

---

## DISTRIBUTED SYSTEMS (COMP9243)

### Lecture 7: Fault Tolerance

#### Slide 1

- ① Failure
  - ② Process Resilience
  - ③ Reliable Communication
  - ④ Recovery
- 
- 

### DEPENDABILITY

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

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

**Slide 2 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.

---

---

### 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)

#### Slide 3

**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

#### Slide 4

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
-

---

## FAULT TOLERANCE

Fault Tolerance:

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

Goal:

→ Automatically recover from partial failure

**Slide 5** → 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

---

## CATEGORISING FAULTS AND FAILURES

Types of Faults:

**Transient Fault:** occurs once then disappear

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

**Slide 6** **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.

**Slide 7** **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

---

**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

**Slide 8**

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

---

---

## FAILURE PREVENTION

Make sure faults don't happen:

- Quality hardware
- Hardened hardware
- Quality software

Slide 9



---

## FAILURE PREDICTION

Deal with expected faults:

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

Slide 10



---

## DETECTING FAILURE

Failure Detector:

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

Slide 11

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:

Slide 12

- ✗ 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
-

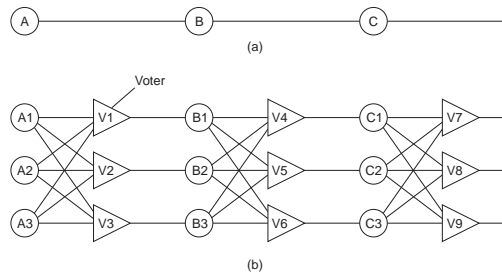
## FAILURE MASKING

Try to hide occurrence of failures from other processes

Redundancy:

- Information redundancy
- Time redundancy
- Physical redundancy

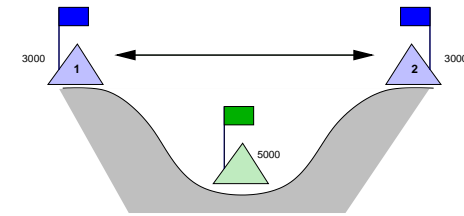
Slide 13



Two Army Problem:

Non-faulty processes but lossy communication.

Slide 15



- 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 COMMUNICATION

Slide 14

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

## RELIABLE POINT-TO-POINT COMMUNICATION

Slide 16

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

Slide 17

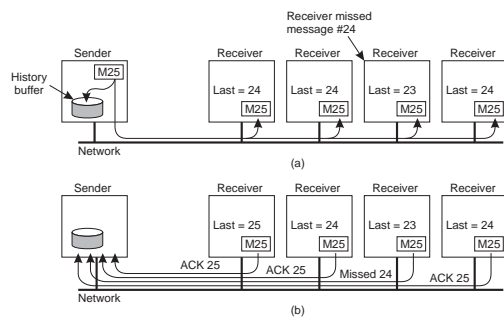
Example: Failure and RPC:

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



Slide 18

### SCALABILITY OF RELIABLE MULTICAST

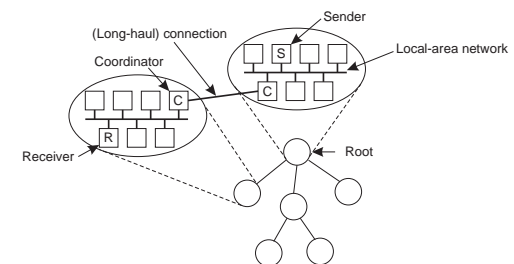
**Feedback Implosion:** sender is swamped with feedback messages

**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 19

**Hierarchical Multicast:**



Slide 20

---

## 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

Slide 21

Flat vs Hierarchical Groups:

- Flat group: all decisions made collectively
  - Hierarchical group: coordinator makes decisions
- 

## 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)

Slide 22

$k$ : 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 REPLICATION

Each replica executes as a state machine:

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

Slide 23

Input Messages:

- All replicas agree on content of input messages
  - All replicas agree on order of input messages
  - **Consensus**
- 

## 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 24

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 25

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 26

**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 27 Assume:

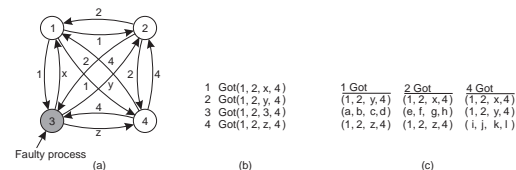
- Execution in rounds
- Timeout to detect lost messages

Byzantine Generals Problem:

Reliable communication but faulty processes.

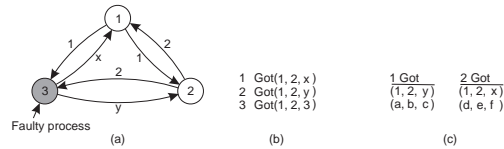
- $n$  generals (processes)
- $m$  are traitors (will send incorrect and contradictory info)
- Need to know everyone else's troop strength  $g_i$
- Each process has a vector:  $\langle g_1, \dots, g_n \rangle$
- (Note: this is actually interactive consistency)

Slide 28



Byzantine Generals Impossibility:

Slide 29



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

Byzantine agreement with Signatures:

Slide 30

- 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

Slide 31

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:

Slide 32

- 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
- no termination

In practice we can get close enough

---

## PAXOS

Goal: a collection of processes chooses a single proposed value

**Proposer** proposes value to choose (leader)

**Slide 33** **Acceptor** accept or reject proposed values

**Learner** any process interested in the result of the consensus

2 phases:

- Propose
- Accept

---

Propose:

- ① Propose: proposer sends a proposal  $\langle seq, value \rangle$  to  $\geq N/2$  acceptors
- ② Promise: acceptors reply.
  - *reject* if  $seq <$  any previously accepted proposal
  - else *accept* and include last accepted value

**Slide 34** **Accept:**

- ① Accept: when  $\geq N/2$  replies, proposer chooses and sends value:
    - highest accepted value received from acceptors
    - else, arbitrary value
  - ② Accepted: acceptors accept value and reply to proposer
  - ③ Propagate value to Learners when  $\geq N/2$  replies received.
- 

---

## PAXOS ERROR CASES

Acceptor fails:

- ✓ As long as a quorum still available

Proposer fails:

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

**Slide 35**

Multiple proposers:

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

---

## ATOMIC MULTICAST

A message is delivered to either all processes, or none

Requires agreement about group membership

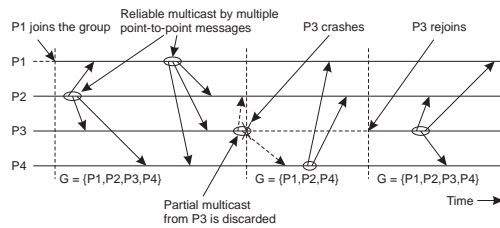
**Slide 36** **Process Group:**

- 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).

Slide 37

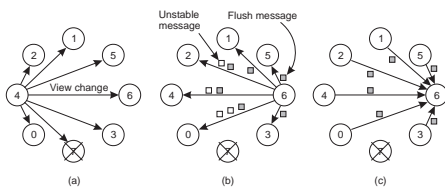


→ view changes and messages are delivered in total order

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

Slide 38



FAILURE RECOVERY

Restoring an erroneous state to an error free state

Issues:

Slide 39

- **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

Slide 40

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 :

- Slide 41**
- 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

---

### Operation-Based Recovery - Logging:

Update in-place together with write-ahead logging

- Every change (update) of data is recorded in a log, which includes:

- Slide 42**
- 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
- 

---

## 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

**Slide 43** → 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:

- Slide 44**
- 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

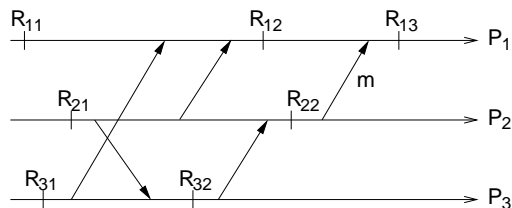
---

## RECOVERY IN DISTRIBUTED SYSTEMS

Slide 45

- 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:

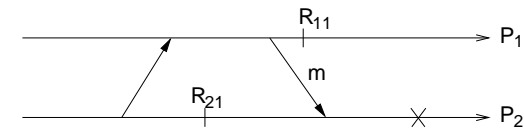


Slide 46

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

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

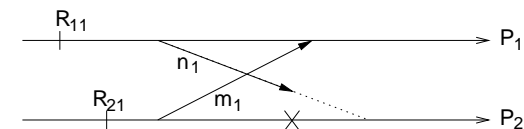
Message Loss:



Slide 47

- Failure of  $P_2 \rightarrow P_2 \rightsquigarrow R_{21}$
- 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:



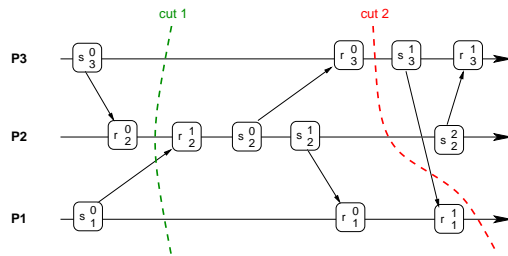
$P_2 \downarrow \rightarrow P_2 \rightsquigarrow R_{21} \rightarrow P_1 \rightsquigarrow R_{11}$ . Note:  $n_1$  in transit

Slide 48

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

## CONSISTENT CHECKPOINTING

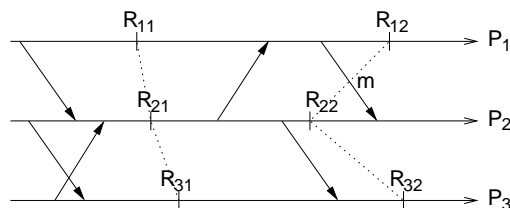
Consistent Cut:



Slide 49

Idea: collect *local checkpoints* in a coordinated way.

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



Slide 50

- $\{R_{11}, R_{21}, R_{31}\}$  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

Slide 51

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

Slide 52

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

Slide 53

### Second Phase:

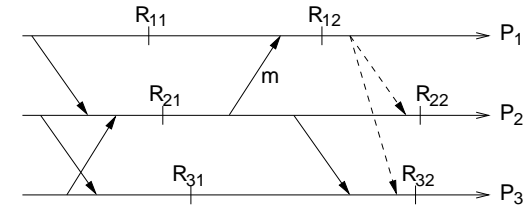
- ① Coordinator  $P_i$  informs all other processes  $P_j$  of decision
- ②  $P_j$  convert or discard tentative checkpoints accordingly

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

Slide 54

## REDUNDANT CHECKPOINTS

Algorithm performs unnecessary checkpoints



Slide 55

- $\{R_{11}, R_{21}, R_{31}\}$  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

## 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 56

---

## CHECKPOINTING ALGORITHM

Messages:

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

**Slide 57** Initialisation: Each  $P$  sets  $OK := true$ ,  $first\_sent = \{0, 0, \dots, 0\}$

Coordinator,  $P_i$ :

- ① send( $t, i, last\_rec_i[j]$ ) to all  $P_j \in cohort_i$
- ② if all replies are  $true$ , send( $p$ ) to all  $P_j \in cohort_i$   
    **else** send( $u$ ) to all  $P_j \in cohort_i$

---

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 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 send( $OK, j$ ) to  $i$

**Slide 58**

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

- ① if  $x = p$  make permanent **else** discard tentative checkpoint
- ② send( $x, j$ ) to all  $P_k \in cohort_j$

