DISTRIBUTED SYSTEMS
Principles and Paradigms
Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

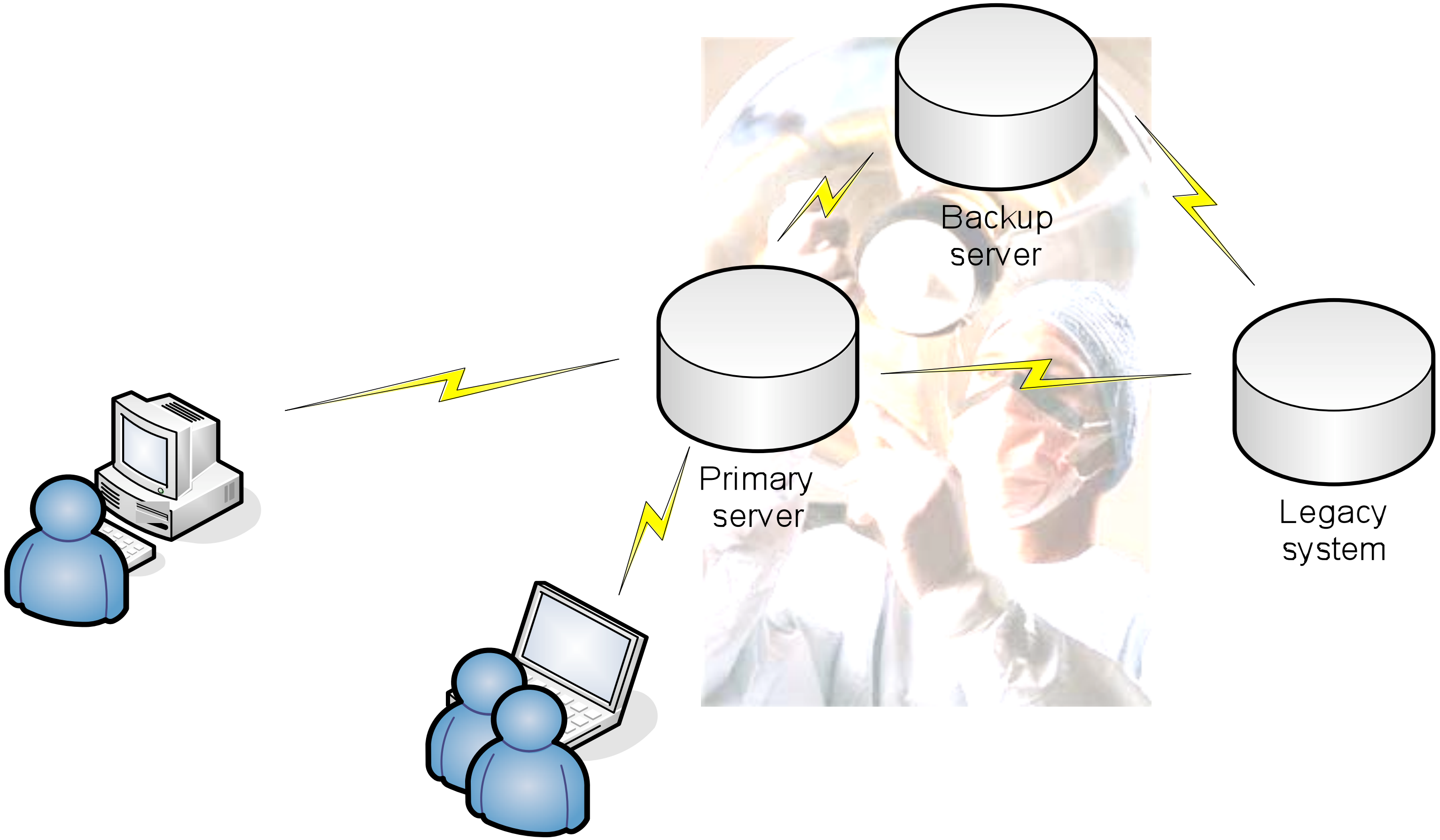
Chapter 8
Fault Tolerance (1)

Plan

- Basic Concepts
- Process Resilience
- Reliable communication
  - Client-server communication
  - Group communication
- Distributed commit
- Recovery

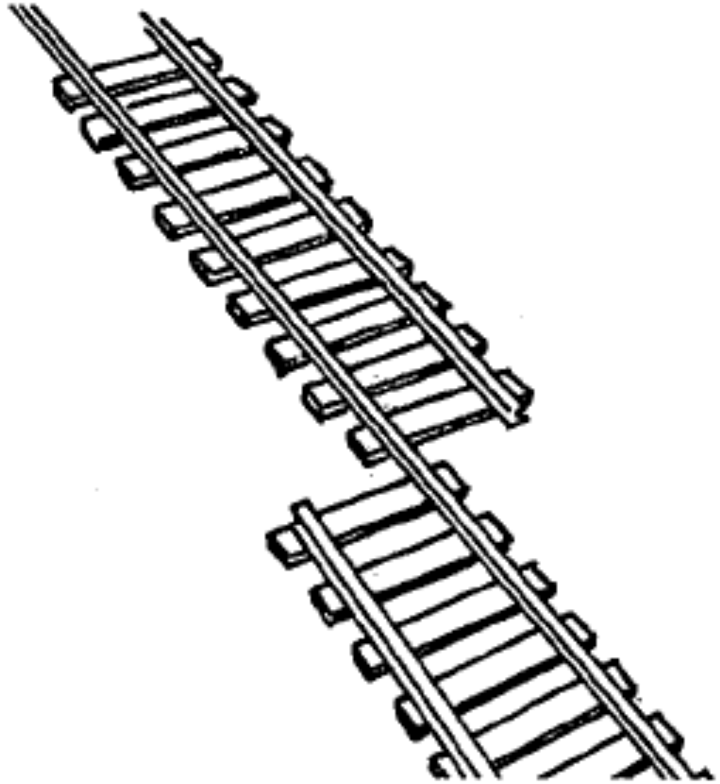
Tolerating failures

What can go wrong?



Some Basic Concepts

- IEEE Std 982
  - Failure
    - *Any deviation of the observed behavior [of a system] from the specified behavior*
  - Error
    - *System state where any further processing by the system will lead to an failure*
  - Fault (aka defect, bug)
    - *Mechanical or algorithmic cause of an error*



Example

- Ariane 5
  - June 4, 1996, 40 seconds after takeoff…
  - Self destruction after abrupt course correction
  - "… caused by the complete loss of guidance and attitude information … due to specification and design errors in the software of the inertial reference system"
  - Loss of 500 million $, but no loss of life
- Fault, error, failure in this case?

Failure Models

- Figure 8-1. Different types of failures.

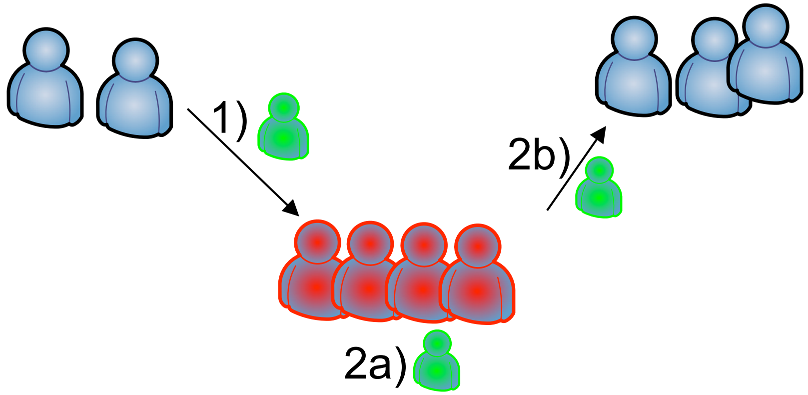
Fail-stop Fail-silent

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*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

