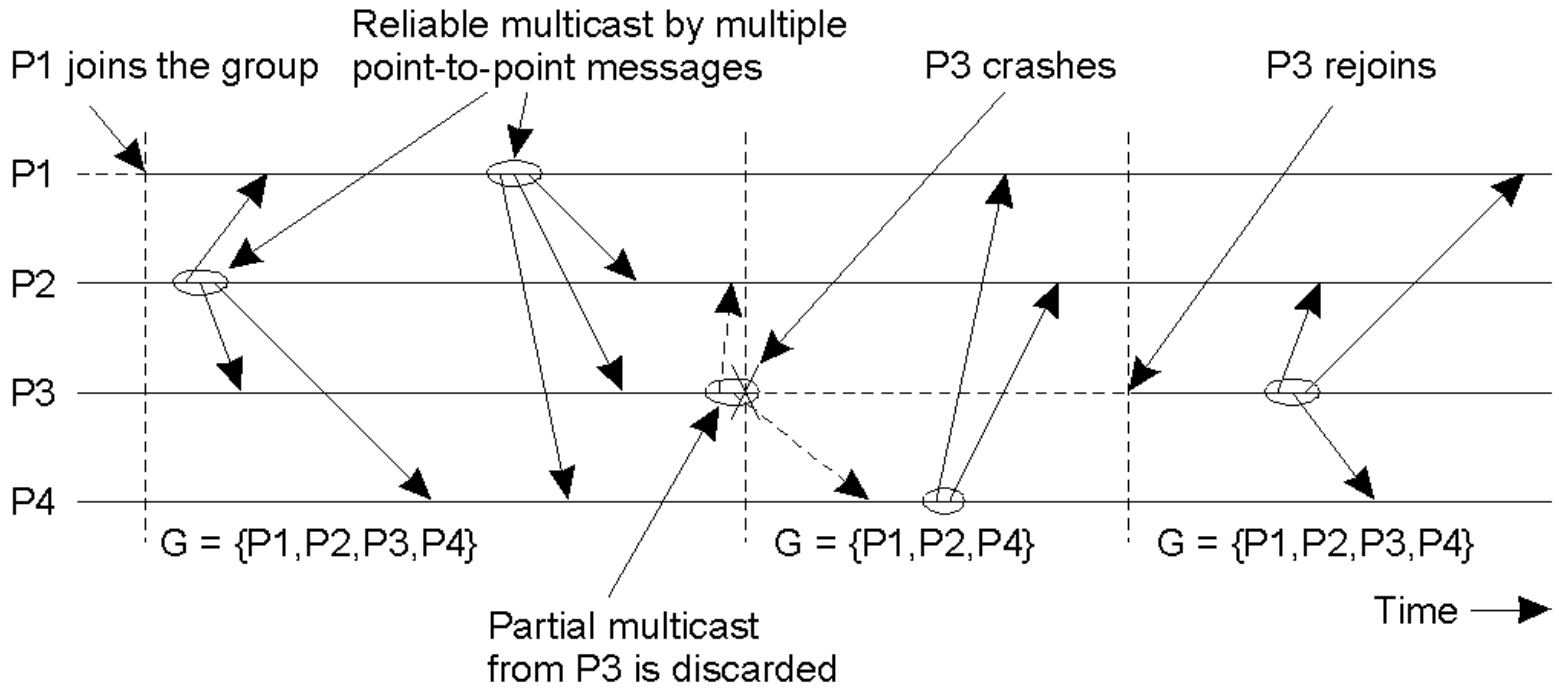
ECE151 – Lecture 12

Chapter 7
Fault Tolerance

Virtual Synchrony



The principle of virtual synchronous multicast.

Guarantee: A message is delivered only to the nonfaulty members of the current group. All members should agree on the current group membership.

Message Ordering

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

Three communicating processes in the same group.
The ordering of events per process is shown along the vertical axis.

Message Ordering

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

Virtual Synchrony

Essence: We consider **views** $V \subseteq RCV(c) \cup SND(c)$

Processes are added or deleted from a view V through **view changes** to V^*; a view change is executed *locally* by each $P \in V \cap V^*$

(1) For each consistent state, there is a **unique view** on which all its members agree. **Note:** implies that all nonfaulty processes see all view changes in the same order

(2) If message m is sent to V before a view change vc to V^*, then either all $P \in V$ that excute vc receive m, or no processes $P \in V$ that execute vc receive m. **Note:** all nonfaulty members in the same view get to see the same set of multicast messages.

