Jenkins, if I want another yes-man, I’ll build one!

Versioning and Eventual Consistency

COS 461: Computer Networks
Spring 2011

Mike Freedman
http://www.cs.princeton.edu/courses/archive/spring11/cos461/
Ordering

• TCP sequence numbers uniquely order packets
  – One writer (sender) sets the sequence number
  – Reader (receiver) orders by seq number

• But recall distributed storage: may be more than one writer
  – One solution: If single server sees all the writes, can locally assign order in the order received, not sent
  – Recall partitioned storage: What about ordering writes handled by different servers?
Time and distributed systems

• With multiple events, what happens first?

A shoots B

B dies
Time and distributed systems

• With multiple events, what happens first?

A dies

B shoots A
Time and distributed systems

• With multiple events, what happens first?

A shoots B
A dies

B shoots A
B dies
Just use time stamps?

• Need synchronized clocks

• Clock synch via a time server
Cristian’s Algorithm

- Uses a *time server* to synchronize clocks
- Time server keeps the reference time
- Clients ask server for time and adjust their local clock, based on the response
  - But different network latency → clock skew?
- Correct for this? For links with symmetrical latency:

\[
RTT = T_{\text{resp received}} - T_{\text{req sent}}
\]

\[
T_{\text{new local}} = T_{\text{server}} + (RTT / 2)
\]

\[
\text{Error}_{\text{clock}} = T_{\text{new local}} - T_{\text{old local}}
\]
Is this sufficient?

• Server latency due to load?
  – If can measure $T_{\text{new local}} = T_{\text{server}} + (\text{RTT} + \text{lag} / 2)$

• But what about asymmetric latency?
  – RTT / 2 not sufficient!

• What do we need to measure RTT?
  – Requires no clock drift!

• What about “almost” concurrent events?
  – Clocks have micro/milli-second precision
Events and Histories

• Processes execute sequences of events

• Events can be of 3 types:
  – local, send, and receive

• The local history $h_p$ of process $p$ is the sequence of events executed by process
Ordering events

• Observation 1:
  – Events in a local history are totally ordered

• Observation 2:
  – For every message $m$, send($m$) precedes receive($m$)
Happens-Before (Lamport [1978])

• Relative time? Define \( \text{Happens-Before} (\rightarrow) \):
  
  – On the same process: \( a \rightarrow b \), if \( \text{time}(a) < \text{time}(b) \)
  
  – If \( p_1 \) sends \( m \) to \( p_2 \): \( \text{send}(m) \rightarrow \text{receive}(m) \)
  
  – Transitivity: If \( a \rightarrow b \) and \( b \rightarrow c \) then \( a \rightarrow c \)

• Lamport Algorithm establishes partial ordering:
  
  – All processes use counter (clock) with initial value of 0
  
  – Counter incremented / assigned to each event as timestamp
  
  – A \text{send} (msg) event carries its timestamp
  
  – For \text{receive} (msg) event, counter is updated by
    \[
    \text{max} (\text{receiver-counter}, \text{message-timestamp}) + 1
    \]
Events Occurring at Three Processes
Lamport Timestamps

Physical time

p1

1 2
a b m1

p2

3 4

p3

1 5
e f m2
Lamport Logical Time

Physical Time

Host 1
Host 2
Host 3
Host 4
Lamport Logical Time

Logically concurrent events!
Vector Logical Clocks

• With Lamport Logical Time
  – e precedes f ⇒ \( \text{timestamp}(e) < \text{timestamp}(f) \), but
  – \( \text{timestamp}(e) < \text{timestamp}(f) \) ⇒ e precedes f
Vector Logical Clocks

• With Lamport Logical Time
  – e precedes f ⇒ \( \text{timestamp}(e) < \text{timestamp}(f) \), but
  – timestamp(e) < timestamp(f) ⇒ e precedes f

• Vector Logical time guarantees this:
  – All hosts use a vector of counters (logical clocks),
    \( i^{th} \) element is the clock value for host i, initially 0
  – Each host i, increments the \( i^{th} \) element of its vector upon an event, assigns the vector to the event.
  – A \text{send}(\text{msg})\ event carries vector timestamp
  – For \text{receive}(\text{msg})\ event,

\[
V_{\text{receiver}}[j] = \begin{cases} 
\max (V_{\text{receiver}}[j], V_{\text{msg}}[j]), & \text{if } j \text{ is not self} \\
V_{\text{receiver}}[j] + 1, & \text{otherwise}
\end{cases}
\]
Vector Timestamps

- \( p_1 \): (1,0,0) to (2,0,0) via paths: a (time 1) → b (time 2) → m_1 (time 2,1,0) → c (time 2,2,0) → d (time 2,2,1) → m_2 (time 2,2,2) → f
- \( p_2 \): (1,0,0) to (2,0,0) via paths: a (time 1) → b (time 2) → m_1 (time 2,1,0) → c (time 2,2,0) → d (time 2,2,1) → m_2 (time 2,2,2) → f
- \( p_3 \): (0,0,1) to (2,2,2) via paths: e (time 0) → m_2 (time 2,2,2) → f

Physical time progression:
- \( m_1 \) and \( m_2 \) represent the movement of elements in the vector space over time.
Vector Logical Time

\[ V_{\text{receiver}}[j] = \begin{cases} \text{Max} \ (V_{\text{receiver}}[j], V_{\text{msg}}[j]), & \text{if } j \text{ is not self} \\ V_{\text{receiver}}[j] + 1, & \text{otherwise} \end{cases} \]
Comparing Vector Timestamps

• $a = b$ if they agree at every element
• $a < b$ if $a[i] \leq b[i]$ for every $i$, but $!(a = b)$
• $a > b$ if $a[i] \geq b[i]$ for every $i$, but $!(a = b)$
• $a \parallel b$ if $a[i] < b[i]$, $a[j] > b[j]$, for some $i,j$ (conflict!)

