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

Transactional Caching

March 2, 2004

Goal:

- o Enable data caching for OODB (navigational), data shipping
- o Why not for relational?
- o exploit large aggregate disk/memory provided by clients
- o Must maintain ACID semantics
- o Caching is only for performance -- not availability!

Semantics:

- o One-copy serializability -- equivalent to serializability without any replication
- o Must be degree 3 for *any* client program
  - Partitioned clients are unavailable and must abort any active transactions
  - But do assume that client side behaves well (at least in the client-side library)
- o Server can always get control back by aborting a transaction (but must control the commit decision)
- o Clients are “second class” replicas, just as in Coda. But Coda choose CAP:AP and here we choose CAP:CP. Server replicas are first class and choose CAP:AC (for both).

Caching vs. Replication:

- o Replication is very ensuring availability of data; only first-class replicas count toward this goal!
- o Caching is dynamic replication with no impact on availability (although if you lost all your replicas you’d probably look in the clients to see if they had a copy)
- o Clients copies are never the master copy; they are always soft state

Other kinds of caching:

- o Metrics: correctness criteria, granularity, costs, workloads
- o Shared-memory multiprocessor caching:
  - Limited concurrency -- the set of processes is known in advance
  - Serialize *actions* rather than *transactions*
  - No need to support durability
  - Must have very low overhead -- very fine grain sharing (every load/store)
- o Distributed Shared Memory (DSM)

    - Same as multiprocessor except the granularity is larger (pages), which opens up room for more complex protocols
- Distributed File Systems
    - Assumes write sharing is rare (backed up by traces)
    - Handles durability, but not isolation
    - Can cache pages or whole files (but large grain either way)

Key question: detect stale data or avoid it?

- Stale == older than most recent *committed* value
- Detection:
    - Check on access, either directly or lazily.
    - Lazy checks assume it is OK to process and must abort if wrong
    - Checks must complete before commit succeeds
- Avoidance:
    - Local copy is always current
    - Server must keep track of all copies (uses a directory)
    - Client must handle event arrivals about state changes, which is more complicated than the detection case, which is always call-return based (i.e. RPC)
    - On commit, all copies must be updated or invalidated (called propagation vs. invalidation in the paper)

Detection taxonomy:

- When is validity (read permission) checked? (update permission is similar)
    - Synchronously (pessimistic)
    - Async: issue check, but start with current copy; on reply we may have to abort
    - Deferred: check right before commit (very optimistic); waste a lot of work if check fails
    - Note: in all cases, client retains this permission until at least commit/abort (2PL). Unlike locks, permission may stay at the client post xact, and is shared by all transactions on that node.
- Change notification hints: notify other of updates, but just a hint
    - None
    - During the transaction -- try to help others avoid wasting work, but if other xacts use your update then may have cascading aborts; instead just invalidate their copy!
    - After commit -- similar but now you can updated others' copies proactively
- Remote update action
    - Invalidate, propagate or dynamic. Dynamic generally wins...

Avoidance Taxonomy:

- Write intention declaration: tell others that their copies may become invalid
    - sync (pessimistic): on write permission fault (after you get permission)

- async: tell them but don't wait -- they may have to abort or you may have to abort (see remote conflict policy below)
- deferred (very optimistic): tell them only at commit -- they are more likely to abort
- Note: no need to do anything for reads -- if you have a copy it's valid (but you might still get aborted depending on optimism)

o Write permission duration

- just this transaction
- until you give it up or the server invalidates (reduces traffic for multiple xact on the same client)

o Remote conflict priority:

- Wait for current readers to finish -- new write blocks until reader xact finishes (active readers serialized before writer)
- Preempt: abort active readers (write serializes first and readers start over)

o Remote update action:

- Invalidate, propagate, dynamic: very similar
- Must complete before xact commits -- propagate requires 2PC to install as part of commit, but invalidate takes one phase since it can't fail (there's no voting about it).