Advanced Topics in Computer Systems, CS262B

Prof Eric A. Brewer

Bayou: Flexible Update Propagation

March 4, 2004

This lecture covers the mechanism of pair-wise update propagation, next lecture looks at conflict detection and resolution. So here we will just say "roll back and roll forward" and next time we cover how to do so.

I. Background

Traditional replication: (synchronous replication)

- o read one, write all (as in transactional caching)
- o not highly available
- o also slow for writes, poor concurrency
- o improvement: write to majority only (quorum)

Instead: read-any and write-any (asynchronous replication)

- o weak consistency but highly available
- o enables large-scale replication
- o Golding, 1992, "ideal for Internet and mobile computing"
- o Anti-entropy (below) as a way to reduce inconsistency over time
- o Goal: eventual consistency -- all servers agree on the committed writes; this implies some servers must reorder their writes, which means rolling back and then forward in the correct order

Epidemic Algorithms

- o Idea: want consistency to spread like an epidemic
- o Can show that with random pair-wise contacts, infection spreads in log n steps
- o But, it is only *probabilistic*: chance of not being infected approaches zero with more steps...
- o Example: Probability of *not* being infected at step i = P_i, then $P_{i+1} = P_i * P_i$, which is the probably that you were not already infected times the probability that the node you contact was also not infected. This approaches zero quickly since $P_i < 1$.

II. Anti-Entropy

Three key properties:

- o **pair-wise** reduction of inconsistency
- o **autonomous**: any pair can make progress toward eventual consistency. Disconnected subgroups can agree on their ordering even if they can't commit.

- apply **deltas**: leads to less traffic (more on this in next lecture)
- Also is one way: sender updates the receiver; but you can obviously repeat in the other direction.

Version 0: don't have a regular database, just keep a log of all updates (like LFS)

- Claim: anti-entropy is just a sorted merge join of the list of updates sorted by timestamps
- After the sort, the receiver reflects the senders updates, but not vice verse
- After both directions, the two are self consistent (but may be inconsistent with others).
- Since, we just keep lists of updates, there are no undo/redo issues.

III. Ordering

Simple Ordering:

- Simplest order: autonomous clocks at every server
- Only guarantee is that two writes at the same server are serializable (have a well-defined order)
- implies timestamps have the form <timestamp, server_id>, and that the order is a partial order and not a total order
- "sorted" lists of updates are thus topologically sorted

Version Vectors:

- goal: quick way to say which updates you know about
- representation: a vector of timestamps, one for each server [<ts1, S1>, <ts2, S2>,..., <tsk, Sk>]
- invariant: updates must be applied in order so that a single timestamp => that all earlier updates have also been applied
- So in Version 0 algorithm, we first get the receivers version vector to see what updates are needed and then send only the deltas for those
- Receiver must apply them in order, as to make *incremental progress* (and they must be indempotent)

Three problems with Version 0:

- doesn't have a database (next lecture)
- doesn't have a total order, which makes eventual consistency suspect (but not impossible)
- timestamps from different sites might be out of order in terms of both real time, and externally visible events (no causal ordering)

Total ordering:

- simple: just sort by timestamp and break ties using the server ids
- This is a total order but may not reflect reality: interaction among servers may imply that certain events happened before others (regardless of the official timestamps)

Causal Ordering:

- Lamport's "happened-before" relation
- Conservative: tells us when the ordering *might* matter (but it might not)
- In general, event e1 happened-before e2 if and only if e1 could have affected the outcome or existence of e2.
- For us, this means that new writes must be ordered after any writes that we know about from other servers (since we see all writes immediately upon receipt)
- Solution: logical clocks
  - clocks roughly follow real time clocks
  - but... if we receive and event with a higher timestamp, advance our local clock
  - ensures that our future writes have a higher timestamp, the thus correctly follows any potential causal ordering: i.e. $e1<e2 \Rightarrow ts1 < ts2$ regardless of which two nodes did the writes
  - it also orders things that were not causal: $ts1 < ts2$ does not imply $e1 < e2$
- We still need to use server ids as a tiebreaker to get a total order

Version vectors revisited:

- We can't just use logical clocks to tell us which updates we need, since we may be missing some updates from the past (there is no prefix property for logical clocks, only for version vector elements)
- Version vectors of logical clock timestamps *do* tell us about causality
- A version vector is just a cut through the topological sort of updates
- $e1 < e2 \Rightarrow VV(e1) < VV(e2)$ where this means that the cut for VV(e1) is strictly before the cut for VV(e2), i.e. every *element* is ordered correctly
- If the cuts intersect, they are unordered, and there is no potential causality

Commit order:

- Need a way to agree on the commit order for eventual consistency
- Simple solution: assign one server as the primary, and have it decide the commit order
- This is trivially serializable!
- new three-part timestamp: <CSN, ts, server>
  - CSN = commit sequence number assigned by the primary, increments with each write
  - tentative writes have CSN = $+\infty$, which means that committed writes are ordered before all tentative writes
  - ts, server sub-elements are the same as before (with logical clocks)
- Now during anti-entropy, we exchange CSN and Version Vector

Eventual Consistency:

- use the total order implied by <CSN, ts, server>
- Committed writes are stable and their order never changes
- For nodes that do not know about a commit, they will eventually re-order their writes to match the commit order.

- No support for commutative writes, they get an order just like everything else, and nodes with the wrong order must reorder commutative writes anyway (because they have no way to know that they are commutative -- more on this in next lecture)

IV. Log truncation:

Keep a VV of the truncation time R.OVV

- this allows receiver to ignore updates that it has already applied and thrown away
- OVV is a precise cut through time; OSN is not enough to tell which writes have been applied!
- Servers discard a prefix of writes for each server, whch means it is precisely captured by OVV
- Writes commit in order for a single server, so we know that a prefix of committed writes is stable

V. Autonomous Server Creation/Retirement

Create/Delete Servers:

- key idea: make the name of a server be relative to another server so that we can timestamp its creation and retirement
- name of server i, Si, = “<ts_k1, Sk>” where Sk is the name of the server that authorized the creation. This is a recursive name -- there is a tree of servers and each server name gives a pointer to its parent.
- Now server creation/deletion is autonomous: the number of servers can increase or decrease over time and differently in different parts of the network
- Because of creation timestamp and logical clocks, we can tell if a receiver doesn’t know about a new server because a) it never found out, in which case the receivers time for Sk is less than ts_k1 (the time of creation), and b) if found out and the new server was deleted already.
- Retirement writes can just go to the server itself, but must be the last write for that server
- A new server will need a full database transfer to get up to date...