Fault Tolerance (Distributed Commit and Checkpoints)

ECE 454: Distributed Computing

Instructor: Dr. Wojciech Golab
wgolab@uwaterloo.ca

Slides are derived from A. S. Tanenbaum and M. Van Steen,
Distributed Systems: Principles and Paradigms, 2nd Edition, Pearson-Prentice Hall, 2006.

Learning objectives

To develop a working knowledge of:

- two-phase commit protocol for distributed atomic transactions
- distributed checkpoints

Distributed commit

The (textbook) correctness properties of transactions are:

- **A**tomicity – all the updates take effect or none of them.
- **C**onsistency – constraints (e.g., referential integrity) are preserved.
- **I**solation – concurrent transactions are unaware of each other.
- **D**urability – updates made by committed transactions are not lost in the event of a failure.

The **distributed commit** problem concerns transaction atomicity (A) in a distributed environment.

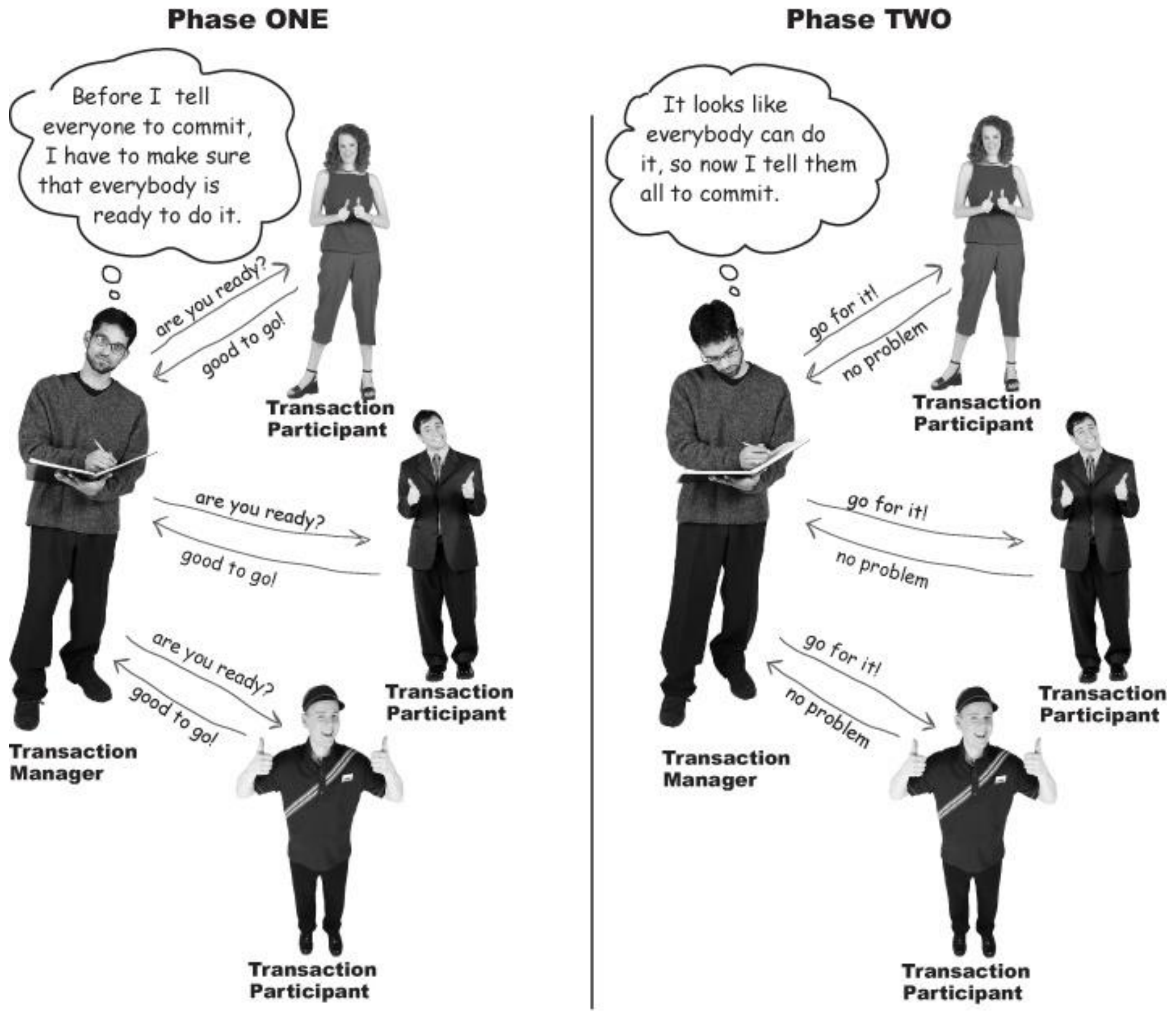
Example: decrease account balance at site 1, increase account balance at site 2, one of the sites fails during the transaction.

