Vector Time Pairs, Version Vectors, and Version Stamps

Russ Cox

rsc@mit.edu

May 28, 2001
PDOS Group Meeting

`http://pdos/~rsc/version.pdf`

Outline

OSDI submission

- The file synchronization problem
- Vector time pairs to the rescue
- The death of version vectors

Cool related tangent

- Version stamps

File Synchronization

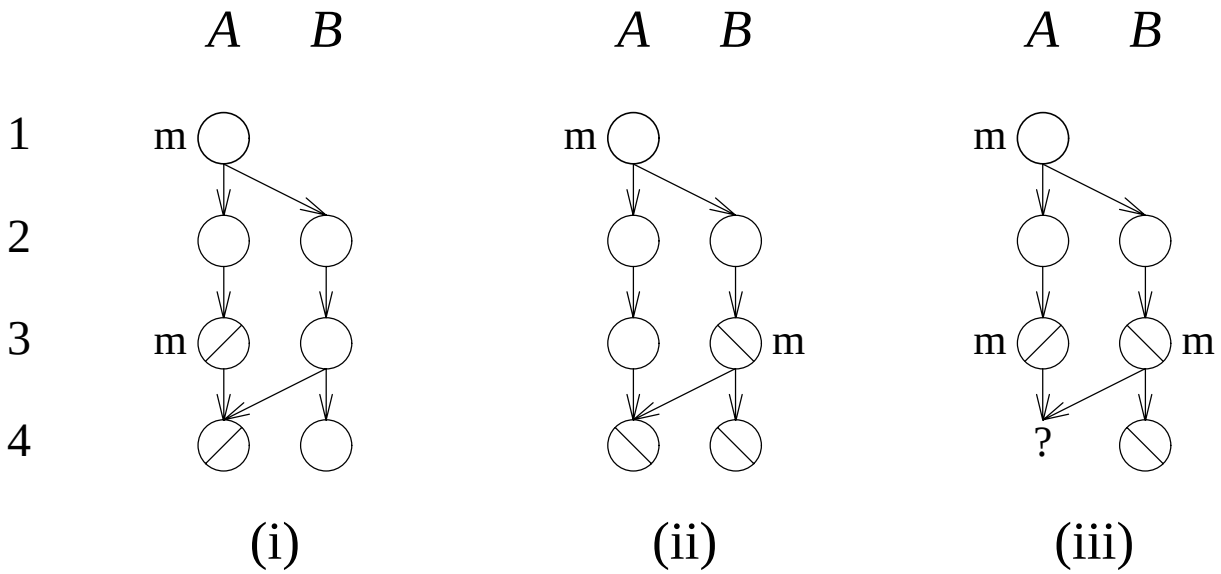
File history:

created on A at A-time 2
modified on B at B-time 5
modified on C at C-time 3
deleted on B at B-time 12

Okay to replace v_1 with v_2 if v_1's history is a prefix of v_2's.

Goal: keep enough data to answer whether one history is a prefix of another.

Examples



(i) *A* makes a change, sync from *B* to *A* does nothing

(ii) *B* makes a change, sync from *B* to *A* propagates change

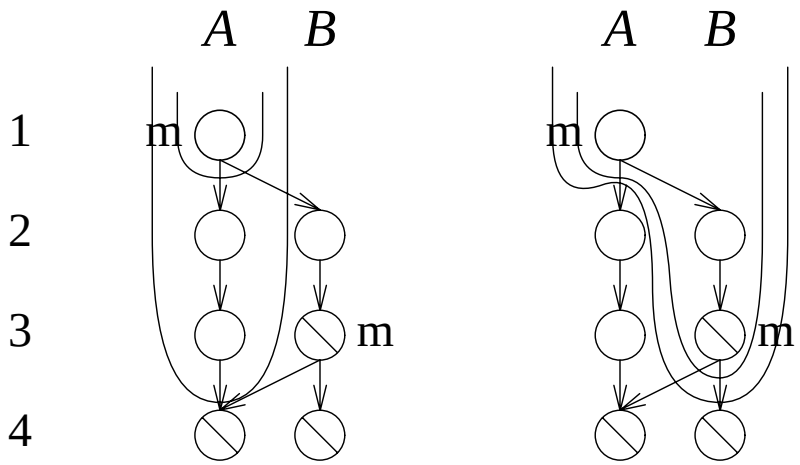
(iii) *A* and *B* make changes, sync detects conflict

Vector time pairs

Keep two vector times:

One tracks synchronization history.

