Distributed Systems (COMP9243)

Lecture 6: Fault Tolerance

1. Failure
2. Reliable Communication
3. Process Resilience
4. Recovery
Dependability

**Availability:** system is ready to be used immediately

**Reliability:** system can run continuously without failure

**Safety:** when a system (temporarily) fails to operate correctly, nothing catastrophic happens

**Maintainability:** how easily a failed system can be repaired

Building a dependable system comes down to controlling failure and faults.
CASE STUDY: AWS FAILURE 2011

- April 21, 2011
- EBS (Elastic Block Store) in US East region unavailable for about 2 days
- 13% of volumes in one availability zone got stuck
- led to control API errors and outage in whole region
- led to problems with EC2 instances and RDS in most popular region
- due to reconfig error and re-mirroring storm.
AWS EBS Overview:

- Region → Availability Zones
- Clusters → Nodes → Volumes
- Volume: replicated in cluster
- Control Plane Services: API for volumes for whole region
- Networks: primary, secondary

What happened?:

- network config problem
- re-mirroring storm
- CP API thread starvation
- node race condition
- CP election overload
Failure

Terminology:

**Failure**: a system fails when it does not meet its promises or cannot provide its services in the specified manner

**Error**: part of the system state that leads to failure (i.e., it differs from its intended value)

**Fault**: the cause of an error (results from design errors, manufacturing faults, deterioration, or external disturbance)

Recursive:

- Failure can be a fault
- Manufacturing fault leads to disk failure
- Disk failure is a fault that leads to database failure
- Database failure is a fault that leads to email service failure
TOTAL VS PARTIAL FAILURE

Total Failure:

All components in a system fail

→ Typical in nondistributed system

Partial Failure:

One or more (but not all) components in a distributed system fail

→ Some components affected
→ Other components completely unaffected
→ Considered as fault for the whole system
Categorising Faults and Failures

Types of Faults:

**Transient Fault**: occurs once then disappear

**Intermittent Fault**: occurs, vanishes, reoccurs, vanishes, etc.

**Permanent Fault**: persists until faulty component is replaced

Types of Failures:

**Process Failure**: process proceeds incorrectly or not at all

**Storage Failure**: “stable” secondary storage is inaccessible

**Communication Failure**: communication link or node failure
Failure Models

Crash Failure: a server halts, but works correctly until it halts

Fail-Stop: server will stop in a way that clients can tell that it has halted.

Fail-Resume: server will stop, then resume execution at a later time.

Fail-Silent: clients do not know server has halted

Omission Failure: a server fails to respond to incoming requests

Receive Omission: fails to receive incoming messages

Send Omission: fails to send messages
Response Failure: a server’s response is incorrect

Value Failure: the value of the response is wrong

State Transition Failure: the server deviates from the correct flow of control

Timing Failure: a server’s response lies outside the specified time interval

Arbitrary Failure: a server may produce arbitrary response at arbitrary times (aka Byzantine failure)
Failure Detector:

- Service that detects process failures
- Answers queries about status of a process

Reliable:

- Failed – crashed
- Unsuspected – hint

Unreliable:

- Suspected – may still be alive
- Unsuspected – hint
Synchronous systems:

→ Timeout
→ Failure detector sends probes to detect crash failures

Asynchronous systems:

✗ Timeout gives no guarantees
→ Failure detector can track *suspected* failures
→ Combine results from multiple detectors
✗ How to distinguish communication failure from process failure?
→ Ignore messages from suspected processes
✓ Turn an asynchronous system into a synchronous one
Fault Tolerance:

→ System can provide its services even in the presence of faults

Goal:

→ Automatically recover from partial failure
→ Without seriously affecting overall performance

Techniques:

→ Prevention: prevent or reduce occurrence of faults
→ Prediction: predict the faults that can occur and deal with them
→ Masking: hide the occurrence of the fault
→ Recovery: restore an erroneous state to an error-free state
Make sure faults don’t happen:

- Quality hardware
- Hardened hardware
- Quality software
Deal with expected faults:

- Test for error conditions
- Error handling code
- Error correcting codes
  - checksums
  - erasure codes
FAILURE MASKING

Try to hide occurrence of failures from other processes

Mask:
① Communication Failure → Reliable Communication
② Process Failure → Process Resilience
Redundancy:

- Information redundancy
- Time redundancy
- Physical redundancy

(a)

(b)
RELIABLE COMMUNICATION

- Communication channel experiences failure
- Focus on masking crash (lost/broken connections) and omission (lost messages) failures
Two Army Problem:

Non-faulty processes but lossy communication.

