CS 3551 DISTRIBUTED COMPUTING

UNIT IV CONSENSUS AND RECOVERY **10**

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System(Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm -- Algorithm for Asynchronous Checkpointing and Recovery

→ state of process + channel

Checkpoint Based Recovery

~ → time intervals

(X) events log X

(X) appln ⇒ Outside world X

1. Uncoordinated checkpointing
2. Coordinated checkpointing
   a. Blocking coordinated checkpointing
   b. Non-blocking checkpoint coordination
3. Impossibility of min-process non-blocking checkpointing
4. Communication-induced checkpointing
   a. Model-based checkpointing
   b. Index-based checkpointing

Checkpoint Based Recovery

- In the checkpoint-based recovery approach, the state of each process and the communication channel is checkpointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints.
- It does does not need to detect, log, or replay non-deterministic events. Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery.
- However, checkpoint-based rollback recovery does not guarantee that prefailure execution can be deterministically regenerated after a rollback.
- It may not be suitable for applications that require frequent interactions with the outside world.

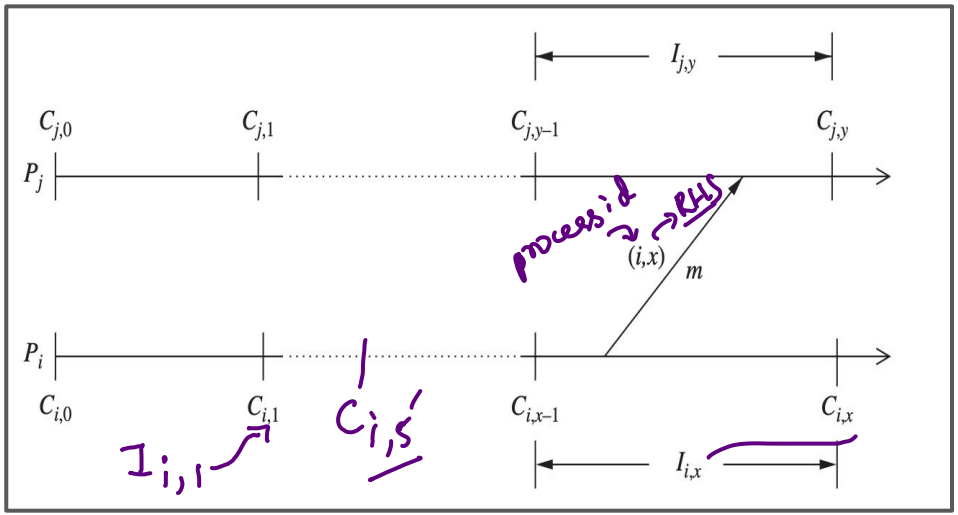
1. Uncoordinated Checkpointing ⇒ BOSS

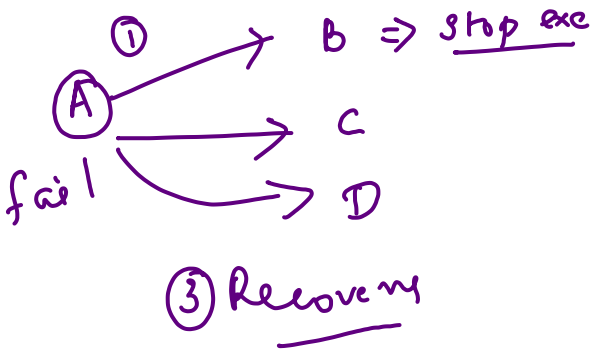
- Each process has autonomy in deciding when to take checkpoints.
- Synchronization overhead is minimal as there is no need for coordination between processes. (Lower runtime overhead).
- Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions

Drawbacks:

1. Domino effect may occur during a recovery.
2. Recovery is slow because processes need to iterate to find a consistent set of checkpoints.
3. Useless Checkpoint:
   a. Since no coordination is done at the time the checkpoint is taken, checkpoints taken by a process may be useless checkpoints.
   b. Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line.
4. forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer required
5. it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line

How consistent global checkpoint is determined?



