Verteilte Systeme
(Distributed Systems)

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at

http://www.infosys.tuwien.ac.at/teaching/courses/
VerteilteSysteme/
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Dependability

What it should have been like

What actually happened
Dependability and trust

- Goal: dependable and secure systems
- The problem (and opportunity) of partial failures
- Tolerating, detecting and recovering from failures
  - Process failures
  - Communication failures
- Reliable communication
  - Client-server communication
  - Group communication and group membership
System boundaries and interaction

- System boundary: system ↔ environment
- System properties:
  - Functional specification: Functionality and performance
  - Behavior: Sequence of states
- Structure: set of (atomic) components
- Service: Behavior as perceived by the user (at the service interface)
- External state: perceivable at the service interface
- → service is a sequence of external states
Dependability

The ability of a system to deliver service that can justifiably be trusted.

The ability of a system to avoid service failures that are more frequent and more severe than is acceptable.
Dependability and security tree

Dependability and Security

Attributes
- Availability
- Reliability
- Safety
- Confidentiality
- Integrity
- Maintainability

Threats
- Faults
- Errors
- Failures

Means
- Fault Prevention
- Fault Tolerance
- Fault Removal
- Fault Forecasting
Dependability Attributes

- **Availability**: Readiness for correct service (usage): system is ready to be used immediately; probability of correct functioning at any given moment in time.

- **Reliability**: Continuity of correct service; system runs continuously over a period of time without failure.

- **Safety**: Absence of catastrophic consequences on the user(s) and the environment.

- **Integrity**: Absence of improper system alterations.

- **Maintainability**: Ability to undergo modifications and repairs.
Security Attributes

- **Availability**: For authorized actions only.
- **Confidentiality**: Absence of unauthorized disclosure of information.
- **Integrity**: Absence of unauthorized system alterations.
The dependability and security specification of a system must include the requirements for the attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and a given use environment.
Threats: Failure

- Failure (Ausfall, Versagen): Event that occurs, when the delivered service deviates from correct (expected/useful) service.
  - Service not compliant with functional specification.
  - Specification does not adequately describe the system function (Uncovers specification faults; subjective and disputeable). → Service outage → service restoration.

- Partial failure → degraded mode.

- Failure cannot be observed easily, usually deduced by error detection or detected by reliable failure detector.
Threats: Error

- Service is sequence of external states!
- Error (Fehler, Abweichung): The part of a system’s total state that may lead to a subsequent service failure – a failure occurs, when the error causes the delivered service to deviate from correct service.
- → observable (external) state, (e.g. message is damaged in transmission) that deviates from the correct service state.
- Detected vs. latent error.
- Many errors do not cause a failure!
Threats: Fault

- Fault (Mangel, Defekt): Adjudged or hypothesized cause of an error (state).
- A (design, programming, manufacturing) defect, that has the potential to generate errors
- Faults can be internal or external: The presence of a vulnerability (internal fault) is necessary for an external fault to cause an error.
- Faults can be dormant or active.
- Goal of debugging is to find the faults. When there is a failure, we try to find the errors (which can be observed) and then trace to the fault(s)
Chain of dependability threats

\[ \text{fault} \xrightarrow{\text{activation}} \text{error} \xrightarrow{\text{propagation}} \text{failure} \xrightarrow{\text{causation}} \text{fault} \xrightarrow{\text{propagation}} \ldots \]

- fault → error
  - a fault which has not been activated by the computation process is dormant
  - a fault is active when it produces an error
- error → failure
  - an error is latent when it has not been recognized
  - an error is detected by a detection algorithm/mechanism
- failure → fault
  - a failure occurs when an error “passes through” and affects the service delivered
  - a failure results in a fault for the system which contains or interacts with the component

Propagation can occur via interaction, composition, creation, and modification
Service failure of component A causes an permanent or transient fault in the system that contains A. It causes an external fault for component B that receives service from A. This fault in B may be activated and lead to error propagation in B.
Means: Fault Control (1)

