Fault Tolerance

Part I Introduction

Part II Process Resilience

Part III Reliable Communication

Part IV Distributed Commit

Part V Recovery

Giving credit where credit is due:

- Most of the lecture notes are based on slides by Prof. Jalal Y. Kawash at Univ. of Calgary and Dr. Daniel M. Zimmerman at CALTECH
- Some of the lecture notes are based on slides by Scott Shenker and Ion Stoica at Univ.of California, Berkeley, Timo Alanko at Univ. of Helsinki, Finland, Hugh C. Lauer at Worcester Polytechnic Institute, Xiuwen Liu at Florida State University
- I have modified them and added new slides

Fault Tolerance

Fault Tolerance

- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults
- Fault tolerance is related to **dependability**

Dependability

Dependability Includes

- Availability
- Reliability
- Safety
- Maintainability

Availability & Reliability (1)

- **Availability**: A measurement of whether a system is *ready to be used immediately*
  - System is up and running at any given moment

- **Reliability**: A measurement of whether a system can *run continuously without failure*
  - System continues to function for a long period of time

Availability & Reliability (2)

- A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable

- A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

Safety & Maintainability

- **Safety**: A measurement of *how safe failures are*
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability**: A measurement of *how easy it is to repair a system*
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically? Self-healing systems?

Faults

- A system **fails** when it cannot meet its promises (specifications)
- An **error** is part of a system state that may lead to a failure
- A **fault** is the cause of the error
- **Fault-Tolerance**: the system can provide services even in the presence of faults
- Faults can be:
  - Transient (appear once and disappear)
  - Intermittent (appear-disappear-reappear behavior)
    - A loose contact on a connector ➔intermittent fault
  - Permanent (appear and persist until repaired)

Failure Models

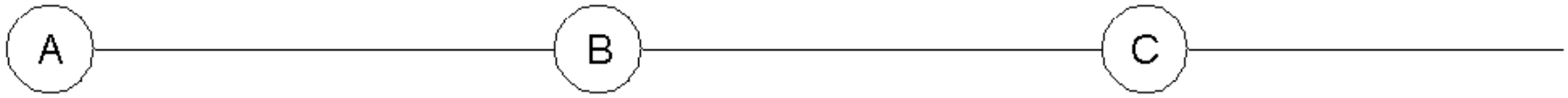
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure (Byzantine failure)	A server may produce arbitrary responses at arbitrary times

Failure Masking ?

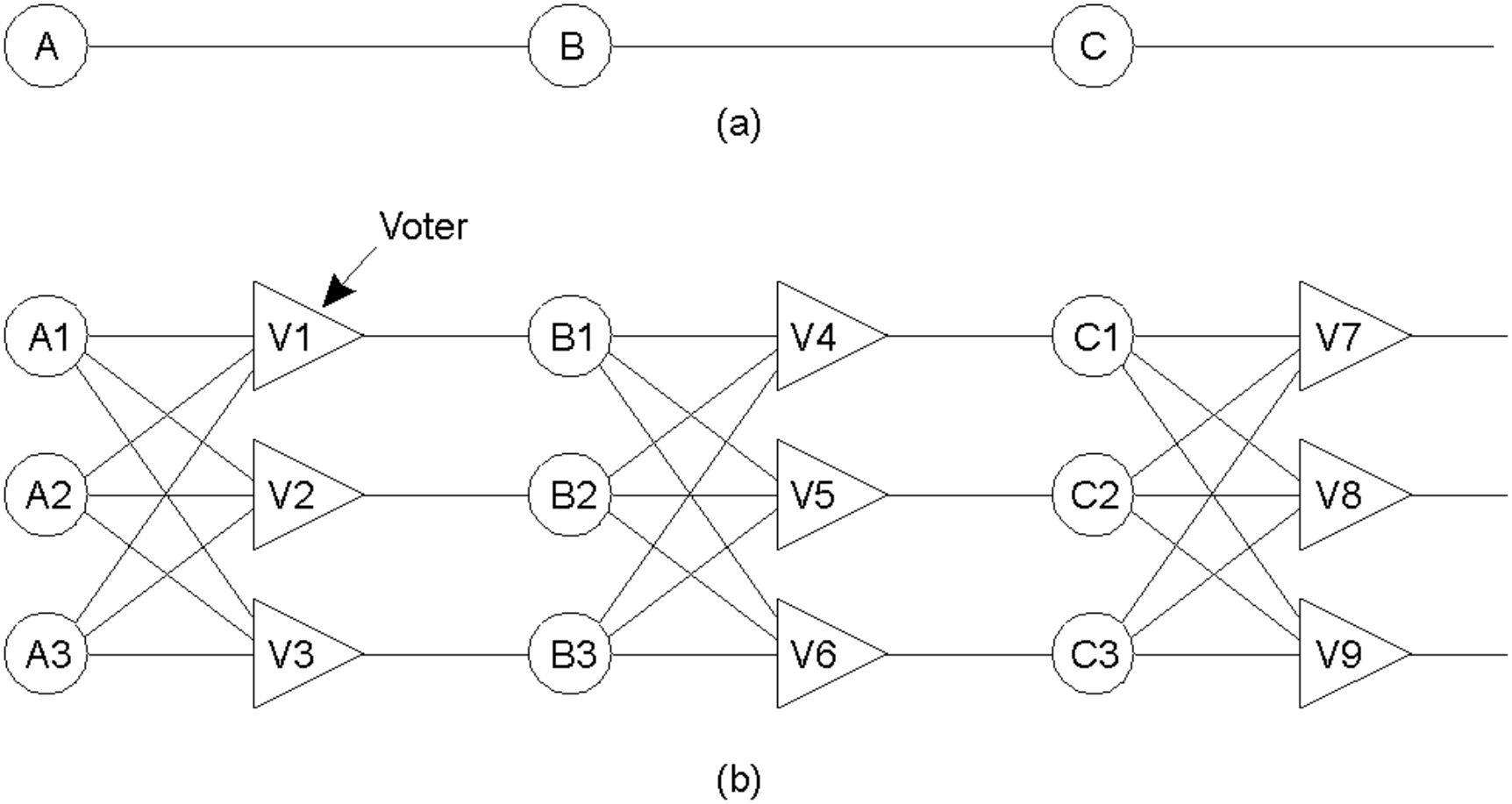
- Redundancy is key technique for hiding failures
- Redundancy types:

1. Information: add extra (control) information
   - Error-correction codes in messages
2. Time: perform an action persistently until it succeeds:
   - Transactions
3. Physical: add extra components (S/W & H/W)
   - Process replication, electronic circuits

Example – Redundancy in Circuits (1)



Example – Redundancy in Circuits (2)



Triple modular redundancy.

Fault Tolerance

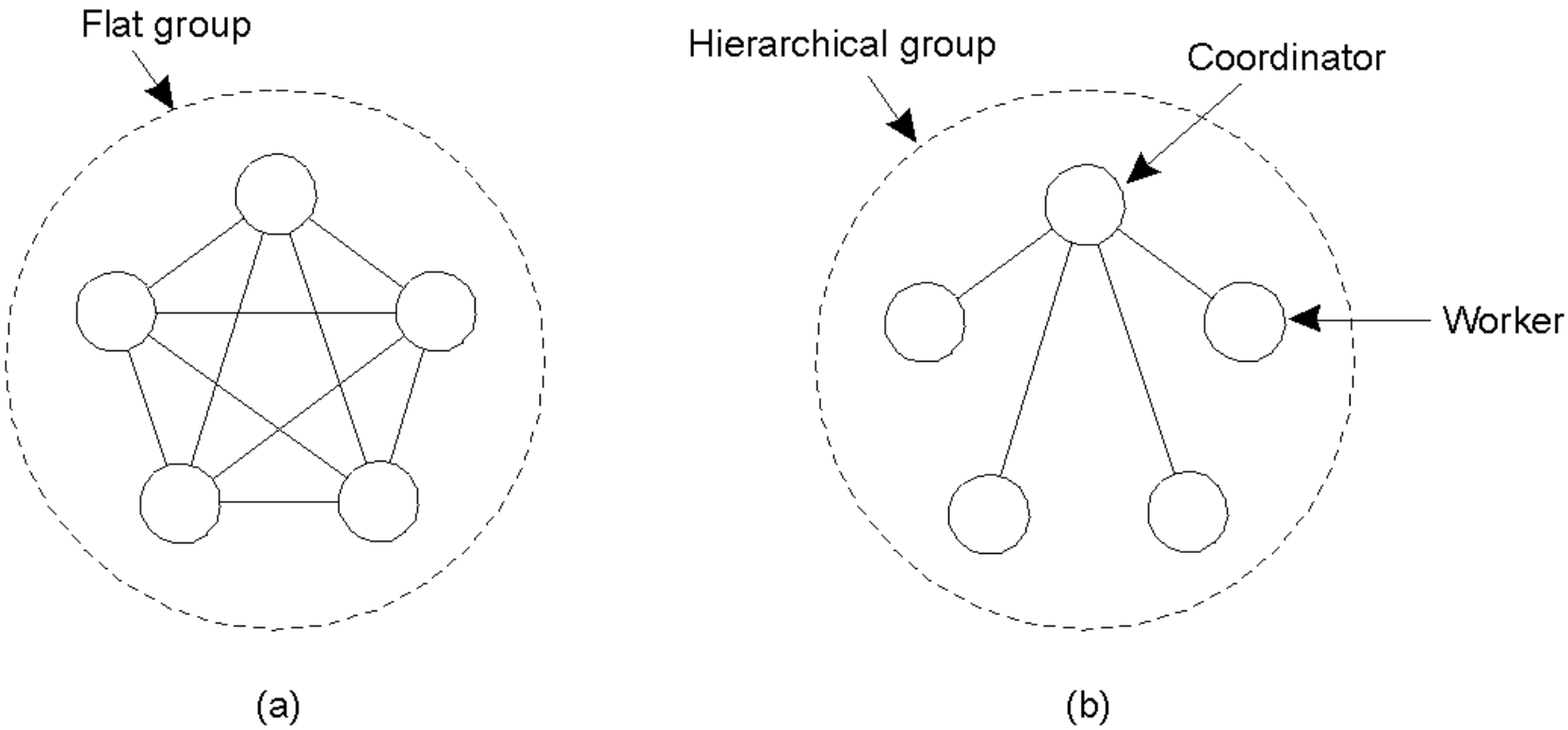
Part II

Process Resilience

Process Resilience

- Mask process failures by replication
- Organize processes into groups, a message sent to a group is delivered to all members
- If a member fails, another should fill in

Flat Groups versus Hierarchical Groups



a) Communication in a flat group.

b) Communication in a simple hierarchical group

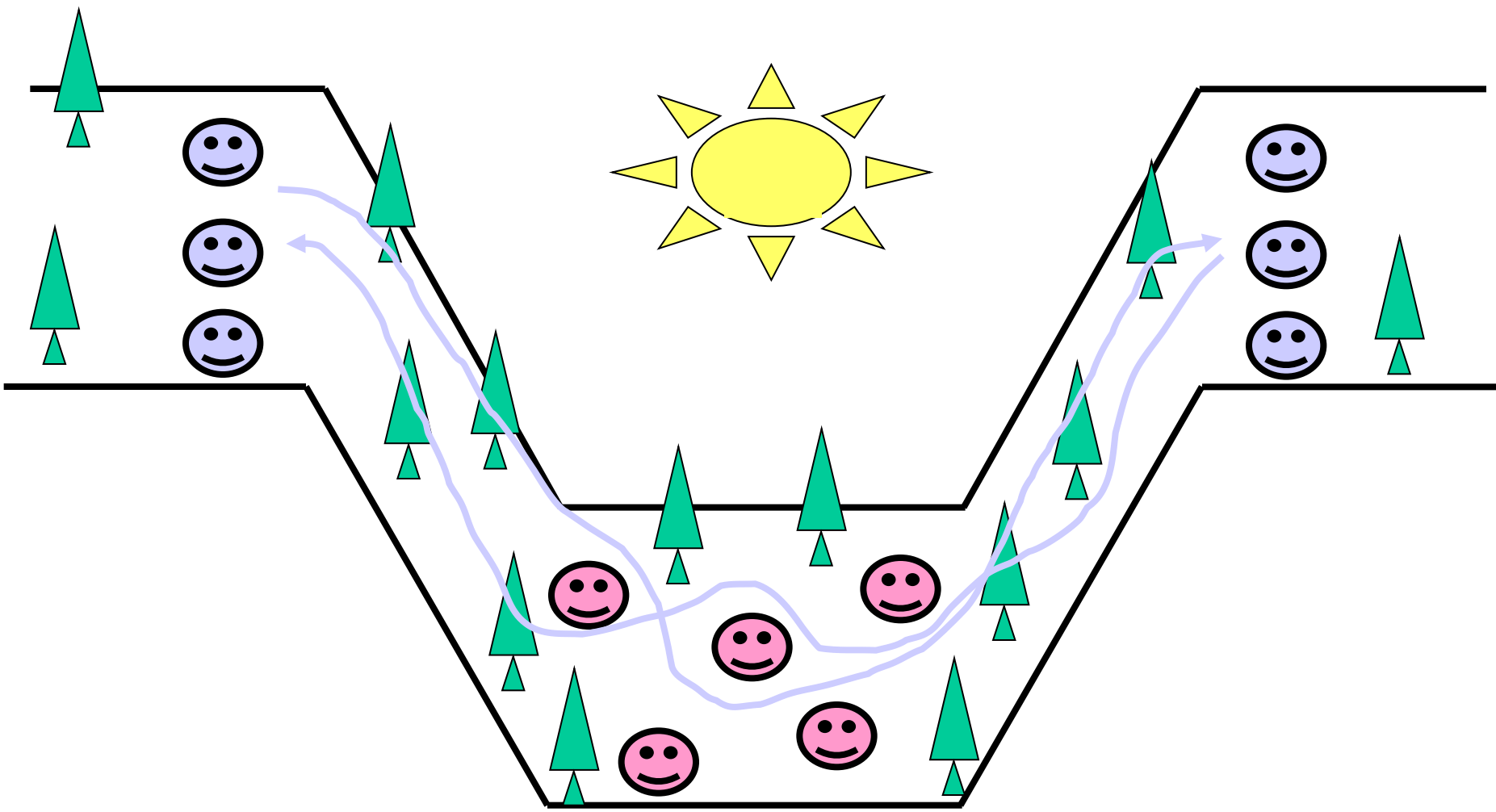
Process Replication

- Replicate a process and group replicas in one group
- How many replicas do we create?
- A system is k fault-tolerant if it can survive and function even if it has k faulty processes
  - For crash failures (a faulty process halts, but is working correctly until it halts)
    - k+1 replicas
  - For Byzantine failures (a faulty process may produce arbitrary responses at arbitrary times)
    - 2k+1 replicas

