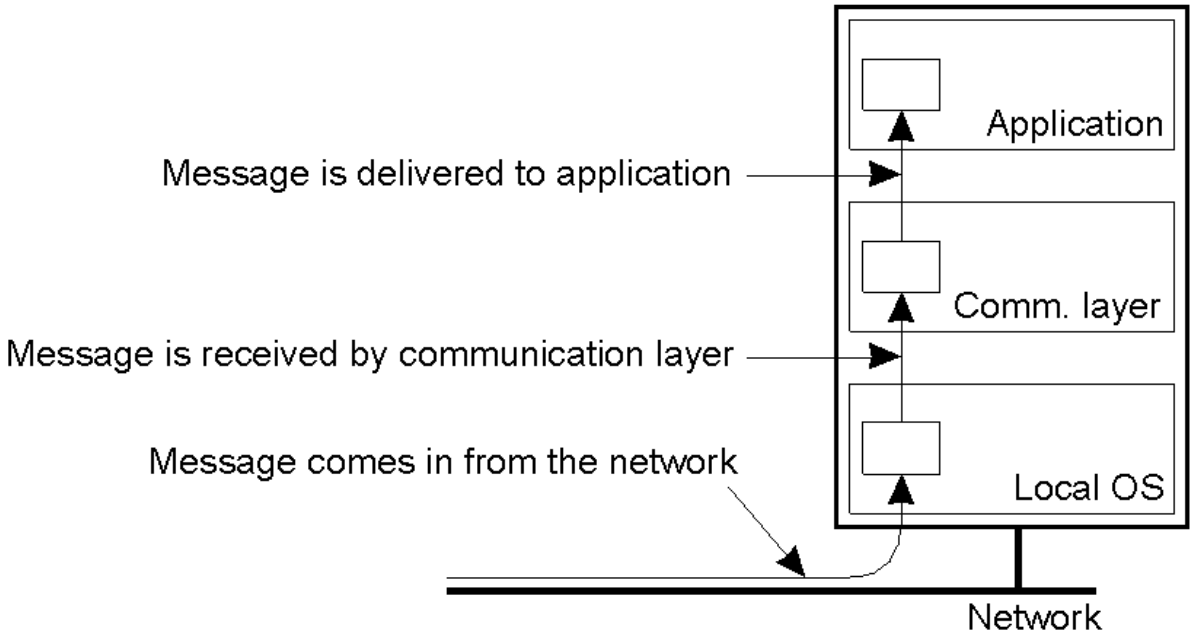
Distributed Systems Lecture 11

Professor Louise E. Moser

Spring 2013

Virtual Synchrony

- **Idea:** Formulate *multicasting*, in the presence of *process failures*, in terms of *process groups* and changes to *group membership*
- Distinguish between *message receipt* and *message delivery*



