Check Pointing and Rollback Recovery

Course: Distributed Computing

Faculty: Dr. Rajendra Prasath

Spring 2019

About this topic

This course covers various concepts in **Check Pointing and Rollback Recovery.** We will also focus on the essential aspects of check pointing and roll back recovery in distributed contexts

RECAP

What did you learn so far?

- ➔ Challenges in Message Passing systems
- ➔ Distributed Sorting
- ➔ Space-Time Diagram
- ➔ Partial Ordering / Causal Ordering
- ➔ Concurrent Events
- ➔ Local Clocks and Vector Clocks
- ➔ Distributed Snapshots
- ➔ Termination Detection
- ➔ Topology Abstraction and Overlays
- ➔ Leader Election Problem in Rings
- ➔ Message Ordering / Group Communications
- ➔ Distributed Mutual Exclusion Algorithms

Topics to focus on …

- Distributed Mutual Exclusion
- Deadlock Detection

- Check Pointing and Rollback Recovery

- Self-Stabilization
- Distributed Consensus
- Reasoning with Knowledge
- Peer – to – peer computing and Overlays
- Authentication in Distributed Systems

For End Semester

Distributed Mutual Exclusion(Recap)

- **No Deadlocks** – No processes should be permanently blocked, waiting for messages (Resources) from other sites
- **No starvation** – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
- **Fairness** - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation)
- **Fault Tolerance** – the algorithm is able to survive a failure at one or more sites

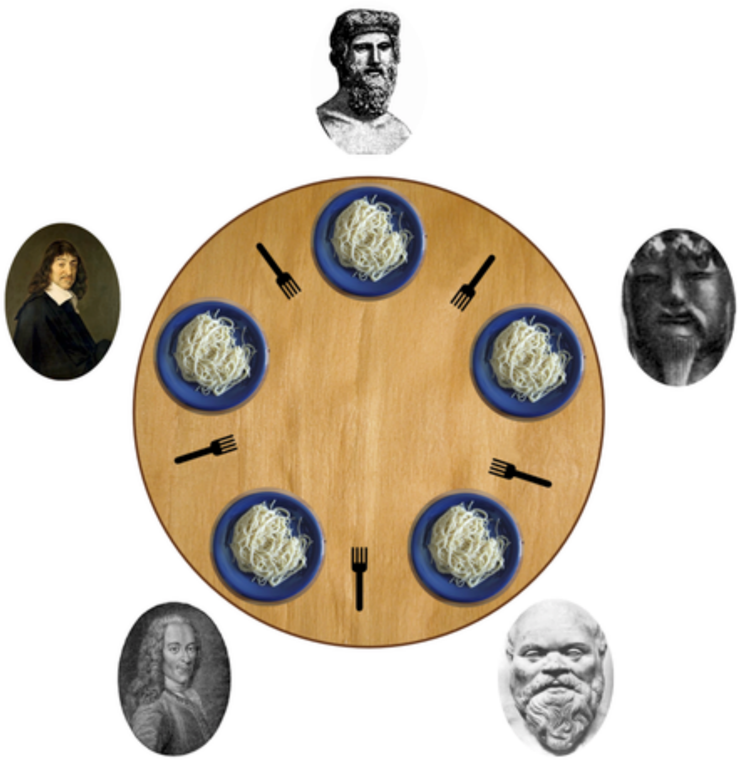
Deadlock – Illustrated (Recap)

- Vehicular Traffic – A real-time scenario

Dining Philosophers (Recap)

- Each philosopher must alternately think and eat
- A philosopher can only eat when they have both left and right forks
- Problem: How to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve?

- Suggest a Simple Solution ??



