Slide 1

## Distributed Systems (COMP9243)

### Lecture 7: 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

### Failure

Terminology:

**Failure**: a system fails when it fails to 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 4

### 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 5

## 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

Slide 6

## 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

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

Slide 7

**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

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

Slide 8

## 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

➜ 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

Slide 9

## Failure Prevention

Make sure faults don't happen:

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


Slide 10

## Failure Prediction

Deal with expected faults:

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



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

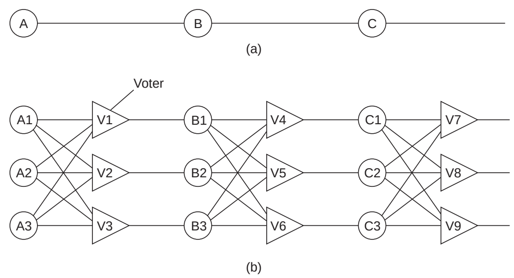
Slide 13

## FAILURE MASKING

Try to hide occurrence of failures from other processes

Redundancy:

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

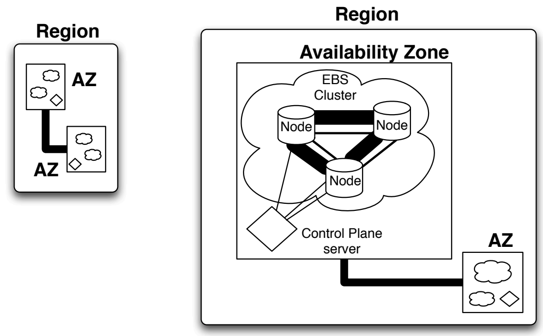



Slide 14

## 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 15




Slide 16

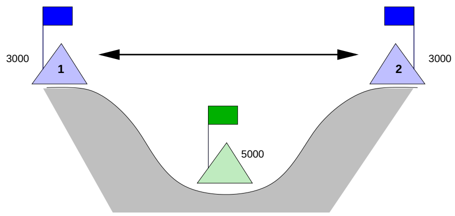
## RELIABLE COMMUNICATION

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

**Slide 17**

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 Point-to-Point Communication

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

**Slide 19**

Example: Failure and RPC:

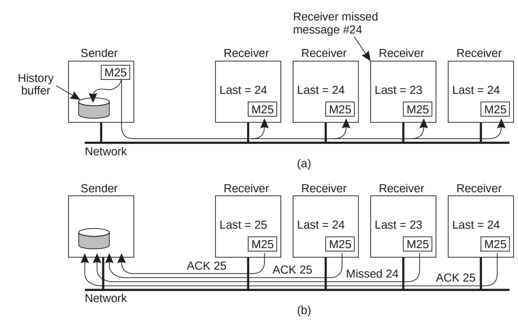
Possible failures:

- ➜ 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 20**

## Reliable Group Communication



Slide 21

## Scalability of Reliable Multicast

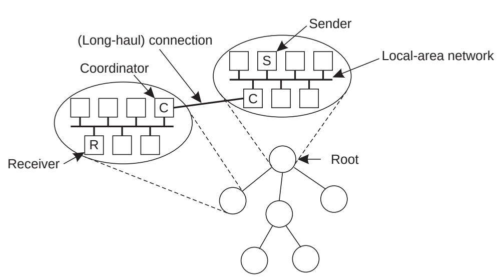
**Feedback Implosion:** sender is swamped with feedback messages

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 22

Hierarchical Multicast:






Slide 23

## 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

Slide 24

## 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 25

## State Machine Replication

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)

Slide 26

## Atomic Multicast

A message is delivered to either all processes, or none

Requires agreement about group membership

Process Group:

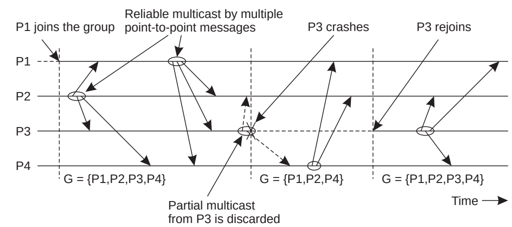
- ➜ Group view: view of the group (list of processes) sender had when message sent
- ➜ Each message uniquely associated with a group
- ➜ All processes in group have the same view



Slide 27

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).