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

---

---

## 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

**Slide 59**

Second Phase:

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

---

## REDUNDANT ROLLBACKS

Processes may roll back unnecessarily

→ Can be avoided by keeping track of messages received

Avoiding Redundant Rollbacks:

**Slide 60**

→ 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)

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

→  $P_i$  only needs to roll back after receiving message

( $r, j, last\_sent_j[i]$ ) if  $last\_rec_i[j] > last\_sent_j[i]$

---

---

## ROLLBACK RECOVERY ALGORITHM

Messages:

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

Initialisation: Each  $P$  sets

→  $resume := true$

→  $last\_rec = \{\infty, \infty, \dots, \infty\}$

→  $W$  according to willingness to roll back

Slide 61

Initiator,  $P_i$ :

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

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

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 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 send( $W_j, j$ ) to  $i$ .

Slide 62

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

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

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

---

## 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 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 send.

→ Works as long as no message was lost.

Slide 63

---

## 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 64

---

**Slide 65**

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.

---

## OPTIMISTIC CHECKPOINTING

Asynchronous Logging:

**Slide 66**

- 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 67**

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.
- 

## OPTIMISTIC CHECKPOINT RECOVERY

**Slide 68**

- 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 (→ domino rollback).
-

## OPTIMISTIC CHECKPOINT RECOVERY ALGORITHM

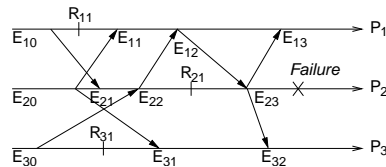
Recovery is initiated by restarting process broadcasting its failure.

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

**Slide 69** Processor  $P_i$  (initiator or not) performs following steps:

- ① for  $k := 1$  to  $N$  do /\*  $N$  is number of processes \*/
- ② for each neighbour  $j$  do
- ③ send( $r, i, n\_sent_{i \rightarrow j}(E_i)$ );
- ④ wait for  $r$  messages from all neighbours;
- ⑤ for each message ( $r, j, s$ ) received do
- ⑥ if  $n\_rcvd_{i \leftarrow j}(E_i) > s$  then /\* have orphan \*/
- ⑦  $E_i := \text{latest } E \text{ suchthat } n\_rcvd_{i \leftarrow j}(E) = s$ ;

Example:



**Slide 70**

- ①  $P_2$  : recover from  $R_{21}$ ;  $E_2 := E_{22}$ ; send( $r, P_2, 2$ ) →  $P_1$ ; send( $r, P_2, 1$ ) →  $P_3$ ;
- ②  $P_1 \leftarrow P_2$ ;  $E_1 := E_{13}$ ;  $n\_rcvd_{1 \leftarrow 2}(E_{13}) = 3 > 2$ :  $E_1 := E_{12}$ ; send( $r, P_1, 2$ ) →  $P_2$ ;
- ③  $P_3 \leftarrow P_2$ ;  $E_3 := E_{32}$ ;  $n\_rcvd_{3 \leftarrow 2}(E_{32}) = 2 > 1$ :  $E_3 := E_{31}$ ; send( $r, P_3, 1$ ) →  $P_2$ ;
- ④  $P_2 \leftarrow P_1$ ;  $n\_rcvd_{2 \leftarrow 1}(E_{22}) = 1 \leq 2$ ; no change; send( $r, P_2, 2$ ) →  $P_1$ ;
- ⑤  $P_2 \leftarrow P_3$ ;  $n\_rcvd_{2 \leftarrow 3}(E_{22}) = 1 \leq 1$ ; no change; send( $r, P_2, 1$ ) →  $P_3$ ;

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

## READING LIST

**Slide 71** Paxos Made Live - An Engineering Perspective Experiences implementing Paxos for Google's Chubby lock server. It turns out to be quite complicated.