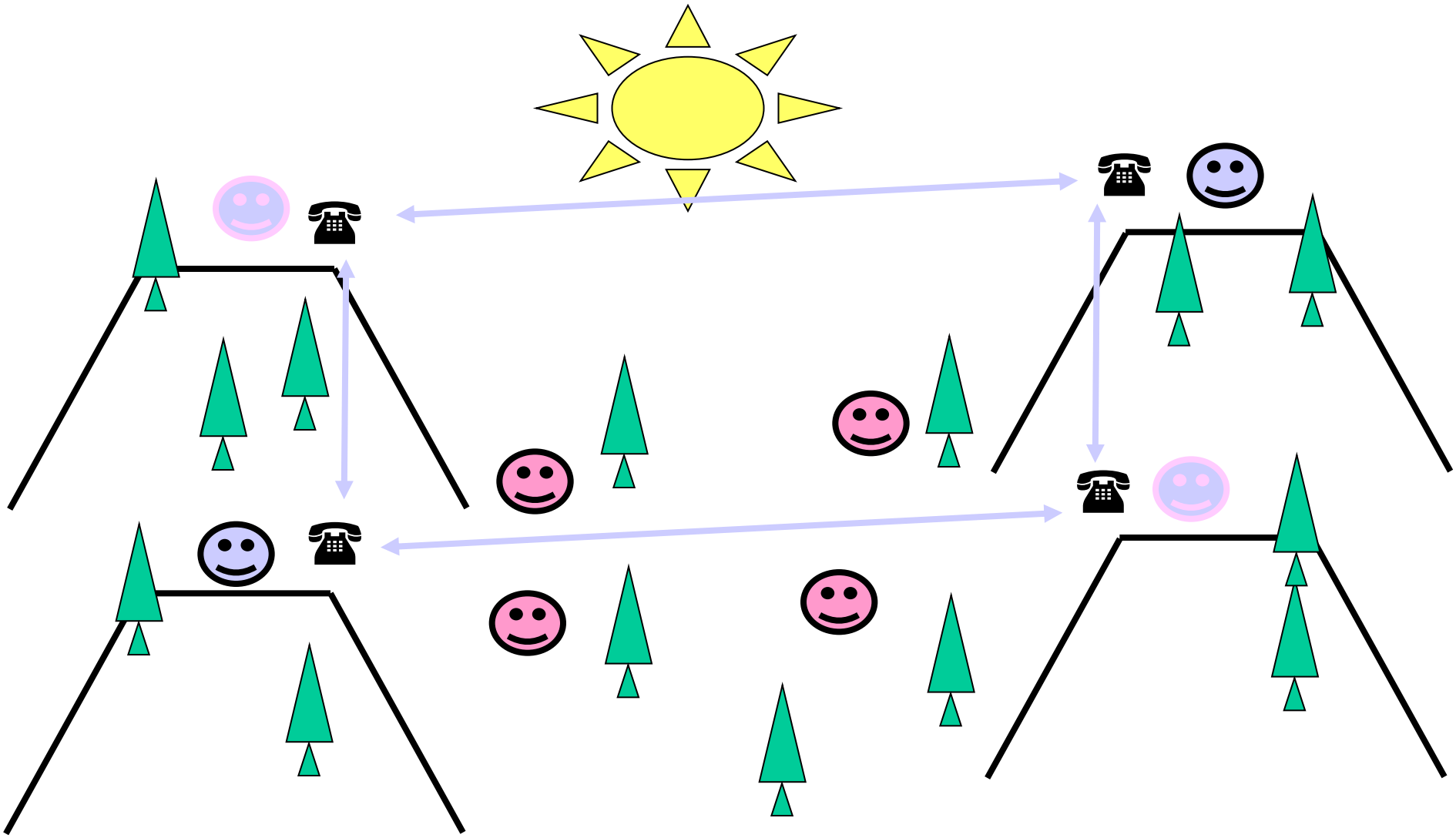
Agreement

- Need agreement in DS:
  - Leader, commit, synchronize
- **Distributed Agreement algorithm**: all non-faulty processes achieve consensus in a finite number of steps
- Perfect processes, faulty channels: two-army
- Faulty processes, perfect channels: Byzantine generals

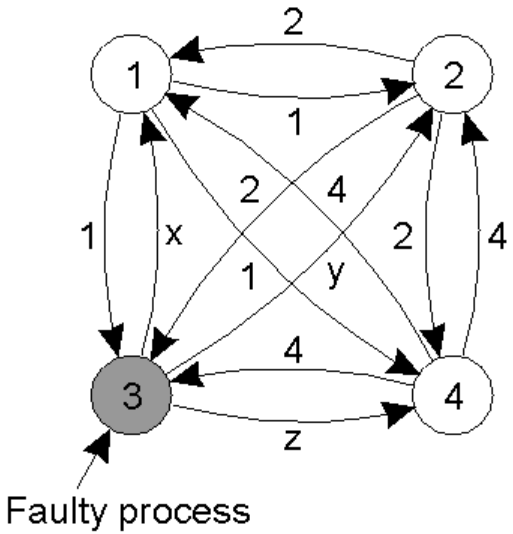
Two-Army Problem



Byzantine Generals Problem



Byzantine Generals -Example (1)



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

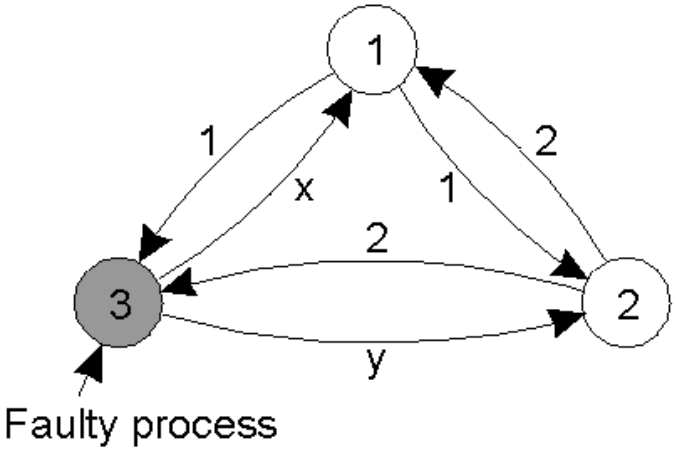
The Byzantine generals problem for 3 loyal generals and1 traitor.

a) The generals announce the time to launch the attack (by messages marked by their ids).

b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step 3, where every general passes his vector from (b) to every other general.

Byzantine Generals –Example (2)



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

The same as in previous slide, except now with 2 loyal generals and one traitor.

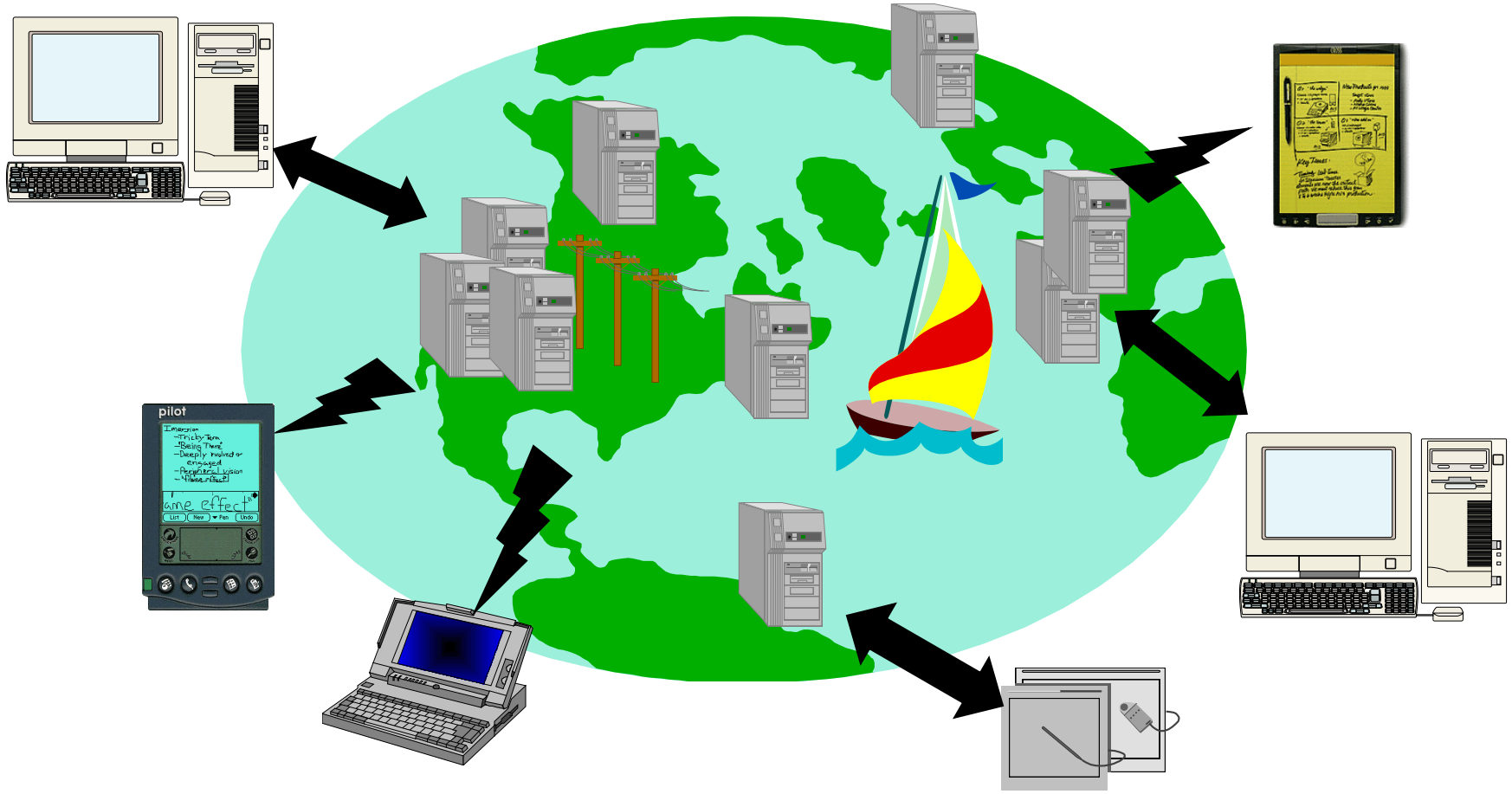
Byzantine Generals

- Given three processes, if one fails, consensus is impossible

- Given N processes, if F processes fail, consensus is impossible if $N \leq 3F$

OceanStore

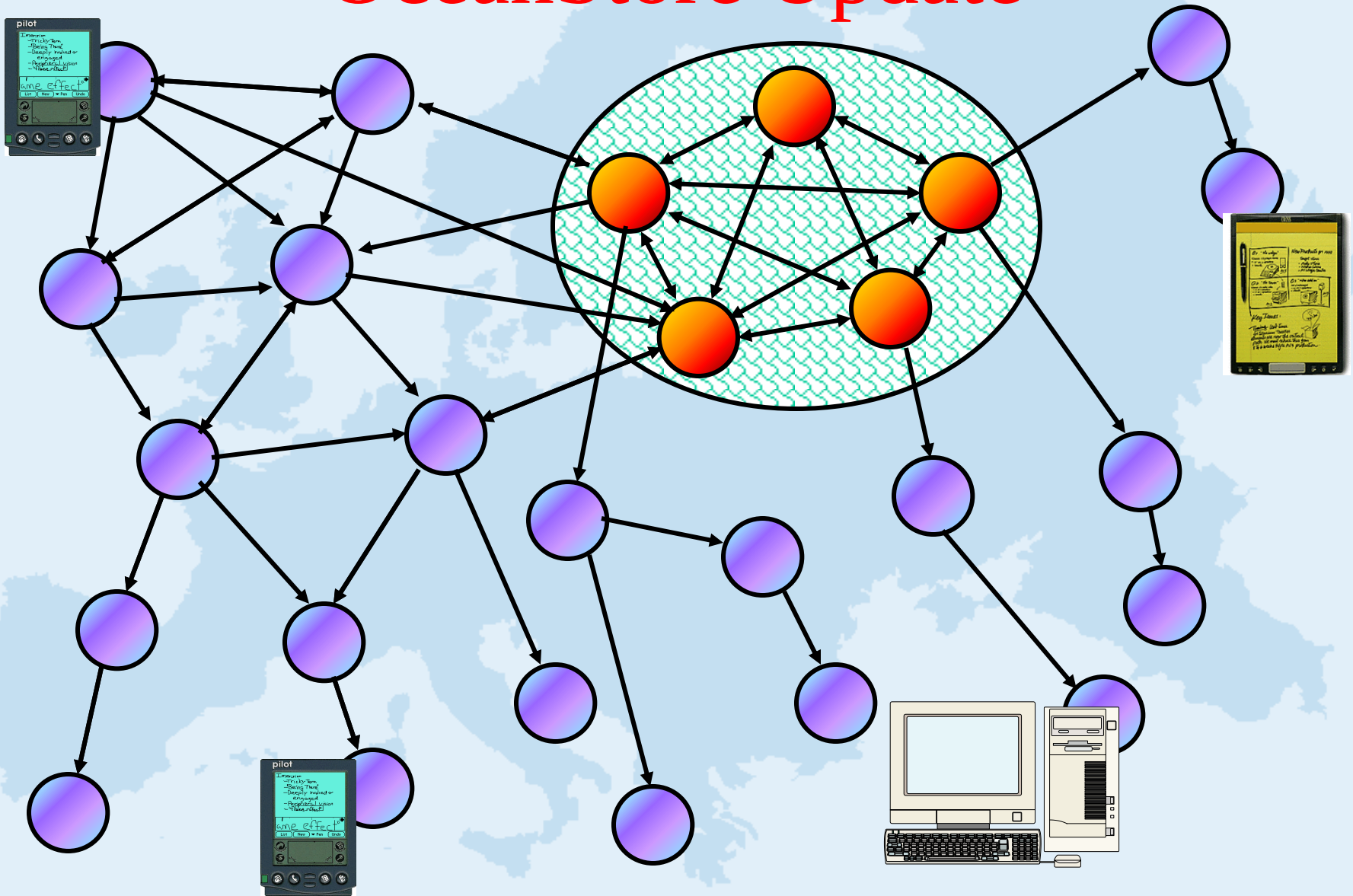
Global-Scale Persistent Storage on Untrusted Infrastructure



Update Model

- Concurrent updates w/o wide-area locking
  - Conflict resolution
    - Updates Serialization
- A master replica?
  - Incompatible with the untrusted infrastructure assumption
- Role of primary tier of replicas
  - All updates submitted to primary tier of replicas which chooses a final total order by following Byzantine agreement protocol
- A secondary tier of replicas
  - the result of the updates is multicast down the dissemination tree to all the secondary replicas

The Path of an OceanStore Update



Chapter 8

Fault Tolerance

Part III

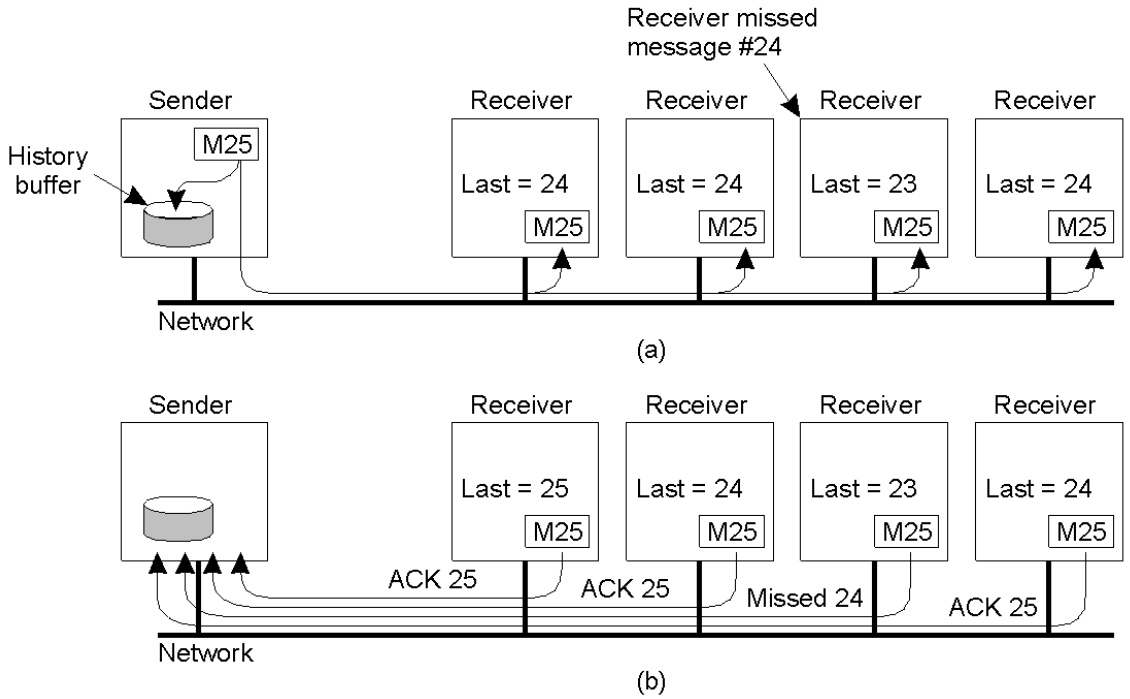
Reliable Communication

Reliable Group Communication

Reliable Group Communication

- When a group is static and processes do not fail
- Reliable communication = deliver the message to all group members
  - Any order delivery
  - Ordered delivery