Virtual Synchrony

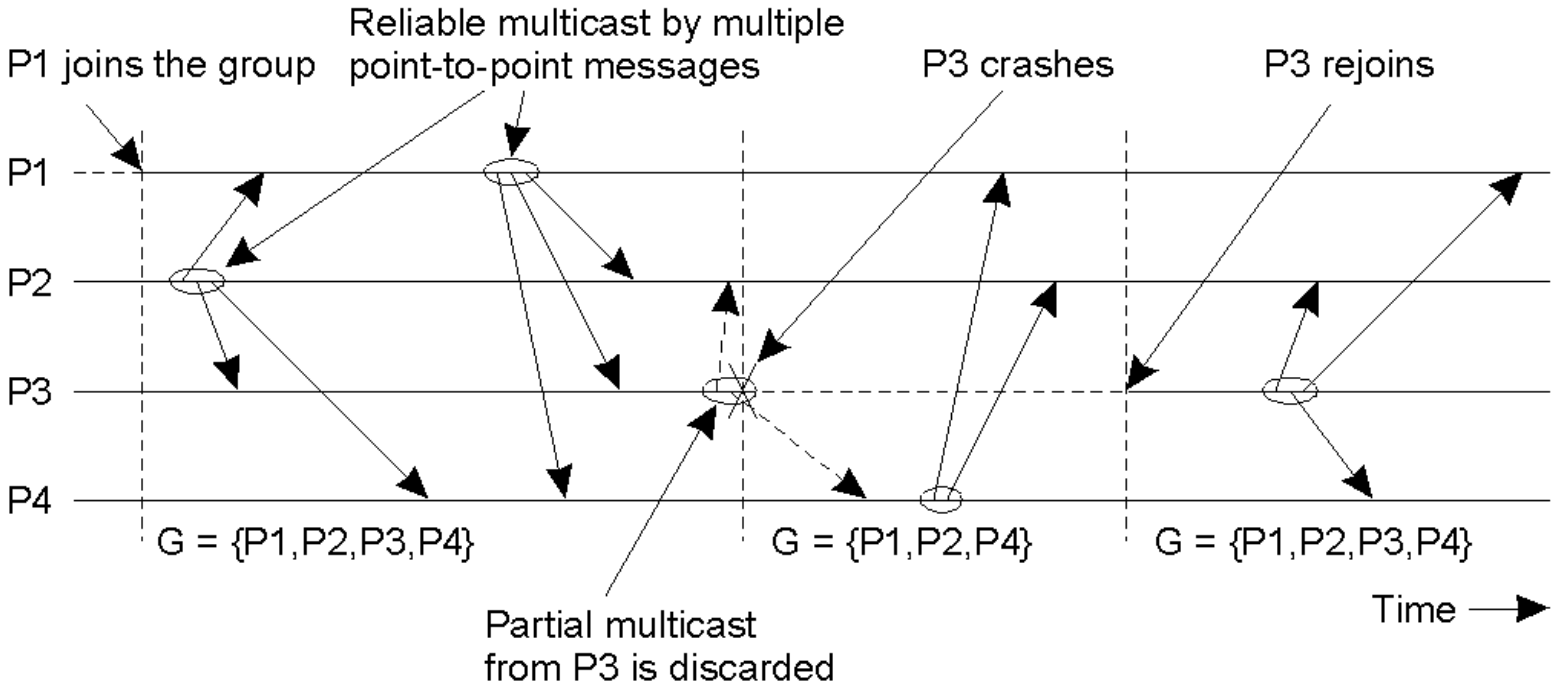
- **Group view:** View *V* of the set of processes in the group that the sender had when it multicast a message *m* to the group
  - **A group view is a list of processes; each process in the list should have the same view of the group**
- **Atomic multicast:** Message *m* should be delivered to *all* processes in view *V* or to *none* of them
- **View change** occurs when a process *joins* the group, or *leaves* the group (either intentionally or due to failure)
  - **View change message *vc* is multicast to announce a process's joining or leaving**

Virtual Synchrony

- Suppose, while message *m* is being multicast to group view *V*, a view change occurs and a view change message *vc* is sent
- Message *m* should be delivered to *all* processes in *V* or to *no* processes in *V* and that should happen before *vc* is delivered
  - **Delivery to *no* processes in *V* is permitted only if the sender of message *m* crashes**
- **Virtual synchronous multicast** (a stricter requirement):
  If the sender of message *m* crashes, then message *m* must be delivered to *all non-faulty* processes in *V* and that must happen before the view change message *vc* is delivered
  - **Thus, all multicasts take place between view changes**
  - **All multicasts in transit while a view change is taking place must be completed before the view change takes effect**

Virtual Synchrony

- **Virtual synchronous multicast**



Message Ordering

- Four different kinds of message ordering for virtual synchronous multicast
  - **Unordered multicast**
  - **FIFO (source) ordered multicast**
  - **Causally ordered multicast**
  - **Totally ordered multicast**

Message Ordering

- **Unordered multicast:** Three processes in the same group, with one sender P1, and two receivers P2 and P3 that receive the messages from P1 in different orders

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

Message Ordering

- **FIFO multicast:** Four processes in the same group with two senders P1 and P4, and two receivers P2 and P3 that receive the messages from the same process P1 (or P4) in the order in which P1 (or P4) sent them but that receive messages from different processes P1 and P4 in different orders

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

Message Ordering

- **Causally ordered multicast:** If message m1 causally precedes message m2 (regardless of whether or not they were multicast by the same sender), then m1 is delivered before m2 is delivered
  - **Causally ordered multicast can be implemented using vector timestamps or a more sophisticated scheme (Trans, Melliar-Smith and Moser)**
- **Totally ordered multicast:** Messages are delivered in the same order (linear sequence) to all members in a group view (regardless of whether message delivery is unordered, FIFO ordered, or casually ordered)
  - **Totally ordered multicast can be implemented using a centralized sequencer (Amoeba, Tanenbaum, et al) or a rotating sequencer (Totem, Moser, Melliar-Smith, et al)**

