Distributed Systems

Fault Tolerance

Course/Slides Credits

Note: all course presentations are based on those developed by Andrew S. Tanenbaum and Maarten van Steen. They accompany their "Distributed Systems: Principles and Paradigms" textbook (1st & 2nd editions). http://www.prenhall.com/divisions/esm/app/author_tanenbaum/custom/dist_sys_1e/index.html

And additions made by Paul Barry in course CW046-4: Distributed Systems

http://glasnost.itcarlow.ie/~barryp/net4.html

Fault Tolerance Basic Concepts

- Dealing successfully with partial failure within a Distributed System.
- Being fault tolerant is strongly related to what are called dependable systems.
- Dependability implies the following:
  1. Availability
  2. Reliability
  3. Safety
  4. Maintainability

Dependability Basic Concepts

- *Availability* – the system is ready to be used immediately.
- *Reliability* – the system can run continuously without failure.
- *Safety* – if a system fails, nothing catastrophic will happen.
- *Maintainability* – when a system fails, it can be repaired easily and quickly (sometimes, without its users noticing the failure).

But, What Is “Failure”?

- A system is said to “fail” when it *cannot meet* its promises.
- A failure is brought about by the *existence* of “errors” in the system.
- The *cause* of an error is a “fault”.

Types of Fault

- There are three main types of ‘fault’:

1. *Transient Fault* – appears once, then disappears.
2. *Intermittent Fault* – occurs, vanishes, reappears; but: follows no real pattern (worst kind).
3. *Permanent Fault* – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

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	A 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

Failure Masking by Redundancy

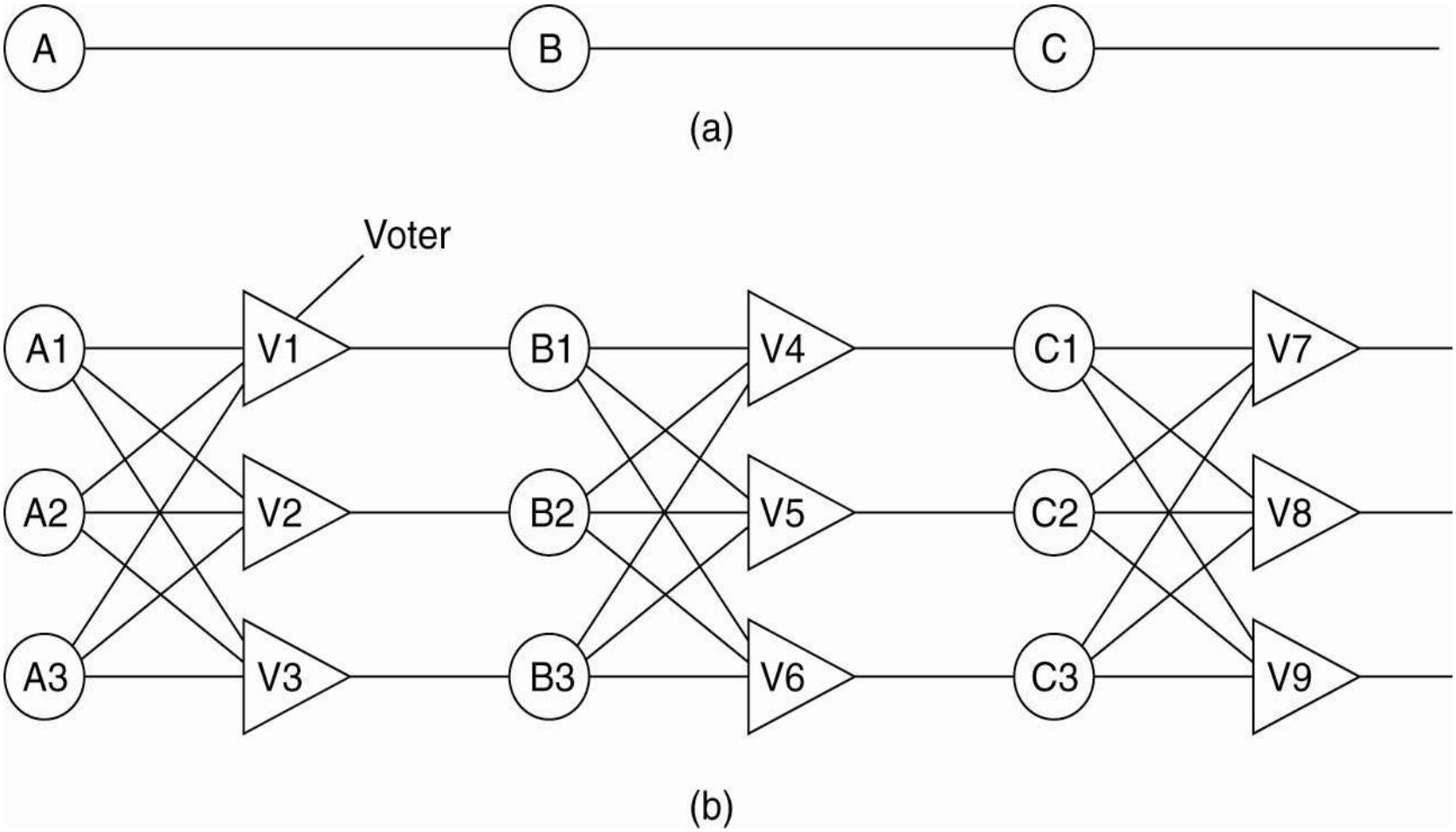
- **Strategy**: hide the occurrence of failure from other processes using *redundancy*.
- Three main types:

1. *Information Redundancy* –

add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).

1. *Time Redundancy* – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).
2. *Physical Redundancy* – add extra (duplicate) hardware and/or software to the system.