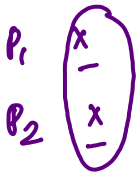


Steps:

1. When a failure occurs, the recovering process initiates rollback by broadcasting a dependency request message to collect all the dependency information maintained by each process.
2. When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.
3. The initiator then calculates the recovery line based on the global dependency information and broadcasts a rollback request message containing the recovery line.
4. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

Coordinated Checkpointing:

1. **Definition:** Processes coordinate checkpointing to ensure consistent global state.
2. **Benefits:** Simplifies recovery, avoids the domino effect, and each process restarts from its latest checkpoint.
3. **Storage:** Requires each process to maintain only one checkpoint, reducing storage overhead and eliminating the need for garbage collection.
4. **Drawbacks:** Involves large latency for committing output, and delays/overhead occur with each new global checkpoint.

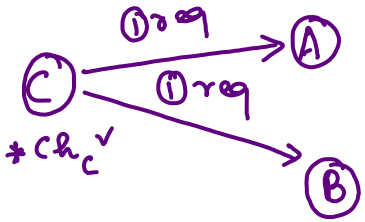


Clock Synchronization:

1. **Ideal Scenario:** If perfectly synchronized clocks exist, a simple method involves all processes agreeing on checkpoint instants triggered by their clocks.
2. **Reality:** Perfectly synchronized clocks are not available.
3. **Approaches for Consistency:**
   - Block message sending during the protocol to ensure consistency.
   - Piggyback checkpoint indices on messages to avoid blocking.

① Blocking Coordinated Checkpoint

1. **Approach:** Coordinated checkpointing involves blocking communications during the protocol execution.
2. **Procedure:**
   - After a process takes a local checkpoint, it remains blocked to prevent orphan messages until the entire checkpointing activity is complete.
   - The coordinator initiates the process by taking a checkpoint and broadcasting a request message to all processes.
   - Upon receiving the message, each process stops execution, flushes communication channels, takes a tentative checkpoint, and sends an acknowledgment back to the coordinator.
   - After receiving acknowledgments from all processes, the coordinator broadcasts a commit message, completing the two-phase checkpointing protocol.
   - Upon receiving the commit message, each process removes the old permanent checkpoint, makes the tentative checkpoint permanent, and resumes execution and message exchange.
3. **Drawback:** Computation is blocked during checkpointing, making non-blocking checkpointing schemes preferable for improved efficiency.



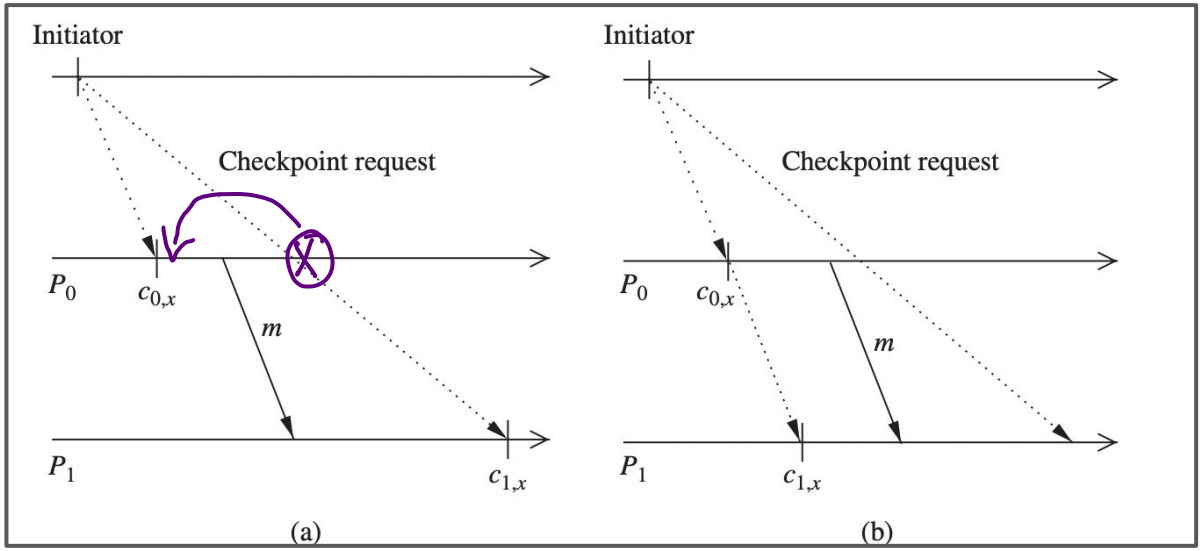
⇒ Stop exe
⇒ flush channel
⇒ tentative
⇒ ack

Coord
A ack ✓
B ack ✓

req ⇒ permanent

Non-Blocking Coordinated Checkpoint

- In this approach the processes need not stop their execution while taking checkpoints.
- A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.



