Distributed Operating Systems
Fault Tolerance

dr inż. Adam Kozakiewicz

akozakie@elka.pw.edu.pl

Institute of Control and Information Engineering
Warsaw University of Technology

Fault Tolerance

1. Basic concepts – terminology
2. Process resilience
   - √ groups and failure masking
3. Reliable communication
   - √ reliable client-server communication
   - √ reliable group communication
4. Distributed commit
   - √ two-phase commit (2PC)
   - √ three-phase commit (3PC)

Dependability

A component providing services to clients may require the services of other components – in that case the component **depends** on some other component.

Dependability

A component C depends on $C*$ if the correctness of C's behavior depends on the correctness of $C*$'s behavior.

Properties of dependability:

- √ **availability** – readiness for usage,
- √ **reliability** – continuity of service delivery,
- √ **safety** – very low probability of catastrophes,
- √ **maintainability** – low difficulty of repair after failure.

In distributed systems components can be processes or channels.

Fault Terminology

- **Failure**: a component's behavior violates its specifications.
- **Error**: the part of the component's state that can lead to failure.
- **Fault**: the cause of an error.

Different fault management techniques:

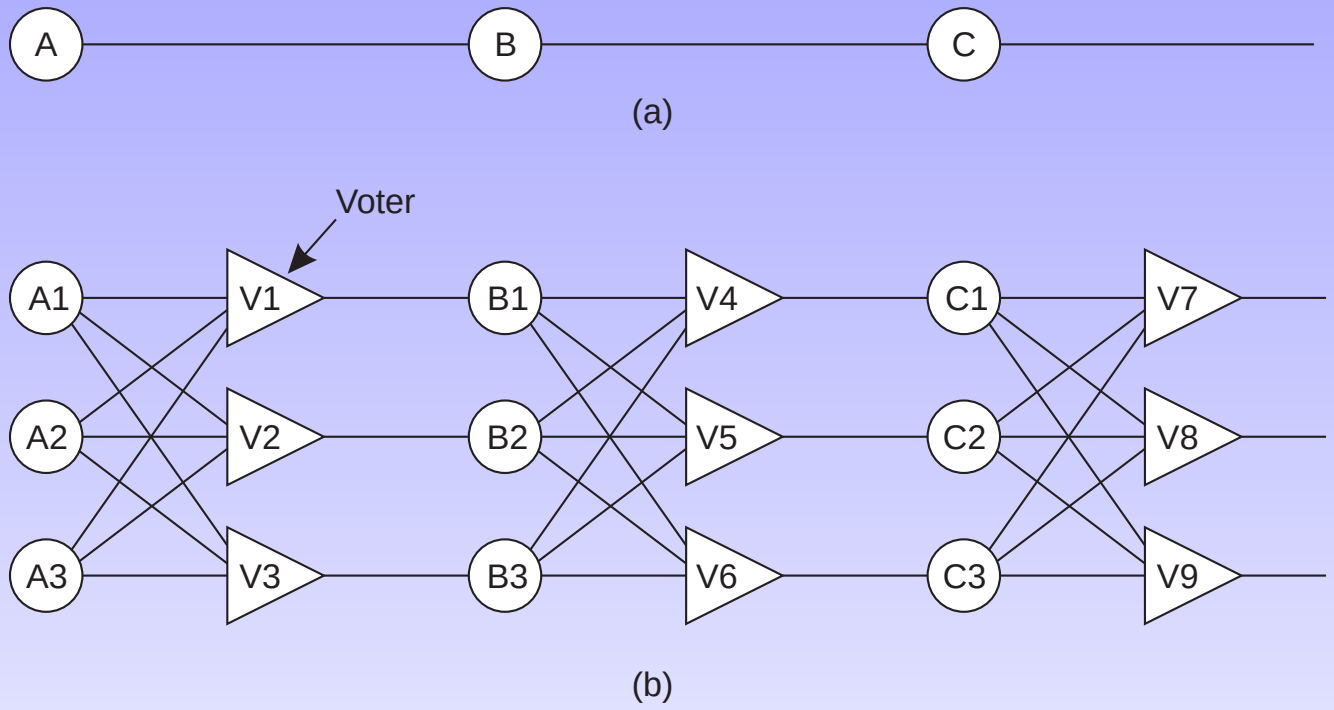
- **Fault prevention**: prevent the occurrence of a fault,
- **Fault tolerance**: mask the presence of faults – build the component so that it is able to meet its specifications in the presence of faults.
- **Fault removal**: reduce the presence, number and seriousness of faults.
- **Fault forecasting**: estimate the present number, future incidence and consequences of faults.