Failure Masking by Redundancy



Triple modular redundancy

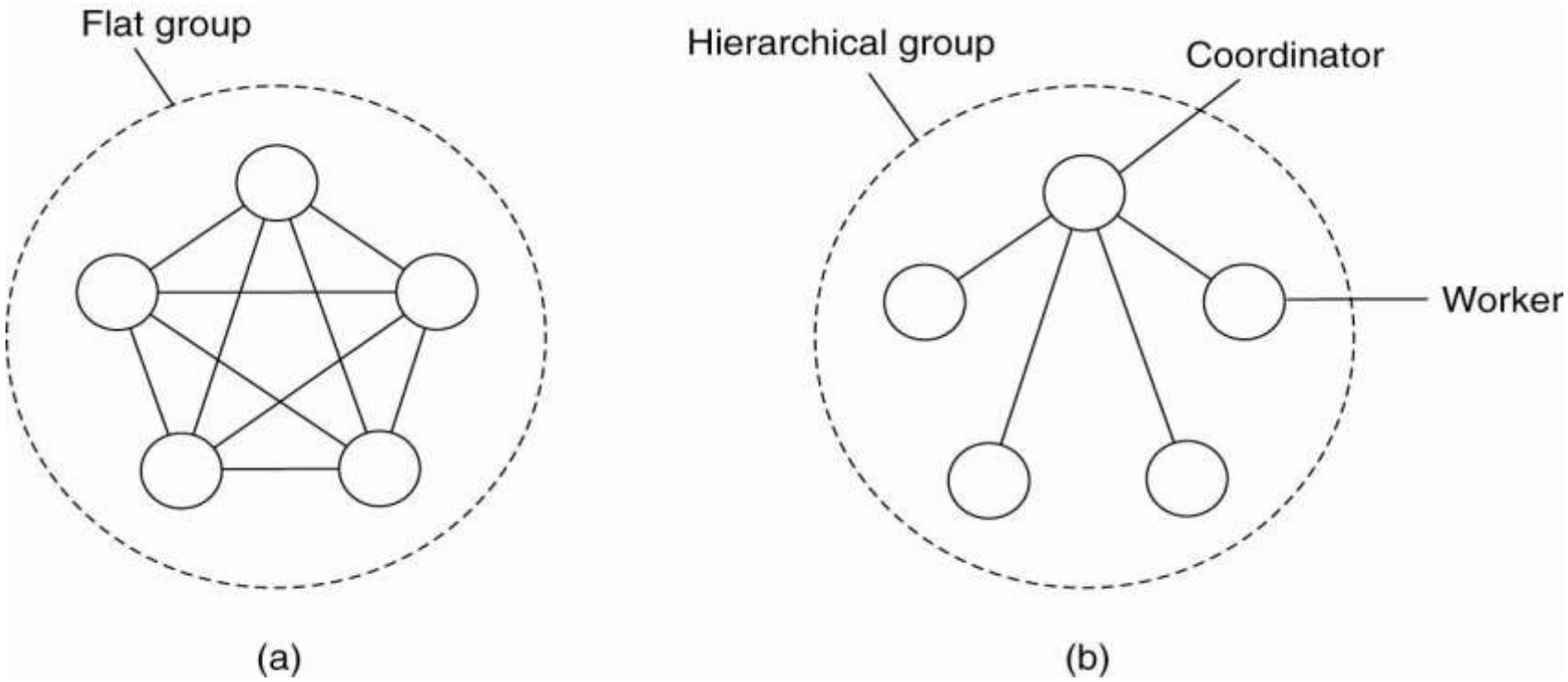
DS Fault Tolerance Topics

- Process Resilience
- Reliable Client/Server Communications
- Reliable Group Communication
- Distributed Commit
- Recovery Strategies

Process Resilience

- Processes can be made fault tolerant by arranging to have a group of processes, with each member of the group being *identical*.
- A message sent to the group is delivered to all of the “copies” of the process (the group members), and then *only one* of them performs the required service.
- If one of the processes fail, it is assumed that one of the others will still be able to function (and service any pending request or operation).

Flat Groups versus Hierarchical Groups



(a) Communication in a flat group.
(b) Communication in a simple hierarchical group.

Failure Masking and Replication

- By organizing a *fault tolerant group of processes*, we can protect a single vulnerable process.
- There are two approaches to arranging the replication of the group:
  1. Primary (backup) Protocols
  2. Replicated-Write Protocols

The Goal of Agreement Algorithms

- "To have all *non-faulty* processes reach consensus on some issue (quickly)."
- The **two-army problem**.
- Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.

History Lesson: The Byzantine Empire

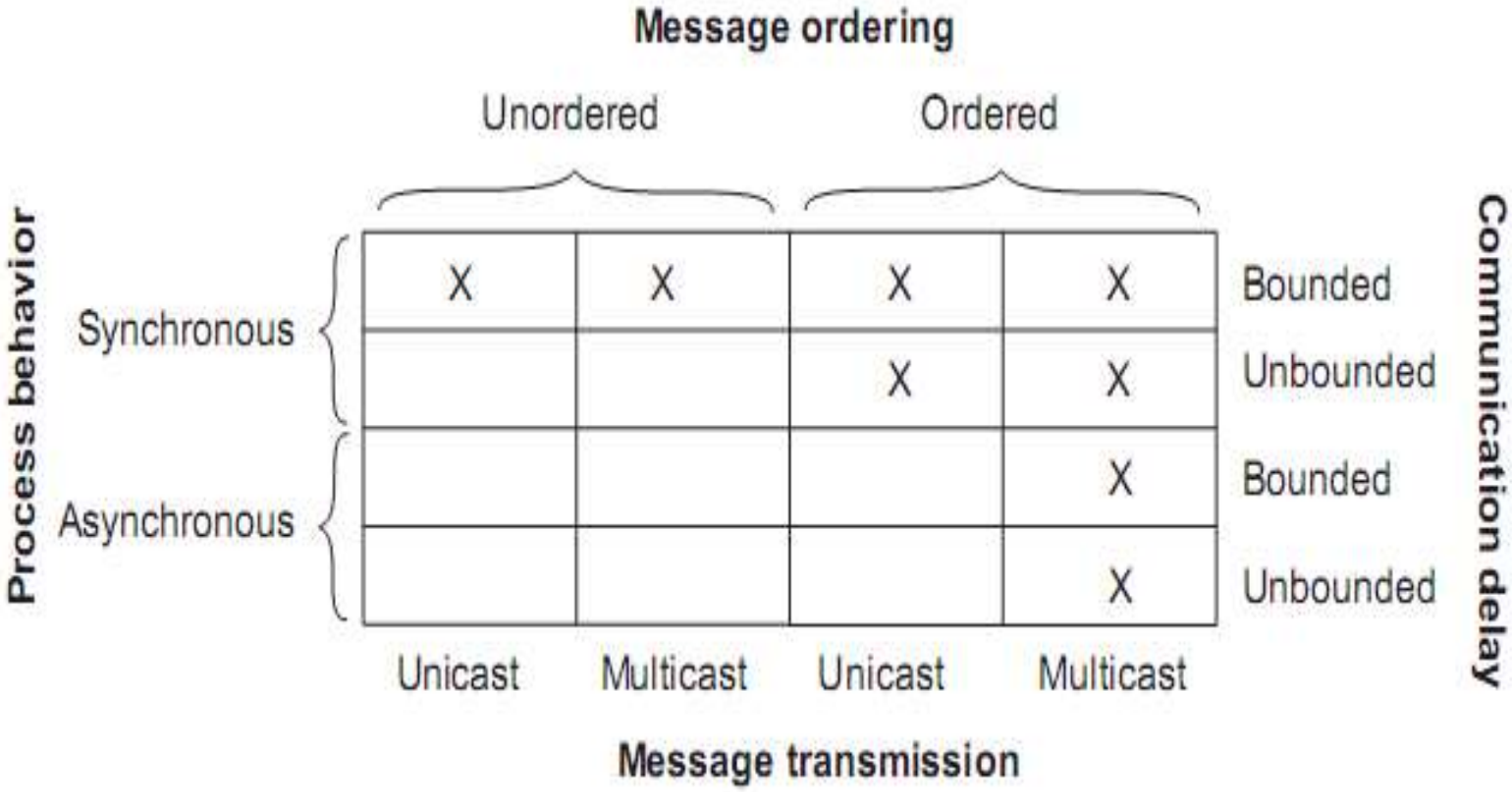
- *Time*: 330-1453 AD.
- *Place*: Balkans and Modern Turkey.
- Endless conspiracies, intrigue, and untruthfulness were alleged to be common practice in the ruling circles of the day (*sounds strangely familiar …*).
- That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurrence can surface in a DS, and is known as ‘Byzantine failure’.
- *Question*: how do we deal with such malicious group members within a distributed system?

