Fault-Tolerant Computer Systems
ECE 60872
Recovery

Saurabh Bagchi
School of Electrical & Computer Engineering
Purdue University

Slides based on ECE442 at the University of Illinois taught by
Profs. Ravi Iyer & Zbigniew Kalbarczyk

Recovery - Basic Concepts

- **Providing fault tolerance involves three phases**
  - Error detection
  - Assessment of the extent of the damage
  - Error recovery to eliminate errors and restart afresh
- **Forward error recovery** - the continuation of the currently executing process from some further point with compensation for the corrupted and missed data. The assumptions:
  - The precise error conditions that caused the detection and the resulting damage can be accurately assessed
  - The errors in the process's (system's) state can be removed
  - The process (system) can move forward
  - Example: exception handling and recovery

Recovery - Basic Concepts (cont.)

- **Backward error recovery** - the current process is rolled back to a certain, error-free, point and re-executes the corrupted part of the process thus continuing the same requested service. The assumptions:
  - The nature of faults cannot be foreseen and errors in the process's (system's) states cannot be removed without re-executing
  - The process (system) state can be restored to a previous error-free state of the process (system)

Comparison Forward & Backward Error Recovery

	Advantages	Disadvantages
Forward error recovery	Relatively low overhead	Very application specific. Dependent on damage assessment and prediction. Inappropriate as a means of recovery for unanticipated damage. Typically higher likelihood of success.
Backward error recovery	A general concept applicable to all systems. Independent of damage assessment, i.e., capable of providing recovery from arbitrary damage. Can be application or system based.	Performance penalty - the overhead to restore a process state can be quite significant. No guarantee that error will not persist when processing is repeated, e.g., permanent fault, software design errors. Some component of the system state may be unrecoverable, e.g., if an error affects an external state.

Checkpoint and Rollback

- **Applicability**
  - When time redundancy is allowed
  - To transient hardware and many software design faults (e.g., timing faults)
  - To both nonredundant and redundant architectures
  - When it is feasible to determine checkpoints in an application
- **Checkpointing**
  - Maintains/saves precise system state or a "snapshot" at regular intervals
    - Snapshot interval can be as small as one instruction
    - Typically, checkpoint interval includes many instructions
    - May not be ideal when there is much error detection latency
- **Rollback recovery**
  - When error is detected
    - Roll back (or restore) process(es) to the saved state, i.e., a checkpoint
    - Restart the computation

Checkpoint and Rollback: What do we need?

- Implement an appropriate error-detection mechanism
  - *Internal to the application:* various self-checking mechanisms (e.g., data integrity, control-flow checking, acceptance tests)
  - *External to the application:* signals (e.g., abnormal termination), missing heartbeats, watchdog timers
- Determine the data to be checkpointed - process state
  - **Volatile states**
    - Program stack (local variables, return pointers of function calls)
    - Program counter, stack pointer, open file descriptors, signal handlers
    - Static and dynamic data segments
  - **Persistent states**
    - User files related to the current program execution (whether to include the persistent state in the process state depends on the application, e.g., the persistent state is often an important part of a long-running application)
- Store the checkpoint data on a stable storage

Checkpoint and Rollback: What do we need? (cont.)

- Determine events to be logged and replayed
  - Messages
  - Events (provoke a message to be sent)
  - Transactions
- Determine checkpoint times based on
  - Elapsed time
  - Message received or sent, e.g., parallel or distributed applications
  - Amount of dirtied state, e.g., database applications
  - Critical function invocation/exit
- Provide procedure to restart the computation
- Provide way to handle a persistent error

Recovery in Distributed/Networked Systems

- Processes cooperate by exchanging information to accomplish a task
  - Message passing (distributed systems)
  - Shared memory (e.g., multiprocessor systems)
- Rollback of one process may require that other processes also roll back to an earlier state.
- All cooperating processes need to establish recovery points.
- Rolling back processes in concurrent systems is more difficult than for a single process due to
  - Domino effect
  - Lost messages
  - Livelocks

Networked/Distributed Systems: Local State

- For a site (computer, process) S_i, its local state LS_i, at a given time is defined by the local context of the distributed application. Let's denote:
  - $send(m_{ij})$ - send event of a message m_{ij} by S_i to S_j
  - $rec(m_{ij})$ - receive event of message m_{ij} by site S_j
  - $time(x)$ - time in which state x was recorded
- We say that
  - $send(m_{ij}) \in LS_i$ iff $time(send(m_{ij})) < time(LS_i)$
  - $rec(m_{ij}) \in LS_j$ iff $time(rec(m_{ij})) < time(LS_j)$
- Two sets of messages are defined for sites S_i and S_j
  - Transit

$transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$

  - Inconsistent

$inconsistent(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \notin LS_i \wedge rec(m_{ij}) \in LS_j\}$

Networked/Distributed Systems: Global State

- A global state (*GS*) of a system is a collection of the local states of its sites, i.e., $GS = \{LS_1, LS_2, \ldots, LS_n\}$, where *n* is the number of sites in the system.
- Consistent global state:

A global state GS is consistent iff

$\forall i, \forall j : 1 \le i, j \le n :: inconsistent(LS_i, LS_j) = \Phi$

- Transitless global state:

A global state GS transitless iff

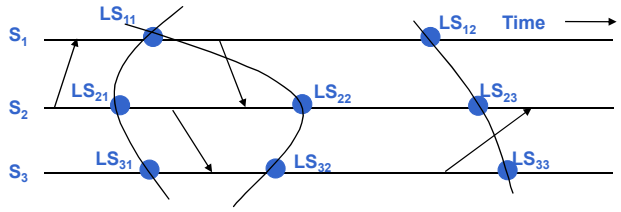
$\forall i, \forall j : 1 \le i, j \le n :: transit(LS_i, LS_j) = \Phi$

All communication channels are empty

- Strongly consistent global state:

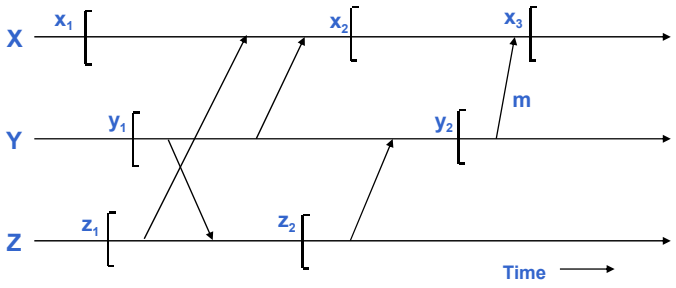
A global state that is both consistent and transitless

Networked/Distributed Systems
Local/Global State - Examples



- What are examples of:
  - Strongly consistent global state
  - Consistent global state
  - Inconsistent global state

Domino Effect



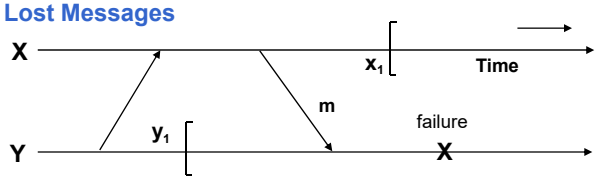
X, Y, Z - cooperating processes

[- recovery points

- **Rollback of X does not affect other processes.**
- **Rollback of Z requires all three processes to roll back to their very first recovery points.**

Lost Messages and Livelocks

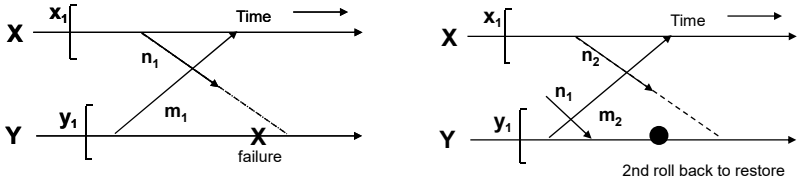
Lost Messages



Message loss due to rollback recovery

Livelocks

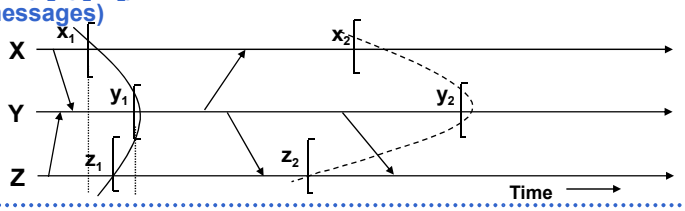
Livelock is a situation in which a single failure can cause an infinite number of rollbacks, preventing the system from making progress



Consistent Set of Checkpoints: Recovery Lines

- A strongly consistent set of checkpoints (recovery line) corresponds to a strongly consistent global state.
  - there is one recovery point for each process in the set during the interval spanned by checkpoints, there is no information flow between any
    - pair of processes in the set
    - a process in the set and any process outside the set
- A consistent set of checkpoints corresponds to a consistent global state.