"Byzantine" failures

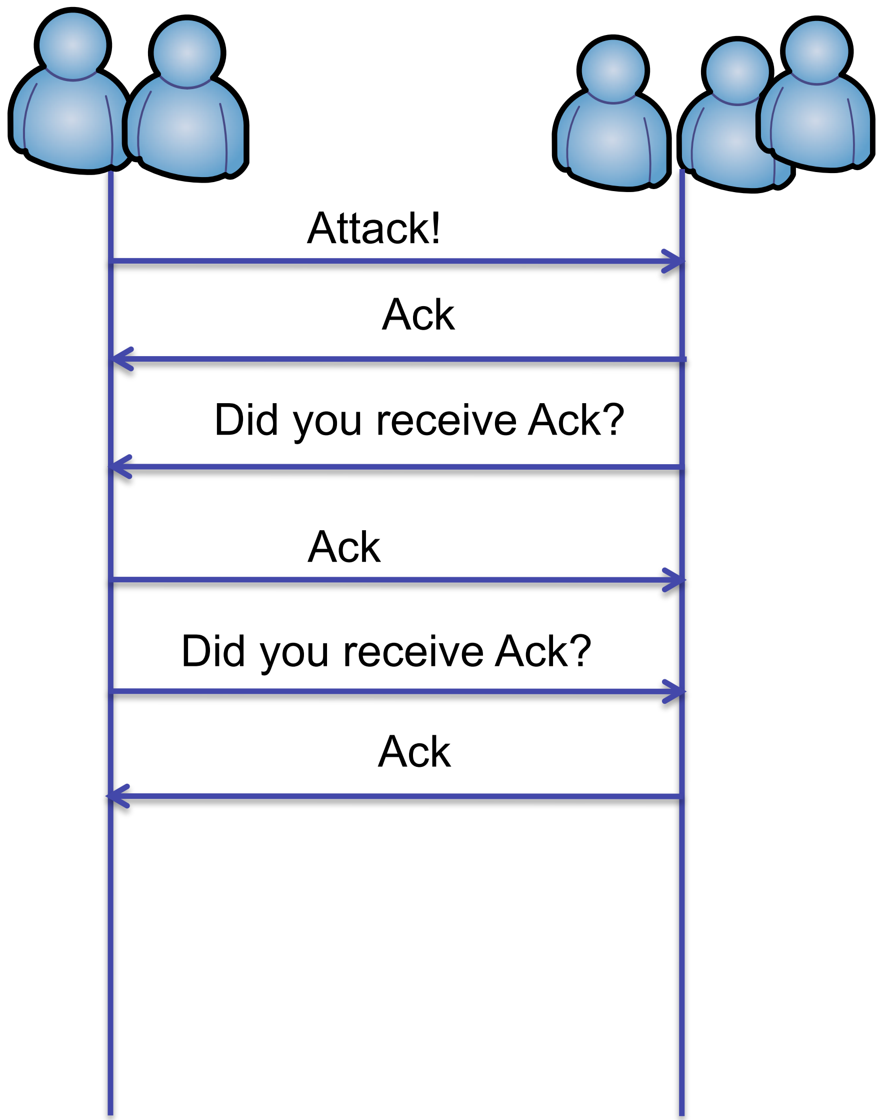
The Two-Army Problem

- Blue army needs to decide whether to attack red army
  - Blue: 2000 + 3000 soldiers
  - Red: 4000 soldiers
- If only one division of blue army attacks -> disaster
- Blue army uses messenger
  - Subject to capture by red army
- How can blue armies reach agreement on attack?
  - They cannot…



Omission failure

Two-Army Problem



Agreement/Consensus is Fundamental

- Synchronization
- Electing a coordinator
- Consistency
- War between little blue and red guys
- A prerequisite for distributed commit
  - More later…

Agreement in Faulty Systems (1)

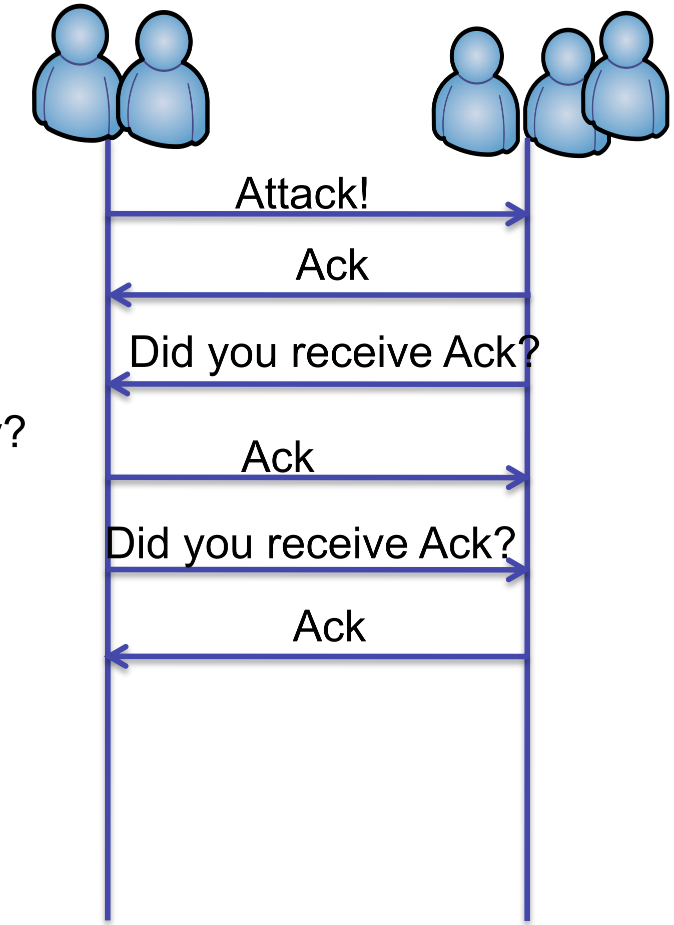
- Possible cases:
  1. Synchronous versus asynchronous systems
     - Processes are *synchronous* iff there exists a constant $s \geq 1$ so that whenever any process has taken at least s steps, all other processes have taken at least one step
  2. Communication delay is bounded or not
     - Delay is *bounded* iff all messages sent by a process arrives within r real-time steps, for some predetermined r
  3. Message delivery is ordered or not
     - Message delivery is *ordered* iff delivery of messages is ordered, when sending of messages are
  4. Message transmission is done through unicasting or multicasting.

- The Two-Army Problem?

Requirements to Agreement

- Consistency
  - All correct processes agree on the same value and that value is final
- Validity
  - The agreed-upon value was the input to one of the correct processes
- Termination
  - Each process decides on a value within a finite number of steps

Why?



Agreement in Faulty Systems

Message ordering

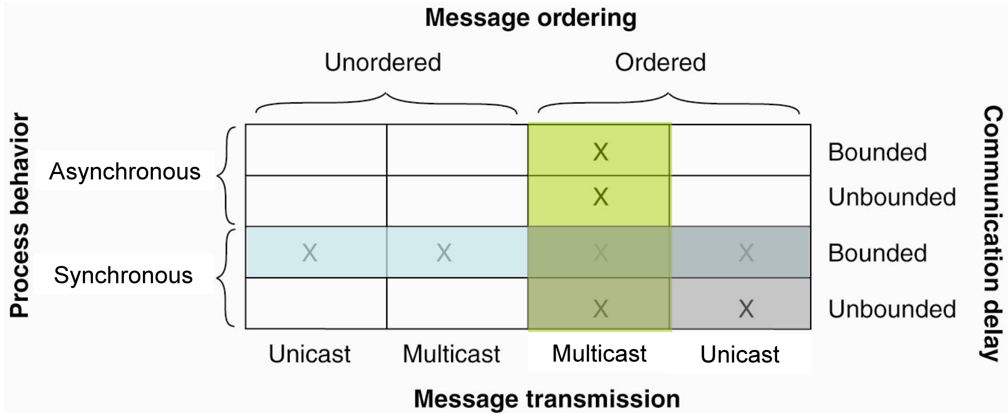
Process behavior	Communication delay	Unordered: Unicast	Unordered: Multicast	Ordered: Multicast	Ordered: Unicast
Asynchronous	Bounded			X	
Asynchronous	Unbounded			X	
Synchronous	Bounded	X	X	X	X
Synchronous	Unbounded			X	X

Message transmission

- Figure 8-4. Circumstances under which distributed agreement/ consensus can be reached. Assumes *fail-stop* failures
- *Note that Figure 8-4 is wrong in [T&S, 2007]*

Agreement in Faulty Systems

- Agreement is possible in
- Case 1
  - Processes are synchronous and communication is bounded
- Case 2
  - Messages are ordered and the transmission mechanism is multicast
- Case 3
  - Processes are synchronous and messages are ordered



Can use time-outs to see if process has failed
Basis, e.g., for three-phase commit