Check Pointing and Rollback Recovery

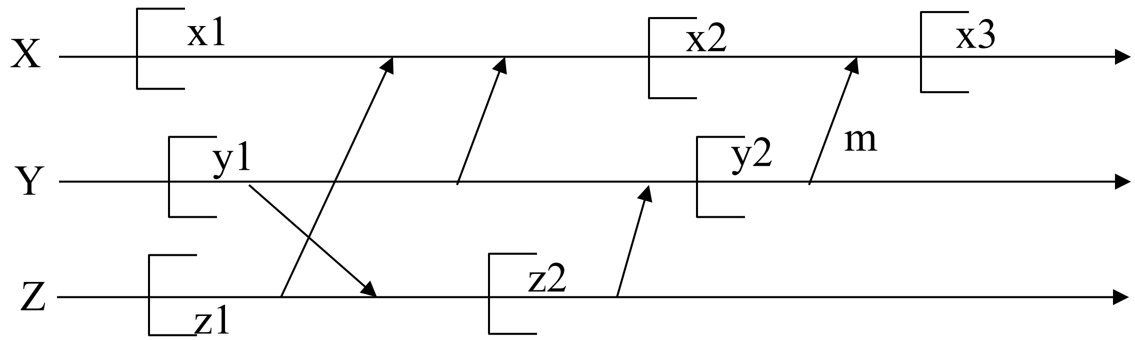
Let us explore Check Pointing and Roll Back Recovery algorithms in distributed systems

Handling Failures / Recovery?

- Failure of a site/node in a distributed system causes inconsistencies in the state of the system.
- Recovery: bringing back the failed node in step with other nodes in the system.
- Failures:
  - Process failure:
    - Deadlocks, protection violation, erroneous user input, etc.
  - System failure:
    - Failure of processor/system. System failure can have full/partial amnesia.
    - It can be a pause failure (system restarts at the same state it was in before the crash) or a complete halt.
  - Secondary storage failure: data inaccessible.
  - Communication failure: network inaccessible.

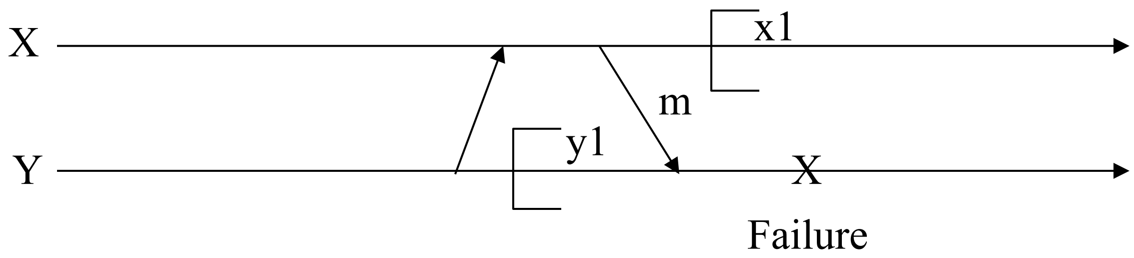
Recovery in Concurrent Systems

- State involves message exchanges in DS
- In distributed systems, rolling back one process can cause the roll back of other processes
- Orphan messages & Domino effect: Assume Y fails after sending m
  - X has record of m at x3 but Y has no record. M → orphan message.
  - Y rolls back to y2 → X should go to x2
  - If Z rolls back, X and Y has to go to x1 and y1 → Domino effect, roll back of one process causes one or more processes to roll back