- Procurement: Ability to deliver a service that can be trusted.
  - Fault prevention (avoidance): Prevent the occurrence or introduction of faults, e.g. QM, methods, design rules like formalism or design diversity, ...
  - Fault tolerance: Avoid service failure in the presence of faults.
Means: Fault Control (2)

- Validation: Reach **confidence** in that (procurement) ability by justifying that the functional, dependability, and security specifications are adequate and the system is likely to meet them.
  - **Fault removal (error removal):** Reduce the number and severity of faults, e.g. verification (static and dynamic analysis), diagnosis, correction
  - **Fault forecasting (error forecasting):** Estimate the present number, the future incidence, and the likely consequences of faults, e.g. evaluation, statistical methods, ...
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Techniques

- Fault tolerance techniques
- Security techniques
- Hardware and IT Infrastructure
  - Virtualization (VM, GRID, and also SOA)
- Maintenance
- Software development methods, tools, and techniques
- Emerging techniques
Fault tolerance techniques

- persistence (databases)
- replication
- group membership and atomic broadcast
- transaction monitors
- reliable middleware with explicit control of quality of service properties
Security techniques

- cryptology
- hardware support (RFID, embedded systems)
- tamper-proof hardware (smart cards)
- privacy and identity policies
- digital rights management
Hardware and IT Infrastructure

- Various interfaces offered by computer systems → Virtual machines
- Sharing of resources on a very large scale (mainly data or computer power for data-intensive applications) → GRID computing
- Computing Power as a configurable, payable Service → Cloud computing
Heterogeneous Resources

Distributed physical clusters and storage
The Grid: Virtualizing Resources

Service “Bus” as GRID middleware

Virtual clusters and storage
Cloud Computing

Computing Power as a configurable, payable Service
Maintenance

Repairs
- Removal of reported faults
  - Corrective Maintenance
- Discovery and removal of dormant faults
  - Preventive Maintenance

Modifications
- Adjustment to environmental changes
  - Adaptive Maintenance
- Augmentation of system’s function
  - Augmentive Maintenance
Software development

- Defects in software products and services ...
  - may lead to failure
  - may provide typical access for malicious attacks

→ The process has to ensure correctness:

Requirements are the things that you should discover before starting to build your product. Discovering the requirements during construction, or worse, when your client starts using your product, is so expensive and so inefficient, that we will assume that no right-thinking person would do it, and will not mention it again.

Robertson and Robertson
Mastering the Requirements Process
... but reality is different

Walking on water and developing software from a specification are easy

– if both are frozen

Edward V. Berard
Life Cycle Approaches
Requirements...

- ... do change – continuously!
- ... are incomplete, so we have to retrofit originally omitted requirements
- ... are competing or contradictory (due to inconsistent needs)
- Many users are inarticulate about precise criteria
- Trade-offs change as well
- Domain know-how changes
- Technical know-how changes
- Complexity may result in emerging properties
Answer on the process level

- Design for change in highly volatile areas!
- Heavy weight (CMM) $\rightarrow$ light weight (ASD) processes
- Development in-the-small: Component, service,... $\rightarrow$ agile development (ASD, XP), MDA, AOP, ...
- Development in-the-large: Procurement/discovery, re-use, composition, generation, deployment, ... $\rightarrow$ Product line, EAI, CBSE, (MDA), SOA, ...
In an extreme environment, following a plan produces the product you intended, just not the product you need.
EAI: Software Cathedral

- Robust, long Lifecycle
- Co-Existent of diverse different Technologies
- dynamic, extensible
- Re-usable Designs
- Based on a common Framework-Architecture
Component-based Software Engineering

„Buy before build. Reuse before buy“
Fred Brooks 1975(!)

Components: CBSE and Product Lines
Components of Mercedes E class cars are 70% equal. Components of Boeing 757 and 767 are 60% equal. → most effort is integration instead of development!

