ever had a file related to F_A. If not, the two are independent, and following the new rule, the situation is treated like F_A being derived from B's deleted version.

As an added benefit, the only metadata about the deleted file that the new algorithm uses is its synchronization time. When the synchronizer notices that a file has been deleted on the local replica, it creates a deletion notice with the file's current synchronization time, but *without* the file's modification or creation times (these are only used for existing files).

After discussing synchronization times for directories and optimizations for storing synchronization times below, we will see that storing these deletion notices has no cost at all — the deletion notice is effectively absorbed into the parent directory's metadata.

The sync is considering a single directory that exists on A and B.
$\mathbf{m}_A$ and $\mathbf{m}_B$ denote their vector modification times.
$\mathbf{s}_A$ and $\mathbf{s}_B$ denote their vector synchronization times.

```
sync(A → B, dir) ≡
    if mA ≤ sB
        // B already has all the changes present in A's tree.
        do nothing
    else
        // Recurse into the tree.
        for each child in dir
            sync(A → B, child)
```

Figure 11: Directory synchronization using vector time pairs.

3.4 Synchronizing File Trees

We noted earlier that a synchronizer could handle file trees by running the single-file algorithm for every file and directory in the tree. Vector time pairs enable important optimizations over this naive implementation.

3.4.1 Network Bandwidth

As discussed in section 2.2.1, we would like synchronizations to require network bandwidth proportional to the set of changed files rather than the entire file system. We can do this with vector time pairs by assigning a synchronization and modification time to directories as well as files. The vector synchronization time of a directory is the element-wise *minimum* of the synchronization times of its children. The modification time of a directory is the element-wise *maximum* of the modification times of its children. Figure 11 shows the sync algorithm for directories. When a directory's modification time on A is less than or equal to its synchronization time on B, all the changes present in the tree on A must also be present in the tree on B, so the tree can be skipped.

3.4.2 Partial Synchronizations

Vector time pairs can accommodate partial synchronizations by simply running the synchronization algorithm on the roots of the subtrees to be synchronized rather than on the main file system root. Figure 12 demonstrates why this approach can coexist with the network bandwidth savings discussed above. A partially synchronized directory has a modification time that, for some elements, is greater than the synchronization time. This reflects the fact that while there are some modifications from particular times present in the subtree, modifications made at the same time to other files in the subtree may not be present.

3.5 Metadata Storage Costs

We now consider the storage cost associated with vector time pairs. It is an important implementation detail, and the