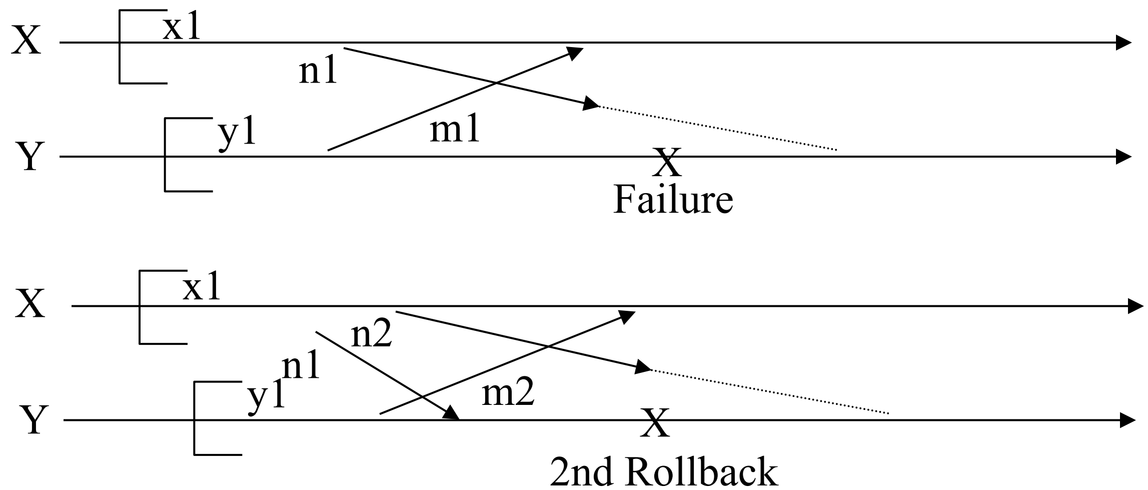


Messages Lost

- If Y fails after receiving m, it will rollback to y1
- X will rollback to x1
- m will be a lost message as X has recorded it as sent & Y has no record of receiving it

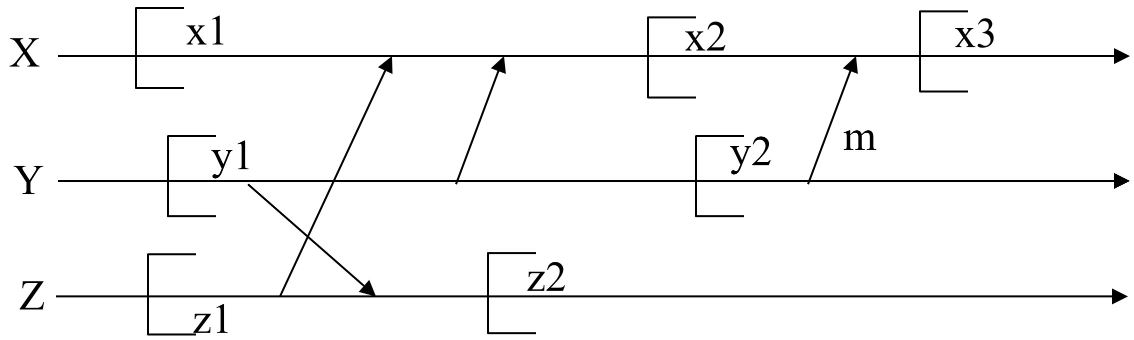


Livelocks



- Y crashes before receiving n1. Y rolls back to y1 → X to x1
- Y recovers, receives n1 and sends m2
- X recovers, sends n2 but has no record of sending n1
- Hence, Y is forced to rollback second time. X also rolls back as it has received m2 but Y has no record of m2
- Above sequence can repeat indefinitely, causing a livelock

Consistent Checkpoints



- Overcoming domino effect and livelocks: checkpoints should not have messages in transit.
- Consistent checkpoints: no message exchange between any pair of processes in the set as well as outside the set during the interval spanned by checkpoints.
- {x1,y1,z1} is a strongly consistent checkpoint

Types of CRR Algorithms

- Synchronous Algorithm
  - Two Phase algorithm proposed by Koo and Toueg
- Asynchronous Algorithm
  - A simple algorithm proposed by Juang & Venkatesan

Consistent Set of Checkpoints

Assumptions:

- Checkpoint, send / recv are atomic
- Take a checkpoint after sending every message
- The set of the most recent checkpoints is always consistent
  - Why? Is it strongly consistent?
- What is the main problem with this approach?
- Take a checkpoint after every K messages sent?
- Is it still consistent?

Synchronous Checkpointing Algo

- Proposed by Koo ad Toueg[1] (1987)

- Assumptions:
  - processes communicate by exchanging messages through channels
  - channels are FIFO, end-to-end protocols cope up with the message loss due to rollback recovery
  - Communication failures do not partition the network
  - Uses two kinds of checkpoints
    - Tentative
    - Permanent

[1] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," in IEEE Transactions on Software Engineering, vol. SE-13, no. 1, pp. 23-31, Jan. 1987. doi: 10.1109/TSE.1987.232562

Phase - 1

- Initiator: take tentative checkpoint
- Initiator requests all other processes to take tentative checkpoint
- All other processes:
  - can respond `yes' or `no'
- Initiator: decide to make checkpoints permanent if everyone has responded `yes’
- A process can fail to take a checkpoint due to the nature of application (e.g.,) lack of log space, unrecoverable transactions

Phase - 2

- If all processes took checkpoints, P_i decides to make the checkpoint permanent.
- Otherwise, checkpoints are to be discarded.
- P_i conveys this decision to all the processes as to whether checkpoints are to be made permanent or to be discarded

Potential Issues

- Between tentative checkpoint and commit/abort of checkpoint process must hold back messages.
- Does this guarantee we have a strongly consistent state?
- Can you construct an example that shows we can still have lost messages?