Agreement in Faulty Systems (1)

- Possible cases:

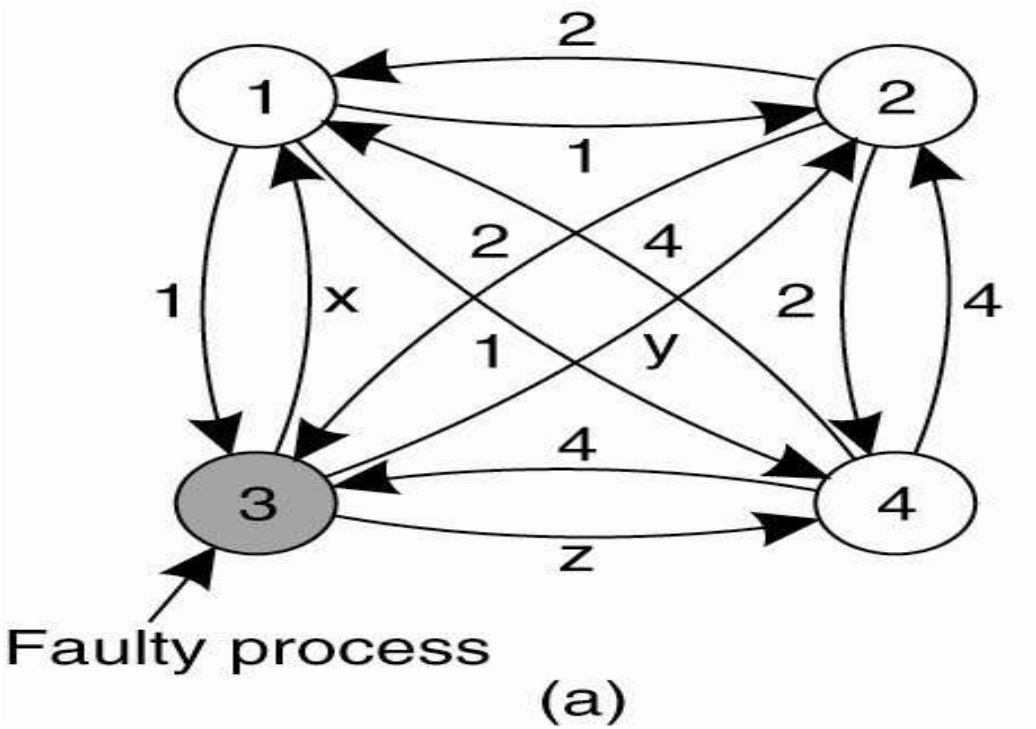
1. Synchronous (lock-step) versus asynchronous systems.
2. Communication delay is bounded (by globaly and predetermined maximum time) or not.
3. Message delivery is ordered (in real-time) or not.
4. Message transmission is done through unicasting or multicasting.

Agreement in Faulty Systems (2)



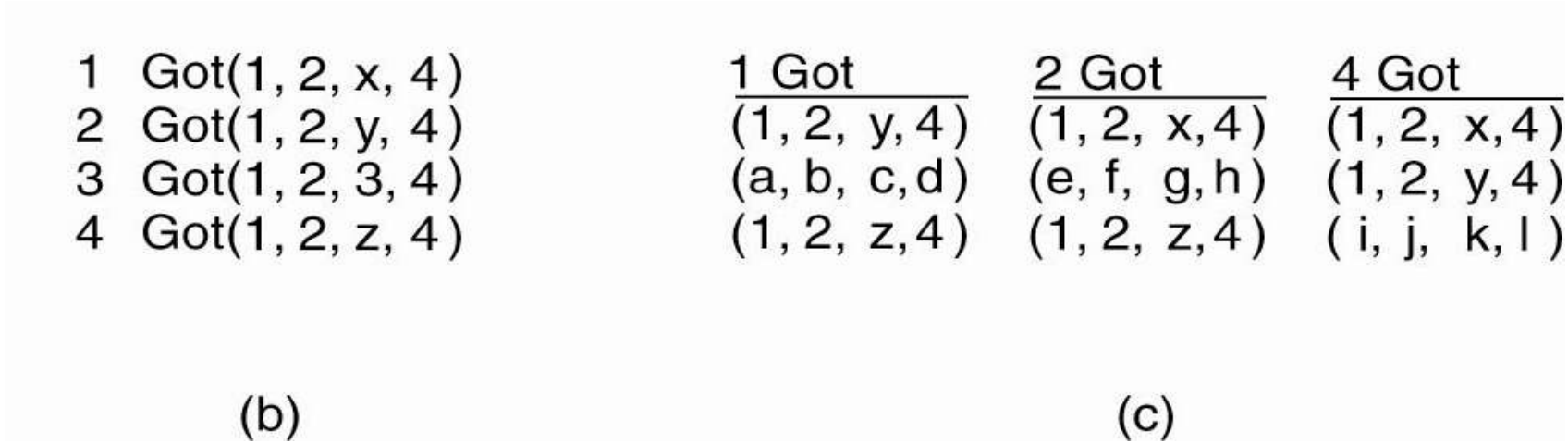
Circumstances under which distributed agreement can be reached

Agreement in Faulty Systems (3)



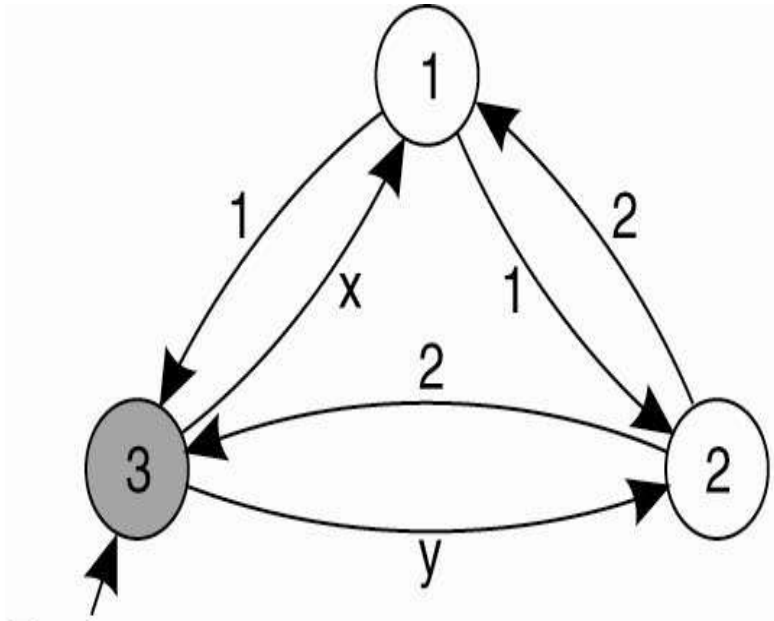
The Byzantine agreement problem for three non-faulty and one faulty process. (a) Each process sends their value to the others.

Agreement in Faulty Systems (4)



The Byzantine agreement problem for three non-faulty and one faulty process. (b) The vectors that each process assembles based on (a). (c) The vectors that each process receives in step 3.

Agreement in Faulty Systems (5)



(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 before, except now with two correct process and one faulty process

Reliable Client/Server Communications

- In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.
- In practice, the focus is on masking *crash* and *omission* failures.
- ***For example***: the point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

RPC Semantics and Failures

- The RPC mechanism works well as long as both the client and server function perfectly.
- Five classes of RPC failure can be identified:

1. *The client cannot locate the server*, so no request can be sent.
2. *The client's request to the server is lost*, so no response is returned by the server to the waiting client.
3. *The server crashes after receiving the request*, and the service request is left acknowledged, but undone.
4. *The server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
5. *The client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

The Five Classes of Failure (1)

- A server in client-server communication:

a) The normal case.

b) Crash *after* service execution.

c) Crash *before* service execution.