Virtual Synchrony and Message Ordering

- Six different versions of virtually synchronous multicast

Multicast	Basic Message Ordering	Total Ordered Delivery
Unordered 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

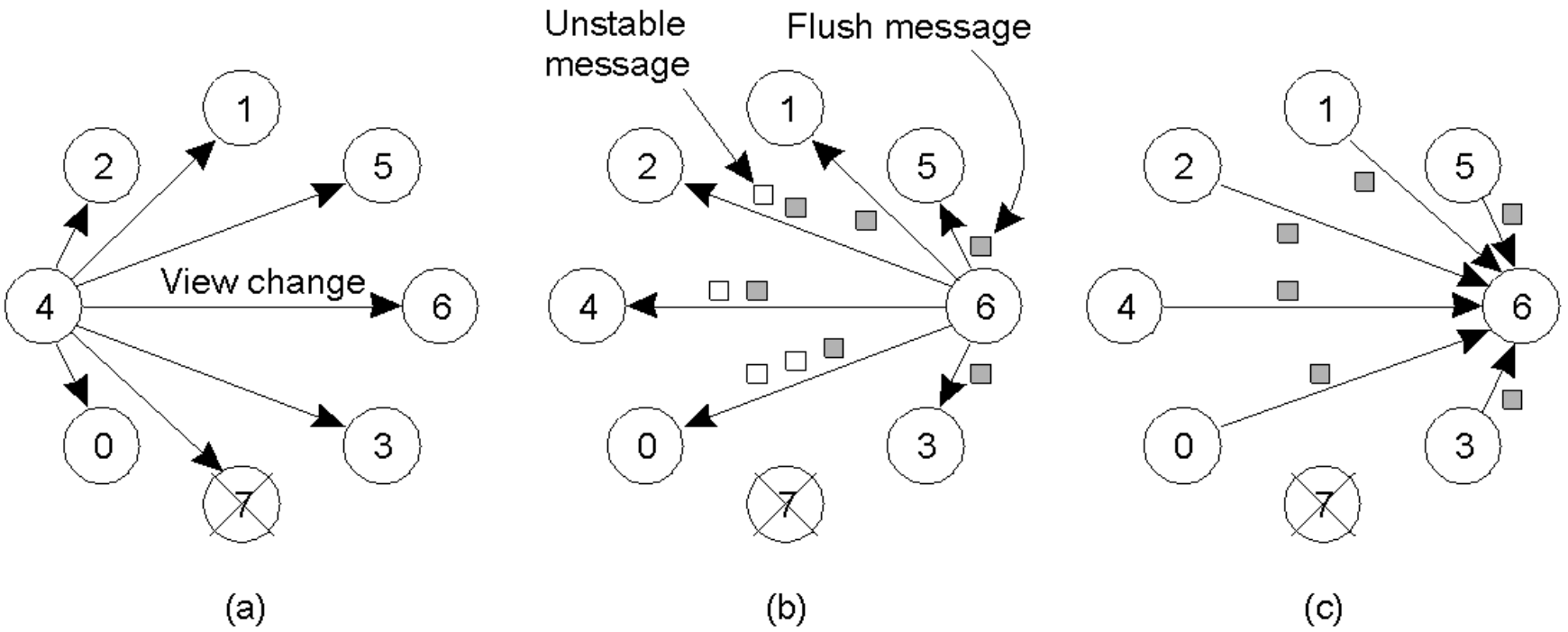
Implementing Virtual Synchrony

- Processes are added to, or removed from, a group view V by a process locally when it receives a view change message *vc*
- All members of a group view V must agree on the group view
  - **All non-faulty members see all group view changes in the same order**
- If message *m* is sent to group view V before a view change *vc*, then all P in V that deliver *vc* deliver *m*, or no processes P in V that deliver *vc* deliver *m*
  - **All non-faulty members of a group view see the same set of multicast messages**
- A message *m* sent to a group view V is delivered only to the members of V
  - **The message is discarded by members in successive views**