Types of Faults

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

Different types of failures. Crash failures are the least severe, arbitrary failures – the worst.

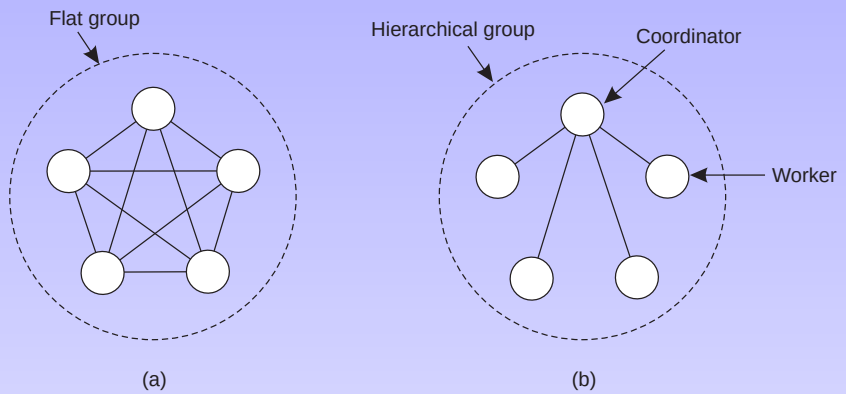
Failure Masking by Redundancy



Triple modular redundancy (TMR).

Process Resilience

Process groups: protection against faulty processes is possible by replicating and distributing computations in a group.



a. **flat groups**: very good fault tolerance due to immediate information exchange in the whole group. Difficult to implement, increases overhead (many messages, completely distributed control).

b. **hierarchical groups**: a single coordinator, not very fault tolerant (single point of failure) or scalable, but easy to implement.

Groups and Failure Masking (1)

Group tolerance

If a group can mask any k concurrent member failures, it is **k-fault tolerant**. k is called **degree of fault tolerance**.

What is the minimum size of a k-fault tolerant group of identical processes processing the same input in the same order?

- √ *crash* or *performance failure semantics* $\Rightarrow k+1$
- √ *arbitrary failure semantics*, using voting $\Rightarrow 2k+1$

Groups and Failure Masking (2)

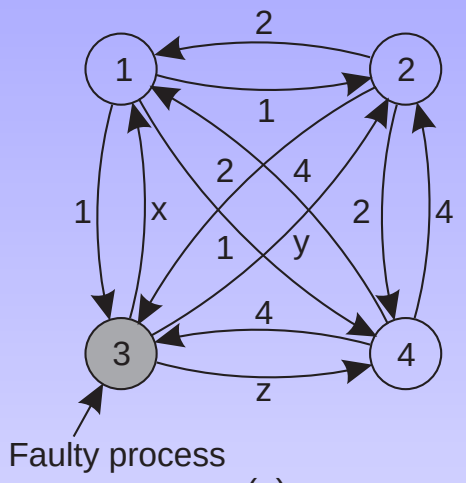
Assumption: distributed computation, that is the members are not identical.

Problem: nonfaulty group members should reach agreement on the same value.

In arbitrary failure semantics $3k + 1$ group members are necessary.

At least $2k + 1$ loyalists are needed to reach a majority vote in the presence of k traitors. This type of situation is called a **Byzantine failure**.

Groups and Failure Masking (3)



(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)

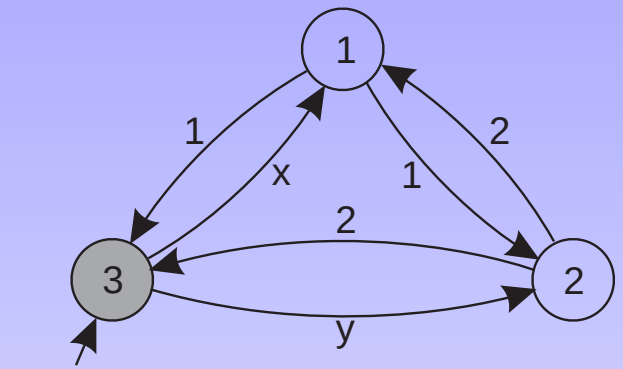
1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

The byzantine generals problem for 3 loyal generals and one traitor.

a. generals announce their troop strengths (in thousands of soldiers).
b. each general assembles a vector of values and announces it.
c. in the end all generals have a set of vectors large enough to find the right answers.