Solution 1

If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message,

Solution 2

- If the channels are non-FIFO, the following two approaches can be used: first, the marker can be piggybacked on every post-checkpoint message.
- When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message.
- Alternatively, checkpoint indices can serve the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked checkpoint index.

receiver index < piggy index

3 Impossibility of min-process non-blocking checkpointing

- A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation.
- Clearly, such checkpointing algorithms will be very attractive. Cao and Singhal [7] showed that it is impossible to design a min-process, non-blocking checkpointing algorithm.
- **Possible Algorithm:**
- **Phase 1:**
  - checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request. Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- **Phase 2:**
  - all processes identified in the first phase take a checkpoint.
  - The result is a consistent checkpoint that involves only the participating processes.
  - In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.
- Based on a concept called “Z-dependency,” Cao and Singhal proved that there does not exist a non-blocking algorithm that will allow a minimum number of processes to take their checkpoints.

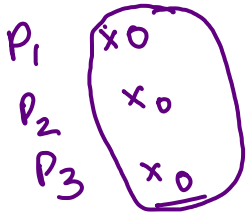
Communication Induced Checkpoint

- **Purpose:** Communication-induced checkpointing prevents the domino effect while allowing processes to take some checkpoints independently.
- **Checkpoint Types:**
  - **Autonomous Checkpoints:** Processes take these independently.
  - **Forced Checkpoints:** Processes are compelled to take these to ensure recovery progress.
- **Process Independence:** Constrained to ensure the recovery line's eventual progress.
- **Reduction of Useless Checkpoints:** Communication-induced checkpointing minimizes or eliminates unnecessary checkpoints.
- **Implementation:**
  - **Piggybacking:** Protocol-related information is added to application messages.
  - **Receiver Action:** The receiver uses this information to decide whether to take a forced checkpoint to advance the global recovery line.
- **Latency and Overhead:** Forced checkpoints may introduce some delay and additional processing overhead.
- **Types of Communication-Induced Checkpointing:**
  - **Model-Based Checkpointing:** System maintains checkpoints and communication structures to prevent the domino effect.
  - **Index-Based Checkpointing:** System uses an indexing scheme for local and forced checkpoints, ensuring consistency across processes with the same index.

Model Based Checkpoint

- **Objective:** Model-based checkpointing prevents inconsistent states among checkpoints caused by communication patterns.
- **Process Action:**
  - A process identifies potential inconsistencies and independently enforces local checkpoints to avoid undesirable patterns.
  - Forced checkpoints are used to prevent these undesirable patterns.
- **Communication:** No control messages are exchanged during normal operation; all necessary information is included in application messages.
- **Decision-Making:** The choice to take a forced checkpoint is made locally based on available information.
- **Domino-Effect-Free Models:**
  - **MRS Model:** Ensures no domino effect by making sure all message-receiving events precede message-sending events within each checkpoint interval. Additional checkpoints are taken when needed.
  - **Rollback Prevention:** Another approach is to prevent the domino effect by taking a checkpoint immediately after every message-sending event.

Index-based checkpointing



- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

LIKE

COMMENT

SHARE

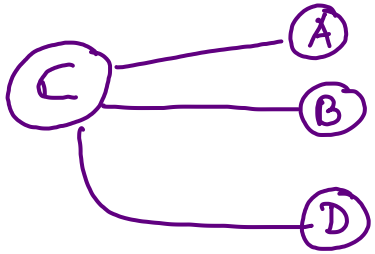
SUBSCRIBE

CS 3551 DISTRIBUTED COMPUTING

UNIT IV CONSENSUS AND RECOVERY **10**

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System(Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm -- Algorithm for Asynchronous Checkpointing and Recovery

Koo–Toueg coordinated checkpointing algorithm



P-1 {
1. Sends out req. to every P_i (tentative)
2. Tentative chkpt + ack
3. @C

ack A, B, D receive ⇒ decision

P-2 { 4. tentative ⇒ permanent

Koo–Toueg coordinated checkpointing algorithm

Recovery

① ask if willing to restart from chkpt?

② yes from all ⇒ Rollback
no ⇒ abort

③ Convey decision to all.

Koo–Toueg coordinated checkpointing algorithm

Objective:

- Takes a consistent set of checkpoints and avoids the domino effect and livelock problems during the recovery.
- Processes coordinate their local checkpointing actions such that the set of all checkpoints in the system is consistent.