(3) A message sent to view V can be delivered only to processes in V, and is discarded by successive views

A reliable multicast algorithm satisfying (1)–(3) is **virtually synchronous**

Virtual Synchrony

A sender to a view V need not be member of V

If a sender $S \in V$ crashes, its multicast message m is *flushed* before S is removed from V: m will never be delivered after the point that $S \notin V$

Note: Messages from S may still be delivered to all, or none (nonfaulty) processes in V before they all agree on a new view to which S does not belong

If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Or we may decide to deliver m to members in V - P

Observation: Virtually synchronous behavior can be seen independent from the ordering of message delivery.

The only issue is that messages are delivered to an *agreed upon* group of receivers

Virtual Synchrony Implementation

The current view is known at each P by means of a delivery list DEST[P]

If $P \in$ DEST[P] then $Q \in$ DEST[P]

Messages received by P are queued in QUEUE[P]

If P fails, the group view must change, but not before all messages from P have been flushed

Each P attaches a (stepwise increasing) **timestamp** with each message it sends

Assume FIFO-ordered delivery; the highest numbered message from Q that has been received by P is recorded in RCVD[P]

The vector RCVD[P] is sent (as a control message) to all members in DEST[P]

Each P records RCVD[P] in REMOTE[P][Q]

Virtual Synchrony Implementation

Observation: REMOTE[*P*][*Q*] shows what *P* knows about message arrival at *Q*

1	2 3 1 5
2	2 2 2 4
3	3 1 4 5
4	4 2 2 4
min	2 1 1 4

A message is **stable** if it has been received by all *Q* (shown as the **min** vector)

Stable messages can be delivered to the next layer (which may deal with ordering). **Note:** Causal message delivery is free

As soon as all messages from the faulty process have been flushed, that process can be removed from the (local) views

