Distributed Systems
Fault Tolerance

dr. Tomasz Jordan Kruk

T.Kruk@ia.pw.edu.pl

Institute of Control & Computation 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 provides services to clients. To provide services, the component may require the services from other components ⇒ a component may **depend** 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** how easy a failed system may be repaired.

For distributed systems, components can be either processes or channels.

Fault Terminology

- √ **Failure**: When a component is not living up to its specifications, a failure occurs.
- √ **Error**: That part of a component's state that can lead to a failure.
- √ **Fault**: The cause of an error.

Different fault management techniques:

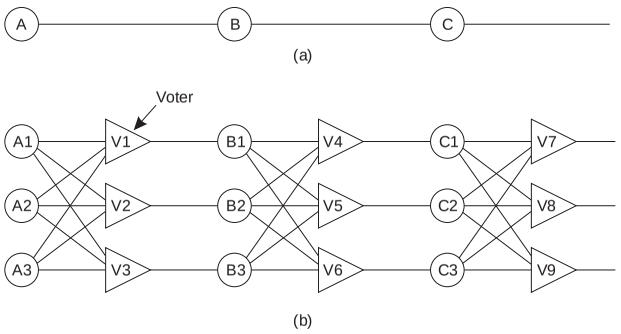
- √ **fault prevention**: prevent the occurrence of a fault,
- √ **fault tolerance**: build a component in such a way that it can meet its specifications in the presence of faults (i.e., mask the presence of faults),
- √ **fault removal**: reduce the presence, number, seriousness of faults,
- √ **fault forecasting**: estimate the present number, future incidence, and the consequences of faults.

Different Types of Failures

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

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

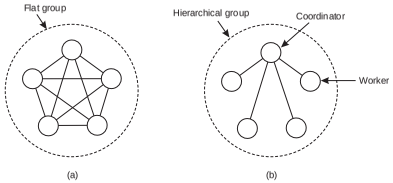
Failure Masking by Redundancy



Triple modular redundancy (TMR).

Process Resilience

Process groups: Protect yourself against faulty processes by replicating and distributing computations in a group.



a. **flat groups**: good for fault tolerance as information exchange immediately occurs with all group members. May impose more overhead as control is completely distributed (hard to implement).

b. **hierarchical groups**: all communication through a single coordinator ⇒ not really fault tolerant and scalable, but relatively easy to implement.

Groups and Failure Masking (1)

Group tolerance

When a group can mask any k concurrent member failures, it is said to be k-fault tolerant (k is called **degree of fault tolerance**).

If we assume that all members are identical, and process all input in the same order. How large does a k-fault tolerant group need to be?

√ assume *crash/performance failure semantics* ⇒ a total of $k+1$ members are needed to survive k member failures.

√ assume *arbitrary failure semantics*, and group output defined by voting ⇒ a total of $2k+1$ members are needed to survive k member failures.

Groups and Failure Masking (2)

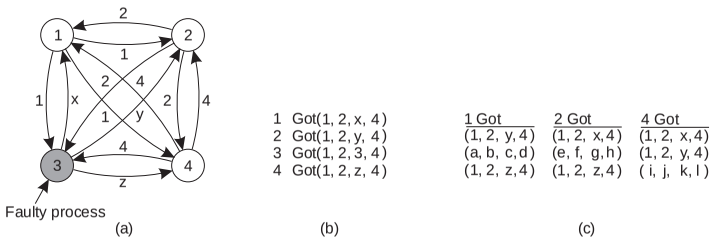
Assumption: Group members are not identical, i.e., we have a distributed computation.

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

Assuming arbitrary failure semantics, we need $3k + 1$ group members to survive the attacks of k faulty members.

We are trying to reach a majority vote among the group of loyalists, in the presence of k traitors $\Rightarrow$ we need $2k + 1$ loyalists. This is also known as **Byzantine failures**.

