Distributed Systems

Fault Tolerance

Abderraouf Gharbi
(Angleichungsleistungen)
Based on: Distributed Systems: Principles and Paradigms – Andrew Tanenbaum

Agenda

- Introduction to Fault Tolerance
- Process Resilience
- Client/Server communication constancy
- Group communication constancy
- Distributed commit
- Recovery

Introduction to Fault Tolerance

Introduction to Fault Tolerance:
Basic Concepts

- Availability
- Reliability
- Safety
- Maintainability

Introduction to Fault Tolerance:
Failure Models

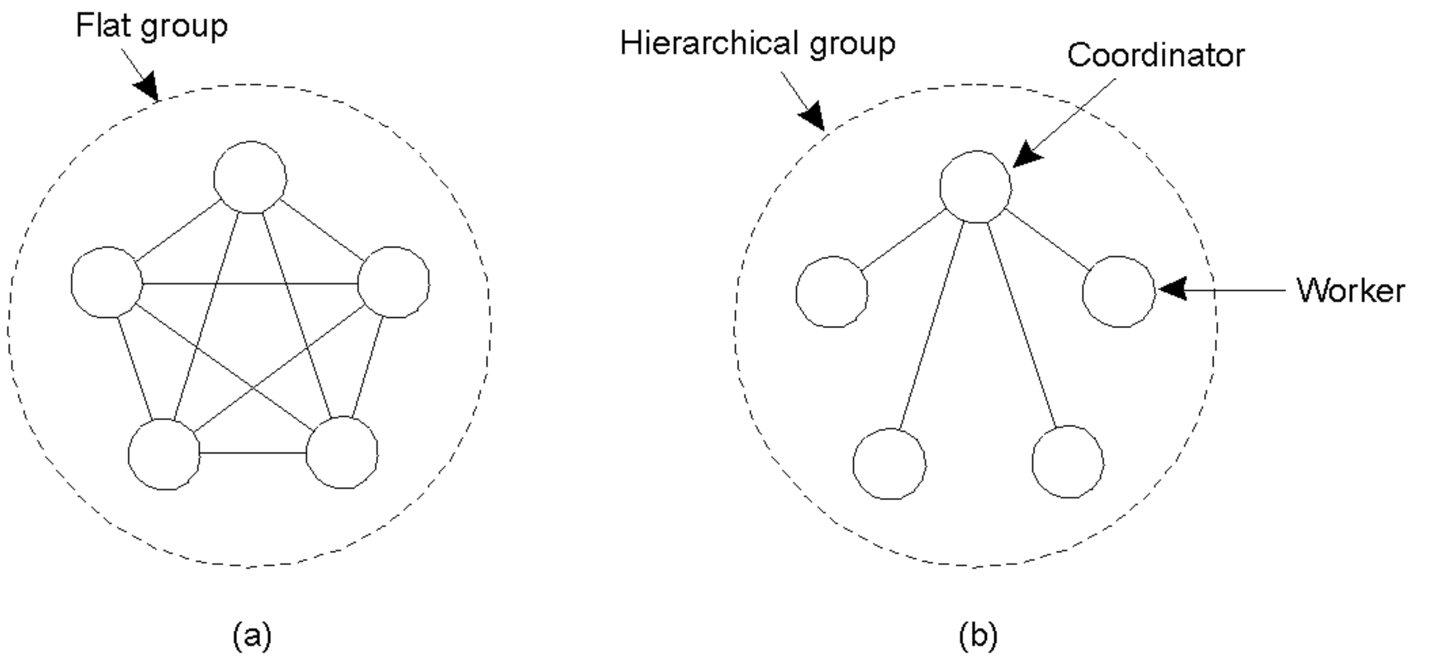
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response 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

Introduction to Fault Tolerance:
Failure Masking Redundancy

- Information redundancy
  - Hamming-Codes
- Time redundancy
  - Timeout & Retransmission
- Physical redundancy
  - TMR, RAID

Process resilience

Process resilience:
Flat and Hierarchical Groups(1)



Communication in a flat group.

Communication in a simple hierarchical group

Process resilience:
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 resilience:
Failure Detection

Issue: How can we **reliably** detect that a process has **actually crashed**?

- General model:
  - Each process is equipped with a **failure detection module**
  - A process p probes another process q for a reaction
  - q reacts → q is **alive**
  - q does not react within t time units → q is **suspected** to have **crashed**

Client/Server communication constancy

Client/Server communication constancy:
Reliable Remote Procedure Call (RPC)

INF Informatik

RPC communication: What can go wrong?