Synchronous Checkpointing: Properties

- All or none of the processes take permanent checkpoints
- There is no record of a message being received but not sent
- Checkpoints may be taken unnecessarily (Give an example!!)
- Can these unnecessarily checkpoints be avoided?

Optimizing Checkpoints

Main IDEA:

- Record all messages sent and received after the last checkpoint (last_recv(x, y), first_sent(x, y))
- When X requests Y to take a tentative checkpoint:
  - X sends the last message received from Y with the request
  - Y takes a tentative checkpoint only if the last message received by X from Y was sent after Y sent the first message after the last checkpoint (Happened before !!)

$$last\_recv(x, y) \geq first\_sent(y, x)$$

- When a process takes a checkpoint, it will ask all other processes (that sent messages to the process) to take checkpoints.

Rollback Recovery: Properties

- There are two phases: Phase 1 and Phase 2
- Assume that between requests to rollback and decision, no one sends other messages
- All or none of the processes restart from checkpoints
- After rollback, all processes resume in a consistent state
- Can have unnecessary rollback: can use a similar technique as the one in taking checkpoints to eliminate unnecessary rollback

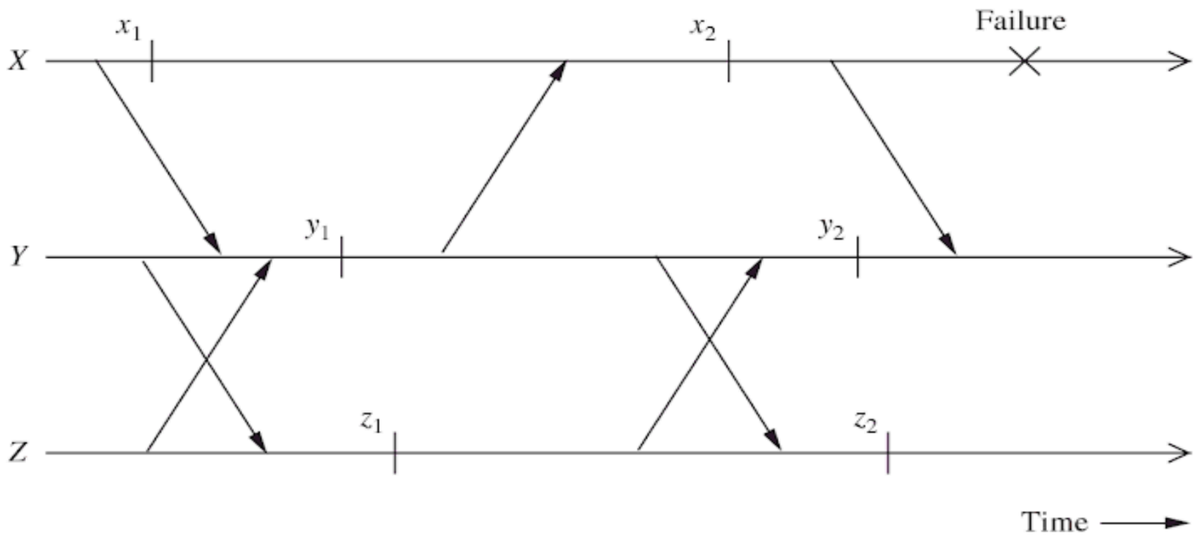
Rollback Recovery

- Phase 1
  - Initiator: check whether all processes are willing to restart from last checkpoints
  - Others: may reply `yes' or `no'

- Phase 2
  - Initiator: propagate go/nogo decision to all processes
  - Others: carry out the decision of the initiator

Unnecessary Rollbacks

- Avoid Rollback in unnecessary situations?
- An example
  - (z_2 does not need to rollback – why?)



