Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

- A theme: two-phase protocols
  - Courtesy Jim Gray:
    - Marriage Ceremony: "Do you?" "I do!" "I now pronounce you..."
    - Theater: "Ready on the set?" "Ready!" "Action!"
    - Contract Law: Offer. Signature. Deal/lawsuit.
  - Actually these protocols are pretty simple
    - Fussy to prove they're safe/correct
    - Even fussier to tune them and maintain proofs, and that's where much of the sweat goes.
- Two Phase Commit and Logging in R*
  - Setup
    - Roles
      - coordinator (transaction manager or TM)
      - subordinate (resource manager, or RM)
    - Goal: All or nothing agreement on commit (single subordinate veto is enough to abort).
      - Also, integrate properly with log processing and recovery.
    - Assumptions
      - Update in place, WAL
      - batch-force log records
    - Desired characteristics
      - guaranteed xact atomicity
      - ability to "forget" outcome of commit ASAP
      - minimal log writes and message traffic
      - optimized performance in no-failure case (the "fast path")
      - exploitation of completely or partially R/O xacts
      - maximize ability to perform unilateral abort
    - In order to minimize logging and comm:
      - rare failures do not deserve extra overhead in normal processing
      - Hierarchical commit better than 2P
  - The basic 2PC protocol with logging (normal processing):
    - Coordinator Log | Messages | Subordinate Log
    - | PREPARE |
    - | | prepare*/abort*
    - | VOTE Y/N |
    - commit*/abort* | |
    - | C/A |
    - | | commit*/abort*
    - | ACK |
    - end | |
    - Rule: never need to ask something that you used to know! Log before ACKing.
      - Since subords force abort/commit before ACKing, they never need to ask coord to remind them about final outcome.
    - Costs:
      - subords: 2 forced log-writes, 2 msgs
      - coord: 1 forced log write, 1 async log write, 2 msgs per subord
      - total: 4n messages, 2N+1 log writes. Delays: 4 message delays, 3 sync writes.
        - we'll tune this down below
  - 2PC and failures
    - Note: 2PC systems are *not available* during a coordinator failure! Yuck!! (See Paxos Commit, below, for discussion)
      - what about subordinate failure?
    - Recovery process protocol:
      - 1 On restart, read log and accumulate committing xacts info in main mem
      - 2 if you discover a local xact in the prepared state, contact coord to find out fate
      - 3 if you discover a local xact that was not prepared, UNDO it, write abort record, forget
      - 4 if a local xact was committing (i.e. this is the coord), then send out COMMIT msgs to subords that haven't ACKed Similar for aborting.
    - Upon discovering a failure elsewhere

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

- If a coord discovers that a subord is unreachable...
  - while waiting for its vote: coord aborts xact as usual
  - while waiting for an ACK: coord gives xact to recovery mgr
- If subord discovers that coord is unreachable...
  - if it hasn't sent a YES vote yet, do unilateral abort
  - if it has sent a YES vote subord gives xact to recovery mgr
- If a recovery mgr receives an inquiry from a subord in prepared state
  - if main mem info says xact is committing or aborting, send COMMIT/ABORT
  - if main mem info says nothing...?