Quality, time to market, but **complexity** → re-use
SOA is an evolution, not a revolution

- EAI – Enterprise Application Integration (MoM)
  (note: Was an argument for CBSE as well)
- WfMS – Workflow Management Systems → BPEL
- CBSE – Components are not obsolete!
  → SOA provide a virtual component model
- WWW – Loose coupling: Heterogeneous, flexible, and
dynamic orchestration
- Re-use (note: Was an argument for CBSE,
  Middleware, ...)
- Interface management (note: -“-)
- Business integration („business goals with IT“)
So, when is software finished?

- Never – as long as it is needed!
- Change (short-/long-term) of ...
  - the system itself (e.g., resource variability)
  - the context (environment, new faults/vulnerabilities)
  - users’ needs and expectations (requirements)
- Uncertainty
  - Contradictory or inconsistent needs (requirements)
- Complexity and emerging behaviour
  - Interactions and interdependencies prevail properties of a systems’ constituents
Emerging techniques

- Control loop
  - adaptiveness
  - self-properties
  - autonomous computing

- Software evolution
  - convergence of design-time and run-time
  - run-time software development

- architectural dependability (e.g., P2P systems)

- bio-inspired methods
Summary

- Distributed systems can suffer partial failures
- Distributed systems can provide fault-tolerance
- Taxonomy and Chain of Threats
- Design-time/run-time convergence

Next lecture →

- Faults can be due to process failures or communication failures
- Process replication (process groups) can help deal with process failures
- Reliable group communication supports the construction of fault-tolerant systems
Verteilte Systeme
(Distributed Systems)

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at

http://www.infosys.tuwien.ac.at/teaching/courses/
VerteilteSysteme/
Dependability and security tree (rep’d)
Service failure of component A causes an permanent or transient fault in the system that contains A. It causes an external fault for component B that receives service from A. This fault in B may be activated and lead to error propagation in B.
So, when is software finished? (rep‘d)

- Never – as long as it is needed!
- Change (short-/long-term) of ...
  - the system itself (e.g., resource variability)
  - the context (environment, new faults/vulnerabilities)
  - users’ needs and expectations (requirements)
- Uncertainty
  - Contradictory or inconsistent needs (requirements)
- Complexity and emerging behaviour
  - Interactions and interdependencies prevail properties of a systems‘ constituents
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Fault classes (1)

- **Phase of creation or occurrence**
  - Development faults
    [occur during (a) system development, (b) maintenance during the use phase, and (c) generation of procedures to operate or to maintain the system]
  - Operational faults
    [occur during service delivery of the use phase]

- **System boundaries**
  - Internal faults
    [originate inside the system boundary]
  - External faults
    [originate outside the system boundary and propagate errors into the system by interaction or interference]

- **Phenomenological cause**
  - Natural faults
    [caused by natural phenomena without human participation]
  - Human-Made faults
    [result from human actions]

- **Dimension**
  - Hardware faults
    [originate in, or affect, hardware]
  - Software faults
    [affect software, i.e., programs or data]
Fault classes (2)

- **Objective**
  - Malicious faults
    [introduced by a human with the malicious objective of causing harm to the system]
  - Non-Malicious faults
    [introduced without a malicious objective]

- **Intent**
  - Deliberate faults
    [result of a harmful decision]
  - Non-Deliberate faults
    [introduced without awareness]

- **Capability**
  - Accidental faults
    [introduced inadvertently]
  - Incompetence faults
    [result from lack of professional competence by the authorized human(s), or from inadequacy of the development organization]

- **Persistence**
  - Permanent faults
    [presence is assumed to be continuous in time]
  - Transient faults
    [presence is bounded in time]
Combinations

