HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Fault Tolerance

Fall 2008

Jussi Kangasharju

Chapter Outline

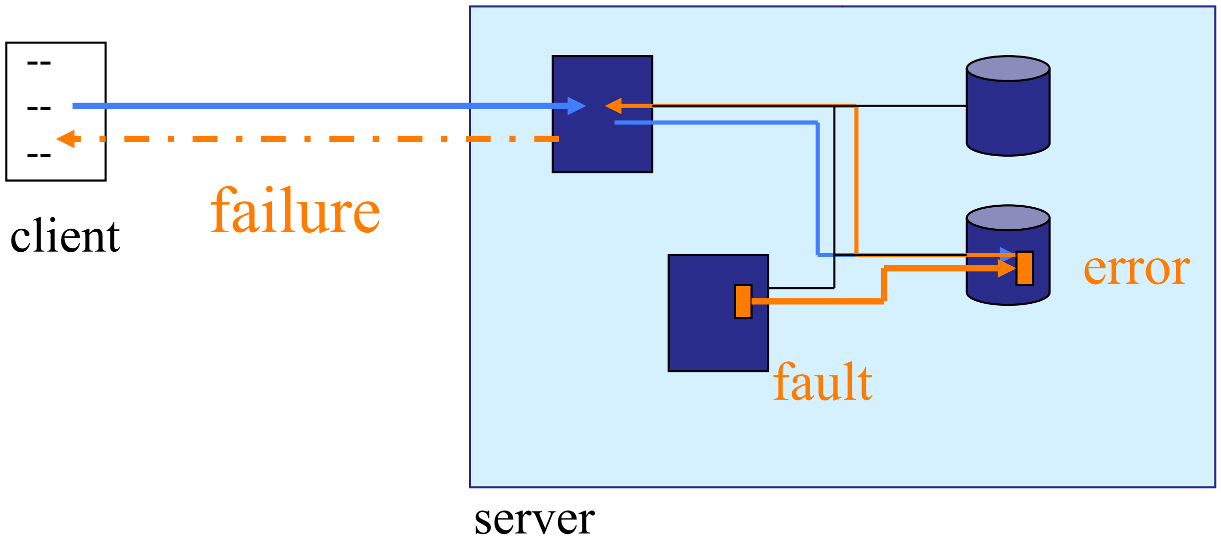
- Fault tolerance
- Process resilience
- Reliable group communication
- Distributed commit
- Recovery

Basic Concepts

Dependability includes

- Availability
- Reliability
- Safety
- Maintainability

Fault, error, failure



- Failure = toimintahäiriö
- Fault = vika
- Error = virhe(tila)

Failure Model

- Challenge: independent failures
- Detection
  - which component?
  - what went wrong?
- Recovery
  - failure dependent
  - ignorance increases complexity

  => taxonomy of failures

Fault Tolerance

- Detection
- Recovery
  - mask the error OR
  - fail predictably
- Designer
  - possible failure types?
  - recovery action (for the possible failure types)
- A fault classification:
  - transient (disappear)
  - intermittent (disappear and reappear)
  - permanent

Failure Models

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

Crash: fail-stop, fail-safe (*detectable*), fail-silent (*seems to have crashed*)

Failure Masking (1)

Detection

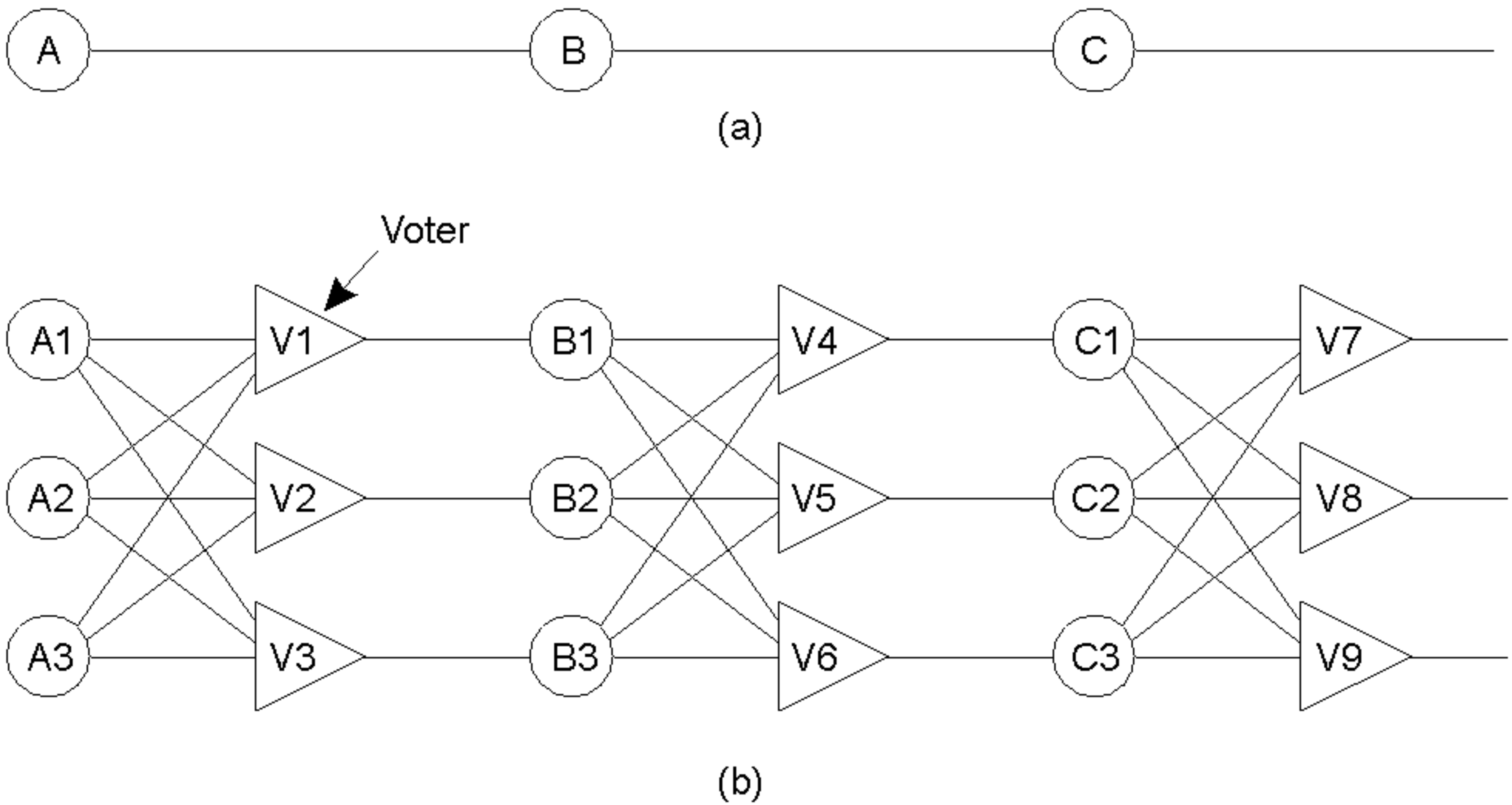
- redundant information
  - error detecting codes (parity, checksums)
  - replicas
- redundant processing
  - groupwork and comparison
- control functions
  - timers
  - acknowledgements

Failure Masking (2)

Recovery

- redundant information
  - error correcting codes
  - replicas
- redundant processing
  - *time redundancy*
    - retrial
    - recomputation (checkpoint, log)
  - *physical redundancy*
    - groupwork and voting
    - tightly synchronized groups

Example: Physical Redundancy



Triple modular redundancy.

Failure Masking (3)

- Failure models vs. implementation issues:

  the (sub-)system belongs to a class

  => certain failures do not occur

  => easier detection & recovery
- A point of view: forward vs. backward recovery
- Issues:
  - process resilience
  - reliable communication

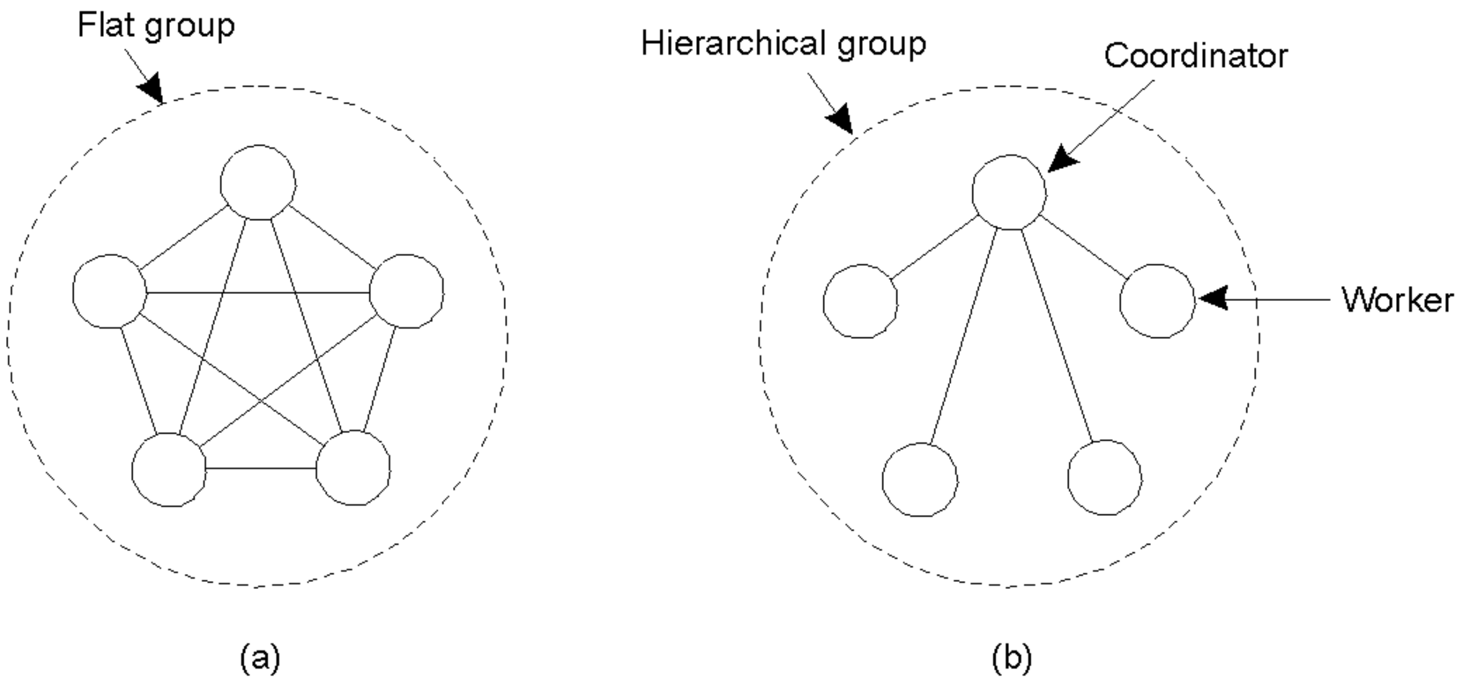
Process Resilience (1)

- Redundant processing: groups
  - Tightly synchronized
    - flat group: voting
    - hierarchical group:

      a **primary** and a **hot standby** (execution-level synchrony)
  - Loosely synchronized
    - hierarchical group:

      a **primary** and a **cold standby** (checkpoint, log)
- Technical basis
  - “group” – a single abstraction
  - reliable message passing

Flat and Hierarchical Groups (1)



Communication in a flat group.