1 → 2 attack!
2 → 1 ack
2: did 1 get my ack?
1 → 2 ack ack
1: did 2 get my ack ack?
etc.

Consensus with lossy communication is impossible.

Why does TCP work?
**RELIABLE POINT-TO-POINT COMMUNICATION**

- Reliable transport protocol (e.g., TCP)
  - ✓ Masks omission failure
  - ✗ Not crash failure
Example: Failure and RPC:

Possible failures:

→ Client cannot locate server
→ Request message to server is lost
→ Server crashes after receiving a request
→ Reply message from server is lost
→ Client crashes after sending a request

How to deal with the various kinds of failure?
RELIABLE GROUP COMMUNICATION

(a) Receiver missed message #24

(b) ACK 25
**Scalability of Reliable Multicast**

**Feedback Implosion:** sender is swamped with feedback messages

**Nonhierarchical Multicast:**

- Use NACKs
- Feedback suppression: NACKs multicast to everyone
- Prevents other receivers from sending NACKs if they’ve already seen one.
  - Reduces (N)ACK load on server
  - Receivers have to be coordinated so they don’t all multicast NACKs at the same time
  - Multicasting feedback also interrupts processes that successfully received message
Hierarchical Multicast:

Sender

Coordinator

(Long-haul) connection

Local-area network

Receiver

Root

Scalability of Reliable Multicast
PROCESS RESILIENCE

Protection against process failures
Groups:

- Organise identical processes into groups
  - Process groups are dynamic
  - Processes can be members of multiple groups
  - Mechanisms for managing groups and group membership
- Deal with all processes in a group as a single abstraction

Flat vs Hierarchical Groups:

- Flat group: all decisions made collectively
- Hierarchical group: coordinator makes decisions
Create groups using replication

**Primary-Based:**
- Primary-backup
- Hierarchical group
- If primary crashes others elect a new primary

**Replicated-Write:**
- Active replication or Quorum
- Flat group
- Ordering of requests (atomic multicast problem)

**k Fault Tolerance:**
- can survive faults in $k$ components and still meet its specifications
- $k + 1$ replicas enough if fail-silent (or fail-stop)
- $2k + 1$ required if byzantine
STATE MACHINE REPLICAION
Client
lock()
Each replica executes as a state machine:

- \textit{state} + \textit{input} \rightarrow \textit{output} + \textit{new state}
- All replicas process same input in same order
- Deterministic: All \textit{correct} replicas produce same output
- Output from incorrect replicas deviates

Input Messages:

- All replicas agree on content of input messages
- All replicas agree on order of input messages
- \textbf{Consensus} (also called \textit{Agreement})

What can cause non-determinism?
**ATOMIC MULTICAST**

A message is delivered to either all processes, or none

Requires agreement about group membership

**Process Group:**

- Group view: view of the group (list of processes) sender had when message sent
- Each message uniquely associated with a group
- All processes in group have the same view
View Synchrony:

A message sent by a crashing sender is either delivered to all remaining processes (crashed after sending) or to none (crashed before sending).

Why? view changes and messages are delivered in total order.
Implementing View Synchrony:

**stable message**: a message that has been received by all members of the group it was sent to.

- Implemented using reliable point-to-point communication (TCP)
- Failure during multicast → only some messages delivered
Agreement

Examples: Election, transaction commit/abort, dividing tasks among workers, mutual exclusion

➔ Previous algorithms assumed no faults
➔ What happens when processes can fail?
➔ What happens when communication can fail?
➔ What happens when byzantine failures are possible

We want all nonfaulty processes to reach and establish agreement (within a finite number of steps)
Variants of the Agreement Problem

Consensus:
- each process proposes a value
- communicate with each other...
- all processes decide on same value
- for example, the maximum of all the proposed values

Interactive Consistency:
- all processes agree on a decision vector
- for example, the value that each of the processes proposed

Byzantine Generals:
- commander proposes a value
- all other processes agree on the commander’s value
Correctness of agreement:

**Termination** all processes eventually decide

**Agreement** all processes decide on the same value

**Validity**
- C the decided value was proposed by one of the processes
- IC the decided value is a vector that reflects each of the processes proposed values
- BG the decided value was proposed by the commander
CONSENSUS IN A SYNCHRONOUS SYSTEM

Assume:

- Execution in rounds
- Timeout to detect lost messages
Byzantine Generals Problem:

Reliable communication but faulty processes.