The Five Classes of Failure (2)

- An appropriate exception handling mechanism can deal with a missing server. However, such technologies tend to be very language-specific, and they also tend to be non-transparent (which is a big DS ‘no-no’).
- Dealing with lost request messages can be dealt with easily using timeouts. If no ACK arrives in time, the message is resent. Of course, the server needs to be able to deal with the possibility of duplicate requests.

The Five Classes of Failure (3)

- Server crashes are dealt with by implementing one of three possible implementation philosophies:
  1. *At least once semantics*: a guarantee is given that the RPC occurred at least once, but (also) possibly more that once.
  2. *At most once semantics*: a guarantee is given that the RPC occurred at most once, but possibly not at all.
  3. *No semantics*: nothing is guaranteed, and client and servers take their chances!
- It has proved difficult to provide *exactly once semantics*.

Server Crashes (1)

- Remote operation: print some text and (when done) send a completion message.
- Three events that can happen at the server:

1. Send the completion message (M),
2. Print the text (P),
3. Crash (C).

Server Crashes (2)

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

Server Crashes (3)

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

Different combinations of client and server strategies in the presence of server crashes

The Five Classes of Failure (4)

- Lost replies are difficult to deal with.
- *Why* was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*? Emmmmm?
- A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*. (For example: a read of a static web-page is said to be idempotent).
- *Nonidempotent* requests (for example, the electronic transfer of funds) are a little harder to deal with. A common solution is to employ *unique sequence numbers*. Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

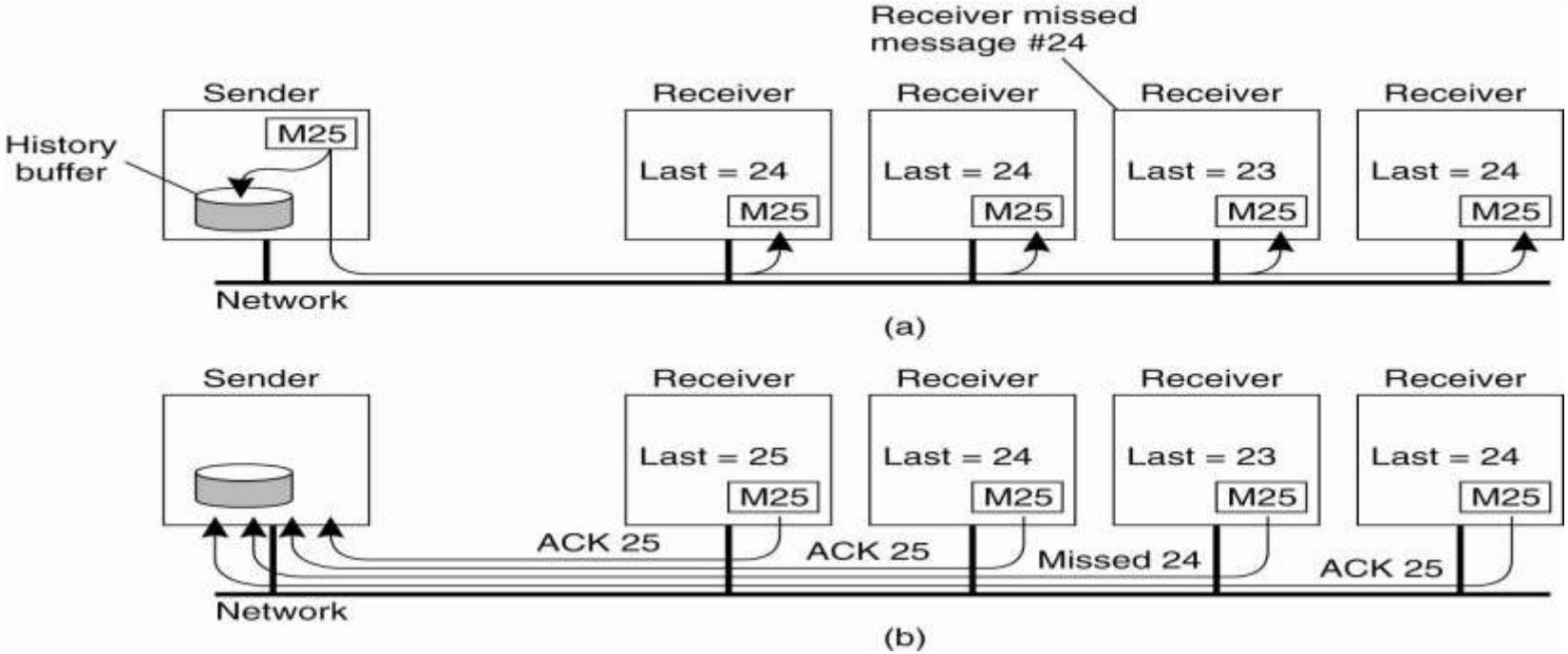
The Five Classes of Failure (5)

- When a client crashes, and when an ‘old’ reply arrives, such a reply is known as an *orphan*.
- Four orphan solutions have been proposed:
  1. *extermination* (the orphan is simply killed-off).
  2. *reincarnation* (each client session has an *epoch* associated with it, making orphans easy to spot).
  3. *gentle reincarnation* (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed).
  4. *expiration* (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).
- In practice, however, none of these methods are desirable for dealing with orphans. Research continues …

Reliable Group Communication

- Reliable multicast services guarantee that all messages are delivered to all members of a process group.
- Sounds simple, but is surprisingly *tricky* (as multicasting services tend to be *inherently* unreliable).
- For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:
  - What happens if a process *joins* the group during communication?
  - Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes



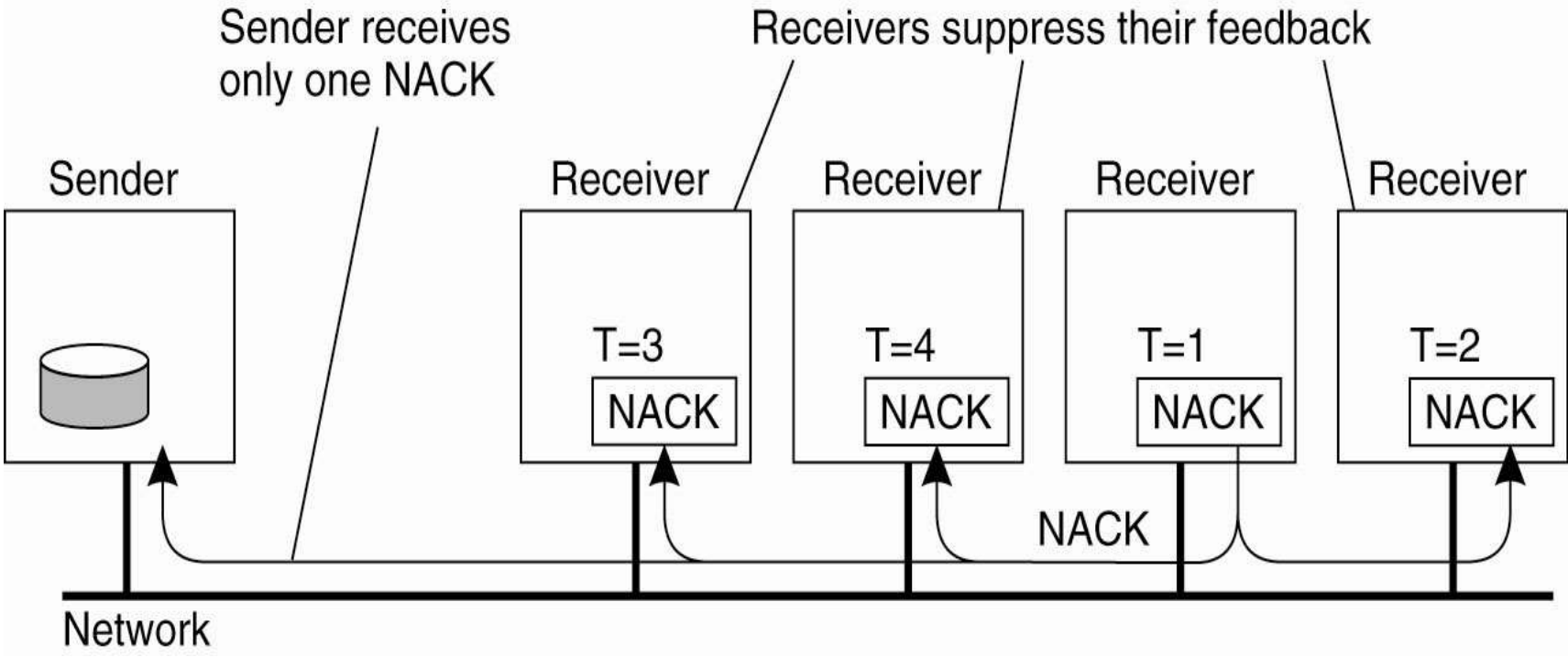
A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.

