Optimistic Replication Using Vector Time Pairs

Russ Cox, *MIT CSAIL*
William Josephson, *Cornell CS*
rsc@csail.mit.edu, wkj@cs.cornell.edu

Abstract

Anyone who uses more than one computer system is aware of the data management problem posed by doing so: sharing files among systems requires propagation of changes to the other systems by some synchronization method. The proliferation of mobile and small devices with low-bandwidth or intermittent network connectivity introduces new constraints on synchronization algorithms.

We present new algorithms for tracking updates in a optimistically replicated hierarchical data. The algorithms use a pair of vector times, one tracking modifications and a second tracking synchronizations; previous systems have typically used only a single vector conflating the two meanings. Algorithms using vector time pairs are considerably simpler and more flexible than traditional algorithms using single vectors. Vector time pairs are also significantly easier to compress than single vectors. These claims are supported by a theoretical analysis as well as simulation of the algorithms on various synthetic workloads derived from real-world traces.

1. Introduction

Data replication plays an increasingly important role in today's computing environments. A typical computer user might store data on a shared file server, an office PC, a home PC, a notebook, and a PDA. The industry-wide SyncML initiative [27] is aimed at making replication of data easier and thus more commonplace than it is today. The increasing storage capacities of handheld devices like PDAs, music players, and even USB memory sticks make them attractive ways to transfer large amounts of data as well. It seems a safe bet that handheld devices will continue to decrease in size but increase in storage capacity. All of these trends suggest that the management of replicas and replicated data will become ever more central in tomorrow's computing environments.

The replication in the examples just given is typically *optimistic*, meaning that any replica can initiate changes to any data. Previous research on the use of optimistic replication has found that conflicts are rare in many applications. For instance, in file systems, most files are written only by one user, who acts as a human write token [19]. Indeed, disconnected AFS [13], Coda [14], and Ficus [7] all employ optimistic replication to provide a high availability file system in the face of limited communication.

Optimistically replicated systems must employ some bookkeeping scheme to propagate changes and also to detect concurrent changes, or conflicts, that need to be reconciled. We present *vector time pairs* as a new way to track changes in an optimistically replicated system. Vector time pair algorithms have the following desirable properties:

- can support an arbitrary number of replicas with arbitrary dynamic communication patterns
- replicas do not need to be added to or removed from the system explicitly
- synchronizations focus very quickly on the data that needs to be transferred
- partial synchronizations are easily implemented
- ‘‘sink’’ replicas, which never initiate file system changes, cause no extra work for other replicas
- the performance of the system is independent of the synchronization frequency among the replicas
- none of the algorithms attempt to construct global knowledge about the system

As discussed later in the paper, no current algorithms used for optimistic replication have all of these properties.

The main contribution of this paper is the insight that traditional synchronization metadata can be separated into metadata tracking ‘‘what you have’’ and metadata tracking ‘‘what you know.’’ The vector time pair algorithms follow immediately from this insight.

Roadmap

In the remainder of this paper, we review the scenarios that a synchronization algorithm must handle (section 2). Then we present the vector time pair algorithms (section 3). We describe one concrete implementation of vector time pairs in a user-level file synchronizer (section 4). Using a simulation framework to level the playing field, we then compare vector time pairs with version vectors (section 5) empirically. Finally we discuss related work and how it has been constrained by less flexible algorithms (section 6).

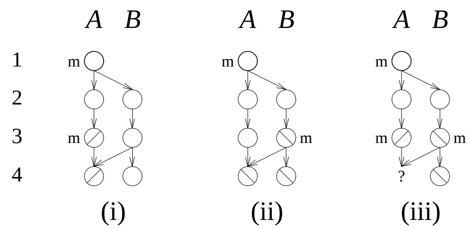
2. Synchronization semantics

Before discussing synchronization algorithms, we must define synchronization. A strictly formal mathematical definition is an ongoing research topic (see, for example, Balasubramanian and Pierce [1], Ramsey and Csirmaz [24], and Pierce and Voilloun [22]), 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 in the original paper on version vectors [20] and because, when restricted to a pair of replicas, it is equivalent to the definition of synchronization used by the popular Unison synchronizer [1].

We consider only unidirectional synchronizations, in which changes from one replica are propagated to a second, but changes on the second do not propagate to the first. There are many cases in which this is desirable. For example, the second machine might be less trusted than the first. More importantly, unidirectional synchronization is the more general case: we can build bidirectional synchronization by applying unidirectional synchronization once in each direction. When we say that *B synchronizes from A*, we mean that a unidirectional synchronization propagated information from *B* to *A*.

We want synchronization to provide a ‘‘no lost updates’’ guarantee. Specifically, suppose each file is represented as a log of updates made over the course of its lifetime, beginning with its initial creation. If two replicas have different copies of a file (call the copies F_A and F_B), it is safe for F_A to replace F_B *only* if F_A's log is a prefix of F_B's. If this is satisfied, then all of the updates represented by F_B are also present in F_A: eliminating F_B will not lose updates. To explore how the ‘‘no lost updates’’ rule guides synchronization, we present a sequence of examples.

Suppose a single file is kept on a pair of replicas *A* and *B*. Consider the following three graphs showing how information about the file might propagate between the replicas:



(i) (ii) (iii)

In each picture, each row shows the state of the replicas at one time unit. Each circle represents the file as it exists on a particular replica at some point in time. Modifications to the file contents are marked by an ‘‘m.’’ As a further reminder, file contents are represented by the patterns inside the circle. Arrows indicate the flow of file system information, so that a downward arrow reflects the file staying on the replica over time, while a diagonal arrow marks a unidirectional synchronization between the replicas at its endpoints. Synchronizations happen ‘‘between’’ time units.

There are three important facts about the notation used in the diagrams:

- First, two changes are considered *independent* if there is no path in the synchronization graph from one to the other. For reasons of brevity, independent changes in the examples are shown as happening at the same time on both replicas, but that is not necessary. The only necessary condition is that the first change is done without knowledge of the second, and vice versa.
- Second, we have shown a global clock marking time on all replicas, but this for the purpose of exposition only. The replicas need not share a common clock: synchronization decisions depend only on the past synchronization history (edges in the graph), not on the exact times of synchronization.
- Third, by clock we mean a monotonically increasing value, typically implemented as an event counter. We do not mean the system's ‘‘wall clock’’ time, which may be unreliable in various ways.

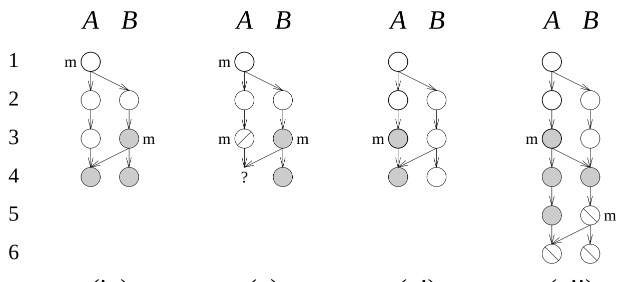
All three graphs begin the same way. At time 1, *A* creates a new file. *B* synchronizes from *A*, so at time 2 the file exists on *B*.[1] At time 3, one or both replicas change the file. Then *A* synchronizes from *B*, with varying results:

- (i) *A* changes its copy of the file. The ensuing synchronization is a no-op, since *B* has no changes to report. (Synchronization in the opposite direction would copy the updated file from *A* to *B*.)
- (ii) *B* changes its copy of the file. The ensuing synchronization copies the new file from *B* to *A*.
- (iii) both *A* and *B* change their copies of the file. The ensuing synchronization cannot safely choose one replica's copy, since doing so would lose the change made by the other replica. The modification histories are (A:1 A:3) and (A:1 B:3). Neither is a prefix of the other. The synchronizer reports a conflict to be resolved by external means, either automatic or manual.

[1] We have not explained the reason the synchronizer chose to copy the file from *A* to *B*. We will cover file creation in the next sequence of examples.