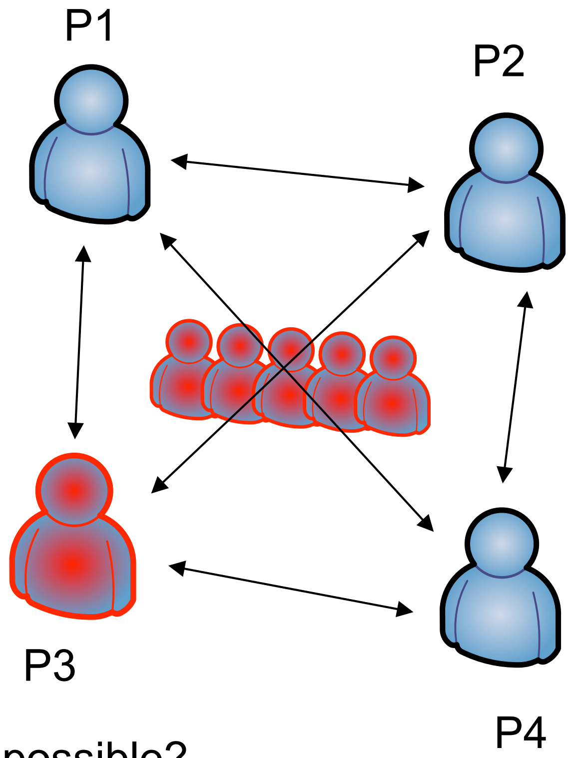
Each multicasts initial value, all non-failed processes choose first value received

Obscure algorithm with an exponential number of messages

Byzantine Generals' Problem

- A group of Byzantine generals camped around an enemy city
  - Must agree on a common battle plan
    - Attack?
    - Retreat?
  - Some of the generals may be traitors
  - Direct communication
- Model
  - Synchronous processes
  - Bounded communication
  - Unicast communication
  - *Maybe – Byzantine rather than fail-safe process failures…*

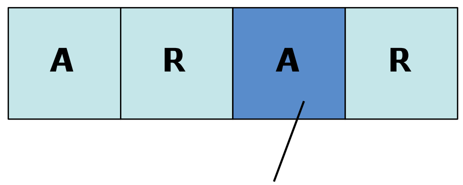
Is agreement possible?



Agreement in Faulty Systems

- Each of *n* generals decides what to do – *v(i)*
- Send *v(i)* to other generals
- Each general decides outcome based on majority of values
  - Assume biased towards attack…
- Two properties wanted
  - Loyal generals decide on the same plan of action
  - A small number of traitors cannot cause loyal generals to adopt a bad plan

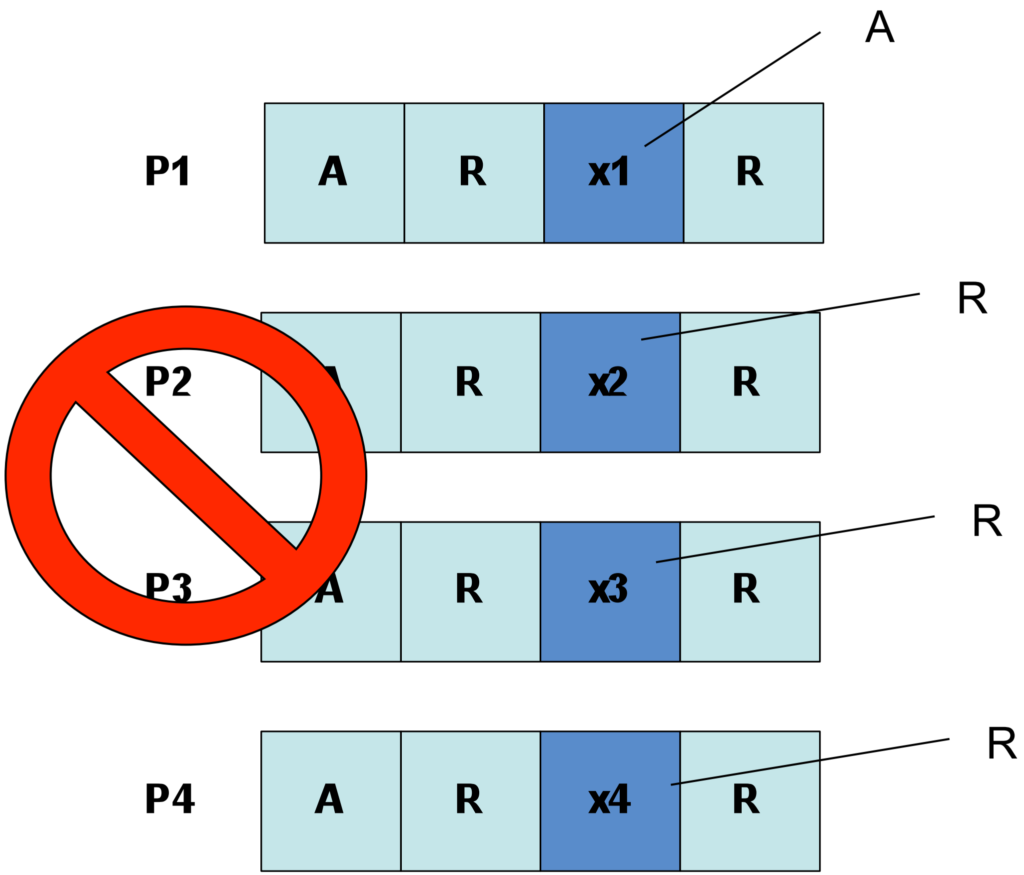
- Assume General 1 receives:



Traitor
May have sent R to others…

- What does he decide?

Trust is Bad



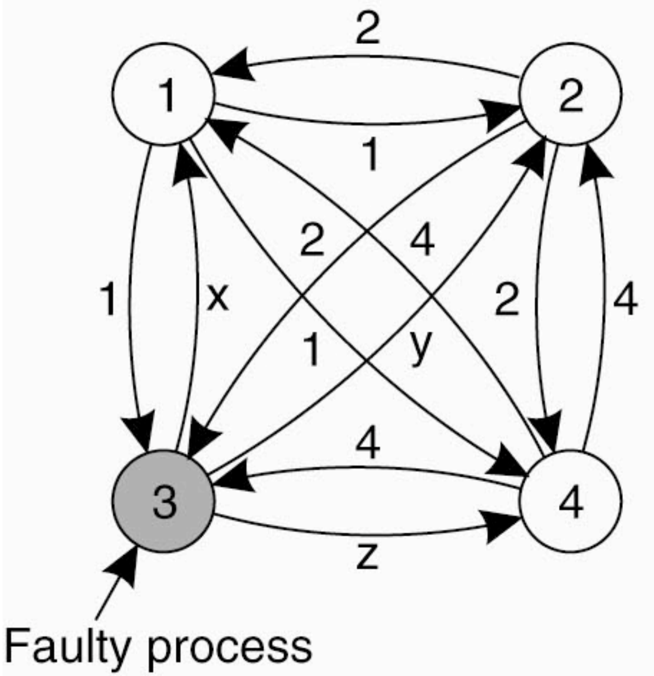
Agreement in Faulty Systems

- More precisely, we want that
  1. The value, *v(i)*, for a loyal general *i* is used by all other loyal generals
  2. Any two loyal generals use the same value of *v(i)* [even if *i* is a traitor]