Communication in a simple hierarchical group

Group management: a group server OR distributed management

Flat and Hierarchical Groups (2)

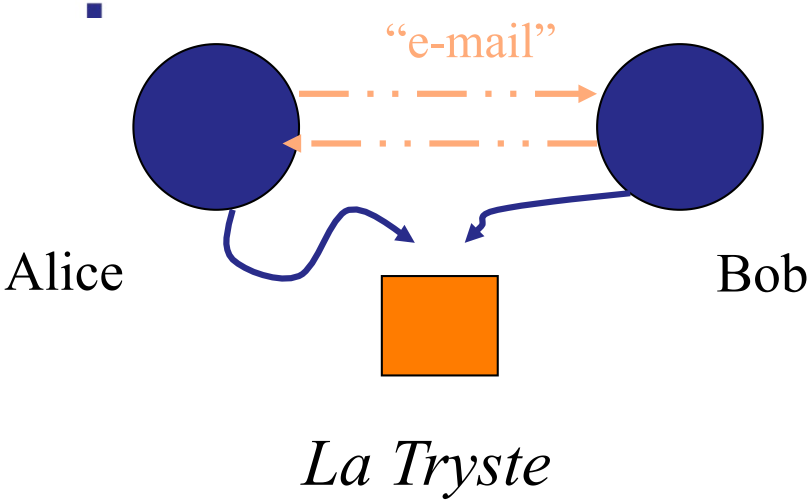
- Flat groups
  - symmetrical
  - no single point of failure
  - complicated decision making
- Hierarchical groups
  - the opposite properties

- Group management issues
  - join, leave;
  - crash (*no notification*)

Process Groups

- Communication vs management
  - application communication: message passing
  - group management: message passing
  - synchronization requirement:
    each group communication operation in a stable group
- Failure masking
  - **k fault tolerant**: tolerates k faulty members
    - fail silent: k + 1 components needed
    - Byzantine: 2k + 1 components needed
  - a precondition: **atomic multicast**
  - in practice: the probability of a failure must be “small enough”

Agreement in Faulty Systems (1)



on a rainy day …

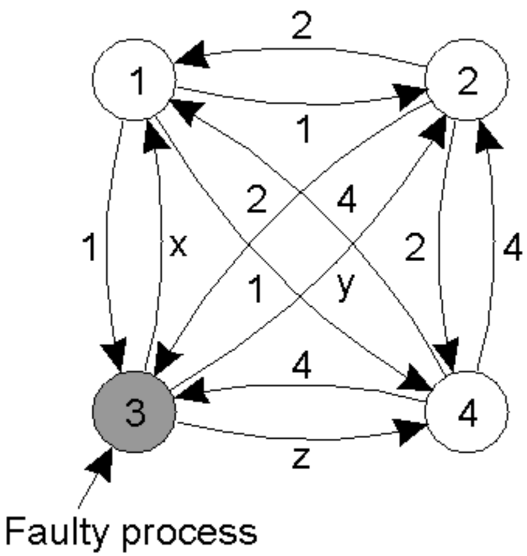
Requirement:
- an agreement
- within a bounded time

Faulty data communication: no agreement possible

Alice -> Bob *Let's meet at noon in front of La Tryste …*
Alice <- Bob *OK!!*
Alice: *If Bob doesn't know that I received his message, he will not come …*
Alice -> Bob *I received your message, so it's OK.*
Bob: *If Alice doesn't know that I received her message, she will not come …*
…

Agreement in Faulty Systems (2)

Reliable data communication, unreliable nodes



(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)

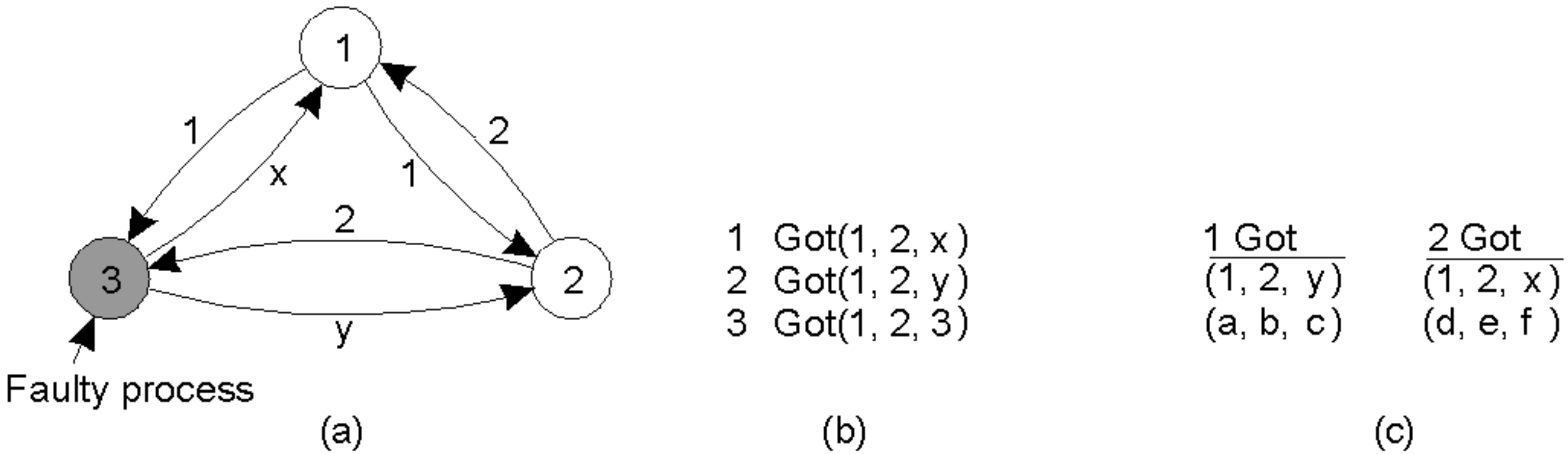
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The Byzantine generals problem for 3 loyal generals and 1 traitor.

a) The generals announce their troop strengths (in units of 1 kilosoldiers).
b) The vectors that each general assembles based on (a)
c) The vectors that each general receives in step 3.

Agreement in Faulty Systems (3)



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

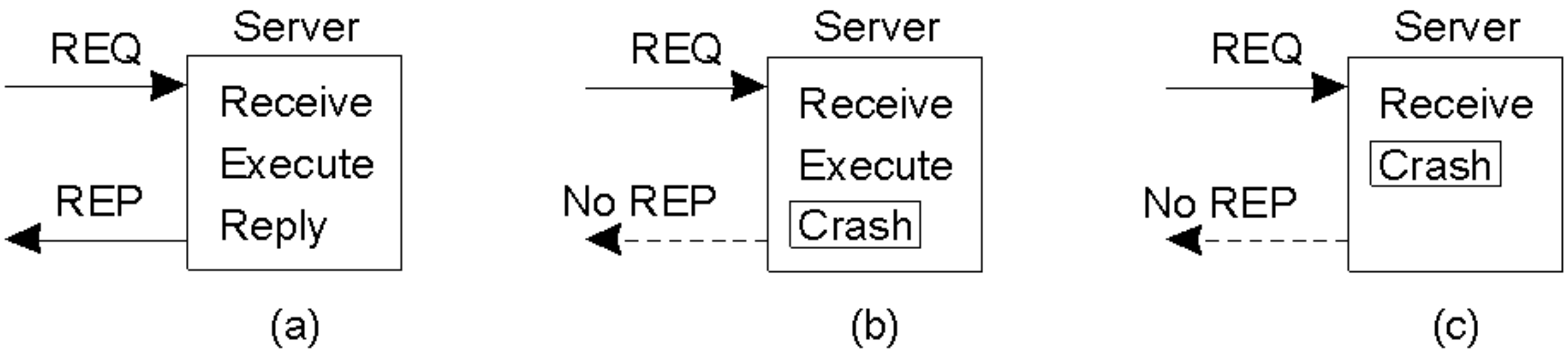
Agreement in Faulty Systems (4)

- An agreement can be achieved, when
  - message delivery is reliable with a bounded delay
  - processors are subject to Byzantine failures, but fewer than one third of them fail
- An agreement cannot be achieved, if
  - messages can be dropped (even if none of the processors fail)
  - message delivery is reliable but with unbounded delays, and even one processor can fail
- Further theoretical results are presented in the literature

Reliable Client-Server Communication

1. Point-to-Point Communication ("reliable")
   - masked: omission, value
   - not masked: crash, (timing)
2. RPC semantics
   - the client unable to locate the server
   - the message is lost (request / reply)
   - the server crashes (before / during / after service)
   - the client crashes

Server Crashes (1)



A server in client-server communication
a) Normal case
b) Crash after execution
c) Crash before execution

Server Crashes (2)

Client | Server

Reissue strategy	Strategy M -> P: MPC	MC(P)	C(MP)	Strategy P -> M: PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

Different combinations of client and server strategies in the presence of server crashes (client's continuation after server's recovery: reissue the request?)

M: send the completion message
P: print the text
C: crash

Client Crashes

- Orphan: an active computation looking for a non-existing parent

- Solutions
  - extermination: the client stub records all calls, after crash recovery all orphans are killed
  - reincarnation: time is divided into epochs, client reboot => broadcast "new epoch" => servers kill orphans
  - gentle incarnation: "new epoch" => only "real orphans" are killed
  - expiration: a "time-to-live" for each RPC (+ possibility to request for a further time slice)

- New problems: grandorphans, reserved locks, entries in remote queues, ….

Reliable Group Communication

- Lower-level data communication support
  - unreliable multicast (LAN)
  - reliable point-to-point channels
  - unreliable point-to-point channels
- Group communication
  - individual point-to-point message passing
  - implemented in middleware or in application
- Reliability
  - acks: lost messages, lost members
  - communication consistency ?

Reliability of Group Communication?