Set $\{x_1, y_1, z_1\}$ is a strongly consistent set of checkpoints
Set $\{x_2, y_2, z_2\}$ is a consistent set of checkpoints (need to handle lost messages)



Synchronous Checkpointing and Recovery
(Koo & Toueg)

- Assumptions
  - Processes communicate by exchanging messages through communication channels
  - Channels are FIFO
  - End-to-end protocols (such a sliding window) are assumed to cope with message loss due to rollback recovery and communication failures
  - Communication failures do not partition the network
- A single process invokes the algorithm
- The checkpoint and the rollback recovery algorithms are not invoked concurrently.

Synchronous Checkpointing and Recovery
(Koo & Toueg)

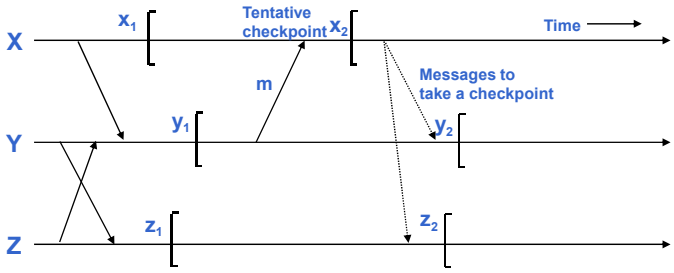
- Two types of checkpoints
  - Permanent - a local checkpoint at a process
  - Tentative - a temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm

Checkpoint Algorithm

- Phase One
  - Initiating process P_i takes a tentative checkpoint and requests that all the processes take tentative checkpoints.
  - Each process informs P_i whether it succeeded in taking a tentative checkpoint.
  - If P_i learns that all processes have taken tentative checkpoints, P_i decides that all tentative checkpoints should be made permanent.
  - Otherwise, P_i decides that all tentative checkpoints should be discarded.
- Phase Two
  - P_i propagates its decision to all processes.
  - On receiving the message from P_i , all processes act accordingly.
  - No process sends message after taking a tentative checkpoint till phase 2 is completed.

Checkpoint Algorithm (cont.)

- Optimization of the checkpoint algorithm
  - A minimal number of processes take checkpoints
  - Processes use a labeling scheme to decide whether to take a checkpoint.
  - All processes from which P_i has received messages after it has taken its last checkpoint take a checkpoint to record the sending of those messages



Labeling Scheme

- Each process uses monotonically increasing labels in its outgoing messages

  $\perp$ = smallest label
  T = largest label

- For any two processes X and Y, let m be the last message that X received from Y after X has taken its last permanent or tentative checkpoint then

$$last\_label\_rcvd_X[Y] = \begin{cases} m.l. & \text{if m exists} \\ \perp & \text{otherwise} \end{cases}$$

- Let m be the first message that X sent to Y after X took its last permanent or tentative checkpoint then

$$first\_label\_sent_X[Y] = \begin{cases} m.l. & \text{if m exists} \\ \perp & \text{otherwise} \end{cases}$$

Labeling Scheme (cont.)

- When X requests Y to take a tentative checkpoint, X sends $last\_label\_rcvd_X[Y]$ along with its request; Y takes a tentative checkpoint only if

$$\mathbf{last\_label\_rcvd_X[Y] \geq first\_label\_sent_Y[X] > \perp}$$

- Checkpoint cohort - Set of all processes that should be asked to take a checkpoint initiated by X

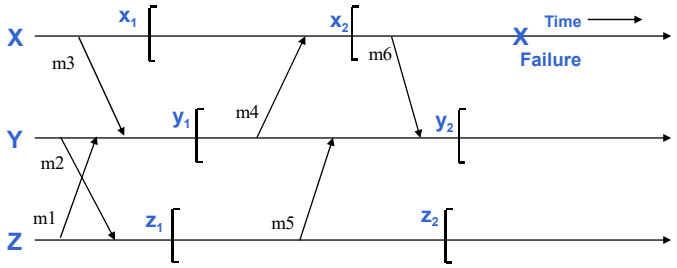
$$\mathbf{ckpt\_cohort_X = \{Y \mid last\_label\_rcvd_X[Y] > \perp\}}$$

Rollback Recovery Algorithm

- Phase One:
  - Process P_i checks whether all processes are willing to restart from their previous checkpoints.
  - A process may reply "no" if it is already participating in a checkpointing or recovering process initiated by some other process.
  - If all processes are willing to restart from their previous checkpoints, P_i decides that they should restart.
  - Otherwise, P_i decides that all the processes continue with their normal activities.
- Phase Two:
  - P_i propagates its decision to all processes.
  - On receiving P_i's decision, the processes act accordingly.