Implementing Virtual Synchrony

- **Question:** What if the sender of a message *m* fails before it completes its multicast?
- **Answer:** The processes that have not yet received message *m* should get it from other processes that did receive *m*
- A message is **stable** if it has been received by *all* processes in the group view to which the message was sent
  - **A process must buffer a message until it knows that the message is stable, because it might have to retransmit it**
  - **When a message is stable, a process can remove (flush) the message from its buffer and deliver the message to the higher layer (which might handle message ordering)**
- When all messages from the faulty sender are stable and have been flushed, a process can remove the faulty process from its group view

Implementing Virtual Synchrony



(a) Process 4 detects that process 7 has crashed, and sends a view change

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

(c) Process 6 installs the new view when it has received a flush message from all other processes

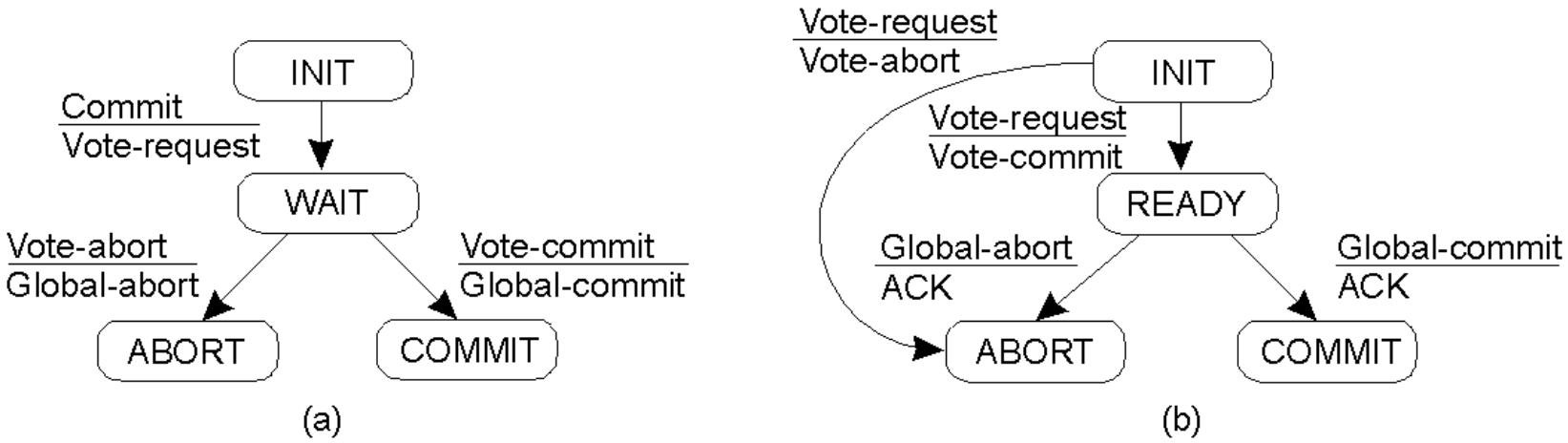
Distributed Commit

- **Two-phase commit**
- **Three-phase commit**
- **Essential issue:** Given a computation that is distributed across processes in a process group, how can we ensure that either *all* processes commit to the final result, or *none* of them do (i.e., how can we ensure **atomicity**)?

Two-Phase Commit

- **Model:** The client that initiates the computation acts as the **coordinator**. The processes required to commit to the final result are the **participants**
- **Phase 1a (Voting):** The coordinator sends VOTE_REQUEST to the participants
- **Phase 1b (Voting):** When a participant receives VOTE_REQUEST, the participant returns either YES or NO to the coordinator. If it sends NO, the participant aborts its local computation
- **Phase 2a (Decision):** The coordinator collects all the votes. If all are YES, the coordinator sends COMMIT to all the participants; otherwise, the coordinator sends ABORT
- **Phase 2b (Decision):** Each participant waits for COMMIT or ABORT from the coordinator and reacts accordingly

Two-Phase Commit



(a) The finite state machine for the coordinator

(b) The finite state machine for a participant

Two-Phase Commit

- Steps taken by the 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;
}
```

Two-Phase Commit

- Steps taken by a 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

- Steps taken for handling incoming decision requests