(a) Message transmission. (b) Reporting feedback

SRM: Scalable Reliable Multicasting

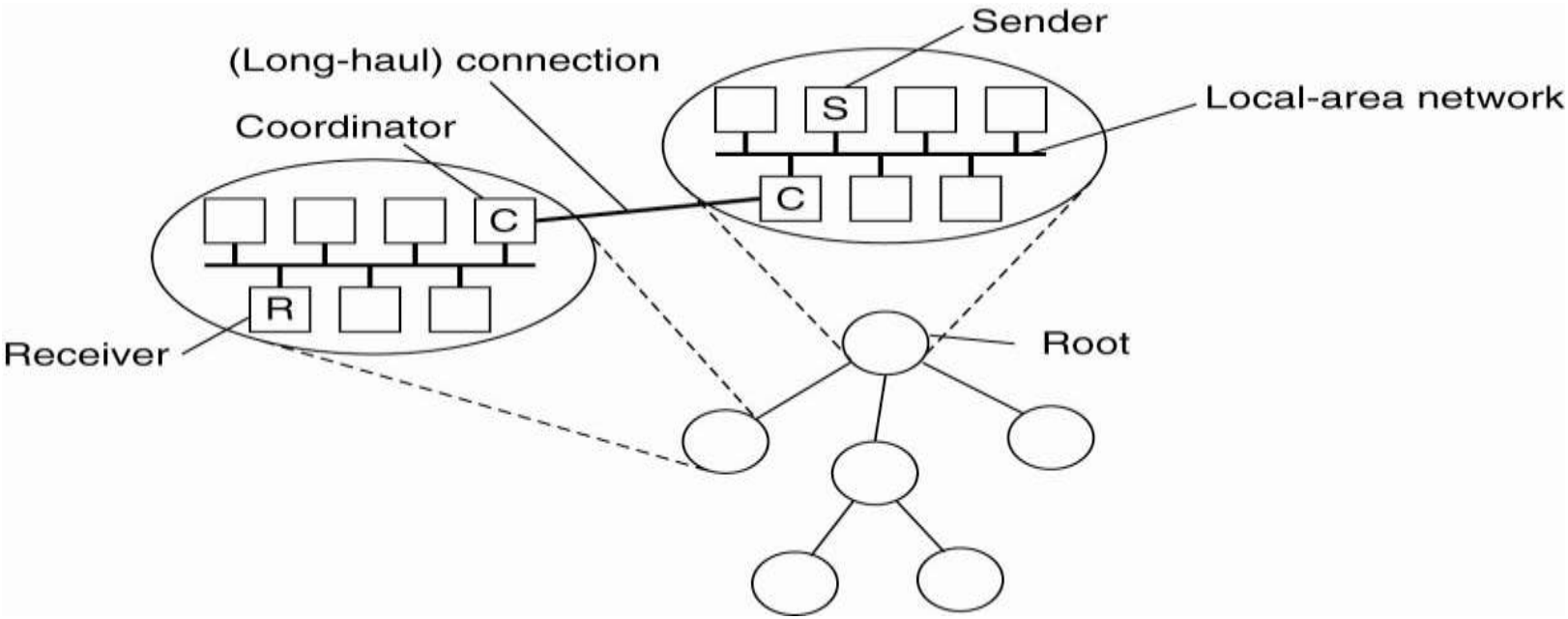
- Receivers *never* acknowledge successful delivery.
- **Only missing messages are reported.**
- NACKs are multicast to all group members.
- This allows other members to suppress their feedback, if necessary.
- To avoid “retransmission clashes”, each member is required to wait a random delay prior to NACKing.