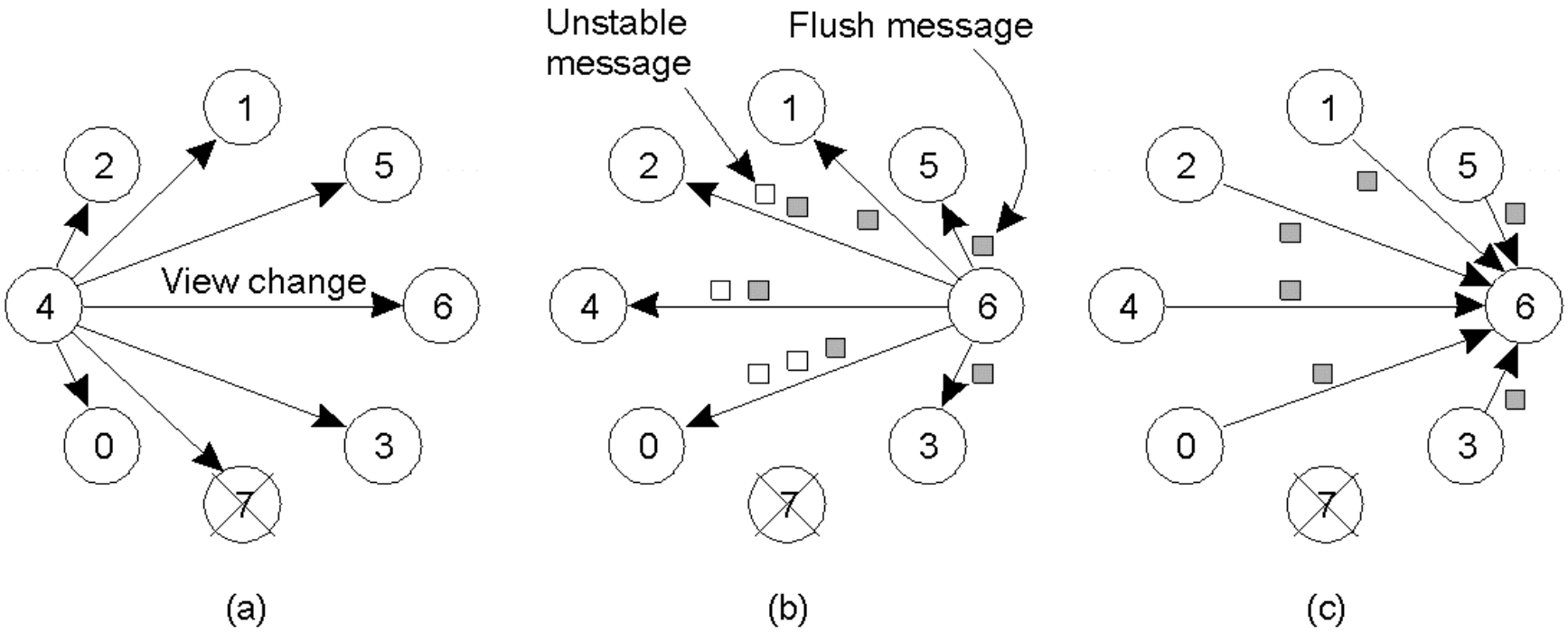
Virtual Synchrony Implementation

Remains: What if a sender *P* failed and not all its messages made it to the nonfaulty members of the current view?

Solution: Select a coordinator which has all (unstable) messages from *P*, and forward those to the other group members.

Note: Member failure is assumed to be detected and subsequently multicast to the current view as a view change. That view change will not be carried out before all messages in the current view have been delivered.

Implementing Virtual Synchrony (2)



a) Process 4 notices that process 7 has crashed, sends a view change

b) Process 6 sends out all its unstable messages, followed by a flush message

c) Process 6 installs the new view when it has received a flush message from everyone else

Implementing Virtual Synchrony

Six different versions of virtually synchronous reliable multicasting.

Multicast	Basic Message Ordering	Total-ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Distributed Commit

Two-phase commit

Three-phase commit

Essential issue: Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (**atomicity**)?

Two-Phase Commit

Model: The client who inititated the computation acts as coordinator; processes required to commit are the participants

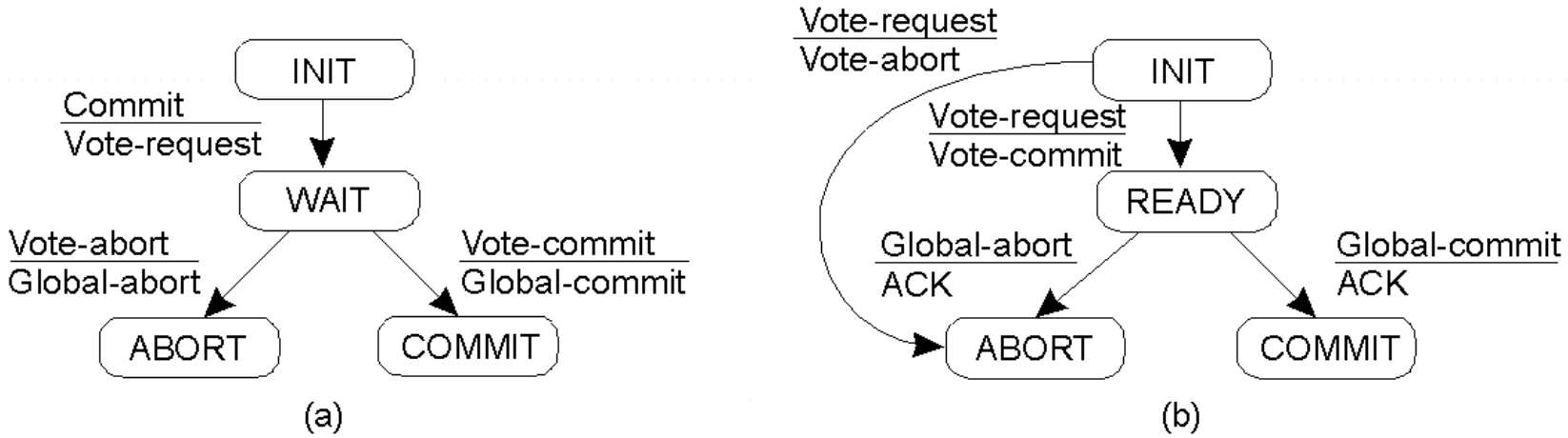
Phase 1a: Coordinator sends VOTE_REQUEST to participants (also called a **pre-write**)

Phase 1b: When participant receives VOTE_REQUEST it returns either YES or NO to coordinator.
If it sends NO, it aborts its local computation

Phase 2a: Coordinator collects all votes;
if all are YES, it sends COMMIT to all participants, otherwise it sends ABORT

Phase 2b: Each participant waits for COMMIT or ABORT and handles accordingly.

