**Slide 1**

**Distributed Systems (COMP9243)**

Lecture 8 (B): Fault Tolerance

A Failure
A Reliable Communication
B Process Resilience
B Recovery

---

**Slide 3**

**Consensus in Practice**

Two Phase Commit:
- Original assumption: No failure

Slide 2

Failures can be due to:
- Failure of communication channels:
  - use timeouts
- Server failures:
  - potentially blocking

---

**Slide 4**

Two-phase commit with timeouts: Worker:

- On timeout sends GetDecision.

Two-phase commit with timeouts: Coordinator:

- On timeout re-sends CanCommit. On GetDecision repeats decision.
Coordinator failure:
- When coordinator crashes start a new recovery coordinator
- Learn state of protocol from workers (what did they vote, what did they learn from coordinator)
- Finish protocol

Coordinator and Worker failure: Blocking 2PC:
- Recovery coordinator can’t distinguish between
  - All workers vote Commit and failed worker already committed
  - Failed worker voted Abort and rest of workers voted Commit
- So can’t make a decision

RAFT
Reliable, Replicated, Redundant, And Fault-Tolerant Consensus
Goal: each node agrees on the same series of operations (log)
Log ordered list of operations
Leader node responsible for deciding how to add operations to the log
Followers nodes that replicate the leader’s log
Two subproblems
Leader Election how to agree on who the leader is
Log Replication how to replicate the leader’s log to the followers

THREE PHASE COMMIT
1. Vote: as in 2PC
2. Pre-commit: coordinator sends vote result to all workers, workers acknowledge
3. Commit: coordinator tells workers to perform vote action
Why does this work?

LEADER ELECTION
Term the time during which a node is leader
Candidate node who wants to become leader
Failed Leader:
- Leader sends regular heartbeat to followers
- Follower sees no communication from leader (election timeout)
- Leader sees heartbeat from a later term (steps down as leader)
Slide 11

Candidate:
- Detects that leader has failed
- Sends Request Vote to all other nodes
- Nodes reply
  - Yes: hasn’t voted yet this term
  - No: has already voted this term
- Majority votes -> candidate becomes leader
- or Timeout -> new election

Slide 12

Log Replication

Client

B

Leader

A

Follower

E

D

Follower

C

Follower

Log Replication

Client

B

Leader

A

Follower

E

D

Follower

C

Follower
**Log Replication**

1. Client sends operation to leader
2. Leader appends to its log
3. Leader sends Append Entries message to followers
4. Followers acknowledge
5. Leader commits operation to log

---

**PAXOS**

Goal: a collection of processes chooses a single proposed value in the presence of failure

- **Proposer** proposes value to choose (leader)
- **Acceptor** accept or reject proposed values
- **Learner** any process interested in the result (chosen value) of the consensus

Chosen Value: value accepted by majority of acceptors

Properties:
- Only proposed values can be learned
- At most one value can be learned
- If a value has been proposed then eventually a value will be learned
**Using Paxos**

Use Paxos for:
- Leader election: choose a leader id
  - single paxos instance, elections starter(s) propose leader id, result in an agreed upon leader.
- View synchrony: order view changes
  - one paxos instance per view change: result in a view change order sequence number
- Total order multicast: order messages
  - one paxos instance per message: result in a message sequence number
- State machine replication: order operations
  - one paxos instance per operation: result in an operation sequence number

**Paxos Algorithm: 3 Phases**

Assuming no failures

**Phase 1: Propose:**
1. Propose: send a proposal \(<\text{seq, value}>\) to \(\geq N/2\) acceptors
2. Promise: acceptors reply.
   - accept (include last accepted value), promised = seq.

**Phase 2: Accept:**
1. Accept: when \(\geq N/2\) accept replies, proposer sends value (as received from acceptor or arbitrary):
2. Accepted: acceptors reply
   - accepted. Remember accepted value.

**Phase 3: Learn:**
1. Propagate value to Learners when \(\geq N/2\) accepted replies received.

**Failures**

What can go wrong before agreement is reached?

**Failure Model:**
- channel: lose, reorder, duplicate message
- process: crash (fail-stop, fail-resume)

**Failure Cases:**
1. Acceptor fails
2. Acceptor recovers/restarts
3. Proposer fails
4. Multiple proposers
   - New proposer
   - Proposer recovers/restarts
**PAXOS Algorithm: 3 Phases**

With Failures!