Groups and Failure Masking (4)



(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 example with too few loyal generals.

Reliable Communication

Methods for providing **reliable communication channels**

Error **detection**:

- √ use checksums in the packets/frames to allow for bit error detection,
- √ number the frames to detect packet loss.

Error **correction**:

- √ add enough redundancy (e.g. better checksums) to allow automatic correction of corrupted packets,
- √ request retransmission of lost (or last N) packets.

We assume point-to-point communication in most of this lecture.

Reliable RPC (1)

What can go wrong during RPC?

1. the client cannot locate the server
2. the request is lost
3. the server crashes
4. the response is lost
5. the client crashes

Solutions:

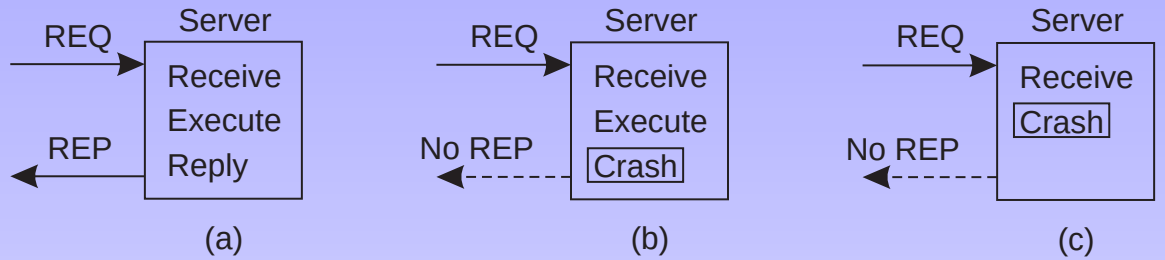
1: Trivial – just inform the client process.

2: No problem – resend the message.

3: **Difficult** – the server is down and nobody knows for sure what was done and what wasn't completed.

Reliable RPC (2)

A contract is necessary – the client must know, what to expect from the server.



(a) normal case (b) crash after execution (c) crash before execution.

Possible RPC server semantics:

- √ **at-least-once-semantics**: the server guarantees that if a request was received, the requested operation will be executed at least once, whatever happens.
- √ **at-most-once-semantics**: the server guarantees that if a request was received, the requested operation will be executed at most once, whatever happens.

Reliable RPC (3)

4: **Difficult** – for the client it looks exactly the same as a crash.

Possible solution: none. **Idempotent** operations do help, though (operations that can be repeated without ill effects).

5: Problem – the server is wasting resources (*orphan computation*).

Possible solutions::

- ★ clients kill all orphans after restart,
- ★ epoch numbers broadcasted when recovering let the servers do the killing,
- ★ timeout – computations must finish and succesfully return the result in a given time, orphans die automatically.

Reliable Multicasting (1)

Basic model: a multicast channel c with two (possibly ovelapping) groups of processes:

- the sender group $SND(c)$ – processes that submit messages to channel c,
- the receiver group $RCV(c)$ – processes that can receive messages from channel c.

`Simple reliability` If process $P \in RCV(c)$ at the time message m was submitted to c and P didn't leave $RCV(c)$, then m should be delivered to P.

`Atomic multicast` A message m submitted to channel c, it will either be delivered to *all* processes $P_i \in RCV(c)$ or *none*.

Reliable Multicasting (2)

In a LAN, reliable multicast is relatively easy:

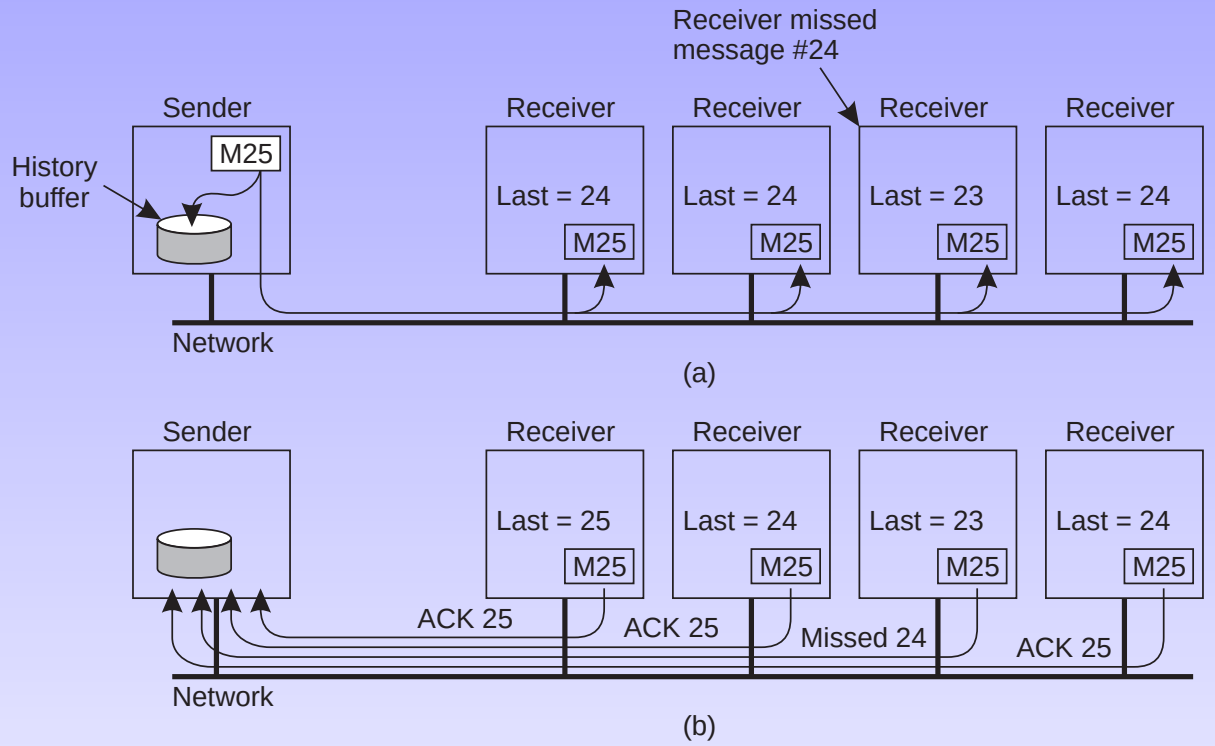
The sender logs messages submitted to c:

- √ when P sends message m, P also stores it in the history buffer,
- √ each receiver acknowledges the receipt of m, or requests retransmission if the message was lost (lack of acknowledgement can be treated as retransmission request),
- √ the sender P removes m from history buffer after receiving all expected acknowledgments.

This algorithm doesn't scale!

- √ if $RCV(c)$ is large, P will be swamped with feedback (ACKs and $NACK$s),
- √ the sender P must know all members of $RCV(c)$, or at least their number.

Basic Reliable Multicasting Schemes

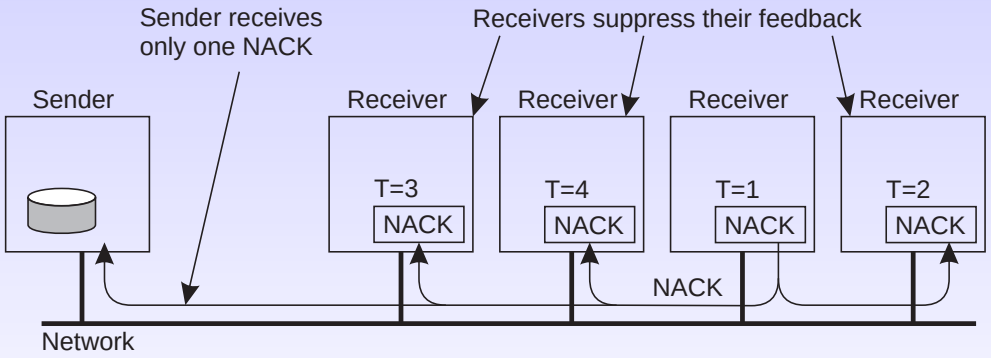


A simple solution to reliable multicasting when all members of $RCV(c)$ are known and reliable.