```
/* executed by a 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 */

}
```

2PC – Participant Failure

- Consider a participant crash in one of its states, and the subsequent recovery
  - **Initial state: No problem, as the participant was unaware of the activity**
  - **Ready state: The participant is waiting to commit or abort. After recovery, the participant needs to know which state transition it should make => log the coordinator's decision**
  - **Abort state: Make entry into the abort state *idempotent*, e.g., remove the workspace of results**
  - **Commit state: Make entry into the commit state *idempotent*, e.g., copy the workspace to stable storage**
- **Note:** For distributed commit, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures

2PC – Coordinator Failure

- The problem is that the coordinator might fail or that the coordinator's decision might be lost or might not be available for some time
- **Possible Solution:** Use *timeouts*. If a participant in the ready state hasn't received the coordinator's decision before the timeout, it tries to find out what the other participants know
- **Note:** A participant *cannot* make a *local* decision. It is dependent on other (possibly failed) processes
- **Question:** Can a participant *not* succeed in getting the information it needs from the other participants?

Two-Phase Commit

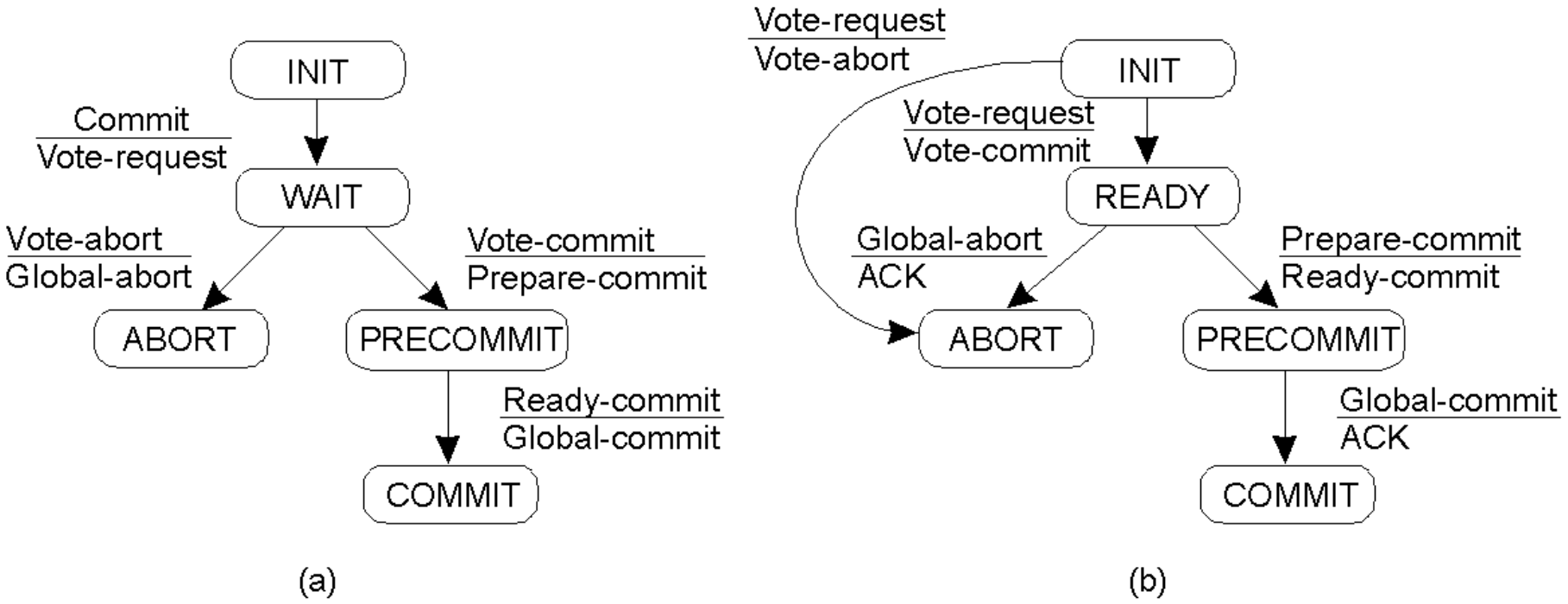
- Actions taken by a participant *P* in the *READY* state when the coordinator failed and it contacted another participant *Q* to determine what it should do

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

Three-Phase Commit

- **Phase 1a (Voting):** The coordinator sends VOTE_REQUEST to the participants
- **Phase 1b (Voting):** When a participant receives VOTE_REQUEST, it returns either YES or NO to the coordinator. If it sends NO, the participant aborts its local computation
- **Phase 2a (Precommit):** The coordinator collects all the votes. If all are YES, the coordinator sends PREPARE to all the participants; otherwise, the coordinator sends ABORT and halts
- **Phase 2b (Precommit):** Each participant waits for PREPARE, or waits for ABORT and then halts
- **Phase 3a (Decision):** The coordinator waits until all participants have ACKed receipt of PREPARE, and then sends COMMIT to all the participants
- **Phase 3b (Decision):** A participant waits for COMMIT

Three-Phase Commit



(a) Finite state machine for the coordinator

(b) Finite state machine for a participant

3PC – Participant Failure

- **Basic issue:** Can a participant find out what it should do after it crashes in the *ready* or *pre-commit* state and then recovers, particularly if the coordinator or other participants failed?
