

---

## 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
- Without seriously affecting overall performance

Slide 5

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-Silent:** clients do not know server has halted

Slide 7

**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*)

---

## DETECTING FAILURE

Synchronous systems:

- Timeout
- Failure detector sends probes to detect crash failures

Slide 9

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?

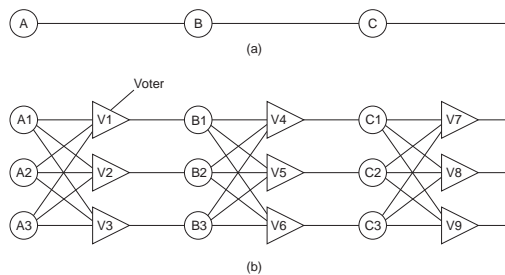
## FAILURE MASKING

Try to hide occurrence of failures from other processes

Redundancy:

- Information redundancy
- Time redundancy
- Physical redundancy

Slide 10



## 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 11

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

Slide 12

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

## AGREEMENT

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

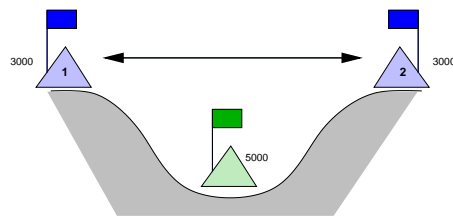
**Slide 13**

- 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 consensus (within a finite number of steps)

**Two Army Problem:**

Non-faulty processes but lossy communication.



**Slide 14**

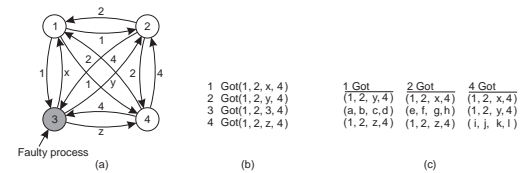
- 1 → 2 attack!
- 2 → 1 ack
- 2: did 1 get my ack?
- 1 → 2 ack ack
- 1: did 2 get my ack ack?
- etc.

**Byzantine Generals Problem:**

Reliable communication but faulty processes.

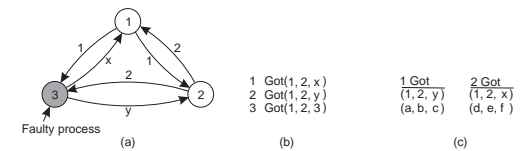
- $n$  generals (processes)
- $m$  are traitors (will send incorrect and contradictory info)
- Need to know everyone else's troop strength  $g_i$
- Each process has a vector:  $\langle g_1, \dots, g_n \rangle$

**Slide 15**



**Byzantine Generals Impossibility:**

**Slide 16**



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

## RELIABLE COMMUNICATION

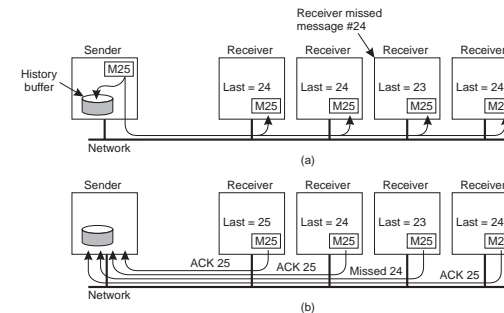
- Communication channel also experiences failure
- Focus on masking crash (lost/broken connections) and omission (lost messages) failures

Slide 17

Reliable Point-to-Point Communication:

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

## RELIABLE GROUP COMMUNICATION



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

Slide 18

How to deal with the various kinds of failure?

## SCALABILITY OF RELIABLE MULTICAST

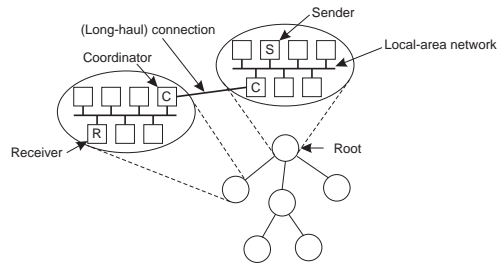
**Feedback Implosion:** sender is swamped with feedback messages

**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 20

Hierarchical Multicast:



Slide 21

ATOMIC MULTICAST

A message is delivered to either all processes, or none

Requires agreement about group membership

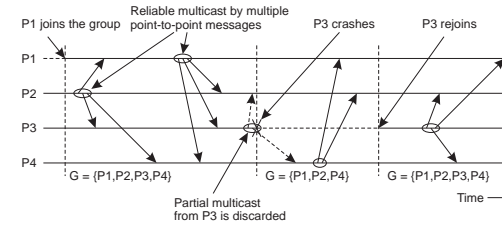
Slide 22 Process Group:

- 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

Virtual 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 23



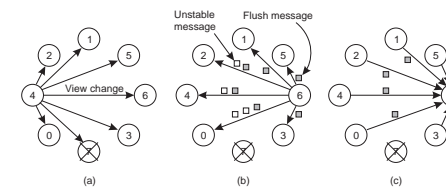
→ view changes and messages are delivered in total order

Implementing Virtual 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 24



---

## FAILURE RECOVERY

Restoring an erroneous state to an error free state

Issues:

Slide 25

- **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 26

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 27

- 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 28

- 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
- 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 29