- 8 basic viewpoints → 256 combinations
- of which 31 are likely
- grouped into three major (overlapping) groups:
  - Development faults: Software defects, hardware flaws, software aging, dependability degradation, dependability gap, legacy integration, ...
  - Physical faults: Production defects, physical deterioration/interference, hardware flaws, ...
  - Interaction faults (including all external faults): Wrong input, viruses, worms, intrusion attempts, physical interference

- system level (failure → fault)
  - node, link, partition
Another fault classification

- **Transient Faults**
  - Occurs once and then disappears
  - If the operation is repeated, the fault goes away
  - Detection may not be always necessary
  - E.g.: A Bird flying through a beam of a microwave transmitter
  - BUT: A transient fault can lead to a permanent error!

- **Permanent Faults**
  - continues to exist, until the faulty component is repaired
  - E.g.: Burnt-out chips, Software Bugs, Disk head crashes

- **Intermittent Faults**
  - Appears, disappears, reappears, ...
  - E.g. A loose contact on a connector
  - Difficult to diagnose
Fault activation reproducability

Ability to identify the activation pattern

Permanent faults (development, physical, interaction)

<table>
<thead>
<tr>
<th></th>
<th>Transient faults (physical, interaction)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elusive faults</td>
</tr>
<tr>
<td></td>
<td>Solid faults</td>
</tr>
<tr>
<td></td>
<td>Intermittent faults</td>
</tr>
</tbody>
</table>
Service failure modes

Failure modes of detecting mechanisms:
- false alarm / undetected

Domain
- Content failures
  - Early timing failures
  - Late timing failures
  - Halt failures
  - Erratic failures
- Signaled failures
  - Unsignedaled failures
- Consistent failures
  - Inconsistent failures
- Minor failures
  - Catastrophic failures

Detectability
- Consistency
- Consequences

Relation between benefit and consequences
Failure domain viewpoint

Content (Correct timing)
- Content Failures
- Early Timing Failures

Timing (Correct content)
- Early service
- Late service
- Late timing Failures

Content and Timing
- Halted service
- Halt Failures
- Erratic service
- Erratic Failures

Silence is a special form of halt
Fail-controlled systems

- Fail-halt (fail-stop) system: Halting failures only. → Often: halting can be detected.
- Fail-passive (fail-silent) systems: Stuck output instead of erratic output (Silence as opposed to babbling). Often crash failures.
  - Other processes may incorrectly conclude that a server has halted whereby the server is only unexpectedly slow!
- Fail-consistent system: No byzantine failures.
- Fail-inconsistent system: Any type of failure.
- Fail-safe system: All minor failures, no catastrophic consequences expected.
Other failure models (Tanenbaum)

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts</td>
</tr>
<tr>
<td>Omission failure</td>
<td>A server fails to respond to incoming requests</td>
</tr>
<tr>
<td>Receive omission</td>
<td>A server fails to receive incoming messages</td>
</tr>
<tr>
<td>Send omission</td>
<td>A server fails to send messages</td>
</tr>
<tr>
<td>Timing failure</td>
<td>A server's response lies outside the specified time interval</td>
</tr>
<tr>
<td>Response failure</td>
<td>The server's response is incorrect</td>
</tr>
<tr>
<td>Value failure</td>
<td>The value of the response is wrong</td>
</tr>
<tr>
<td>State transition failure</td>
<td>The server deviates from the correct flow of control</td>
</tr>
<tr>
<td>Inconsistent failure (Byzantine)</td>
<td>A server may produce two-faced responses at arbitrary times</td>
</tr>
</tbody>
</table>

Failure Effect: benign or malign (safety-critical)
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Fault Tolerance (one of the means)

- A system is fault tolerant, if service failure can be avoided when faults are present in the system.
- FT needs redundancy.
- Generic vs. application-specific.
- Fault tolerance as opposed to a system whose individual components are highly reliable, but whose organization is not fault tolerant.
- Levels: System made FT against failure of its components (masks the failure of a subsystem at higher levels)
  → Fault/failure chain (e.g. network layers)
Fault tolerance techniques