The graphs allow an alternate characterization of the ‘‘no lost updates’’ requirement. A version v_1 of a file may be replaced by another version v_2 if v_2 is the descendant of some node with the version v_1.

The three graphs above illustrate a synchronizer’s choices when presented with differing versions of a file: do nothing, copy from one replica to another, or report a conflict. The next four graphs illustrate the choices a synchronizer has when confronted with two replicas, only one of which has an existing file. In these graphs, a light grey filled circle marks that the file has been deleted from the replica.



(iv) (v) (vi) (vii)

As before, our examples all start with *A* and *B* holding the same file at time 2. The synchronizer’s behavior when presented with only one copy of a file depends on the past history:

- (iv) If *B* deleted the file, synchronization propagates the deletion to *A*.
- (v) If *B* deleted the file, but *A* modified it independently, synchronization reports a conflict: the deletion is incompatible with the update.
- (vi) If *A* deleted the file, synchronization is a no-op. (Synchronization in the opposite direction will delete the file from *B*.)
- (vii) If *A* deleted the file and propagated the deletion to *B* but then *B* created a new version of the file, synchronization propagates the new file to *A*.

Thus the possible choices are: delete a file, report a conflict, do nothing, or create a file.

In all seven cases, it should be clear that the correct decision is entirely determined by the ‘‘no lost updates’’ principle.

3. Vector time pairs

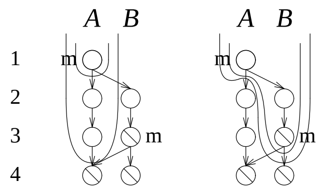
In this section we present the vector time pair algorithms. Starting with the synchronization of a single file, we consider the time and space requirements and in both cases find effective ways to reduce the requirements.

In our algorithm, each replica maintains two *vector times* [4] [15] for each replicated file and directory. A vector time is an array of local machine times (event counters), one for each replica in the system.

The first vector time is a *modification time*; it tracks the modification history of the file. The vector entry for replica *r* is the time when *r* last modified the file. For example, at the end of the scenario depicted in figure (ii) above, the file on *A* has modification time $(A{:}1\ B{:}3)$. The second vector time is a *synchronization time*; it tracks the synchronization history of the file. The vector entry for replica *r* is the time when this file last passed through *r* on its way to the current replica. At the end of scenario (ii), the file on *A* has synchronization time $(A{:}4\ B{:}3)$.

The modification time tracks ‘‘what we have’’ in the file; in the example, $(A{:}1\ B{:}3)$ records that we have the changes made by *A* at time 1 and *B* at time 3. In contrast, the synchronization time tracks ‘‘what we know’’ about the file; In the example, $(A{:}4\ B{:}3)$ records that we know about the file as it existed on *A* at time 4.

Another way to think about the vector times is that they encode the reachable ancestors of a file version (graph node). The modification time specifies, for each replica, the most recent reachable ancestor at which a modification occurred. The synchronization time specifies, for each replica, the most recent reachable ancestor on that replica. If we know one node on a replica is an ancestor, we also know that all earlier nodes on that replica are ancestors. Because of this, the synchronization and modification times encode two related ancestor sets for a given node. The synchronization and modification sets provide enough information to make synchronization decisions. As an example, consider scenario (ii) again. The nodes involved in the synchronization producing $A{:}4$ are $A{:}3$ and $B{:}3$, which have synchronization and modification sets as shown. The inner boundary marks the modification set, while the outer boundary marks the synchronization set:



It is safe to replace $A{:}3$ with $B{:}3$ because $A{:}3$’s modification set is contained in $B{:}3$’s synchronization set: $B{:}3$ ‘‘knows’’ about the file that $A{:}3$ ‘‘has.’’

This set comparison is implemented using vector time comparisons. Comparing two vector times *u* and *v* has four possible outcomes:

- $u = v$. The vectors agree on every element.
- $u < v$. The entries in *u* are less than or equal to the corresponding entries in *v*, but $u \neq v$.
- $u > v$. The entries in *u* are greater than or equal to the corresponding entries in *v*, but $u \neq v$.
- $u \parallel v$. None of the above hold: *u* and *v* are incompara-

ble. Some entries in *u* are greater than the corresponding entries in *v*, while some are less.

These are the same outcomes that would be reached comparing the corresponding sets of nodes to see whether one was a subset of the other.

The general synchronization decision can be expressed as:

```
// synchronize file from $A$ to $B$
sync(file) ≡
    if $m_A \le s_B$
        // $B$ knows about all of $A$'s updates
        $s_B = \max(s_A, s_B)$
    else if $m_B \le s_A$
        // $A$ knows about all of $B$'s updates
        copy file from $A$ to $B$
        $s_B = \max(s_A, s_B)$
    else
        // $A$ doesn't know about some of $B$'s updates
        // and $B$ doesn't know about some of $A$'s updates.
        // No way to proceed without losing updates.
        report conflict
```

The vector times m_A, m_B, s_A, and s_B are specific to the particular file being synchronized, but writing m_A rather than $m_A(\text{file})$ is less cluttered. The comparison $m_A \le s_B$ checks whether *A*'s modification set is contained in *B*'s synchronization set. That is, it checks that all the updates present in *A* are also present in *B*.

The first two tests check whether all of one copy's updates are present in the other copy. If so, the more complete copy is used. In the final ‘‘else,’’ each copy has updates not present in the other, so using either one would lose updates. We cannot do that, so we report a conflict.

3.1. Time requirements

We have an algorithm that correctly synchronizes a single file. One way to synchronize an entire file system is to run the algorithm for each file in the system. When the file system contains many files and only a few need to be synchronized, running synchronization on every file in the system wastes both time and inter-replica communication bandwidth. It would be useful to be able to prune irrelevant paths from the search, quickly homing in on the changed files.

We can do this by keeping a synchronization and modification time for directories as well as files. The synchronization time of a directory is the elementwise minimum of the synchronization times of its children. In the set analogy, the synchronization set is the intersection of the children's synchronization sets. The modification time of a directory is the elementwise maximum of the modification times of its children. In the set analogy, the modification set is the union of the children's modification sets.

When synchronizing, a directory can be skipped if the modification set on one replica is a subset of the synchronization set on the other replica and vice versa. The following algorithm only visits directories that need synchronization:

```
// synchronize tree rooted at dir from $A$ to $B$
sync(dir) ≡
    if $m_A \le s_B$
        $B$ has all the changes on $A$; do nothing
    else
        for each child in dir $m_A(child) \le s_B$]
            sync(child)
    $s_B = \max(s_A, s_B)$
```

The file synchronization algorithm must be tweaked as well. After copying a file from *A* to *B*, the modification time of the file's parent directory must be set, along with that parent's parent directory, and so on to the root of the tree.

```
// synchronize file from $A$ to $B$
sync(file) ≡
    if $m_A \not\le s_B$ and $m_B \not\le s_A$
        report conflict
    else if $m_A \le s_B$
        $B$ is up-to-date
        $s_B = \max(s_A, s_B)$
    else
        copy file from $A$ to $B$
        $m_B = m_A$
        mtimeup(parent(file), $m_B$)
        $s_B = \max(s_A, s_B)$

mtimeup(path, m) ≡
    $m_B(\text{path}) = \max(m_B(\text{path}), m)$
    if path has parent
        mtimeup(parent(path))
```

Using this algorithm reduces the time required to identify the needed synchronization tasks. Whereas the previous algorithm ran in time proportional to the number of files in the system, this algorithm runs in time proportional to the number of changed files.

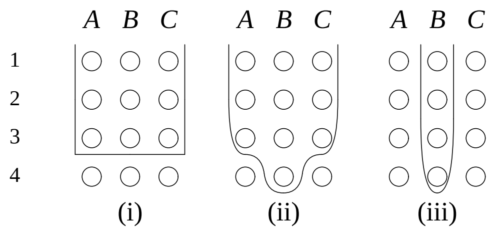
3.2. Space requirements

Each replica must maintain two vector times for every file and directory in the file system. Each vector time has a size proportional to the number of replicas in the system. In a naive implementation, a system with *R* replicas storing *D* directories and *F* files requires $O(R(D+F))$ space. This space requirement can be reduced significantly with two important optimizations.