  - **Note: The coordinator and the participants on their way to commit never differ by more than one state transition**
- If a participant times out in the *ready* state, it can find out from the coordinator or other participants whether it should abort, or enter the pre-commit state
- If a participant already made it to the *pre-commit* state, it can safely commit
  - **However, it is not allowed to do so for the sake of other failing processes**

3PC – Coordinator Failure

- If the coordinator fails, the processes might need to elect a new coordinator
- The new coordinator consults the participants
- If *any* participant reports ABORT, then it is certain that no participant has committed and it is safe to *abort*
- If *any* participant reports PRECOMMIT, then it is certain that no participant has aborted and it is safe to *commit*
- If *no* participant reports either ABORT or PRECOMMIT, then it is certain that no participant has committed and it is safe to *abort*

Recovery

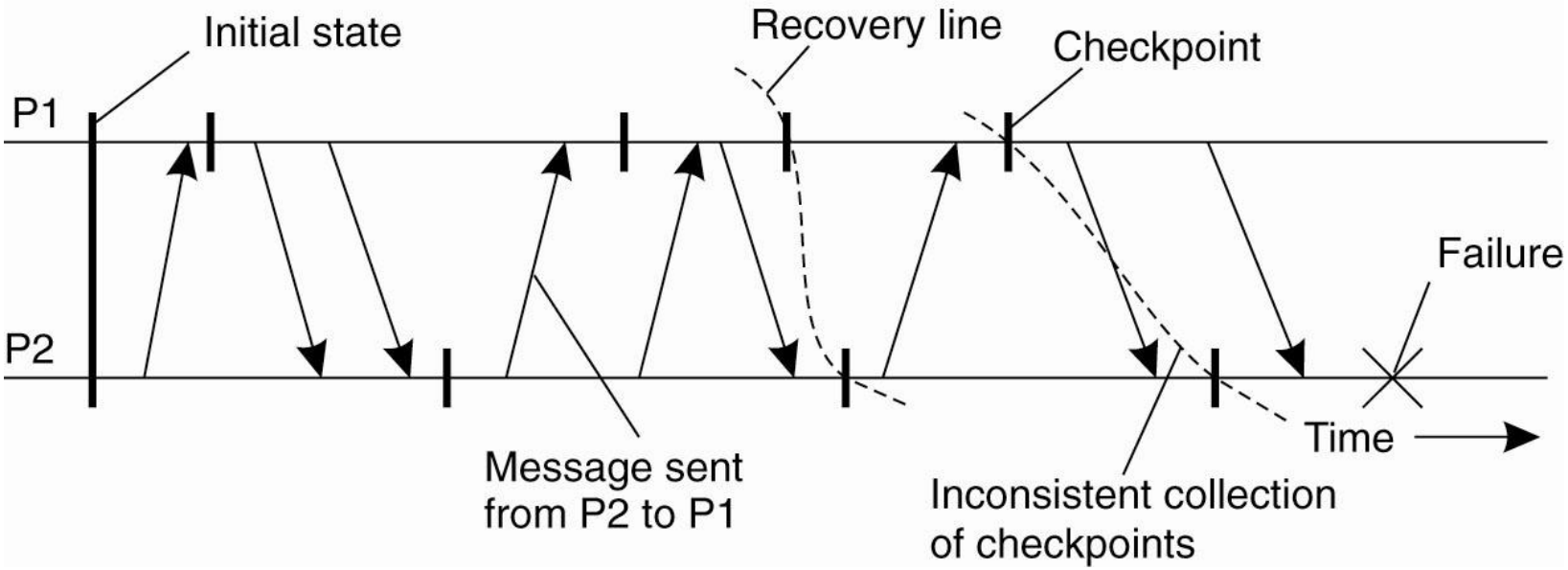
- The Problem
- Checkpointing
- Message Logging and Replay
- Recovery from Stable Storage

Recovery

- **Essence:** When a failure occurs, we need to bring the system to 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, and establish ***recovery points (checkpoints)***
- **Note:** Recovery in distributed systems is complicated, because processes need to cooperate in identifying a ***consistent state*** from which to recover

