Slide 1

DISTRIBUTED SYSTEMS (COMP9243)

Lecture 8: Fault Tolerance

① Failure
② Reliable Communication
③ Process Resilience
④ Recovery

Slide 2

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.

Slide 3

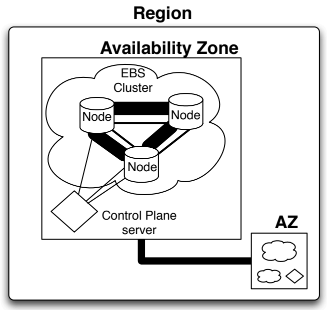
CASE STUDY: AWS FAILURE 2011

- 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.*
- `http://aws.amazon.com/message/65648/`

Slide 4

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?:

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

Slide 5

Solution:

- Disconnect bad cluster
- Throttle re-mirroring
- Add more disk space
- Slowly un-throttle re-mirroring
- Volumes unstuck → reconnect cluster
- 0.07% data lost

Lessons learned:

- Back off
- Re-establish connectivity to previous replicas
- Shorter timeouts
- Snapshot stuck volumes
- CP: one AZ shouldn't crash another AZ
- Make it easier to use multiple AZs

Slide 6

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

Slide 7

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

Slide 8

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: secondary storage is inaccessible

Communication Failure: communication link or node failure

Slide 9

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

Slide 10

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*)

Slide 11

DETECTING 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

Slide 12

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

Slide 13

Fault Tolerance

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
- ➜ **Masking**: hide the occurrence of the fault
- ➜ **Prediction**: predict the faults that can occur and deal with them
- ➜ **Recovery**: restore an erroneous state to an error-free state

Slide 14

Failure Prevention

Make sure faults don't happen:

- ➜ Quality hardware
- ➜ Hardened hardware
- ➜ Quality software

Slide 15

Failure Prediction

Deal with expected faults:

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

Slide 16

Failure Masking

Try to hide occurrence of failures from other processes

Mask:

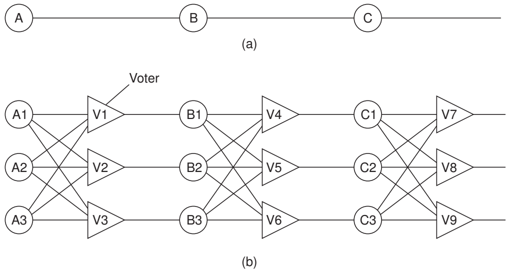
① Communication Failure → Reliable Communication

② Process Failure → Process Resilience

Slide 17

Redundancy:

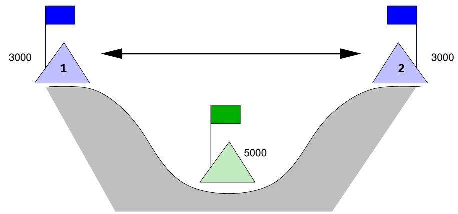
- ➜ Information redundancy
- ➜ Time redundancy
- ➜ Physical redundancy



Slide 19

Two Army Problem:

Non-faulty processes but lossy communication.



- ➜ $1 \rightarrow 2$ attack!
- ➜ $2 \rightarrow 1$ ack
- ➜ 2: did 1 get my ack?
- ➜ $1 \rightarrow 2$ ack ack
- ➜ 1: did 2 get my ack ack?
- ➜ etc.

Consensus with lossy communication is impossible. Why does TCP work?

Slide 18

Reliable Communication

- ➜ Communication channel experiences failure
- ➜ Focus on masking crash (lost/broken connections) and omission (lost messages) failures

Slide 20

Reliable Point-to-Point Communication

- ➜ Reliable transport protocol (e.g., TCP)
  - ✓ Masks omission failure
  - ✗ Not crash failure

Slide 21

Example: Failure and RPC:

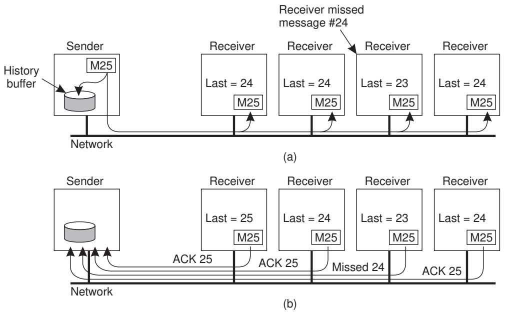
Possible failures:

➜ 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?