Both rely on the fact that the underlying vector representation omits zero entries, as we have have done in this presentation. That is, it is cheaper to store $(A{:}3)$ (implicitly $(A{:}3\ B{:}0)$) than $(A{:}3\ B{:}1)$.

The first optimization comes from the observation that, by definition, synchronization times are monotonically increasing along each path in the file system. Thus, for a given file or directory, we can store just the delta between its synchronization time and its parent's synchronization time. This scheme is further helped by the observation that that there are likely to be very few distinct synchronization times on a given replica. To see why, suppose that synchronizations 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. In the presence of full synchronizations, the file system metadata will regularly be in states in which there is a single unique synchronization time. 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` or `/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. Because of this, most of the stored deltas will be empty (zero)! If there are S unique synchronization times throughout the tree, the storage cost of the synchronization times is only $O(RS+F)$: the extra F is a constant space per-file to store the zero delta.

The second optimization comes from the fact that modification times can often be reduced to singleton vectors. Consider the following example:



Suppose that a file or directory with modification time $(A{:}2\ B{:}2\ C{:}2)$ exists on replica B, depicted in (i). Now suppose that the file is modified on B at time 3. The standard vector time manipulations would change the middle element of the vector to $B{:}3$, yielding $(A{:}2\ B{:}3\ C{:}2)$, depicted in (ii). We argue that given the synchronization algorithm, it is safe instead to set the modification time to the singleton vector $(B{:}3)$, depicted in (iii).

The synchronization algorithm only compares modification times to synchronization times. Because the time on B is currently 3, synchronization vector times s on other replicas cannot contain a B element greater than or equal to 3. Whether we set the modification time m to $(A{:}2\ B{:}3\ C{:}2)$ or to $(B{:}3)$, the test $m \leqq s$ will be false for every s currently on other replicas. We also need to check that at an s will not be introduced later such that $(A{:}2\ B{:}3\ C{:}2) \nleqq s$ but $(B{:}3) \leqq s$, but this is simply reasoned. In order for a replica to get a synchronization time s greater than $(B{:}3)$, it must know about the contents of B at time 3. But since $(A{:}2\ C{:}2)$ were part of the contents of B at time 3, $(A{:}2\ C{:}2) \leqq s$ so $(A{:}2\ B{:}3\ C{:}2) \leqq s$. Thus $(B{:}3)$ and $(A{:}2\ B{:}3\ C{:}2)$ are indistinguishable from the point of view of our synchronization algorithm. The same argument applies in general: whenever a replica notes a local modification to a file or directory, it can safely set the modification time on that file or directory to the singleton vector corresponding to the current time on the replica. This can result in a space savings if the vector representation omits the zero entries, as we have done in this presentation.

Since file modification times are propagated between replicas by direct copying, file modification times will stay singletons as they pass from replica to replica. Directory modification times are incorporated via the ''max'' operator, so directory times will still be full vectors in most cases. Since only the directory times are R-element vectors, the space requirement for the modification times reduces to $O(RD+F)$.

Together, these optimizations reduce the per-replica space requirement for metadata to $O(RS+RD+F)$. Since S is expected to be very small, this reduces to $O(RD+F)$.

3.3. Delete propagation

The algorithm presented so far requires replicas to keep metadata about deleted files. This metadata can be thought of as a *deletion notice* that propagates in the same manner as the file itself would have. It is desirable that replicas be able to destroy deletion notices eventually, in order to reclaim the space they occupy. A deletion notice covers not only the most recently deleted file but all deleted earlier incarnations of the file.

The easiest way to allow destruction of deletion notices is to change the algorithm used for file synchronization when one of the two files is actually a deletion notice. Specifically, we restructure the algorithm so that only the synchronization time of the notice is needed. In order to do this, we must add a third piece of metadata:

each extant file is tagged with its creation time, a singleton vector.

With this extra data, we can make decisions about create and delete operations while requiring that the deletion notice contribute only a synchronization time. The new algorithm is shown below. (We show a bidirectional synchronization for brevity. The appropriate unidirectional algorithms can be obtained by restricting changes to A or to B as needed.)

```
// file exists on $A$, but not on $B$
sync(file) ≡
    if $c_A \le s_B$
        // $B$ has a deletion notice for $A$'s file ...
        if $m_A \le s_B$
            // ... and the deletion covers all $A$'s changes
            delete file on $A$
            $\mathrm{mtimeup}_A(file, \mathrm{now}_A)$
            $s_A = \max(s_A, s_B)$
        else
            // ... but $A$ has new changes to the file
            report conflict
    else
        // $B$ has never heard of this file
        copy file from $A$ to $B$
        $m_B = m_A$
        $\mathrm{mtimeup}_B(file, m_A(p))$
        $s_A = s_B = \max(s_A, s_B)$
```

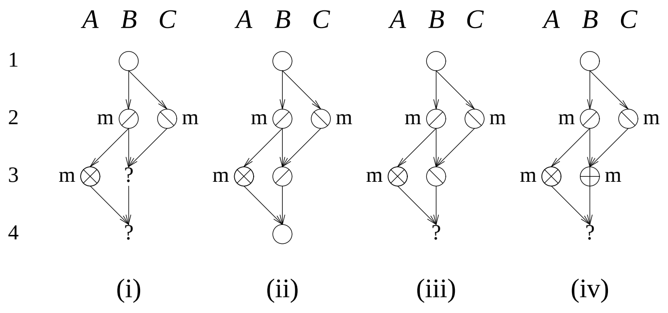
Having the creation time c_A lets the algorithm decide whether B has ever heard of the file that A has. If $c_A \le s_B$ (that is, B knows about the creation of the file) yet B does not have the file, then B must also know about a deletion. The algorithm must then check whether B knows about all the changes to the file that A has. If so, then the deleted version of the file B knows about included all the updates A has, so deleting A's copy will destroy only updates that were destroyed by the original deletion. On the other hand, if B does not know about all of A's changes, then a conflict must be reported: deleting A's copy would lose updates that the original deletion did not cover. Finally, at the outer ‘‘else’’ now, if $c_A \not\le s_B$, then B does not know anything about this file and should create.

Once the directory containing the deletion notice is completely synchronized, the synchronization time on the directory and the deletion notice will be the same. From that point onward, they will always be the same (unless, of course, a new deletion notice is received). Thus the deletion notice can be removed. If the synchronization time from the deletion notice is needed in the future, the synchronization time on the directory can be used in its place. Collection of deletion notices meshes well with the synchronization time delta encoding described above: once the delta is zero, the deletion notice can be removed.

Destroying deletion notices is a *local* operation; it depends only on other metadata already kept by the replica. By storing some extra metadata for each active file (the creation time), we have reduced the amount of metadata for each deleted file to something that can be reclaimed easily.

3.4. Conflict resolution

Although the synchronization algorithm relies on external means to resolve conflicts, a good synchronization algorithm needs to be able to propagate the resolutions, so that the same conflicts are not reported multiple times. For example, consider the synchronization sequence in (i) here:



Whether a conflict occurs at $B{:}4$ depends on the resolution of the conflict at $B{:}3$. (ii)-(iv) illustrate the cases:

- (ii) If the conflict at $B{:}3$ is resolved in favor of $B{:}2$'s copy then there should be no conflict at $B{:}4$, because $A{:}3$ is derived from $B{:}2$.
- (iii) If the conflict at $B{:}3$ is resolved in favor of $C{:}2$'s copy then there should be a conflict at $B{:}4$, because $A{:}3$ is not derived from $C{:}2$.
- (iv) If the conflict at $B{:}3$ is resolved by a modification that merges $B{:}2$'s and $C{:}2$'s copies, then there should be a conflict at $B{:}4$, because $A{:}3$ is not derived from $B{:}3$.