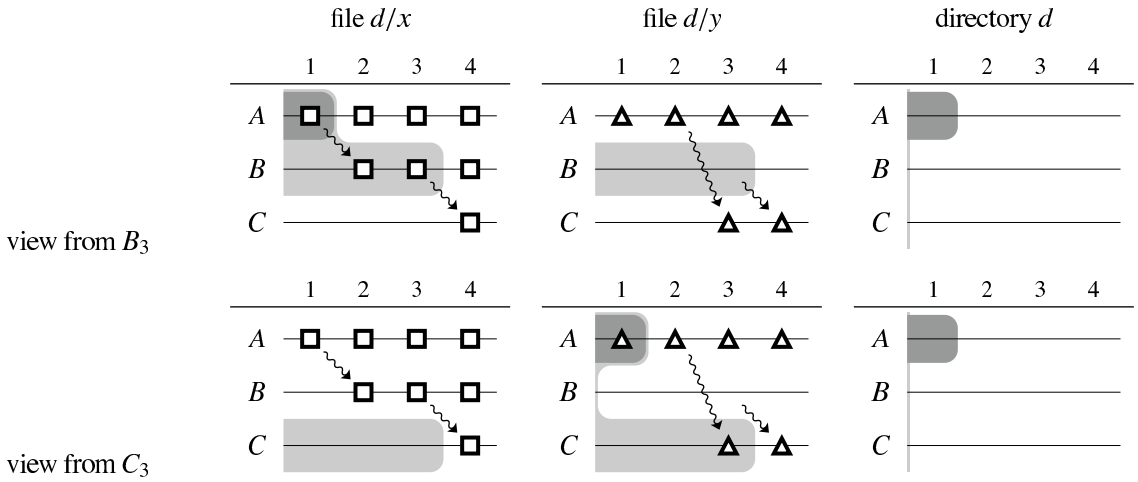


Figure 12: The effect of partial synchronizations on the vector time pair for a directory. A creates two different files x and y in the directory d at time A_1. A partial sync copies x to replica B, and another partial sync copies y to replica C. The vector synchronization time for d is defined as the elementwise minimum of the vector synchronization times of d's children; in this case, {}. The vector modification time for d is defined as the elementwise maximum of the vector modification times of d's children; in this case, $\{A \mapsto 1\}$. The fact that the modification time for d is *not* less than the synchronization time for d indicates that d is only partially synchronized. In contrast, version vectors cannot express this situation, so they cannot be used to determine quickly which subtrees are up-to-date and need not be examined further.

optimizations here directly affect the storage required for metadata about deleted files.

In a system with D directories and F files shared across R replicas, the base cost of storing vector time pairs for a single replica is $O(R \cdot (D+F))$. Two optimizations effectively eliminate the need to store vectors for the files in system, bringing the cost down to $O(RD+F)$, where D is the number of directories in the tree. One optimization applies specifically to synchronization times; the other applies specifically to modification times.

We assume that the underlying vector representation omits zero entries, as we have done in this paper. That is, we assume that it is cheaper to store $\{A \mapsto 3\}$ (implicitly $\{A \mapsto 3, B \mapsto 0\}$) than to store $\{A \mapsto 3, B \mapsto 1\}$.

3.5.1 Encoding Synchronization Times

The first optimization comes from the observation that, by definition, vector synchronization times are monotonically increasing along each path in the file system. For a given file or directory, we need to store only the vector difference between the file's or directory's vector synchronization time and its parent directory's vector synchronization time.

For most synchronization patterns, these differences will be zero vectors. To see why, suppose that syncs always successfully synchronize the entire file system. The synchronization histories (and thus the synchronization times) for all files and directories in the system will be identical. Now suppose that the synchronization times have diverged, and there are many different synchronization times on a particular replica. Once that replica has synchronized its entire file system with each of the other replicas, directly or indirectly, all files and directories on the replica will be equally up-to-date and thus have identical synchronization times. Regular partial synchronizations of entire subtrees will have the same unifying effect on those subtrees, so even if full synchronizations are rare, partial synchronizations might focus on large trees like `/usr/local`, `/home/you`, or `C:\My Documents`, having a similar effect: there will only be a small number of unique synchronization times on the replica, one for each separately synchronized subtree. Thus the differences are zero vectors, requiring almost no storage.

This same argument explains why deletion notices (described earlier) require no storage. After the synchronizer has scanned a directory containing a recently deleted file, the directory and the deleted file will have the same synchronization time. Since the synchronization time is the only metadata associated with a deletion notice, the deletion notice (including the deleted file's name!) can be thrown away. If information about an unknown file is ever needed, the deletion notice can be reconstructed using the parent directory's current synchronization time.

If there are S unique synchronization times throughout the tree, the optimization reduces the total storage cost of the synchronization times from $O(R \cdot (D+F))$ to $O(RS+(D+F))$. (The extra $D+F$ is a constant amount of space to store the zero for each file.)

3.5.2 Encoding Modification Times

The second optimization comes from the observation that modification times can often be reduced to scalars without changing the result of comparisons.

In the synchronization algorithms (see Figures 9 and 11), vector modification times only appear in expressions of the form $\mathbf{m} \leq \mathbf{s}$. The last element in the file's modification history sequence determines the result of the comparison. For example, if the change A_5 is made to a file version with his-

	Time (s)				
	`copy`	`nop`	`change1`	`change*`	`remove`
100 Mb/s					
Tra	88.20	2.59	2.32	70.20	9.61
Rsync	34.73	2.45	2.34	41.98	0.81
Unison	67.86	2.05	2.67	140.60	9.61
1000 Mb/s					
Tra	61.14	1.69	4.67	49.86	6.45
Rsync	28.65	1.81	1.97	38.50	0.94
Unison	41.47	1.82	1.52	108.92	8.88

Figure 13: Raw performance comparison between Tra, Rsync, and Unison. Tra performs competitively with Rsync and Unison, except for `copy`, where Tra does not pipeline enough RPCs.

tory $\langle A_1, B_3 \rangle$, there will never be a replica that knows about A_5 but not about A_1 and B_3. That is, a synchronization history for any version of this file either will not include A_5 or will include all of A_1, B_3, and A_5. Therefore, using a modification time of $\{A \mapsto 5\}$ will have the same effect in future synchronizations as using the true modification time $\{A \mapsto 5, B \mapsto 3\}$.