Checkpointing

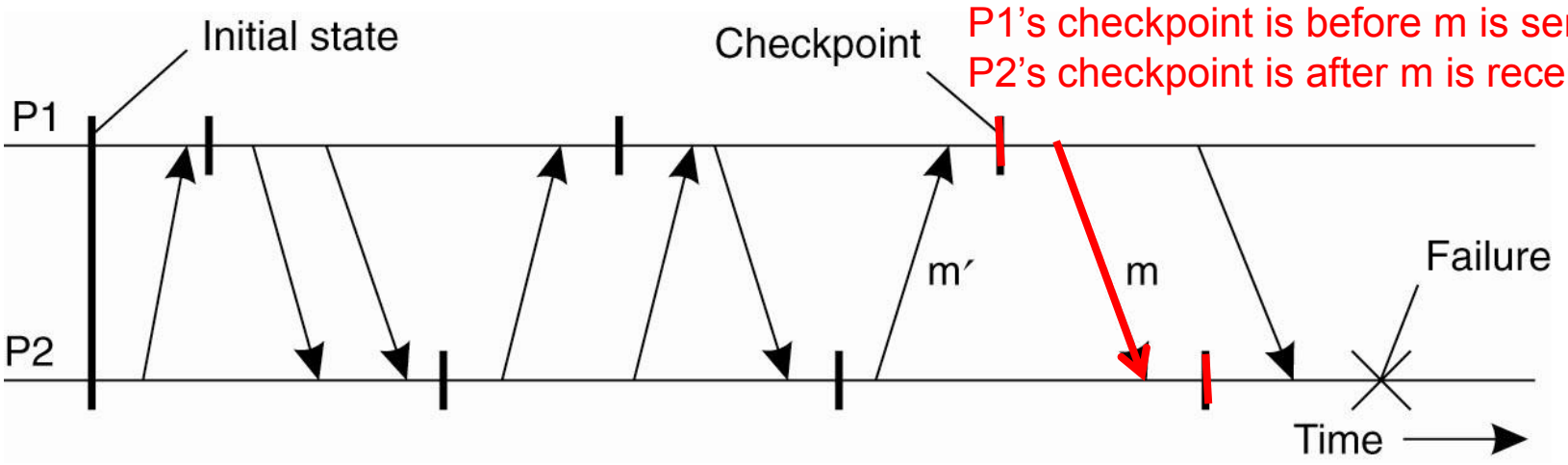
- **Checkpoint** – The state of a process, at a particular point in a computation, that is stored on disk



- Assuming that processes regularly checkpoint their states, the **recovery line** (like a consistent snapshot) corresponds to the most recent **consistent global checkpoint**

Cascaded Rollback

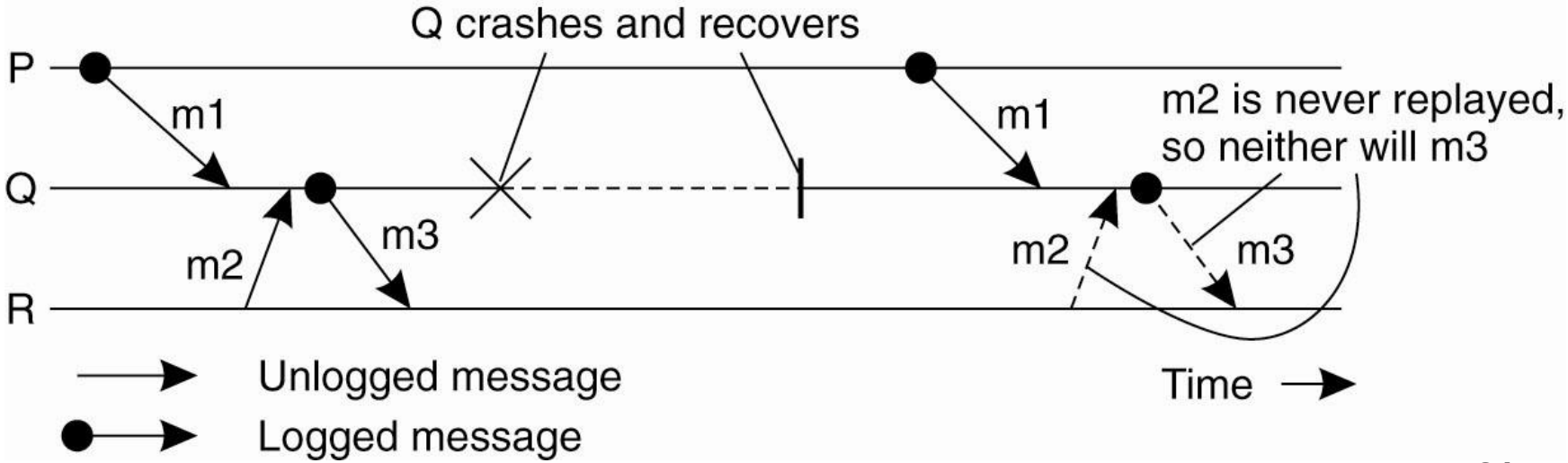
- Independent checkpointing. The domino effect



- If checkpointing is done at the "wrong" instants, the recovery line might be at system startup time => **cascaded rollback**

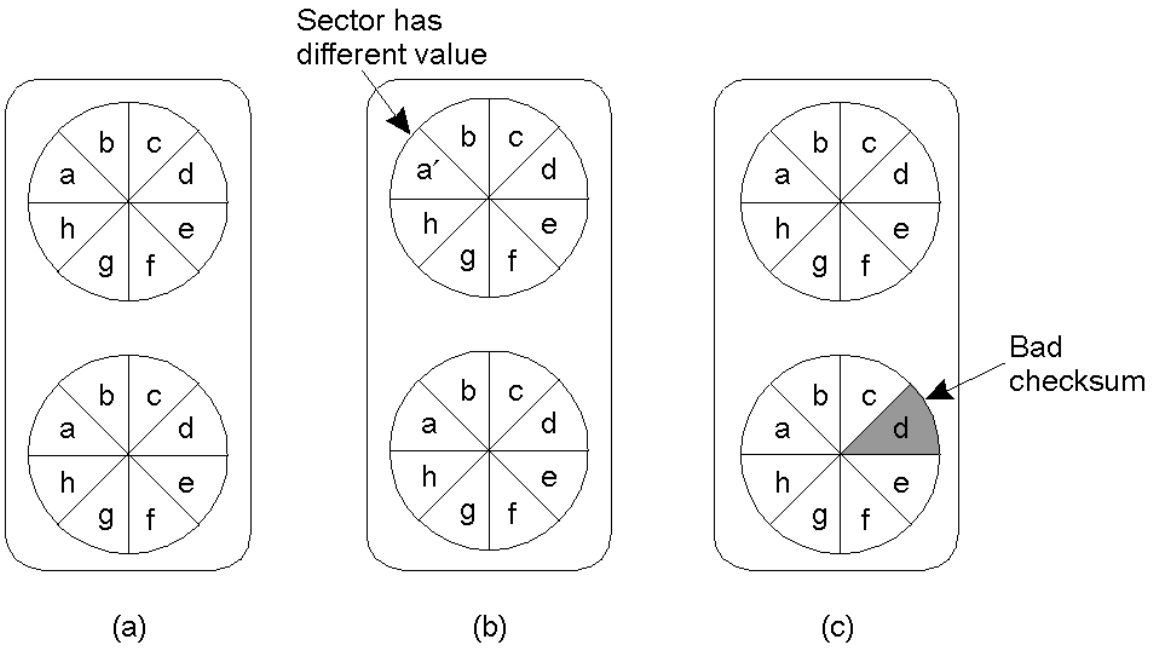
Message Logging and Replay

- Messages sent after a checkpoint must be *logged*
  - **m1 and m3 are logged but m2 is not**
- Logged messages must be *replayed* during recovery
- Incorrect replay of messages during recovery
  - **m2 and m3 are not replayed leading to an incorrect result**



Recovery from Stable Storage

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



After a crash:

(a) If both disks are identical, we're in good shape

(b) If both have *good checksums* but *different data values*, choose the *main disk*

(c) If *one checksum is ok* but the *other checksum is bad*, choose the *disk with the good checksum*

If both have bad checksums, we're **not** in good shape