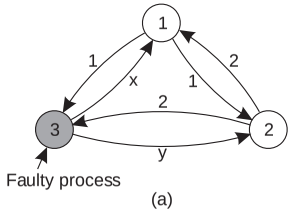
Groups and Failure Masking (3)



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

a. the generals announce their troop strengths (in thousands of soldiers),
b. the vectors that each general assembles based on (a),
c. the vectors that each general receives in step 3.

Groups and Failure Masking (4)



The same as before, except now with 2 loyal generals and one traitor.

Reliable Communication

So far concentrated on **process resilience** (by means of process groups). What about **reliable communication channels**?

Error **detection**:

- √ framing of packets to allow for bit error detection,
- √ use of frame numbering to detect packet loss.

Error **correction**:

- √ add so much redundancy that corrupted packets can be automatically corrected,
- √ request retransmission of lost, or last N packets.

Most of this work assumes point-to-point communication.

Reliable RPC (1)

What can go wrong during RPC?

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

Notes:

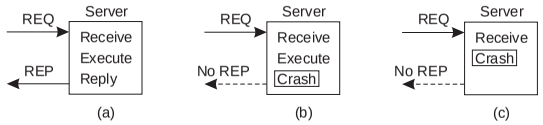
1: relatively simple - just report back to client,

2: just resend message,

3: server crashes are harder as no one knows what server had already done.

Reliable RPC (2)

If server crashes no one knows what server had already done. We need to decide on what we expect from the server.



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

Possible different RPC server semantics:

- √ **at-least-once-semantics**: the server guarantees it will carry out an operation at least once, no matter what.
- √ **at-most-once-semantics**: the server guarantees it will carry out an operation at most once.

Reliable RPC (3)

4: Detecting lost replies can be hard, because it can also be that the server had crashed. You don't know whether the server has carried out the operation.

Possible solution: None, except that one can try to make your operations **idempotent** – repeatable without any harm done if it happened to be carried out before.

5: Problem: The server is doing work and holding resources for nothing (called doing an orphan computation).

Possible solutions:

- ⋆ orphan killed (or rolled back) by client when it reboots,
- ⋆ broadcasting new epoch number when recovering ⇒ servers kill orphans,
- ⋆ requiring computations to complete in a T time units, old ones simply removed.

Reliable Multicasting (1)

Basic model: There is a multicast channel c with two (possibly overlapping) groups:

- √ the sender group $SND(c)$ of processes that submit messages to channel c,
- √ the receiver group $RCV(c)$ of 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 does not leave $RCV(c)$, m should be delivered to P.

Atomic multicast How to ensure that a message m submitted to channel c is delivered to process $P \in RCV(c)$ only if m is delivered to all members of $RCV(c)$.

Reliable Multicasting (2)

If one can stick to a local-area network, reliable multicasting is ”easy”.

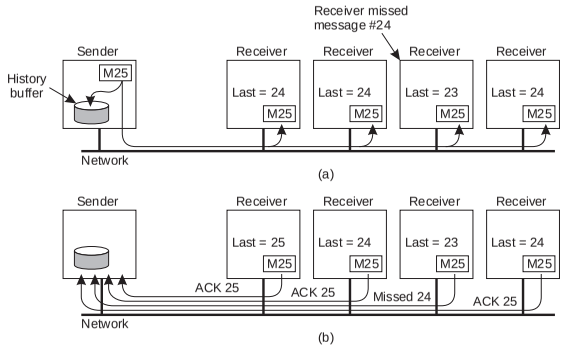
Let the sender log messages submitted to channel c:

- √ if P sends message m, m is stored in a history buffer,
- √ each receiver acknowledges the receipt of m, or requests retransmission at P when noticing message lost,
- √ sender P removes m from history buffer when everyone has acknowledged receipt.

Why doesn't this scale? The basic algorithm doesn't scale:

- √ if $RCV(c)$ is large, P will be swamped with feedback (ACKs and $NACK$s),
- √ sender P has to know all members of $RCV(c)$.