Basic Reliable-Multicasting Schemes



A simple solution to reliable multicasting when all receivers are known and assumed not to fail

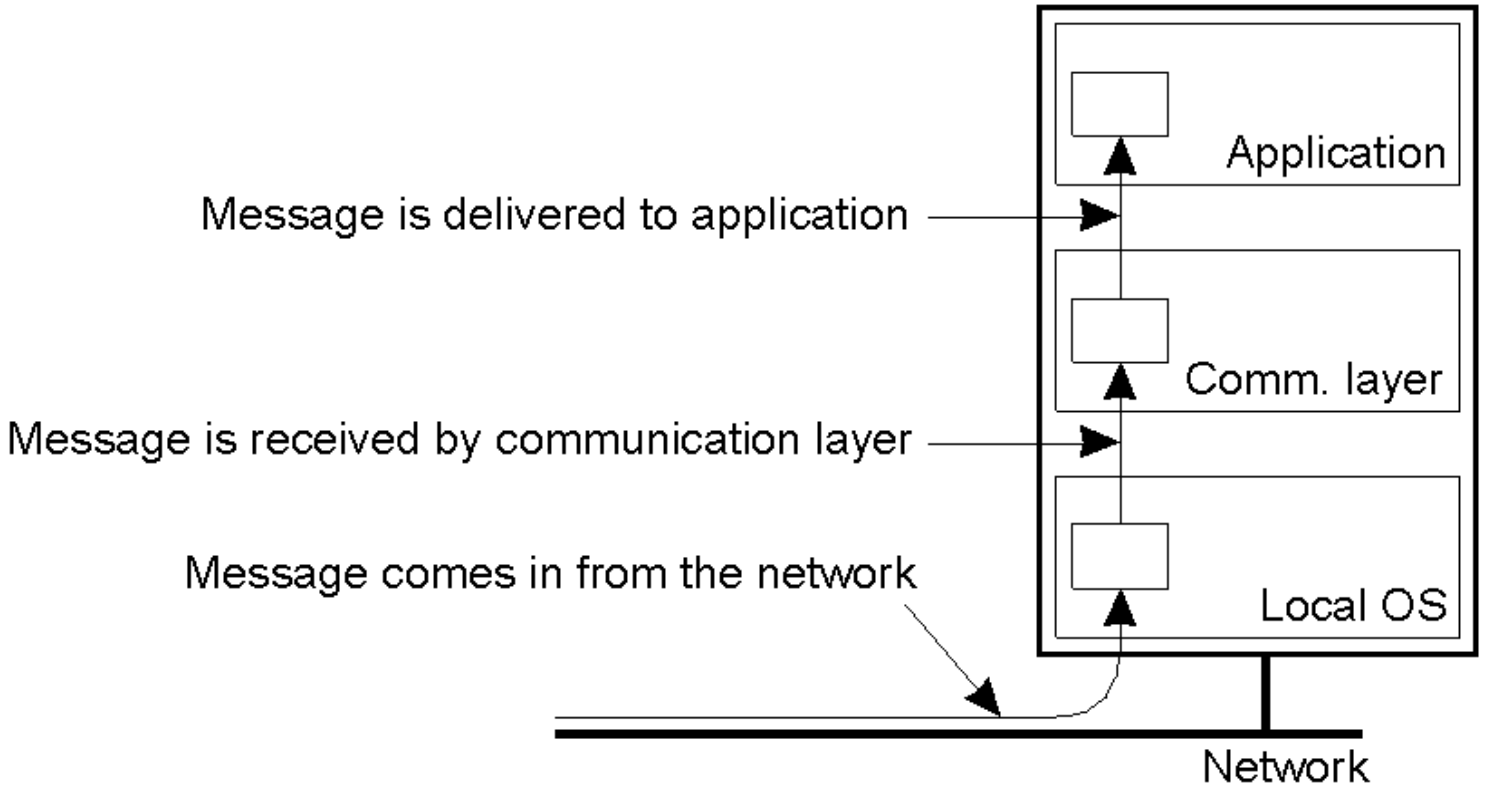
a) Message transmission

b) Reporting feedback

Atomic Multicast

- All messages are delivered in the same order to “all” processes
- **Group view**: the view on the set of processes contained in the group
- **Virtual synchronous multicast**: a message m multicast to a group view G is delivered to all non-faulty processes in G

Virtual Synchrony System Model



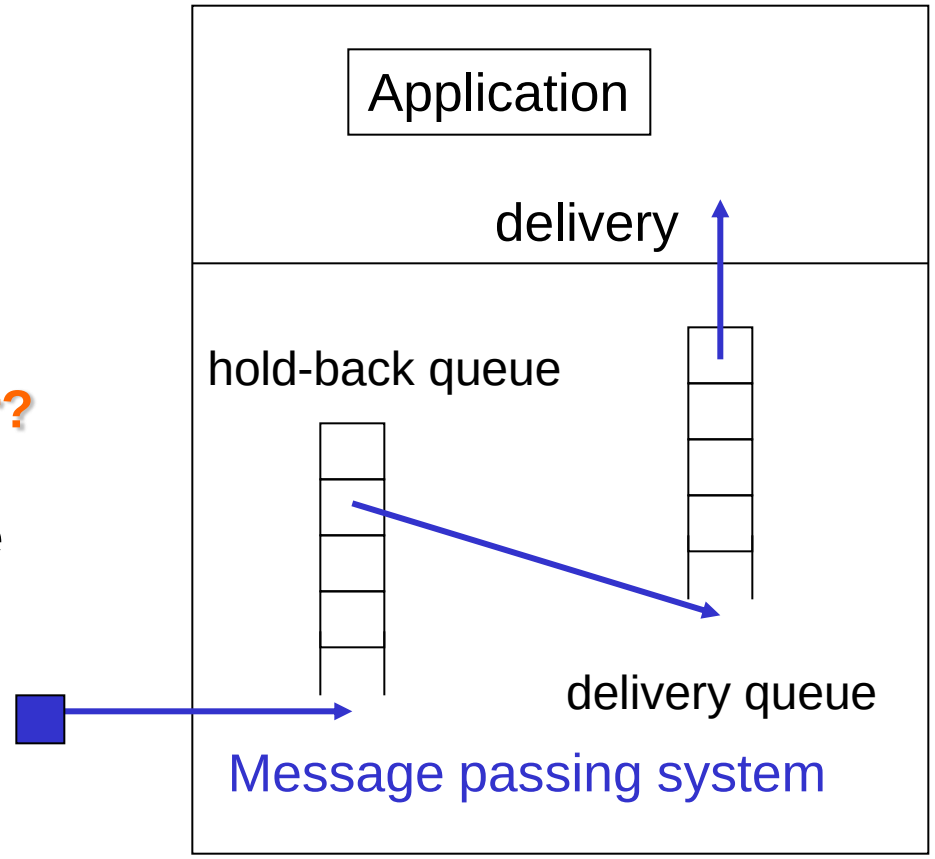
The logical organization of a distributed system to distinguish between message receipt and message delivery

Message Delivery

Delivery of messages

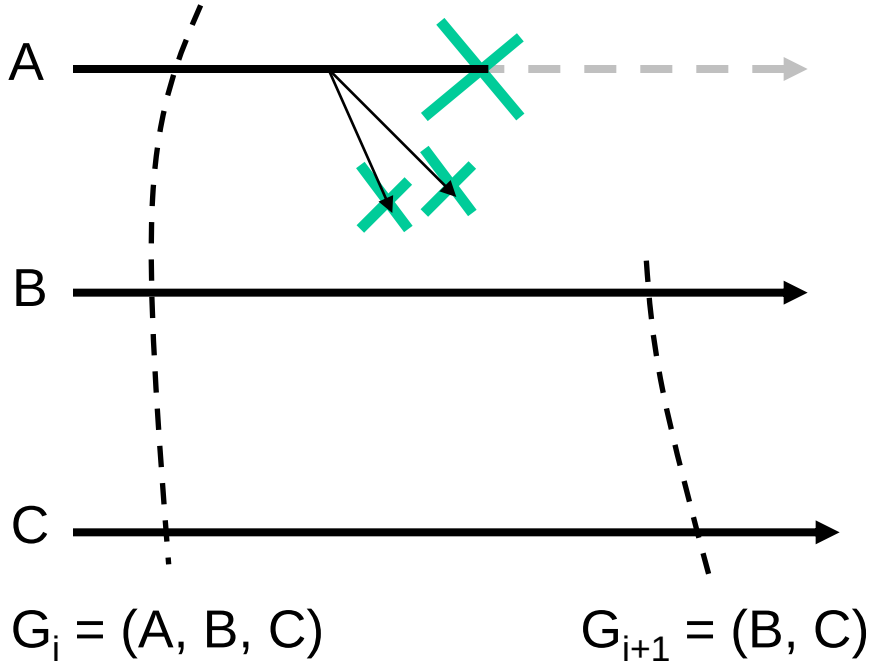
- new message => HBQ
- decision making
  - delivery order
  - **deliver or not to deliver?**

- the message is allowed to be delivered: HBQ => DQ
- when at the head of DQ: message => application (application: *receive* …)

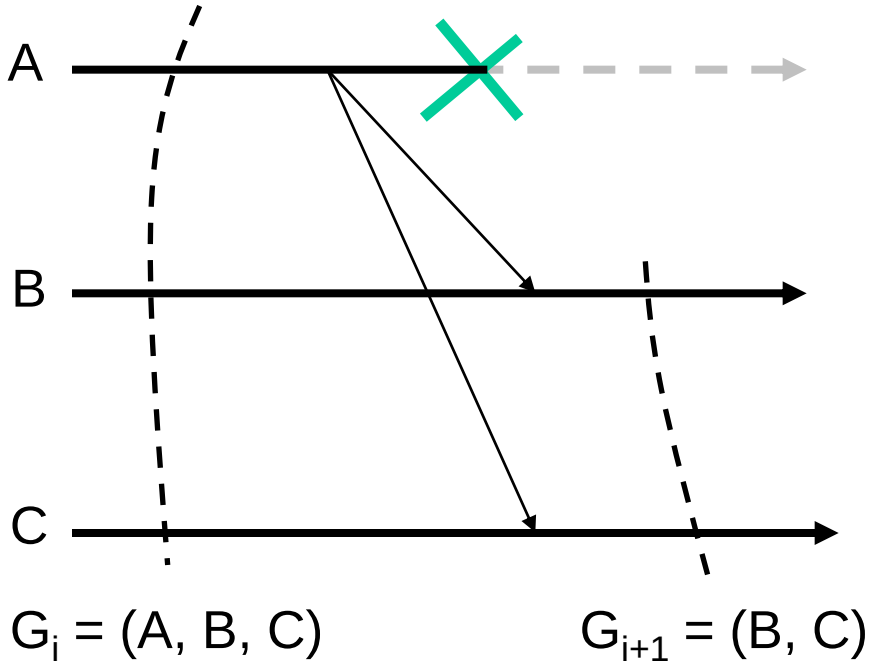


Virtual Synchronous Multicast

a) Message is **not** delivered

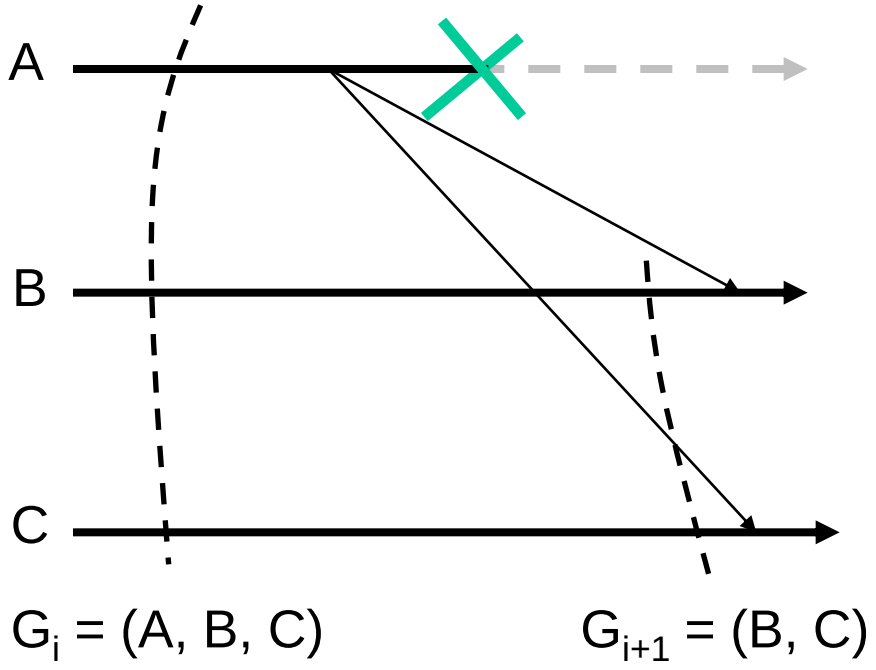


b) Message is delivered

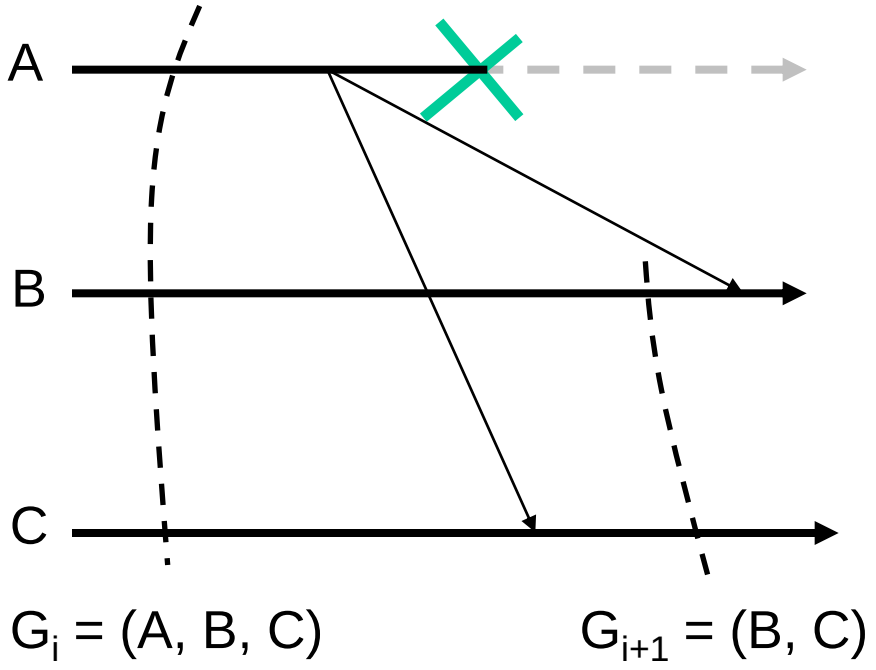


Virtual Synchronous Multicast

a) Message is **not** delivered

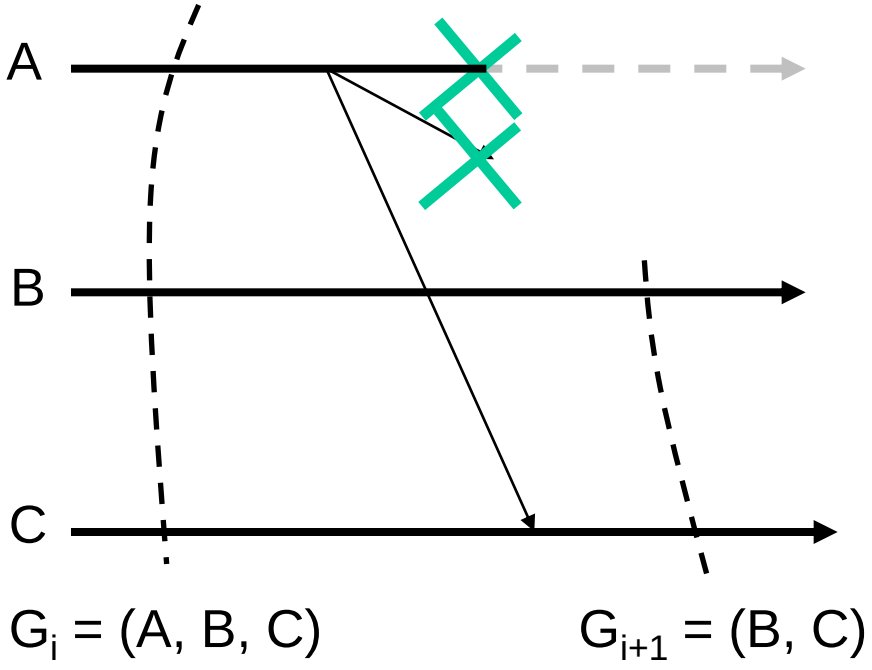


b) ???