This is handled easily by the vector times: when conflicts are resolved, the modification time on the file is set to the chosen resolution, and the synchronization time is set to the union (elementwise max) of the two synchronization times on the conflicting files. Doing this records that the replica storing the resolved file ‘‘knows’’ about both the files that conflicted even if it decides to keep only some of the changes.

3.5. Replica deletion

Many replicated systems track the current state of other replicas and change behavior based on whether the system as a whole is up-to-date. In such systems, a replica that is no longer participating keeps the whole system from being up-to-date. In order to make progress the replica must be removed, explicitly or implicitly. For example, Ficus and Coda require explicit removals, while Pangaea [25] removes a replica if it is inactive for 30 days. Neither solution is particularly attractive: explicit removal requires manual effort and introduces the possibility of human error, while implicit removal is only approximate.

Using vector time pairs removes this restriction: replicas that are no longer participating cause no trouble for the rest of the replicas.

3.6. Partial synchronization

It is worth mentioning explicitly that the algorithm as presented ‘‘remembers’’ partial synchronizations. That is, suppose we synchronize only the directory subtree rooted at `/usr/sys`. This will update the synchronization time of `/usr/sys` and all files and directories below it. If we then we do a full synchronization with a replica that has no additional changes to `/usr/sys`, the updated synchronization time on the directory will cause the algorithm to ‘‘remember’’ that the partial synchronization occurred, and thus it will not consider `/usr/sys` further.

The method of recording partial synchronizations falls out naturally from the rest of the algorithms. This seems a trivial point, but we will see later that other systems have struggled quite a bit with partial synchronization.

3.7. Algorithm summary

In summary, each replica stores a vector time pair for each replicated file or directory. The pair includes a *synchronization time*, which encodes what the replica ‘‘knows’’ about a file or directory and a *modification time*, which encodes what changes the replica ‘‘has’’ for a given file or directory. The two can be different because the synchronization time includes knowledge of non-changes: if a file has synchronization time ($A{:}4$) but modification time ($A{:}1$) then we know about everything that happened up until time 4 on replica *A*, but the most recent change that we have is from time 1.

Having both vector times facilitates pruning of synchronizations so they only consider relevant files, encoding of conflict resolutions, and recording of the results of partial synchronizations. The storage requirements for the vector times can be significantly reduced by delta-encoding the synchronization times.

4. Implementation

We have implemented the vector time pair algorithms in a user-level file system synchronizer called Tra (the name has been changed for anonymity purposes). We built an initial prototype in 1,800 lines of Python. The production version of Tra is implemented in 12,000 lines of portable C. The amount of operating system-specific code is minimal, typically between 200 and 500 lines for each system. Tra runs on various Unix flavors, including Digital Unix, FreeBSD, Linux, NetBSD, OpenBSD, and SunOS. In this section we describe the implementation of Tra, starting with a high-level feature overview and then considering some of the low-level engineering.

4.1. High-level features

Tra delivers on the features promised by the vector time pair algorithms:

- allows arbitrary communication patterns
- synchronization time proportional to amount of changed data
- easy partial synchronization
- unidirectional or bidirectional synchronization
- support for ‘‘sink’’ replicas
- has no explicit replica deletion

We use Tra via a simple command-line user interface. The command

```
% tra [ −1ab ] repl-a repl-b [ path ... ]
```

synchronizes the replicas *repl-a* and *repl-b*. Tra runs a partial synchronization rooted at each named *path*. If no paths are named, Tra performs a full synchronization.

By default, synchronization is bidirectional. The `−1` flag causes synchronization to be unidirectional, with information flowing only from *repl-a* to *repl-b*.

Conflicts are reported during the synchronization. They can be forced to resolve in favor of the copy on *repl-a* or *repl-b* by invoking `tra` with the `−a` or `−b` flags. A typical synchronization consists of running `tra` followed by `tra −a` or `tra −b` or both, explicitly naming the paths to resolve.

4.1.1. Sink replicas

A *sink* replica is one which accepts changes from other systems but never initiates or propagates them. (It is tempting to call such replicas *read-only* because they never write changes to other servers, but that is confusing since they are written to by other servers.) Sink replicas can be implemented by always running `tra −1` with the sink replica as *repl-b*. Such synchronizations are no-ops for its local data and metadata do not change at all. That is, sink replicas cause zero overhead for the rest of the system. Thus the system can scale to support an arbitrary number of sink replicas.

We considered making ‘‘sink’’ an explicit property tied to a replica, but it seems that whether a replica is a sink often depends on who the replica is synchronizing with. For example, an operating system distribution using Tra might have a handful of engineers who synchronize with each other and a few distribution servers. Synchronizations between the developers and the servers would be bidirectional. On the other hand, the system's users could use Tra to synchronize their own machines with the distribution servers. In that case, the user machines would act as sinks. Thus, whether a replica is a sink in a particular synchronization could be a policy decision made based on authentication or some other higher-level mechanism.

4.1.2. Automatic conflict reconciliation

Once conflicts are detected, the default behavior is to report them, leaving the user to resolve them. Files might be identical have different modification histories, causing the vector time pair algorithm to report a conflict. Tra could report this as a conflict, but in the common case this is useless: the user has little choice but to merge the two and plod on. Instead, Tra automatically resolves conflicts involving identical files. This behavior can be turned off, but we have found that it is a good default. It is particularly useful when merging two file trees that have previously been kept in sync by other means.

Early prototypes of Tra allowed the user to specify an external program to handle reconciliation. We found the most common reconciliation was the one above. With that case handled, we have not felt the need for a user-level reconciler enough to implement it in the current version of Tra. Coda [14] used a user-level reconciler to great effect for certain kinds of files, so it seems likely that we will add support for it to Tra in the future.

4.1.3. Error messages

We quickly discovered that good error messages are essential to understanding Tra's behavior. When it detected an update/update conflict, the original prototype printed

```
% tra titanic lusitania usr/sys
usr/sys/ken/prf.c: update/update conflict
%
```

which was clearly not enough to understand what was going wrong. Unfortunately, the vector times do not help much. Printing

```
% tra lusitania titanic usr/sys
usr/sys/ken/prf.c: update/update conflict
  lusitania's copy last modified at britannic:34
  titanic's copy last modified at titanic:762
%
```

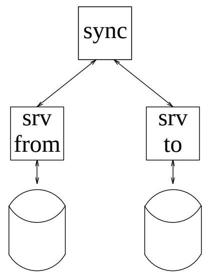
is only slightly better. We learn that the conflict is between a write on `britannic` (that has propagated to `lusitania`) and a write on `titanic`, but the times are counters, meaningless to the user. To address this, we added wall clock times to the vector entries. The wall clock times are ignored in the vector time comparisons but propagate with the rest of the vector. If Tra is running on a file system that records the user who last modified each file, it records and propagates that information too. These make for much better error messages:

```
% tra lusitania titanic usr/sys
usr/sys/ken/prf.c: update/update conflict
  lusitania's copy last modified on britannic \
    Sun Nov 11 17:33:01 EST 2001 (#34)
  titanic's copy last modified on titanic \
    Mon Nov 12 09:12:31 EST 2001 (#762) by ken
%
```

In this example, `britannic` does not record the last writer, so there is no last writer shown in the first record.

4.2. Low-level engineering

Internally, Tra is structured as a synchronizer program (`sync`) that coordinates the synchronization of a ‘‘from’’ and ‘‘to’’ replica server (`srvs`) via remote procedure calls:



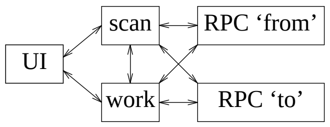
The `srv` programs are charged with maintaining a local database of synchronization and modification times, which the `sync` program queries and modifies throughout the synchronization. The database also typically contains system-dependent signatures of the files; these signatures are used by the `srv` programs to detect when a file changes. The `srvs` use a combination of the `st_dev`, `st_ino`, and `st_mtime` or `st_mtimespec` fields of the stat structure.