One tracks modification history.



$\text{mod}_A = (A{:}1) \quad \text{mod}_B = (A{:}1\ B{:}3)$

$\text{sync}_A = (A{:}3) \quad \text{sync}_B = (A{:}1\ B{:}3)$

B can replace *A* because mod_A is included in $sync_B$:

$mod_A \leqq sync_B$.

Could just look at mod_B and mod_A:

Don’t go there.

Vector time pairs: sync process pruning

Define mod time of a directory to be *max* of mod times of its children.

Define sync time of a directory to be *min* of sync times of its children.

If $mod_A \leqq sync_B$ for directory *d*, don't need to recur into *d* when propagating changes from *A* to *B*.

Vector time pairs: mod time pruning

Modifications to a file are sequenced by synchronization, which only does entire-file copies.

If we only keep last modification, the outcome of the $mod \leqq sync$ comparison doesn't change.

Modification times on files can be stored as singletons rather than full vectors.

- when using R replicas, $O(R)$ per file becomes $O(\log R)$ per file.

Modifications on directories aren't sequenced.

- one replica might see create *f*, remove *g*.
  other might hear about changes as remove *g*, create *f*.

Singleton trick only works for files being copied in full.

- Might merge mail spool files automatically.
  Modifications wouldn't be sequenced anymore.

Vector time pairs: sync time pruning

If we sync the entire file system each time we run the synchronizer, every file and directory will have the same sync vector time.

Can store a list of sync times and use pointers into list. If there are S different sync times, $O(R)$ becomes $O(\log S)$.

Better trick: we know $sync(d) \leqq sync(d/f)$. Remove unnecessary elements as you recur down the tree to store f's sync time, and put them back as you fetch it.

If $sync(d) = (A{:}1\ \ B{:}2)$ and $sync(d/f) = (A{:}1\ \ B{:}3)$, only need to store $(B{:}3)$ for d/f.

Much easier than maintaining the sync list.

Space is about the same.

Vector time pairs: deletion notice removal

Deletion notices propagate like other file changes.

Want to remove deletion notices once they're not needed anymore.

It's easy; trust me.

All smoke and mirrors in the bookkeeping: synchronization algorithm never knows they're gone.

Similar to sync time pruning trick.

Version vectors

Keep only modification time.

Use it as both modification and synchronization time.

Not as expressive.

Encode conflict resolutions

Defer until later.

Version vectors

Doesn’t allow modification time pruning trick.

- Takes up more space.

Doesn’t allow synchronization time pruning trick.

- Still takes up more space.

Doesn’t allow file system sync pruning.

- Takes time, bandwidth during synchronization.

Requires garbage collection for deletion tracking

- Takes up time during programming, provides home for bugs.

Cannot encode copying conflict resolutions.

- Inflexibility in user interface.

Ivy

Presents a view of a file system from multiple sort-of-independent logs of file system operations:

create, read, write, remove, rename, chmod

Uses ‘‘version vectors’’ to decide how to interleave the logs and to check for conflicting writes.

Example:

A’s log is up to seqnum 7.
A knows about *B*’s log up to seqnum 3.
A writes a new log record with seqnum 8.
In the Ivy paper, record has version vector $(A{:}7\ \ B{:}3)$.

In the presentation here:

Record has *modification time* $A{:}8$.

Record has *synchronization time* $(A{:}7\ \ B{:}3)$.

Actually, the synchronization time is $(A{:}8\ \ B{:}3)$, but this discrepancy is handled in implementation by treating equal vectors on different machines as incomparable.

Ivy, ii

Ivy gets encoding of some conflict resolutions for free.

Suppose *A* and *B* partition and both modify a file.

> *A* now has a file with modification time $A{:}9$ and synchronization time $(A{:}9\ B{:}3)$.
>
> *B* now has a file with modification time $B{:}6$ and synchronization time $(A{:}4\ B{:}6)$.
>
> Partition heals, *A* notices conflict, writes a new file.
>
> *A* now has a file with modification time $A{:}11$ and synchronization time $(A{:}11\ B{:}6)$.
>
> *B* will accept *A*'s write as newer than *B*'s current version.

Can't encode ''*A* says keep *A*'s file, toss *B*'s.''

> would need modification time $A{:}9$, but can't reuse sequence number.

