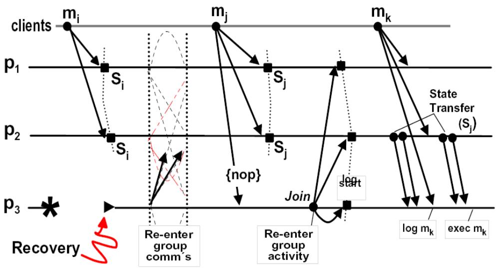
8. Recovery

Replica Recovery

- Consider a replicated service in which replicas can crash and recover (crash-recovery fault model)
- Recovery from crashes requires that the recovering replica obtains current state from other replicas
- Without stable storage:
  - Cooperative recovery from other replica(s) without stable storage
  - **Complete state transfer** can make recovery very slow
  - Lose state if all replicas fail
- With stable storage:
  - Recover some past state before failure
  - Cooperative recovery of missing commands from other replica(s)
  - Execute commands from the log until current state
  - State recovery is much faster

Recovery with state transfer



- Recovering replica (p3) starts by resuming communication with the replica set, e.g. if the set was using some form of group communication
- It starts receiving all messages, but still discards them
- Next, sends a request to join the replica group, delivered in total order to all replicas, including the joining replica, marking a **cut Sj** in the global system state
- The join request triggers a state-transfer operation (after join is completed)
- p2 checkpoint its state at this point (Sj), and sends it to p3
- p3 starts logging any messages that arrive after the cut Sj
- New requests (m_k) can continue to be processed by all replicas except p3

Checkpoint-based Rollback-recovery

(Scope and basics)

- Rollback-recovery is mostly suited for long running applications
  - Scientific applications (simulation, optimization)
  - Telecommunication applications
- Used in local and distributed computations
- Requires stable storage that survives failures
- State information is periodically saved (checkpoints)
- Other information may also be saved (logs)

- After a failure, the saved information can be used to restart the computation from an intermediate state

Issues in checkpointing

- At what level should we checkpoint?
  - Should it be made transparent to the user?
  - What are the trade-offs involved in the decision?
- How many checkpoints should be done?
  - Frequently?
  - Sparingly?
- When should they be done?
  - Upon every event, or only when some events occur?
- How to minimize checkpoint overhead?
- How to do distributed checkpoints?
  - Coordinated?
  - Uncoordinated?

Checkpointing approaches

- Uncoordinated checkpoints
  - Every process takes a checkpoint independently from other processes
  - Since there is no coordination, it may be possible that checkpoints are not consistent across system (domino effect)
- Coordinated checkpoints
  - Processes coordinate to meet a consistent global state before taking the checkpoint
  - Only the last checkpoint needs to be kept
- Communication induced checkpoints
  - Processes take checkpoints independently
  - But they select when to take a checkpoint, to avoid the domino effect (see next slide)

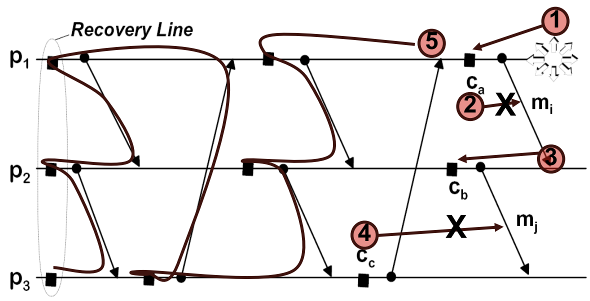
Domino effect

(Message-passing systems)

- What is the domino effect?
  - Rollback of one process meets an inconsistent global state, forces rollback of another process, which in turn forces first process to rollback again, and so on
- This can only happen with independent checkpointing
- The solution is to use coordinated checkpointing
  - A system-wide consistent state is saved
  - The Chandy-Lamport algorithm is a widely used solution

Domino effect

(illustrated)



1. p_1 fails, and then recovers, rolling back to checkpoint C_a
2. Evidence of sending message m_i no longer exists
3. So, p_2 is forced to rollback to checkpoint C_b
4. However, this “unsends" message m_j and p_3 is forced back to C_c
5. Rollback propagation will bring system back to initial state

Logging

- Checkpointing allows recovering from a past state
- If done frequently:
  - No need to go too far back in state
  - Redoing undone actions will be faster
- but
  - Overhead of checkpointing can become unbearable
  - The problem is amplified if state is large
- Solution: **reduce frequency of checkpoints!**

- Unfortunately this may not be possible if some actions
  - Leave traces outside the system
  - Are not deterministic

Log-based rollback-recovery

- The solution to 1) avoid frequent checkpoints and 2) ensure deterministic recovery is to log all non-deterministic events
- The state can then be reconstructed from the most recent checkpoint + log
  - All the non-deterministic events are replayed in their exact order

- Major approaches to log-based rollback-recovery:
  - Pessimistic logging: every event is logged before taking effect
  - Optimistic logging: asynchronous logs, which may be lost
  - Causal logging: keeps track of causal relations among events

Log-based rollback-recovery

- Assumes that non-deterministic events can be logged
  - E.g., Received messages or internal events
    - Sent messages are deterministic events!
  - The execution is a sequence of deterministic state intervals

- Some checkpoints are taken to avoid long roll-backs
- After failure the state is recovered using checkpoints and logged information of non-deterministic events
- After recovery there will be no orphan processes
  - No process will be in a state that depends on a non-reproducible non-deterministic event

Log-based rollback-recovery
(Protocols)

- Pessimistic log-based rollback-recovery
  - Guarantee that orphans are not created
  - Simple recovery, garbage collection and output commit
  - High performance overhead
- Optimistic log-based rollback-recovery
  - Reduced performance overhead
  - Orphans may be created
  - Recovery, garbage collection and output commit may be harder
- Causal log-based rollback-recovery
  - Guarantee that orphans are not created
  - Reduced performance overhead and fast output commit
  - Complex recovery and garbage collection