Scalable RM: Feedback Suppression

Idea: process P **suppresses its own feedback** when it notices another process Q already asking for retransmission.
Assumptions:

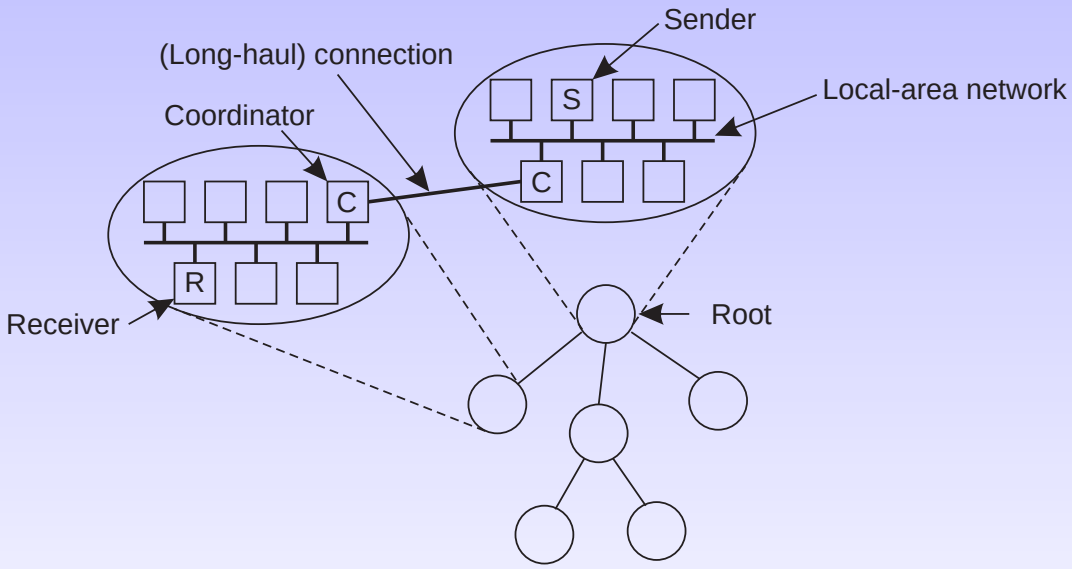
- √ all receivers listen also to the common feedback channel,
- √ each process P schedules its feedback randomly (adding delay) and suppresses it if another feedback message is observed,
- √ random schedule ensures that usually only one feedback message is sent.



Scalable RM: Hierarchical Solutions

Idea: the feedback channel should be hierarchical, with all feedback messages sent only to the root. The intermediate nodes aggregate feedback before passing it on.

Main challenge: dynamic construction of feedback trees.