Informal definition of distributed transaction commitment

- Transaction commitment is initiated by a coordinator after the transaction execution phase, during which each participant discovers whether it is able to commit the transaction or not.
- When asked by the coordinator, each participant casts a vote (yay or nay) to commit or abort the transaction globally.
- The coordinator collects the votes, computes the global decision, and then shares the decision with the participants.
- The transaction must abort globally if any participant votes to abort.
- In the absence of failures, the transaction must commit globally if every participant votes to commit.

Two-phase commit (2PC)

Source: Kathy Sierra and Bert Bates
https://www.safaribooksonline.com/library/view/head-first-ejb/0596005717/ch09s02.html



Two-phase commit (2PC)

Two-phase commit (2PC) is a coordinator-based distributed transaction commitment protocol. High-level idea:

Phase 1: Coordinator asks participants whether they are ready to commit. Participants respond with votes.

Phase 2: Coordinator examines votes and decides the outcome of the transaction. If all participants vote to commit then the transaction is committed successfully, otherwise it is aborted.

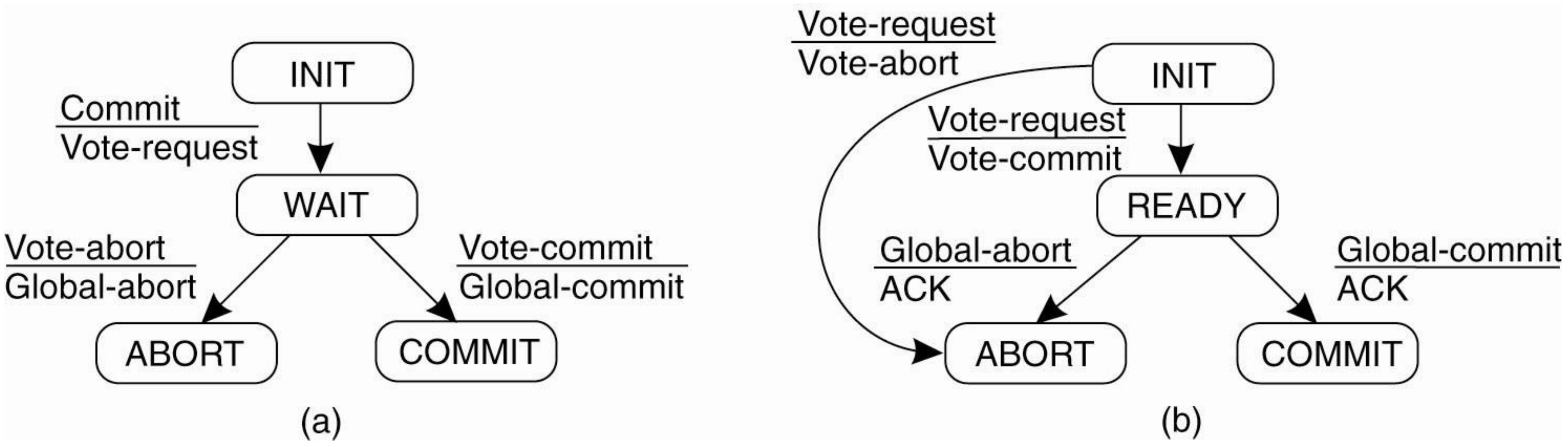
Assumptions: Synchronous processes and bounded communication delays, crash-recovery failures, processes have access to stable storage for logging recovery information.