- An aside: Hierarchical 2PC
  - If you have a tree-shaped process graph
    - root (which talks to user) is a coord
    - leaves are subords
    - interior nodes are both
      - after receiving PREPARE, propagate to children.
      - vote after children. any NO below causes a NO vote (this is like stratified aggregation!)
      - after receiving COMMIT record, force-write log, ACK to parent, and propagate to children. similar for ABORT.
- Tuning approach 1: Presumed Abort
  - recall... if main-mem says nothing, coord says ABORT
  - SO... coord can forget a xact immediately after deciding to abort it! (write abort record, THEN forget)
  - abort can be async write
    - no ACKS required from subords on ABORT
    - no need to remember names of subords in abort record, nor write end record after abort
    - if coord sees subord has failed, need not pass xact to recovery system; can just ABORT.
  - Look at R/O xacts:
    - subords who have only read send READ VOTEs instead of YES VOTEs, release locks, write no log records
      - logic is: READ & YES = YES, READ & NO = NO, READ & READ = READ
      - if all votes are READ, there's no second phase
      - commit record at coord includes only YES sites
      - Tallying up the R/O work: N+1 msgs, no disk writes. Delays: 1 msg delay.
- Tuning approach II: Presumed Commit
  - Should be the fast path, can we do it fast?
  - Inverting the logic:
    - require ACK for ABORT, not COMMIT!
    - subords force abort* record, not commit
    - no info? presume commit!
  - Problem!
    - subord prepares
    - coord crashes
    - on restart, coord aborts and forgets
    - subord asks about the xact, coord says "no info = commit!"
    - subord commits, but everybody else does not.
  - Solution:
    - coord records names of subords on stable storage before allowing them to prepare ("collecting" record)
    - then it can tell them about aborts on restart
    - everything else analogous (mirror) to P.A.
    - Tallying up R/O work: N+1 msgs, 2 diskwrites (collecting*, commit), Delays: 1 diskwrite delay, 1 msg delay.
- Costs of the variants
  - 2PC commit: 2N+2 writes, 4N messages. Delays: 3 write delays, 4 msg delays
  - PA commit: 2N+2 writes, 4N messages. Delays: 3 write delays, 4 msg delays
  - PC commit: 2N+2 writes, 3N messages. Delays: 3 write delays, 3 msg delays.
  - PA *always* beats plain 2PC
  - PA beats PC for R/O transactions
  - for xacts with only one writer subord, PC beats PA (PA has an extra ACK from subord)
  - for n-1 writer subords, PC much better than PA (PA forces n-1 times at subords on commits, sends n extra msgs)

<u>Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos</u>

    - choice between PA and PC could be made on a xact-by-xact basis!
      - "query" optimization? Overlog?
- Paxos
  - Setup
    - 3 roles being played
      - A single Proposer ("Leader"), proposes "values"
        - Leader-election protocol is well-known and predates this work
      - Acceptor, part of protocol to decide on "choosing" values
      - Learner, hears about "chosen" values
    - Goal: majority agreement to "choose" a proposed value
      - Imagine a single Consensus Box. Now emulate that with a distributed set of machines that can tolerate failure.
      - Non-triviality: only proposed values can be learned
      - "Consistency": 2 learners cannot learn different values
      - Liveness: if value C has been proposed, and enough processes are alive, eventually each learner will learn some value
    - Assumptions
      - Async machines
      - Independent, fail-stop failures
        - will tolerate F/(2F+1) nodes failing *simultaneously*.
        - vs. 2PC. vs. Byzantine Agreement.
      - msgs lost, delayed, reordered, but not corrupted.
  - The basic Paxos protocol
    - <u>Proposer</u> | <u>Acceptors</u> | <u>Learner</u>
    - prepare(n)→ | |
    - |← promise (m,w) |
    - Accept(n,v) → | |
    - |← accepted → |
    - | | broadcast →
    - notes:
      - acceptors only promise(m,w) if $m < n$ and they haven't promised something higher than n already
        - w is the last value *accepted* (or null)
      - proposer only issues accepts if a majority promised. if all acceptor returned null w's, proposed gets to choose v (the *free* case). else v is the w it received with the highest associated m (the *forced* case).
        - why should a proposer bother accepting if it is forced by a non-null w?
    - <u>Costs</u>
      - 4F messages, 4 message delays.
  - Paxos with failures
    - Acceptor failures
      - First, note that all majorities overlap by 1
        - Whenever a majority of acceptors is non-failed in future, previously accepted values will be stored with associated numbers.
      - Second, note how promises help
    - Learner failures
      - trivial
    - Proposer failures
      - Leader-election will replace proposer on failure
      - Proposer can fail any time before accept with no confusion
      - Fail after Accept msg sent out causes trouble: dueling proposers
        - new leader will be elected, and if old leader recovers she won't know she's no longer leader
        - prepare(n) will fail
        - new leader may try to restart with prepare(n+1)
          - gets promises
        - old leader recovers and tries to restart with prepare(n+1)
          - gets NACKs
        - old leader tries prepare(n+2)
          - gets promises
        - new leader tries to accept(n+1)

