Recovering from a Crash

- If INIT: abort locally and inform coordinator
- If Ready, contact another process Q and examine Q’s state

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Three-Phase Commit

Two phase commit: problem if coordinator crashes (processes block)
Three phase commit: variant of 2PC that avoids blocking
Replication for Fault Tolerance

• Basic idea: use replicas for the server and data

• Technique 1: split incoming requests among replicas
  – If one replica fails, other replicas take over its load
  – Suitable for crash fault tolerance (each replica produces correct results when it is us).

• Technique 2: send each request to all replicas
  – Replicas vote on their results and take majority result
  – Suitable for BFT (a replica can produce wrong results)
    • 2PC, 3PC, Paxos are techniques

Consensus, Agreement

• Consensus protocols
• Achieve reliability in presence of faulty processes
  – requires processes to agree on data value needed for computation
  – Examples: whether to commit a transaction, agree on identity of a leader, atomic broadcasts, distributed locks
• Properties of a consensus protocol with fail-stop failures
  – Agreement: every correct process agrees on same value
  – Termination: every correct process decides some value
  – Validity: If all propose v, all correct processes decides v
  – Integrity: Every correct process decided at most one value and if it decides v, someone must have proposed v.
2PC, 3PC Problems

- Both have problems in presence of failures
  - Safety is ensured but liveness is not
- 2PC
  - must wait for all nodes and coordinator to be up
  - all nodes must vote
  - coordinator must be up
- 3PC
  - handles coordinator failure
  - but network partitions are still an issue
- Paxos: how to reach consensus in distributed systems that can tolerate non-malicious failures?
  - majority rather than all nodes participate

Paxos: fault-tolerant agreement

- Paxos lets nodes agree on same value despite:
  - node failures, network failures and delays
- Use cases:
  - Nodes agree X is primary (or leader)
  - Nodes agree Y is last operation (order operations)
- General approach
  - One (or more) nodes decides to be leader (aka proposer)
  - Leader proposes a value and solicits acceptance from others
  - Leader announces result or tries again
- Proposed independently by Lamport and Liskov
  - Widely used in real systems in major companies
Paxos Requirements

• Safety (Correctness)
  – All nodes agree on the same value
  – Agreed value X was proposed by some node

• Liveness (fault-tolerance)
  – If less than N/2 nodes fail, remaining nodes will eventually reach agreement
  – Liveness not guaranteed if steady stream of failures

• Why is agreement hard?
  – Network partitions
  – Leader crashes during solicitation or after deciding but before announcing results,
  – New leader proposes different value from already decided value,
  – More than one node becomes leader simultaneously....

Paxos Setup

• Entities: Proposer (leader), acceptor, learner
  – Leader proposes value, solicits acceptance from acceptors
  – Acceptors are nodes that want to agree; announce chosen value to learners

• Proposals are ordered by proposal #
  – node can choose any high number to try to get proposal accepted
  – An acceptor can accept multiple proposals
    • If prop with value v chosen, all higher proposals have value v

• Each node maintains
  – n_a, v_a: highest proposal # and accepted value
  – n_h: highest proposal # seen so far
  – my_n: my proposal # in current Paxox
Paxos operation: 3 phase protocol

• **Phase 1 (Prepare phase)**
  – A node decides to be a leader and propose
  – Leader chooses \( \text{my}_n > \text{n}_h \)
  – Leader sends \(<\text{prepare, my}_n>\) to all nodes
  – Upon receiving \(<\text{prepare, n}>\) at acceptor
    • If \( n < n_h \)
      – reply \(<\text{prepare-reject}>\) /* already seen higher # proposal */
    • Else
      – \( n_h = n \) /* will not accept prop lower than \( n \) */
      – reply \(<\text{prepare-ok, n}_a, v_a}>\) /* send back previous prop, value/
      – /* can be null, if first */

• **Phase 2 (accept phase)**
  – If leader gets prepare-ok from **majority**
    • \( V = \) non-empty value from highest \( n_a \) received
    • If \( V = \) null, leader can pick any \( V \)
    • Send \(<\text{accept, my}_n, V>\) to all nodes
  – If leader fails to get majority prepare-ok
    • delay and restart Paxos
  – Upon receiving \(<\text{accept, n, V}>\)
    • If \( n < n_h \)
      – reply with \(<\text{accept-reject}>\)
    • else
      – \( n_a=n \); \( v_a = V \); \( n_h = h \); reply \(<\text{accept-ok}>\)
Paxos Operation

- **Phase 3 (decide)**
  - If leader gets accept-ok from majority
    - Send <decide, v_a> to all learners
  - If leader fails to get accept-ok from a majority
    - Delay and restart Paxos

- **Properties**
  - P1: any proposal number is unique
  - P2: any two set of acceptors have at least one node in common
  - P3: value sent in phase 2 is value of highest numbered proposal received in responses in phase 1
Issues

• Network partitions:
  – With one partition, will have majority on one side and can come to agreement (if nobody fails)

• Timeouts
  – A node has max timeout for each message
  – Upon timeout, declare itself as leader and restart Paxos

• Two leaders
  – Either one leader is not able to decide (does not receive majority accept-oks since nodes see higher proposal from other leader) OR
  – one leader causes the other to use it value

• Leader failures: same as two leaders or timeout occurs

Recovery

• Techniques thus far allow failure handling
• Recovery: operations that must be performed after a failure to recover to a correct state
• Techniques:
  – Checkpointing:
    • Periodically checkpoint state
    • Upon a crash roll back to a previous checkpoint with a consistent state
Independent Checkpointing

- Each processes periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Problem: if most recent checkpoints form inconsistent cut, will need to keep rolling back until a consistent cut is found
- Cascading rollbacks can lead to a domino effect.

Coordinated Checkpointing

- Take a distributed snapshot [discussed in Lec 11]
- Upon a failure, roll back to the latest snapshot
  - All process restart from the latest snapshot
Logging

- Logging: a common approach to handle failures
  - Log requests / responses received by system on separate storage device / file (stable storage)
    - Used in databases, filesystems, ...
- Failure of a node
  - Some requests may be lost
  - Replay log to “roll forward” system state

Message Logging

- Checkpointing is expensive
  - All processes restart from previous consistent cut
  - Taking a snapshot is expensive
  - Infrequent snapshots => all computations after previous snapshot will need to be redone [wasteful]
- Combine checkpointing (expensive) with message logging (cheap)
  - Take infrequent checkpoints
  - Log all messages between checkpoints to local stable storage
  - To recover: simply replay messages from previous checkpoint
    - Avoids recomputations from previous checkpoint