Virtual Synchronous Multicast

a) ???

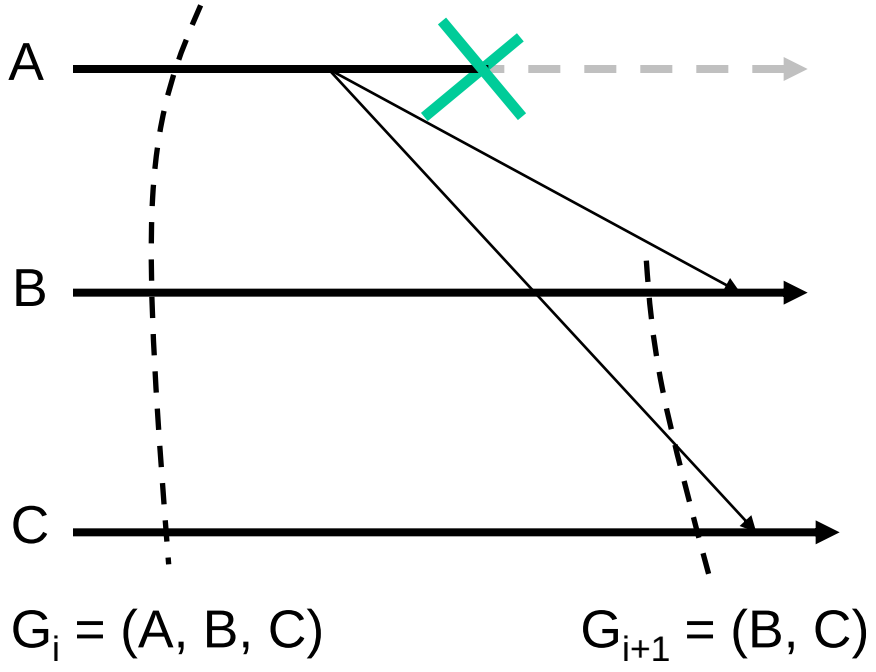


Reliability of Group Communication?

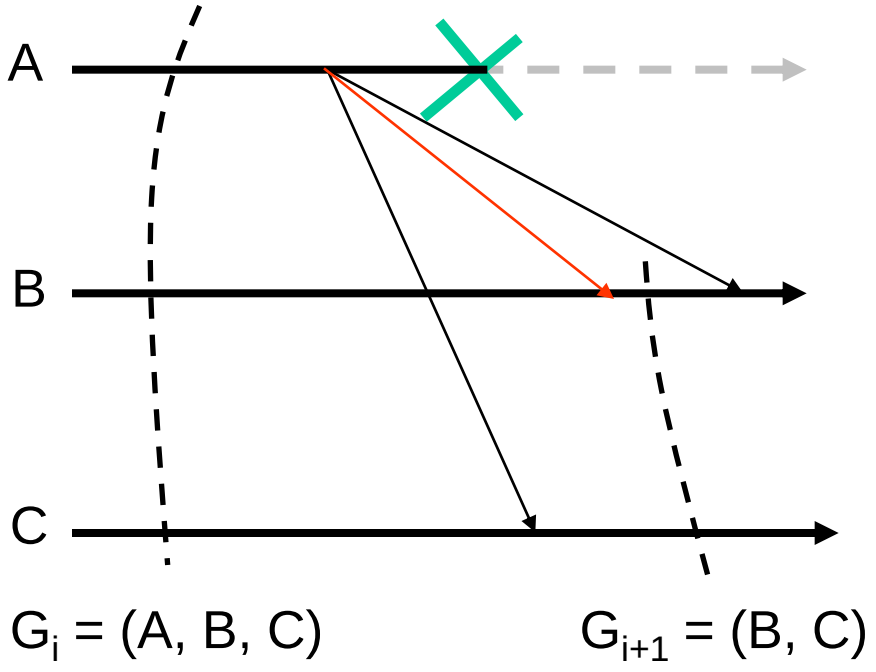
- A sent message is received by all members
  (acks from all => ok)
- Problem: during a multicast operation
  - an old member disappears from the group
  - a new member joins the group
- Solution
  - membership changes synchronize multicasting
    ⇒ during a MC operation no membership changes
  - Virtual synchrony: “all” processes see message and membership change in the same order

Virtual Synchronous Multicast

a) Message is **not** delivered



b) Message is delivered

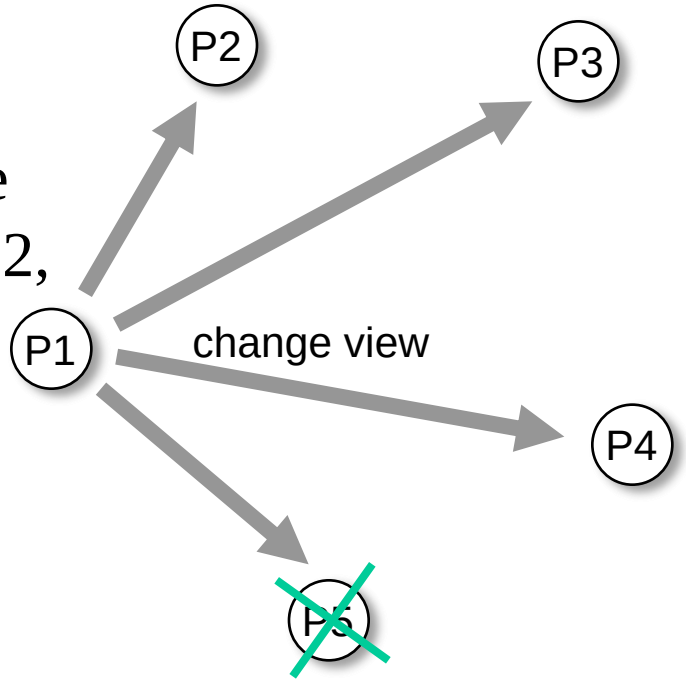


Virtual Synchrony Implementation: [Birman et al., 1991]

- Only **stable** messages are delivered
- **Stable message**: a message received by all processes in the message's group view
- Assumptions (can be ensured by using TCP):
  - Point-to-point communication is reliable
  - Point-to-point communication ensures FIFO-ordering
- How to determine if a message is stable?

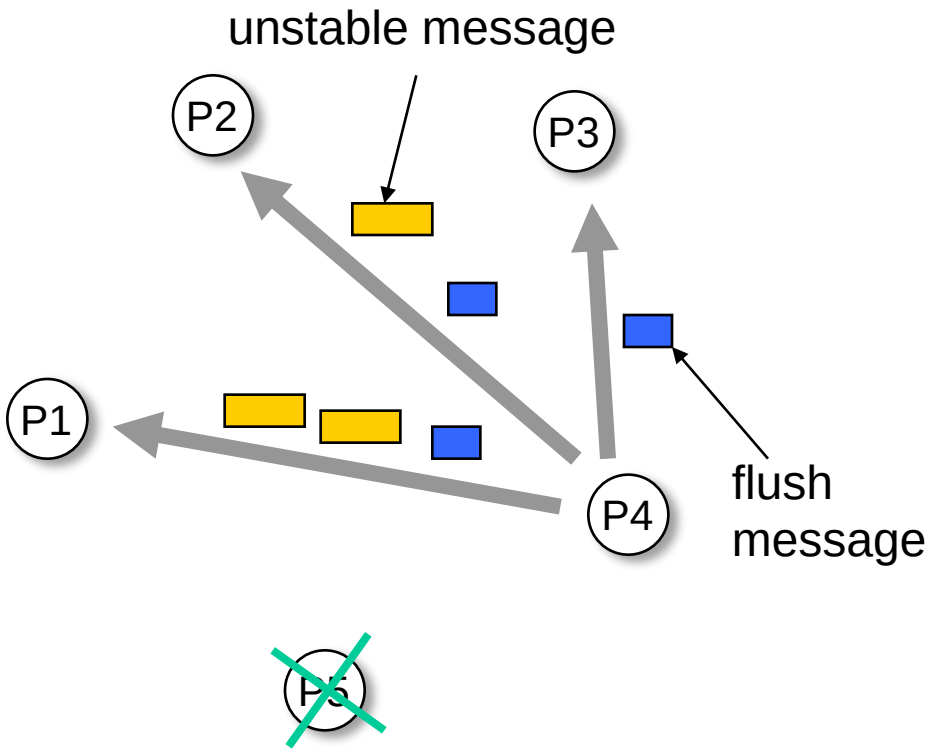
Virtual Synchrony Implementation: Example

- G_i = {P1, P2, P3, P4, P5}
- P5 fails
- P1 detects that P5 has failed
- P1 send a “view change” message to every process in G_{i+1} = {P1, P2, P3, P4}



Virtual Synchrony Implementation: Example

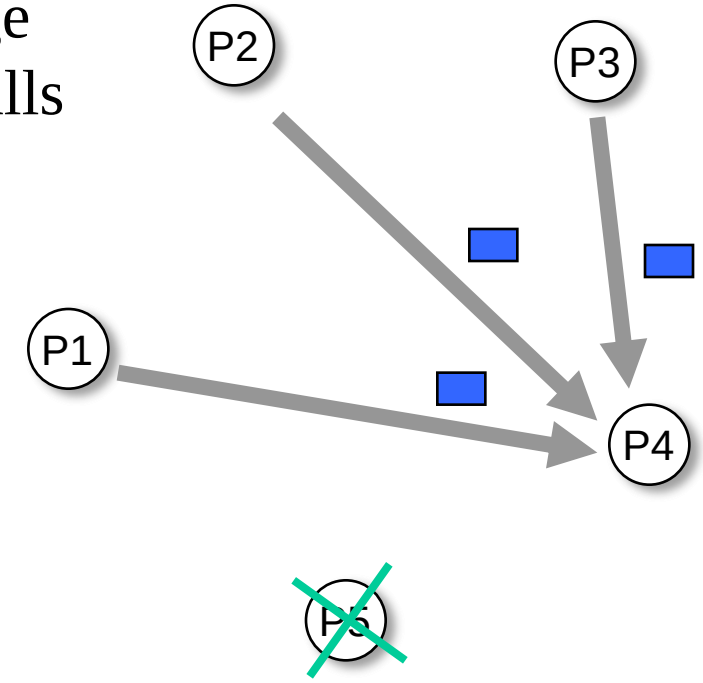
- Every process
  - Send each unstable message *m* from G_i to members in G_{i+1}
  - Marks *m* as being stable
  - Send a flush message to mark that all unstable messages have been sent



Virtual Synchrony Implementation: Example

Every process

- After receiving a flush message from all processes in G_{i+1} installs G_{i+1}



Announcement

- 2nd Midterm in the week after Spring Break
  - March 27, Wednesday

- Chapters 6, 7, 8.1, & 8.2

Distributed Commit

- **Goal**: Either **all** members of a group decide to perform an operation, or **none** of them perform the operation

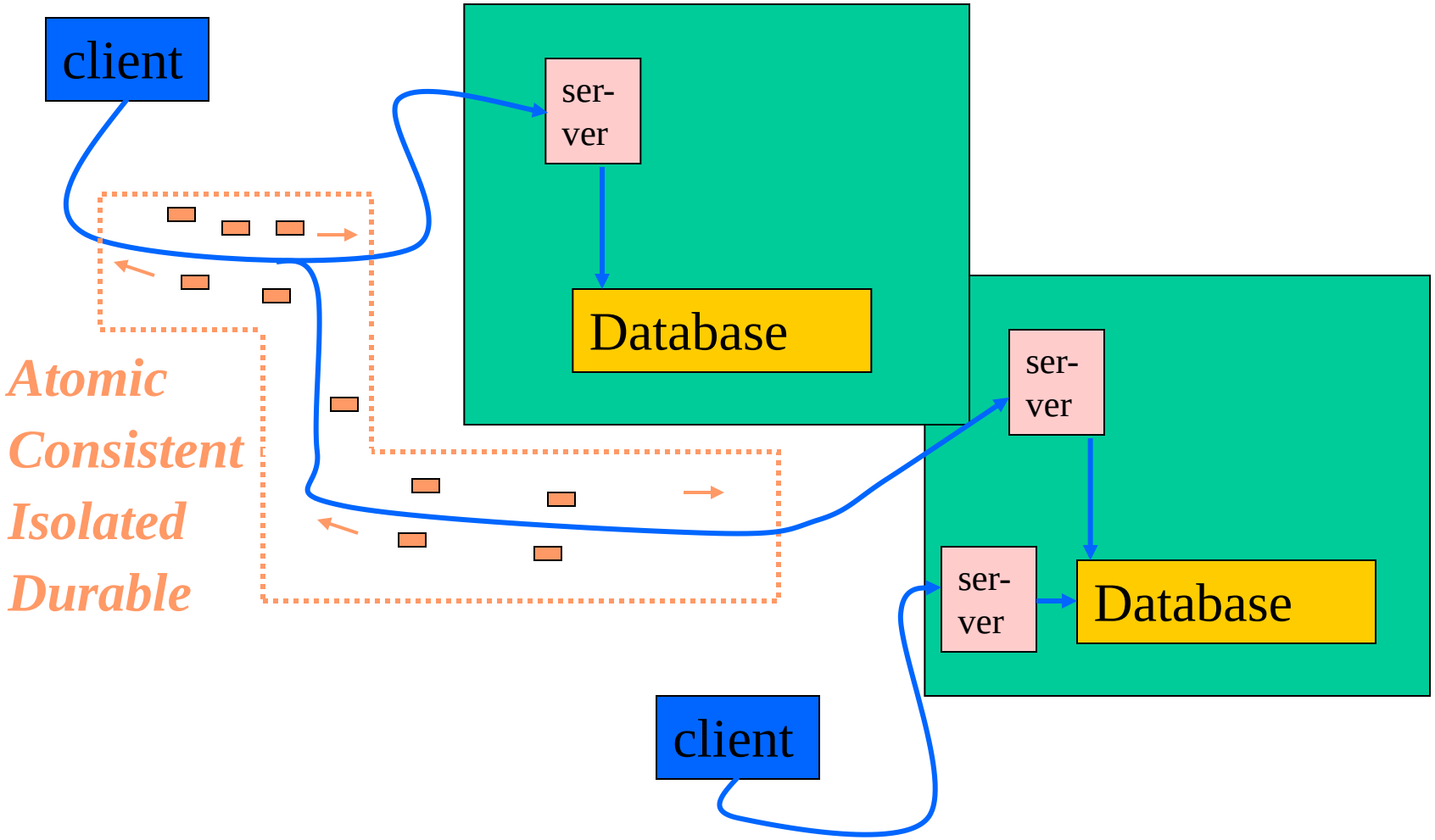
- **Atomic transaction**: a transaction that happens completely or not at all