What happens if the coordinator crashes?

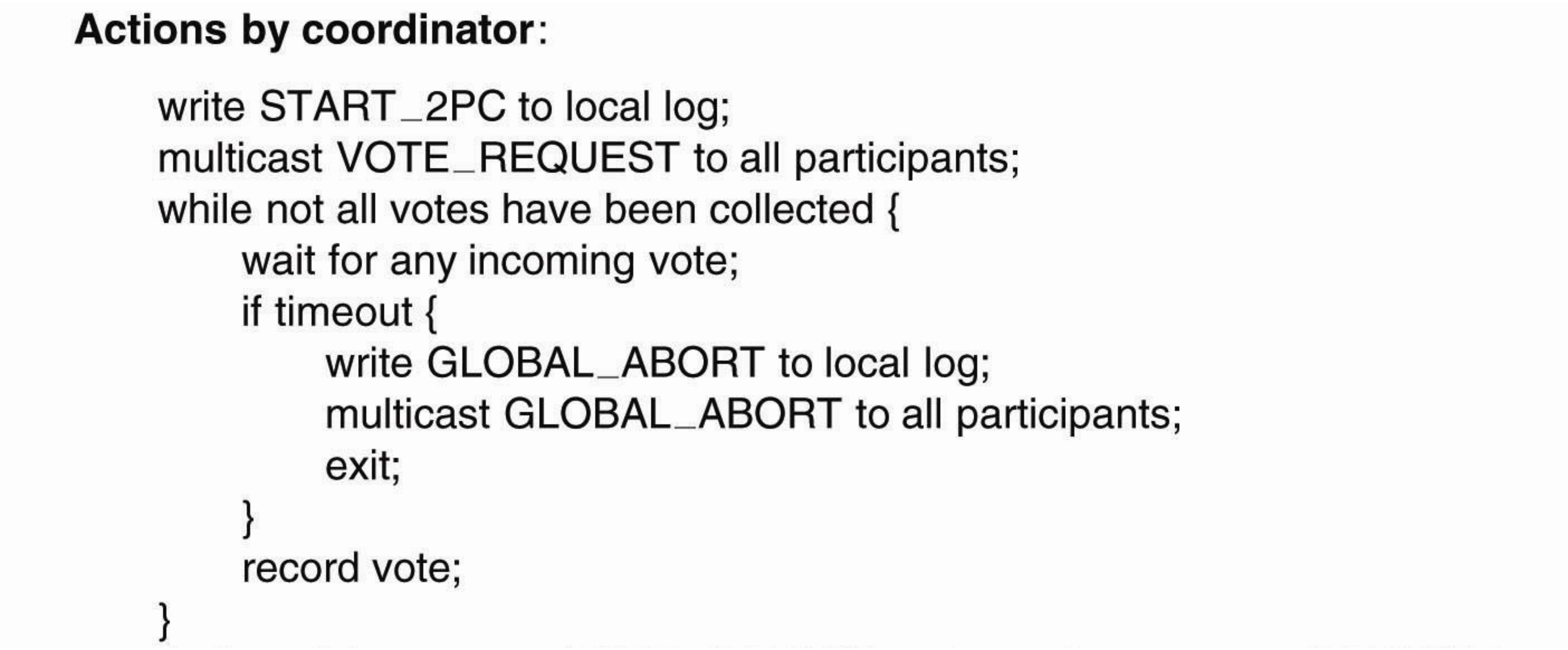
What happens if a participant and the coordinator both crash?

2PC: states and state transitions

State transition diagrams for the coordinator (a) and for a participant (b).