Because of the split between `sync` and `srvs`, the programs may all run on different systems: keeping the `srv` file system sweeps local is a big win, as is being able to run `sync` on either the ‘‘from’’ or the ‘‘to’’ system (or a third system).

The replica names provided to `sync` are paths to 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` [30], local execution, or anything else.

4.2.1. Communication structure

Internally, the `sync` module is structured as a collection of cooperatively scheduled user-level threads communicating via queues and stacks, in the style of Hoare's CSP [12] and Pike's Newsqueak [23]. (We use a simple custom-built threading library that is about 1000 lines and requires about 20 lines of code for each new architecture.) The main departure from the CSP/Newsqueak model is that some queues are arbitrarily buffered to avoid deadlocks. There is no visible locking in the main program: all synchronization and scheduling is managed via the communication channels. The diagram below illustrates the main modules and the communication patterns between them:



The connections to the RPC modules are unbuffered channels over which RPC messages are sent.

The buffered queues and stacks hold `SyncPath` structures, which represent paths currently being synchronized. The user interface sends paths to be scanned to the scan module using a buffered stack. The scan module fetches paths off the stack and scans them. It recursively handles directories by pushing the children of the directory onto the stack. We use a stack instead of a queue to force a depth first traversal of the file system rather than a breadth first traversal, the latter requiring much more buffer space. When the scan module decides that there is file system work to be done, like copying or removing a file, it updates the `SyncPath` structure and adds it to the work queue. The work module carries out the operation. If an error occurs while processing a path in either module, the `SyncPath` is marked with the error and added to an event queue read by the user interface. When synchronization of a particular path is finished, the `SyncPath` is sent along a return queue to the module that originally queued it (either the user interface or the scan module). Structuring the main synchronization in this manner allows the user interface flexibility to change the configuration. For example, the command-line interface has a `–n` flag that runs the synchronization in ‘‘no-op’’ mode, printing the operations that would be carried out but not actually executing them. This mode is implemented by replacing the work module with a different module that prints the work to be done rather than doing it. As another example, an extra module could be inserted ahead of the work module to confirm the work queue with the user or to run a plausibility checker to avoid replicating accidental data loss from one site to another [6].

4.2.2. Multithreading

Tra is almost always invoked with at least one of the two `srvs` on another computer across a network. The RPC modules allow multiple outstanding RPCs; to take advantage of this, the sync and work modules run multiple threads that process `SyncPaths` from the respective queues.

We would like to keep enough RPCs outstanding to make good use of the network and keep TCP’s congestion window large. This is more important on low bandwidth and high latency network connections. At the same time, we would like to keep the work queue and scan queue lengths short to avoid wasting resources. Exploring appropriate dynamic adjustment of the thread allocations is future work; we hope that some of the techniques used in SEDA [29] will be useful here.

4.2.3. Deletion notices

One of the most satisfactory parts of Tra is the code that deals with the cleanup of deletion notices. The `srv` programs interact with a database of `Stat` structures containing the metadata for each file and directory in the system. Deletion notices are specially marked `Stat` structures. When the database code is asked to store a deletion notice for a path, it silently discards the notice if the synchronization time of its parent directory is large enough. Similarly, when the database code is asked for the `Stat` structure for a path about which there is no information, it synthesizes a deletion notice using the parent directory’s synchronization time. The synchronization algorithm implementation is blissfully unaware of this behind-the-scenes shuffling, just as it is unaware of the delta encoding of the synchronization times. As far as the `sync` module is concerned, deletion notices never disappear.

4.2.4. Quick copy and comparison

Tra implements a variant of the rsync [28] algorithm for quickly copying a file when the destination already has a similar file. The variant inserts boundaries using a rolling hash function as in [17] to reduce the number of hashes that need to be computed. This algorithm implicitly assumes that two different blocks of data will always have different SHA1 hashes. The auto-resolver described earlier also uses SHA1 hashes to avoid copying the file just to discover that it is identical to the file it already had. Henson [11] presents interesting arguments against the use of so-called ‘‘compare-by-hash’’ algorithms. We decided that we’re willing to take the risk. Paranoid users can disable both optimizations.

5. Evaluation

We evaluate vector time pairs and Tra by comparing them to relevant related systems. Vector time pairs are qualitatively different from other algorithms used to track synchronizations. Thus, we first present a qualitative evaluation, concentrating on whether the various algorithms support particular features. Vector time pairs are similar enough in functionality to vector time that it is meaningful to quantitatively compare a few performance characteristics of the two schemes. We do that second. Finally, to establish that the implementation of Tra is competitive with other similar tools, we present a rough performance comparison.

5.1. Qualitative evaluation

Systems supporting optimistic replication have traditionally used (usually one of, but sometimes combinations of) three basic methods:

- central coordination
- logging
- version vectors

This section compares vector time pairs with these methods on various points. Some of the points will be revisited quantitatively in the next section. First we give a brief overview of the algorithms.

By central coordination we mean a system where one master server coordinates a collection of replicas. Replicas only synchronize with the master, not with each other. The master keeps a per-file version number of its own, and the replicas need only keep track of which version of each file they currently have. A typical example of a centrally coordinated system is the CVS [2] source control system.

In a logging system, replicas keep their own logs of changes they make to the data. A replica only ever writes to its own log, but it reads the logs of all the other replicas, using all the logs to construct its conception of the data.

Version vector schemes associate a vector time called a version vector with each file in the replicated system [20]. Version vectors can be thought of as a conflation of modification and synchronization times. The general synchronization algorithm using version vectors is the algorithm using modification and synchronization times, but with the version vector *v* substituted for the two vector times *m* and *s*.

```
// synchronize from $A$ to $B$
sync(file) $\equiv$
    if $v_A \leqq v_B$
        $B$ is up-to-date; do nothing
    else if $v_B < v_A$
        $B$ can safely be updated by $A$
    else
        return conflict
```

As we will see, the conflation of the two ideas into one vector makes version vector algorithms more complicated than vector time pair algorithms.

5.1.1. Loose replica membership

In particularly dynamic systems, it is desirable that replicas can come and go easily and with little overhead.

Systems based on central coordination have no problem adding more replicas: the replicas keep track of their state relative to the master, and the master needn't keep any per-replica state at all.

Logging systems typically do have a large overhead associated with adding new replicas, since each replica maintains its own log, and the other replicas need to know about the log. Most operations scale with the number of logs, so this overhead is significant.

Version vector systems also have a large overhead associated with adding and removing replicas. Having more replicas makes the version vectors larger. Version vector systems typically use a two-phase distributed garbage collection algorithm to decide when particular updates have been distributed to all replicas in the system. Once an update has been applied everywhere, the metadata concerning that update can be destroyed. These global algorithms are complicated and depend on having all replicas available to participate. This in turn requires having a precise idea of which replicas are in the system. Even ‘‘sink’’ replicas would need to be included in these proceedings. Thus other replicas must be informed when a replica leaves. The generation of these notices is problematic: if it is automated, the system might incorrectly decide a replica is dead when in fact it is on an extended vacation. On the other hand, if it is manual, it is cumbersome and open to human error. Version vectors can be used without the global algorithms, but the global algorithms are often necessary to help the system scale better: otherwise old data lingers in the system forever.

The vector time pair algorithms avoid these problems by restructuring the metadata so that maintenance algorithms are always purely local operations, eliminating the need for replicas to track the system as a whole. ‘‘Sink’’ replicas are particularly cheap: the first-class replicas even record their presence in any way.

5.1.2. Performance independent of synchronization frequency

In a system with a diverse replica set (as would happen with a collection of PDAs and other mobile devices), it is desirable that the performance of the system does not depend on the slowest participant. In this case by slow we mean that the replica does not synchronize with others very frequently.