Disadvantages

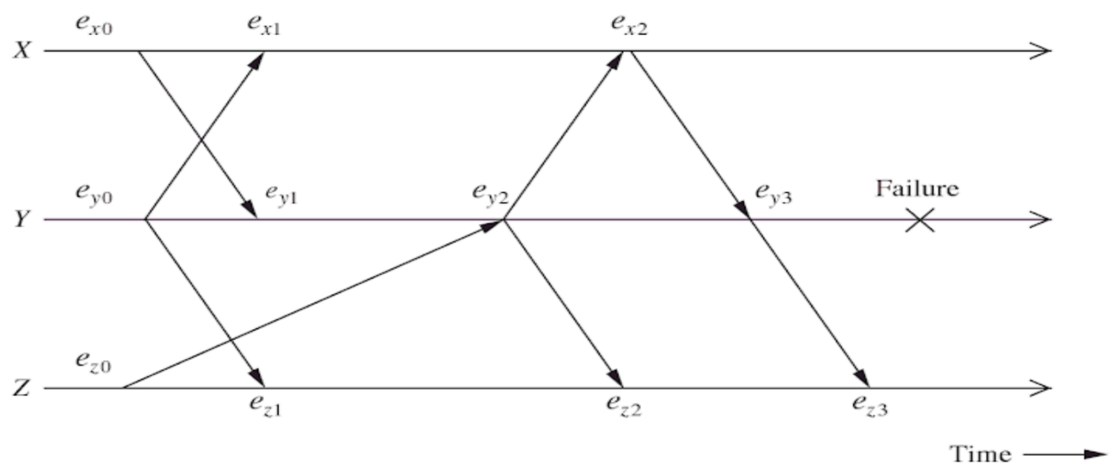
- Check Pointing Algorithm generates **message traffic**
- Synchronization delays are introduced
- These costs may seem high if failures between checkpoints are unlikely

Asynchronous Approach

- Take multiple local checkpoints independently
- After a failure, try to find a consistent set of recent checkpoints
- All incoming messages between local checkpoints are logged
  - pessimistic approach: log each message before processing
  - optimistic approach: buffer messages & log in batches
- Why is the second approach called optimistic?
- What are the advantages and disadvantages of each approach?

An Event Driven Computation

- A process waits until it receives a message; then processes the received message; changes its state and sends zero or more messages to its neighbors and then waits to receive the next message
- The current state and the contents of the messages sent depend on its previous state and the content of the message
- Events are identified by unique numbers (increasing)



Asynchronous Checkpointing Algo

- Proposed by Juang & Venkatesan[2]

Assumptions:

- Communication channels are reliable
- Communication channels are FIFO
- Communication channels have no buffer size limits
- Message transmission delay is bounded
- Underlying system is Event-Driven, with locally timestamped (monotonically increasing numbers) events: Each event waits for a message, processes the message, changes process state, and sends a number of messages

[2] https://www.utdallas.edu/~venky/pubs/crash-rec-icdcs91.pdf

Basic Idea

- At each event, a triplet {s, m, msgs_sent} is put in the the log: s is the state, m is the message causing the event, msgs_sent is the set of messages sent.

Two data structures used:

- RCVD(i, j, checkpoint) -- the number of message received by processor i from processor j at checkpoint,
- SENT(i, j, checkpoint) -- the number of messages sent from i to j at checkpoint.
- Use the message send/recv counts to determine the point to rollback.

Algorithm

At process i:

- If i is a process that is recovering from a failure, checkpoint = the latest event logged in the stable storage.
- else checkpoint = latest event that took place.
- for k = 1 to N do
  - send ROLLBACK(i, SENT(i, j, checkpoint)) to all neighbors j
  - wait for ROLLBACK messages from all neighbors
  - for every ROLLBACK(j, c) received
    - if (RCVD(i, j, checkpoint) > c) then
    - find the latest event e such that RCVD(i, j, e) = c
    - checkpoint = e

Is the algorithm consistent?

- In each iteration:

  At least one processor will rollback to its final recovery point unless current recovery point is consistent
- Answer: YES / NO

- Complexity of this algorithm?
  - will it be greater than O(n) where n is the total number of message exchanges?