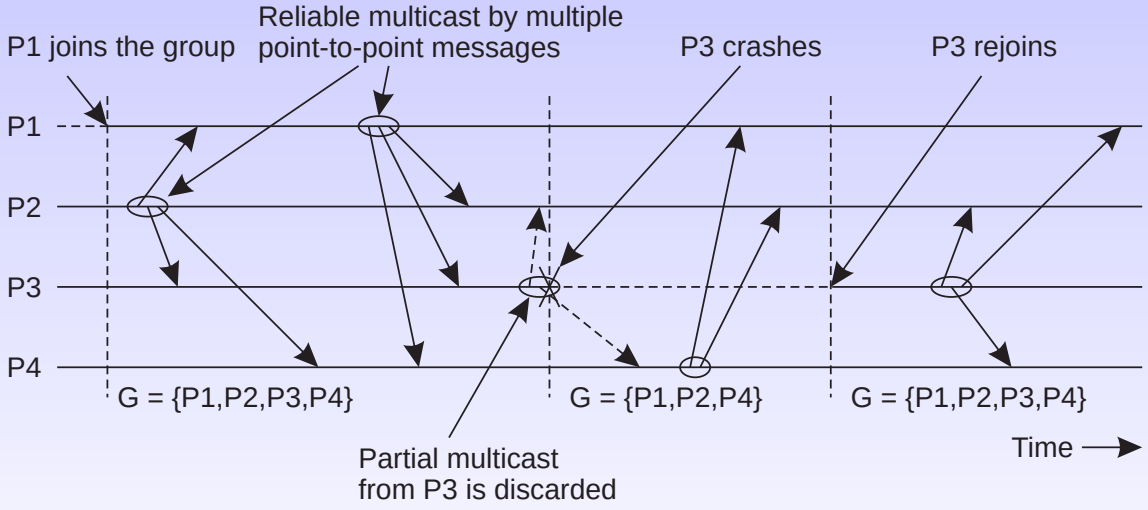


Atomic Multicast

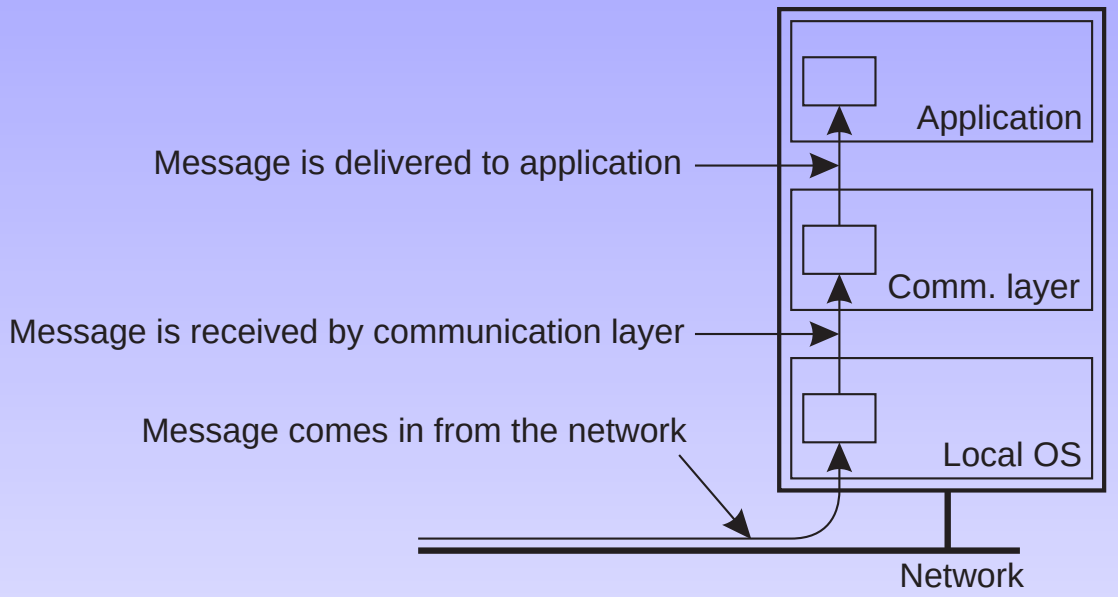
Idea: formulate reliable multicasting in terms of process groups in the presence of process failures and changes to group membership.

Guarantee: the message will be delivered to all non-faulty members of the current group and only to them. All members must agree on the current group membership.

Keyword: *virtually synchronous multicast*.



Virtual Synchrony (1)



Logical organization of a distributed system – message delivery and receipt are two different events!

Virtual Synchrony (2)

Idea: We consider **views** $V \subseteq RCV(c) \cup SND(c)$.

Processes are added or removed from V through view changes to $V*$. The view change is executed locally by each process $P \in V \cap V*$.

1. For each consistent state, there is a unique view on which all its members agree. Note: this implies, that all non-faulty processes see all view changes in the same order.
2. If a message m was sent to V before a view change vc to $V*$, then either all $P \in V$ that execute vc receive m, or none of them. Note: all non-faulty members in the same view receive the same set of multicast messages.
3. A message m sent to view V can only be delivered to processes in V and is discarded by successive views.

A reliable multicast algorithm satisfying 1. – 3. is **virtually synchronous**.

Virtual Synchrony (3)

Note: the sender of a message m to view V does not have to be a member of V.

If the sender S is in V and crashes, then m is flushed before S is removed from V: m will never be delivered after the point that $S \notin V$.

Note: messages from S may still be delivered to all (or none) non-faulty members of V *before* they all agree on the new view without S in it.

If a receiver P fails, the message m may be lost, but we know exactly what was received in V, so it can be recovered. Alternatively, we can deliver m to all messages in $V - P$.

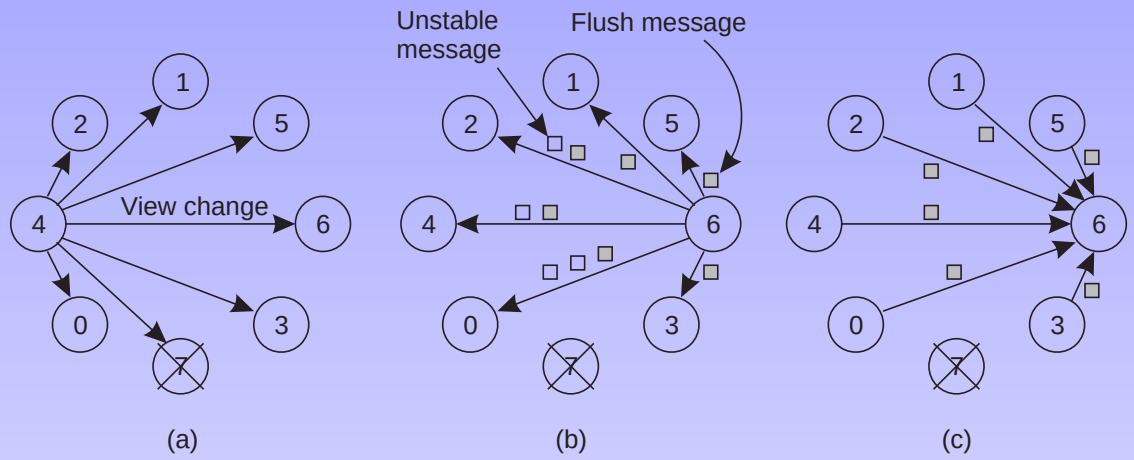
Observation: Virtually synchronous behavior can be seen independent from the ordering of messages. The only issue is that messages are delivered to an agreed upon group of receivers.