Nonhierarchical Feedback Control



Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the suppression of others

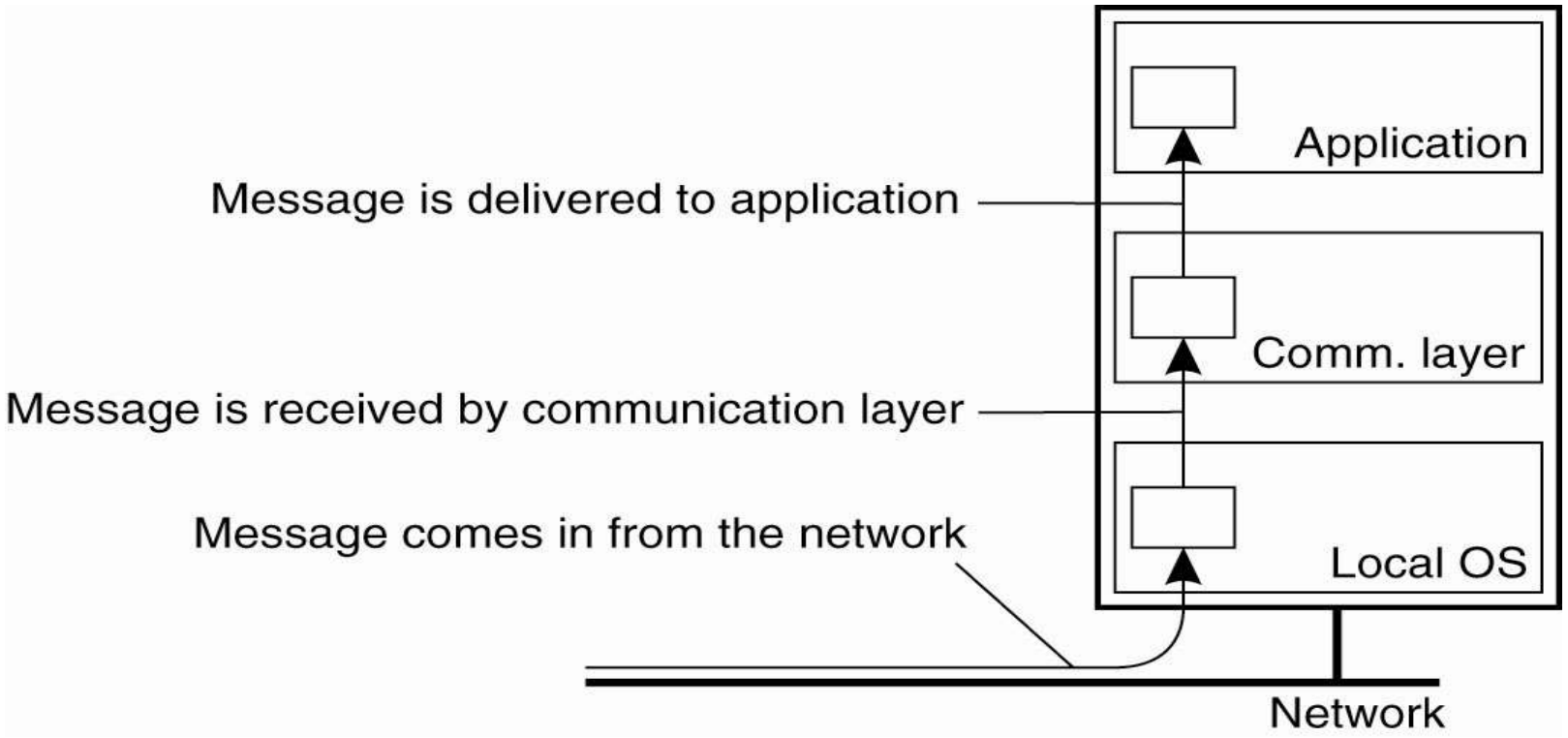
Hierarchical Feedback Control



The essence of hierarchical reliable multicasting.

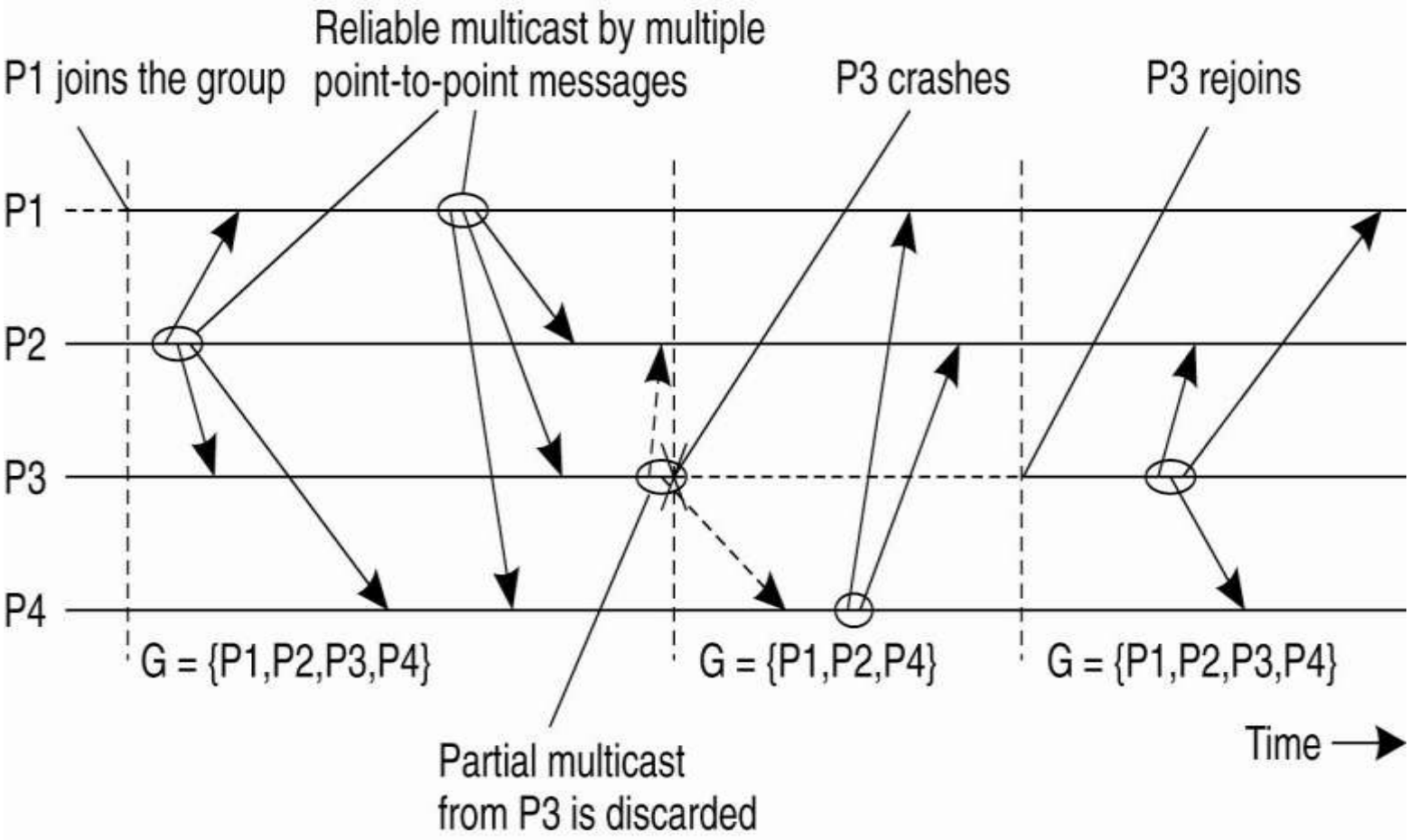
Each local coordinator forwards the message to its children and later handles retransmission requests.

Virtual Synchrony (1)



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

Virtual Synchrony (2)



The principle of virtual synchronous multicast

Message Ordering (1)

- Four different orderings are distinguished:

1. Unordered multicasts
2. FIFO-ordered multicasts
3. Causally-ordered multicasts
4. Totally-ordered multicasts

Message Ordering (2)

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

Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

Message Ordering (3)

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	

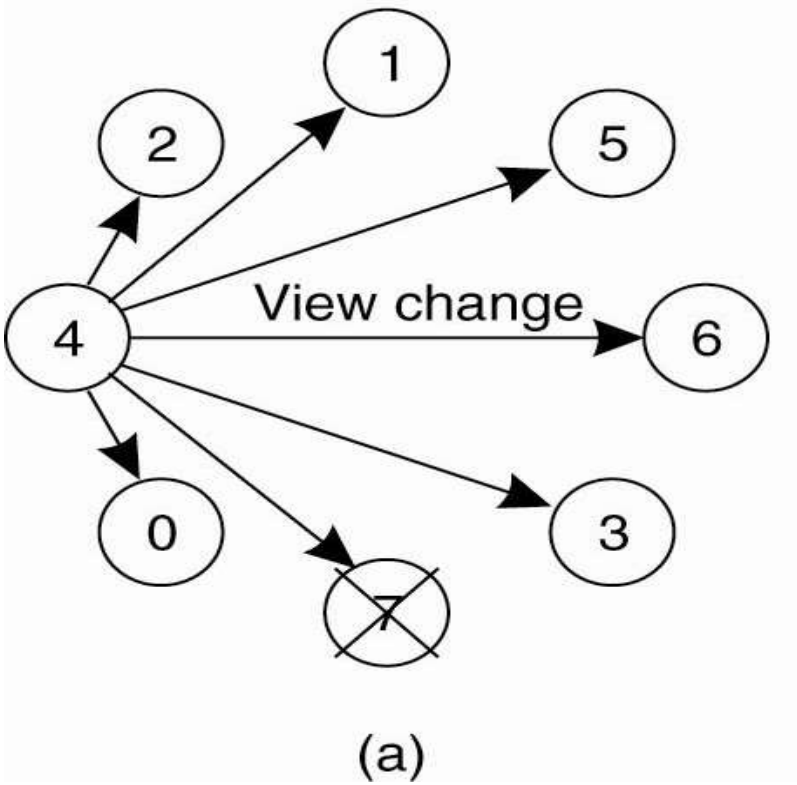
Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