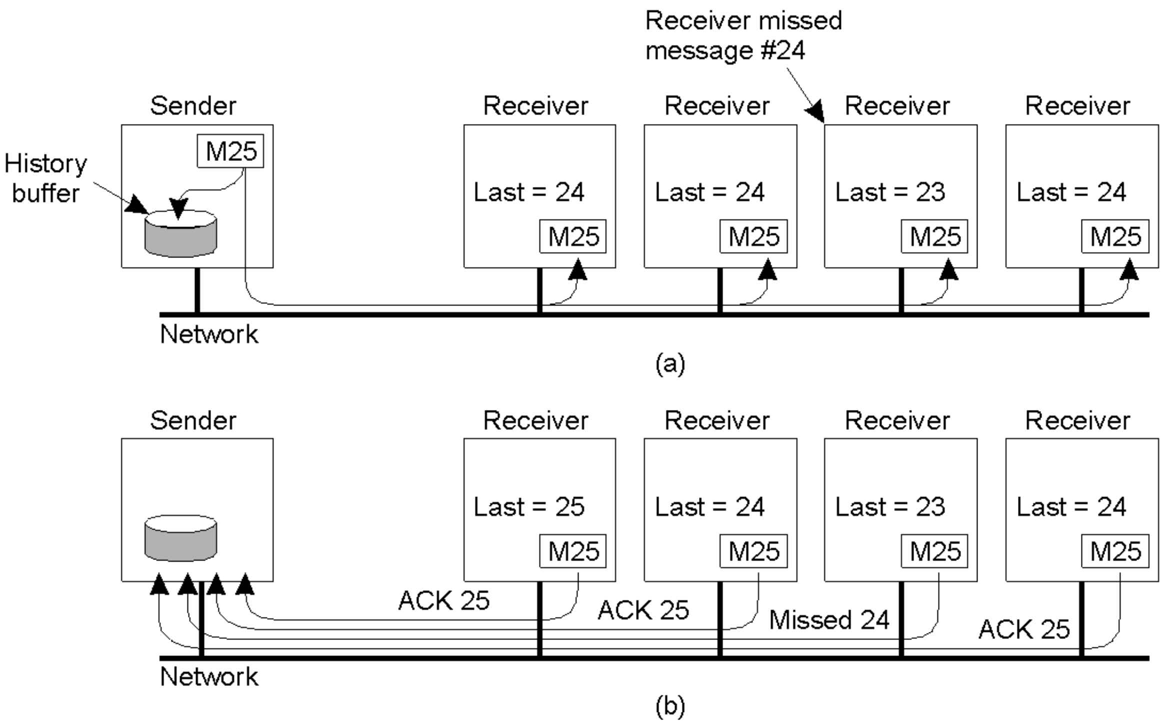
- A sent message is received by all members

  (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 an MC operation no membership changes

    An additional problem: the sender disappears (remember: multicast ~ for (all P_i in G) {send m to P_i})

Basic Reliable-Multicasting Scheme



Message transmission

Reporting feedback

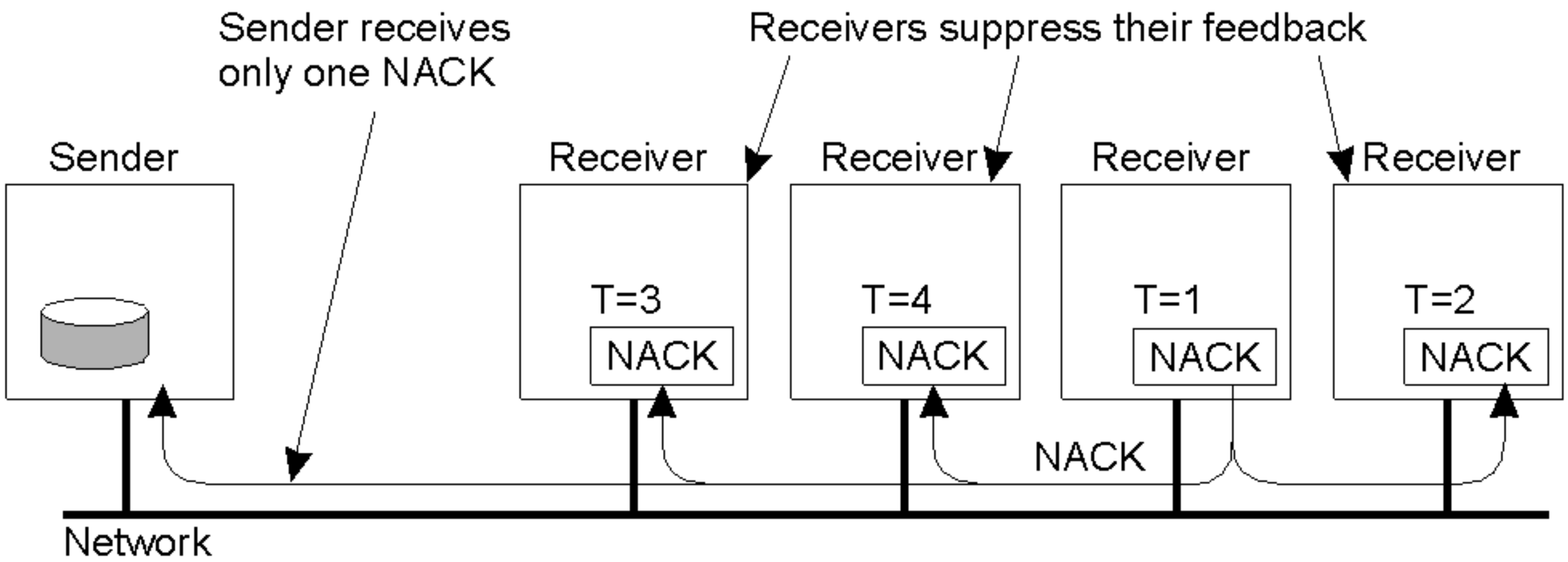
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail

Scalability?

Feedback implosion !

Scalability: Feedback Suppression

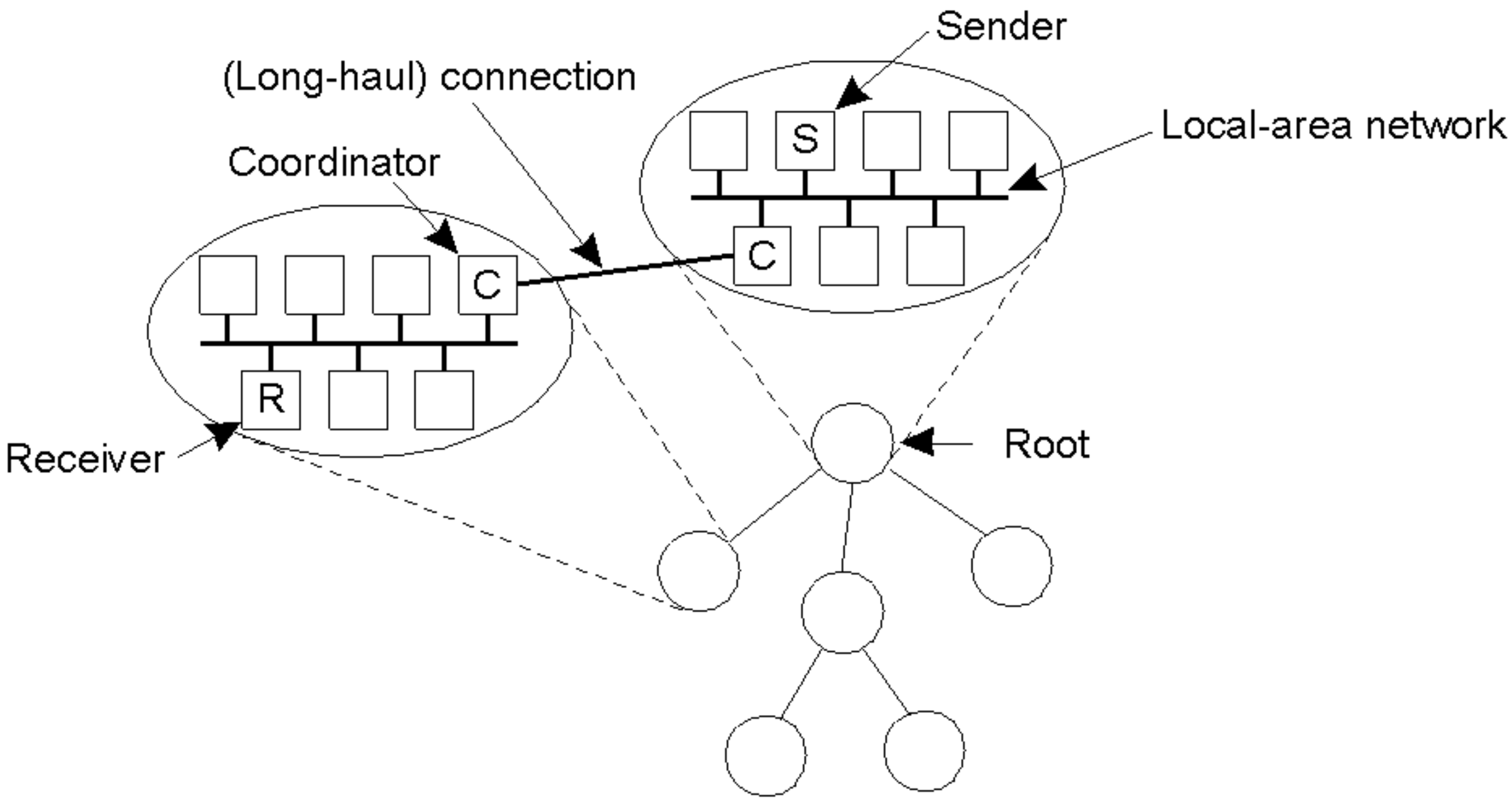
1. Never acknowledge successful delivery.



2. Multicast negative acknowledgements – suppress redundant NACKs

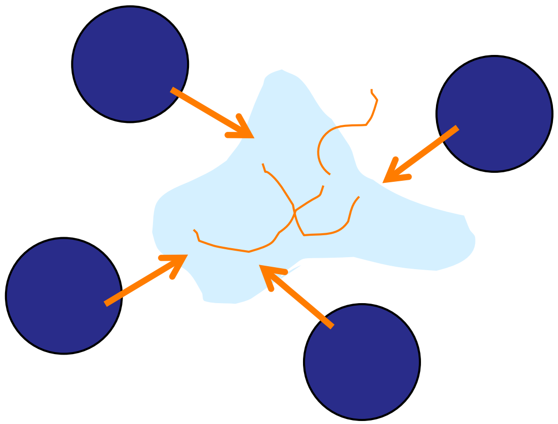
   Problem: detection of lost messages and lost group members

Hierarchical Feedback Control



The essence of hierarchical reliable multicasting.
a) Each local coordinator forwards the message to its children.
b) A local coordinator handles retransmission requests.