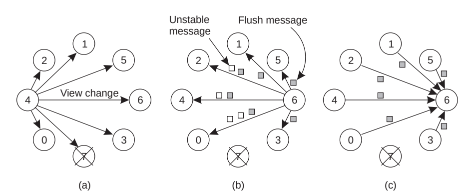

- ➜ view changes and messages are delivered in total order

Slide 28

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 29

## 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)

Slide 30

## 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



Slide 31

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

Slide 32

## Consensus in a Synchronous System

Assume:

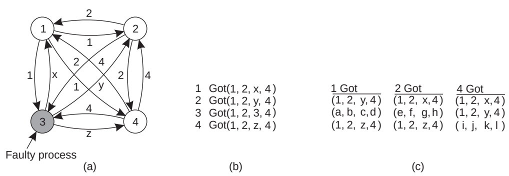
- ➜ Execution in rounds
- ➜ Timeout to detect lost messages

**Slide 33**

## Byzantine Generals Problem:

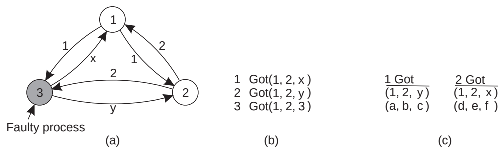
Reliable communication but faulty processes.

- ➜ $n$ 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)




**Slide 34**

## Byzantine Generals Impossibility:




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



**Slide 35**

## 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 36**

# CONSENSUS IN AN ASYNCHRONOUS SYSTEM

## Assume:

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

Slide 37

## 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 vs univalent 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 38

## 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 39

## Paxos Algorithm: 3 Phases

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

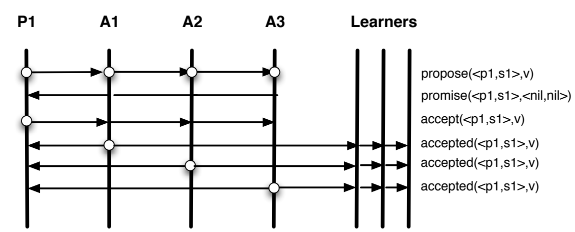
- *reject* if seq $<$ promised.
- else *accepted*. Remember accepted value.

Phase 3: Learn:

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

Slide 40

## Simple Case



Slide 41

## Failures

Failure Model:

channel : lose, reorder, duplicate message

process : crash (fail-stop, fail-resume)

Failure Cases:

① Acceptor fails

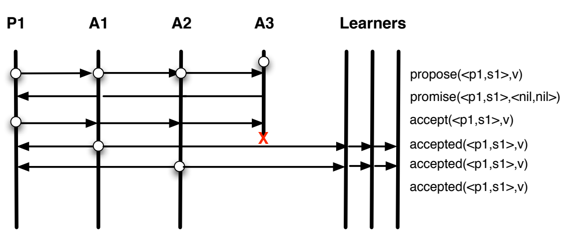
② Proposer fails

③ Multiple proposers

Slide 42

## Acceptor Fails

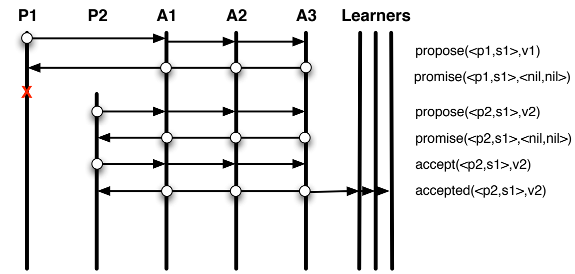
- ☑ As long as a quorum still available
- ➜ Restart: Must remember last accepted value(s)