**Phase 1: Propose:**
1. Propose: send a proposal \(<\text{seq}, \text{value}>\) to \(\geq N/2\) acceptors
2. Promise: acceptors reply.
   - reject if \(\text{seq} < \text{seq of previously accepted value}\)
   - else accept (include last accepted value). \(\text{promised} = \text{seq}\).

**Slide 21**

**Phase 2: Accept:**
1. Accept: when \(\geq N/2\) accept replies, proposer sends value (as received from acceptor or arbitrary):
2. Accepted: acceptors reply
   - reject if \(\text{seq} < \text{promised}\).
   - else accepted. Remember accepted value.

**Phase 3: Learn:**
1. Propagate value to Learners when \(\geq N/2\) accepted received.

---

**ACCEP TOR FAILS**

- As long as a quorum still available
- Restart: Must remember last accepted value(s)

**Slide 22**

**PROPOSER FAILS**
- Elect a new leader
- Continue execution
- New proposer will choose any previously accepted value

**Slide 23**

**MULTIPLE PROPOSERS**
**MULTIPLE PROPOSERS**

- For example: crashed proposer returns and continues
- Duelling proposers
- No guaranteed termination
- Heuristics to recognise situation and back off

---

**OPTIMISATION AND MORE INFORMATION**

**Opportunities for optimisation:**

- Reduce rounds
  - Phase 1: reject: return highest accepted seq
  - Phase 2: reject: return promised seq
- Reduce messages
  - Piggyback multiple requests and replies
  - Pre-propose multiple instances (assumes Proposer rarely fails)

**More information:**

*Paxos Made Live - An Engineering Perspective*  Experiences implementing Paxos for Google's Chubby lock server. It turns out to be quite complicated.

---

**FAILURE RECOVERY**

Restoring an erroneous state to an error free state

**Issues:**

- Reclamation of resources: locks, buffers held on other nodes
- Consistency: Undo partially completed operations prior to restart
- Efficiency: Avoid restarting whole system from start of computation

---

**FORWARD VS. BACKWARD ERROR RECOVERY**

**Forward Recovery:**

- Correct erroneous state without moving back to a previous state.
- Example: erasure correction - missing packet reconstructed from successfully delivered packets.
- Possible errors must be known in advance

**Backward Recovery:**

- Correct erroneous state by moving to a previously correct state
- Example: packet retransmission when packet is lost
  - General purpose technique.
  - High overhead
  - Error can reoccur
  - Sometimes impossible to roll back (e.g. ATM has already delivered the money)
**BACKWARD RECOVERY**

General Approach:
- Restore process to recovery point
- Restore system by restoring all active processes

Specific Approaches:

**Operation-based recovery**
- Keep log (or audit trail) of operations (like transactions)
- Restore to recovery point by reversing changes

**State-based recovery**
- Store complete state at recovery point (checkpointing)
- Restore process state from checkpoint (rolling back)

Log or checkpoint recorded on stable storage

**State-Based Recovery - Checkpointing:**
- Take frequent checkpoints during execution

**Checkpointing:**
- **Pessimistic vs Optimistic**
  - Pessimistic: assumes failure, optimised toward recovery
  - Optimistic: assumes infrequent failure, minimises checkpointing overhead
- **Independent vs Coordinated**
  - Coordinated: processes synchronise to create global checkpoint
  - Independent: each process takes local checkpoints independently of others
- **Synchronous vs Asynchronous**
  - Synchronous: distributed computation blocked while checkpoint taken
  - Asynchronous: distributed computation continues while checkpoint taken

**CHECKPOINTING OVERHEAD:**
- Frequent checkpointing increases overhead
- Infrequent checkpointing increases recovery cost

**Decreasing Checkpointing Overhead:**
- **Incremental checkpointing:** Only write changes since last checkpoint:
  - Write-protect whole address space
  - On write-fault mark page as dirty and unprotect
  - On checkpoint only write dirty pages
- **Asynchronous checkpointing:** Use copy-on-write to checkpoint while execution continues
  - Easy with UNIX fork
- **Compress checkpoints:** Reduces storage and I/O cost at the expense of CPU time