Rollback Recovery Algorithm (cont.)

- Optimization
  - A minimum number of processes roll back
  - Processes use a labeling scheme to decide whether they need to roll back
  - Y will restart from its permanent checkpoint only if *X* is rolling back to a state where the sending of one or more messages from *X* to *Y* is being undone



Labeling Scheme - extension

- For any two processes X and Y, let m be the last message that X sent to Y before its last permanent checkpoint. Then

$$last\_label\_sent_X[Y] = \begin{cases} m.l & \text{if m exists} \\ T & \text{otherwise} \end{cases}$$

- When X requests Y to restart from the permanent checkpoint, it sends $last\_label\_sent_X[Y]$ along with this request
- Y will restart from its permanent checkpoint only if

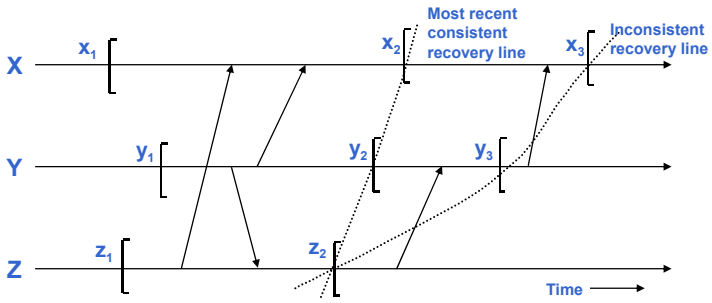
$$\mathbf{last\_label\_rcvd_Y[X] > last\_label\_sent_X[Y]}$$

Synchronous Checkpointing
Disadvantages

- Additional messages must be exchanged to coordinate checkpointing.
- Synchronization delays are introduced during normal operations.
  - No computational messages can be sent while the checkpointing algorithm is in progress.
- If failure rarely occurs between successive checkpoints, then the checkpoint algorithm places an unnecessary extra load on the system, which can significantly affect performance.

Asynchronous Checkpointing and Recovery

- Checkpoints at each process are taken independently without any synchronization among the processors.
- There is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints.
- The recovery algorithm must search for the most recent consistent set of checkpoints before it initiates recovery.



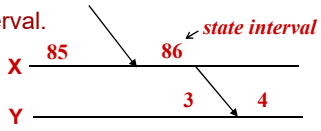
Asynchronous Checkpointing and Recovery (cont.)

- All incoming messages are logged at each process.
  - This minimizes the amount of computation to undo during a rollback.
  - The messages received after setting the recovery point can be processed again.
- Message logging
  - Pessimistic: An incoming message is logged before it is processed
    - This slows down the computation, even when there are no failures.
  - Optimistic: Processors continue to perform the computation, and the message received are stored in volatile storage and logged at certain intervals.
    - Messages that are not logged (stored on stable storage) can be lost in the event of rollback.
    - This does not slow down the underlying computation.

Optimistic Message Logging

- Messages not necessarily logged before being processed.
- Unlogged messages are not available during recovery.
- States in other processes that causally depend upon lost messages are called *orphan states*.
- Processes that have orphan states must rollback.

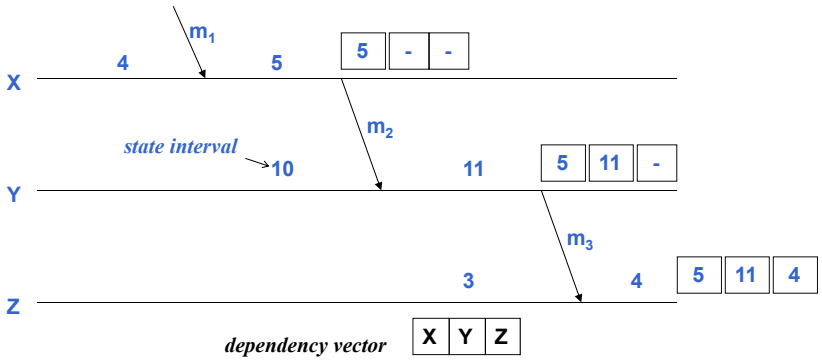
- Dependencies tracked through *state intervals*:
  - Process consists of sequence of state intervals.
  - Receipt of message starts a new state interval.
  - Outgoing messages dependent upon current state interval of a process



Optimistic Message Logging (cont.)