Basic Multicast



Guarantee:

the message will eventually be delivered to all member of the group (during the multicast: a fixed membership)

Group view: G = $\{p_i\}$
"delivery list"

Implementation of *Basic_multicast(G, m)* :

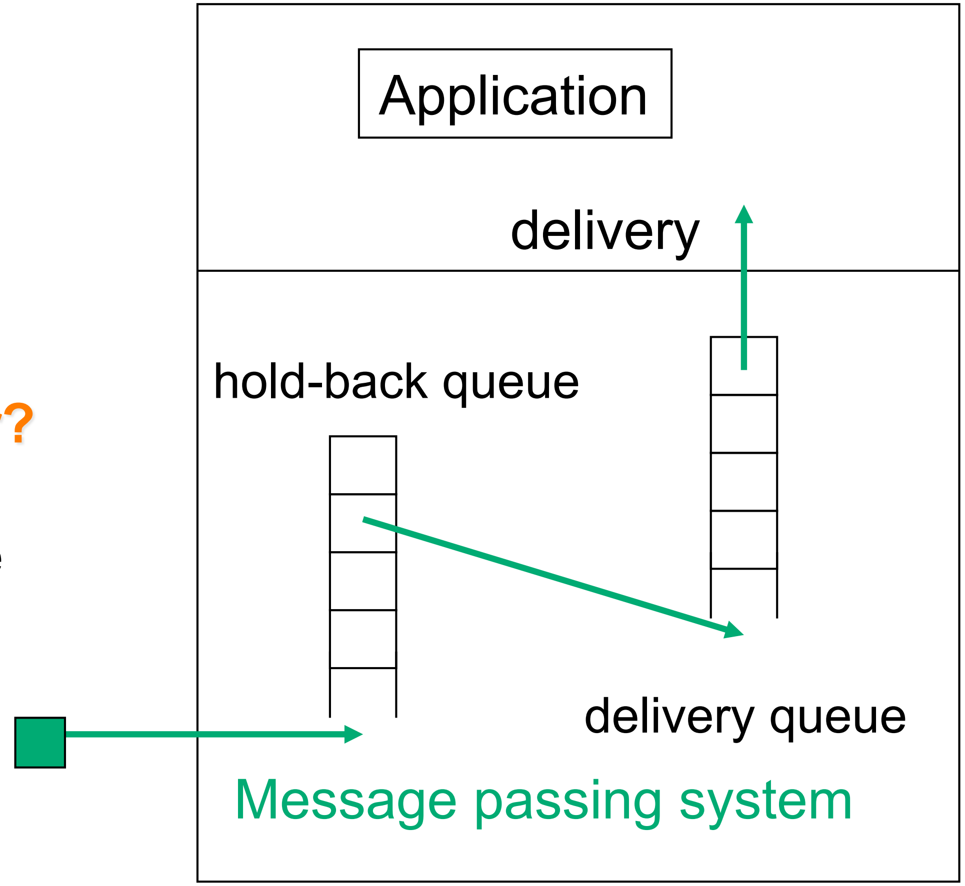
1. for each p_i in G: *send*(p_i,*m*) (a reliable one-to-one send)
2. on *receive(m)* at p_i : *deliver(m)* at p_i

Message Delivery

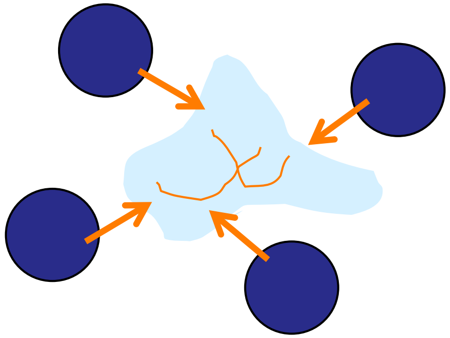
Delivery of messages

- 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* …)



Reliable Multicast and Group Changes



Assume

- reliable point-to-point communication
- group G={p_i}: each p_i : groupview

Reliable_multicast (G, m):

if a message is delivered to one in G,

then it is delivered to all in G

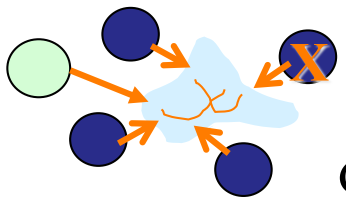
- Group change (join, leave) => change of groupview
- Change of group view: update as a multicast *vc*
- ***Concurrent group_change and multicast***

=> concurrent messages ***m*** and ***vc***

Virtual synchrony:

all *nonfaulty processes see* ***m*** *and* ***vc*** *in the same order*

Virtually Synchronous Reliable MC (1)



Group change: $G_i = G_{i+1}$

Virtual synchrony: “all” processes see m and vc in the same order

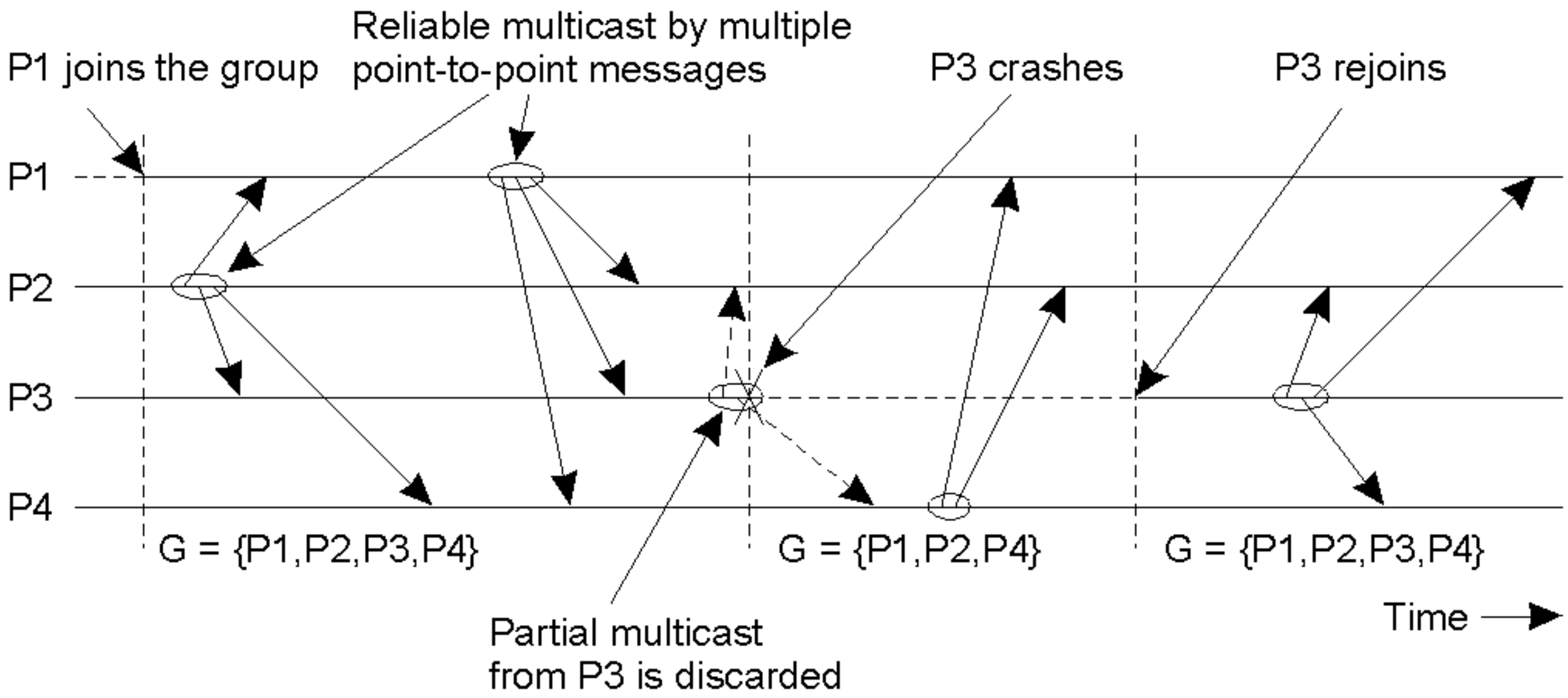
- m, vc => m is delivered to **all nonfaulty** processes in G_i (alternative: this order is not allowed!)
- vc, m => m is delivered to all processes in G_{i+1} *(what is the difference?)*

Problem: the sender fails *(during the multicast – why is it a problem?)*

Alternative solutions:

- m is delivered to all other members of G_i (=> ordering m, vc)
- m is ignored by all other members of G_i (=> ordering vc, m)

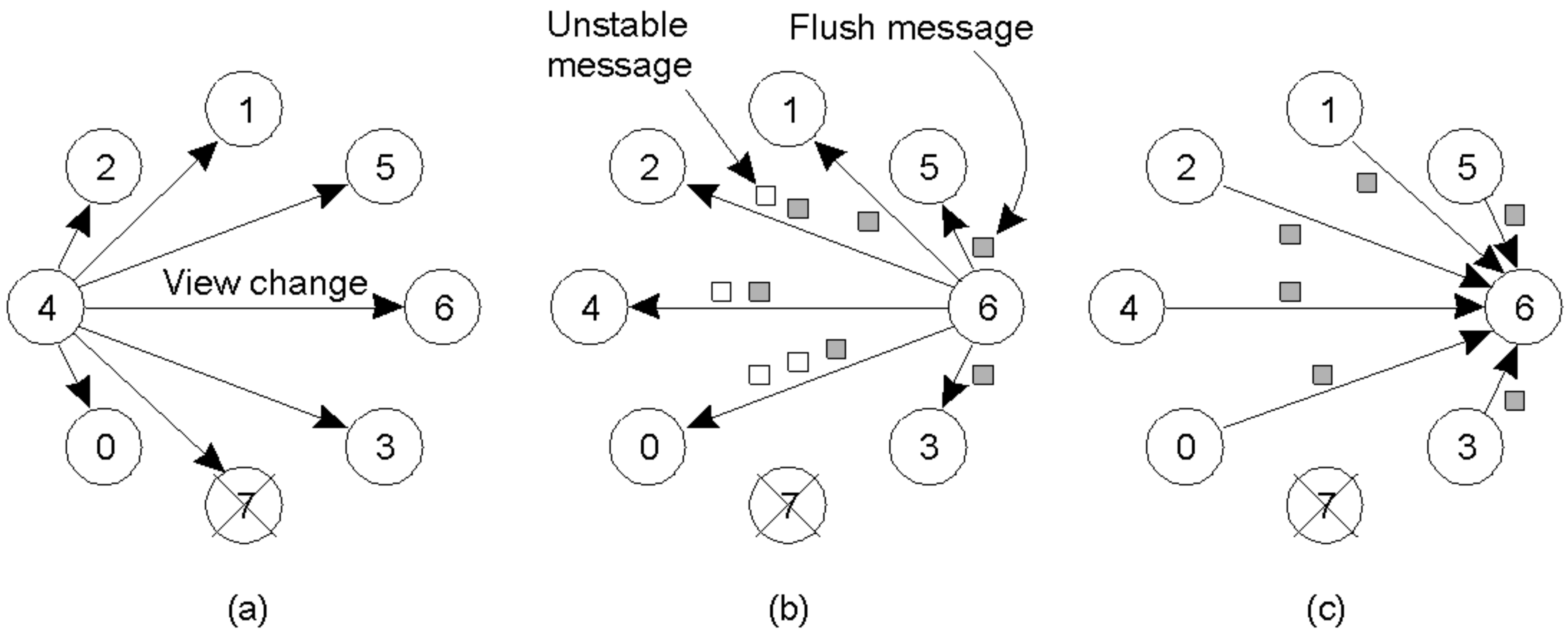
Virtually Synchronous Reliable MC (2)