Two-Phase Commit



a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.

Two-Phase Commit

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.

Two-Phase Commit

actions by coordinator:

```
while START _2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Outline of the steps taken by the coordinator in a two phase commit protocol

Two-Phase Commit

Steps taken by participant process in 2PC.

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

Two-Phase Commit

actions for handling decision requests: /* executed by separate thread */

```
while true {
   wait until any incoming DECISION_REQUEST is received; /* remain blocked */
   read most recently recorded STATE from the local log;
   if STATE == GLOBAL_COMMIT
      send GLOBAL_COMMIT to requesting participant;
   else if STATE == INIT or STATE == GLOBAL_ABORT
      send GLOBAL_ABORT to requesting participant;
   else
      skip;  /* participant remains blocked */
```

Steps taken for handling incoming decision requests.

2PC – Failing Participant

Observation: Consider participant crash in one of its states, and the subsequent recovery to that state:

Initial state: No problem, as participant was unaware of the protocol

Ready state: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make => log the coordinator's decision

Abort state: Merely make entry into abort state *idempotent*, e.g., removing the workspace of results

Commit state: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation: When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.

2PC – Failing Coordinator

Observation: The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually lost)

Alternative: Let a participant *P* in the ready state timeout when it hasn't received the coordinator's decision;

P tries to find out what other participants know.

Question: Can *P not* succeed in getting the required information?

Observation: Essence of the problem is that a recovering participant cannot make a **local** decision: it is dependent on other (possibly failed) processes

Three-Phase Commit

Phase 1a: Coordinator sends VOTE_REQUEST to participants

Phase 1b: When participant receives VOTE_REQUEST it returns either YES or NO to coordinator.
If it sends NO, it aborts its local computation

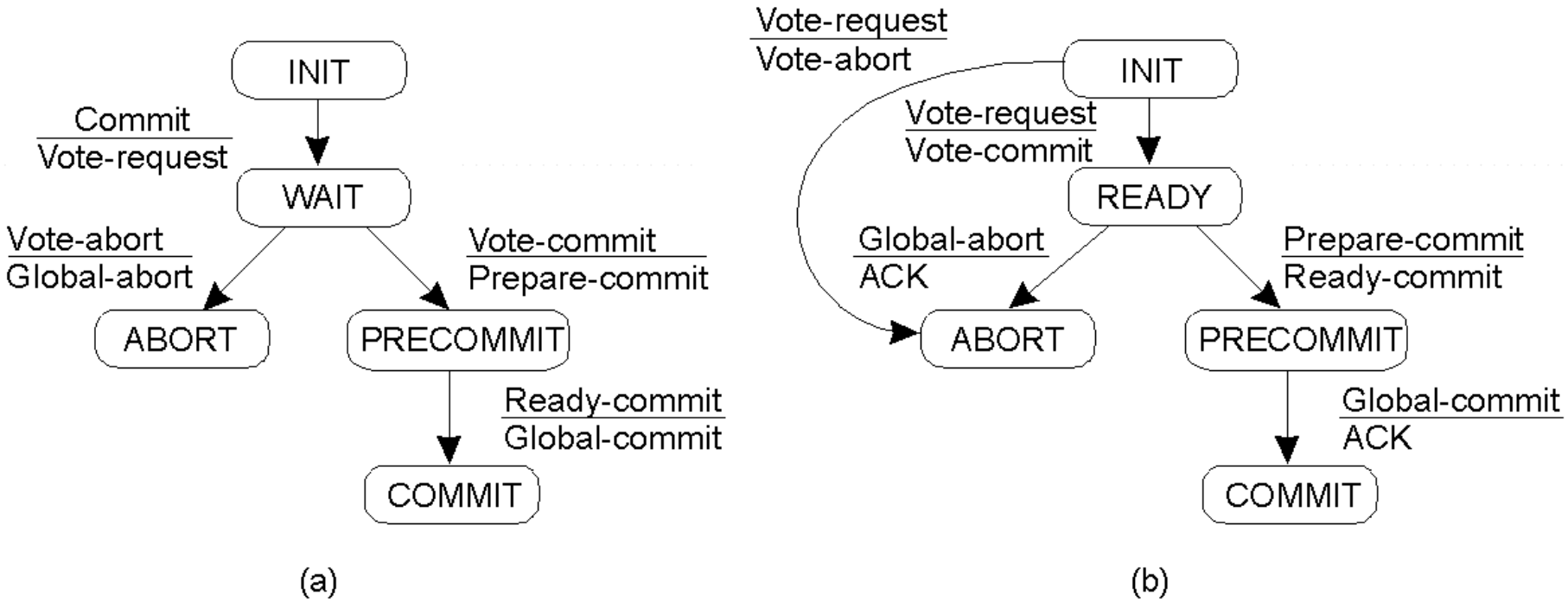
Phase 2a: Coordinator collects all votes;
if all are YES, it sends PREPARE to all participants, otherwise it sends ABORT, and halts

Phase 2b: Each participant waits for PREPARE, or waits for ABORT after which it halts