• If one history is prefix of other, then one vector timestamp $<\text{ other}$

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

...What’s a notion of consistency?

- Global total ordering? See Wednesday
- Today: Something weaker!
• Concurrent writes may be seen in a different order on different machines.

• Writes that are potentially causally related must be seen by all processes in the same order.
Causal Consistency

<table>
<thead>
<tr>
<th>Host 1</th>
<th>$W(x,a)$</th>
<th>$W(x,c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Host 2</th>
<th>$a=R(x)$</th>
<th>$W(x,b)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Host 3</th>
<th>$a=R(x)$</th>
<th>$b=R(x)$</th>
<th>$c=R(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Host 4</th>
<th>$a=R(x)$</th>
<th>$c=R(x)$</th>
<th>$b=R(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $W(x,b)$ and $W(x,c)$ are concurrent
  - So all processes may not see them in same order

- Hosts 3 and 4 read $a$ and $b$ in order, as potentially causally related. No causality for $c$, however.
Examples: Causal Consistency

<table>
<thead>
<tr>
<th>Host 1</th>
<th>W(x, a)</th>
<th>W(x, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Host 3</td>
<td></td>
<td>b=R(x)</td>
</tr>
<tr>
<td>Host 4</td>
<td></td>
<td>a=R(x)</td>
</tr>
</tbody>
</table>

- Host 1 and Host 3 are consistent.
- Host 2 and Host 4 are inconsistent.
Causal Consistency

• Requires keeping track of which processes have seen which writes
  
  – Needs a dependency graph of which op is dependent on which other ops
  
  – ...or use vector timestamps!
Where is consistency exposed?

- Original model b/w processes with local storage
- What if extend this to distributed storage application?
  - If single server per key, easy to locally order op’s to key
  - Then, causal consistency for clients’ op’s to different keys
  - What if key at multiple servers for fault-tolerance/scalability?
    - Servers need consistency protocol with replication
Partial solution space for DB replication

• Master replica model
  – All writes (& ordering) happens at single master node
  – In background, master replicates data to secondary
  – Common DB replication approach (e.g., MySQL)

• Multi-master model
  – Write anywhere
  – Replicas run background task to get up to date

• Under either, reads may not reflect latest write!
Eventual consistency

• If no new updates are made to an object, after some inconsistency window closes, all accesses will return the same “last” updated value

• Prefix property:
  – If Host 1 has seen write $w_{i,2}$: $i^{th}$ write accepted by host 2
  – Then 1 has all writes $w_{j,2}$ (for $j<i$) accepted by 2 prior to $w_{i,2}$

• Assumption: write conflicts will be easy to resolve
  – Even easier if whole-”object” updates only
Systems using eventual consistency

• DNS: each domain assigned to a naming authority
  – Only master authority can update the name space
  – Other NS servers act as “slave” servers, downloading DNS zone file from master authority
  – So, write-write conflicts won’t happen

$ ORIGIN coralcdn.org.

@ IN SOA ns3.fs.net. hostmaster.scs.cs.nyu.edu. (18 ; serial
1200 ; refresh
600 ; retry
172800 ; expire
21600 ) ; minimum
Typical impl of eventual consistency

• Distributed, inconsistent state
  – Writes only go to some subset of storage nodes
    • By design (for higher throughput)
    • Due to transmission failures
    • Declare write as committed if received by “quorum” of nodes

• “Anti-entropy” (gossiping) fixes inconsistencies
  – Use vector clock to see which is older
  – Prefix property helps nodes know consistency status
  – If automatic, requires some way to handle write conflicts
    • Application-specific merge() function
    • Amazon’s Dynamo: Users may see multiple concurrent “branches” before app-specific reconciliation kicks in
## Amazon’s Dynamo: Back-end storage

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorums</td>
<td>Availability and durability when some replicas not available.</td>
</tr>
<tr>
<td>Recovering permanent failures</td>
<td>Anti-entropy using crypto-hash trees</td>
<td>Synchronizes divergent replicas in background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership and failure detection</td>
<td>Avoids needing centralized registry.</td>
</tr>
</tbody>
</table>

![Diagram showing client requests and request routing to Dynamo instances via Amazon S3.](image)
Summary

• Global time doesn’t exist in distributed system

• Logical time can be established via version #’s

• Logical time useful in various consistency models
  – Strong > Causal > Eventual

• Wednesday
  – What are algorithms for achieving strong consistency?
  – What’s possible among distributed replicated?
    • Strong consistency, availability, partition tolerance: Pick two