Assumptions of Checkpointing Algorithm:

- Processes communicate by exchanging messages through communication channels. Communication channels are FIFO.
- It is assumed that end-to-end protocols (such as the sliding window protocol) exist to cope with message loss due to rollback recovery and communication failure.
- Communication failures do not partition the network.

Permanent vs Tentative:

1. A permanent checkpoint is a local checkpoint at a process and is a part of a consistent global checkpoint.
2. A tentative checkpoint is a temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm.
3. In case of a failure, processes roll back only to their permanent checkpoints for recovery.

Checkpoint - Phase 1

1. An initiating process Pi takes a tentative checkpoint and requests all other processes to take tentative checkpoints.
2. Each process informs Pi whether it succeeded in taking a tentative checkpoint.
3. A process says “no” to a request if it fails to take a tentative checkpoint, which could be due to several reasons, depending upon the underlying application.
4. If Pi learns that all the processes have successfully taken tentative checkpoints, Pi decides that all tentative checkpoints should be made permanent; otherwise, Pi decides that all the tentative checkpoints should be discarded.

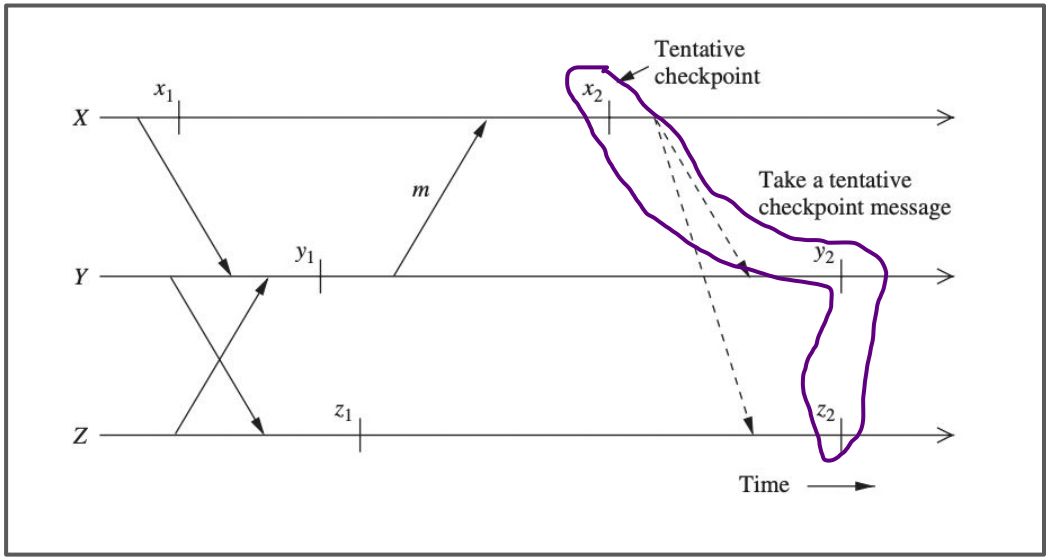
Checkpoint - Phase 2

1. Pi informs all the processes of the decision it reached at the end of the first phase.
2. A process, on receiving the message from Pi, will act accordingly.
3. Therefore, either all or none of the processes advance the checkpoint by taking permanent checkpoints.
4. The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the underlying computation until it is informed of Pi’s decision.

Correctness ✓

- A set of permanent checkpoints taken by this algorithm is consistent because of the following two reasons:
  - Either all or none of the processes take permanent checkpoints;
  - no process sends a message after taking a tentative checkpoint until the receipt of the initiating process's decision, as by then all processes would have taken checkpoints.
- Thus, a situation will not arise where there is a record of a message being received but there is no record of sending it

Optimization ✓



x_2, y_2, z_1

Rollback Recovery Algorithm

Assumption

- a single process invokes the algorithm.
- It also assumes that the checkpoint and the rollback recovery algorithms are not invoked concurrently

Phase 1

- An initiating process Pi sends a message to all other processes to check if they all are willing to restart from their previous checkpoints.
- A process may reply “no” to a restart request due to any reason (e.g., it is already participating in a checkpoint or recovery process initiated by some other process).
- If Pi learns that all processes are willing to restart from their previous checkpoints, Pi decides that all processes should roll back to their previous checkpoints. Otherwise, Pi aborts the rollback attempt and it may attempt a recovery at a later time.