Slide 22

Reliable Group Communication



Slide 23

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.

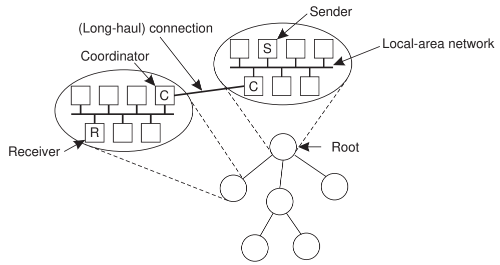
✓ Reduces (N)ACK load on server

✗ Receivers have to be coordinated so they don't all multicast NACKs at same time

✗ Multicasting feedback also interrupts processes that successfully received message

Slide 24

Hierarchical Multicast:



Slide 25

Process Resilience

Protection against process failures

Slide 26

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

Slide 27

Replication

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)

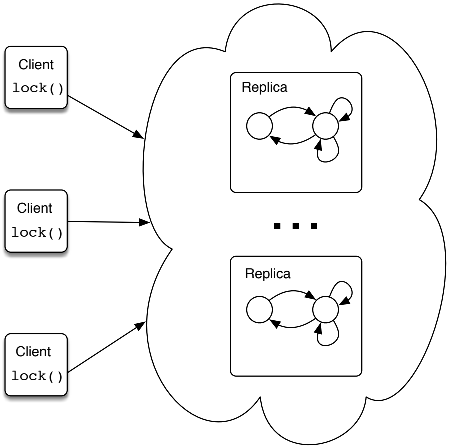
k Fault Tolerance:

- ➜ 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 if byzantine

Slide 28

State Machine Replication

Slide 29



Slide 30

Each replica executes as a state machine:

- *state + input -> output + new state*
- All replicas process same input in same order
- Deterministic: All 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
- Consensus (also called Agreement)

What can cause non-determinism?

Slide 31

ATOMIC MULTICAST

A message is delivered to either all processes, or none

Requires agreement about group membership

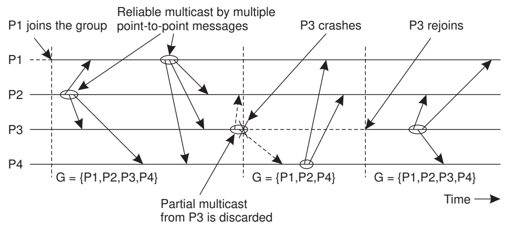
Process Group:

- 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

Slide 32

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).



- view changes and messages are delivered in total order Why?

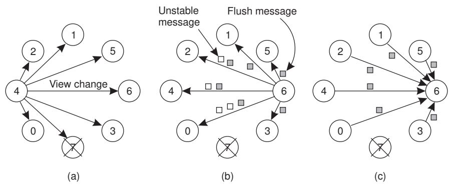
Implementing View Synchrony:

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

Slide 33



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

Slide 34

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

Slide 35

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

Slide 36

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

Slide 37

Assume:

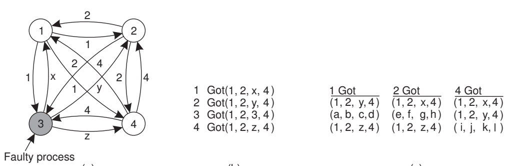
- ➜ Execution in rounds
- ➜ Timeout to detect lost messages

Slide 38

Byzantine Generals Problem:

Reliable communication but faulty processes.

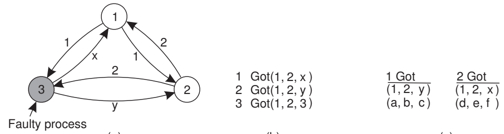
- ➜ n generals (processes)
- ➜ m are traitors (will send incorrect and contradictory info)
- ➜ Need to know everyone else's troop strength g_i
- ➜ Each process has a vector: $\langle g_1, ...g_n \rangle$
- ➜ (Note: this is actually interactive consistency)



(a) (b) (c)

Slide 39

Byzantine Generals Impossibility:



(a) (b) (c)

- ➜ If m faulty processes then $2m + 1$ nonfaulty processes required for correct functioning

Slide 40

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

Slide 41

Consensus in an Asynchronous System

Assume:

- ➜ Arbitrary execution time (no rounds)
- ➜ Arbitrary message delays (can't rely on timeout)