Protocol in [T&S, 2007] for One Traitor

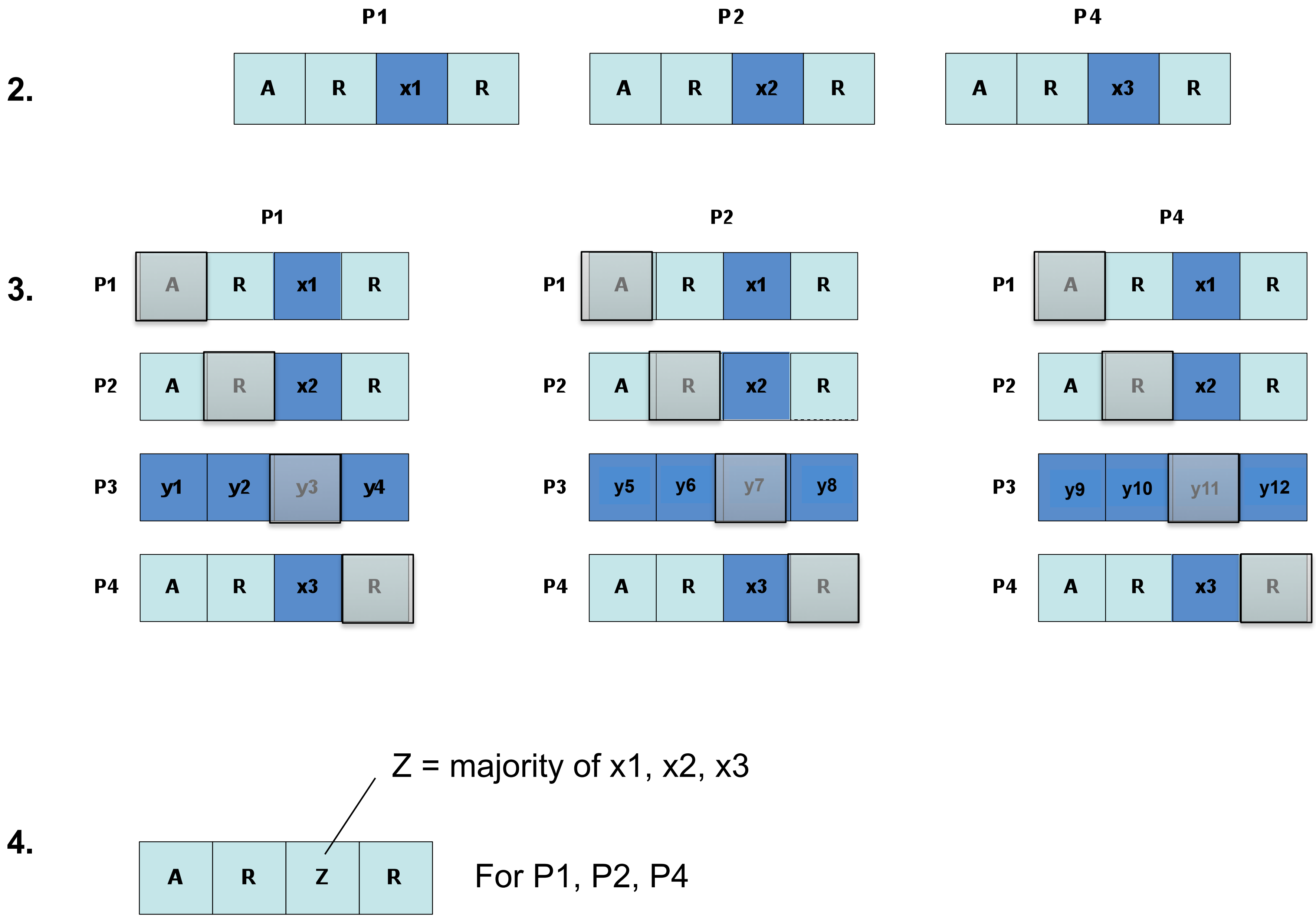
Pages 332-333 are drunken nonsense!

1. P_i unicast $v(i)$ to all
2. P_i assembles vector of received values $[v(1), \ldots, v(n)]$
3. P_i unicast $[P_i, [v(1), \ldots, v(n)]]$
4. P_i assembles result vector
   – A) For each $j \neq i$, look at j'th element of vectors not received from j
   – B) j'th element of result vector is majority of elements of A)



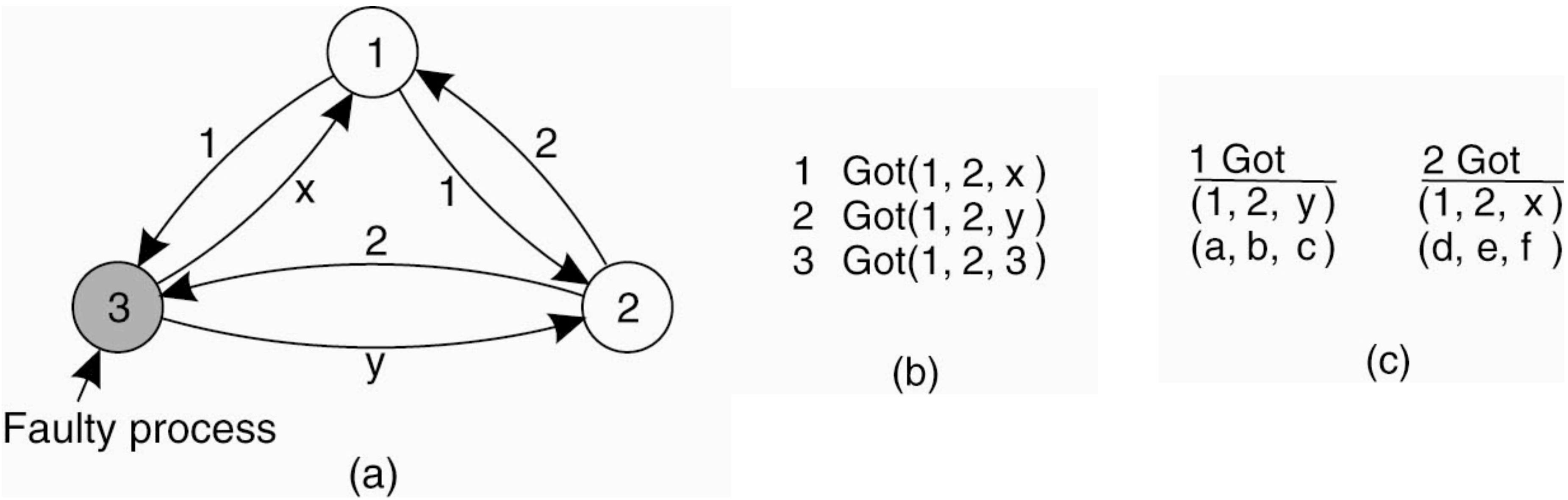
(a)

Rationale: *j* may be the traitor so don't trust him!


2.
P1
A R x1 R
P2
A R x2 R
P4
A R x3 R
3.
P1
P1 A R x1 R
P2 A R x2 R
P3 y1 y2 y3 y4
P4 A R x3 R
P2
P1 A R x1 R
P2 A R x2 R
P3 y5 y6 y7 y8
P4 A R x3 R
P4
P1 A R x1 R
P2 A R x2 R
P3 y9 y10 y11 y12
P4 A R x3 R
4.
Z = majority of x1, x2, x3
A R Z R
For P1, P2, P4

Agreement in Faulty Systems

- In general for Byzantines failures:
  - Agreement can be reached iff for *m* faulty processes, there are at least *2m + 1* non-faulty processes
- In particular Byzantine agreement is impossible for 2 correct and 1 faulty process



Reliable Client-Server Computing

- Point-to-point communication
  - E.g., using TCP
  - Masks omission failures via acknowledgements and retransmissions (using timeouts)
- RPC…

RPC Semantics in the Presence of Failures

- Five different classes of failures that can occur in RPC systems:

1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

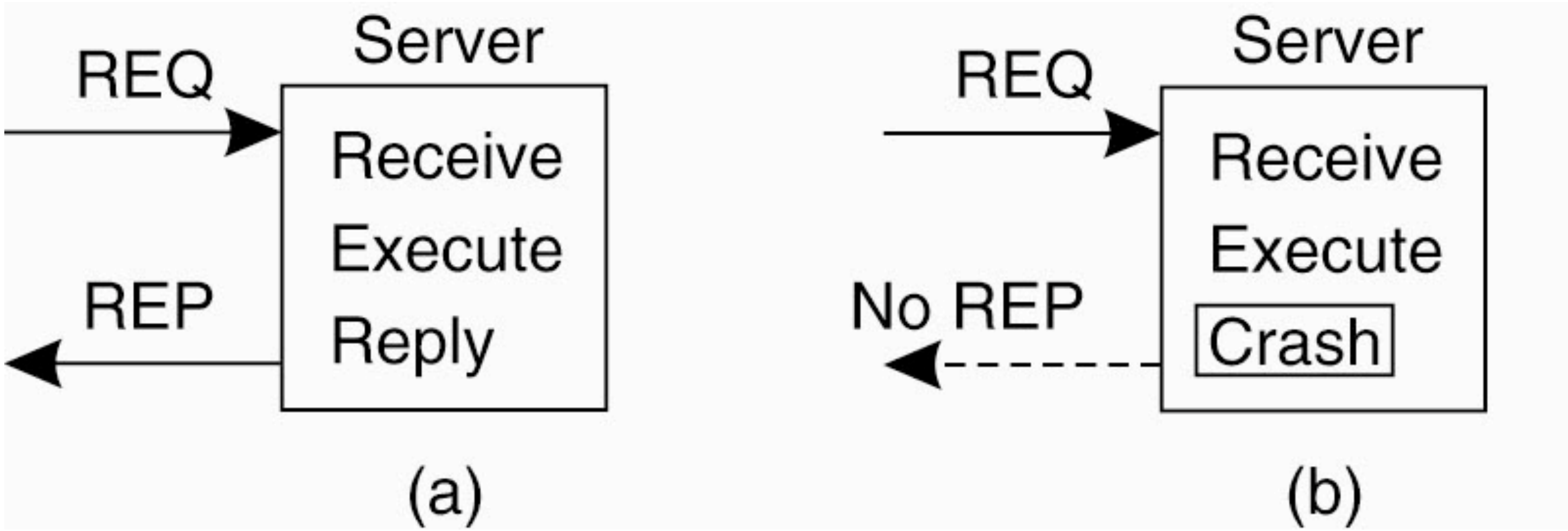
1. Client Unable to Locate Server

- Not much to do except to throw Exception…
- Java RMI?
  - Causes
    - Registry not available
    - Server not bound in registry
    - Server may have incompatible version
    - RMI wire protocol version