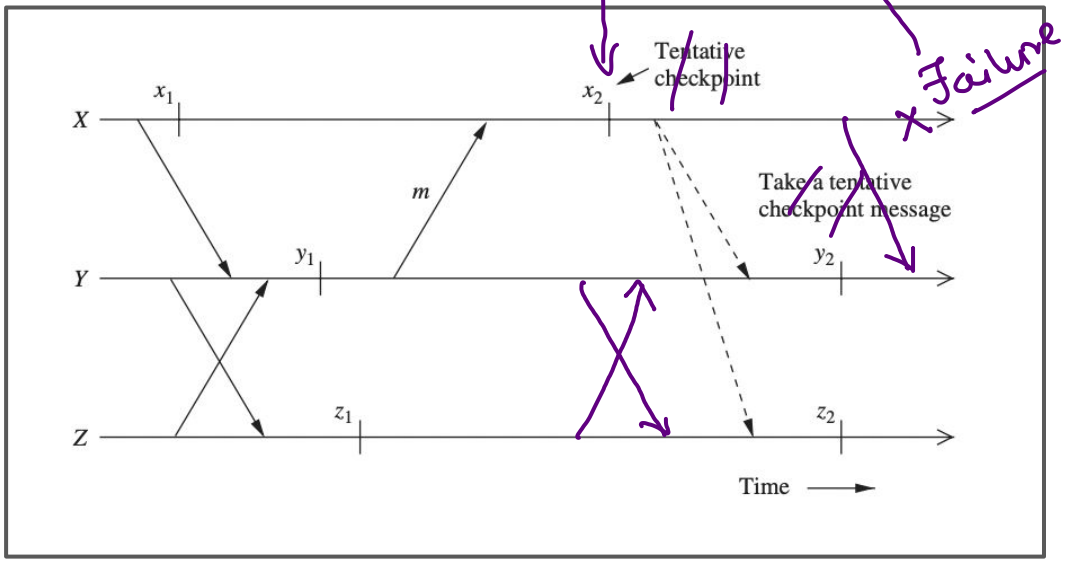
Phase 2

- Pi propagates its decision to all the processes.
- On receiving Pi’s decision, a process acts accordingly.
- During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for Pi’s decision.

Correctness

- All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state (the checkpointing algorithm takes a consistent set of checkpoints).

Optimization



LIKE

COMMENT

SHARE

SUBSCRIBE

CS 3551 DISTRIBUTED COMPUTING

UNIT IV CONSENSUS AND RECOVERY **10**

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System(Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm -- Algorithm for Asynchronous Checkpointing and Recovery

② Juang–Venkatesan algorithm for asynchronous checkpointing and recovery

assume
1) fifo channel
2) delay finite

When msg arrives @ process?
1) processes recd. msg
2) $S \Rightarrow S'$
3) sends zero/more msgs to other process.