Implementing Virtual Synchrony (1)

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

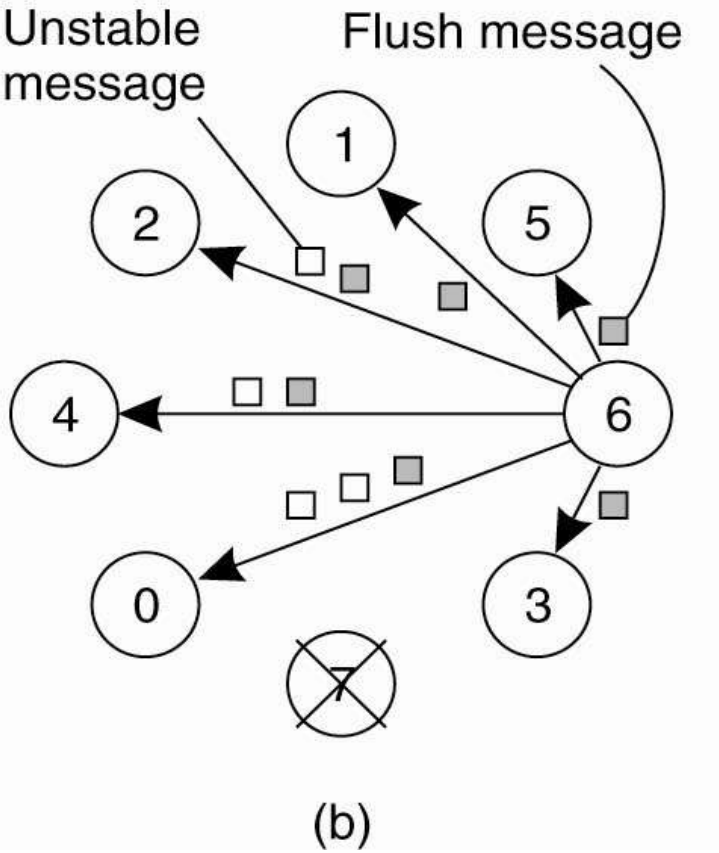
Six different versions of virtually synchronous reliable multicasting

Implementing Virtual Synchrony (2)



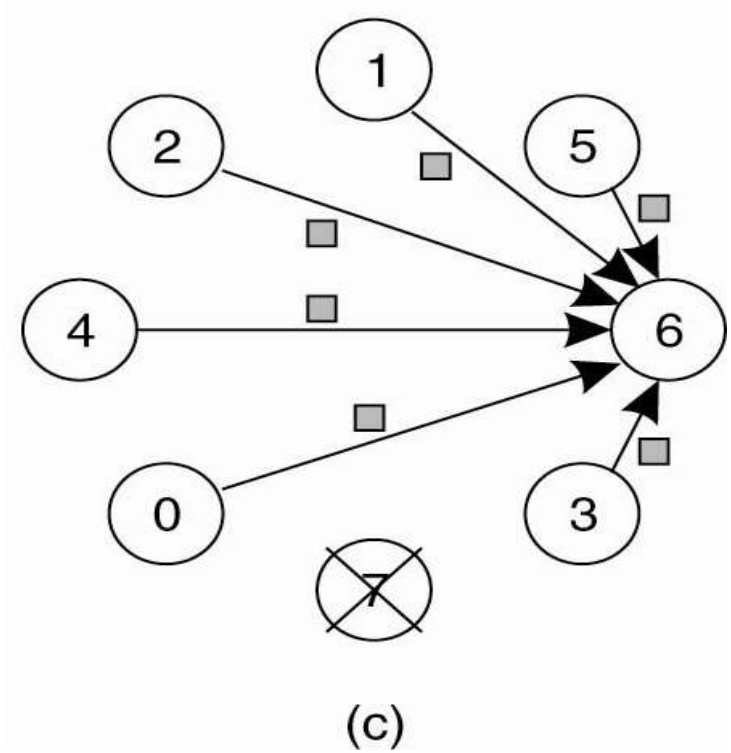
(a) Process 4 notices that process 7 has crashed and sends a view change

Implementing Virtual Synchrony (3)



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

Implementing Virtual Synchrony (4)



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

Distributed Commit

- **General Goal:**
  - *We want an operation to be performed by all group members, or none at all.*
  - [In the case of atomic multicasting, the operation is the delivery of the message.]

There are three types of “commit protocol”:

1. single-phase commit
2. two-phase commit
3. three-phase commit

Commit Protocols

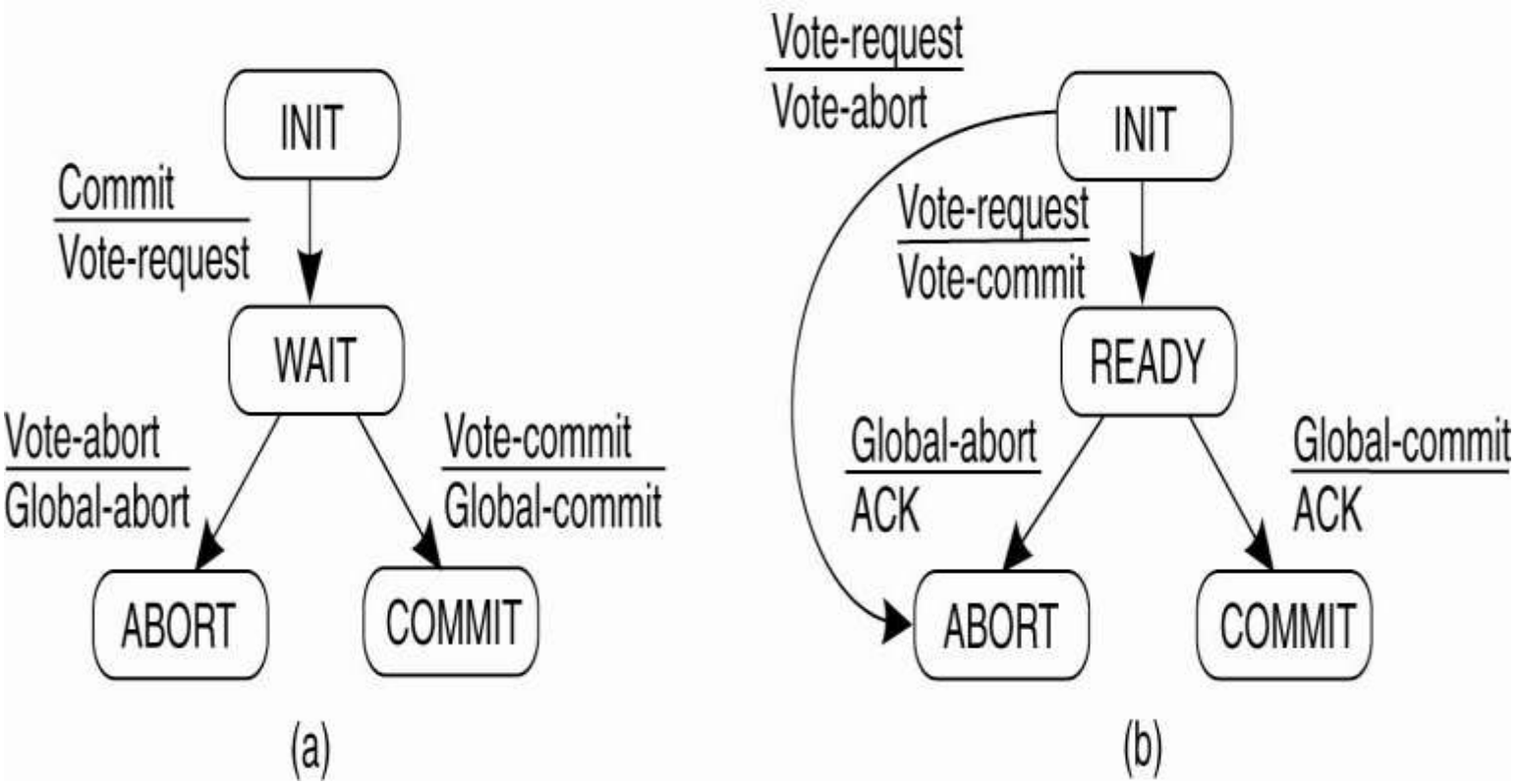
- **One-Phase Commit Protocol**:
  - An elected coordinator tells all the other processes to perform the operation in question.
- But, what if a process cannot perform the operation? There's no way to tell the coordinator! Whoops …
- **The solutions**:
  - The *Two-Phase* and *Three-Phase Commit Protocols*.