**RECOVERY IN DISTRIBUTED SYSTEMS**
- Failed process may have causally affected other processes
- Upon recovery of failed process, must undo effects on other processes
- Must roll back all affected processes
- All processes must establish recovery points
- Must roll back to a consistent global state
Domino Effect:

- $P_1$ fails $\rightarrow$ roll back: $P_1 \cap R_{13}$
- $P_2$ fails $\rightarrow$ $P_2 \cap R_{22}$
- Orphan message $m$ is received but not sent $\rightarrow$ $P_1 \cap R_{12}$
- $P_3$ fails $\rightarrow$ $P_3 \cap R_{32} \rightarrow P_2 \cap R_{21} \rightarrow P_1 \cap R_{11}, P_3 \cap R_{31}$

Messaging dependencies plus independent checkpointing may force system to roll back to initial state.

Message Loss:

- Failure of $P_2 \rightarrow P_2 \cap R_{22}$
- Message $m$ is now recorded as sent (by $P_2$) but not received (by $P_2$), and $m$ will never be received after rollback
- Message $m$ is lost
- Whether $m$ is lost due to rollback or due to imperfect communication channels is indistinguishable!
- Require protocols resilient to message loss

Livelock:

- $P_2 \parallel P_1 \cap R_{21} \rightarrow P_1 \cap R_{11}$. Note: $n_1$ in transit

- Pre-rollback message $n_1$ is received after rollback
- Forces another rollback $P_2 \cap R_{21}, P_1 \cap R_{11},$ can repeat indefinitely

Consistent Checkpointing:

Consistent Cut:
Idea: collect local checkpoints in a coordinated way.

- Set of local checkpoints forms a global checkpoint.
- A global checkpoint represents a consistent system state.

\[ R_{11}, R_{21}, R_{31} \] form a strongly consistent checkpoint:
- No information flow during checkpoint interval
- All messages recorded as received must be recorded as sent

\[ R_{12}, R_{22}, R_{32} \] form a consistent checkpoint:
- All messages recorded as received must be recorded as sent

**Strongly consistent checkpointing** requires quiescent system
- Potentially long delays during blocking checkpointing

**Consistent checkpointing** requires dealing with message loss
- Not a bad idea anyway, as otherwise each lost message would result in a global rollback
- Note that a consistent checkpoint may not represent an actual past system state

**Synchronous Checkpointing**

Processes coordinate local checkpointing so that most recent local checkpoints constitute a consistent checkpoint

**Assumptions:**
- Communication is via FIFO channels.
- Message loss dealt with via
  - Protocols (such as sliding window), or
  - Logging of all sent messages to stable storage
- Network will not partition

**Local checkpoints:**
- **permanent:** part of a global checkpoint
- **tentative:** may or may not become permanent

**Synchronous Algorithm**

- Global checkpoint initiated by a single coordinator
- Based on 2PC

**First Phase:**
1. Coordinator \( P_1 \) takes tentative checkpoint
2. \( P_1 \) sends message to all other processes \( P_i \) to take tentative checkpoint
3. \( P_i \) reply to \( P_1 \) whether succeeded in taking tentative checkpoint
4. \( P_i \) receives **true** reply from each \( P_j \) \( \rightarrow \) decides to make permanent
   \( P_i \) receives at least one **false** \( \rightarrow \) decides to discard the tentative checkpoints
Second Phase:
1. Coordinator $P_i$ informs all other processes $P_j$ of decision
2. $P_j$ convert or discard tentative checkpoints accordingly

Consistency ensured because no messages sent between two checkpoint messages from $P_i$.

Rollback Recovery

First Phase:
1. Coordinator sends "$r" messages to all other processes to ask them to rollback
2. Each process replies true, unless already in checkpoint or rollback
3. If all replies are true, coordinator decides to rollback, otherwise continue

Second Phase:
1. Coordinator sends decision to other processes
2. Processes receiving this message perform corresponding action

Homework
- Find a Paxos or Raft library and implement a replicated state machine using it.
- Hacker’s edition:
  - Implement the Paxos or Raft library (e.g., in Erlang).
**Reading List**

Optional

**In Search of an Understandable Consensus Algorithm** Paper describing (and motivating) Raft.

**Paxos Made Live - An Engineering Perspective** Experiences implementing Paxos for Google’s Chubby lock server. It turns out to be quite complicated.