In a centrally coordinated system, the coordinator need not record any per-replica information. The replicas need not record any information about other replicas either. Thus, performance *must* be independent of synchronization frequency: each replica is oblivious to the existence of the others.

Logging systems are mostly independent of synchronization frequency as well: they have no operations that depend on action by other replicas. Still, the price they pay for this is that old information stays in the system forever. It is not typical for logging systems to prune their logs once all replicas have been made aware of the changes; if a logging system wanted to do this, it would need to resort to global knowledge algorithms like version vectors employ.

Version vector systems are independent of synchronization frequency only if they do not use the global knowledge algorithms to do background cleanup. The global algorithms require two rounds in which every replica participates, and thus they are only as fast as the slowest participant.

The vector time pair algorithms are not sensitive to synchronization frequency. If a replica does not synchronize often, then it alone bears the burden of tracking its state versus the state of the other replicas. The synchronization time allows an out-of-date replica to reason ‘‘I wouldn’t have known about that, so it must be new’’ or ‘‘I should know about that, so I must have deleted it.’’ The replicas individually keep track of how much they know. In contrast, in the version vector algorithms, replicas keep track of how much everyone else knows.

5.1.3. Partial replicas

In a diverse system, another desirable feature is partial replicas, replicas which store only a fraction of the full replicated data set. For example, a user might synchronize his entire home directory between a collection of machines at work but only synchronize his email and calendar to his PDA or his home computer.

As usual, centrally coordinated systems have no problem supporting this. The individual replicas keep track of which data they have from the server and what version it is. The server never needs to know that the replica is only storing a subset of the data.

Logging systems also have no problem supporting this: the partial replica would just ignore the events in the other replica logs: incorporation of log entries about non-replicated data is a no-op.

As usual, version vector systems have trouble with partial replicas only in the parts concerning the global knowledge algorithms. In order to carry out the algorithms correctly, the other replicas must be aware of whether a given replica is interested in a particular file. With some effort, this can be made to work, but it is awkward: the partial replica would be constantly telling the other replicas that it didn’t care about the proceedings involving this or that file.

The vector time pair algorithms support partial replicas easily. Like in the other algorithms, it is easy for a replica simply to ignore the parts of the data it does not want to see. Unlike in the logging and version vector systems, the vector time pair algorithms do not care about global state, and thus the fact that the replica is partial needn’t be announced to all.

5.1.4. Quick synchronization

In a system where the amount of data is large but there are relatively few changes between synchronizations, it is desirable that synchronizations take time proportional to the amount of changed data rather than the amount of data as a whole.

Centrally coordinated systems could add synchronization times along with an event counter on the central server to implement such quick synchronizations. This would effectively be a centrally coordinated system with a logical time pair instead of a vector time pair, because only one system is generating changes. The centrally coordinated systems of which we are aware do not do this. Instead, they synchronize the entire data set at each time.

Synchronization in logging systems is exactly proportional to the amount of changed data, since synchronization is just reading the new parts of the log.

Version vector systems cannot implement quick synchronization without adding synchronization times (at which point it would be a vector time pair system). Since there is no easy way for a replica to summarize its state for another replica, synchronization has no choice but to examine every piece of data in the system.

Because synchronization in centrally coordinated systems and version vector systems takes time proportional to the amount of data being synchronized, systems built in this manner typically make it easy for the user to specify a partial synchronization. For example, by default CVS synchronizes only the current directory and its children. This default compensates for the slowness of a full synchronization.

As discussed earlier, vector time pairs focus in on the changed files very quickly: one replica can easily summarize its state to the other, and then the second can send only the relevant changes.

5.1.5. Metadata storage requirements

It is important that the synchronization algorithms not have excessive metadata storage requirements, espe-

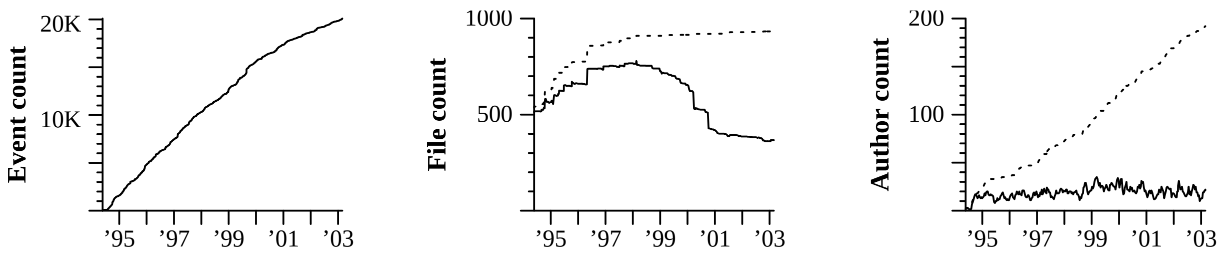


Figure 1. Statistics about the FreeBSD CVS traces used. The *x* axis always plots real time, from May 1994 to February 2003. In the file graph, the dotted line plots the total number of unique files seen until that time, while the solid line plots the number of files currently in the repository. In the author graph, the dotted line plots the total number of unique authors seen until that time, while the solid line plots the number of authors who have made changes to the repository in the last 30 days.

cially if they are to be used on small devices or in large systems.

Centrally coordinated systems have minimal metadata to store. The server must maintain a version number for each existing file, and the clients must remember that version number for each file they have copied. The storage requirements do not change as the number of replicas increases.

A simple logging system keeps the entire log from the beginning of time. This results in increasing storage costs as time moves forward. Old log pieces can be retired after an expiration time (at the risk of excluding slow replicas) or after determining that the log entries have propagated to all replicas (at the cost of a global knowledge algorithm).

Version vector systems have less information than logging systems but it still grows as the number of replicas increases. If there are *R* replicas in the system, each replica must store an *R*-element vector for each file. Thus the metadata storage required per-replica increases linearly in the number of replicas. If global knowledge algorithms or expiration times are used, this storage can be reduced to be proportional to the amount of recent activity, with the same drawbacks that such techniques introduced for logging above.

5.2. Quantitative algorithm evaluation

Version vectors and vector time pairs are similar enough in functionality that that a quantitative comparison of time and space requirements is meaningful. In order to compare them, we built a simulation environment to measure the time and storage costs of the algorithms running synthetic workloads.

5.2.1. Workload

We do not know of any common synchronization benchmarks or traces of file activity on optimistically replicated file systems that include details about when synchronizations took place. Instead, we constructed a synthetic workload using an optimistically replicated but centrally synchronized system: the CVS version control system. Specifically, we started with a CVS trace of activity in the i386-specific part of the FreeBSD kernel source tree (`/usr/src/sys/i386`) from May 1994 through February 2003. For each file creation, deletion, and modification, the trace lists the date and the person who made the change. The trace lists 20,605 events, 938 files, and 192 authors. In any given 30 day window, only a couple dozen authors are actively making changes. Figure 1 describes the data in more detail.

The CVS traces are not a perfect fit for our simulations: while there is information about when users ‘‘checked in’’ changes to the central tree (`cvs commit`) there is no information about when users ‘‘checked out’’ changes made by others (`cvs update`). To use the traces, we must add ‘‘checkout’’ events artificially. We created two workloads from the traces. The first, called `chain`, starts with all initial files on a replica named `start`. When an author makes changes, he first synchronizes with the previous author to make changes. For example, this trace:

```
# time author op (m=modify) file
70082 bde m sys/i386/isa/ipl_funcs.c
79738 iwasaki m sys/i386/apm/apm.c
79738 iwasaki m sys/i386/include/apm_bios.h
85569 peter m sys/i386/eisa/if_fea.c
93477 alc m sys/i386/isa/ipl_funcs.c
```

shows four authors modifying various files in the tree. When we use the trace to generate the `chain` workload, we insert synchronization events as shown here:

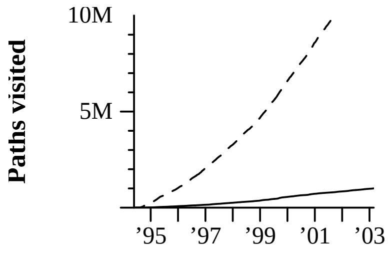
```
  sync from start to bde
70082 bde m sys/i386/isa/ipl_funcs.c
```

```
 sync from bde to iwasaki
79738 iwasaki m sys/i386/apm/apm.c
79738 iwasaki m sys/i386/include/apm_bios.h
 sync from wasaki to peter
85569 peter m sys/i386/eisa/if_fea.c
 sync from peter to alc
93477 alc m sys/i386/isa/ipl_funcs.c
```

This trace has the drawback that inactive replicas are not communicating with active ones, which puts version vector algorithms at a disadvantage. To level the playing field, we created three more workloads p365, p90, and p30 by inserting random extra synchronizations into the chain workload. For each day in the trace, each replica that has been active at some point in the past chooses to synchronize with some random other active replica with probability $1/p$. For example, in the p365 trace, replicas initiate background synchronizations at the rate of about once per year, in addition to synchronizations necessary to follow the trace.

