```
6.824 2015 Lecture 11: Optimism, Causality, Vector Timestamps

Consistency so far
  Concurrency forces us to to think about meaning of reads/writes
    e.g. if P1 has seen P2's write, has P3 seen P2's write too?
  Sequential consistency: everyone sees same read/write order (IVY)
  Release consistency: everyone sees writes in unlock order (TreadMarks)

Sequential and release consistency require central component:
  must ask before each operation to ensure ordering
  IVY: read faults and write faults -> ask page's manager
  TreadMarks: acquire and release -> ask lock manager

Central component can be undesirable
  Bottleneck, single point of failure, requires network connectivity
  Can we get rid of it?

Starting a new class of distributed systems:
  No central component
  Support disconnected or partially connected operation
  Use optimistic approach (always allow, clean up later)
  Provide eventual and causal consistency

Example -- peer-to-peer chat
  We each have a computer attached to internet
  Can send at any time (optimistic), no central ordering
  Recv msg -> add to end of chat window

Do we care about message ordering for chat?
  Network may deliver in different order at different participants
  Suppose Alice is auctioning something
  Joe: $10
  Fred: $20
  Alice: the high bid is $20
  Maybe Sam sees:
    Joe: $10
    Alice: the high bid is $20

What went wrong in this example?
  Alice "computed" her message based on certain inputs
  Only makes sense to Sam if he has seen those inputs too
  Why not have Alice's message describe what Alice had seen?
    Could fix Sam's order w/o requiring central component

Definition: x causally precedes y
  x precedes y if:
    M0 does x, then M0 does y
    M0 does x, M0 sends msg to M1, M1 does y
  transitive closure
  x and y could be writes, or msgs, or file versions
  also "y causally depends on x"

Definition: causal consistency
  if x causally precedes y, everyone sees x before y

Pro: we can implement w/o central component
Con: not a total order -- some events have no relative order

Slow implementation of causal consistency
  Unique ID for every msg
```

```
    Node keeps set of all msg IDs received -- "history"
    When sending m, send current history set, too
    Receiver delays incoming msg m until has received everything in m's set

  History sets will grow huge -- can we abbreviate?
    Each node numbers its msgs 1, 2, 3, &c
    Each message carries a vector:
      [ 4, 3, 7 ]
      Means sender had seen through msg 4 from H0, 3 from H1, 7 from H2
    This notation doesn't grow over time, unlike history sets
    Called a Vector Timestamp or Version Vector
      VTi[j] = x means host i has seen all of j's messages through x

  VT comparisons
    to answer "should msg A be displayed before msg B?"
    a < b if:
      forall i: a[i] <= b[i]  AND  exists j: a[j] < b[j]
      i.e. a summarizes a proper prefix of b
      i.e. a causally precedes b
    a || b if:
      exists i,j: a[i] < b[i] and a[j] > b[j]
      i.e. neither summarizes a prefix of the other
      i.e. neither causally precedes the other

  Many systems use VT variants, but for somewhat different purposes
    TreadMarks, Ficus, Bayou, Dynamo, &c
    "I've seen everyone's updates up to this point"
    "event x preceded event y"
    VTs are compact and decentralized

  Example use of VTs: CBCAST -- "causal broadcast" protocol
    General-purpose ordering protocol, e.g. for peer-to-peer chat
    From Cornell Isis research project
    Key property:
      Delivers messages to individual nodes in causal order
      If a causally precedes b, CBCAST delivers a first
    [diagram: node, msg buf, VC, chat app]
    Each node keeps a local vector clock, VC
      VCi[j] = k means app at node i has seen all msgs from j up through k
    send(m) at node i:
      VCi[i] += 1
      broadcast(m, i, VCi)
    on receipt of broadcast(m, i, v) at node j:
      j's CBCAST library buffers the message
      release to application only when:
        VCj >= v, except v[i] = VCj[i] + 1
        i.e. node j has seen every msg that causally precedes m
      VCj[i] = v[i]
        so sends will reflect receipt of m

  Example:
    All VCs start <0,0,0>
    M0 sends <1,0,0>
    M1 receives <1,0,0>
    M1 sends <1,1,0>
    M2 receives <1,1,0> -- must delay
    M2 receives <1,0,0> -- can process, unblocks other msg

  Why fast?
    No central manager, no global order
    If no causal dependencies, CBCAST doesn't delay messages
```

```
    Example:
      M0 sends <1,0>
      M1 sends <0,1>
      Receivers are allowed to deliver in either order

  Causal consistency still allows more surprises than sequential
    Only causally related events are ordered
      So nodes can disagree on order of non-dependent events
    Sam can still see:
      Joe: $10
      Fred: $20
      Bob: $30
      Alice: the high bid is $20
    Causal consistency only says Alice's msg will be delivered after
      all msgs she had seen when she sent it
    *Not* that it will be delivered before all msgs she hadn't seen

  TreadMarks uses VTs to order writes to same variable by different machines:
    M0: a1 x=1 r1  a2 y=9 r2
    M1:              a1 x=2 r1
    M2:                           a1 a2 z = x + y r2 r1
    Could M2 hear x=2 from M1, then x=1 from M0?