Basic Reliable-Multicasting Schemes

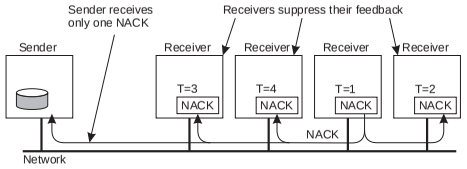


A simple solution to reliable multicasting when all receivers are known and are assumed not to fail: (a) message transmission and (b) reporting feedback.

Scalable RM: Feedback Suppression

Idea: Let a process P **suppress its own feedback** when it notices another process Q is already asking for a retransmission.
Assumptions:

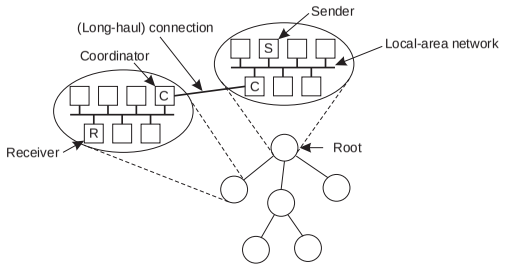
- √ all receivers listen to a common feedback channel to which feedback messages are submitted,
- √ process P schedules its own feedback message randomly, and suppresses it when observing another feedback message.
- √ random schedule needed to ensure that only one feedback message is eventually sent.



Scalable RM: Hierarchical Solutions

Idea: Construct a hierarchical feedback channel in which all submitted messages are sent only to the root. Intermediate nodes aggregate feedback messages before passing them on.

Main challenge: Dynamic construction of the hierarchical feedback channels.

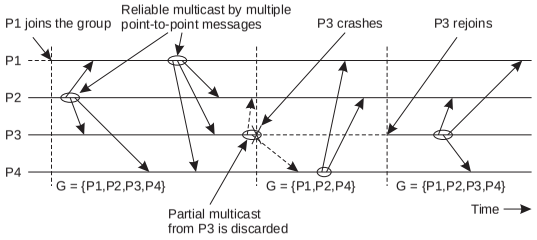


Atomic Multicast

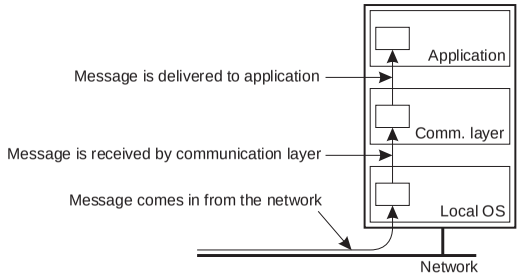
Idea: Formulate reliable multicasting in the presence of process failures in terms of process groups and changes to group membership.

Guarantee: A message is delivered only to the non-faulty members of the current group. All members should agree on the current group membership.

Keyword: Virtually synchronous multicast.



Virtual Synchrony (1)



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

Virtual Synchrony (2)

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

Processes are added or deleted from a view V through view changes to $V*$. A view change is to be executed locally by each $P \in V \cap V*$

1. for each consistent state, there is a unique view on which all its members agree. Note: implies that all non-faulty processes see all view changes in the same order,
2. if message m is sent to V before a view change vc to $V*$, then either all $P \in V$ that excute vc receive m, or no processes $P \in V$ that execute vc receive m. Note: all non-faulty members in the same view get to see the same set of multicast messages,
3. a message sent to view V can be delivered only to processes in V, and is discarded by successive views.

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

Virtual Synchrony (3)

A sender to a view V need not be member of V,

If a sender $S \in V$ crashes, its multicast message 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) processes in V before they all agree on a new view to which S does not belong

If a receiver P fails, a message m may be lost but can be recovered as we know exactly what has been received in V. Alternatively, we may decide to deliver m to members in $V - P$

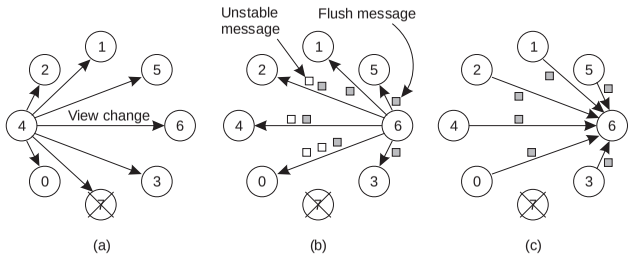
Observation: Virtually synchronous behavior can be seen independent from the ordering of message delivery. The only issue is that messages are delivered to an agreed upon group of receivers.