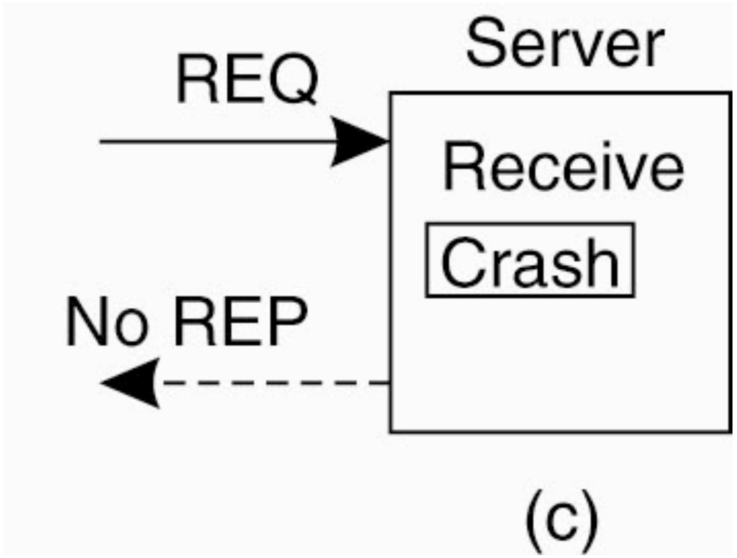
2. Request Message Lost

- Client may retransmit message after timer expires
- Java RMI?
  - Handled by using TCP (or similar)

3. Server Crashes

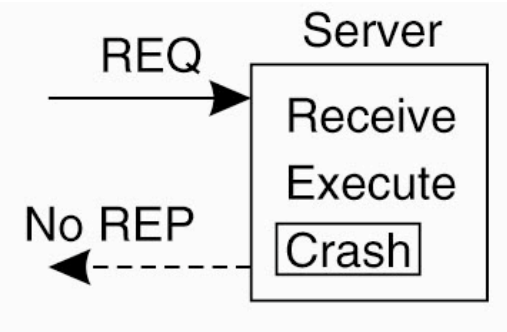


- Figure 8-7. A server in client-server communication.
  (a) The normal case.
  (b) Crash after execution.
  (c) Crash before execution.

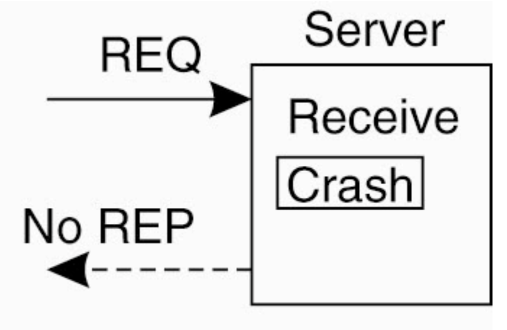


3. Server Crashes

- Client cannot distinguish causes for No REP
  - But correct behavior may differ for (b) and (c)
- Possible approaches for stub
  - At-least-once semantics
    - Client retries until it gets reply
  - At-most-once semantics
    - Client gives up and reports failure
  - Exactly-once semantics
    - Impossible…



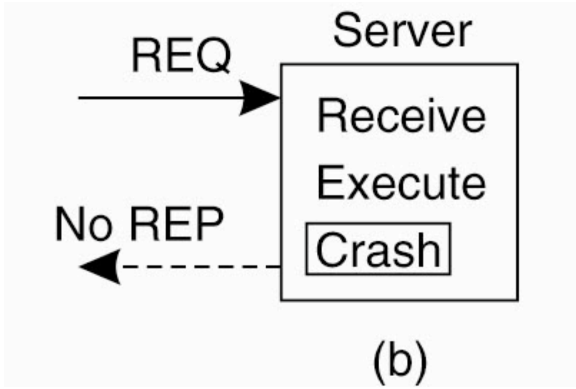
(b)

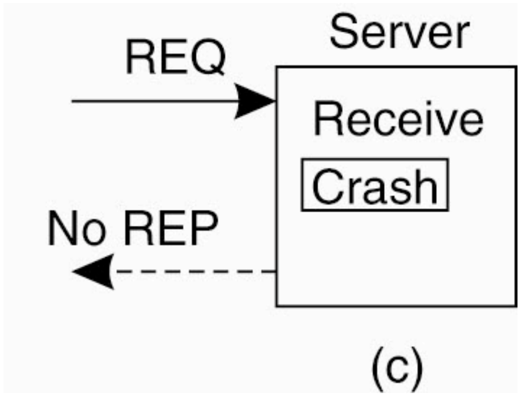


(c)

3. Server Crashes

- Java RMI again?
  - Does not really help per se when server crashes
  - At-most-once semantics partially guaranteed by using TCP





3. Server Crashes

- Exactly-once semantics again
- Example
  - Client wants to print document on server
  - Three events that can happen at example server:
    - Send the completion message (M),
    - Print the text (P),
    - Crash (C).

3. Server Crashes

- These events can occur in six different orderings:

1. M →P →C
   - A crash occurs after sending the completion message and printing the text.
2. M →C (→P)
   - A crash happens after sending the completion message, but before the text could be printed.
3. C (→P →M)
   - A crash happens before the server could do anything.
4. P →M →C
   - A crash occurs after sending the completion message and printing the text.
5. P →C (→M)
   - The text printed, after which a crash occurs before the completion message could be sent.
6. C (→M →P)
   - A crash happens before the server could do anything.

3. Server Crashes

- Assume server crashes, subsequently recovers, and notifies clients
  - What should client do?
- Client can never know whether server crashed before or after printing

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	**MPC**	**MC(P)**	**C(MP)**	**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

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

4. Lost Reply

- If at-least-once semantics desirable
  - Resend request until reply is received
  - *Idempotent* procedures needed
- Add sequence number on request
  - Check on server against sequence number
  - Need stateful server
  - How long should server keep track of sequence numbers?

Idempotent Operations

- Operations, *O*, such that *O(x)* = *O(O(x))*
  - F(x) = x * 0 is idempotent
  - G(x) = x + 1 is not
  - Print it not
  - Read “next” block of a file is not idempotent
  - Read block 3 of a file is idempotent
  - Write “next” block of a file is not idempotent
  - Write block 3 of a file is idempotent

Server

Strategy M → P			Strategy P → M		
MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
OK	OK	OK	OK	OK	OK
OK	ZERO	ZERO	OK	OK	ZERO
OK	OK	ZERO	OK	OK	ZERO
OK	ZERO	OK	OK	OK	OK