events < Volatile — fast ⇒ power off
Stable

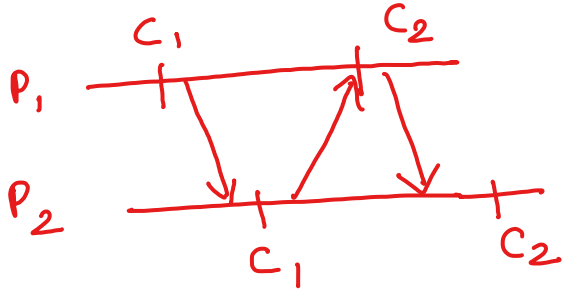
Checkpointing after event

{ s, m (msg receive), msgs sent }
↳ msgs sent out

⇒ ① Volatile
↓
Stable

↓
State of proc. before event

Juang–Venkatesan algorithm for asynchronous checkpointing and recovery



Send = Receive

$Send_{i \to j}(Chkpt_i)$

$Recd_{i \leftarrow j}(Chpt_i)$

if fail

1) Rollback to latest chkpt
2) send ⇒ transmit
3) receive msgs from others

@ C_x

send = receive ✓

P_1

$Send_{P_1 \to P_2}(C_1) = 0$

$Send_{P_1 \to P_2}(C_2) = 1$

P_2

C_1

$Recd_{P_2 \leftarrow P_1}(C_1) = 1$

$Recd_{P_2 \leftarrow P_1}(C_2) = 2$ //

Juang–Venkatesan algorithm for asynchronous checkpointing and recovery

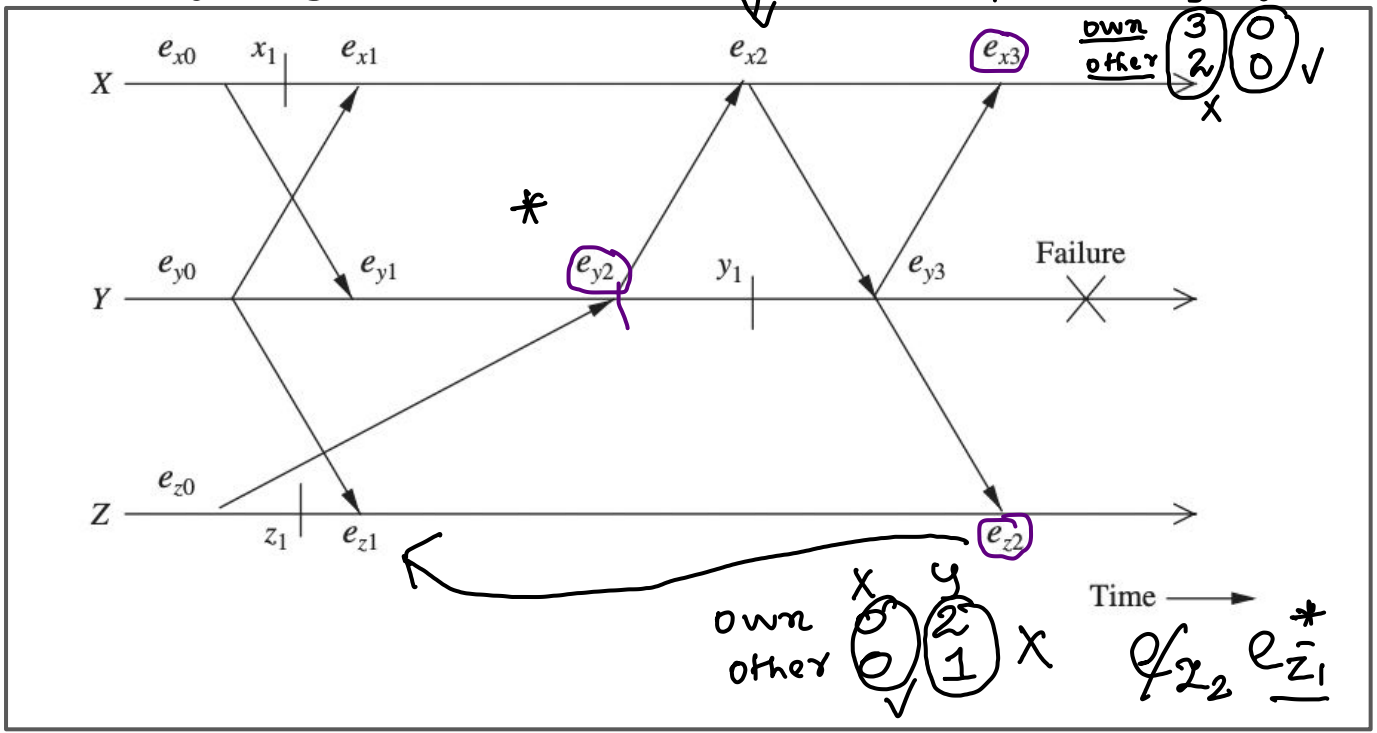
System Model and Assumptions:

- Communication channels are reliable, deliver the messages in FIFO order, and have infinite buffers.
- The message transmission delay is arbitrary, but finite.
- The processors directly connected to a processor via communication channels are called its neighbors.
- The underlying computation or application is assumed to be event-driven: a processor P waits until a message m is received, it processes the message m, changes its state from s to s , and sends zero or more messages to some of its neighbors.
- Then the processor remains idle until the receipt of the next message.
- The new state s and the contents of messages sent to its neighbors depend on state s and the contents of message m. The events at a processor are identified by unique monotonically increasing numbers, ex0, ex1, ex2,
- **Storage can be:**
    - Volatile Log - Less time but data lost when power is lost
    - Stable Storage - More time but data not lost.