2PC: coordinator actions

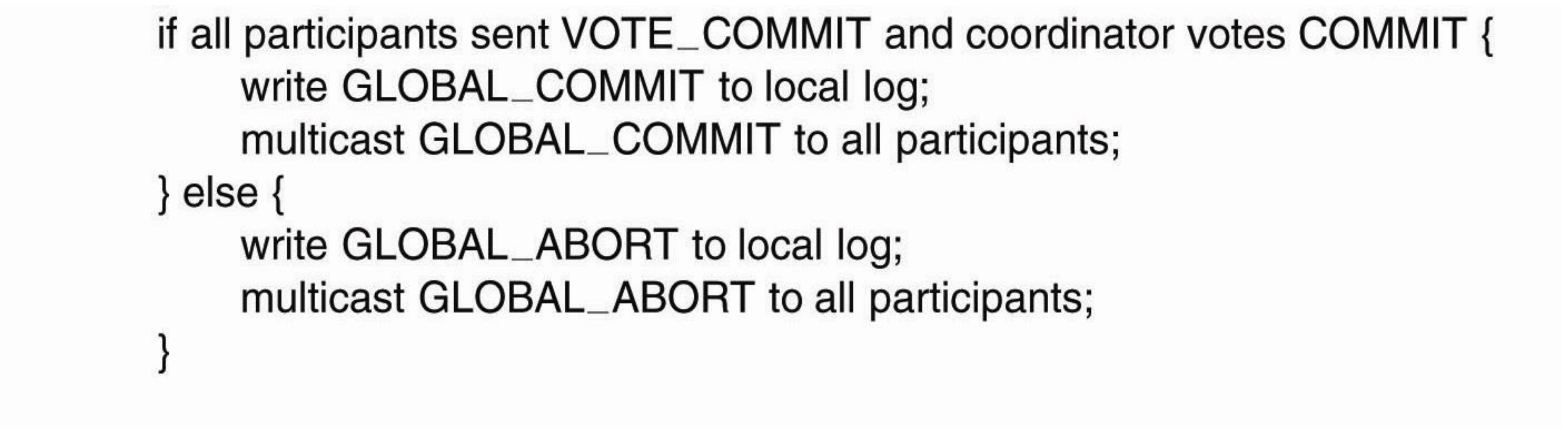


Actions by coordinator:

```
write 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;
}
  . . .
```