The principle of virtual synchronous multicast:

- a reliable multicast, and if the sender crashes
- the message may be delivered to all or ignored by each

Implementing Virtual Synchrony (1)



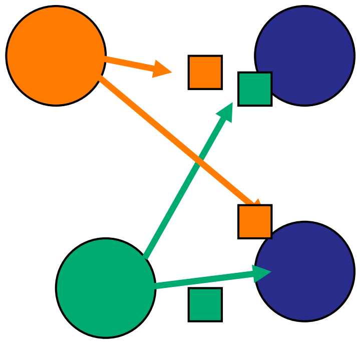
b) Process 6 sends out all its unstable messages, followed by a flush message

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

Implementing Virtual Synchrony (2)

- Communication: reliable, order-preserving, point-to-point
- Requirement: all messages are delivered to all nonfaulty processes in G
- Solution
  - each p_j in G keeps a message in the hold-back queue until it knows that all p_j in G have received it
  - a message received by all is called **stable**
  - only stable messages are allowed to be delivered
  - view change G_i => G_{i+1} :
    - multicast **all unstable messages** to all p_j in G_{i+1}
    - multicast a **flush message** to all p_j in G_{i+1}
    - after having received a flush message from all: install the new view G_{i+1}

Ordered Multicast



Need:
all messages are delivered in the intended order

1. If p: multicast(G,m) and if (for any m')
   - for **FIFO** multicast(G, m) **<** multicast(G, m')
   - for **causal** multicast(G, m) **->** multicast(G, m')
   - for **total** if at any q: deliver(m) < deliver(m')
2. then for all q in G : deliver(m) < deliver(m')

Reliable FIFO-Ordered Multicast

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

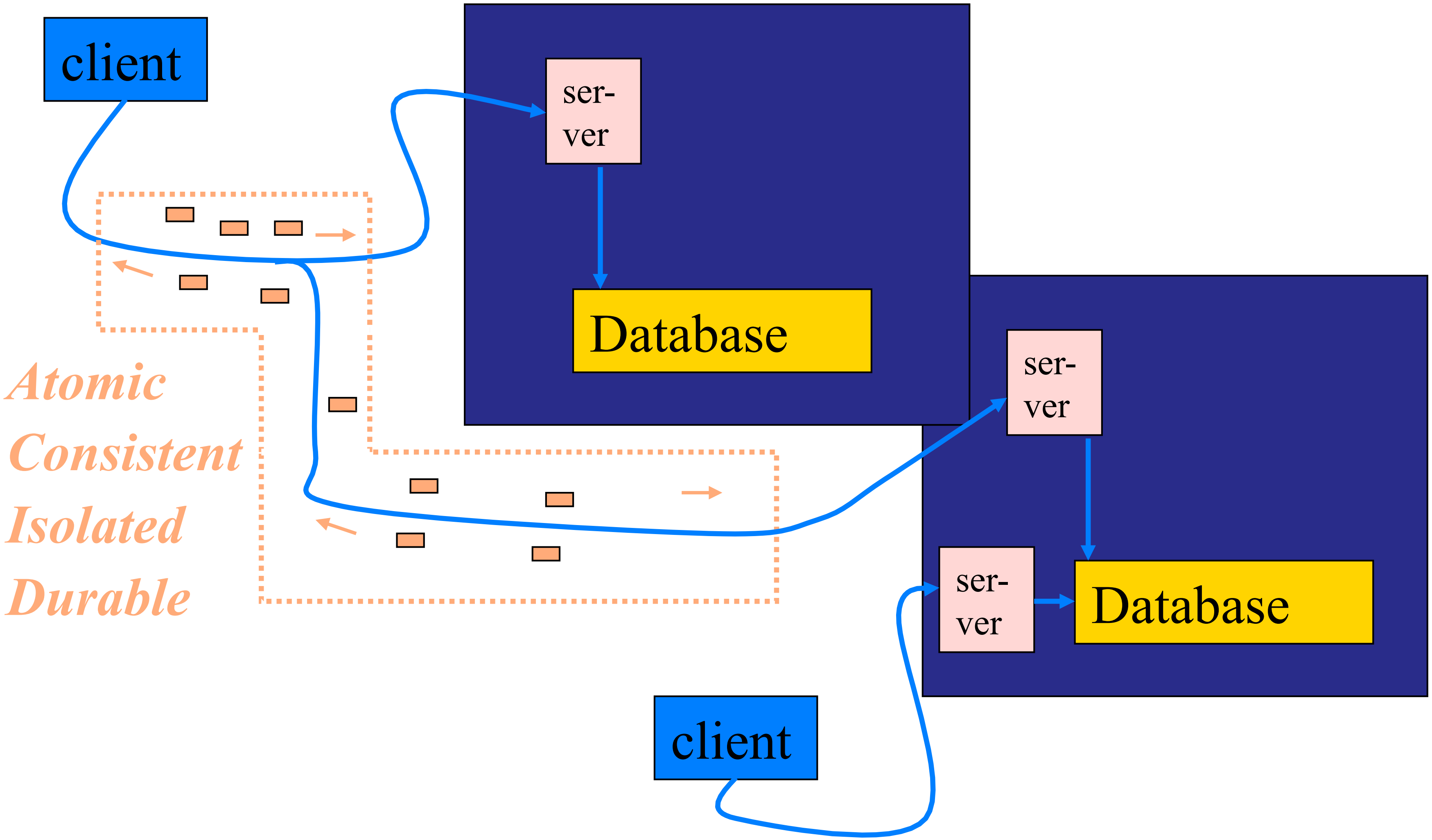
Virtually Synchronous Multicasting

Virtually synchronous 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

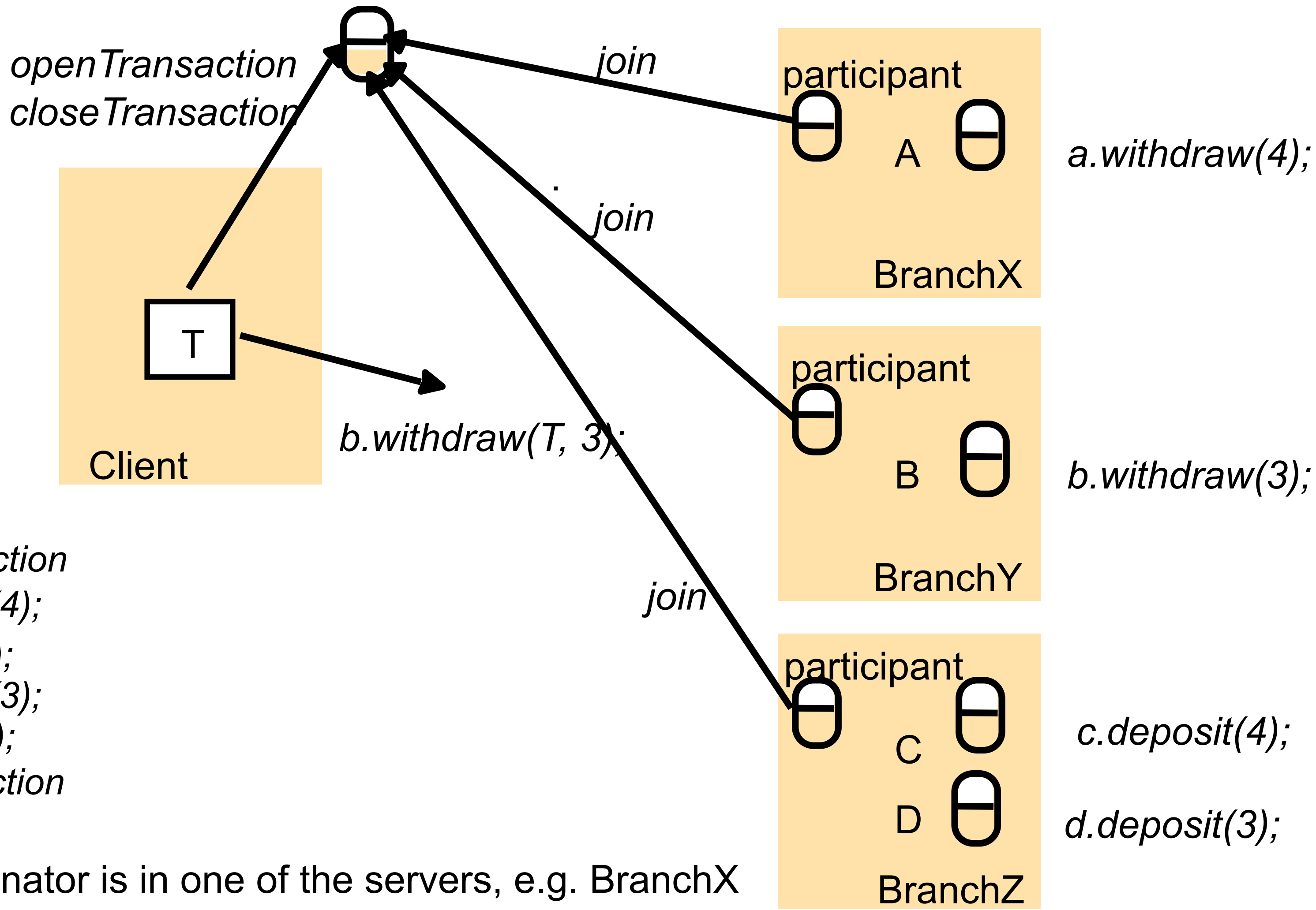
Six different versions of virtually synchronous reliable multicasting

- virtually synchronous: everybody or nobody (members of the group)
  (sender fails: either everybody else or nobody)
- **atomic multicasting**:
  virtually synchronous reliable multicasting with totally-ordered delivery.