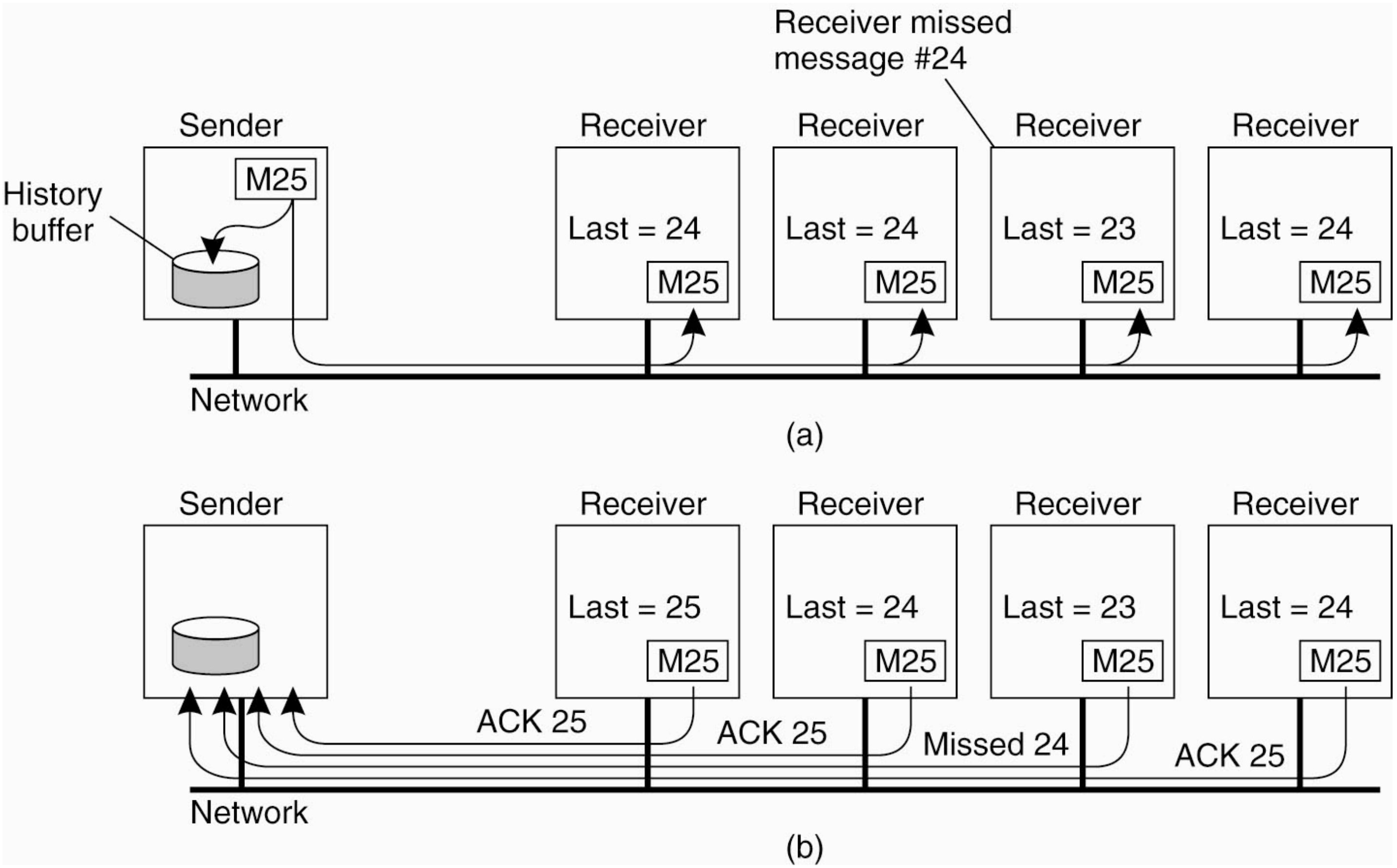
5. Client Crashes

- Can lead to *orphan* computations
  - Orphans may use up valuable resources
  - Orphans may confuse rebooting clients
- Solutions
  - Orphan extermination
    - Log RPCs to stable storage before sending them
    - Kill of orphans on reboot
  - Reincarnation
    - Client broadcasts epoch when it (re)boots
    - Orphans from previous epochs are killed
  - Gentle reincarnation
    - Only kill orphans if parent cannot be reached
  - Expiration
    - Allocate quantum of time to RPC

Reliable Group Communication

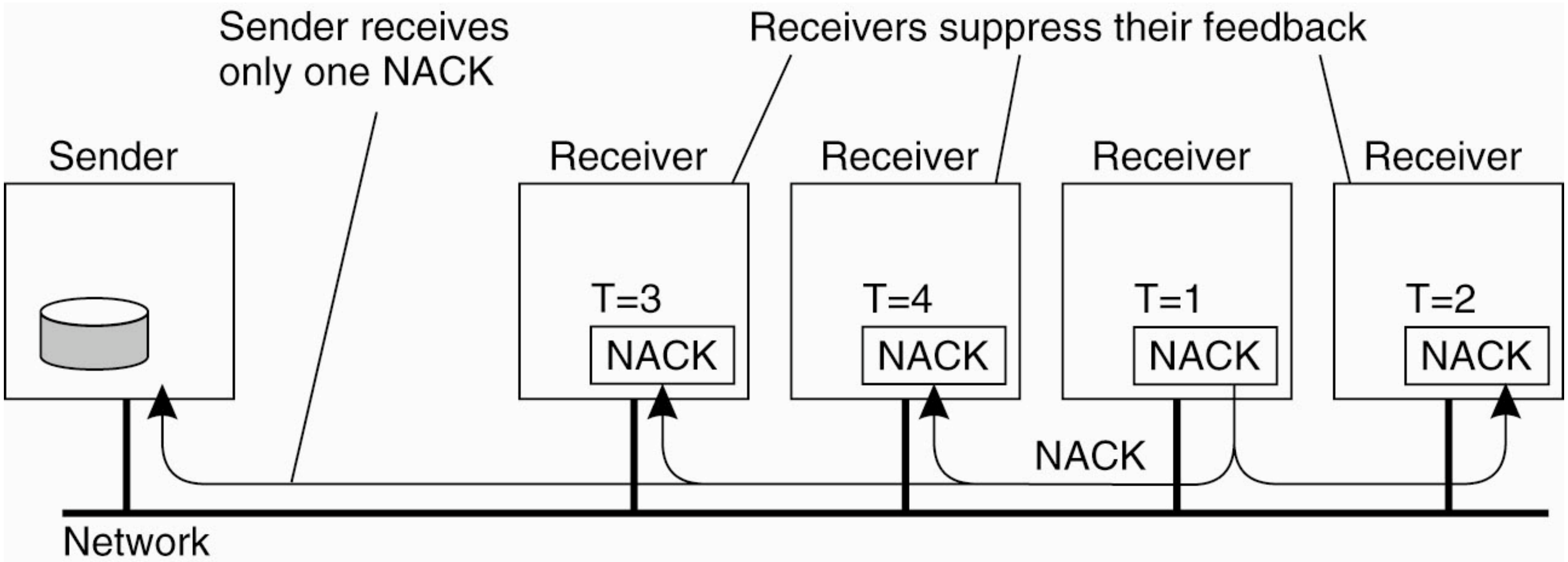
- Maintain process group
  - E.g., multicast requests to it, receive answer from at least one member
- Issues
  - "Reliable"?
  - If members join/leave while multicasting?
  - If members fail while multicasting?
- Build on top of unreliable multicasting

Basic Reliable-Multicasting Schemes



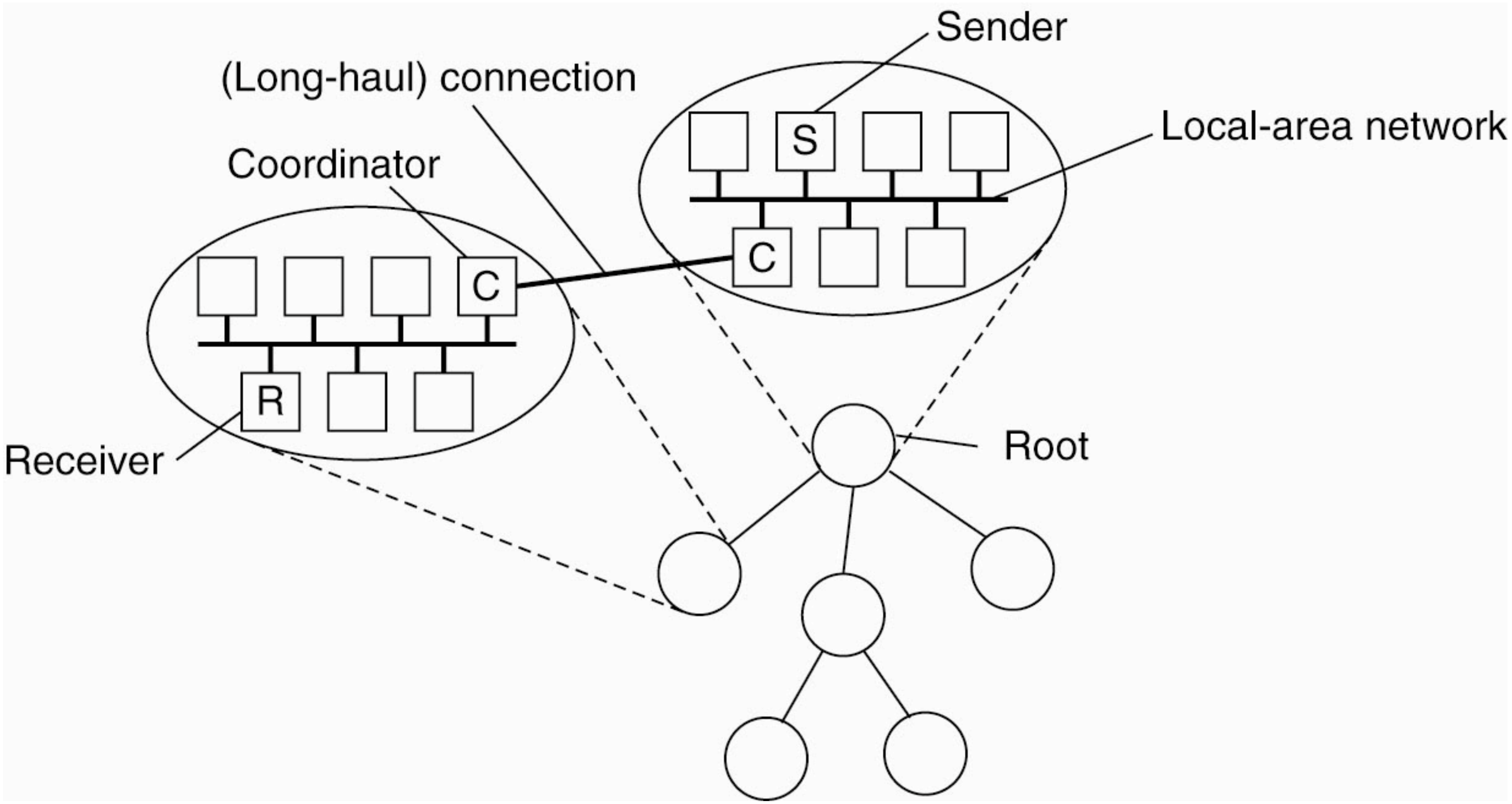
- Figure 8-9. A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.
- (a) Message transmission. (b) Reporting feedback.