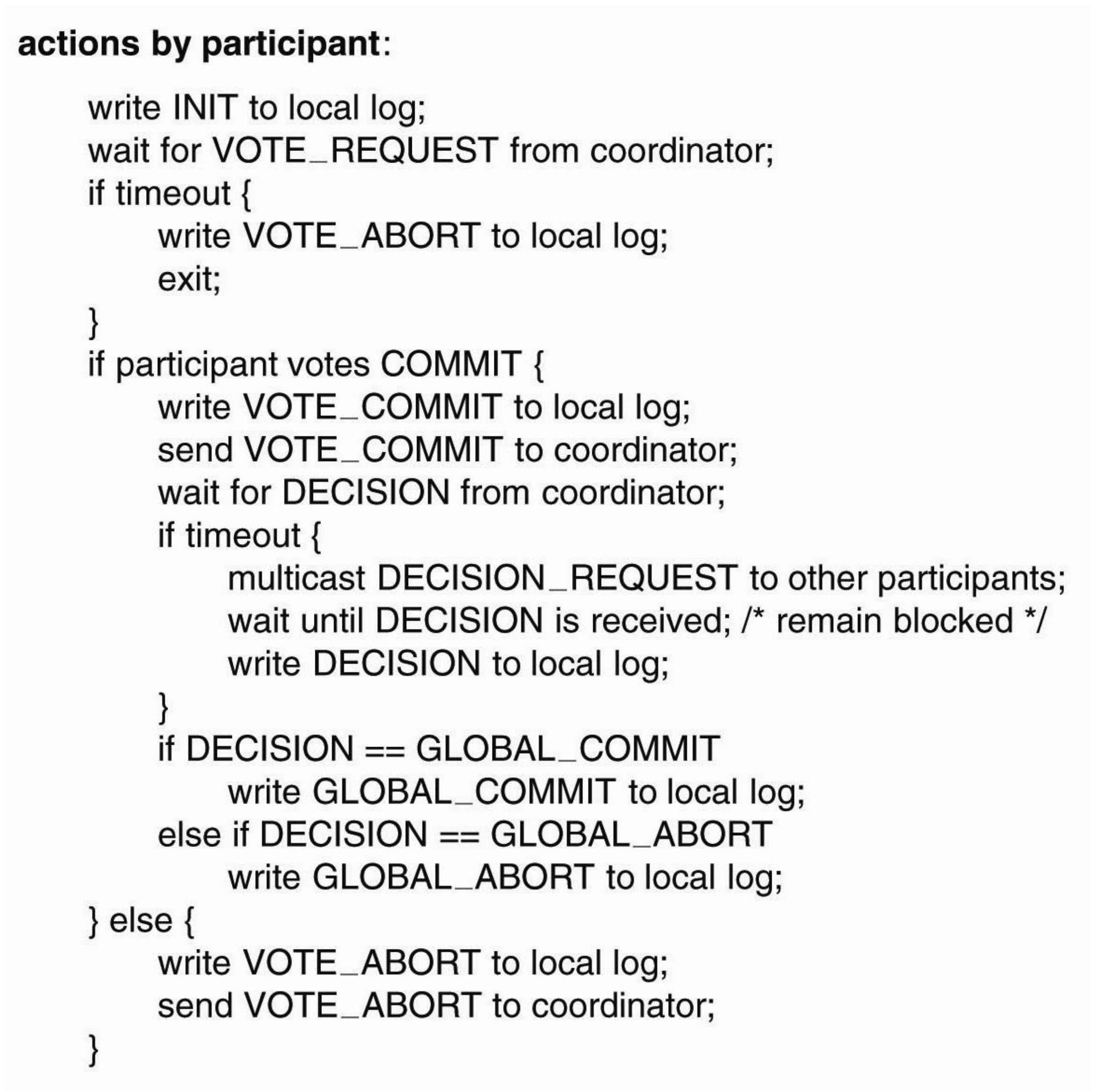
2PC: coordinator actions

. . .



```
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;
}
```

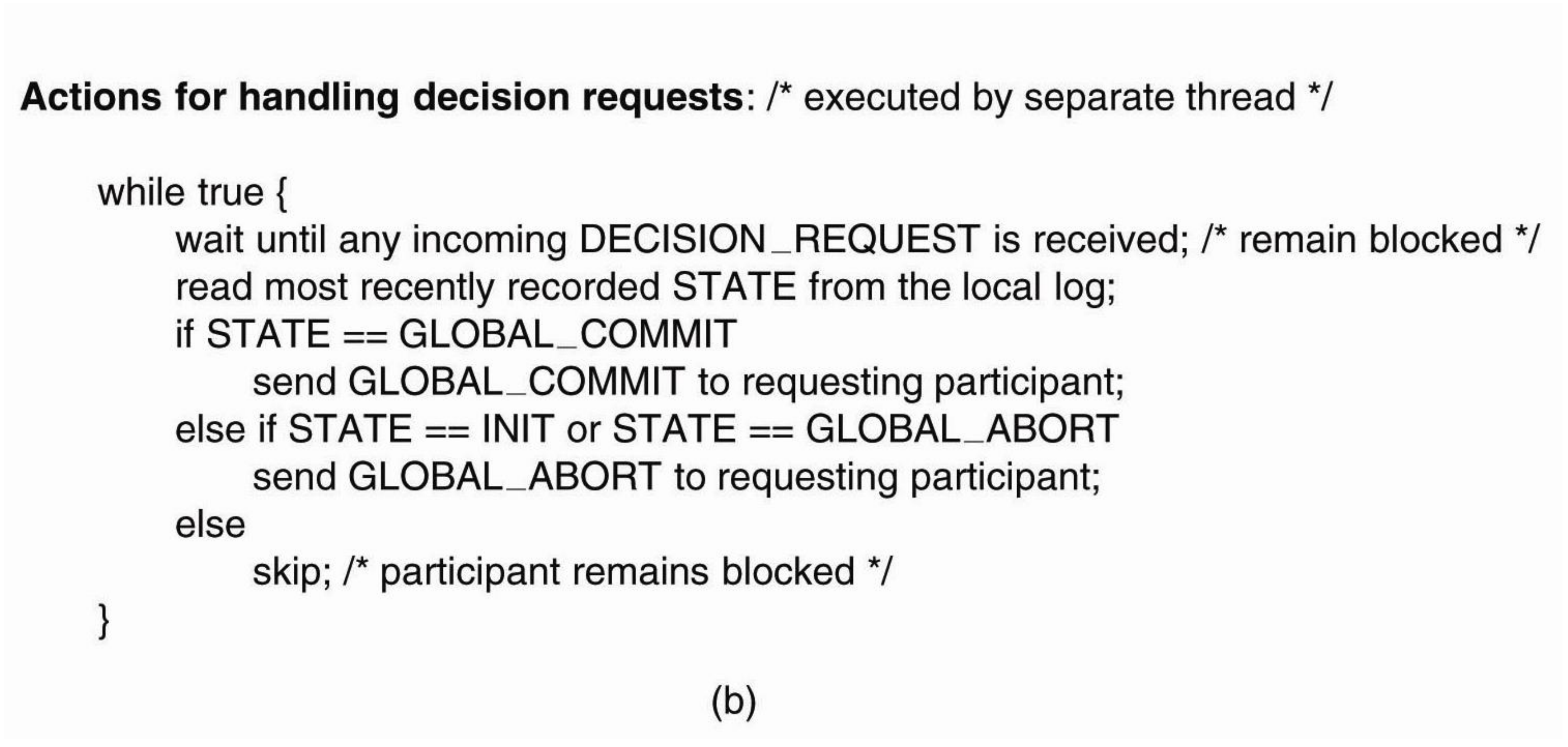
2PC: participant actions