      How does M2 know what to do?

  VTs are often used for optimistic updating of replicated data
    Everyone has a copy, anyone can write
    Don't want IVY-style MGR or locking: network delays, failures
    Need to sync replicas, accept only "newest" data, detect conflicts
    File sync (Ficus, Coda, Rumor)
    Distributed DBs (Amazon Dynamo, Voldemort, Riak)

  File synchronization -- e.g. Ficus
    Multiple computers have a copy of all files
    They can't always talk to each other ("disconnected operation")
    User can always modify local copy of file -- optimistic
    Merge changes later

  Scenario:
    user has files replicated at work, at home, on laptop
    hosts may be disconnected: no WiFi, turned off
    edit on H1 for a while, sync changes to H2
    edit on H2, sync changes to H3
    edit on H3, sync to H1

  Goa: eventual consistency
      i.e. replicas often differ, but converge after enough syncs

  Goal: no lost updates
    Only OK for sync to copy version v2 over version v1 if
      v2 includes all updates that are in v1.

  Example 1:
    Focus on a single file
    H1: f=1 ->H2       ->H3
    H2:          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: f=1 ->H2 f=2
  H2:                  f=0 ->H1
  H2's mtime will be bigger.

Should the file synchronizer use "0" and discard "2"?
  After all, f=0 was a later update than f=2.
  No: that would violate the No Lost Updates goal.
  E.g. you and I both made changes to the file.

What do do if concurrent updates to the same data?
  Sync should somehow detect this "conflict" situation.
  Sometimes conflicts can be automatically resolved once detected.
  Sometimes the user has to figure out how to resolve.
  Conflicts are a necessary consequence of optimistic writes.

How to decide if version v2 contains all of v1's updates?
  I.e. when no updates would be lost by replacing v1 with v2.
  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.

We can use VTs to compress these histories!
  Each host remembers a VT per file
  Number each host's writes to a file (or assign wall-clock times)
  Just remember # of last write from each host
  VT[i]=x => file version includes all of host i's updates through #x

VTs for Example 1:
  After H1's change: v1=<1,0,0>
  After H2's change: v2=<1,1,0>
  v1 < v2, so H2 ignores H3's copy (no conflict since <)
  v2 > v1, so H1/H3 would accept H2's copy (again no conflict)

VTs for Example 2:
  After H1's first change: v1=<1,0,0>
  After H1's second change: v2=<2,0,0>
  After H2's change: v3=<1,1,0>
  v3 neither < nor > v1
    thus neither has seen all the other's updates
    thus there's a conflict

What if there *are* conflicting updates?
  VTs can detect them, but then what?
  Depends on the application.
  Easy: mailbox file with distinct immutable messages, just union.
  Medium: changes to different lines of a C source file (diff+patch).
  Hard: changes to the same line of C source.
  Reconciliation must be done manually for the hard cases.
  Today's paper is all about reconciling conflicts

How to think about VTs for file synchronization?
  They detect whether there was a serial order of versions
  I.e. when I modified the file, had I already seen your modification?
    If yes, no conflict
    If no, conflict
  Or:
```

```
    A VT summarizes a file's complete version history
    There's no conflict if your version is a prefix of my version

What about file deletion?
  Can H1 just forget a file's VT if it deletes the file?
    No: when H1 syncs w/ H2, it will look like H2 has a new file.
  H1 must remember deleted files' VTs.
  Treat delete like a file modification.
    H1: f=1  ->H2
    H2:           del  ->H1
    second sync sees H1:<1,0> H2<1,1>, so delete wins at H1
  There can be delete/write conflicts
    H1: f=1  ->H2  f=2
    H2:            del  ->H1
    H1:<2,0> vs H2:<1,1> -- conflict
    Is it OK to delete at H1?

Can a node ever discard a deleted file's VT?
  Similar danger: discard VT, then sync w/ node that didn't discard.

How Ficus forgets about a deleted file's VT
  H1: del f ->all seen f->all  done f->all   forget f
  H2:             seen f->all  done f->all   forget f
  H3:             seen f->all  done f->all   forget f
      |-- phase 1 -----------|-- phase 2 --|
  Phase 1: accumulate set of nodes that have seen delete
    terminates when == complete set of nodes
  Phase 2: accumulate set of nodes that have completed Phase 1
    when == all nodes, can totally forget the file
  If H1 then syncs against H2,
    H2 must be in Phase 2, or completed Phase 2
    if in Phase 2, H2 knows H1 once saw the delete, so need not tell H1 abt file
    if H2 has completed Phase 2, it doesn't know about the file either

A classic problem with VTs:
  Many hosts -> big VTs
  Easy for VT to be bigger than the data!
  No very satisfying solution

Many file synchronizers don't use VTs -- e.g. Unison, rsync
  File modification times are enough if only two parties, or star
  Need to remember time of last sync, time of modification
    Conflict if both nodes' modification times are greater than last sync
  VTs needed if you want any-to-any sync with > 2 hosts

Summary
  Replication + optimistic updates for speed, high availability
  Causal consistency yields sane order of optimistic updates (CBCAST)
  Vector Timestamps detect conflicting updates
    by compactly summarizing update histories
```