Assumptions

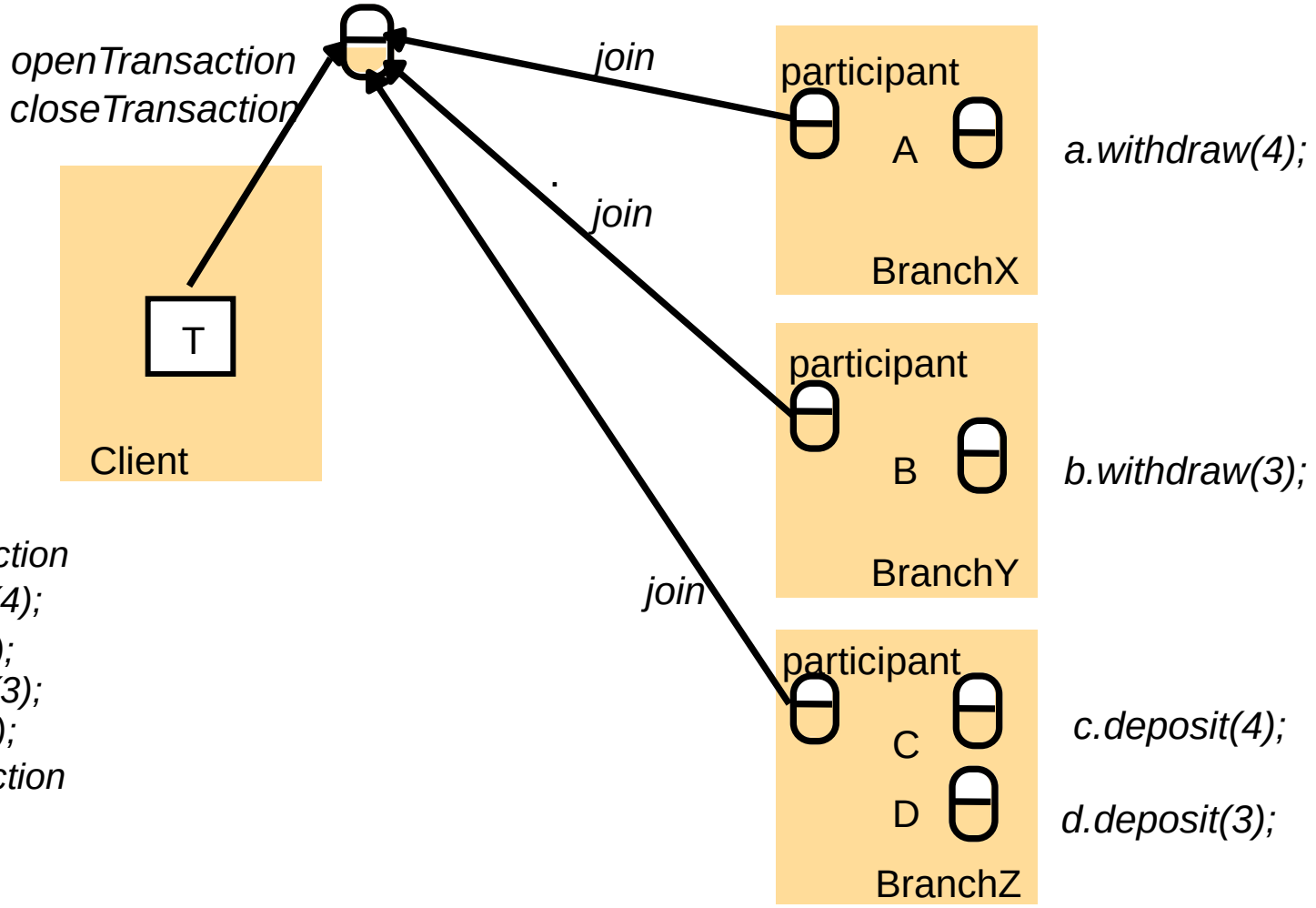
- Failures:
  - Crash failures that can be recovered
  - Communication failures detectable by timeouts

- Notes:
  - Commit requires a set of processes to agree…
  - …similar to the Byzantine generals problem…
  - … but the solution much simpler because stronger assumptions

Distributed Transactions



A Distributed Banking Transaction

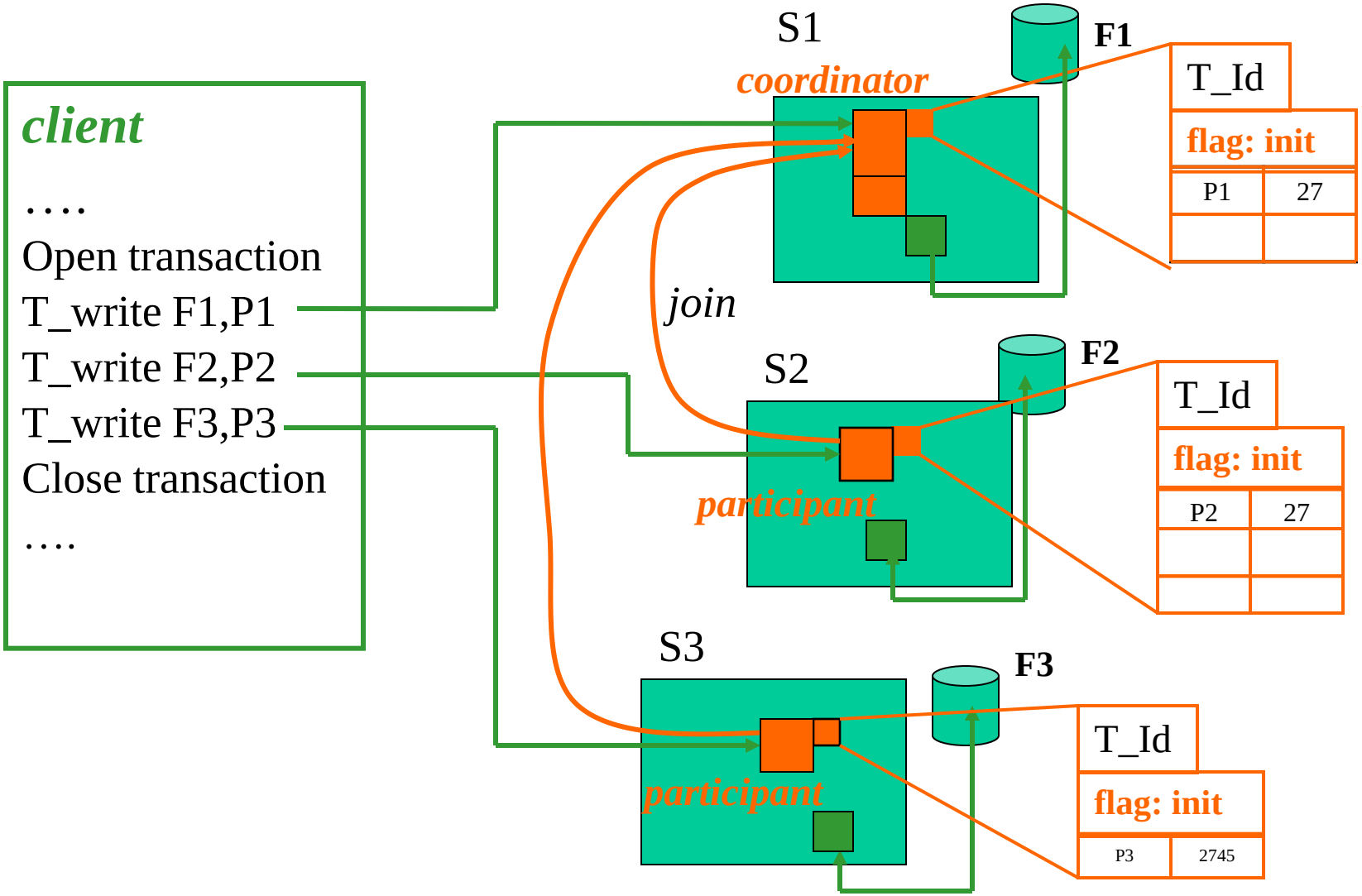


T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

One-phase Commit

- One-phase commit protocol
  - One site is designated as a coordinator
  - The coordinator tells all the other processes whether or not to locally perform the operation in question
  - This scheme however is not fault tolerant

Transaction Processing (1)



Transaction Processing (2)

client

….

Open transaction

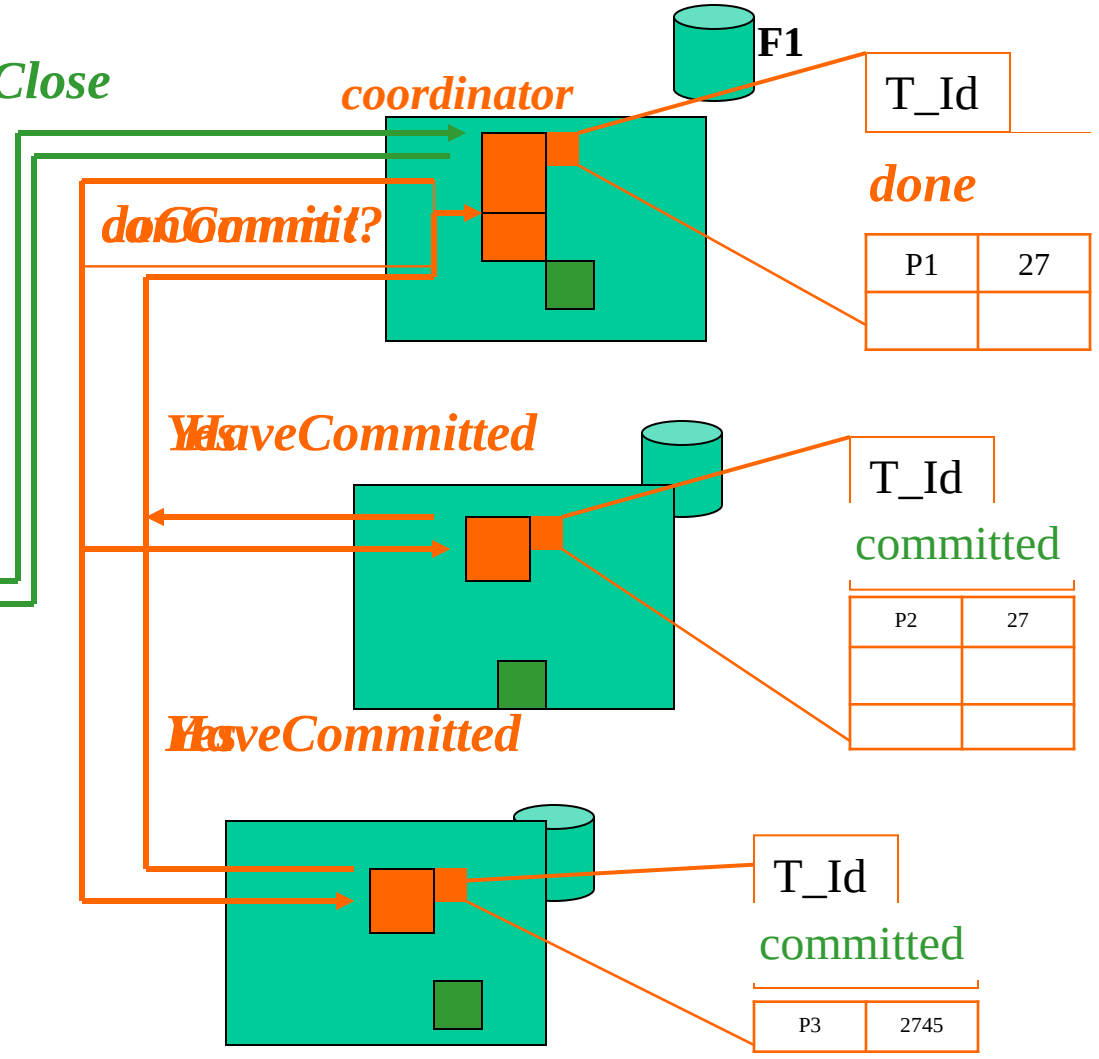
T_read F1,P1

T_write F2,P2

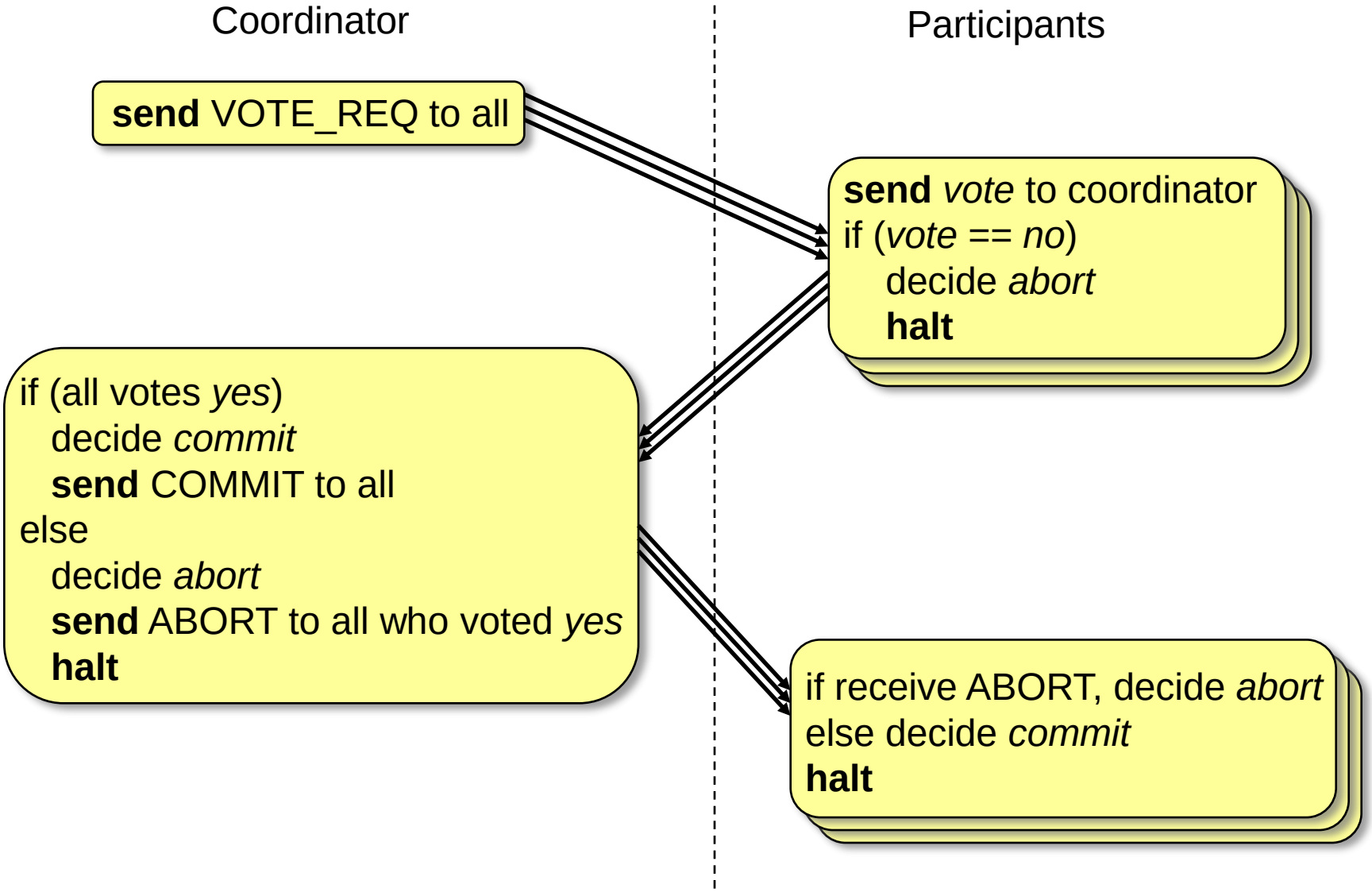
T_write F3,P3

Close transaction

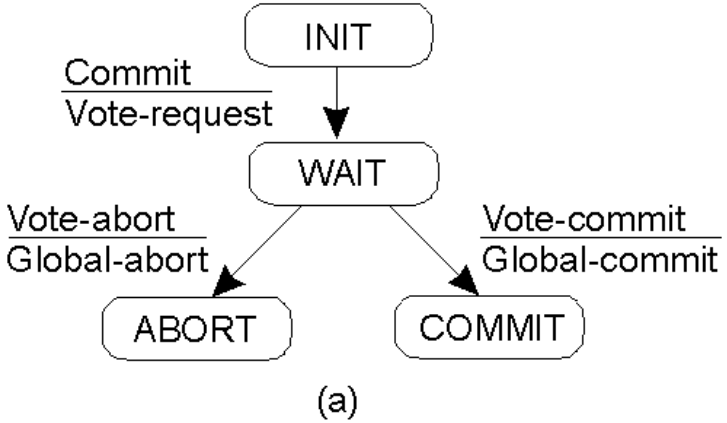
….