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;
}
```

2PC: participant actions



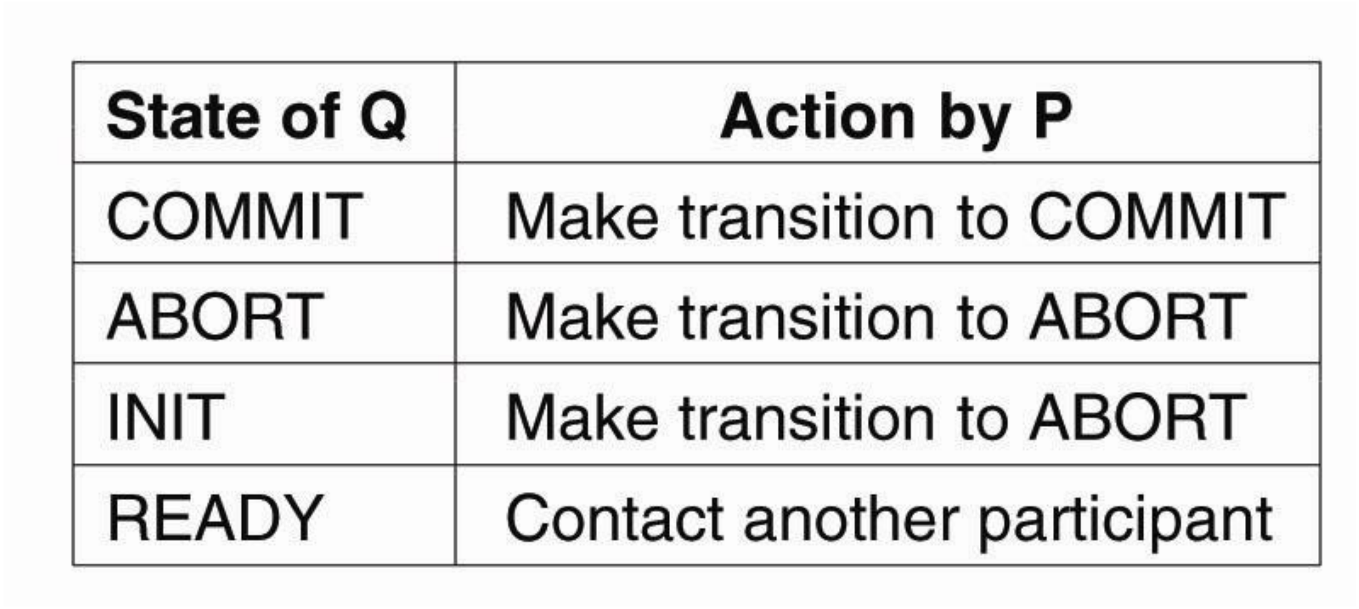
Actions for handling decision requests: /* executed by separate thread */