1. Error Detection
   [identifies the presence of an error]

2. Error Handling
   [eliminates errors from the system state]

3. Fault Handling
   [prevents faults from being activated again]

Concurrent Detection
[takes place during normal service delivery]

Preemptive Detection
[takes place while normal service delivery is suspended; checks the system for latent errors and dormant faults]

Rollback
[brings the system back to a saved state that existed prior to error occurrence; saved state: checkpoint]

Rollforward
[state without detected errors is a new state]

Compensation
[the erroneous state contains enough redundancy to enable error to be masked]

Diagnosis
[identifies and records the cause(s) of error(s), in terms of both location and type]

Isolation
[performs physical or logical exclusion of the faulty components from further participation in service delivery, i.e., makes the fault dormant]

Reconfiguration
[either switches in spare components or reassigns tasks among non-failed components]

Reinitialization
[checks, updates and records the new configuration and updates system tables and records]
Strategies for Fault Tolerance

- (Error) Detection and (system) recovery (on demand)
  - Backward recovery: Reset to a stored error-free system state (e.g., database rollback)
  - Forward recovery: Set to a new error-free system state (e.g., real-time systems)

- Fault masking and recovery
  - masking through systematic use of compensation
  - masking only may lead to loss of redundancy
  - → error detection (and possibly fault handling) eventually necessary

- Homeostasis (no detection, ongoing recovery)
Failure masking by redundancy (1)

- Redundancy is the key for fault-tolerance. There can be no FT without redundancy!
- Redundancy = those parts that are not needed for correct functioning, if no FT is provided:
  - Information: e.g. Hamming code
  - Time: operations are performed repeatedly (e.g. with transient or intermittent faults): e.g. message re-send
  - Physical
    - Hardware
    - Software: Processes, Data, including the replica management instructions
- Biology: 2 eyes, lungs, ... (true redundancy?)
Fault Tolerance

Fault-tolerance in the domain of space

- active redundancy
  - parallel fail-silent components

- passive or standby redundancy
  - hot standby:
    standby component is operating
  - cold standby:
    standby components starts only in case of a failure

- voting, triple modular redundancy (TMR)
Failure masking by redundancy (2)

Triple modular redundancy (TMR)

(a)

(b)
Process Resilience

- Dealing with process failures: As in hardware, we can introduce redundancy to cope with process failures.

- Process groups: Replace single process with a group of replicated processes in order to mask faulty processes.
  - Addressing
  - Communication
  - Membership

- As long as a sufficient number of processes are present in a group, service can be provided despite faults in some processes. The non-faulty processes must agree on the result.
Process replication

- How to replicate *processes*?
  - **Primary-based**: primary-backup, hierarchical group (primary = coordinator): if primary crashes, backup starts election — slow failover
  - **Replicated-write**: quorum based or active replication, flat group, no single point of failure, but expensive distributed coordination
Failure masking

- How much redundancy (k fault tolerant)?
  - fail-stop or fail-silent: k+1
  - fail-passive (fail-consistent) with or w/o distributed agreement: 2k+1
  - arbitrary (malicious, two-faced, byzantine) without distributed agreement: 2k+1
  - byzantine (arbitrary failures, malicious, two-faced) with distributed agreement: 3k+1
  - It is therefore wise to provide enough error-detection logic inside a component to guarantee fail-silent behaviour at the system level!

- How can k be estimated???
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Agreement (consensus)

- e.g. electing a coordinator, commit a transaction, divide up tasks among workers, synchronize
- Goal: have all non-faulty processes reach and establish consensus
- Depending on
  - communication reliability
  - crash-failure semantics of processes
  - possibility of failure detection
  - degree of clock synchronization
Synchronous vs. asynchronous

- Synchronous system model
  - Known bound on message transmission delay
  - Processors execute in locksteps

- Asynchronous system model
  - No fixed upper bound on message transmission delay
  - No fixed bound on how much time elapses between consecutive steps of a processor

- Synchronous model allows correct (deterministic) crash detection while asynchronous model does not!
Agreement (consensus) problems

1. Synchronous, reliable communication, but processes exhibit arbitrary failure (including omission) ꔷ Byzantine generals problem.