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

RPC communication: Solutions

1. Report back to client
2. Resend message

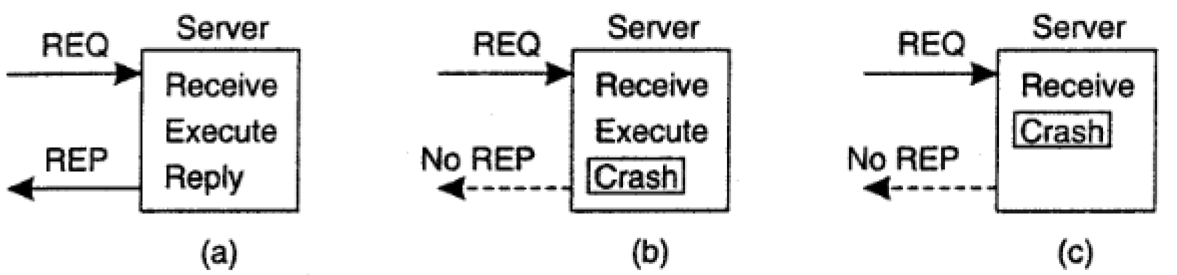
*A **stub*** in distributed computing **is a piece of code** that converts parameters passed between client and server during a remote procedure call (RPC).

Client/Server communication constancy: **Reliable RPC**

RPC communication: Solutions

Server crashes

- server crashes are harder as you don't what it had already done.



→ We need to decide on what we expect from the server:

- ❖ At-least-once-semantics
- ❖ At-most-once-semantics

Client/Server communication constancy: **Reliable RPC**

RPC communication: Solutions

Server response is lost

- 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 opperation

→ One way of solving this problem is to try to structure all the requests in an idempotent way!

RPC communication: Solutions

Client crashes

- What happens if a client sends a request to a server to do some work and crashes before the server replies?

→ The server is doing work and holding resources for nothing! (called doing on **orphan** computation)

→ Orphan is killed (or rolled back) by client when it reboots

→ Broadcast new epoch number when recovering → servers kill orphans

Group communication constancy

Group communication constancy: Reliable-Multicasting(1)

Basic model:

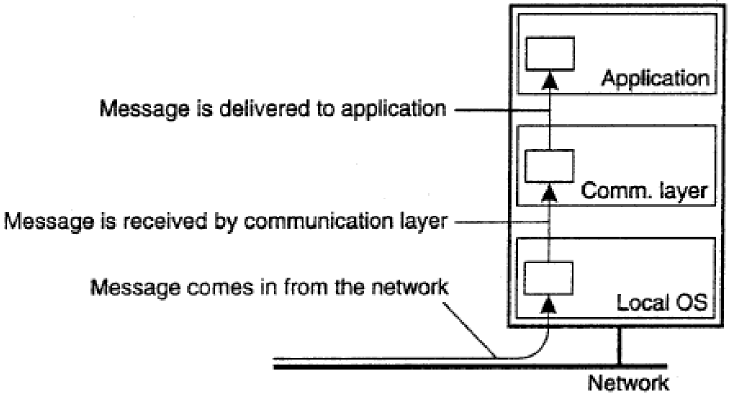
We have a **multicast channel** c with two (possibly overlapping) groups:

- **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

Group communication constancy:
Reliable-Multicasting(2)

- **Simple reliable:** if process P ∈ RCV(c) at the time messages m was submitted to c, and P does not leave RCV(c), m should be delivered to P

- **Atomic multicast:** How can we ensure that a message m submitted to channel c is delivered to process P ∈ RCV(c) only if m is delivered to all members of RCV(c)



Group communication constancy:
Reliable-Multicasting(3)

Principle:

Let the sender log messages submitted to channel c:

- If P sends messafes m, m is stored in a **history buffer**
- Each reveiver 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

Distributed commit

Distributed commit:
Two-Phase Commit

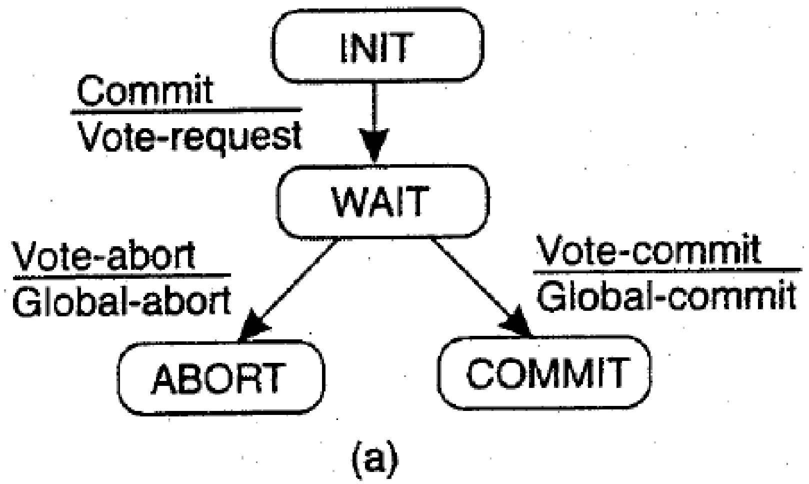
Model:

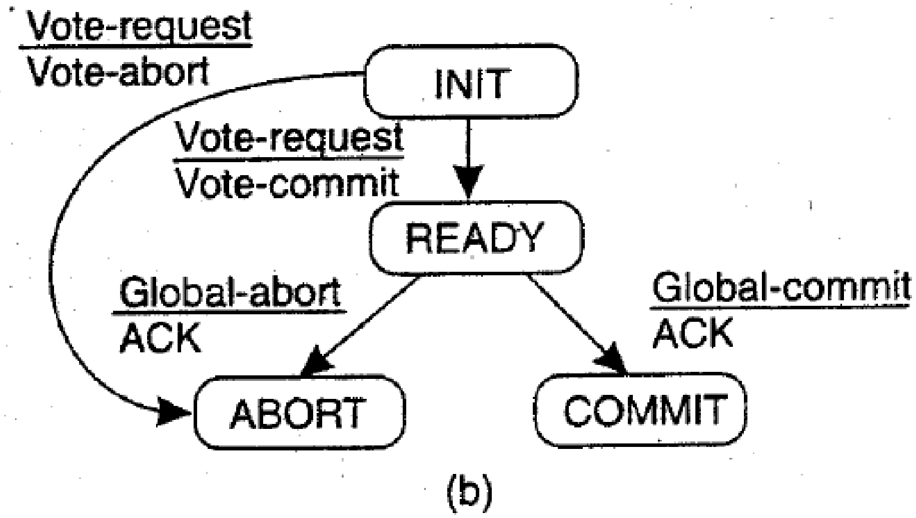
The client who initiated the computation acts as coordinator, processes required to commit are the participants

- **Phase 1a:** Coordinator sends vote-request to participants (also called a **pre-write**)
- **Phase 1b:** When participant receives vote-request it returns either vote-commit or vote-abort to coordinator. If it sends vote-abort, it aborts ist local computation

- **Phase 2a:** Coordinator collects all votes, if all are vote-commit, it sends global-commit to all participant, otherwise it sends global-abort
- **Phase 2b:** Each participant waits for global-commit or global-abort and handles accordingly.