This optimization does not apply to directories. Because a directory's modification time is the element-wise maximum of the modification times of its children, we cannot identify a "last change," the presence of which implies the presence of all the other changes. Put another way, the optimization depends on the fact that changes in the history of a file are necessarily sequenced while changes to a directory are not.

The optimization therefore reduces the total storage cost of the modification times from $O(R(D+F))$ to $O(RD+F)$.

3.5.3 Storage Summary

With these two simple optimizations, the storage cost of vector time pairs reduces from $O(R(D+F))$ to $O(RD+RS+F)$, where S is the number of different subtrees that have only been partially synchronized. Since we expect S to be (much) smaller than the total number of directories, this is just $O(RD+F)$. In contrast, version vectors require $O(R \cdot F)$ space unless global coordination algorithms are used.

In most situations, the number of files will be much larger than the number of directories. We inspected some arbitrarily chosen large file trees — a Linux kernel tree, a TeX distribution, our own home directories, and two Windows installations — and found that the ratio of files to directories ranged between 10x and 30x.

4 Tra: Experience with Vector Time Pairs

We have implemented vector time pairs in a file synchronizer called Tra. In this section we describe the implementation of Tra and then evaluate the implementation against other file synchronizers and theoretical predictions from the analysis in the previous sections.

4.1 Implementation

A complete description of Tra or any file synchronizer is outside the scope of this paper [8]. There are many important details that must be considered carefully in order to ensure consistency between the vector time pair database and the local file system. There are also interesting user interface questions, such as how to present and resolve conflicts, how to specify partial synchronizations, and how to report the synchronization status. In this section we describe only enough about Tra to understand the performance results in the next section.

Tra is implemented as three programs communicating via RPC. The main program, `tra`, coordinates the sync. Using `ssh`, it starts a `trasrv` slave programs on the two replicas involved in the sync and then uses RPCs to inspect the two replicas and run sync operations. In order to have many RPCs in flight and thus use the network well, `tra` is structured as a large number of worker threads each directing the synchronization of a single file or directory.

Each `trasrv` scans the local file system for changes at startup, using system-dependent file generation numbers and modification times to decide when a file has changed. (For systems on which this is unreliable, `trasrv` can be configured to compare the cryptographic checksums of each file with its last known checksum in order to detect changes.)

The vector time pairs for local files are stored in a custom hierarchical database. The database uses write-ahead logging to ensure consistency even in the face of crashes. The database access code compresses vector time pairs (as described in section 3.5) and removes deletion notices (as described in section 3.3.2) when storing them, but provides uncompressed vector time pairs to and recreates deletion notices for the rest of the system. Because these operations require only local knowledge, the main code and the protocol are not complicated by the details of vector time pair compression or deletion notice reclamation.

When copying files, Tra uses an algorithm similar to Rsync's to avoid needlessly transferring common file segments. Tra runs the algorithm over RPCs. Rsync is a bulk copy program rather than a file synchronizer, so it can avoid using RPCs, instead treating the network connection as two unidirectional bulk data transfers [19].

4.2 Evaluation

We evaluate Tra in two ways. First, we compare its raw speed against that of Rsync and Unison, as a sanity check that the implementation is reasonably efficient. Because Rsync doesn't suffer from RPC round-trip times and also doesn't have to maintain any metadata about the files, Rsync's performance represents an unachievable lower bound for Tra and Unison.

Second, we run Tra on workloads designed to demonstrate the asymptotic behaviors promised in section 3.[1]

[1] Ideally, we would like to run Rumor on the same workloads, but Rumor has not been maintained for a number of years and is written in a pre-standard dialect of C++. Even after we fixed the compilation errors due to changes in the C++ language, Rumor failed to run except on trivial test cases, presumably due to changes in the semantics of the various C++ libraries.

