# 6.824 2006 Lecture 12: Vector Timestamps

```
Topic: Vector Timestamps / Version Vectors
  Usually used for file synchronizers
  Treadmarks use them also

so far we've talked about enforcing some order of operations
  either implicitly because they go thru file server
  or because of e.g. Ivy DSM sequential consistency

Nice semantics, but sometimes awkward
  requires lots of chatter, w/ file server or other clients

can you sensibly operate out of your cache
  without seeing other people's updates?

example: disconnected operation with laptops
  source-code repository, cooperating programmers
  each programmer's laptop has a complete copy of the files
    want to be able to modify files while disconnected
  programmers synchronize files pair-wise: "file synchronizer"
  how to decide which files to copy, and in which direction?
    non-conflicting changes: take latest version
    conflicting changes: report an error, ask humans to resolve
  "optimistic replication"

Example 1:
  Just focus on a modifications to a single file "f"
  H1: w(f)1 ->H2       ->H3
  H2:            w(f)2
  H3:                       ->H2
  What is the right thing to do?
  Is it enough to simply take file with latest modification time?
  Yes in this case, as long as you carry them along correctly.
    I.e. H3 remembers mtime assigned by H1, not mtime of sync.

Example 2:
  H1: w(f)1 ->H2 w(f)2
  H2:                  w(f)0 ->H1
  H2's mtime will be bigger.
  Should the file synchronizer use "0" and discard "2"?
    No! They were conflicting changes. We need to detect this case.
    Modification times are not enough by themselves

What is the principle here?
  We're not *enforcing* a sequential update history.
    No locks, no Ivy-style owners.
  But *if* updates were actually sequential,
    we want to detect it and carry along most recent copy.
    This is where the "optimism" comes in.
  And if updates were not sequential,
    we want to detect the problem (after the fact).

So what is the problem with using wall-clock time?
  Certainly if T1 < T2, T1 does not supersede T2.
  *But* if T2 > T1, that doesn't imply that T2 supersedes T1.
  Ordered in time doesn't mean everyone agrees on the order.
```

How can we decide whether we're looking at a sequential history?
  We could record each file's entire modification history.
  List of hostname/localtime pairs.
  And carry history along when synchronizing between hosts.
  For example 1:   H2: H1/T1,H2/T2   H3: H1/T1
  For example 2:   H1: H1/T1,H1/T2   H2: H1/T1,H2/T3
  Then its easy to decide if version X supersedes version Y:
    If Y's history is a prefix of X's history.
    This exactly captures our desire for an *ordered* sequence of updates.
  Note we're not comparing times, so this works w/ out-of-sync host clocks.

Why is complete history not such a great solution?
  The histories grow without bound if we modify a file a lot.

Can we compress the histories?
  Proposal: Vector Timestamps.
  Summarize a history w/ each host's highest time.
  So throw away all but the last history record for each host.
  Last TS from each host preserves information essential to
    detecting conflicting updates:
    Had I seen your latest change when I made my change?
  Only care about latest change.

A vector timestamp is a vector of times, one entry per host.
  The times are (as in the histories) local wall-clock times.
    Though they could be any monotonic count: local version number.
  If a and b are vector time-stamps:
  a = b if they agree at every element.
  a <= b if a[i] <= b[i] for every i.
  a >= b
  a || b if a[i] < b[i], a[j] > b[j], for some i,j.
    i.e. a and b conflict

If one history was a prefix of the other, then one vector timestamp
  will be less than the other.

If one history is not a prefix of the other, then (at least by example)
  VTs will not be comparable.

So now we can perform file synchronization.
  Laptops carry along a version vector with each file, rather
    than just the single timestamp of last change.
  When a pair of laptops syncs a file, they accept the file with
    the higher vector timestamp.
  And signal an error if the VTs conflict.
  Illustrate with the two examples.

What if there *are* conflicting updates?
  VTs can detect them, but then what?
  Depends on the application.
  Easy: mailbox file with distinct messages, just merge.
  Medium: changes to different lines of a C source file.
  Hard: changes to the same line of C source.
  Reconciliation must be done manually for the hard cases.

CVS keeps extra information, equiv to complete history.
So it can generate diffs from common ancestor.
Much better than diffing the two final versions against each other.
Which is newer? Or did they both change?

What about file deletion?
Deletes have to have VTs so you can detect delete/write conflicts.
Could treat as a special kind of write.
But then you have to remember deletion timestamp indefinitely.
In case some host out there hasn't heard about it.
Can we garbage collect deletes?

We've used VTs to *detect* violations of sequential consistency.
We can also use VTs to help *ensure* consistency.
TreadMarks uses a similar technique...