Distributed commit:
Two-Phase Commit



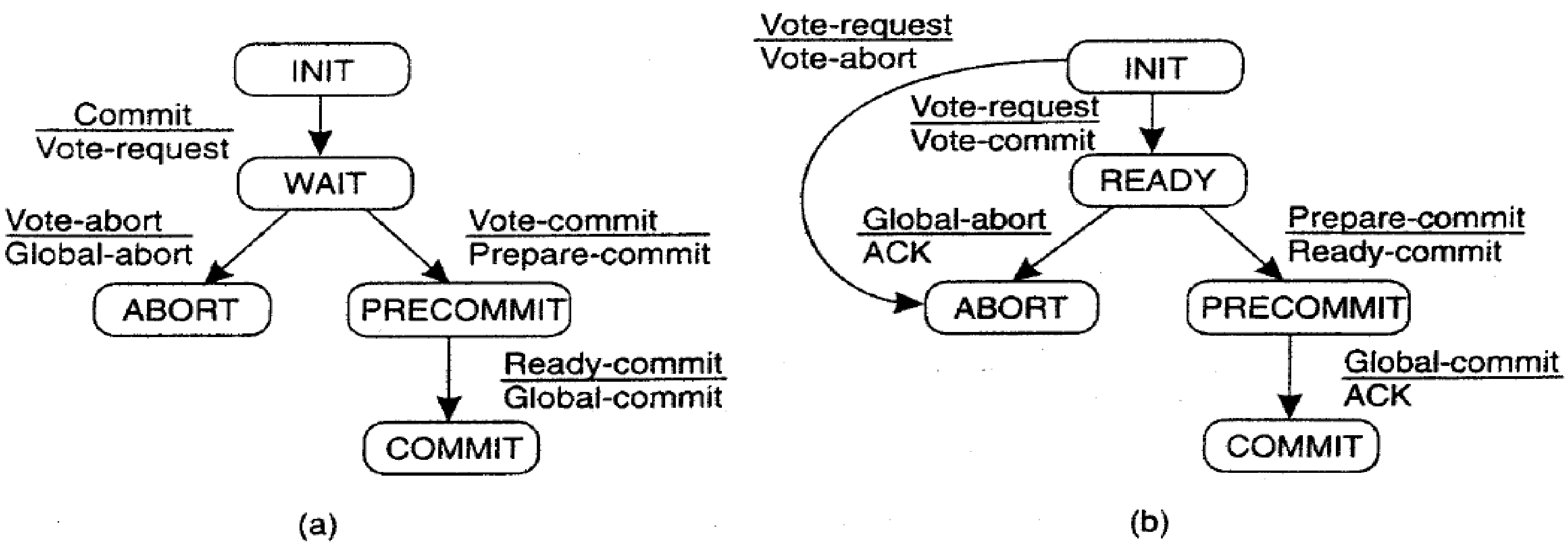


Coordinator

Participant

Distributed commit:
Three-Phase Commit

→ A problem with the two-phase commit protocol is that when the **coordinator** has **crashed**, participants may not be able to reach a final decision.



Recovery

Recovery:
Introduction

When a failure occurs, we need to bring the system into an error-free state:

- **Forward error recovery**: Find a new state from which the system can continue operation
- **Backward error recovery**: Bring the system back into a previous error-free state

→ Use backward error recovery, requiring that we etablish **recovery points**

Recovery:
Independent Checkpointing

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let CP[i](m) denote m^{tH} checkpoint of process Pi and INT[i](m) the interval between CP[i](m - 1) and CP[i](m)
- When process Pi sends a message in interval INT[i](m), it piggybacks (i,m)
- When process Pj receives a message in interval INT[j](n), it records the dependency INT[i](m) → INT[j](n)
- The dependency INT[i](m) → INT[j](n) is saved to stable storage when taking checkpoint CP[j](n)

Recovery:
coordinated Checkpointing

Use a two-phase blocking protocol:

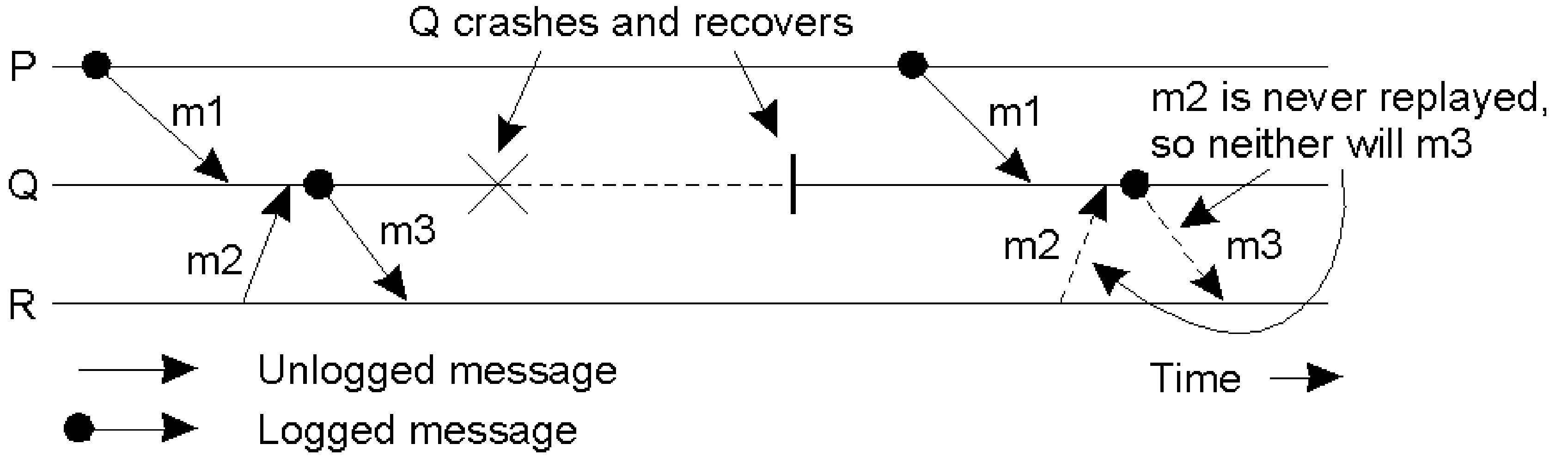
- A coordinator multicasts a checkpoint request message.
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and repots back that it has taken a checkpoint.
- When all checkpoints have been confirmed at the coodinator, the latter broadcasts a checkpoint done message to allow all processes to continue.

Recovery:
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.



Thank you!