Slide 42

Impossibility of Consensus with One Failure

Impossible to guarantee consensus with ≥ 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

$\rightarrow$ no termination

In practice we can get close enough

Slide 43

Consensus in Practice

Two Phase Commit:

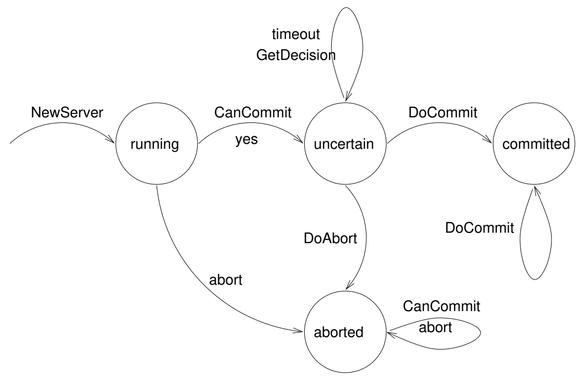
- ➜ Original assumption: No failure

Failures can be due to:

- ➜ **Failure of communication channels:**
  - use timeouts
- ➜ **Server failures:**
  - potentially blocking

Slide 44

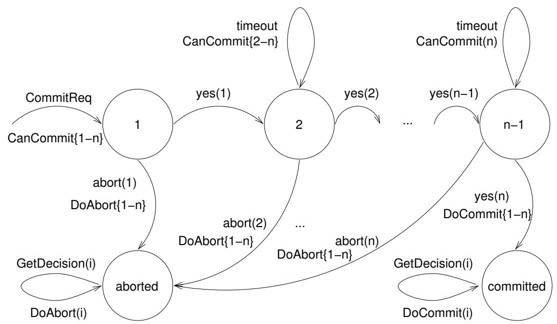
Two-phase commit with timeouts: Worker:



- ➜ On *timeout* sends `GetDecision`.

Slide 45

Two-phase commit with timeouts: Coordinator:



➜ On *timeout* re-sends `CanCommit`, On *GetDecision* repeats decision.

Slide 46

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

Slide 47

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?

Slide 48

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

Slide 49

Leader Election

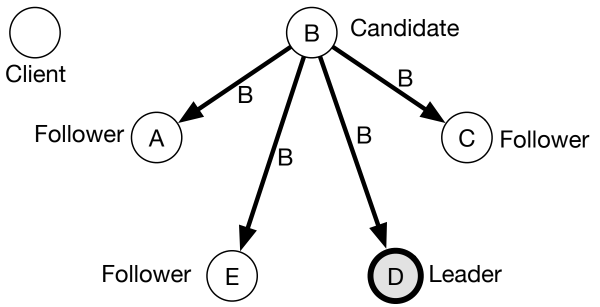
Term the time during which a node is leader

Candidate node who wants to become leader

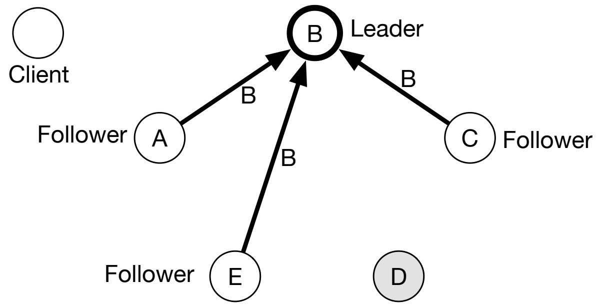
Failed Leader:

- ➜ 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 50



Slide 51



Slide 52

Candidate:

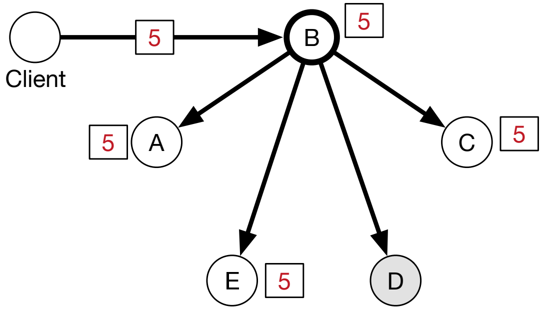
- ➜ Detects that leader has failed
- ➜ Sends *Request Vote* to all other nodes
- ➜ Nodes reply