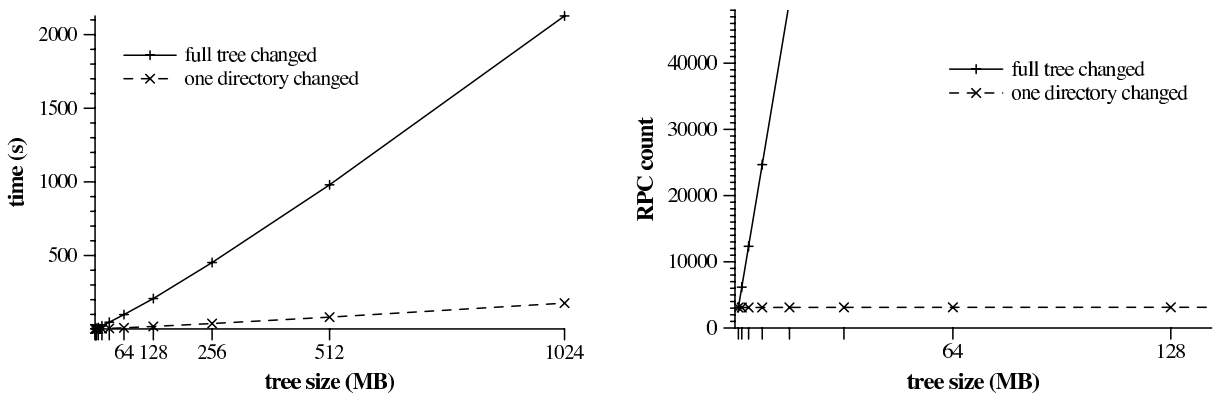


Figure 14: Elapsed time and RPC counts for syncing an N-megabyte file tree in full and also for syncing the tree when only 256 files in one directory have changed. The times for both cases are linear due to the time required to scan the file system for changes, but the RPC counts for the single changed directory case grow with the depth of the directory, $\log N$ in this case.

4.2.1 Comparison: Rsync, Unison, and Tra

To check the efficiency of the implementation, we compare Tra against Rsync and Unison for a few demonstrative workloads:

`copy` Copy a large newly created tree (Linux 2.6.5 kernel source) from one replica to another. The tree comprises 932 directories and 15,193 text files, a total of 217 megabytes of data.

`nop` After executing `copy`, sync again. Since the two replicas are identical, the sync is a no-op.

`change1` After executing `copy`, change a single file on one replica and then sync, causing the change to be propagated.

`change*` After executing `copy`, append a newline to to every file in the source tree and then sync, causing the changes to be propagated.

`remove` After executing `copy`, remove the tree from one replica and then sync, causing removal on the other replica.

We run each workload on two different machine pairs, one connected by 100Mb/s ethernet and one connected by 1Gb/s ethernet. We report the elapsed time for each workload.

All three programs are configured to use `ssh` to communicate between replicas. We patched `ssh` to set the `TCP_NODELAY` option to disable Nagle's algorithm (by default, `ssh` does this only for sessions using pseudo-terminals). Figure 13 shows the results.

In contrast to Tra and Unison, Rsync uses a natural pipelining by avoiding the use of RPCs, resulting in higher throughput when large data transfers are involved as in `copy` and `change*`. Tra and Unison take longer to run `remove` because they must update all their database entries. Tra takes longer to run the `change1` test because walking down the tree to a leaf node requires syncing all the siblings of the path as well. The Linux tree is short and fat, not such a good case for Tra.

4.2.2 Asymptotic behavior of Tra

To check that the asymptotic behaviors of the implementation match the behaviors promised in section 3, we run Tra on a series of workloads designed to demonstrate these behaviors. We use the same machine and network setup as in the previous section. Since we will be varying the size of the file tree, we use an automatically-generated file set. Specifically, when we want an N-megabyte file tree we use a balanced binary tree of height $\lceil \log_2 N \rceil$ with N leaf directories, each containing 256 four-kilobyte files with random binary contents.

Time

Section 2.2.1 argued that syncs using vector time pairs only need to use network bandwidth to discuss directories along the path to changed files. To test this, we measure the elapsed time and number of RPCs required to copy an N-megabyte file tree from one replica to another, as well as the elapsed time and number of RPCs required to synchronize the same tree after changing all 256 files in one leaf directory. The tests were run between two computers connected via 1Gb/s ethernet.

Figure 14 shows the results. The time required to synchronize the tree when all files need copying is linear in the size of the tree, as is the RPC count.