5.2.2. Time requirements

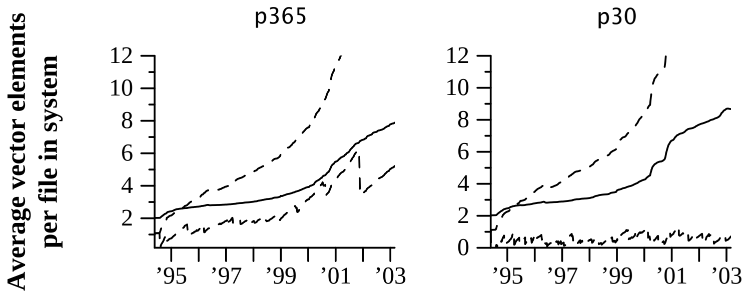
Measuring time as total number of paths visited during the synchronization, vector time pairs perform better than version vectors, as expected. The comparison is similar on each of the four data sets, so we only show the graph for p30 here:



The dashed line is version vectors and the solid line is vector time pairs.

5.2.3. Space requirements

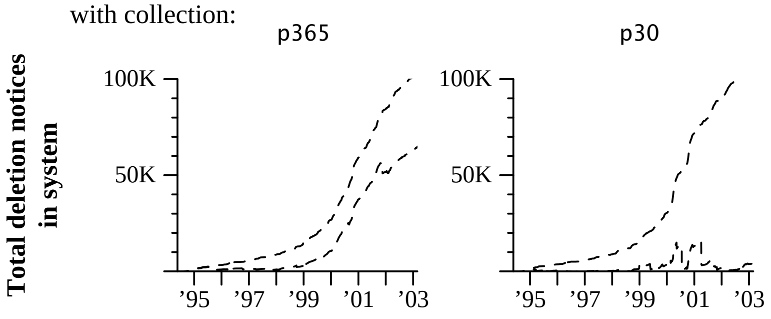
The amount of space required by both basic version vectors and vector time pairs is mostly independent of the number of synchronizations. (Since modification times get wider as the system gets older, increased synchronization does a better job of propagating them, so there is a small storage increase.) However, the version vector compression algorithm, which uses global knowledge to remove unneeded vector elements, depends heavily on the rate of synchronization. We show the storage required for the p365 and p30 traces here:



We plot the number of vector elements stored divided by the total number of files in the system, yielding the average number of vector elements per file. Again the upper dashed line is basic version vectors and the solid line is vector time pairs. The lower dashed line is version vectors with the global knowledge algorithm. With frequent synchronizations, version vectors require about the same storage as vector time pairs. With only infrequent synchronizations, though, vector time pairs do a better job. Of course, as soon as a single replica stalls, the global knowledge algorithm will stop making progress, and the lower line will move back up to follow the upper line.

5.2.4. Deletion notices

The number of deletion notices in a version vector system depends heavily on the synchronization rate. These graphs show the number of deletion notices present in the system for version vectors without collection and with collection:



Again, as soon as one replica falls behind in the version vector system, the bottom line will rise to follow the top line. The line for vector time pairs is the *x* axis: since all synchronizations are full, deletion notices are collected immediately.

5.3. Implementation evaluation

It is difficult to qualitatively compare Tra to other programs. The program closest in spirit to Tra is Rumor, which uses version vectors. We expect Tra to be simpler because it uses vector time pairs rather than version vectors. Unfortunately, there is little that can be said except that Rumor is definitely larger, consisting of 82,000 lines of C++ augmented by 9,000 lines of Perl. Much of the size difference between Tra and Rumor appears to be due to differences in the implementation styles rather than to the algorithms behind the tools. In any case, Rumor is written in a now obsolete dialect of C++ and does not compile on modern systems. We tried running an older Linux distribution with the appropriate compiler but had trouble finding supported hardware.

A quantitative comparison emphasizes software engineering as much or more than the underlying algorithms. Still, to show that the vector time pairs are not causing undue inefficiency, we ran some microbenchmarks to see how Tra compares to Unison and rsync for raw performance. The benchmarks are `copy`: copy a 195MB source tree containing 12,000 files from one replica to a freshly initialized empty replica, `nop`: synchronize the two 195MB trees after they are in sync, `copy1`: propagate a newly-created 6MB file, and `remove1`: propagate the removal of a single file. The tests were run on a 1.2GHz Athlon running FreeBSD 4.5 with 1024MB of memory and an HP NetRaid-4M RAID 0 array. In order to reduce cache effects, we read the contents of every file in the tree before running `copy`, and we walked the whole tree fetching metadata before running the other tests. The results are:

benchmark	Tra	rsync	Unison
`copy`	235	51	33
`nop`	6.5	1.6	2.4
`copy1`	6.2	4.0	3.2
`remove1`	6.1	2.9	3.0

The large time difference between Tra and the other programs is mainly due to overhead in the RPC layer. Since the test was run on a single machine, rsync and Unison kept all operations local while Tra still set up pipes and separate `srv` processes. The current implementation of Tra does not perform aggressive batching of writes: many threads are executing RPCs in parallel, but the small RPC packets (tens of bytes each) are being written one at a time via the `write` system call. We believe that batching the many outgoing RPCs into a small number of larger `write` calls will help this. This hypothesis is supported by the `copy1` results: when copying the 6MB file, we come close to Unison and rsync's performance, because then we are issuing large RPCs and thus being penalized less. The slowness of Tra has to do with our RPC layer rather than the algorithms themselves.

6. Related work

The synchronization of replicated data is an old problem. We view vector time pairs as another step in the development of increasingly flexible solutions.

Even today, many replicated systems declare an owner for each piece of data, and then only that owner can publish changes to the data. This is the approach taken in the domain name system (DNS) [16] and the Netlib mathematical software repository [6], in addition to countless others. The approach works well when there are clearly defined areas of responsibility (as in DNS and Netlib) but grows cumbersome when ownership is more transient.

The next common approach, exemplified by the CVS [2] source control system, is to declare one server the single point of truth for all the data and have owners make changes by going through the server. The server can enforce serialization of changes, making sure that an edited version of a file will not be accepted if it was created starting from an outdated copy. This approach requires that all users interact with a central server, which may not always be easy or even possible, especially if the users are geographically dispersed. The central server is a single point of failure and must be maintained explicitly.

The invention of version vectors by Parker *et*al. [20] enabled the construction of replicated systems without the ‘‘central server’’ restriction. The most common examples of version vector-based systems are the Ficus [7] and Coda [14] distributed file systems. A more recent example is the Pangaea wide-area file system [25]. Vector time pairs offer real benefits to all these cases.

Ficus was very careful about trying to reduce network bandwidth for distant replicas [10]; as shown earlier, vector time pairs allow even more frugal bandwidth use. Using vector time pairs would have eliminated the need for the cumbersome distributed garbage collection algorithms as well [8].