  Yes hasn't voted yet this term

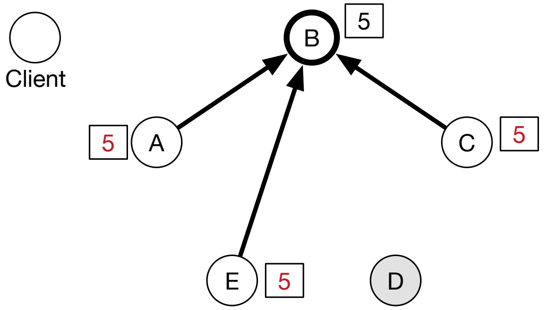
  No has already voted this term
- ➜ Majority votes -> candidate becomes leader
- ➜ or Timeout -> new election

Slide 53

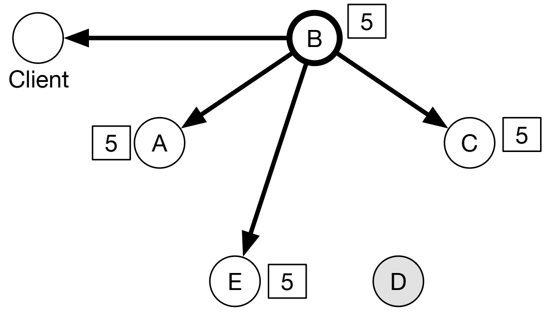
Log Replication



Slide 54



Slide 55



Slide 56

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

Slide 57

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

Slide 58

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

Slide 59

Paxos Algorithm: 3 Phases

Assuming no failures

Phase 1: Propose:

① Propose: send a proposal *<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

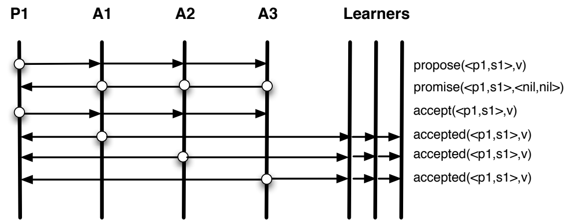
- *accepted*. Remember accepted value.

Phase 3: Learn:

① Propagate value to Learners when $\geq N/2$ *accepted* replies received.

Slide 60

Simple Case



Slide 61

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

Slide 62

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.
- *reject* if seq < seq of previously accepted value
- else *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
- *reject* if seq < promised.
- else *accepted*. Remember accepted value.

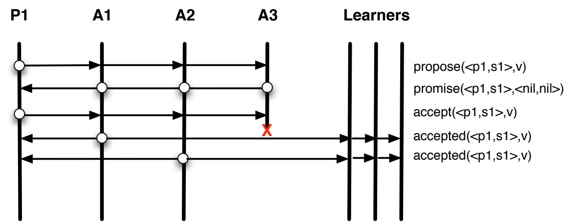
Phase 3: Learn:

① Propagate value to Learners when $\geq N/2$ *accepted* received.

Slide 63

Acceptor Fails

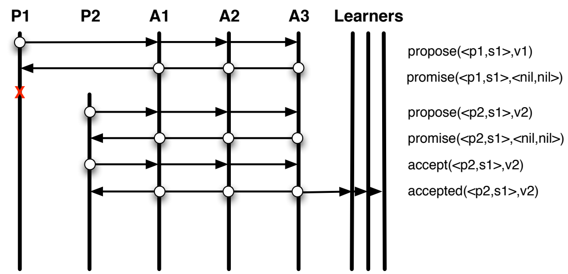
☑ As long as a quorum still available
➜ Restart: Must remember last accepted value(s)



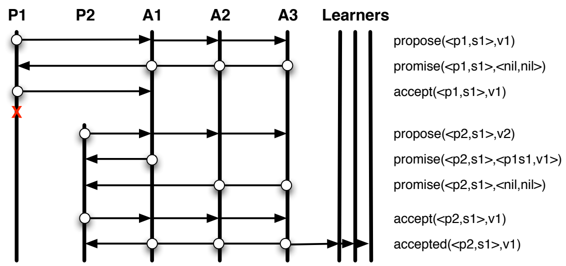
Slide 64

Proposer Fails

➜ Elect a new leader
➜ Continue execution
☑ New proposer will choose any previously accepted value



Slide 65



Slide 66

Multiple Proposers

- ➜ For example: crashed proposer returns and continues
- ✗ Dueling proposers
- ✗ No guaranteed termination
- ☑ Heuristics to recognise situation and back off

Slide 67

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.