### 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

Slide 30

**Asynchronous checkpointing:** Use copy-on-write to checkpoint while execution continues

- Easy with UNIX fork()

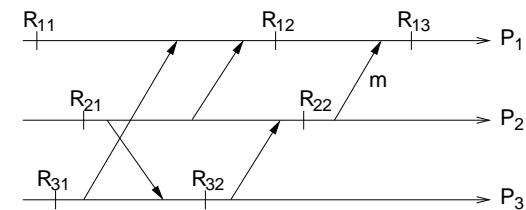
**Compress checkpoints:** Reduces storage and I/O cost at the expense of CPU time

## RECOVERY IN DISTRIBUTED SYSTEMS

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

Slide 31

### Domino Effect:

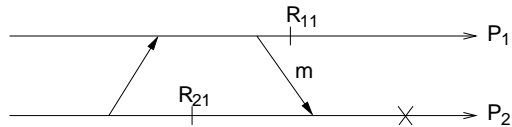


Slide 32

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

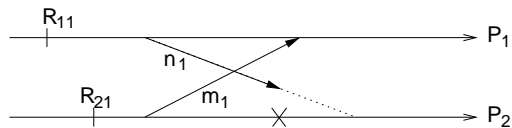
Messaging dependencies plus independent checkpointing may force system to roll back to initial state

Message Loss:

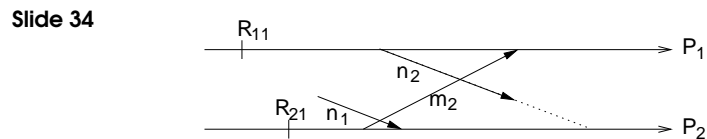


- Slide 33**
- Failure of  $P_2 \rightarrow P_2 \curvearrowright R_{21}$
  - Message  $m$  is now recorded as sent (by  $P_1$ ) but not received (by  $P_2$ ), and  $m$  will never be received after rollback
  - Message  $m$  is lost
  - Whether  $m$  is lost due to rollback or due to imperfect communication channels is indistinguishable!
  - Require protocols resilient to message loss

Livelock:



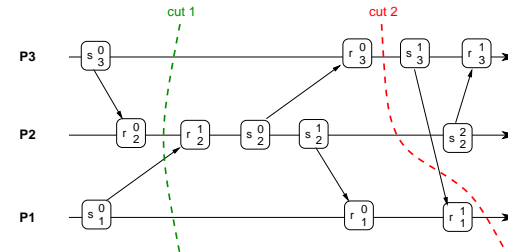
$P_2 \downarrow \rightarrow P_2 \curvearrowright R_{21} \rightarrow P_1 \curvearrowright R_{11}$ . Note:  $n_1$  in transit



- Slide 34**
- 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

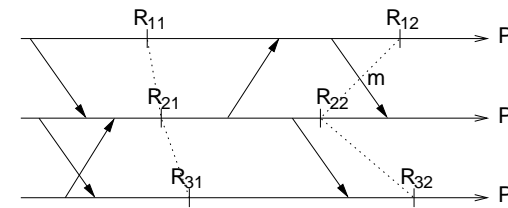
## CONSISTENT CHECKPOINTING

Consistent Cut:



**Slide 35**

- 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 36**

- $\{R_{11}, R_{21}, R_{31}\}$  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 37

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

How to take a consistent checkpoint?:

- Simple solution: Each process checkpoints immediately after sending a message
- ✗ High overhead
- Reducing this to checkpointing after  $n$  messages,  $n > 1$ , is **not** guaranteed to produce a consistent checkpoint!
- Require some coordination during checkpointing

### SYNCHRONOUS CHECKPOINTING

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

Assumptions:

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

Local checkpoints:

**permanent:** part of a global checkpoint

**tentative:** may or may not become permanent

### SYNCHRONOUS ALGORITHM

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

First Phase:

- ① Coordinator  $P_i$  takes tentative checkpoint
- ②  $P_i$  sends  $t$  message to all other processes  $P_j$  to take tentative checkpoint
- ③  $P_j$  reply to  $P_i$  whether succeeded in taking tentative checkpoint
- ④  $P_i$  receives *true* reply from each  $P_j$  → decides to make permanent  
 $P_i$  receives at least one *false* → decides to discard the tentative checkpoints

Slide 39

Second Phase:

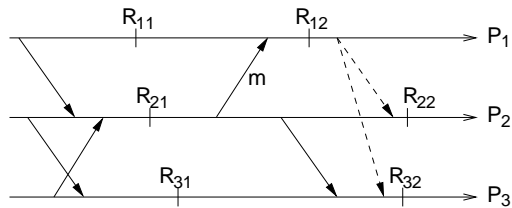
- ① Coordinator  $P_i$  informs all other processes  $P_j$  of decision
- ②  $P_j$  convert or discard tentative checkpoints accordingly

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

Slide 40

## REDUNDANT CHECKPOINTS

Algorithm performs unnecessary checkpoints



Slide 41

- $\{R_{11}, R_{21}, R_{31}\}$  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 42

## CHECKPOINTING ALGORITHM

Messages:

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

Slide 43

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 44

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 45

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:

→ 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 47

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 48

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.

### Slide 49

- 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.

---

## MESSAGE LOGGING

### Slide 50

- 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

---

## Problems with Message Logging:

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

### Slide 51

- 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 52

- 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