Coda distinguished between servers, which were connected to each other via high-speed links, and clients, which may only be intermittently connected. It used version vectors to track changes made by servers, but treated clients as second-class citizens to keep the number of replicas included in the vectors small. Changes made by a client must be shepherded by a server. This two-level split kept the version vector algorithms efficient. Using vector time pairs would have removed the need for this artificial separation. Coda does not bother to implement the distributed garbage collection algorithms. Instead it is assumed that the central servers are connected well enough to coordinate in a simultaneous conversation.

Because Pangaea uses version vectors, it must keep track of exactly which replicas are in the system. 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 a replica is currently active in the system.

Bayou [21] and Ivy [18] 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.

Fluid replication [3] proposes to create well-connected replicas called WayStations on demand in order to help the performance of less well-connected replicas such as handheld wireless devices. Vector time pairs are a natural match for a fluid replication, since they easily tolerate replicas being created and destroyed dynamically. Depending on the level of consistency desired, fluid replication uses version vectors or logging. Fluid replication's least common ancestor (LCA) algorithm is a heuristic attempting to construct small sets of relevant changes to propagate to less well-connected replica. Vector time pairs should be applicable in the situations that LCA is intended to cover. A detailed comparison of how vector time pairs perform relative to LCA is future work.

User-level synchronizers like Rumor [9] and Unison [1] were discussed in detail earlier. Using vector time pairs enables the various features present in Tra but absent in these.

As mentioned earlier, optimistic replication plays an important role in the use of PDAs and other mobile devices. The SyncML [27] initiative is but one example of the current industry fervor for synchronization. Past examples have included Microsoft Briefcase and the various HotSync PDA programs. The SyncML proposal defers most of the hard bookkeeping by using a central server that tracks the movement of the replicated data. Using the central server makes the bookkeeping very easy for the mobile devices, avoiding the problems that plague version vectors. Using vector time pairs would be another way to avoid these problems, without reintroducing the problems associated with a central server.

7. Future work

In our minds, the most significant shortcoming of this work is the lack of a formal proof that the algorithms are correct. We have tested our implementations on a large test suite that includes all the examples in this paper. It would be quite another thing to prove that the algorithms given here behave exactly as desired. As mentioned in section 2, even formalizing the problem statement is difficult, and a number of mostly equivalent formalizations have been proposed. The only synchronizer we know of with a given proof of correctness is Unison, but Unison's approach is restricted to two replicas, which significantly simplifies both their algorithms and their proofs. Precise formalization of synchronization is still an important open problem.

The space required to store the modification times for directories is higher than we would like. We are exploring ways to reduce this storage safely, but without global knowledge of the system.

The command-line interface to `tra` is functional, but a richer interface would be convenient for interactive use. We have only just begun to consider how to build a graphical interface, which would allow fine control over the synchronization process. Although the Unison interface is definitely a step in the right direction, we believe that the question of what makes a good interface to a file synchronizer is still open.

We plan to collect traces of synchronization activity by Tra users in order to better understand what people expect from file synchronizers and to check whether our expected usage patterns (in particular, the regular total synchronizations) hold in practice. Such traces would also be useful for future algorithm designers.

8. Conclusions

As mobile devices proliferate, optimistic replication will serve an increasingly central role in our day-to-day computing environments. We have presented *vector time pairs* as a new method for tracking optimistically replicated data. The fundamental 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.

We have also described an implementation of vector time pairs in a real application, a user-level file system synchronizer. The Tra source code and simulator are available via anonymous CVS and a web interface. See

url removed for anonymity

for more information.

9. Acknowledgements

Removed for anonymity.

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] Brian Berliner, ‘‘CVS II: Parallelizing Software Development’’, *Proceedings of the USENIX Winter 1990 Technical Conference*, pp. 341−352, 1990.

[3] Landon P. Cox and Brian D. Noble, ‘‘Fast Reconciliations in Fluid Replications’’, International Conference on Distributed Computer Systems (ICDCS), April 2001.

[4] C. Fidge, ‘‘Logical time in distributed computing systems’’, *IEEE Computer*, Vol. 24, No. 8 (August 1991), pp. 28−33.

[5] Jim Gray, Pat Helland, Patrick O’Neil, and Dennis Shasha. ‘‘The Dangers of Replication and a Solution’’, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, 173−182.

[6] Eric Grosse. ‘‘Repository Mirroring’’, *ACM Transactions on Mathematical Software*, Vol. 21, No. 1, pp. 89−97 (1995).

[7] Richard Guy, ‘‘Ficus: A Very Large Scale Reliable Distributed File System’’, Ph.D. thesis, UCLA report CSD-910018, 1991.

[8] Richard Guy, Gerald Popek, and Thomas W. Page, Jr. ‘‘Consistency Algorithms for Optimistic Replication’’, *Proceedings of the First International Conference on Network Protocols*, IEEE, October 1993.

[9] Richard Guy, Peter Reiher, 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.

[10] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek, ‘‘Primarily Disconnected Operation: Experiences with Ficus’’, *Proceedings of the Second Workshop on Management of Replicated Data*, pp. 2−5, November 1992.

[11] Val Henson, ‘‘An Analysis of Compare-by-Hash’’, *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003, pp. 13−18.

[12] C. A. R. Hoare, ‘‘Communicating Sequential Processes,’’ *Communications of the ACM* 21(8) (August 1978), 666−677.

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

[14] 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).

[15] Friedemann Mattern, ‘‘Virtual Time and Global States of Distributed Systems’’ , *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B. V., 1989, pp. 215−226.

[16] P. Mockapetris and K. Dunlap, ‘‘Development of the Domain Name System’’ , *Proceedings of the ACM SIGCOMM Symposium*, Stanford, CA., 1988.

[17] Athicha Muthitachaeroen, Benjie Chen, and David Mazières, ‘‘A Low-bandwidth Network File System’’, *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP ’01)*, Chateau Lake Louise, Banff, Canada, October 2001, 174−187.

[18] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. ‘‘Ivy: A Read/Write Peer-to-Peer File System’’, *Proceedings of the Fifth Symposium on Operating Systems Design and* Implementation (OSDI)" , Boston, MA, December 2002.

[19] Brian D. Noble and M. Satyanarayanan, ‘‘An Empirical Study of a Highly Available File System’’, *Proceedings of the 1994 ACM SIGMETRICS Conference*, Nashville, TN, 1994.

[20] 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.

[21] Karin Peterson, Mike J. Spreitzer, Douglsa B. Terry, Marvin M. Theimer, and Alan J. Demers. ‘‘Flexible Update Propagation for Weakly Consistent Replication’’, *Proceedings of the 16th Annual Symposium on Operating Systems Principles (SOSP-16)*, Saint Malo, France, October 5-8, 1997, 288−301.

[22] Benjamin Pierce and Jérôme Vouillon, ‘‘Specifying a File Synchronizer’’, draft as of 2002. `http://citeseer.nj.nec.com/pierce02specifying.html`

[23] Rob Pike, ‘‘A Concurrent Window System’’, *Computing Systems*, 2(2) (Spring 1989), 133−153.

[24] Norman Ramsey and Elod Csirmaz, ‘‘An Algebraic Approach to File Synchronization’’, *Foundations of Software Engineering*, (September 2001), pp. 175−185.

[25] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam, ‘‘Taming aggressive replication in the Pangaea wide-area file system’’ , *Proceedings* of Implementation (OSDI)" , Boston, MA, December 2002.

[26] 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.

[27] SyncML Home Page. `http://www.syncml.org/`. Current as of September 22, 2003.

[28] Andrew Tridgell and Paul Mackerras. *The rsync algorithm*, Technical report TR-CS-96-05, Australian National University, 1995.

[29] Matt Welsh, David Culler, and Eric Brewer. ‘‘SEDA: An Architecture for Well-Conditioned, Scalable Internet Services’’, *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, October, 2001.

[30] Tatu Ylonen. ‘‘SSH—Secure Login Connections Over the Internet’’ , *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA (July 1996), pp. 37−42.