2. Synchronous system, perfect processes, but communication unreliable ꔷ Two army (coordinated attack) problem.

3. Asynchronous communication, communication reliable, but arbitrarily slow (individual messages can be delayed). At least one process may fail (silently). ꔷ FLP
Three byzantine generals

Faulty processes are shown shaded

- Communication pairwise, reliable, instantaneous (e.g. phone call)
- Traitors may actively prevent loyal generals from reaching agreement by feeding incorrect and contradictory information
Four byzantine generals

- 3m+1 processes are needed for agreement with m faulty processes (using unsigned messages)
- Recursive algorithm is quite expensive (m+1)
The two-army problem:
1. Sparta and Carthage together can beat Bad guys but not individually. Therefore, they have to decide to attack at exactly the same time.
2. Sparta general sends a message to Carthage general to attack at noon.
3. How does he know that Carthage general agrees?
Impossibility of asynchronous consensus

- „FLP“ – Fischer, Lynch, Paterson 1985:
  It is impossible to design a deterministic consensus algorithm in an asynchronous distributed system subject to even a single process crash failure.

- Any protocol guaranteed to produce only correct outcomes, can be indefinitely delayed by a complex pattern of link failures.

- To guarantee progress one needs:
  - higher quality of the communication line
  - a degree of clock synchronization (long timeout helps with high probability, but slows down the system)
  - accurate *enough* failure detection
Agreement (consensus) summary

- In an asynchronous system, no algorithm can guarantee agreement (consensus) if either
  - one process can be faulty (fails silently) [FLP] or
  - the channel is unreliable (two army problem),
- because arbitrarily slow processes (or channels) are indistinguishable from crashed ones.
- Generally, many results are known, when agreement is possible and when not.
- Techniques in practice include: Masking faults, failure detectors, partial/nearly synchronous, randomization.
What can we do?

- **Masking faults**: e.g. *persistent* storage to survive crash failure → transactions. Crashed process behaves like a correct but sometimes slow process (restart).

- **Consensus using failure detectors**: e.g. timeout, remaining processes agree that some (e.g., slow) process „failed“. Effectively, an asynchronous system can be turned into a synchronous one with a proper failure detection subsystem.

- „Nearly“ synchronous: e.g., read, process, and write the network in one atomic step (plus bounded communication and multicast) → „critical section“ without interrupt

- **Consensus using randomization**: „Adversary“ is hindered by an element of chance. → probabilistic algorithms

- **Live with uncertainty**: oK in many cases!
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Reliable Client/Server Communication

- Faulty processes
- Communication failures (channel)
  - focus is on crash and omission
  - also: timing or arbitrary (e.g. duplicate message)
- Point-to-point communication
  - reliable transport protocol, e.g. TCP (masks omission)
  - BUT: connection crash often not masked (exception, new connection setup – perhaps automatically)
- Higher level communication facilities: RMI and RPC semantics (communication transparency in the presence of failures?)
Failure classes in C/S communication

1. Binding: Client cannot locate server
2. Client request is lost
3. Server crashes after receiving request
4. Reply message is lost
5. Client crashes after sending request
1. Client cannot locate server

- e.g. server down, wrong client stub (older version)
- Exceptions
  - not in every language
  - destroys transparency

2. Lost request message

- Timer, expiry (no ACK), retransmission
- may falsely result in “cannot locate server“
- retransmission detection required
3. Server crash (1)

A server in client-server communication:

a) Normal case
b) Crash after execution (client has to report failure)
c) Crash before execution (client could re-transmit)
d) ➔ Correct treatment differs, BUT: client cannot differentiate!
3. Server crash (2)