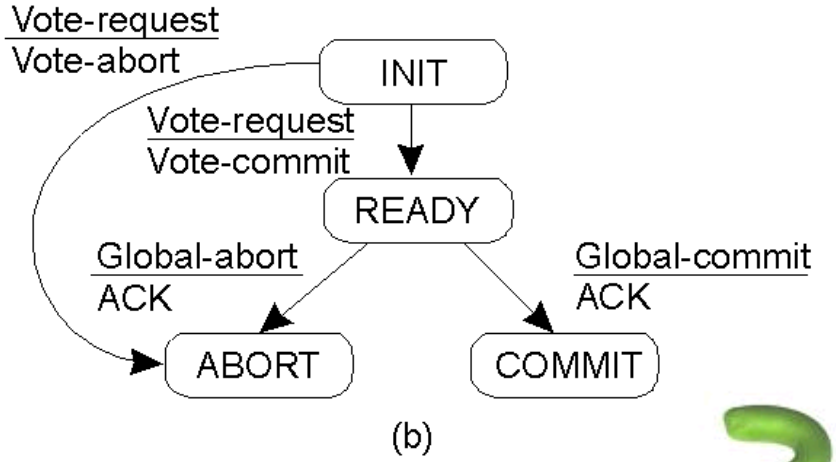


Two Phase Commit (2PC)



Two-Phase Commit (1)

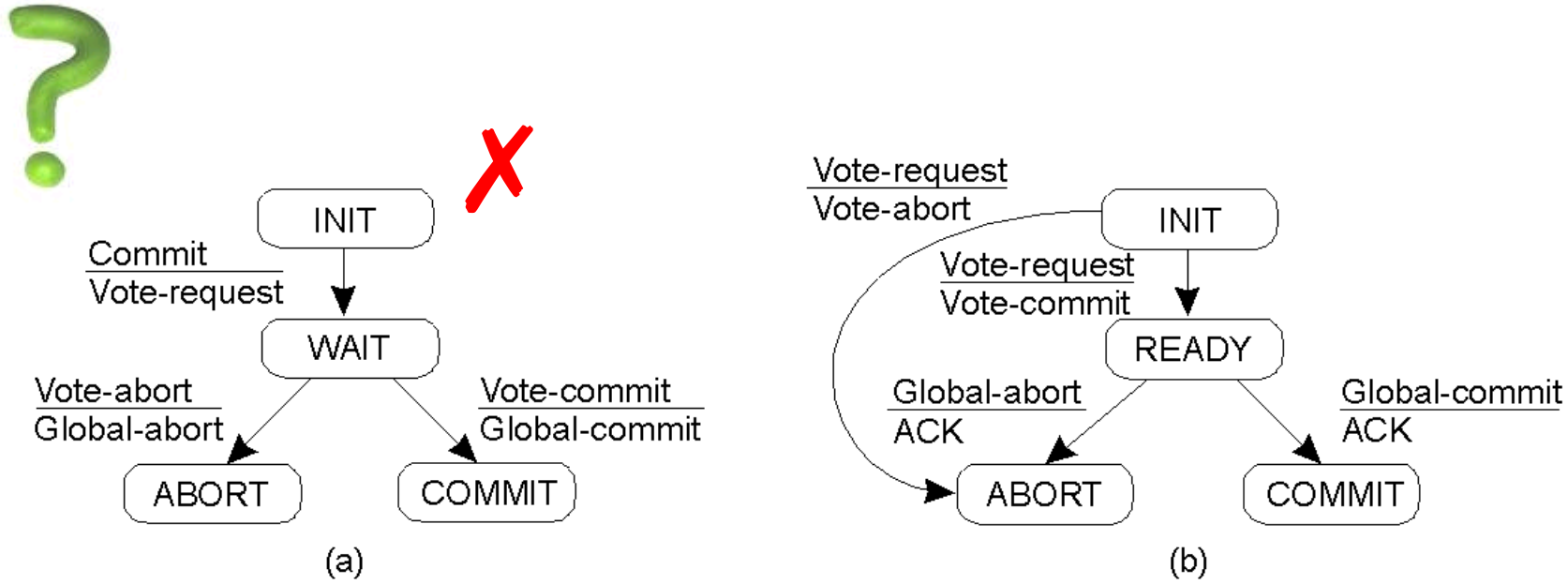




a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.

Two-Phase Commit (2)



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

b) The finite state machine for a participant.

Two-Phase Commit (3)

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 (4)

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;
}
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 2PC.

Two-Phase Commit (5)

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 (6)

