Advanced Topics in Computer Systems, CS262B
Prof Eric A. Brewer

Bayou: Managing Update Conflicts

March 9, 2004

This lecture covers conflict resolution; the previous lecture looked at update propagation and ordering.

I. Background

Clients make writes autonomously, and need only contact one server to perform a write

- read-any and write-any (asynchronous replication)
- weak consistency but highly available
- enables large-scale replication
- Anti-entropy as a way to reduce inconsistency over time
- Logical clocks to capture causal ordering
- 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**

II. Anti-Entropy Revisited

Pair-wise reduction of inconsistency

Autonomous:

- any pair can make progress toward eventual consistency
- disconnected subgroups can agree on their ordering even if they can't commit.

Apply **logical deltas**:

- leads to less traffic
- physical deltas can get confused if something is deleted and re-added -- need to know the process not just outcome
- logical deltas enable automatic resolution and simplify reordering

Also is one way: sender updates the receiver; but you can obviously repeat in the other direction.

Basic algorithm:

- sender gets the CSN, OSN, VV from the receiver
- Normal case: S.OSN < R.CSN < S.CSN
    - R is missing some committed updates
    - Send all of the missing committed writes (R.CSN + 1 through S.CSN)

        - Then use version vector to determine missing tentative writes
        - Receiver may have some tentative writes than are not committed. This is detected when we walk through the missing committed writes; instead of sending the write, we just send the commit notification. (We can tell that a receiver knows about the write via the version vector.)
        - If we receive any writes that are in the past in logical time, then we must roll back and roll forward (at R only)
    - if R.CSN < S.OSN, then receiver is missing updates that we threw out!
        - Roll back all tentative writes of S to the time of S.OSN
        - Send database to R, and also update R.CSN=R.OSN=S.OSN, R.OVV=S.OVV
        - Now merge tentative writes as above (roll forward)

III. Conflicts

Basic problem:

- receiver learns about updates that are in the "past"
- must roll back the database and then roll forward

All tentative writes may conflict with these new past writes (committed or not)

- need to detect conflicts
- ideally, resolve them automatically
- not always possible
- worse: may have had real side effects (e.g. print check) => **can't really allow real effects until writes commit, which is not a highly available process!!!**
- all writes must be undoable, including their side effects
- UI issues: need to visually distinguish tentative writes -- calendar entry should change color when it commits. keep in mind: Bayou is *not* trying for transparency -- tentative writes *should* be exposed.

Conflict Detection: dependency checks

- idea: execute a function that confirms a precondition, if the precondition doesn't hold, we have a conflict
- example: find overlapping meetings (via an SQL-like query). precondition is that this set is empty
- detects read-write conflicts, similar to optimistic concurrency control (e.g. atomic compare and swap); precondition is that read values haven't changed (Note: this is a value-based test, which means it can be fooled by the ABA problem!)
- better example: precondition for withdrawal is only that their be *enough* money, not that it has the same amount as before!
- key result: reduce the number of conflicts via a very narrow definition of conflict!
- a few problems: need a query language to describe dependencies -- this seem awkward

for many applications...

Conflict resolution: Merge Procedures

- written in a high-level interpreted language (but not the same as the dependency check query language!)
  - language is Tcl with some restrictions
  - in practice, merge procedures are "typed" and use a template, where each type has a template that the app fills in with the specifics for this write. Avoids having to rewrite the common code for one class (type) of writes
- can have embedded data, but must be deterministic
- key idea: merge is not only app specific, but also write specific
  - example: alternate times for *this* meeting
- why separate detection and resolution? hope is that detection is lighter weight, and that conflicts are rare
-
- conflict resolution still fail, but we have reduced the chances.
- no support for unresolved conflicts other than an error log -- so these better be rare. Claim is that this is outside the scope of Bayou, but I don't agree...
- conflicts may cascade: e.g. the merge procedure selects an alternate time that causes conflicts for upcoming writes
- Coda has auto conflict resolution for directory operations; these could have been written using dependency checks and merge procedures

How many redos?

- depends only on the number of reorderings, not on the conflicts!
- a write must be undone/redone to maintain the global order, even it is already serializable! (e.g. commutative operations)
- however, writes that are already serializable won't have a conflict, so they are easy to redo...

Redo must be deterministic

- idea: start at same state, apply same updates in the same order, then same end state (on all servers)
- this means dependency checks and merges must return the same result on all servers
  - can't fail due to lack of local resources or local configuration issues!
  - solution: fixed resource bound so that failures will occur uniformly on all servers
  - this seems somewhat hard in practice (need very consistent configurations)

Stable writes:

- need to know when writes commit

    - allows progress of real actions
    - affects UI
    - special API for asking about commit status
  - Which server should be the primary?
    - really should be different for different namespaces (apps)
    - example: calendar primary might be the laptop, while file system primary is probably a centralized server
  - Writes are NOT committed in logical clock order!
    - old writes may arrive after a write has committed
    - only guarantee is that writes from the same server commit in order
    - hope is that merge procedures fix everything up...

IV. Tuple Store

Essentially a SQL database

  - in-memory relational database
    - relational helps with the query language for dependency checks
    - in-memory simplifies implementation issues, but may be a limitation in practice!
    - logs are on disk to ensure durability
    - also need the stable checkpoint on disk (since we truncate the log)
  - need to track two versions: tentative and committed
    - each tuple has two extra bits: in tentative view, in committed view
    - queries return these bits, which can then be used to filter results
    - not that clear what happens on a join or a projection; what “views” does the resulting tuple support?
  - during anti-entropy, roll back to earliest newly inserted write (usually a committed write, since they precede all tentative writes)