→ $n$ generals (processes)

→ $m$ are traitors (will send incorrect and contradictory info)

→ Need to know everyone else’s troop strength $g_i$

→ Each process has a vector: $\langle g_1, \ldots, g_n \rangle$

→ (Note: this is actually interactive consistency)
Byzantine Generals Impossibility:

If $m$ faulty processes then $2m + 1$ nonfaulty processes required for correct functioning
Byzantine agreement with Signatures:

- Digitally sign messages
- Cannot lie about what someone else said
- Avoids the impossibility result
- Can have agreement with 3 processes and 1 faulty
Consensus in an Asynchronous System

Assume:

- Arbitrary execution time (no rounds)
- Arbitrary message delays (can’t rely on timeout)
IMPOSSIBILITY OF CONSENSUS WITH ONE FAILURE

Impossible to guarantee consensus with $\geq 1$ faulty process

Proof Outline:

- Fischer, Lynch, Patterson (FLP) 1985
- The basic idea is to show circumstances under which the protocol remains forever indecisive
- Bivalent (any result is possible) vs univalent (only single result is possible) states
  1. There is always a bivalent start state
  2. Always possible to reach a bivalent state by delaying messages
     → no termination

In practice we can get close enough
Two Phase Commit:

- Original assumption: No failure

Failures can be due to:

- **Failure of communication channels:**
  - use timeouts

- **Server failures:**
  - potentially blocking
Two-phase commit with timeouts: Worker:

- **running**
  - NewServer
  - CanCommit: yes
  - DoCommit
  - DoAbort
- **uncertain**
  - timeout
  - GetDecision
- **committed**
  - DoCommit
- **aborted**
  - CanCommit: abort

→ 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
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?
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
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
**Example: Leader Election**

**Conceptually**

- Replica leader = A
- Replica leader = A
- Replica leader = A
- Replica leader = A
- Client leader(A)
- Client leader(B)

**With Paxos**

- Client propose(A)
  - leader = learn()
- Client propose(B)
  - leader = learn()

- Acceptor
- Acceptor
- Acceptor
- Acceptor
- Acceptor
- run_acceptor();
API:

val propose(proposed_val)
run_acceptor()
val learn()

Client: Proposer and Learner:

propose("A");
leader = learn();

Replica: Acceptor:

while(1) {
    run_acceptor();
}
Multi Paxos

→ Paxos allows you to agree on one value
→ But, typically need to choose multiple values
  • agree on values
  • agree on order of values
→ Run multiple instances of Paxos in sequence
→ Each instance to choose a single value
→ Add instance id to algorithm
→ Track completed instances
→ On failure, restart or join last completed instance +1
**Example: State Machine Replication**

**Conceptually**

![Conceptual Diagram]

- **Client op1**
- **Client op2**
- **Replica op2 op1**
- **Replica op2 op1**
- **Replica op2 op1**

**With Paxos**

![Paxos Diagram]

- **Proposer**
  - propose(op1);
  - propose(op2);

- **Learner**
  - op = learn();
  - op();

- **Acceptors**
  - run_acceptor();
  - Learner
  - op = learn();
  - op();
API:

val run_proposer(iid, proposed_val)
run_acceptor(iid)
val learn(iid)

Client:

while (1){
    ...
    send(leader, nextop);
    ...
}

Replica: Learner:

while(1) {
    op = learn(i++); exec_op(op);
}

Example: State Machine Replication
Replica: Proposer (leader):

while(1) {
    receive op
    do { chosen = run_proposer(i++, op); } while (chosen != op)
}

Replica: Acceptor:

while(1) {
    run_acceptor(i++);
}
PAXOS ALGORITHM: 3 PHASES

Assuming no failures

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

Phase 2: Accept:
① Accept: when ≥ \(N/2\) \textit{accept} replies, proposer sends value (as received from acceptor or arbitrary):
② Accepted: acceptors reply
   • \textit{accepted}. Remember accepted value.

Phase 3: Learn:
① Propagate value to Learners when ≥ \(N/2\) \textit{accepted} replies received.
**Simple Case**

**P1** | **A1** | **A2** | **A3** | **Learners**
---|---|---|---|---
propose(<p1,s1>,v)
promise(<p1,s1>,<nil,nil>)
accept(<p1,s1>,v)
accepted(<p1,s1>,v)
accepted(<p1,s1>,v)
accepted(<p1,s1>,v)
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 \(<seq, value>\) to \(\geq N/2\) acceptors
2. Promise: acceptors reply.
   - \textit{reject} if \(seq < seq\) of previously accepted value
   - else \textit{accept} (include last accepted value). \emph{promised} = \(seq\).