Distributed Transactions



A distributed banking transaction

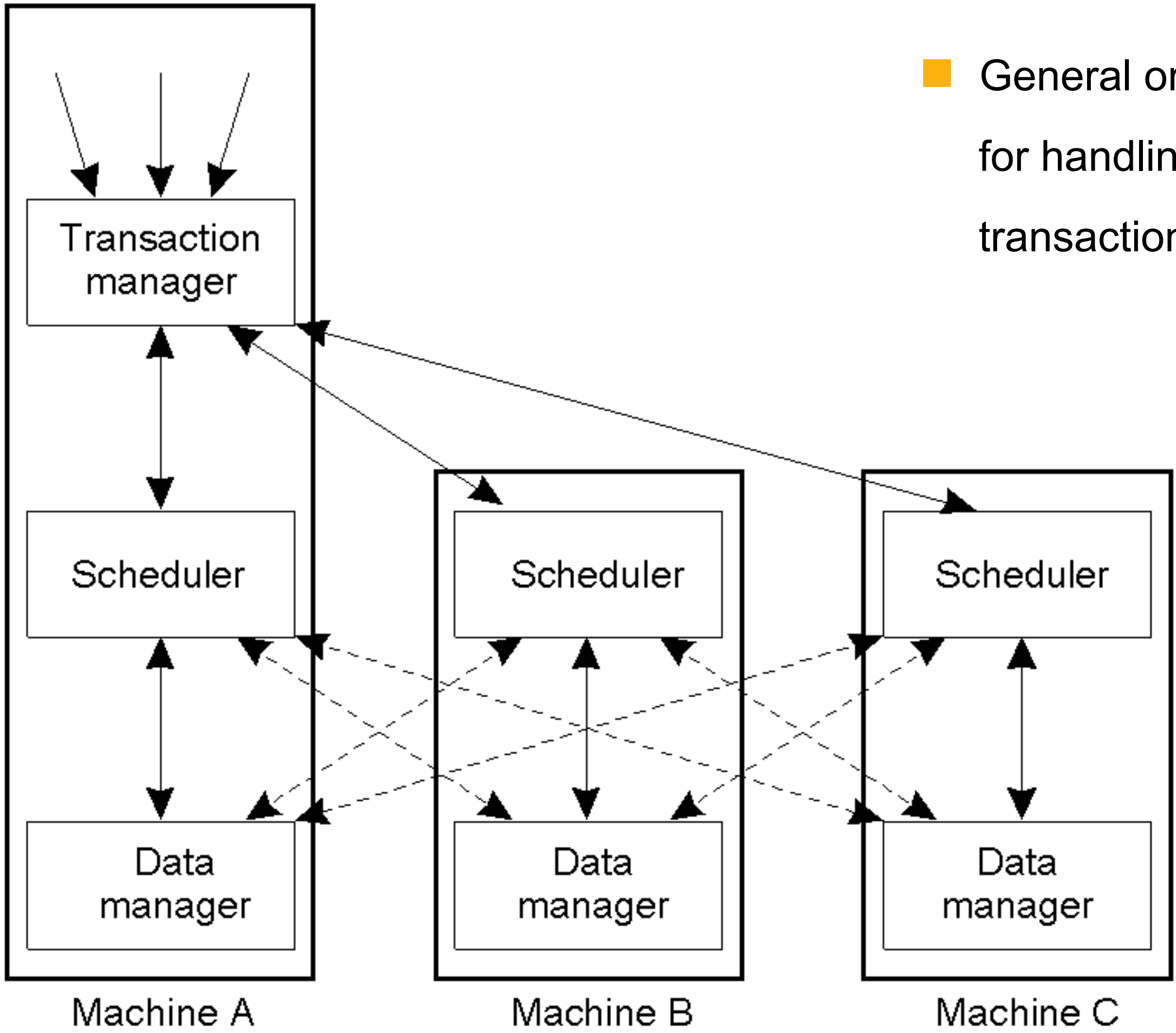


T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

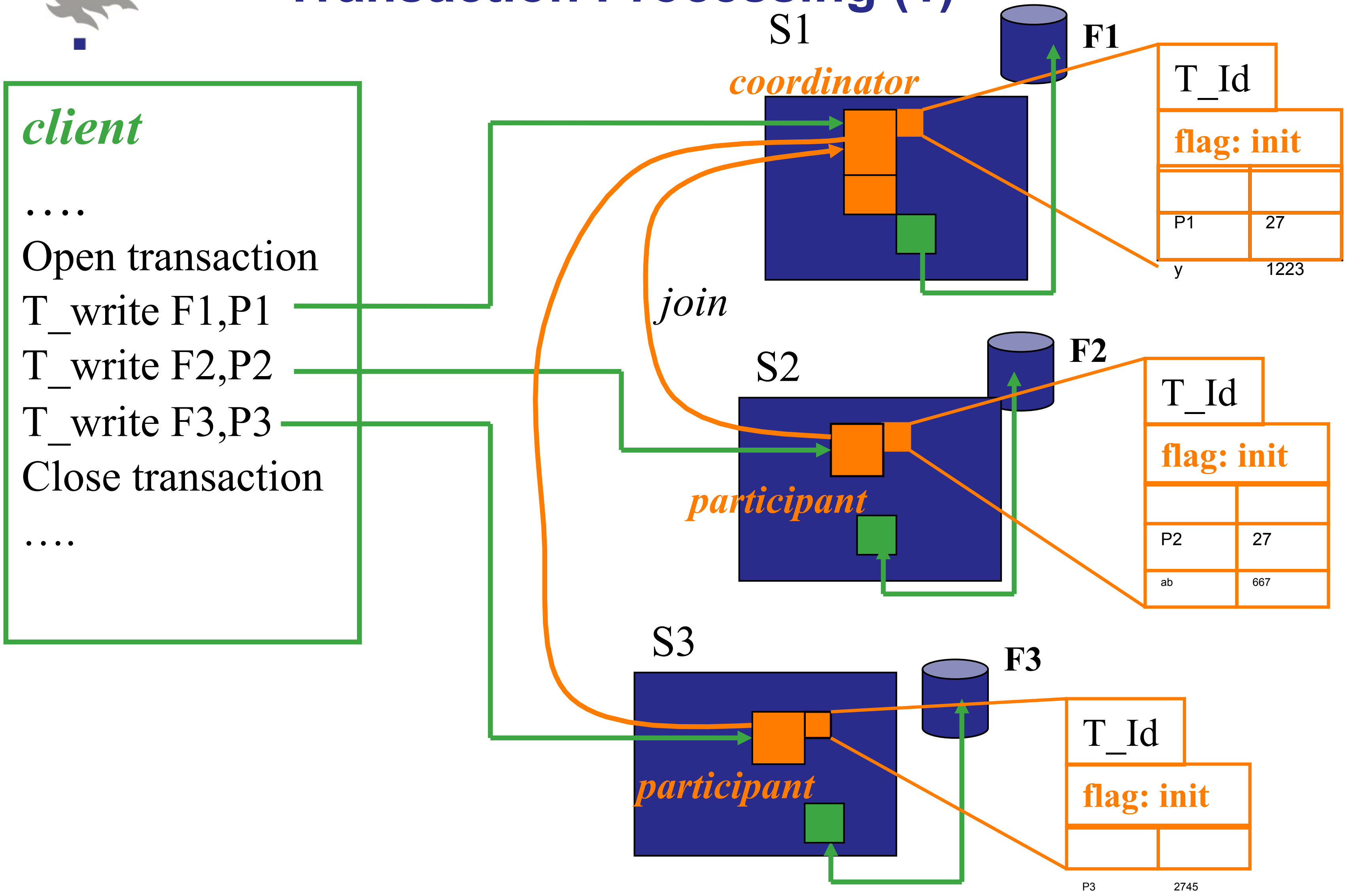
Figure 13.3

Concurrency Control



- General organization of managers for handling distributed transactions.

Transaction Processing (1)



Transaction Processing (2)

client

….

Open transaction

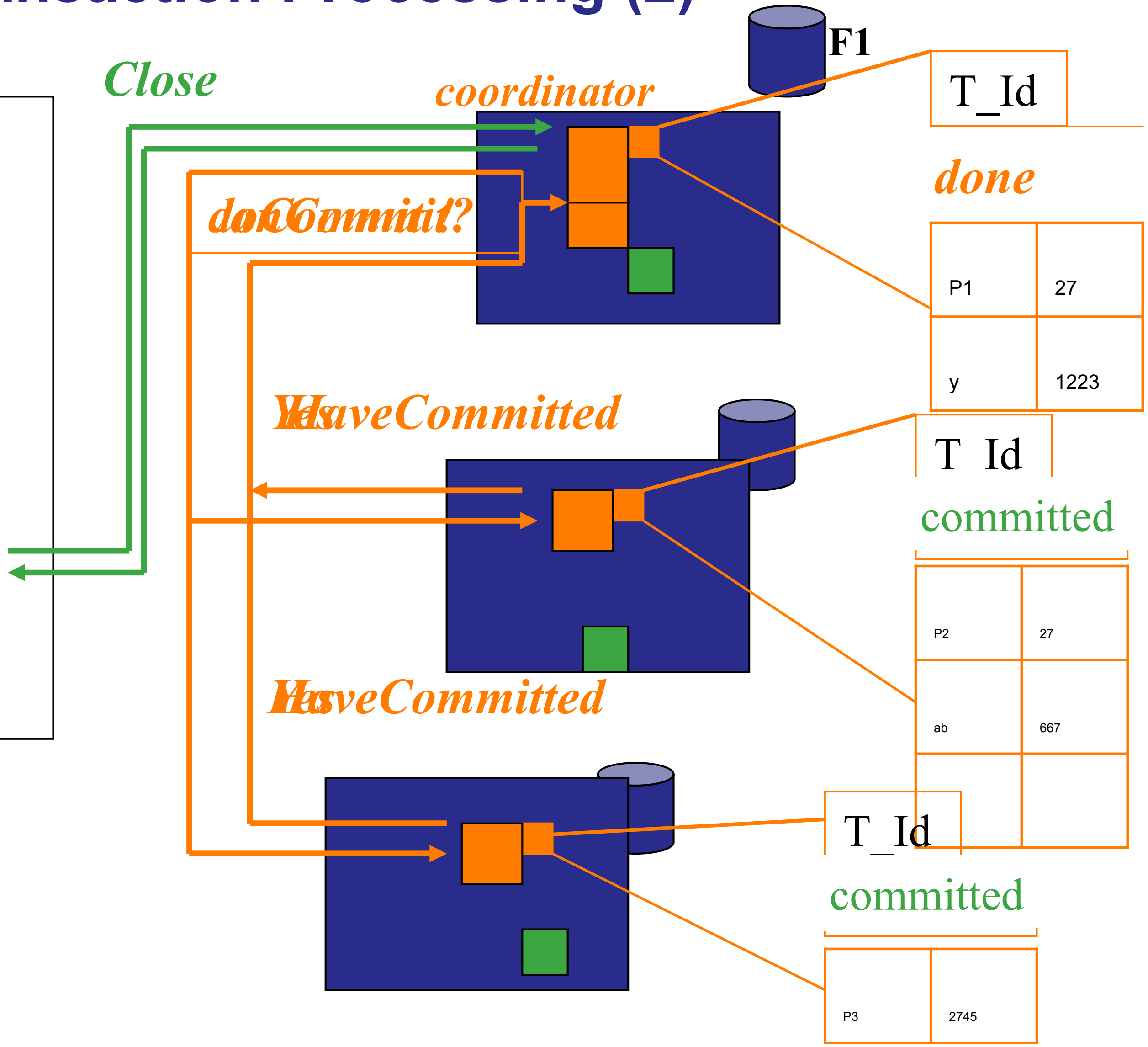
T_read F1,P1

T_write F2,P2

T_write F3,P3

Close transaction

….