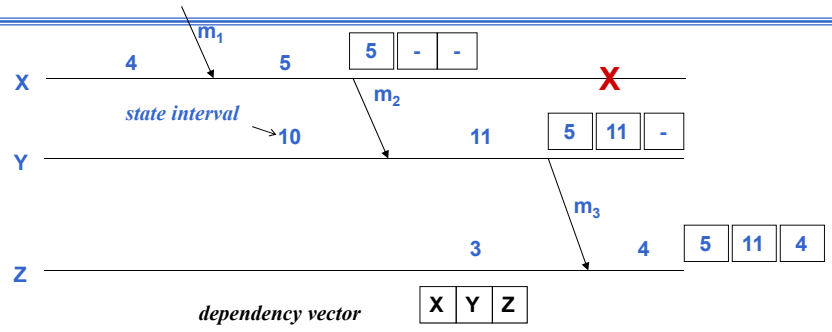
- Each process keeps a *dependency vector*:
  - One entry per process in the system.
  - Entry for process *j* specifies latest state interval in process *j* on which the process is dependent.
- Dependency vector piggybacked on outgoing messages.
- Receivers update their own dependency vector from piggybacked vector.
- Causal dependencies propagated through piggybacked vector.

Piggybacked Dependency Vector



- Example shows dependency vector being updated as time progresses.
- Dependency vector of Z after receipt of m_3 shows that Z is dependent upon state 5 of X and state 11 of Y.

Recovery

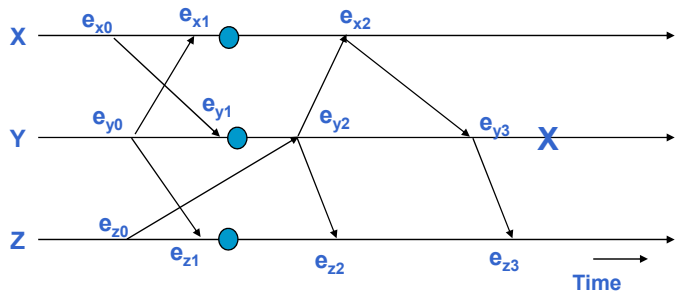


- X fails; if X has not logged m_1 to disk at time of failure, then m_1 is unrecoverable.
- Cannot guarantee that state 5 of X can be recreated exactly as before.
- All states dependent on state 5 of X are orphan states.
- When X recovers, it broadcasts to other processes that it can recreate its state up to state 4.
- Other processes check their dependency vectors and rollback if they are dependent on a state interval of X greater than 4.

Asynchronous Checkpoint and Recovery Algorithm: An Example

- Communication channels are reliable.
- Messages are delivered in the order in which they were sent.
- Each process keeps track of the number of messages that were
  - Sent to other processes
  - Received from other processes
- A process, upon restarting (after failure) broadcasts a message that it had failed.
- All processes determine orphan messages by comparing the numbers of messages sent and received.
- The process rolls back to a state where the number of messages received (at the process) is not greater than the number of messages sent (according to the state at other processes).

Asynchronous Checkpoint and Recovery Algorithm: An Example (cont.)



If Y rolls back to a state e_{y1}, then

- Y has sent only one message to X
- X has received two messages from Y thus far
- X must roll back to a state preceding e_{x1} (to be consistent with Y's state)
- For similar reasons, Z must also roll back

References

- D. K. Pradhan, “Fault Tolerant Computer System Design”, Chapter 3.10 (“Forward recovery”).
- Singhal and Shivaratri, “Advanced Concepts in Operating Systems”, Chapter 12: Recovery (“Backward recovery”).
- Synchronous Checkpointing: “Checkpointing and Rollback-Recovery for Distributed Systems” by R. Koo and S. Toueg, IEEE Transactions on Software Engineering, Jan 1987, pp. 23-31.
- Asynchronous Checkpointing: “Crash Recovery with Little Overhead” by T. Juang and S. Venkatesan, 11th International Conference on Distributed Computing Systems, May 1991, pp. 454-461.