```
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 */
}
```

(b)

2PC: participant actions

If a participant P does not receive a commit or abort decision from the coordinator within a bounded period of time, it may try to learn the decision from another participant Q.



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

2PC: recovery from failure

What if the coordinator crashes?

- A participant is able to make progress as long as it received the decision from the coordinator despite the crash, or if it was able to learn the decision from another participant.
- In general the transaction is safe to commit if all participants voted to commit (all in READY or COMMIT state), and safe to abort otherwise. However, the protocol (as described here and in the book) blocks if all participants are READY until the coordinator recovers. (Recovery protocol is not shown.)

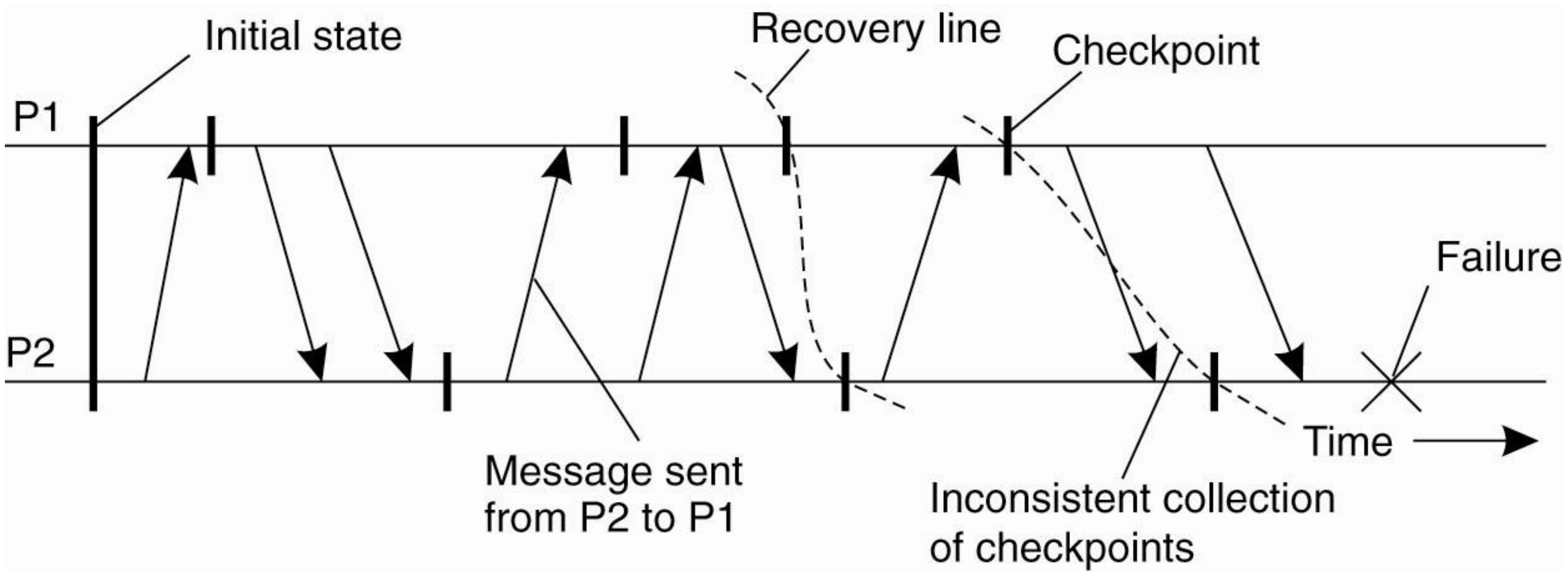
What if a participant and the coordinator both crash?

- A smarter implementation of 2PC can deal with the failure of the coordinator, but the simultaneous failure of the coordinator and one participant makes it difficult to determine whether the participants are all READY.

Distributed Checkpoints