Phase 3a: (Prepare to commit) Coordinator waits until all participants have ACKed receipt of PREPARE message, and then sends COMMIT to all

Phase 3b: (Prepare to commit) Participant waits for COMMIT

Three-Phase Commit



a) Finite state machine for the coordinator in 3PC

b) Finite state machine for a participant

3PC – Failing Participant

Basic issue: Can *P* find out what it should it do after crashing in the ready or pre-commit state, even if other participants or the coordinator failed?

Essence: Coordinator and participants on their way to commit, never differ by more than one state transition

Consequence: If a participant timeouts in ready state, it can find out at the coordinator or other participants whether it should abort, or enter pre-commit state

Observation: If a participant already made it to the pre-commit state, it can always safely commit (but is not allowed to do so for the sake of failing other processes)

Observation: We may need to elect another coordinator to send off the final COMMIT

Recovery

Introduction

Checkpointing

Message Logging

Recovery: Background

Essence: When a failure occurs, we need to bring the system into an error-free state:

Forward error recovery: Find a new state from which the system can continue operation

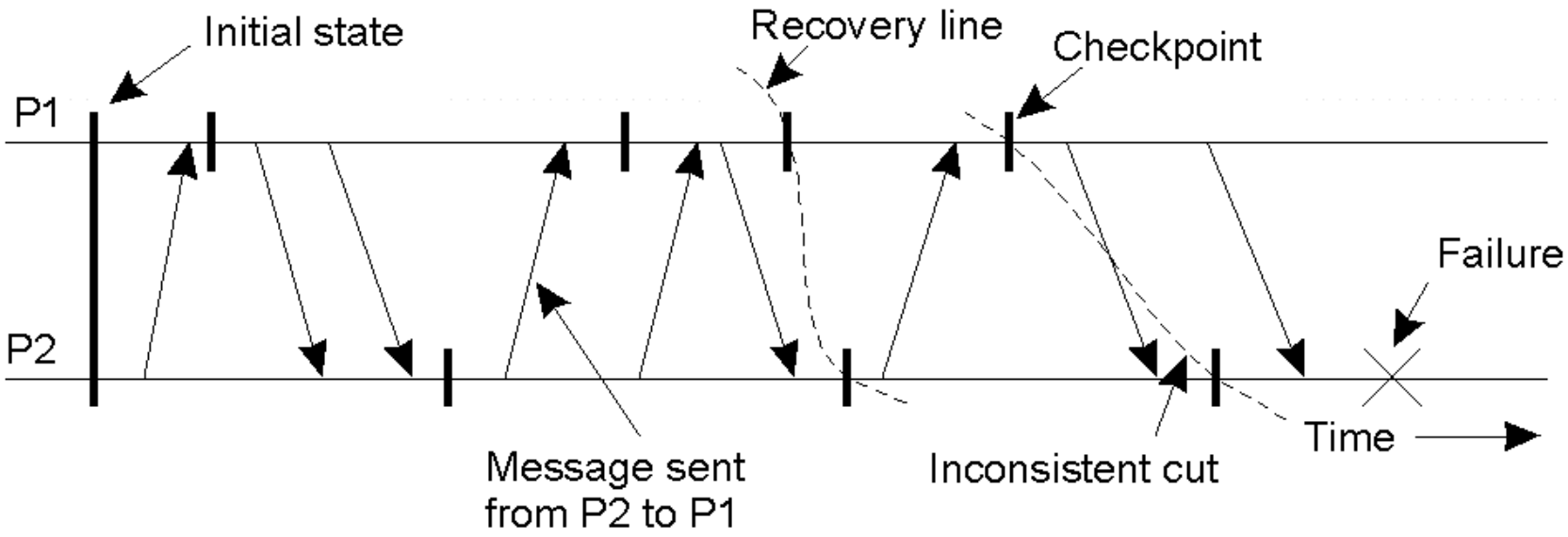
Backward error recovery: Bring the system back into a *previous* error-free state

Practice: Use backward error recovery, requiring that we establish **recovery points**

Observation: Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a **consistent state** from where to recover

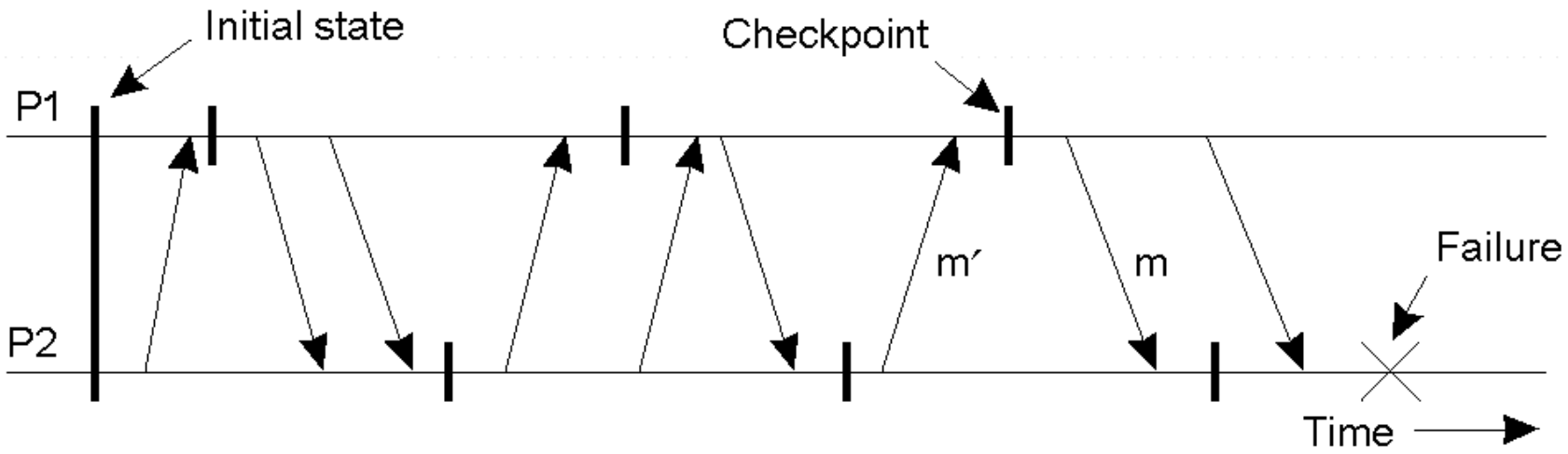
Checkpointing

A recovery line.



Recovery line: Assuming processes regularly **checkpoint** their state, the most recent **consistent global checkpoint**.

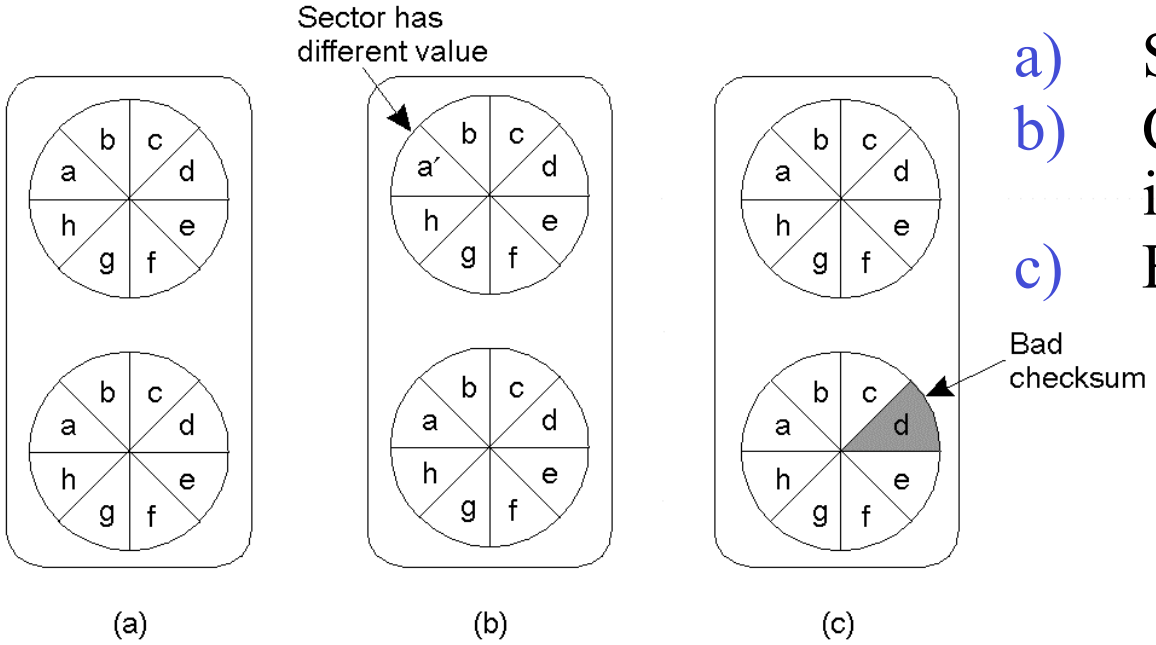
Cascaded Rollback



The domino effect.

Observation: If checkpointing is done at the "wrong" instants, the recovery line may lie at system startup time => **cascaded rollback**

Recovery Stable Storage



a) Stable Storage
b) Crash after drive 1 is updated
c) Bad spot

After a crash:

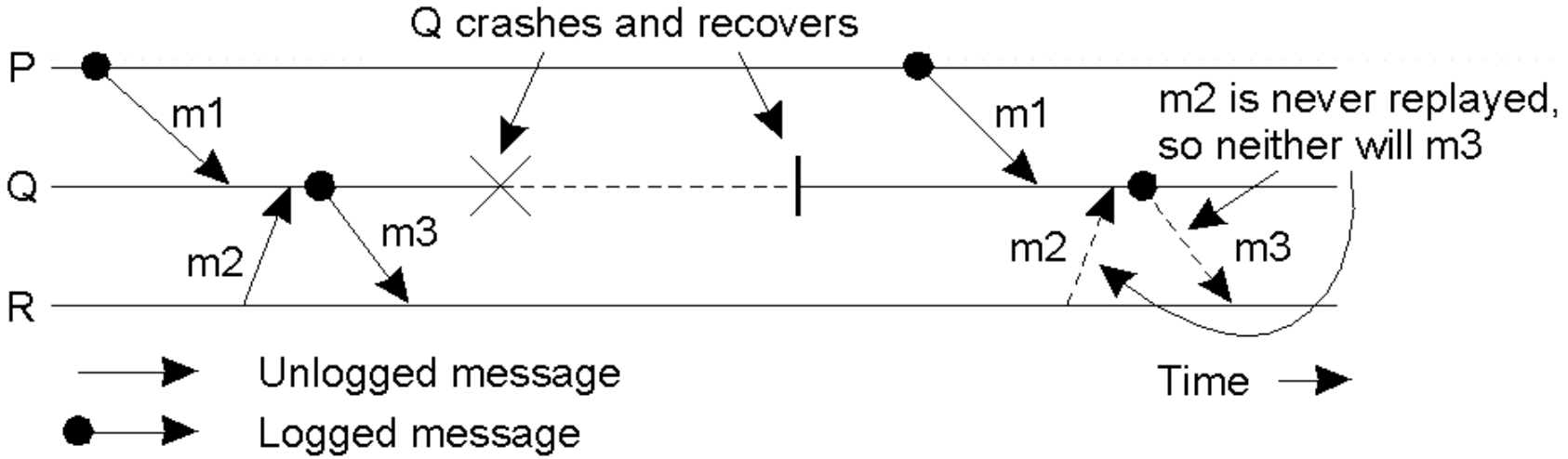
If both disks are identical: you're in good shape.

If one is bad, but the other is okay (checksums):
choose the good one.

If both seem okay, but are different:
choose the main disk.

If both aren't good: you're **not** in a good shape.

Message Logging



Incorrect replay of messages after recovery,
leading to an orphan process.