Asynchronous Checkpointing

- After executing an event, a processor records a triplet (s, m, msgs_sent) in its volatile storage,
  - s is the state of the processor before the event
  - m is the message (including the identity of the sender of m, denoted as m.sender) whose arrival caused the event
  - msgs_sent is the set of messages that were sent by the processor during the event.
- Therefore, a local checkpoint at a processor consists of the record of an event occurring at the processor and it is taken without any synchronization with other processors.
- Periodically, a processor independently saves the contents of the volatile log in the stable storage and clears the volatile log. (Equivalent to taking a local checkpoint)

Recovery Algorithm



e_{x_2} * y-fail ⇒ Restart e_{y_2}

Send ⇒ transmit

X - Rollback(Y, 2)

Z - Rollback(Y, 1)

X @ e_{x_3}

Y - RB(X, 2)

Z - RB(X, 0)

Z @ e_{z_2}

X - RB(Z, 0)

Y - RB(Z, 1)

Recd e_{y_2} X - 1, Z - 1 e_{x_3} Y - 3, Z - 0 e_{z_2} ⇒ X - 0, Y - 2

Procedure RollBack_Recovery: processor p_i executes the following:

STEP (a)

if processor p_i is recovering after a failure **then**

$CkPt_i$:= latest event logged in the stable storage

else

$CkPt_i$:= latest event that took place in p_i {The latest event at p_i can be either in stable or in volatile storage.}

end if

STEP (b)

for $k = 1$ to N {N is the number of processors in the system} **do**

for each neighboring processor p_j **do**

compute $SENT_{i \to j}(CkPt_i)$

send a $ROLLBACK(i, SENT_{i \to j}(CkPt_i))$ message to p_j

end for

for every $ROLLBACK(j, c)$ message received from a neighbor j **do**

if $RCVD_{i \leftarrow j}(CkPt_i) > c$ {Implies the presence of orphan messages} **then**

find the latest event e such that $RCVD_{i \leftarrow j}(e) = c$ {Such an event e may be in the volatile storage or stable storage.}

$CkPt_i := e$

end if

end for

end for{for k}

The recovery algorithm

Notation and data structure

The following notation and data structure are used by the algorithm:

- $RCVD_{i \leftarrow j}(CkPt_i)$ represents the number of messages received by processor p_i from processor p_j, from the beginning of the computation until the checkpoint $CkPt_i$.
- $SENT_{i \rightarrow j}(CkPt_i)$ represents the number of messages sent by processor p_i to processor p_j, from the beginning of the computation until the checkpoint $CkPt_i$.

Basic idea

Since the algorithm is based on asynchronous checkpointing, the main issue in the recovery is to find a consistent set of checkpoints to which the system can be restored. The recovery algorithm achieves this by making each processor keep track of both the number of messages it has sent to other processors as well as the number of messages it has received from other processors. Recovery may involve several iterations of roll backs by processors. Whenever a processor rolls back, it is necessary for all other processors to find out if any message sent by the rolled back processor has become an orphan message. Orphan messages are discovered by comparing the number of messages sent to and received from neighboring processors. For example, if $RCVD_{i \leftarrow j}(CkPt_i) > SENT_{j \rightarrow i}(CkPt_j)$ (that is, the number of messages received by processor p_i from processor p_j is greater than the number of messages sent by processor p_j to processor p_i, according to the current states of the processors), then one or more messages at processor p_i are orphan messages. In this case, processor

p_j must roll back to a state where the number of messages received agrees with the number of messages sent.

Description of the algorithm

When a processor restarts after a failure, it broadcasts a *ROLLBACK* message that it has failed.[1] The recovery algorithm at a processor is initiated when it restarts after a failure or when it learns of a failure at another processor. Because of the broadcast of *ROLLBACK* messages, the recovery algorithm is initiated at all processors. The algorithm is shown in Algorithm 13.1.

The rollback starts at the failed processor and slowly diffuses into the entire system through *ROLLBACK* messages. Note that the procedure has $|N|$ iterations. During the kth iteration ($k \neq 1$), a processor p_i does the following: (i) based on the state $CkPt_i$ it was rolled back in the $(k-1)$th iteration, it computes $SENT_{i \to j}(CkPt_i)$ for each neighbor p_j and sends this value in a *ROLLBACK* message to that neighbor; and (ii) p_i waits for and processes *ROLLBACK* messages that it receives from its neighbors in kth iteration and determines a new recovery point $CkPt_i$ for p_i based on information in these messages. At the end of each iteration, at least one processor will rollback to its final recovery point, unless the current recovery points are already consistent.

LIKE

COMMENT

SHARE

SUBSCRIBE