Virtually Synchronous Reliable Multicasting

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

Different versions of virtually synchronous reliable multicasting.

Virtual Synchrony – Implementation



a. 4 sees that 7 crashed and broadcasts a view change,

b. 6 sends all its unstable messages, followed by a flush message,

c. when 6 receives a flush message from all others, it installs the new view.

Distributed Commit

- √ Two-phase commit (2PC)
- √ Three-phase commit (3PC)

Essential issue: How to ensure **atomicity** of commits in a distributed computation? In other words, how to make sure, that in a group of processes either all processes introduce a change of state, or none of them do?

Two-Phase Commit (1)

Model: The client who initiated the computation acts as coordinator, the processes required to commit are the participants.

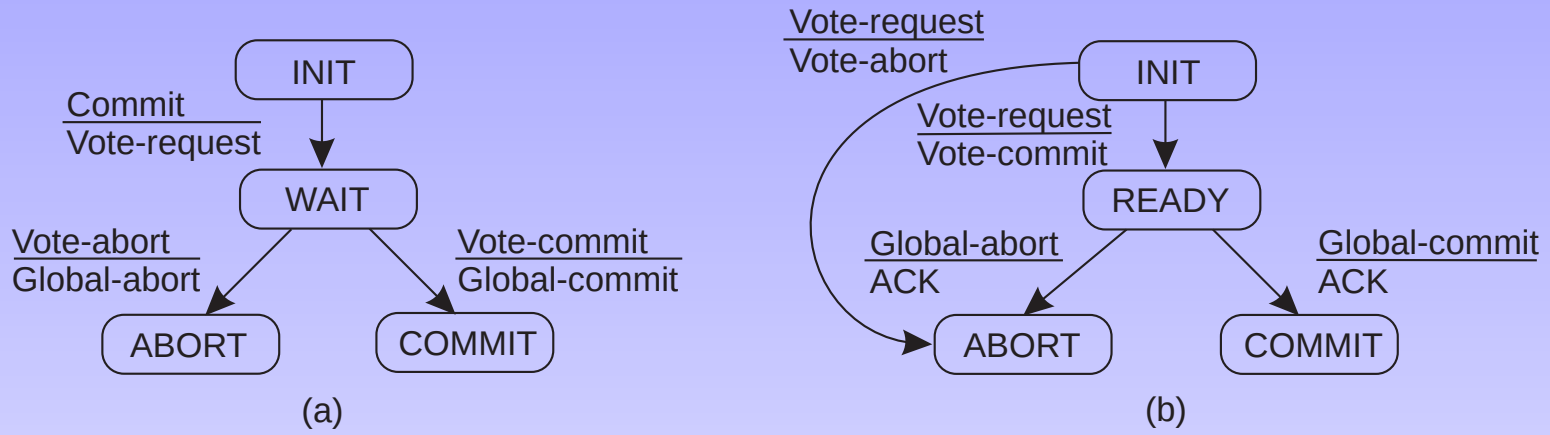
Phase 1a The coordinator sends VOTE_REQUEST to all participants (also called a pre-write).

Phase 1b When a participant receives VOTE_REQUEST it replies with a YES or NO. In the latter case the local computation is aborted.

Phase 2a The coordinator collects all votes. If all votes are YES, it sends COMMIT to all participants, otherwise it sends ABORT.

Phase 2b Participants await a COMMIT or an ABORT message and react accordingly.

Two-Phase Commit (2)



a. the finite state machine for the coordinator,

b. the finite state machine for the participant.

2PC – Failing Participant (1)

If a participant crashes in one of its states and subsequently recovers, its actions depend on the last remembered state.