Nonhierarchical Feedback Control



- Figure 8-10. Several receivers have scheduled a request for retransmission, but the first retransmission request
leads to the suppression of others.

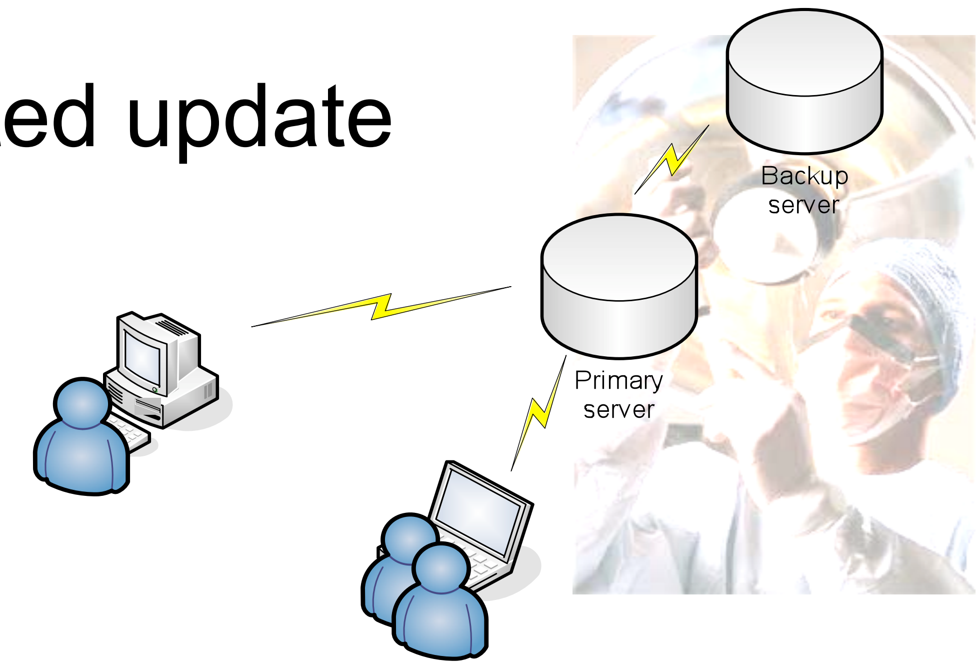
Hierarchical Feedback Control



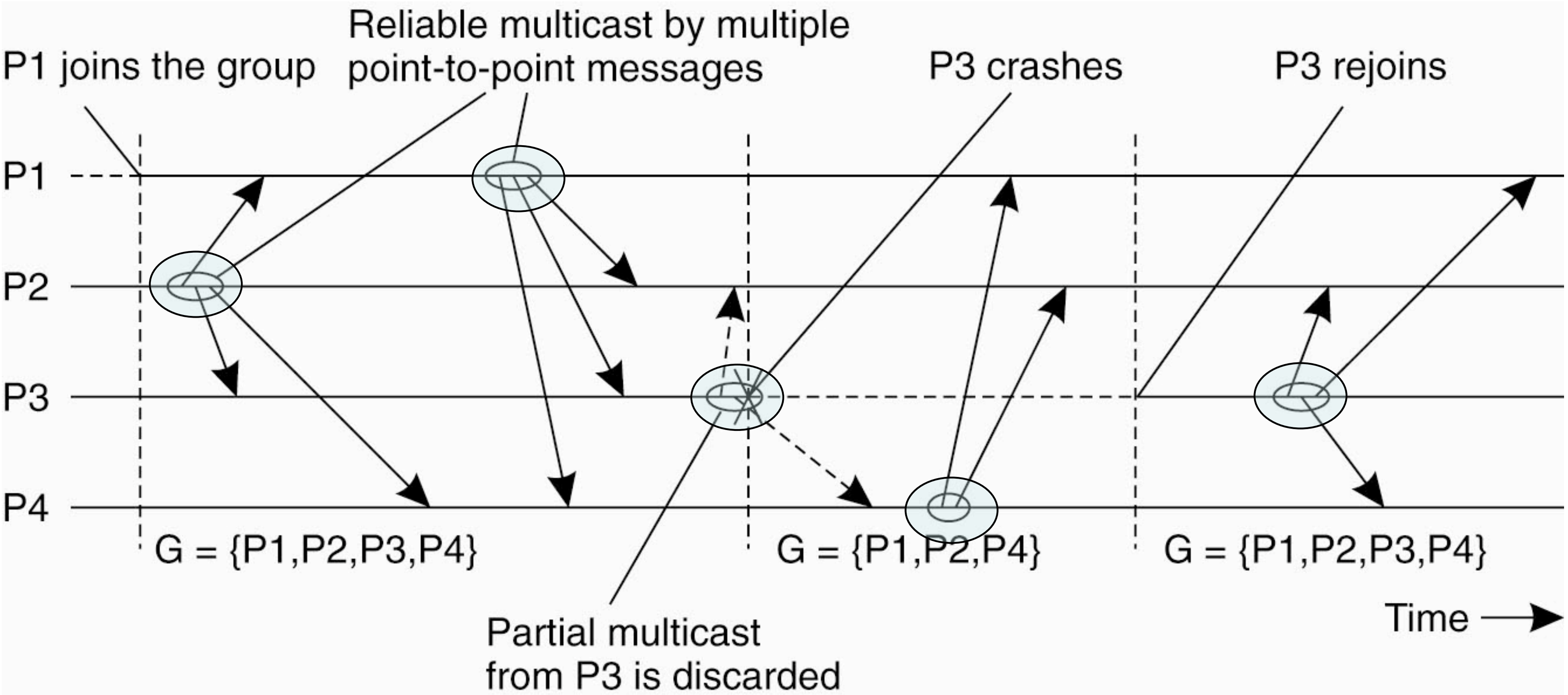
- Figure 8-11. The essence of hierarchical reliable multicasting.
- Each local coordinator forwards the message to its children and
- later handles retransmission requests.

Atomic Multicast

- Guarantee that message is delivered to all group members or none at all
  - At message delivery need to agree on group membership
- E.g., for distributed update



Virtual Synchrony



- Figure 8-13. The principle of virtual synchronous multicast.

Virtual Synchrony

- Reliable multicasting
  - Different orderings possible
    - E.g., none, FIFO, causal, total
- View-based
  - Multicast from non-faulty process delivered to all non-faulty processes in view
  - Multicast from failed process delivered to or ignored by all non-faulty processes in view

Implementing Virtual Synchrony

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

- Figure 8-16. Six different versions of virtually synchronous reliable multicasting.