Slide 68

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

Slide 69

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)

Slide 70

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

Slide 71

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

Slide 72

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

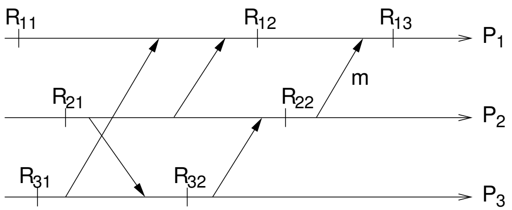
Slide 73

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*

Slide 74

Domino Effect:

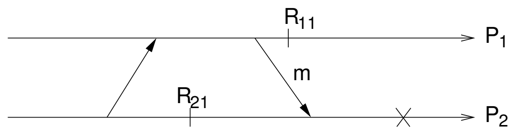


- ➜ P_1 fails → roll back: $P_1 \curvearrowright R_{13}$
- ➜ P_2 fails → $P_2 \curvearrowright R_{22}$
  Orphan message m is received but not sent → $P_1 \curvearrowright R_{12}$
- ➜ P_3 fails → $P_3 \curvearrowright R_{32}$ → $P_2 \curvearrowright R_{21}$ → $P_1 \curvearrowright R_{11}, P_3 \curvearrowright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

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

Slide 75

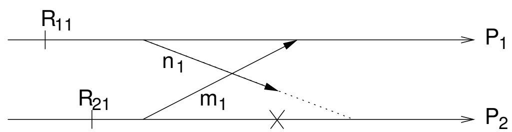
Message Loss:



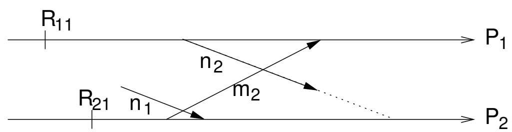
- ➜ Failure of P_2 → $P_2 \curvearrowright R_{21}$
- ➜ Message m is now recorded as sent (by P_1) but not received (by P_2), and m will never be received after rollback
- ➜ Message m is *lost*
- ➜ Whether m is lost due to rollback or due to imperfect communication channels is indistinguishable!
- → Require protocols resilient to message loss

Slide 76

Livelock:



$P_2\Downarrow$ → $P_2 \curvearrowright R_{21}$ → $P_1 \curvearrowright R_{11}$. Note: n_1 in transit

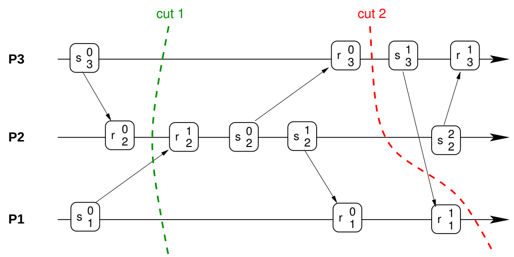


- ➜ Pre-rollback message n_1 is received after rollback
- ➜ Forces another rollback $P_2 \curvearrowright R_{21}, P_1 \curvearrowright R_{11}$, can repeat indefinitely

Slide 77

Consistent Checkpointing

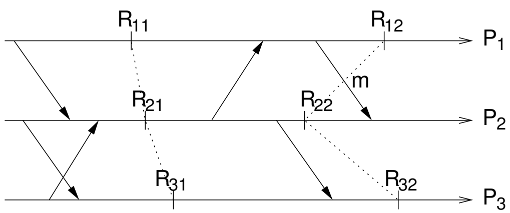
Consistent Cut:



Slide 78

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

Slide 79

- ➜ **Strongly consistent checkpointing** requires quiescent system
  $\rightarrow$ 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!
- $\rightarrow$ Require some coordination during checkpointing

Slide 80

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

Slide 81

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

Slide 82

Second Phase:

① Coordinator P_i informs all other processes P_j of decision

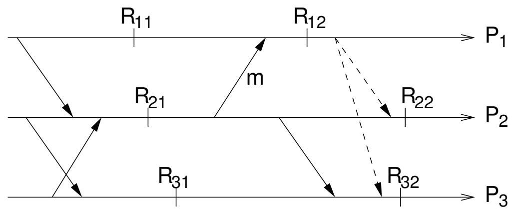
② P_j convert or discard tentative checkpoints accordingly

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

Slide 83

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

Slide 84

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

Slide 85

- ➜ Look up a recent failure of a large distributed system. What went wrong? How could the problem has been avoided? What lessons were learned?
- ➜ 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

Slide 86

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.