- **At least once semantics**: Try until ACK (reply)
- **At most once semantics**: Try only once, then give up immediately and report failure
- **Guarantee nothing** (easy to implement)
- **We would like**: „Exactly once semantics“, but there is no way to guarantee this.
- **Server strategies**: ACK request plus completion message just before or after issuing execution
- **Client strategies**: never re-send, always re-send, re-send only if ACK’d, re-send only if *not* ACK’d
- 3 crash orderings per server strategy
- \(3 \times 2 \times 4 = 24\) combinations to consider
3. Server crash (3)

- Different combinations of client and server strategies in the presence of server crashes.
- No combination works correctly under all possible event sequences, because after all, the client cannot know, whether the server crashed just before or after execution.

<table>
<thead>
<tr>
<th>Reissue strategy</th>
<th>Strategy M -&gt; P</th>
<th>Strategy P -&gt; M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MPC</td>
<td>MC(P)</td>
</tr>
<tr>
<td>Always</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Never</td>
<td>OK</td>
<td>ZERO</td>
</tr>
<tr>
<td>Only when ACKed</td>
<td>DUP</td>
<td>OK</td>
</tr>
<tr>
<td>Only when not ACKed</td>
<td>OK</td>
<td>ZERO</td>
</tr>
</tbody>
</table>
Lost reply problem

- Problem: lost request, server crash or slow, lost reply can not be distinguished
- **Idempotent** messages can safely be repeated, but it is too restrictive in practice, to structure all requests as idempotent messages
- **Sequence number**, mark initial request separately, do not carry out retransmission, but answer it (i.e. send a response to the client) → stateful server
5. Client crash

- Unwanted active computation: orphan
- waste resources, stale locks, confusing replies
- Solutions:
  1. **Extermination**: client stub log, orphan killed after reboot (expensive, grand-orphans, partitions)
  2. **Reincarnation**: Reboot starts new epoch, all computations from earlier epochs killed (some may survive, but can be detected later due to old epoch no.)
  3. **Gentle reincarnation**: computation only killed, if owner cannot be found
  4. **Expiration**: Each RMI/RPC has expiry T. After reboot, clients waits T. Problem: Reasonable T.
Dependability and fault tolerance

- Taxonomy
- Techniques and challenges
- Classification
- Fault tolerance and redundancy
- Agreement (consensus)
- Reliable client server
- Group communication and membership
Reliable multicast

- Multicast is an essential element of many distributed algorithms
- Example: process groups, active replication
- A reliable multicast (group communication) is necessary for providing fault-tolerant distributed algorithms
- Group membership:
  - static: processes do not fail, join, leave
  - dynamic: reliable = delivery to all non-faulty group members, but agreement is needed, what the group currently looks like when a message is to be delivered
Nonhierarchical Feedback Control

- Feedback suppression (avoids feedback implosion):
  - NACK only
  - first (multicast) retransmission request (after random delay) leads to the suppression of others
  - retransmission (not necessarily original sender) is also multicast

- Scales well, but processes retain copies of delivered messages indefinitely
Hierarchical Feedback Control

The essence of hierarchical reliable multicasting.

a) Each local coordinator forwards the message to its children.

b) A local coordinator handles retransmission requests.

c) Scales well, but dynamic tree construction is a remaining problem.
Total, FIFO and causal ordering (1)

- **FIFO** ordering: If a process issues F₁ and then F₂, then every process will deliver F₁ before F₂ (partial ordering)
- **Causal** ordering: If C₁ happened-before C₂, then every process will deliver C₁ before C₂ (partial ordering)
- **Total** ordering: If a process delivers T₁ before T₂, then all processes deliver T₁ before T₂
- Causal ordering implies FIFO ordering
- We do not assume or imply reliability (can be combined)
Notice the consistent ordering of totally ordered messages $T_1$ and $T_2$, the FIFO-related messages $F_1$ and $F_2$ and the causally related messages $C_1$ and $C_3$ – and the otherwise arbitrary delivery ordering of messages.