Operations for Two-Phase Commit Protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant) Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

Figure 13.4

Communication in Two-phase Commit Protocol

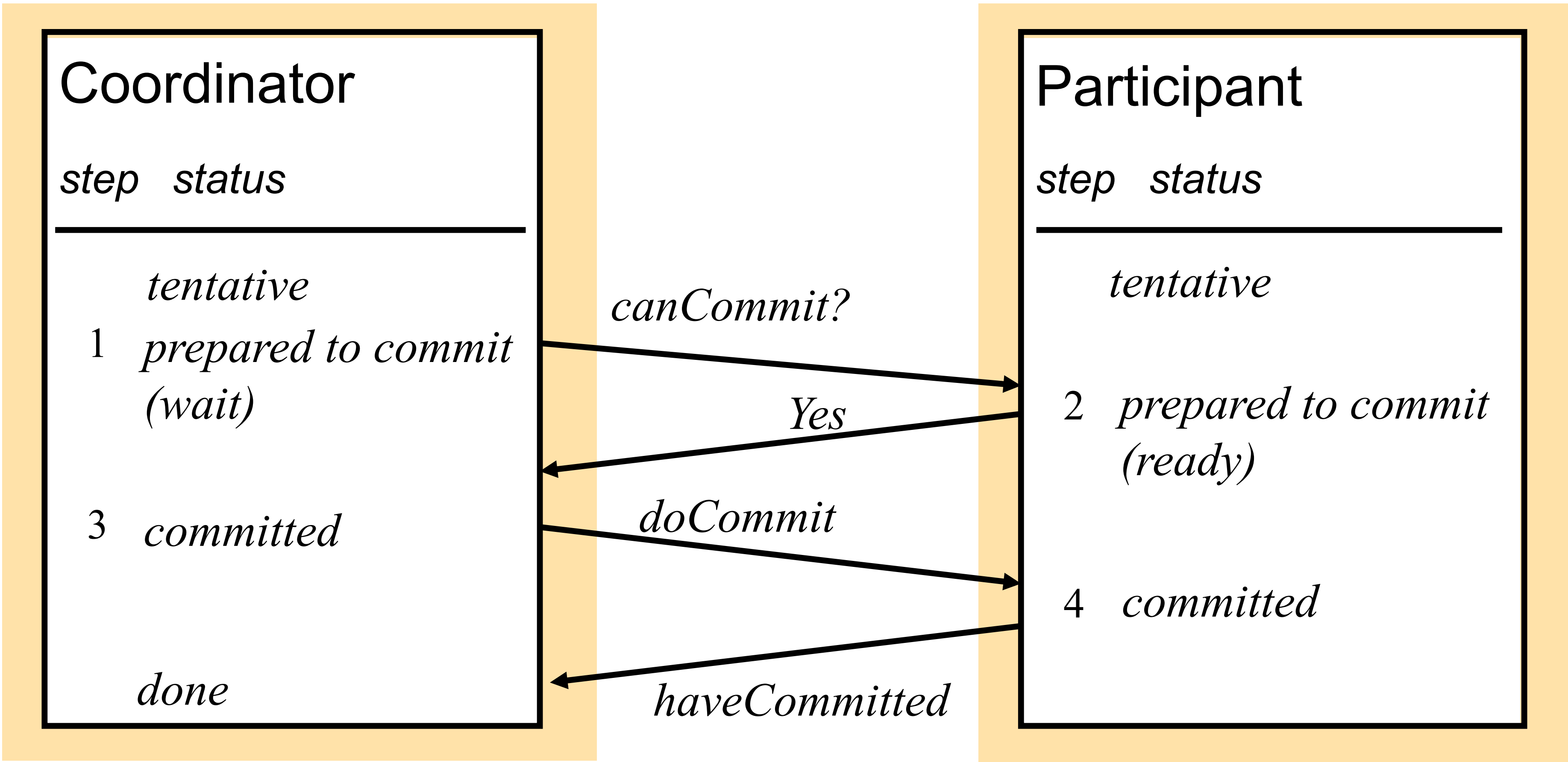


Figure 13.6

The Two-Phase Commit protocol

Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Figure 13.5

Failures

- A message is lost
- Node crash and recovery (memory contents lost, disk contents preserved)
  - transaction data structures preserved (incl. the state)
  - process states are lost
- After a crash: transaction recovery
  - tentative => abort
  - aborted => abort
  - wait *(coordinator)* => abort (resend canCommit ?)
  - ready *(participant)* => ask for a decision
  - committed => do it!

Two-Phase Commit (1)

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

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;
}
```

Two-Phase Commit (2)

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;
}
```

Steps taken by participant process in 2PC.

Two-Phase Commit (3)