The Two-Phase Commit Protocol

- First developed in 1978!!!
- *Summarized: GET READY, OK, GO AHEAD.*

1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

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)

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 (3)

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;
}
```

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

Two-Phase Commit (4)

```
. . .   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 a two-phase commit protocol

Two-Phase Commit (5)

(a) The steps taken by a 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;
    }
```

(a)

Two-Phase Commit (6)

```
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;
        }
```

(a)

```
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)

(b) The steps for handling incoming decision requests

Big Problem with Two-Phase Commit

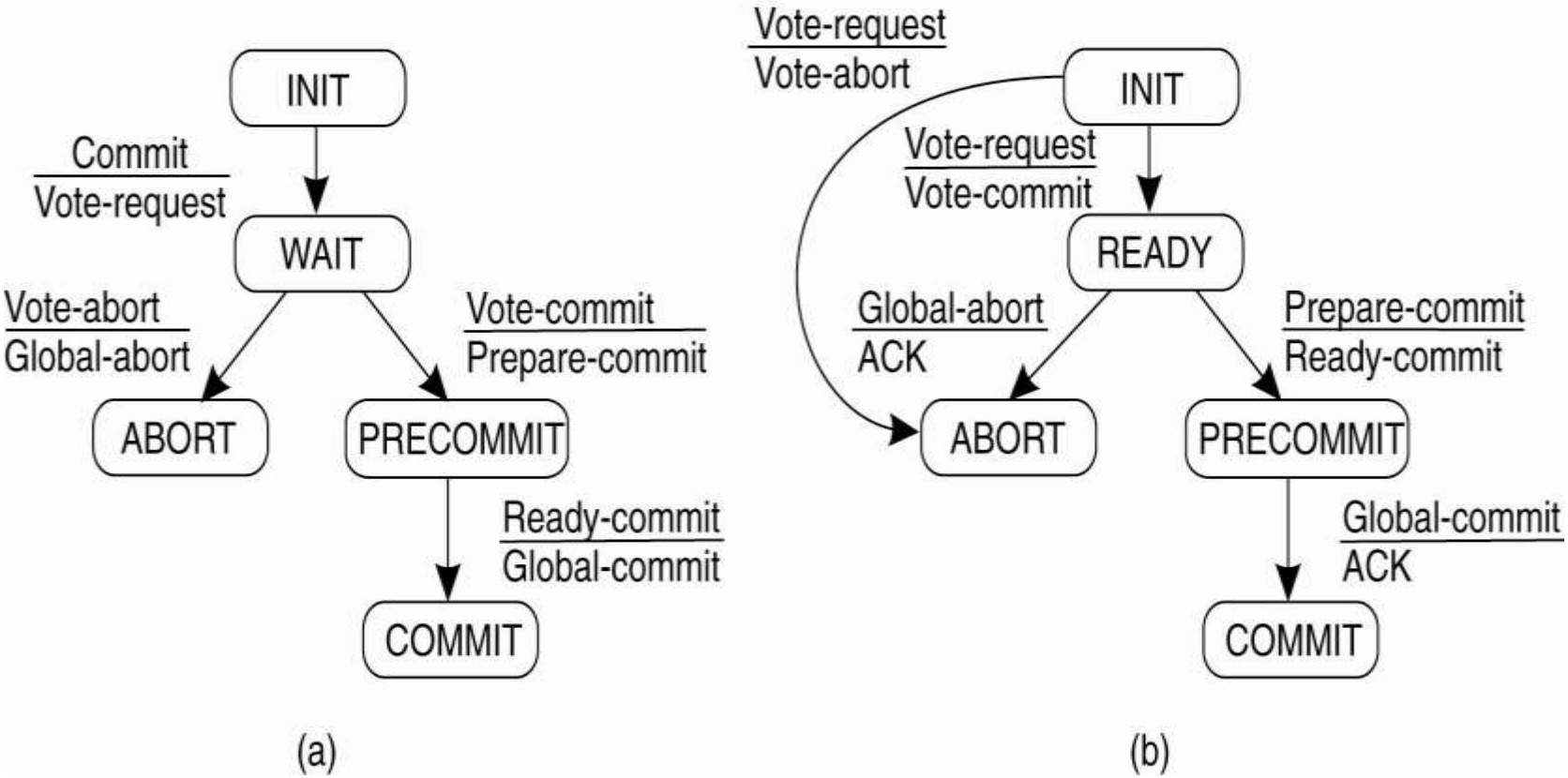
- It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.
- If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers* ...
- Two-Phase Commit is known as a **blocking-commit protocol** for this reason.
- The solution? *The Three-Phase Commit Protocol*.

Three-Phase Commit (1)

- The states of the coordinator and each participant satisfy the following two conditions:

1. There is no single state from which it is possible to make a transition directly to either a COMMIT or an ABORT state.
2. There is no state in which it is not possible to make a final decision, and from which a transition to a COMMIT state can be made.

Three-Phase Commit (2)



(a) The finite state machine for the coordinator in 3PC.
(b) The finite state machine for a participant.

Recovery Strategies

- Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state.
- Recovery from an error is *fundamental* to fault tolerance.
- Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.

Forward and Backward Recovery

- **Major disadvantage of Backward Recovery**:
  - Checkpointing can be very expensive (especially when errors are very rare).
  - [Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of checkpointing.].
- **Major disadvantage of Forward Recovery**:
  - In order to work, all potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

Recovery Example

- **Consider as an example:**
  Reliable Communications
- *Retransmission* of a lost/damaged packet is an example of a backward recovery technique.
- When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets, then this is known as *Erasure Correction*. This is an example of a forward recovery technique.