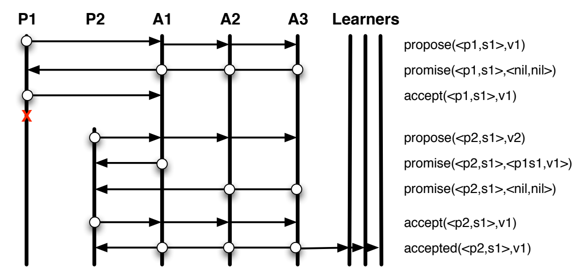

Slide 43

## Proposer Fails

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




Slide 44



Slide 45

## Multiple Proposers

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

Slide 46

## Multi Paxos

- ➜ 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 competed instances
- ➜ On failure, restart or join *last completed instance +1*



Slide 47

## Using Paxos

Use Paxos for:

- ➜ Total order multicast: order messages
- ➜ State machine replication: order operations
- ➜ Leader election: choose a leader id
- ➜ Replicated storage: order writes
- ➜ View synchrony: order view changes

Slide 48

## Example: State Machine Replication

API:

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

Client:

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

Slide 49

Replica: Proposer:

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

Replica: Acceptor:

```
while(1) {
  run_acceptor(i++);
}
```

Replica: Learner:

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

Slide 50

## 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 51

## 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 52

## 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 53

## 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
- 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 54

Operation-Based Recovery - Logging:

Update in-place together with write-ahead logging

- ➜ Every change (update) of data is recorded in a log, which includes:
  - Data item name (for identification)
  - Old data item state (for *undo*)
  - New data item state (for *redo*)
- ➜ Undo log is written *before* update (write-ahead log).
- → Transaction semantics



Slide 55

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 56

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

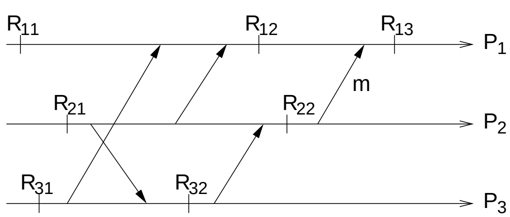
**Slide 57**

## 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 58**

Domino Effect:



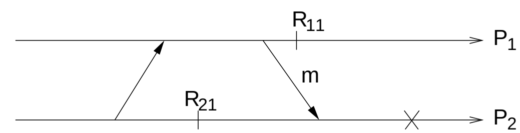

- ➜ $P_1$ fails → roll back: $P_1 \curvearrowright R_{13}$
- ➜ $P_2$ fails → $P_2 \curvearrowright R_{22}$
  *Orphan message* $m$ is received but not sent → $P_1 \curvearrowright R_{12}$
- ➜ $P_3$ fails → $P_3 \curvearrowright R_{32} \to P_2 \curvearrowright R_{21} \to P_1 \curvearrowright R_{11}, P_3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

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



**Slide 59**

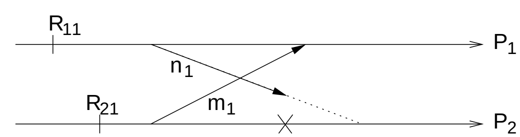
Message Loss:



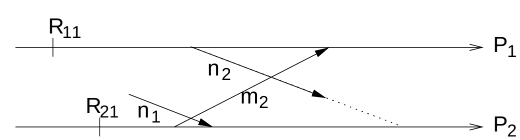

- ➜ Failure of $P_2 \to P_2 \curvearrowright R_{21}$
- ➜ Message $m$ is now recorded as sent (by $P_1$) but not received (by $P_2$), and $m$ will never be received after rollback
- ➜ Message $m$ is *lost*
- ➜ Whether $m$ is lost due to rollback or due to imperfect communication channels is indistinguishable!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

**Slide 60**

Livelock:




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

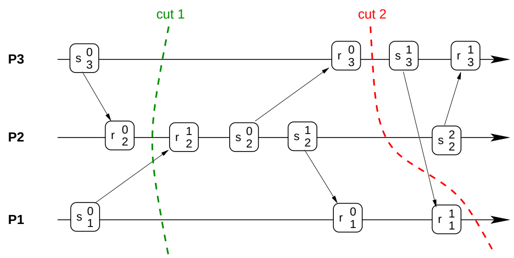



- ➜ Pre-rollback message $n_1$ is received after rollback
- ➜ Forces another rollback $P_2 \curvearrowright R_{21}, P_1 \curvearrowright R_{11}$, can repeat indefinitely

Slide 61

## Consistent Checkpointing

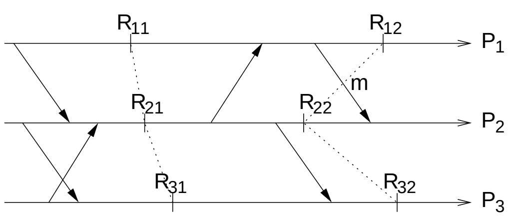
Consistent Cut:




Slide 62

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 63

- ➜ **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

Slide 64

## 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 65

## 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 $P_j$ $\rightarrow$ decides to make permanent
$P_i$ receives at least one $false$ $\rightarrow$ decides to discard the tentative checkpoints

Slide 66

Second Phase:

① Coordinator $P_i$ informs all other processes $P_j$ of decision

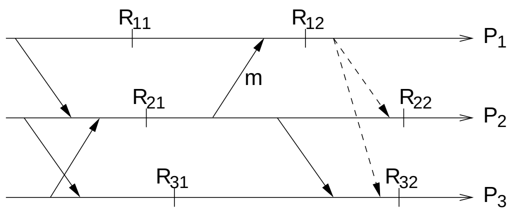
② $P_j$ convert or discard tentative checkpoints accordingly

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

Slide 67

## 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 68

## AVOIDING REDUNDANT CHECKPOINTS

Keep track of messages sent to avoid redundant checkpoints

- ➜ Associate each message $m$ with *label* $m.l$, incremented at each message
- ➜ Each process maintains three vectors:
  - $last\_rec_i[j] := m.l$, where $m$ is last msg $m$ received by $P_i$ from $P_j$ since last checkpoint ($last\_rec_i[j] = 0$ if none)
  - $first\_sent_i[j] := m.l$, where $m$ is first msg $m$ sent by $P_i$ to $P_j$ since last checkpoint ($first\_sent_i[j] = 0$ if none)
  - $cohort_i := \{j | last\_rec_i[j] > 0\}$, set of processes from which $P_i$ has received a message since last checkpoint
- ➜ $P_j$ only needs to take a checkpoint after receiving a control message $t$ ("take tentative checkpoint") from $i$ if $last\_rec_i[j] \geq first\_sent_j[i] > 0$

Slide 69

## Checkpointing Algorithm

Messages:

➜ $t$: take tentative, $p$: make permanent, $u$: undo checkpoint

Initialisation: Each $P$ sets $OK := true$, $first\_sent = \{0, 0, \cdots, 0\}$

Coordinator, $P_i$:

① $\text{send}(t, i, last\_rec_i[j])$ to all $P_j \in cohort_i$

② **if** all replies are $true$, $\text{send}(p)$ to all $P_j \in cohort_i$
**else** $\text{send}(u)$ to all $P_j \in cohort_i$

Slide 70

Others, $P_j$: upon receiving $(t, i, last\_rec_i[j])$

① **if** $OK_j$ **and** $last\_rec_i[j] \geq first\_sent_j[i] > 0$

② take tentative checkpoint

③ $\text{send}(t, j, last\_rec_j[k])$ to all $P_k \in cohort_j$

④ **if** all replies are $true$, $OK := true$ **else** $OK := false$

⑤ $\text{send}(OK, j)$ to $i$

Others, $P_j$: upon receiving message $x \in \{p, u\}$ from $P_i$:

① **if** $x = p$ make permanent **else** discard tentative checkpoint

② $\text{send}(x, j)$ to all $P_k \in cohort_j$

**Note:** $O(n^2)$ messages



Slide 71

## 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

Slide 72

## Redundant Rollbacks

Processes may roll back unnecessarily

➜ Can be avoided by keeping track of messages received

Avoiding Redundant Rollbacks:

➜ Message labelling as before

➜ Two additional vectors:

  ➜ $last\_sent_i[j] := m.l$, where $m$ is *last* msg $m$ sent by $i$ to $j$ since last checkpoint ($last\_sent_i[j] = \infty$ if none)

  ➜ $r\_cohort_i := \{j | i$ communicates with $j\}$

➜ $P_i$ only needs to roll back after receiving message $(r, j, last\_sent_j[i])$ if $last\_rec_i[j] > last\_sent_j[i]$

Slide 73

## Rollback Recovery Algorithm

Messages:

- $r$: rollback request, $d$: do rollback, $c$:continue

Initialisation: Each $P$ sets

- $resume := true$
- $last\_rec = \{\infty, \infty, \cdots, \infty\}$
- $W$ according to willingness to roll back

Initiator, $P_i$:

① $\mathrm{send}(r, i, last\_sent_i[j])$ to all $P_j \in r\_cohort_i$

② **if** all replies are $true$, $\mathrm{send}(d)$ to all $P_j \in cohort_i$
**else** $\mathrm{send}(c)$ to all $P_j \in cohort_i$

Slide 74

Others, $P_j$: upon receiving $(r, i, last\_sent_i[j])$

① **if** $W_j$ **and** $last\_rec_j[i] > last\_sent_i[j]$ **and** $resume_j$:

② $resume_j = false$,

③ $\mathrm{send}(r, j, last\_sent_j[k])$ to all $P_k \in r\_cohort_j$,

④ **if** all replies are $true$, $W_j := true$ **else** $W_j := false$,

⑤ $\mathrm{send}(W_j, j)$ to $i$.

Others, $P_j$: upon receiving message $x \in \{c, d\}$ from $P_i$:

① **if** $x = d$ roll back **else** continue,

② $\mathrm{send}(x, j)$ to all $P_k \in r\_cohort_j$.



Slide 75

## Asynchronous Checkpointing

Let processes checkpoint independently (unsynchronised) and construct a consistent state during recovery.

- Source of inconsistencies are *orphan messages*.
- Consistent state can be obtained by:
  ① Restarting failed process from latest checkpoint, and
  ② *rolling forward* the restarted process past the point where the last message was sent prior to failure
- All $\mathrm{send}$ operations during roll-forward are suppressed.
- Except for timing, the result is indistinguishable from restarting from a (non-existent) checkpoint taken after the last $\mathrm{send}$.
- Works as long as no message was lost.

Slide 76

## Message Logging

- Suppressing outgoing messages during roll-forward requires knowledge of the number of messages the failed process had sent prior to failure.
  - *Log* the send count in *stable storage*.
- Any attempted receive of a lost message will terminate roll-forward.
  - Log all *incoming* messages in stable storage
  - During roll-forward replay incoming messages from log

Slide 77

Problems with Message Logging:

*Roll-forward* assumes *deterministic behaviour* of all processes

➜ Possible dependence of behaviour on uncontrollable factors (resident set).

➜ Possible inconsistencies between process IDs (different ID after restart!)

➜ Interrupt processing imposes time constraint, interrupts are *asynchronous* wrt. program messages.

- May need to checkpoint before handling any interrupt!

➜ Multithreaded processes introduce a degree of non-determinism.

Require careful implementation and appropriate OS support.