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 */
```

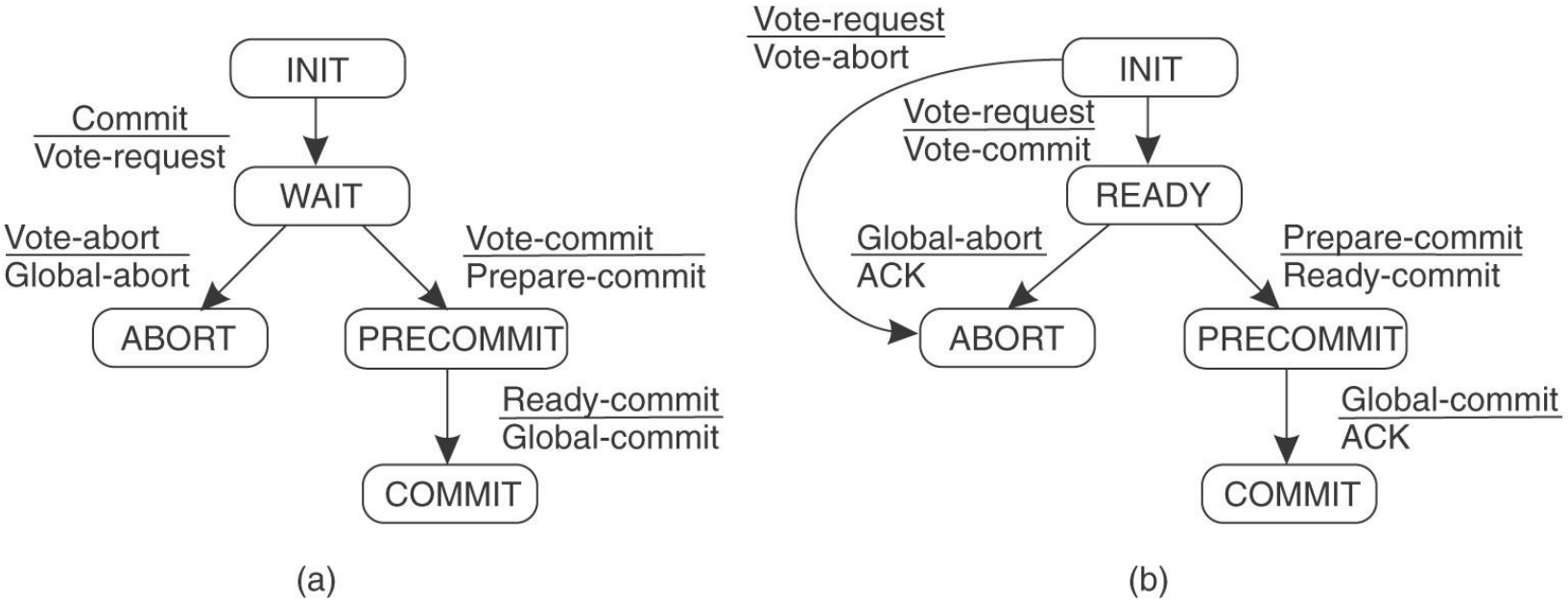
Steps taken by participant process for handling incoming decision requests.

Two-Phase Commit(7)

- When all participants are in the ready states, no final decision can be reached

- Two-phase commit is a blocking commit protocol

Three-Phase Commit (1)



- There is no state from which a transition can be made to either *Commit* or *Abort*
- There is no state where it is not possible to make a final decision and from which transition can be made to *Commit*
- ⇒ non-blocking commit protocol

Three-Phase Commit (2)

- Coordinator sends *Vote_Request* (as before)
- If all participants respond affirmatively,
  - Put *Precommit* state into log on stable storage
  - Send out *Prepare_to_Commit* message to all
- After all participants acknowledge,
  - Put *Commit* state in log
  - Send out *Global_Commit*

Three-Phase Commit (3)

- Coordinator blocked in *Wait* state
  - Safe to abort transaction
- Coordinator blocked in *Precommit* state
  - Safe to issue *Global_Commit*
  - Any crashed or partitioned participants will commit when recovered

Three-Phase Commit (4)

- Participant blocked in *Precommit* state
  - Contact others
  - Collectively decide to commit
- Participant blocked in *Ready* state
  - Contact others
  - If any in *Abort*, then abort transaction
  - If any in *Precommit*, the move to *Precommit* state
  - If all in *Ready* state, then abort transaction

Chapter 8

Fault Tolerance

Part V

Recovery

Recovery

- We've talked a lot about fault tolerance, but not about what happens after a fault has occurred
- A process that exhibits a failure has to be able to recover to a correct state
- There are two basic types of recovery:
  - Backward Recovery
  - Forward Recovery

Backward Recovery

- The goal of *backward recovery* is to bring the system from an erroneous state back to a prior correct state
- The state of the system must be recorded - *checkpointed* - from time to time, and then restored when things go wrong
- Examples
  - Reliable communication through packet retransmission

Forward Recovery

- The goal of *forward recovery* is to bring a system from an erroneous state to a correct new state (not a previous state)

- Examples:
  - Reliable communication via erasure correction, such as an (n, k) block erasure code

More on Backward Recovery

- Backward recovery is far more widely applied
- The goal of *backward recovery* is to bring the system from an erroneous state back to a prior correct state
- But, how to get a prior correct state?
  - Checkpointing
  - Checkpointing is costly, so it's often combined with *message logging*

Stable Storage

- In order to store checkpoints and logs, information needs to be stored safely - not just able to survive crashes, but also able to survive hardware faults

- RAID is the typical example of stable storage

Checkpointing

- Related to checkpointing, let us first discuss the *global state* and the *distributed snapshot algorithm*

Determining Global States

- **The global state of a distributed computation is**
  - *the set of local states of all individual processes involved in the computation*

    +
  - *the states of the communication channels*

- **How?**

Obvious First Solution…

- Synchronize clocks of all processes and ask all processes to record their states at known time t

- Problems?
  - Time synchronization possible only approximately
    - distributed banking applications: no approximations!
  - Does not record the state of messages in the channels

Global State

- We cannot determine the exact global state of the system, but we can record a snapshot of it

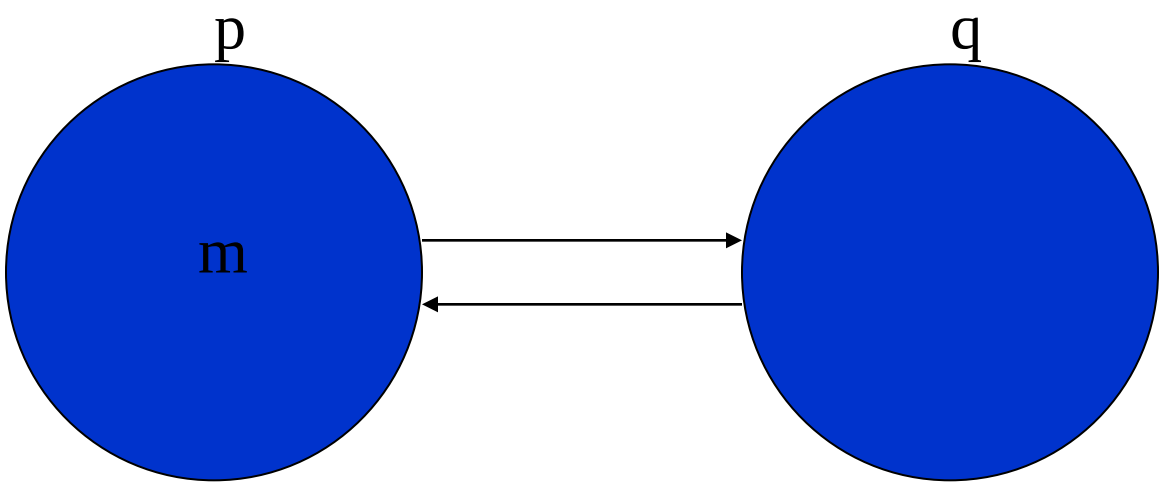
- **Distributed Snapshot**: a state the system might have been in [Chandy and Lamport]

A naïve snapshot algorithm

- Processes record their states at *any* arbitrary points
- A designated process collects these states

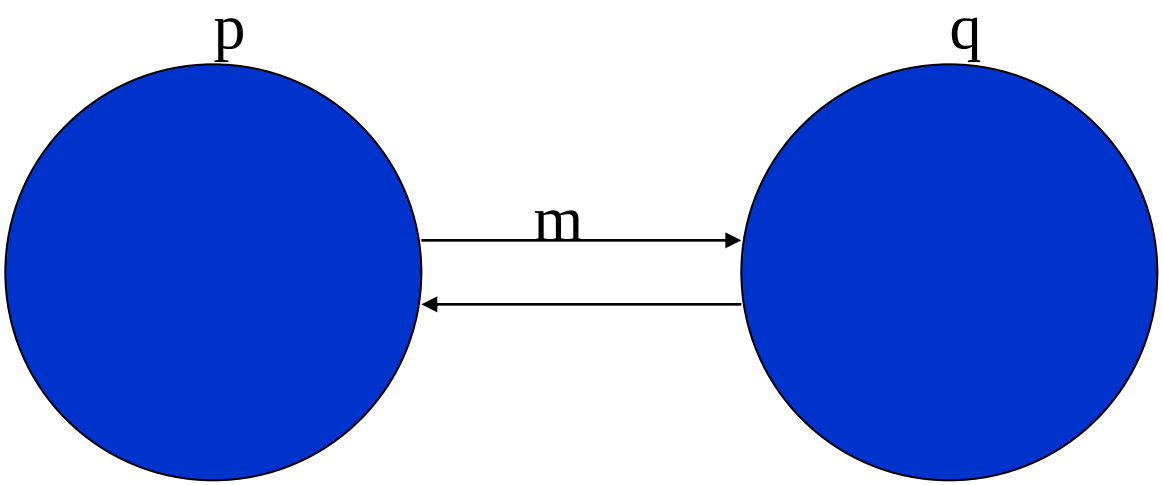
- \+ So simple!!
- \- Correct??

Example
Producer Consumer problem

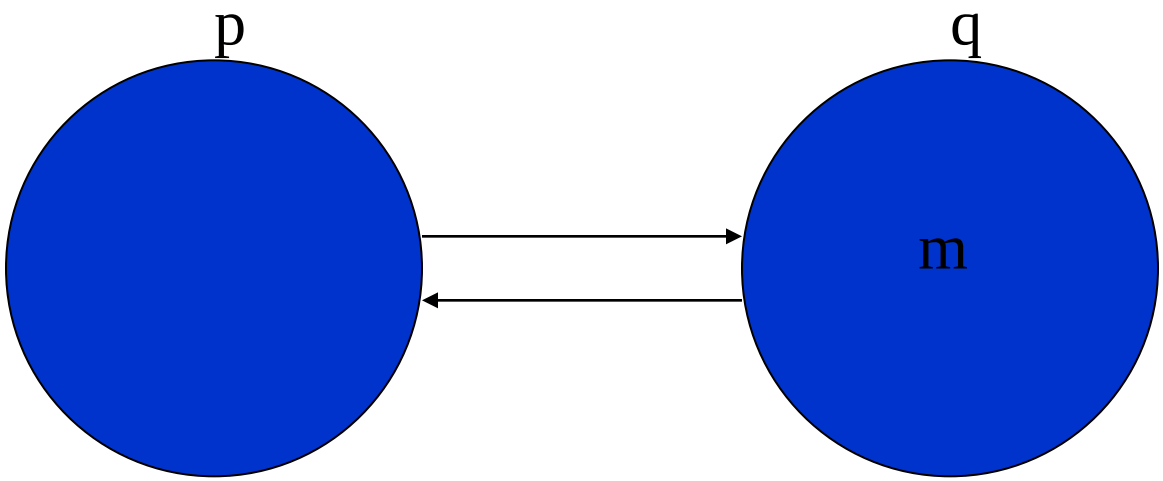


- p records its state

Example

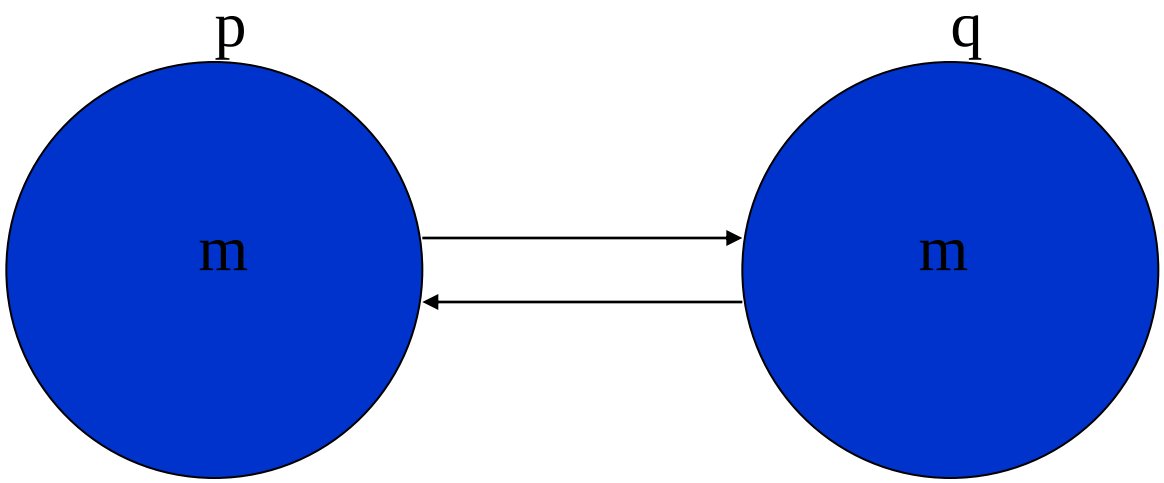


Example



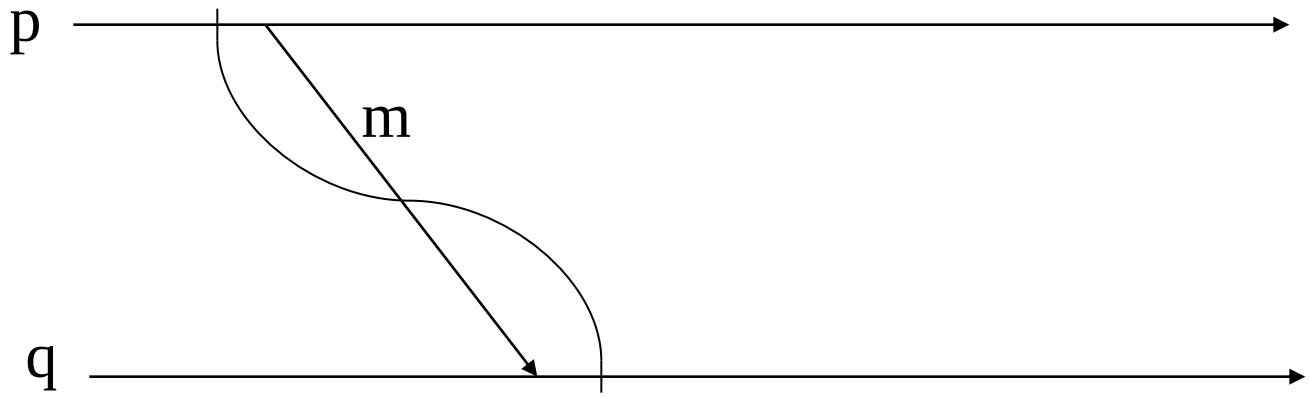
- q records its state

Example
The recorded state



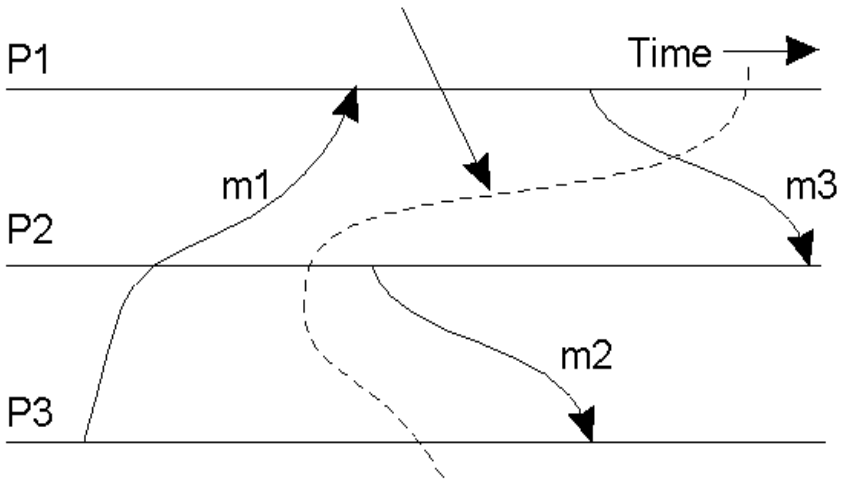
The sender has *no* record of the sending
The receiver *has* the record of the receipt

What's Wrong?



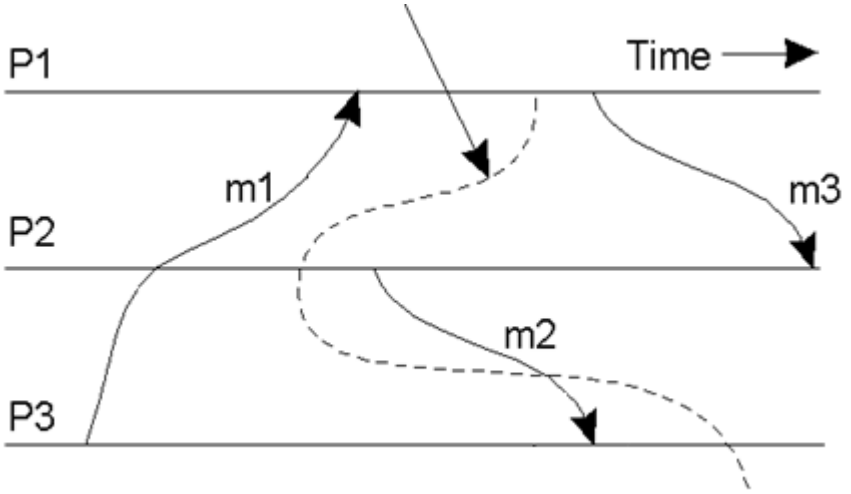
- Result:
  - Global state has record of the receive event but no send event violating the happens-before concept!!

Cut



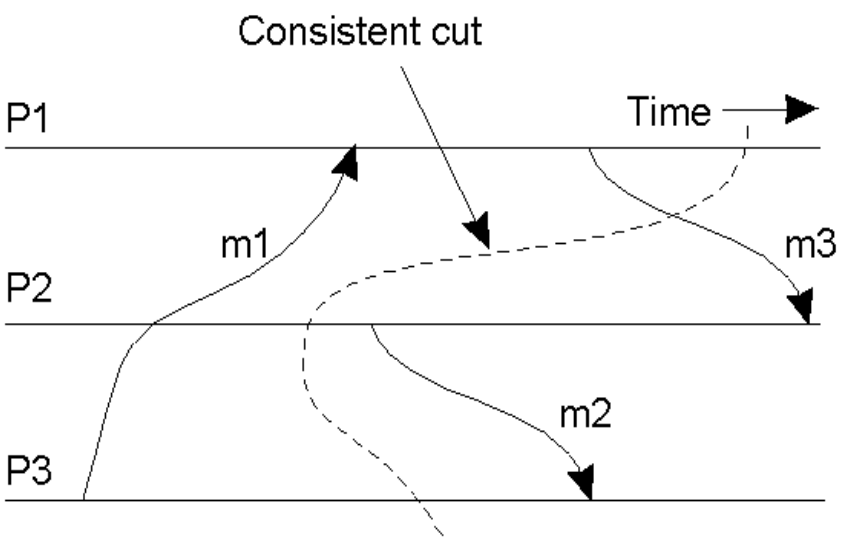
A consistent cut (**meaningful global state**) ?

Cut

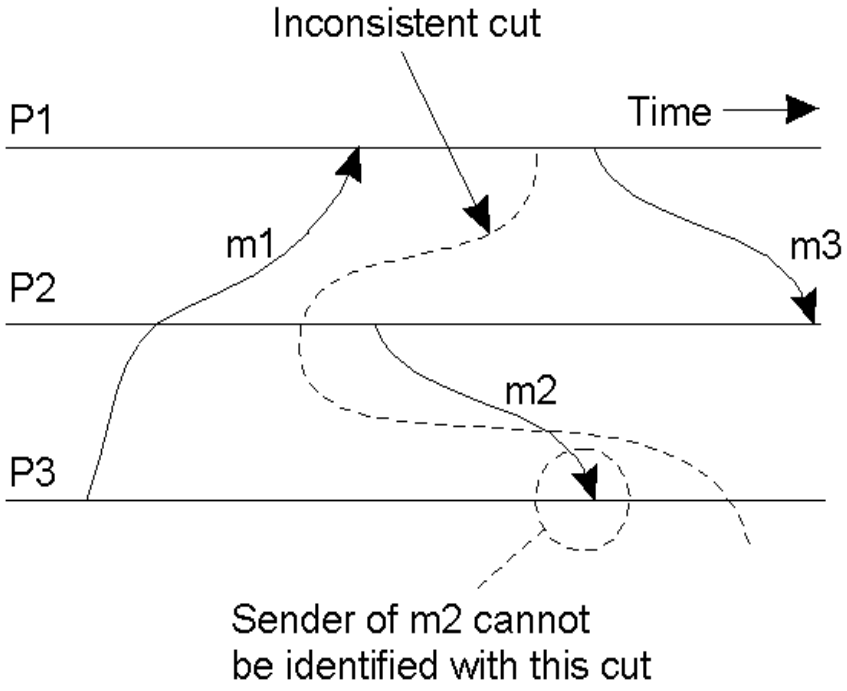


A consistent cut (**meaningful global state**) ?

Cuts



(a)



(b)

a) A consistent cut (**meaningful global state**)

b) An inconsistent cut

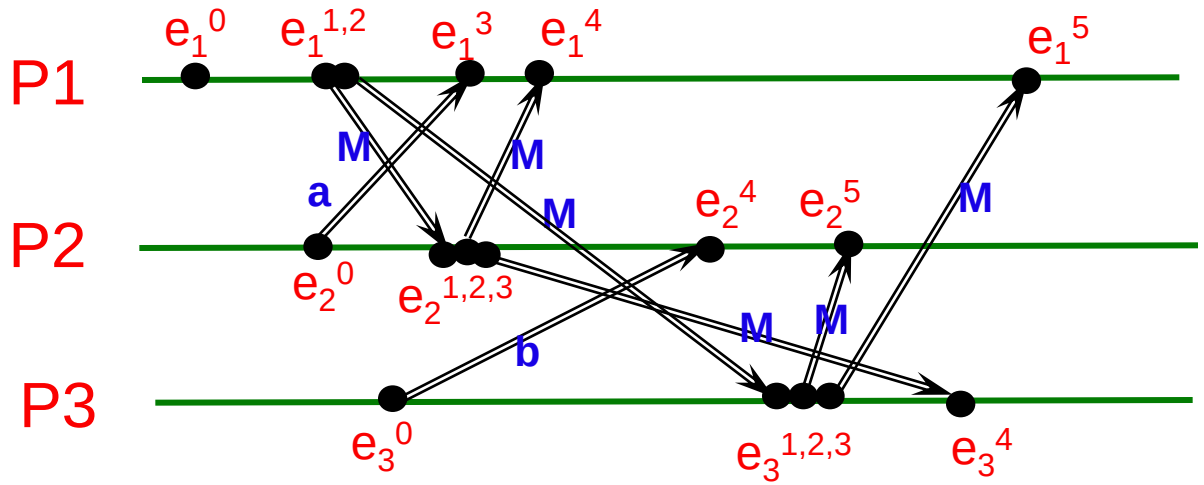
The "Snapshot" Algorithm

- **Records a set of process and channel states such that the combination is a consistent GS.**
- ***Assumptions:***
  - **All messages arrive intact, exactly once**
  - **Communication channels are unidirectional and FIFO-ordered**
  - **There is a comm. path between any two processes**
  - **Any process may initiate the snapshot (sends Marker)**
  - **Snapshot does not interfere with normal execution**
  - **Each process records its state and the state of its incoming channels**

The "Snapshot" Algorithm (2)