  - Explore the details … !!

Summary

- Recovery in Distributed / Concurrent Systems
- Checkpointing
  - Consistent set of checkpoints
- Rollback recovery
  - Synchronous Algorithm (Koo and Toueg)
  - Asynchronous Algorithm (Juang & Venkatesan)

- Stay tuned … More to come up … !!

How to reach me?

- ➔ Please leave me an email:
  rajendra [DOT] prasath [AT] iiits [DOT] in

- ➔ Visit my homepage @

  - ➔ http://www.iiits.ac.in/FacPages/index-rajendra.html

  OR

  - ➔ http://rajendra.2power3.com

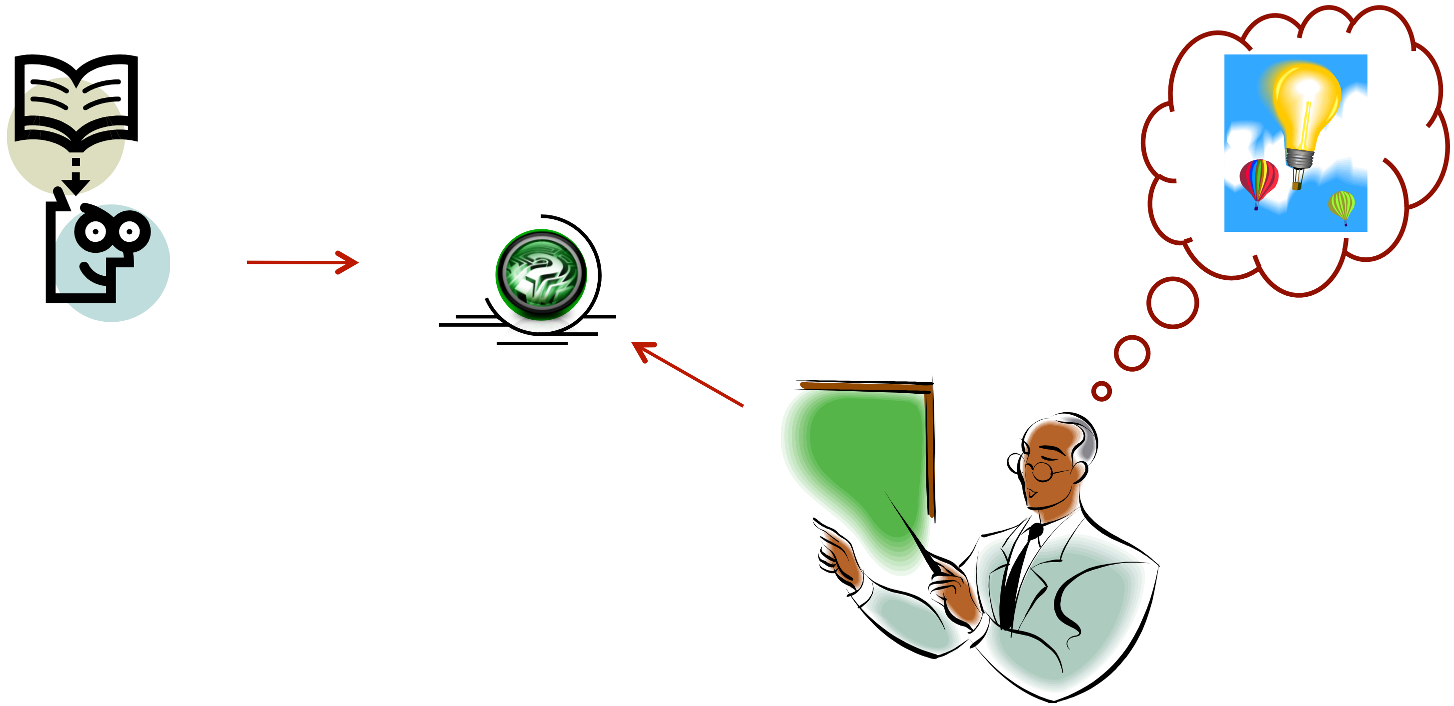
Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Thanks …



… Questions ???