The time required to synchronize the tree when only one directory's files need copying is still linear in the size of the tree, because the tree must still be scanned in full by the local `trasrv` to determine which files have changed since the last scan. The RPC count grows with the depth of the changed leaf directory, $\log N$. The sync requires 6 RPCs to handle each directory along the path to the changed leaf directory. (The other approximately 3000 RPCs copy the megabyte of data.) This matches the analysis in section 2.2.1 — the sync does not recurse into unchanged subtrees.

The overall performance of a sync could be made sublinear by concentrating on the local `trasrv` scan; for example, file system support for quickly determining files and directories changed since a given local time would suffice. Even

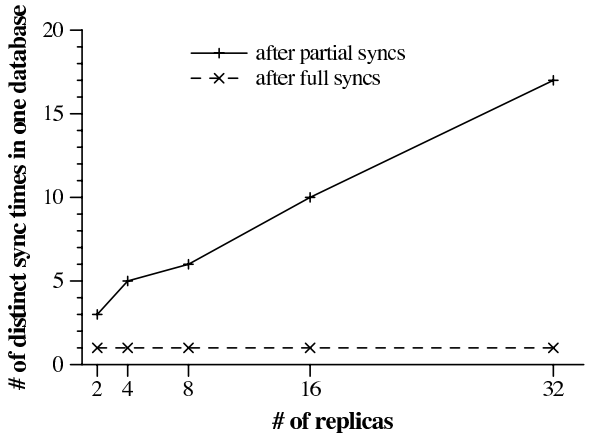


Figure 15: The number of distinct sync times in a file tree after partial synchronization of N randomly chosen leaf directories among N different replicas. Each chosen leaf directory ends up with its own sync time, and the unchosen part of the tree keeps its initial sync time, for a maximum possible $N + 1$ distinct sync times. The directories are chosen randomly with replacement, so there may be fewer than N unique chosen directories, resulting in a smaller sync time count.

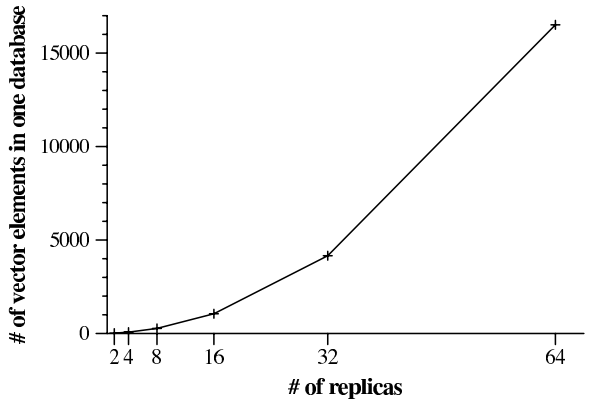


Figure 16: The number of vector elements in a database after N full synchronizations of a file tree with N leaf directories containing N files each. Each replica modifies every file in the tree before syncing with the next replica. For this case, vector time pairs use $O(RD + F) = O(N^2)$ storage while version vectors would use $O(RD + RF) = O(N^3)$ storage.

without file system support, when synchronizing over a slow network, network communication would be the bottleneck, so the linearity of the local file system scan would not be noticed.

Space

Section 3.5 argued that the number of distinct sync times in a file tree depended only on the synchronization pattern, and that full synchronizations would bring the sync times completely in line. To test this, we replicated a 32-megabyte file tree to each of N replicas in turn (sync from replica 1 to 2, from 2 to 3, and so on). Then a second round synced a randomly chosen leaf directory from 1 to 2, synced that same leaf directory and another (possibly the same) randomly chosen leaf directory from 2 to 3, synced those two leaf directories and another from 3 to 4, and so on, ending with N leaf directory syncs from N to 1. Finally, we ran a third round of N full syncs from 1 to 2, 2 to 3, and so on, ending with a full sync from N to 1.

Figure 15 shows the number of distinct sync times present in the tree on replica 1 after the second round and after the third round. The number of distinct sync times after the second round cannot exceed $N + 1$, because there are N distinct sync times induced by the partial syncs of the N chosen leaf directories, plus one sync time for all the directories and files not involved in any partial sync. Sometimes there are fewer than $N + 1$ distinct sync times, because the same leaf directory was chosen at random multiple times. The number of distinct sync times after the third round is always exactly one; as predicted, the full syncs have restored uniformity to the sync times.