Slide 78

## Optimistic Checkpointing

Asynchronous Logging:

➜ Log messages to volatile memory

➜ Flush to stable store asynchronously

➜ On failure, unflushed log is lost, resulting in inconsistent state

➜ Construct consistent state by rolling back orphan processes

Do synchronisation required between checkpointing and logging



Slide 79

Assumptions:

Communication Channels

➜ Reliable

➜ Ordered (FIFO)

➜ Infinite buffer size

➜ Finite (but arbitrary) delays

Event-Driven Computation

➜ Event is receipt of message

➜ Process waits for message

➜ Upon receipt sends 0 or more messages

➜ Each event logged to volatile storage as $\{e, m, msgs\_sent\}$, where $msgs\_sent$ is set of messages sent

➜ Log is flushed to stable storage asynchronously.

Slide 80

## Optimistic Checkpoint Recovery

➜ Each process $P_i$ keeps track of:

- $n\_rcvd_{i \leftarrow j}(E)$: # messages received from $P_j$ (up to event $E$),
- $n\_sent_{i \rightarrow j}(E)$: # messages sent to $P_j$ (up to event $E$).

➜ Upon restart, compare local message count with that of neighbours:

- neighbour $P_j$ is orphan if $n\_rcvd_{j \leftarrow i} > n\_sent_{i \rightarrow j}$,
- must roll back until $n\_rcvd_{j \leftarrow i} \leq n\_sent_{i \rightarrow j}$.

➜ $P_j$'s rollback may orphan other processes ($\rightarrow$ domino rollback).

## Optimistic Checkpoint Recovery Algorithm

Slide 81

Recovery is initiated by restarting process broadcasting its failure.

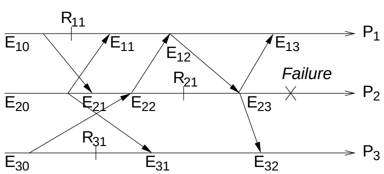
- ➜ restarting process $P_i$ sets
  $E_i :=$ *latest event on stable log*,
- ➜ process $P_j$ receiving failure messages sets
  $E_j :=$ *latest event that happened at* $P_j$.

**Processor** $P_i$ (initiator or not) performs following steps:

**for** $k := 1$ **to** $N$ **do** /* $N$ *is number of processes* */
  **for** each neighbour $j$ **do**
    $\text{send}(r, i, n\_sent_{i \to j}(E_i))$;
  **wait** for $r$ messages from all neighbours;
  **for** each message $(r, j, s)$ received **do**
    **if** $n\_rcvd_{i \leftarrow j}(E_i) > s$ **then** /* *have orphan* */
      $E_i :=$ *latest* $E$ *suchthat* $n\_rcvd_{i \leftarrow j}(E) = s$;

Slide 82

Example:




① $P_2 : recover\,from R_{21};\ E_2 := E_{22};\ \text{send}(r, P_2, 2) \to P_1;\ \text{send}(r, P_2, 1) \to P_3$;

② $P_1 \leftarrow P_2; E_1 := E_{13};\ n\_rcvd_{1 \leftarrow 2}(E_{13}) = 3 > 2 :\ E_1 := E_{12};\ \text{send}(r, P_1, 2) \to P_2$;

③ $P_3 \leftarrow P_2;\ E_3 := E_{32};\ n\_rcvd_{3 \leftarrow 2}(E_{32}) = 2 > 1 :\ E_3 := E_{31};\ \text{send}(r, P_3, 1) \to P_2$;

④ $P_2 \leftarrow P_1;\ n\_rcvd_{2 \leftarrow 1}(E_{22}) = 1 \leq 2$; *no change*; $\text{send}(r, P_2, 2) \to P_1$;

⑤ $P_2 \leftarrow P_3;\ n\_rcvd_{2 \leftarrow 3}(E_{22}) = 1 \leq 1$; *no change*; $\text{send}(r, P_2, 1) \to P_3$;

Recovery state $\{E_{12}, E_{22}, E_{31}\}$, roll back to $\{R_{11}, R_{21}, R_{31}\}$.



## Reading List

Slide 83

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