**Hybrids:**
- FIFO-total
- Causal-total
Message ordering

Epochs are separated by group membership changes
Six different versions of virtually synchronous reliable multicasting regarding ordering within epochs

<table>
<thead>
<tr>
<th>Multicast</th>
<th>Basic Message Ordering</th>
<th>Total-ordered Delivery?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable multicast</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>FIFO multicast</td>
<td>FIFO-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Causal multicast</td>
<td>Causal-ordered delivery</td>
<td>No</td>
</tr>
<tr>
<td>Atomic multicast</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>FIFO atomic multicast</td>
<td>FIFO-ordered delivery</td>
<td>Yes</td>
</tr>
<tr>
<td>Causal atomic multicast</td>
<td>Causal-ordered delivery</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The hold-back queue

When delivery guarantees are met, messages are delivered from the delivery queue to the hold-back queue. After processing, they are delivered to their final destination.
Implementation model

- Multicast queue at each server node
- Multicast messages are stored in queue on arrival
- Messages are numbered (or timestamped) in some way
- Depending on desired order of delivery, messages are delivered from queue to the process after some coordination with queues of other servers
- Ordering can be expensive, application-specific message semantics can be more efficient ("end-to-end"-argument)
Totally-Ordered multicasting

- Clients multicast their updates with (Lamport) timestamp (FIFO, reliable)
- Upon receipt, the message is put into local queue ordered by timestamp
- Server acknowledges receipt of requests by multicast (for total ordering);
- Eventually all processes will have the same copy of the local queue
- A message that is at the head of the queue and has been acknowledged by all processes is delivered to server process (respective ACKs are deleted)
- Updates may not be done in “correct order” but they are done in the same order at all nodes
The ISIS algorithm for total ordering

P1, P2, P3, P4

1 Message

1 Proposed Seq

3 Agreed Seq
Causal ordering using vector timestamps
Design Issues for Process Groups

- Organize identical processes in a group
- Purpose: Collection of processes as a single abstraction
- Multicast is a key issue: all requests arrive at all servers in the same order (atomic multicast)
- Groups may be dynamic: mechanisms are needed to manage groups and group memberships
- Open groups vs. closed groups
- Flat groups vs. hierarchical groups
- A process can be member in several groups
Open and closed groups

Closed group

Open group
**Flat and hierarchical groups**

- **How can a message be delivered to all members of a group?**
- **Flat group**: no single point of failure
- **(Simple) hierarchical group**: Co-ordinator; decision making is easier
Group membership

- Creating and deleting groups; processes joining and leaving (or crashing)
  - group server: easy, efficient, but single point of failure
  - distributed group membership service (e.g. by reliable multicasting)
- Joining and leaving operations must be synchronous with data messages (e.g. by converting this operation into a sequence of messages sent to the whole group)
- Crashing may be more difficult to detect (fail-stop is too strong, usually fail-silent assumed)
- How to rebuild a group consistently.
Virtual Synchrony

Concepts: Group view and view delivery

- Either all (non-faulty) processes in the group receive the multicast in the same view, or none receives it (agreement, atomicity)
- The view delivery itself is totally ordered
View-synchronous GC

a (allowed).

b (allowed).

c (disallowed).

d (disallowed).
Summary

- Dependability is a holistic concept
- Distributed systems can suffer partial failures
- Distributed systems can provide fault-tolerance
- Faults can be due to process failures or communication failures
- Process replication (process groups) can help deal with process failures
- Reliable communication can be built on top of unreliable communication mechanisms
- Lost-reply problem has to be dealt with in client/server architectures
- A reliable multicast (group communication) is in many cases necessary for providing fault-tolerant distributed algorithms