Section 3.5 also argued that the storage cost of the metadata database was $O(RD + F)$ rather than the $O(RD + RF)$ cost of version vectors. To test this, we created a set of file trees as follows. Each tree has N leaf directories containing N files each, a total of N^2 files. We then replicated the tree to each of N replicas, modifying every file in the tree on each replica along the way. Since $D = 2N - 1$ (there are $N - 1$ internal directories in addition to the N leaf directories), but $F = N^2$, the predicted $O(RD + F)$ storage cost should cause a $O(N^2)$ database, but if there is an RF term in the storage cost, it will cause the database size to grow as N^3.

Figure 16 shows the total number of vector elements in the metadata database on replica 1 after the N syncs. The size is quadratic, not cubic, in N, confirming section 3.5.

The total number of vector elements is $4N^2 + 2N - 1$. Each of the N^2 files has 1-element modification time and 1-element creation time. Each of the $2N-1$ directories has an N-element modification time and a 1-element creation time. The sync time on the root is an N-element vector. The sync times on the rest of the tree are identical to the sync time on the root, so the stored differences require no vector elements. The total is exactly the count observed in practice.

5 Related Work

Synchronization of replicated data is a well-studied problem. Vector time pairs are a refinement of our communication timestamps system [2]. We view vector time pairs as another step in the development of version vectors. The main alternative to version vectors is logging.

5.1 Version Vectors

A large number of distributed systems have used version vectors. The main problems with version vectors are that they cannot record conflict resolutions (causing occasional false conflicts), they require distributed consensus algorithms to reclaim space occupied by deletion notices, and they require bandwidth proportional to the number of files

in the system to identify the set of changed files during a synchronization. Real-world version vector systems have worked around one or more of these shortcomings in various ways. Vector time pairs provide an elegant way to solve all these shortcomings.

The most common examples of version vector-based systems are the Ficus [4] and Coda [10] distributed file systems. A more recent example is the Pangaea wide-area file system [17]. We discuss each system in more detail and describe how they could benefit from vector time pairs.

Ficus (and its user-level successor, Rumor) is careful about trying to reduce network bandwidth for distant replicas [7], using "last-update times" that record the last time two replicas directly communicated with each other and completed a sync of the entire file system. This reduces the bandwidth costs during synchronization (achieving goal 4) but only for full synchronizations (effectively giving up goal 5). The last-update times can be viewed as a primitive version of vector synchronization times. In contrast to last-update times, vector time pair algorithms record vector synchronization times for every directory and file in the tree and propagate them via indirect communication as well as direct communication. Ficus also introduced distributed consensus algorithms to reclaim space occupied by deletion notices and by the version vectors themselves [5]. These algorithms are *global operations*, requiring knowledge of the entire system. In contrast, vector time pair systems can reclaim space occupied by deletion notices and significantly compress the metadata for extant files using *local operations* that only require knowledge of the local replica.

Coda also uses version vectors. To make them scale better, it distinguishes between servers, which are connected to each other via high-speed links, and clients, which may only be intermittently connected. It uses version vectors to track changes made by servers, but treats clients as second-class citizens. Changes made by a client must be shepherded by a server. This two-level split keeps the version vectors small and avoids the need to track client membership in the system. Using vector time pairs would remove the need for this separation. Coda does not bother to implement Ficus's distributed garbage collection algorithms. Instead, it assumes that the central servers are connected well enough to coordinate in a simultaneous conversation.

Pangaea is a wide-area file system with dynamic membership. Because it uses version vectors, it must track this membership precisely. If a replica has not been seen for 30 days, the replica is ejected by agreement among the remaining replicas. Such a complicated protocol would not be necessary if Pangaea used the vector time pair algorithms, which are not sensitive to whether particular replicas are currently active in the system.

5.2 Logging

Bayou [14] and Ivy [13] are distributed systems designed to enable collaboration among many participants. The details of data distribution are quite different, but the synchronization structure is very similar. Replicas make changes in a private log, which is then shared with the rest of the system. Each replica keeps a single version vector tracking how much of each other replica's log it has received. Because the log structure imposes a linear ordering on events at a given replica, the version vector is really being used as a synchronization time, yielding many of the benefits and clarity of vector time pairs. The log structure imposes a restriction too: the synchronization must be totally ordered. It is not possible for a replica to pick up another replica's change 5 without changes 1 through 4. If these changes involve different objects, selective application of the changes might well be desirable. Using vector time pairs directly would allow Bayou and Ivy to synchronize individual objects independently. This is more of a benefit in Ivy, which implements a distributed file system, than in Bayou, which is intended as a more general framework and might not have a concept of individual objects, depending on the application.