Message Ordering

Violates total 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	

- Figure 8-15. Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

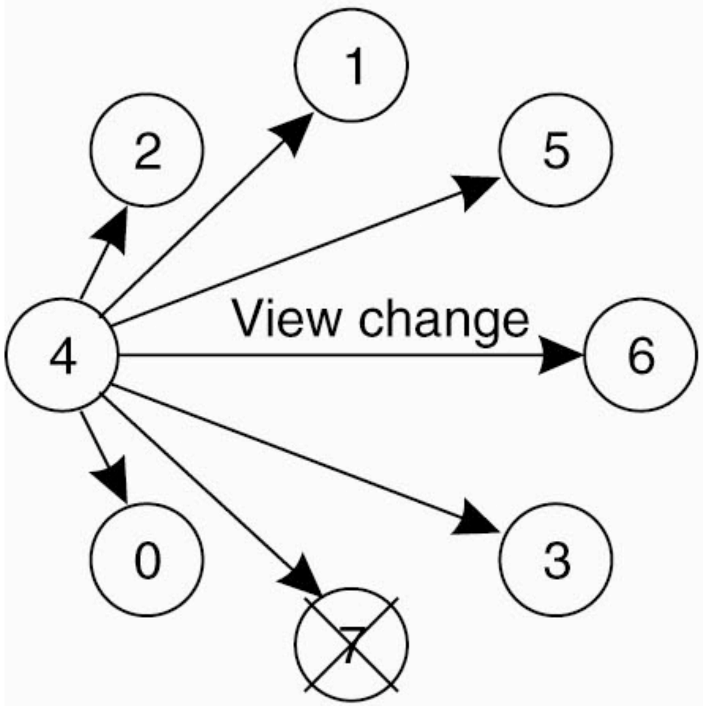
Implementing Virtual Synchrony

- Main issue
  - Handle group membership/view changes
- Assume reliable, point-to-point communication
  - A process *p* that wants to multicast *m* uses point-to-point communication of *m* to each view member
  - E.g., TCP used in the ISIS toolkit
- What if p fails during multicasting?
  - Some processes may have received *m* others may not
  - Failure detection + view change

Failure Detection

- Essentially two approaches used
  - Pinging
    - Are you alive? -> Yes!
  - Heartbeats
    - I am alive!
- Very crude
- Virtual synchrony may also *suspect* processes and just *shun* them

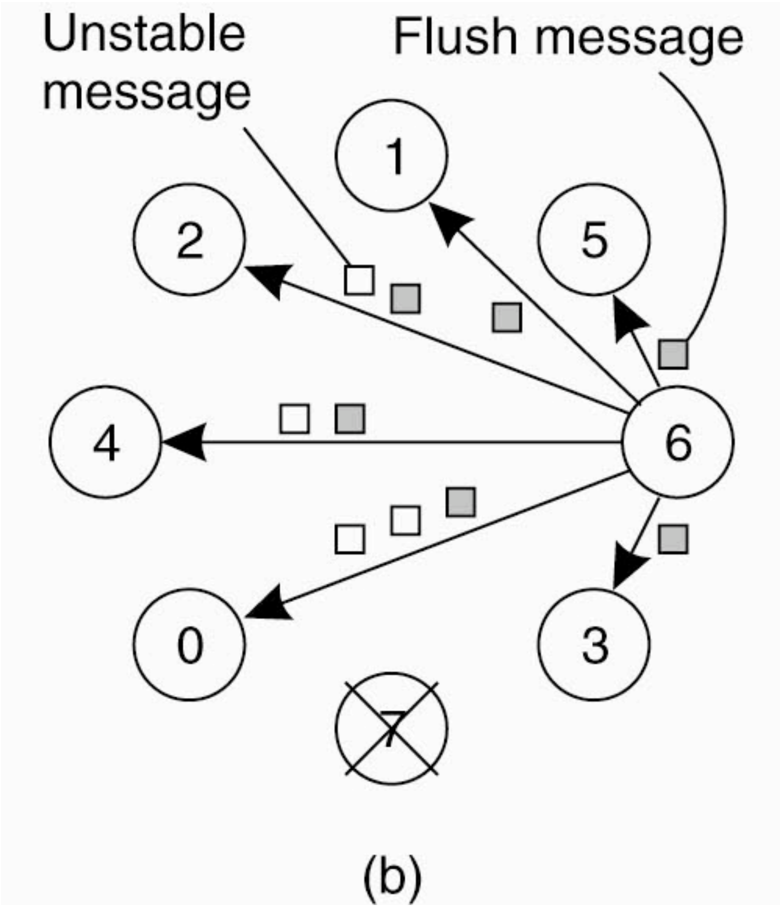
Implementing Virtual Synchrony



(a)

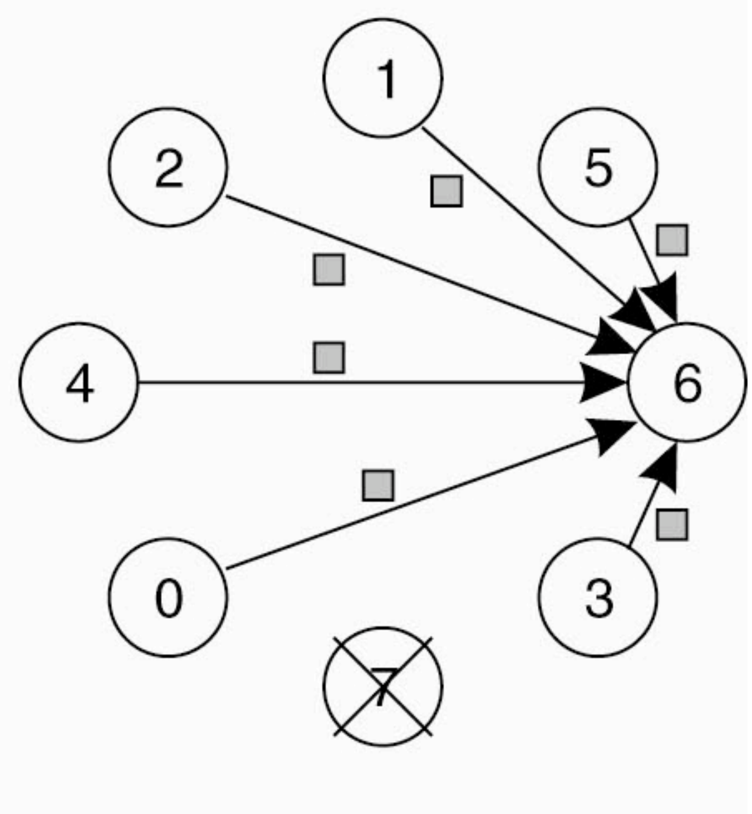
- Figure 8-17. (a) Process 4 notices that process 7 has crashed and sends a view change.

Implementing Virtual Synchrony



- Figure 8-17. (b) Process 6 sends out all its unstable messages, followed by a *flush* message
- All messages that have been received by at least one process will be delivered to all

Implementing Virtual Synchrony



(c)

- Figure 8-17. (c) Process 6 installs the new view when it has received a *flush* message from everyone else.

JGroups

```
<config>
<UDP mcast_send_buf_size="32000" mcast_port="45566" ucast_recv_buf_size="64000" mcast_addr="228.8.8.8"
     loopback="true" mcast_recv_buf_size="64000" max_bundle_size="60000" max_bundle_timeout="30"
     use_incoming_packet_handler="false" use_outgoing_packet_handler="false" ucast_send_buf_size="32000"
     ip_ttl="32" enable_bundling="false"/>
<PING timeout="2000" num_initial_members="3"/>

<MERGE2 max_interval="10000" min_interval="5000"/>
<FD timeout="2000" max_tries="4"/>
<VERIFY_SUSPECT timeout="1500" down_thread="false" up_thread="false"/>
<pbcast.NAKACK max_xmit_size="8192" use_mcast_xmit="false" gc_lag="50"
               retransmit_timeout="100,200,300,600,1200,2400,4800"/>
<UNICAST timeout="1200,2400,3600"/>
<pbcast.STABLE stability_delay="1000" desired_avg_gossip="20000" max_bytes="0"/>
<FRAG frag_size="8192" down_thread="false" up_thread="false"/>
<pbcast.GMS print_local_addr="true" join_timeout="3000" join_retry_timeout="2000" shun="true"/>
<CAUSAL/>
</config>
```

Failure detector

Reliable communication

Handle group membership

Summary

- Independent failures is a defining characteristic of distributed systems
  - Fault tolerance and reliability are fundamental to distributed systems
- Process failures
  - Replication/process groups is a way of handling failure
  - There are limits to fault tolerance – agreements
    - Fail-stop failures
    - Byzantine failures
- Communication failures
  - Client-server communication
  - Group communication