Brewer/Hellerstein CS262 Spring 2008: 2PC and Paxos

- gets NACKs
- etc,.
- Leader-election will eventually solve this
- Many variants -- see Wikipedia entry
  - Multi-Paxos: for continuous stream of consensus tasks. Skips Phase 1.
    - Very typical implementation
    - (Actually, we can always skip Phase 1, even without multi)
  - Cheap Paxos: let F of the 2F+1 machines be slow
  - Fast Paxos: skip phase 1, let clients initiate phase 2 via broadcast to proposer and acceptors
  - Byzantine Paxos: allows for nodes to be malicious.
- Paxos and distributed state machines
  - A nice model (the usual model!) for reasoning about fault-tolerant systems is the distributed state machine
    - multiple clients
    - server implemented by multiple nodes running redundant copies of the same deterministic state machine
    - how do we ensure that each machine runs the same commands in the same order?
      - a Paxos leader (proposer) serializes all client requests.
      - it uses Paxos to get consensus on the content of the n'th request
    - if leader fails, leader election picks a new one. recovery works out pretty well:
      - even if we have dueling leaders!
        - Phase 1 of Paxos is used to get one of the leaders to "win" the nth Paxos round
        - Only in Phase 2 does that leader actually issue the command.
          - the command for for round *n* is only chosen after Phase 2 for round *n-1* completes
          - hence to choose a command, you have to be all caught up on history, and hence choose the "right" one.
      - how does a new leader "catch up"
        - well, it had been a listener, so it has a partial view of history
        - start by issuing Phase 1 requests for any gaps in history, and *all "future" rounds* (expained below)
          - will learn the history from the Promise responses
        - run Phase 2 for all the promises that responded with a value
          - at minimum local execution of the commands
          - to complete the sequence of historical commands, replace any remaining gap commands with no-op proposals
        - what does it mean to do phase one for all future rounds (infinitely many)?
          - propose a *single sequence number* in one message, representing an unbounded number of rounds
            - acceptor can simply say OK
- Paxos Commit
  - Gray & Lamport 2006!! (from a 2004 TR)
  - History: Skeen's Non-Blocking (3-Phase) Commit
    - Handle the case of a failed transaction coordinator
      - multiple coordinators and failover
      - nobody every nailed this down (specific algorithm with correctness proofs)
  - Paxos makes this really simple
    - we can have multiple coordinators (transaction managers), and their decisions on commit are handled by Paxos
      - client issues "prepare" to multiple coordinators
      - subordinates respond "prepared" to all coordinators
      - Paxos used to deal with coordinator decisions if any of the coords fail.
        - Note -- still unanimous decision by subordinates! Majority used at coordinators.
      - Same logging all around
      - A version of this due to Mohan in 1983 (with a slower consensus protocol)
      - Paxos Commit also includes an optimization over the Mohan solution
        - coordinator need not be the Paxos proposer!
        - subordinates don't respond to coordinator prepare. instead, they serve as Paxos proposers for their own status
        - coordinators are Listeners on those proposals, and can issue commits upon getting a majority for *each* subordinate
        - saves one round of messages
        - Acceptors in Paxos must log each accepted message before sending it.
    - Total cost (with all optimizations): (N-1)(2F+3) msgs, N+F+1 writes. 4 message delays, 2 write delays.
  - Full paper is (typically) complex and full of fussy detail