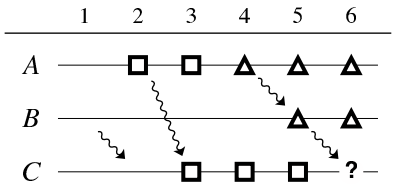


Figure 17: A synchronization pattern that Unison's "change lists" approach cannot handle.
In the sync from B to C at time 6, B's version and C's version have both changed since the no-op sync from B to C at time 2, so Unison will report a conflict. The versions are not actually in conflict — △ is derived from □.

5.3 Unison

Unison is a file synchronizer similar to Rumor and Tra. Its design and implementation is simplified by only handling the case where a pair of computer synchronizes. When two computer synchronize, each creates a list of files changed since the last time they synchronized. Files on only one computer's list are copied to the other computer. If a file is on both lists, Unison reports a conflict.

Sending "changed lists" allows Unison to use a minimal amount of network bandwidth to identify the set of changed files, while also easily supporting partial synchronizations. Unison can also reclaim deletion notices easily — once a deletion on one computer has propagated to the other computer, all trace of the deleted file can be removed.

Unfortunately, Unison either imposes restrictions on synchronization patterns or suffers from false conflicts. Using Unison's "synchronize a pair of computers" primitive, it is possible to synchronize a network of computers, but only if the synchronization pattern has no cycles. For example, if we want to synchronize computers A, B, and C, we could use Unison by deciding that A and B will synchronize, and B and C will synchronize, but A and C will never synchronize. If a cycle is ever completed, Unison will not be able to distinguish changes made on one computer from changes

made on another, as shown in Figure 17.

6 Conclusion

We have presented *vector time pairs*, a new way to track optimistically replicated data. Our main insight is that synchronization history should be maintained separately from modification history. This separation yields algorithms significantly simpler than those used with traditional version vectors, with significant improvements in functionality. We hope that the use of vector time pairs will make the optimistically replicated systems of the future easier to build, to manage, and to use.

7 Acknowledgments

We thank Ken Birman, Frans Kaashoek, Max Krohn, Robert Morris, Norman Ramsey, Margo Seltzer, and Michael Walfish for useful feedback and helpful discussions during the past four years. We thank Frans, Robert, and Margo a second time for their encouragement. We are also grateful to a large number of anonymous reviewers for feedback on drafts of this paper.

Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

William Josephson is supported by a National Science Foundation Graduate Research Fellowship.

References

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

[2] Russ Cox and William Josephson. Communication timestamps for file system synchronization. Technical Report TR-01-01, Computer Science Group, Harvard University, 2001.

[3] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[4] Richard Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. PhD thesis, University of California at Los Angeles, 1991.

[5] Richard Guy, Gerald Popek, and Thomas W. Page Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.

[6] Richard Guy, Peter Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *Proceedings of the 17th International Conference on Conceptual Modeling: Workshop on Mobile Data Access*, Singapore, 1998.

[7] John S. Heidemann, Thomas W. Page Jr., Richard C. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 2–5, November 1992.

[8] Trevor Jim, Benjamin C. Pierce, and Jérôme Vouillon. How to build a file synchronizer. Manuscript; available from `http://www.cis.upenn.edu/~bcpierce/papers`, 2004.

[9] D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Soughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Steven Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[10] James J. Kistler and M. Satyanarayanan. Disconnected operating in the coda file system. *ACM Transactions on Computing Systems*, 6(1):1–25, February 1992.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishes B. V., 1989.

[13] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[14] Karin Peterson, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th Annual Symposium on Operating Systems Principles (SOSP 16)*, pages 288–301, Saint Malo, France, October 1997.

[15] Benjamin C. Pierce and Jérôme Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of CIS, University of Pennsylvania, 2004.

[16] Norman Ramsey and Elod Csirmaz. An algebraic approach to file synchronization. *Foundations of Software Engineering*, pages 175–185, September 2001.

[17] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming agressive replication in the pangaea wide-area file system. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[18] Reinhard Schwarz and Fridedmann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[19] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.