INIT no problem – the participant didn't even know that the protocol started.

READY the participant is waiting for a COMMIT or an ABORT, so it must now contact the coordinator (single point of failure after the protocol has ended!) or another participant to know the decision.

ABORT just abort (entry into the abort state should be idempotent).

COMMIT just commit (entry into the commit state should also be idempotent).

Temporary workspaces make this simpler (abort/commit is idempotent in this case).

2PC – Failing Participant (2)

Contacting other participants instead of the coordinator:
A recovering participant P contacts participant Q:

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

If all participants are in the READY state, then the protocol blocks. Apparently the coordinator is failing.

2PC – Coordinator

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;
}
```

2PC – Participant

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
```

2PC – Handling Decision Requests

```
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 */
    }
```

Actions for handling decision requests (recovery protocol), executed by a separate thread.

Three-Phase Commit (1)

Problem: with 2PC, when the coordinator crashes, the participants may not be able to reach a final decision and may need to remain blocked until the coordinator recovers.

Solution: **three-phase commit protocol** (**3PC**). The states of the coordinator and each participant should satisfy the following conditions:

- √ there is no single state, from which a direct transition to either a COMMIT or ABORT state is possible,
- √ there is no single state in which it is not possible to make a final decision, but from which a transition to a COMMIT state can be made.

Note: 3PC is rarely used, as 2PC is usually good enough.

Three-Phase Commit (2)

Phase 1a The coordinator sends VOTE_REQUEST to all participants.

Phase 1b Participants respond to VOTE_REQUEST with YES or NO – in the latter case the local computation is aborted.

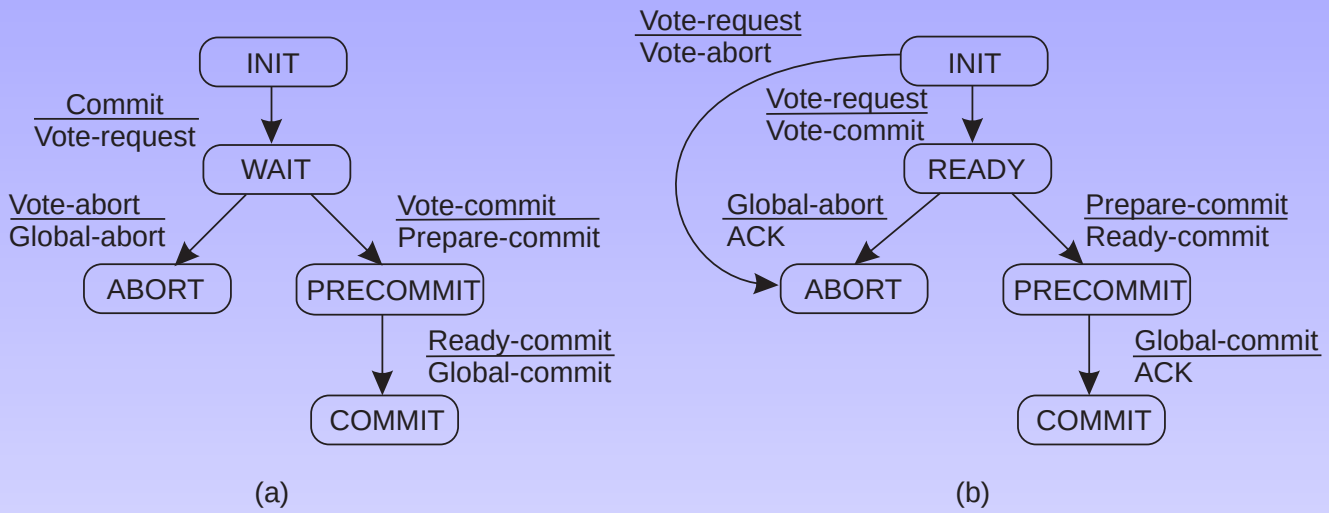
Phase 2a The coordinator collects all votes. If all participants voted YES, then PREPARE is sent, otherwise ABORT is sent and the coordinator's role ends.

Phase 2b All participants wait for PREPARE or ABORT. If PREPARE was received, it is acknowledged with ACK, otherwise the participant aborts and it's role ends.

Phase 3a The coordinator waits until all participants confirm the receipt of PREPARE, then it sends COMMIT to all participants.

Phase 3b All participants wait for COMMIT and react to it.

Three-Phase Commit (3)



a. finite state machine for the coordinator,

b. finite state machine for the participant.