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.

Implementing Virtual Synchrony



a. process 4 notices that process 7 has crashed and sends a view change.
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.

Distributed Commit

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

Essential issue: Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (**atomicity**)?

Two-Phase Commit (1)

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

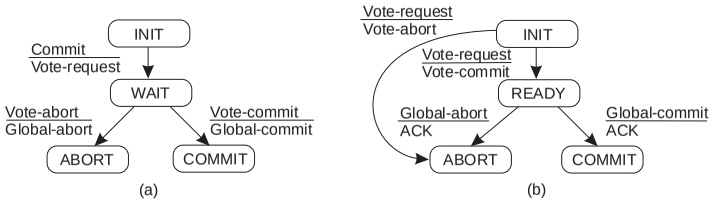
Phase 1a Coordinator sends VOTE_REQUEST to participants (also called a pre-write).

Phase 1b When participant receives VOTE_REQUEST it returns either YES or NO to coordinator. If it sends NO, it aborts its local computation.

Phase 2a Coordinator collects all votes; if all are YES, it sends COMMIT to all participants, otherwise it sends ABORT.

Phase 2b Each participant waits for COMMIT or ABORT and handles accordingly.

Two-Phase Commit (2)



a. the finite state machine for the coordinator in 2PC,
b. the finite state machine for a participant.

2PC – Failing Participant (1)

Consider participant crash in one of its states, and the subsequent recovery to that state:

initial state no problem, as participant was unaware of the protocol,

ready state participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make → log the coordinator's decision,

abort state merely make entry into abort state idempotent, e.g., removing the workspace of results,

commit state also make entry into commit state idempotent, e.g., copying workspace to storage.

When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.

2PC – Failing Participant (2)

Alternative: When a recovery is needed to the Ready state, check what the other participants are doing. This approach avoids having to log the coordinator's decision.
Assume recovering participant P contacts another 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

Result: If all participants are in the ready state, 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 executed by separate thread.

Three-Phase Commit (1)

Problem: with 2PC when the coordinator crashed, 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 satisfy the following conditions:

- √ there is no single state from which it is possible to make a transition directly to either a COMMIT or ABORT state,
- √ 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.

Note: not often applied in practice as the conditions under which 2PC blocks rarely occur.

Three-Phase Commit (2)

Phase 1a Coordinator sends VOTE_REQUEST to participants

Phase 1b When participant receives VOTE_REQUEST it returns either YES or NO to coordinator. If it sends NO, it aborts its local computation

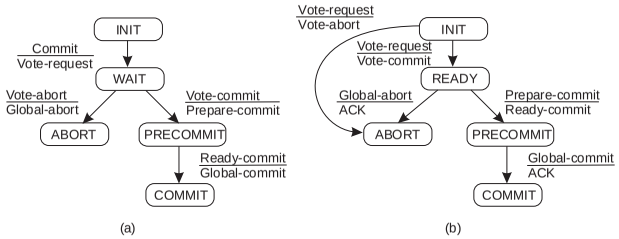
Phase 2a Coordinator collects all votes; if all are YES, it sends PREPARE to all participants, otherwise it sends ABORT, and halts

Phase 2b Each participant waits for PREPARE, or waits for ABORT after which it halts

Phase 3a (Prepare to commit) Coordinator waits until all participants have ACKed receipt of PREPARE message, and then sends COMMIT to all

Phase 3b (Prepare to commit) Participant waits for COMMIT

Three-Phase Commit (3)



a. finite state machine for the **coordinator** in 3PC,
b. finite state machine for the **participant**.