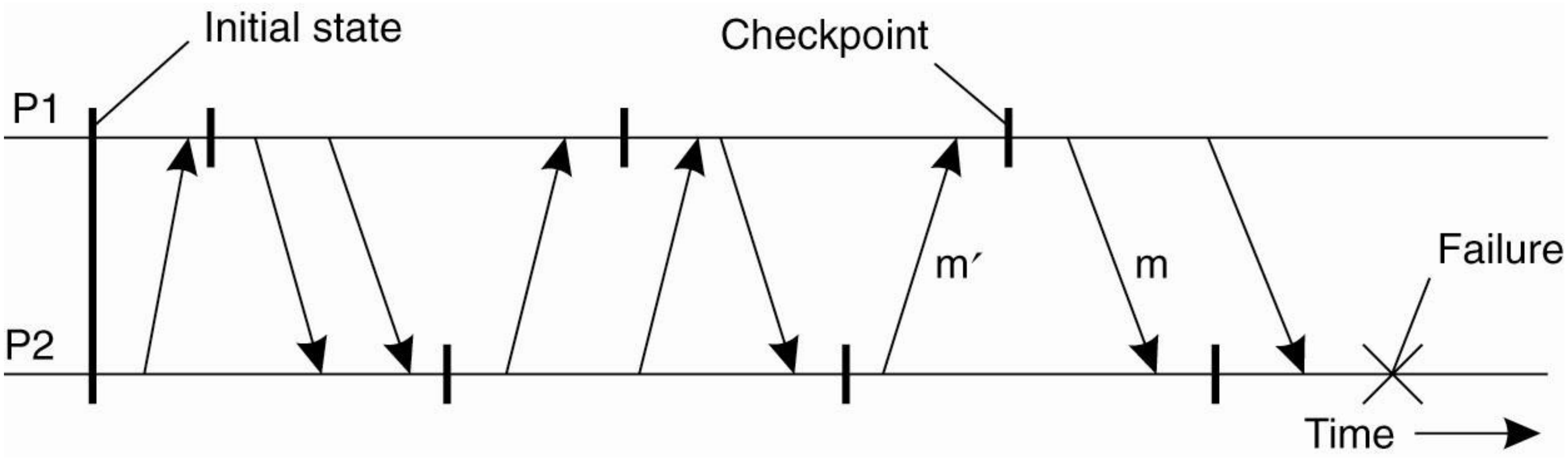
Distributed checkpoints

Recovery is possible only if the collection of checkpoints taken by individual processes forms a **distributed snapshot**: if the receive event for a message is in the snapshot then so is the corresponding send event. The most recent distributed snapshot is called a **recovery line**.


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

Distributed checkpoints

If the most recent checkpoints taken by processes do not provide a recovery line then successively earlier checkpoints must be considered. This is known as the **domino effect**, and is vaguely similar to cascading rollback in ECE 356.



Distributed checkpoints

The following **coordinated checkpointing algorithm** ensures that a recovery line is created:

Phase 1: The coordinator sends a CHECKPOINT_REQUEST message to all processes. Upon receiving this message a processes does the following:

- pause sending new messages to other processes (e.g., queue outgoing messages temporarily)
- takes a local checkpoint
- returns an acknowledgment to the coordinator.

Phase 2: Upon receiving acknowledgments from all processes, the coordinator sends a CHECKPOINT_DONE message to all processes, which then resume processing messages.