Phase 2: Accept:

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

Phase 3: Learn:

1. Propagate value to Learners when \(\geq N/2\) \emph{accepted} received.
ACCEPTR FAILS

☑️ As long as a quorum still available

→ Restart: Must remember last accepted value(s)

<table>
<thead>
<tr>
<th>P1</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>Learners</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>propose(&lt;p1,s1&gt;,v)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>promise(&lt;p1,s1&gt;,&lt;nil,nil&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accept(&lt;p1,s1&gt;,v)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accepted(&lt;p1,s1&gt;,v)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>accepted(&lt;p1,s1&gt;,v)</td>
</tr>
</tbody>
</table>
PROPOSER FAILS

→ Elect a new leader
→ Continue execution
✔ New proposer will choose any previously accepted value

propose(<p1,s1>,v1)
promise(<p1,s1>,<nil,nil>)
propose(<p2,s1>,v2)
promise(<p2,s1>,<nil,nil>)
accept(<p2,s1>,v2)
accepted(<p2,s1>,v2)

P2
A1
A2
A3
Learners

P1
P1 A1 A2 A3 Learners

accept(<p2,s1>,v1)
promise(<p2,s1>,<nil,nil>)
propose(<p1,s1>,v1)
accept(<p1,s1>,v1)
promise(<p2,s1>,<nil,nil>)

P ROPOSER
F AIL S
61
MULTIPLE PROPOSERS

→ For example: crashed proposer returns and continues
   ✗ Dueling 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:

\[ \begin{align*} 
\rightarrow & \quad P_1 \text{ fails } \rightarrow \text{ roll back: } P_1 \sim R_{13} \\
\rightarrow & \quad P_2 \text{ fails } \rightarrow P_2 \sim R_{22} \\
& \quad \text{Orphan message } m \text{ is received but not sent } \rightarrow P_1 \sim R_{12} \\
\rightarrow & \quad P_3 \text{ fails } \rightarrow P_3 \sim R_{32} \rightarrow P_2 \sim R_{21} \rightarrow P_1 \sim R_{11}, P_3 \sim R_{31} 
\end{align*} \]

Messaging dependencies plus independent checkpointing may force system to roll back to initial state
Message Loss:

- Failure of $P_2 \rightarrow P_2 \sim R_{21}$
- Message $m$ is now recorded as sent (by $P_1$) 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:

\[ P_2 \downarrow \rightarrow P_2 \bowtie R_{21} \rightarrow P_1 \bowtie R_{11}. \text{ Note: } n_1 \text{ in transit} \]

\[ \rightarrow \text{ Pre-rollback message } n_1 \text{ is received after rollback} \]
\[ \rightarrow \text{ Forces another rollback } P_2 \bowtie R_{21}, P_1 \bowtie R_{11}, \text{ can repeat indefinitely} \]
CONSISTENT CHECKPOINTING

Consistent Cut:

![Diagram of consistent checkpointing with nodes labeled P1, P2, and P3 and arrows indicating the cut points cut 1 and cut 2.](image-url)
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

$$\{R_{12}, R_{22}, R_{32}\}$$ 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

How to take a consistent checkpoint?:

→ Simple solution: Each process checkpoints immediately after sending a message
   - High overhead

→ Reducing this to checkpointing after $n$ messages, $n > 1$, is **not** guaranteed to produce a consistent checkpoint!
→ Require some coordination during checkpointing
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 $P_i$ takes tentative checkpoint
2. $P_i$ sends $t$ message to all other processes $P_j$ to take tentative checkpoint
3. $P_j$ reply to $P_i$ whether succeeded in taking tentative checkpoint
4. $P_i$ receives *true* reply from each $P_j$ → decides to make permanent
   - $P_i$ receives at least one *false* → 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 $P_i$
REDUNDANT CHECKPOINTS

Algorithm performs unnecessary checkpoints

→ \( \{R_{11}, R_{21}, R_{31}\} \) form a (strongly) consistent checkpoint
→ Checkpoint \( \{R_{12}, R_{22}, R_{32}\} \) initiated by \( P_1 \) is strongly consistent
→ \( R_{32} \) is redundant, as \( \{R_{12}, R_{22}, R_{31}\} \) is consistent
ROLLBACK RECOVERY

First Phase:

① Coordinator sends “r” messages to all other processes to ask them to roll back
② Each process replies $true$, unless already in checkpoint or rollback
③ If all replies are $true$, coordinator decides to roll back, otherwise continue

Second Phase:

① Coordinator sends decision to other processes
② Processes receiving this message perform corresponding action
HOMEWORK

→ Find a Paxos library and implement a replicated state machine using it.

Hacker’s edition:

→ Implement the Paxos library (e.g., in Erlang).
 optional

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