- **1. Marker sending rule for initiator process P_0**
  - After P_0 has recorded its state
    - for each outgoing channel C, sends a marker on C
- **2. Marker receiving rule for a process P_k, on receipt of a marker over channel C**
  - if P_k has not yet recorded its state
    - records P_k's state
    - records the state of C as "empty"
    - turns on recording of messages over other incoming channels
    - for each outgoing channel C, sends a marker on C
  - else
    - records the state of C as all the messages received over C since P_k saved its state

Snapshot Example



1- P1 initiates snapshot: records its state (S1); sends Markers to P2 & P3; turns on recording for channels C21 and C31

2- P2 receives Marker over C12, records its state (S2), sets state(C12) = {} sends Marker to P1 & P3; turns on recording for channel C32

3- P1 receives Marker over C21, sets state(C21) = {a}

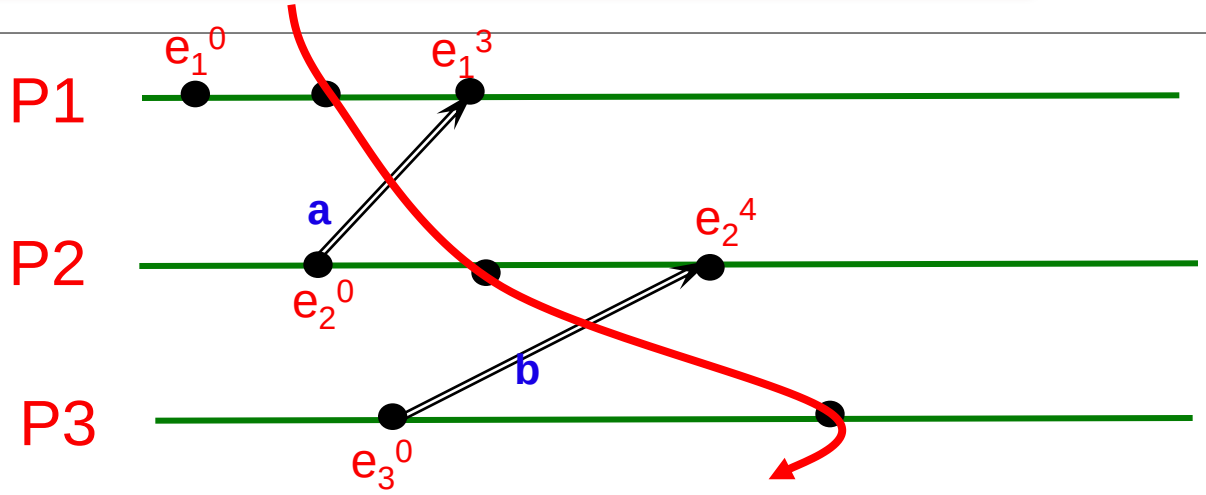
4- P3 receives Marker over C13, records its state (S3), sets state(C13) = {} sends Marker to P1 & P2; turns on recording for channel C23

5- P2 receives Marker over C32, sets state(C32) = {b}

6- P3 receives Marker over C23, sets state(C23) = {}

7- P1 receives Marker over C31, sets state(C31) = {}

Snapshot Example

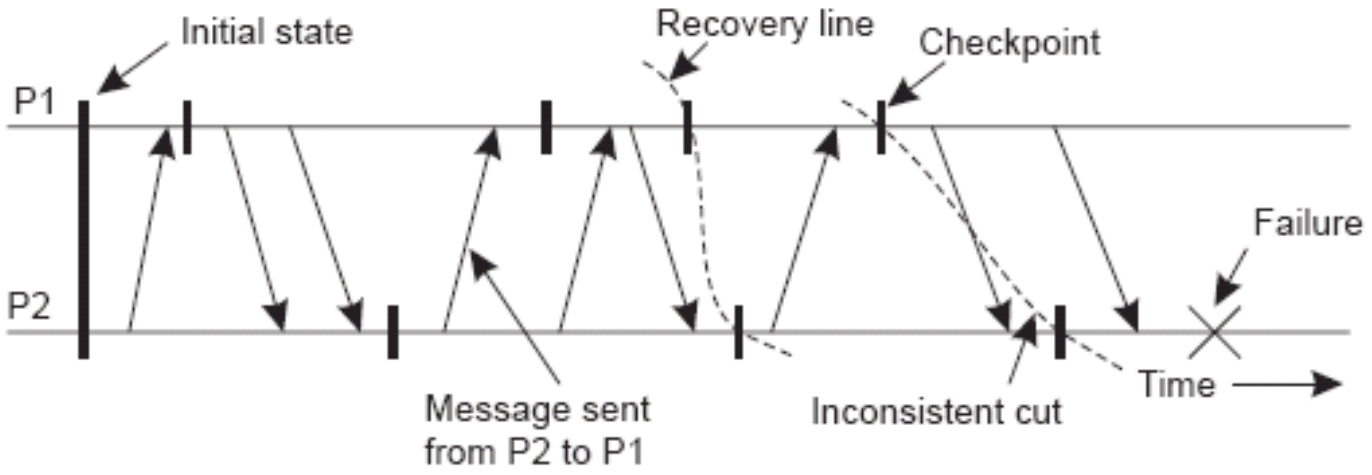


Distributed Snapshot Algorithm

- When a process finishes local snapshot, it collects its local state (S and C) and sends it to the initiator of the distributed snapshot
- The initiator can then analyze the state
- One algorithm for distributed global snapshots, but it's not particularly efficient for large systems

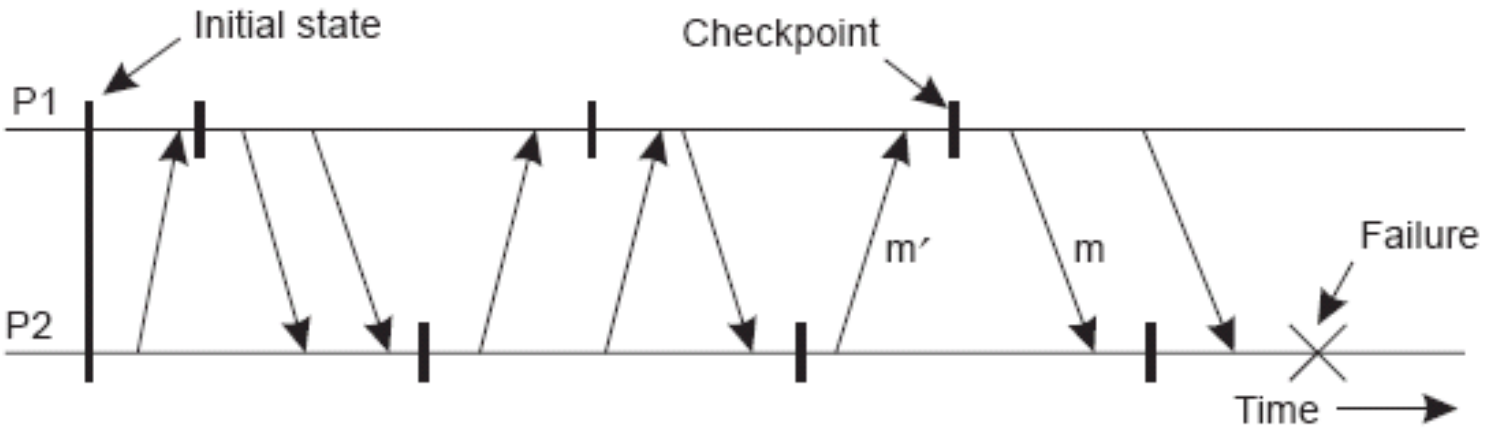
Checkpointing

- We've discussed distributed snapshots
- The most recent distributed snapshot in a system is also called the *recovery line*



Independent Checkpointing

- It is often difficult to find a recovery line in a system where every process just records its local state every so often - a *domino effect* or cascading rollback can result:



Coordinated Checkpointing

- To solve this problem, systems can implement *coordinated checkpointing*
- We've discussed one algorithm for distributed global snapshots, but it's not particularly efficient for large systems
- Another way to do it is to use a two-phase blocking protocol (with some coordinator) to get every process to checkpoint its local state "simultaneously"

Coordinated Checkpointing

- Make sure that processes are synchronized when doing the checkpoint
- Two-phase blocking protocol
  1. Coordinator multicasts *CHECKPOINT_REQUEST*
  2. Processes take local checkpoint
     - Delay further sends
     - Acknowledge to coordinator
     - Send state
  3. Coordinator multicasts CHECKPOINT_DONE

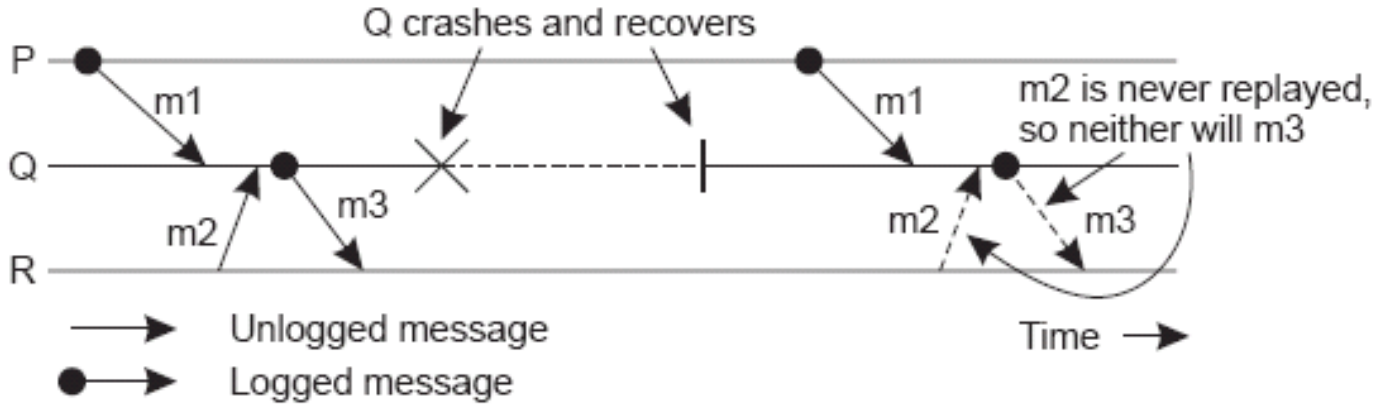
Message Logging

- Checkpointing is expensive - message logging allows the occurrences between checkpoints to be *replayed*, so that checkpoints don't need to happen as frequently

Message Logging

- We need to choose when to log messages
- Message-logging schemes can be characterized as *pessimistic* or *optimistic* by how they deal with *orphan processes*
  - An orphan process is one that survives the crash of another process but has an inconsistent state after the other process recovers

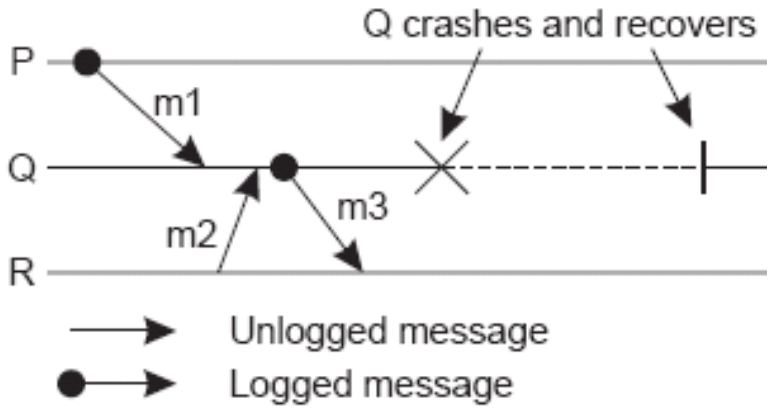
Message Logging



- An example of an incorrect replay of messages

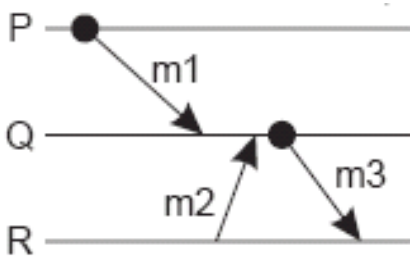
Message Logging

- We assume that each message *m* has a header containing all the information necessary to retransmit *m* (sender, receiver, sequence no., etc.)
- A message is called *stable* if it can no longer be lost - a stable message can be used for recovery by replaying its transmission



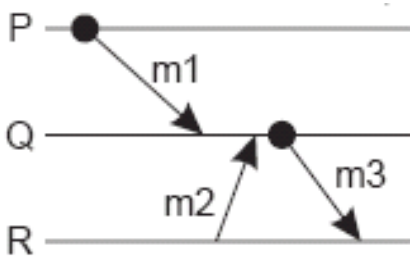
Message Logging

- Each message *m* leads to a set of dependent processes *DEP(m)*, to which either *m* or a message causally dependent on *m* has been delivered



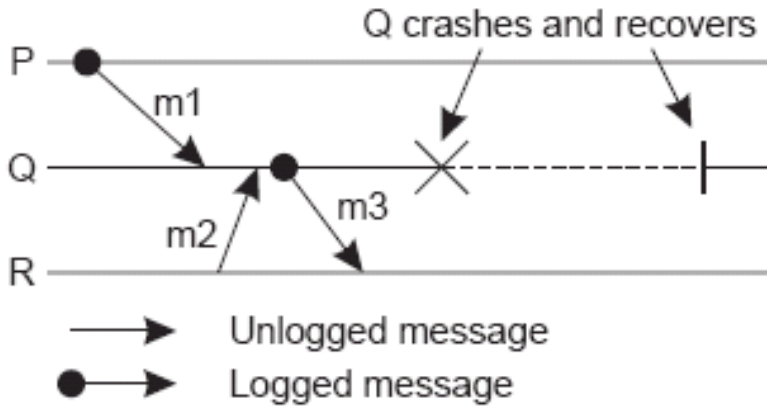
Message Logging

- The set *COPY(m)* consists of the processes that have a copy of *m*, but not in their local stable storage - any process in *COPY(m)* could deliver a copy of *m* on request



Message Logging

- Process *Q* is an orphan process if there is a nonstable message *m*, such that *Q* is contained in *DEP(m)*, and every process in *COPY(m)* has crashed

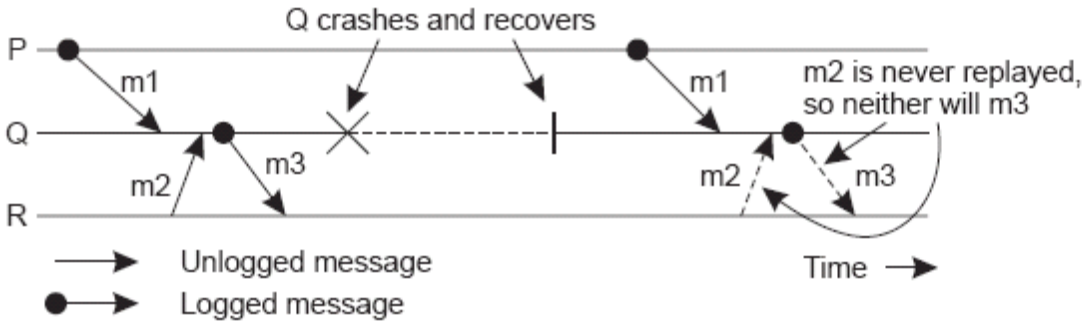


Message Logging

- To avoid orphan processes, we need to ensure that if all processes in *COPY(m)* crash, no processes remain in *DEP(m)*

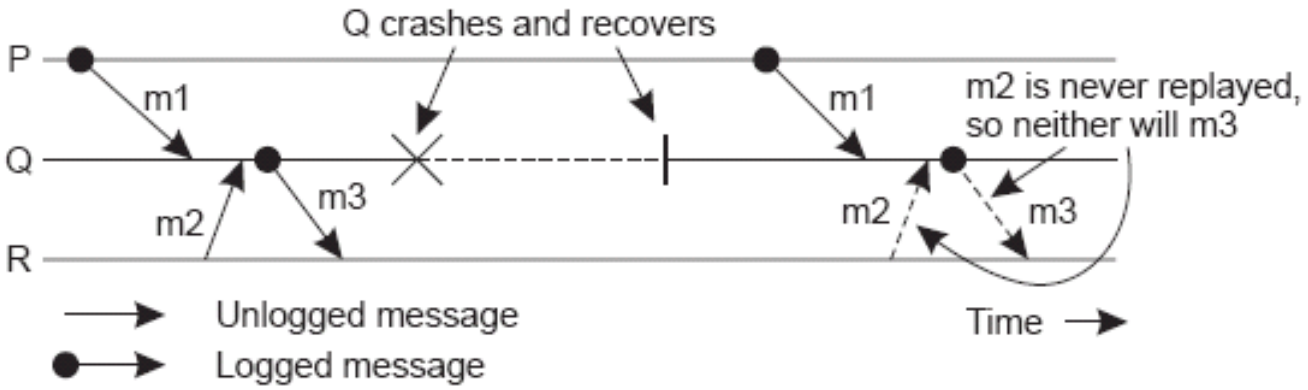
Pessimistic Logging

- For each *nonstable* message *m*, ensure that at most one process *P* is dependent on *m*
- The worst that can happen is that *P* crashes without *m* ever having been logged
- No other process can have become dependent on *m*, because *m* was nonstable, so this leaves no orphans



Optimistic Logging

- The work is done after a crash occurs, not before
- If, for some *m*, each process in *COPY(m)* has crashed, then any orphan process in *DEP(m)* gets rolled back to a state in which it no longer belongs in *DEP(m)*



Optimistic Logging

- The work is done after a crash occurs, not before
- If, for some *m*, each process in *COPY(m)* has crashed, then any orphan process in *DEP(m)* gets rolled back to a state in which it no longer belongs in *DEP(m)*
- Dependencies need to be explicitly tracked, which makes this difficult to implement - as a result, pessimistic approaches are preferred in real-world implementations