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 */
```

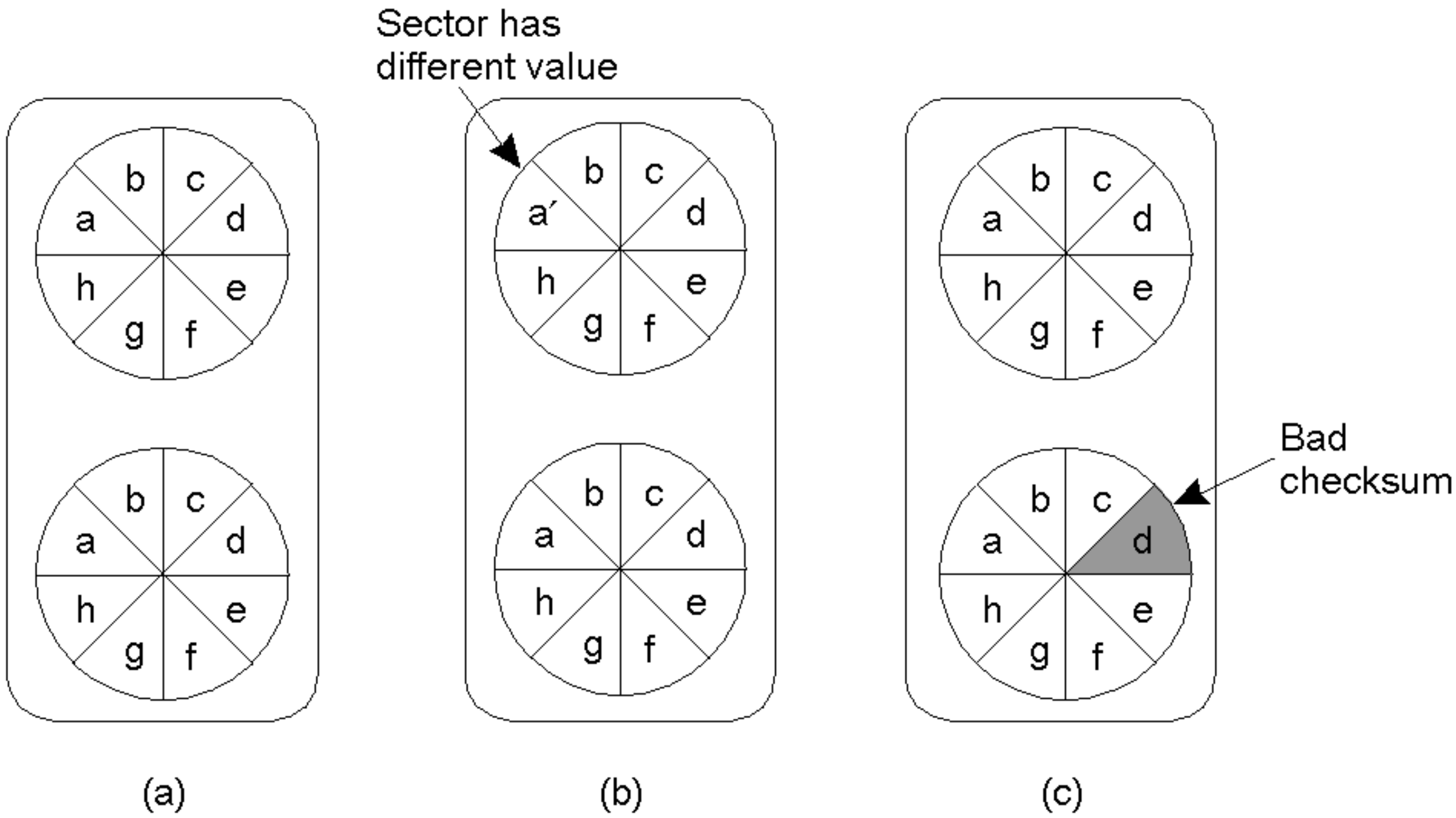
Steps taken for handling incoming decision requests.

Recovery

- Fault tolerance: recovery from an error (erroneous state => error-free state)
- Two approaches
  - backward recovery: back into a previous correct state
  - forward recovery:
    - detect that the new state is erroneous
    - bring the system in a correct new state

    challenge: the possible errors must be known in advance
  - forward: continuous need for redundancy backward:
    - expensive when needed
    - recovery after a failure is not always possible

Recovery Stable Storage



Stable Storage

Crash after drive 1 is updated

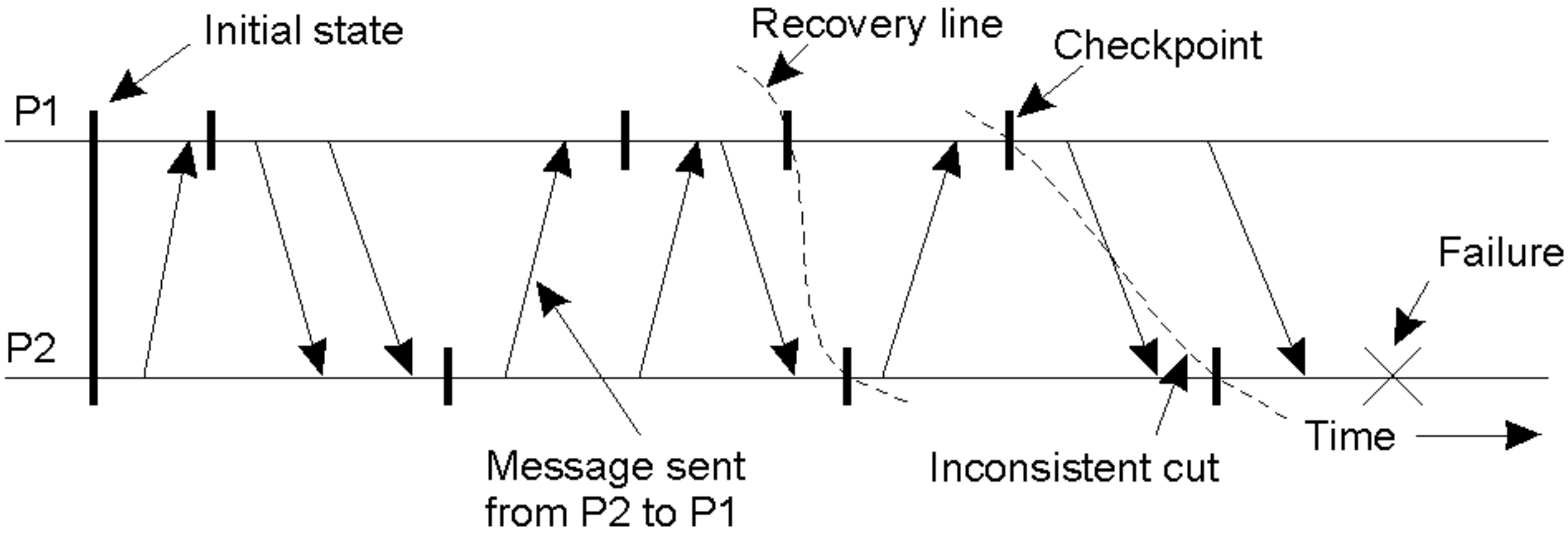
Bad spot

Implementing Stable Storage

- Careful block operations (fault tolerance: transient faults)
  - careful_read: {get_block, check_parity, error=> N retries}
  - careful_write: {write_block, get_block, compare, error=> N retries}
  - irrecoverable failure => report to the “client”
- Stable Storage operations (fault tolerance: data storage errors)
  - stable_get:
    {careful_read(replica_1), if failure then careful_read(replica_2)}
  - stable_put: {careful_write(replica_1), careful_write(replica_2)}
  - error/failure recovery: read both replicas and compare
    - both good and the same => ok
    - both good and different => replace replica_2 with replica_1
    - one good, one bad => replace the bad block with the good block

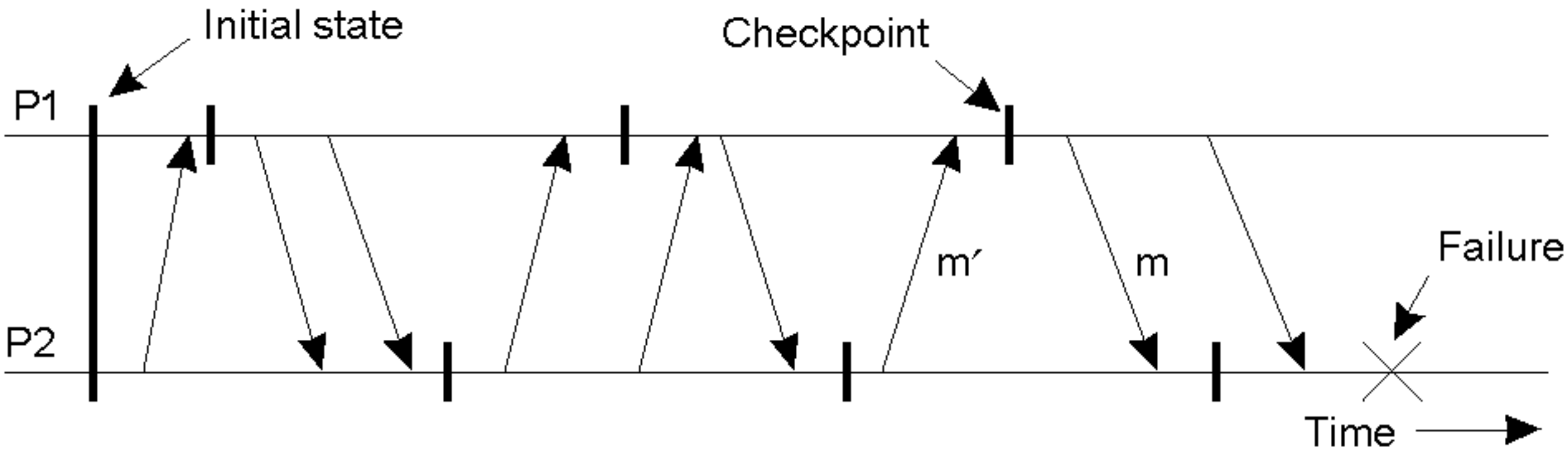
Checkpointing

Needed: a consistent global state to be used as a **recovery line**



A recovery line: the most recent distributed snapshot

Independent Checkpointing



Each process records its local state from time to time
⇒ difficult to find a recovery line

If the most recently saved states do not form a recovery line
⇒ rollback to a previous saved state (threat: the domino effect).

A solution: coordinated checkpointing

Checking of Dependencies

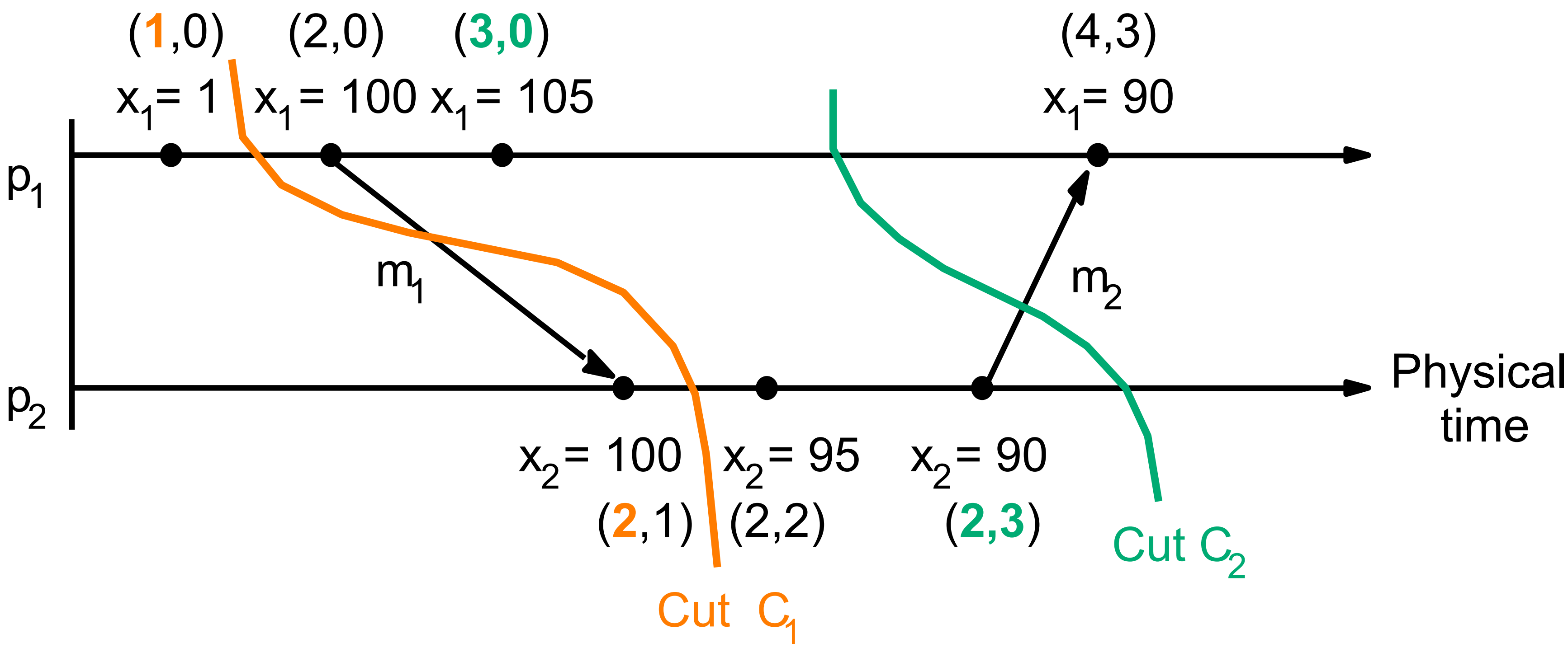
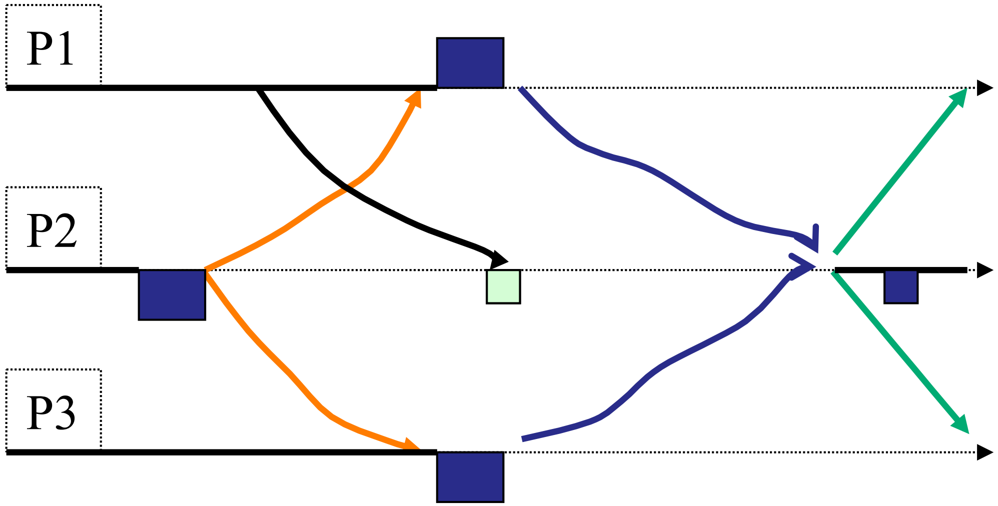


Figure 10.14 Vector timestamps and variable values

Coordinated Checkpointing (1)

- Nonblocking checkpointing
  - see: distributed snapshot (Ch. 5.3)
- Blocking checkpointing
  - coordinator: multicast CHECKPOINT_REQ
  - partner:
    - take a local checkpoint
    - acknowledge the coordinator
    - wait (and queue any subsequent messages)
  - coordinator:
    - wait for all acknowledgements
    - multicast CHECKPOINT_DONE
  - coordinator, partner: continue

Coordinated Checkpointing (2)



checkpoint request

ack

checkpoint done

local checkpoint

message

Message Logging

Improving efficiency: checkpointing and message logging

Recovery: most recent checkpoint + replay of messages

Problem: Incorrect replay of messages after recovery may lead to orphan processes.