Paxos

15-712 Fall 2007

Some slides in this lecture borrowed from Mike Reiter, Robert Morris
(Several slides in this section borrowed from):
Introduction to Agreement Algorithms

Mike Reiter
Distributed Systems

• A collection of computing devices that can communicate with each other

• How are distributed systems different from sequential ones?
  – May be impossible to observe the global state
  – Can incur *partial* failures (devices or communication)
  – Measures are different
    • Time is still important, but messages are, too
  – Much more difficult to reason about and get right
Agreement Problems

• High-level goal: Processes in a distributed system reach agreement on a value

• Numerous problems can be cast this way
  – Transactional commit, atomic broadcast, ...

• The system model is critical to how to solve the agreement problem—or whether it can be solved at all
  – Failure assumptions
  – Timing assumptions
Failure Model

• A process that behaves according to its I/O specification throughout its execution is called **correct**.
• A process that deviates from its specification is **faulty**.
• There are many gradations of faulty. Two of interest are:

  - **Crash failures**: A faulty process halts execution prematurely.
  - **Byzantine failures**: *No assumption* about behavior of a faulty process.
Timing Model

• Specifies assumptions regarding delays between
  – execution steps of a correct process
  – send and receipt of a message sent between correct processes
• Again, many gradations. Two of interest are:

- **Synchronous**
  Known bounds on message and execution delays.

- **Asynchronous**
  *No assumptions* about message and execution delays (except that they are finite).
Today

• Crash-failure
• Asynchronous

• Next week:
  – Byzantine failure
  – Sync & Async
Consensus

• Each process begins with a value
• Each process can irrevocably decide on a value
• Up to $t < n$ processes may be faulty

• Problem specification
  – **Termination**: Each correct process decides some value.
  – **Agreement**: Correct processes do not decide different values.
  – **Validity**: If all processes begin with the same input, then any value decided by a correct process must be that input.
Consensus: Synchronous Crash Model

Algorithm for $S_i$

$S_i \leftarrow \{\text{initial value}\}$

for $k = 1 \ldots t+1$

send $S_i$ to all processes

receive $S_j$ from $j$ for all $j$

$S_i \leftarrow S_i \cup \left( \bigcup_j S_j \right)$

decide min($S_i$)
Example with $t = 1$

End of round 1

End of round 2

decide 0

decide 0

{0}

{1}

{0,1}

{0,1}

{1}

{1}

{0,1}

{0,1}
Consensus: Asynchronous Crash Model

Theorem [Fischer, Lynch, Paterson]: There is no algorithm to solve consensus in an asynchronous system for any $t \geq 1$.

At least, if you want termination.
But that’s okay - we’ll scrap that requirement...
Refresher
Refresher

• 2-phase commit
  – Have to wait for all nodes + coord to be up
  – Have to know how each node voted
  – coord must be up to decide
  – Works, but system is down while any one component is down: long repair times
Back to State Machine Replication

• Works for any replicated service
  – storage, lock server (Google’s chubby), etc.
  – Every replica must see same operations in same order
    • If deterministic ops, all replicas will be in same state
Strawman: Primary/Backup

• Primary assigns order of ops, sends them to all replicas
  – What if primary fails?
    • What about operation in flight when primary failed?
    • Need to pick a new primary
    • But can’t have two, or order is wrong!
  – Simple approaches don’t work
    • Lowest #’d server? Partition / lost pings => 2 primaries
Basic system structure

• Ordinary (non-failure) operation:
  – Pick a primary
  – Let it sequence things
  – Works efficiently and happily

• But make sure that on failure
  – The system is *always* correct
  – How can we do this?
Agreement

• Leader chooses proposed value to agree on
  – Broadcasts to all participants, tries to assemble majority
  – If majority respond, life is good
• What if leader crashes after contacting only some nodes?
• What if got majority, then failed?
• What if two leaders simultaneously?
Paxos

• Three phases
  – Each node maintains state:
    • Na, Va: Highest N that node has accepted and value V
    • Np: highest N seen in any PREPARE

• Phase 1:
  – Some node decides it’s a leader
  – Picks unique proposal # n > higher known #s
  – Sends PREPARE(n) to every node
  – recv(PREPARE(n)):
    – if n > Np
      • return RESPONSE(Na, Va)
      • Np = n
Phase 1

A

PREPARE(n)

B

C

D
Phase 2

- If response from majority of nodes
  - If RESPONSE(n, v) has a value
    - v = value of highest n
    - else v = pick anything
  - send ACCEPT(n, v) to all nodes
- on recv(ACCEPT(n,v))
  - if n >= Np
    - Na = n
    - Va = v
- If majority accept, we have a value!
  - But we might not know! Leader crash b4 report...
Phase 3

• Tell everyone the agreed-upon answer
Failures: Multiple Leaders

- Two leaders must use different $n$
  - Augment $n$ with node ID, etc.
- A: PREPARE(5)
- A,B,C: RESPONSE(5, v)
- D: PREPARE(6)
- B,C,D: RESPONSE(6,v)
- A: ACCEPT(5, v)
- B,C: No! We want to hear $\geq 6$
- A: PREPARE(7)
- D: ACCEPT(6, v)
- B,C: No! We want to hear $\geq 7$
- ...
Multiple Leaders

• Can continue forever
  – But won’t in most failures
  – Broadcast leader election, random backoff, etc.
  – Could even use more robust leader election (may be useful in wide-area): gossip, etc.
Leader failure

• Before sending ACCEPTs
  – Some other node will decide to become leader
  – Old leader never reached agreement, so just ignore
  – Our new $N > \text{old } N$ will ensure that their old requests are flushed out even if they’re delayed
Failure after sending ACCEPT?

• Key idea:
  – Once a majority agrees, it can never un-agree
  – Why? They send back the value they agreed upon
    • Two majorities *must* overlap, so new leader will always hear old agreed-upon value
      • If leader hears a v, it must pick that v as its own
  • (Same as ensuring correctness with two leaders (but not progress))
Requires persistence

• e.g., node reboot after RESPONSE
  – L1 PREPARE(10). node X Np = 10
  – L2: PREPARE(11); majority intersecting only at node X response. node X Np = 11
    • L2 picks a value v=200
  – X crashes & reboots, resets Np (ERROR!)
  – L1 sends ACCEPT(n=10, v=100)
    • It’s accepted! Node X forgot...

L1 majority

L2 majority

X
Optimizations

• Doing this every time is *expensive*
  – Can amortize across multiple requests using a view
  – Use Paxos to agree on a \{leader, view, participant set\}
  – First req from new leader: Normal paxos
  – Subsequent reqs: Directly send “accept”, respond back “accepted”.

Paxos in Practice

• Example: Google’s “Chubby” lock server
  – Uses paxos to manage locks & leases & leader election
  – But then most services use cheaper mechanisms (e.g., using the leader)
  – Much like the optimizations to using Paxos itself
    • Pick a leader, let it do the work in the absence of failures