Version vectors are dead, i

In log-based systems (Ivy, Bayou), data associated with a given local time is atomic (e.g., one log record) and updates are sequenced (e.g., don't read log record $n + 1$ before n).

The natural use of version vectors treats log sequence number as modification time, version vector as synchronization time.

Still can't encode simple conflict resolutions.

Version vectors are dead, ii

In non-log-based systems (Coda, Ficus, Rumor, Tra), many changes might have the same local time and updates aren't sequenced.

Can lump all updates into large atomic, sequenced units.

- Now we have a log again, with possibly huge records.
- Likely to be clumsy, inflexible.
- Keeps benefits of having synchronization time.
- Still can't encode simple conflict resolutions.

Can give up version vector as synchronization time.

- More natural, more flexible than forced log.
- Lose benefits of having synchronization time.

Just use vector time pairs.

- `http://pdos/~rsc/tra/osdi.pdf`

Context Switch!

There's a fair amount of overhead in using something like Ivy or Tra or Coda or ...

For little things, it would be nice to have a `cp` and `mv` that watched out for you:

```
g% echo hello world >a
g% vcp a b
g% echo hello again >>a
g% vcp a b
g%
```

but

```
g% echo hello world >a
g% vcp a b
g% echo goodbye world >>b
g% vcp a b
vcp: will not overwrite changes to b \
    (suggest vcp b a; or use –f to force)
g% echo hello again >>a
g% vcp a b
vcp: a and b are incomparable (use –f to force)
g%
```

Version stamps

Described in ‘‘Version Stamps — Decentralized Version Vectors,’’ Paulo Sérgio Almeida, Carlos Baquero, Victor Fonte, Universidade do Minho, Portugal.

Search ResearchIndex or Google.

They built a suite of command-line tools called `panasync`, presented at 2000 European SIGOPS Workshop.

6500 lines of C++

I built the `vcp` and `vmv` in the examples, to make sure I understood it.

425 lines of C for version stamp manipulation (50% I/O)

230 lines of shell script (90% error checking)

`~rsc/bin/sh/vcp,vmv`

Version stamps, ii

Managing version vectors/vector times when machines come and go with great frequency is a pain.

- Names must be unique.
- Need to clean up dead names.
- Name operations too frequent to leave to user.

Completely different approach

- Observe we only need to compare versions that exist simultaneously.
- Toss all other information.

Operations are join two copies, fork one copy into two.

- `cp f1 new-f2` is fork
- `cp f1 f2` is join followed by fork
- `mv f1 new-f2` is no-op; just rename the file.
- `mv f1 f2` is join.

Version stamps, iii

Keep two ids:

Who knows about this version of the file.

Who we are.

Ids are half-open power-of-two-sized subsets of $[0,1)$.
$[1/4-1/2)+[1/2-1)$

Aka boolean-valued function on
unit interval.

Aka boolean-valued function on
countable number of boolean variables. $01+1$

Aka sets of binary strings, reduced by rule $x0+x1\equiv x$.
$00+01+1\equiv 0+1\equiv e$

Version stamps, iv

Keep two ids [*vers*|*self*].

vers: the replicas that know about this version of the file.

self: our id.

An update changes [*vers*|*self*] into [*self*|*self*].

Now only we know about this version of the file.

Multiple updates without intervening joins/forks are equivalent to a single update.

Version stamps, v

$[vers \mid self]$ forks into $[vers \mid self0]$ and $[vers \mid self1]$.

Split *self* into left and right half.

Could use any disjoint fork of *self* into two ids.

$[vers \mid self_A]$ and $[vers \mid self_B]$ join into $[vers \mid self_A + self_B]$, with appropriate reductions.

$[0 \mid 000]$ and $[0 \mid 001 + 01]$ join to
$[0 \mid 000 + 001 + 01] = [0 \mid 0]$.

Safe to copy $vers_A$ over $vers_B$ if $vers_A \subseteq vers_B$:

0 can replace *e*, $001 + 111$ can replace $00 + 11$.

vers is copied with data: $[0 \mid 001]$ joined and copied over $[e \mid 100]$ yields $[0 \mid 001 + 100]$.

vers reduces with *self*:

$[* \mid 01 + 1]$ and $[1 \mid 00]$ join to $[1 \mid 00 + 01 + 1] = [* \mid *]$.

Summary

Vector time pairs are the way to go.

Version vectors are dead.

Version stamps are cool.

Not sure what to do with them.