Advanced Distributed Systems

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

http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/

Dependability

What it should have been like

What actually happened

Dependability

- Motivation and lecture overview
- Fault tolerance
- Reliable client server
- Agreement (consensus)

Holistic dependability

Contributing factors are ...

- technical
- social and cultural (corporate culture)
- psychological (*perceived* dependability)
- managerial (processes) and economical
- fostering learning is a key
- simplicity is generally an enabler for dependability

Perceived Dependability

- Suicide vs. homicide?
- Traffic casualties: Car vs. Plane?
- 2006, Germany:
  - 983 homicides
  - 9.765 suicides
- Traffic casualties:
  - 5.091 (Germany, 2006)
  - 730 (Austria, 2006)
  - 500 (Plane crashes, 2006, worldwide)
- de.statista.com, BKA, DLR

Techniques

- Fault tolerance techniques
- Security techniques
- System architecture and distribution
  → The 8 fallacies of distributed systems
- Hardware and IT Infrastructure
  Virtualization (VM, GRID, Cloud, and SOA)
- Software development methods, tools, and techniques
- Software maintenance
- 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

The 8 Fallacies of Distributed Computing

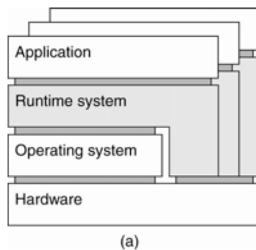
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

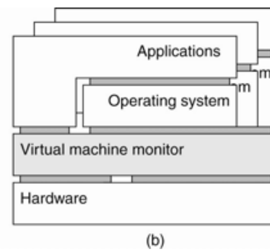
Essentially everyone, when they first build a distributed application, makes the above eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences. (*Peter Deutsch*)

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

Architectures of Virtual Machines

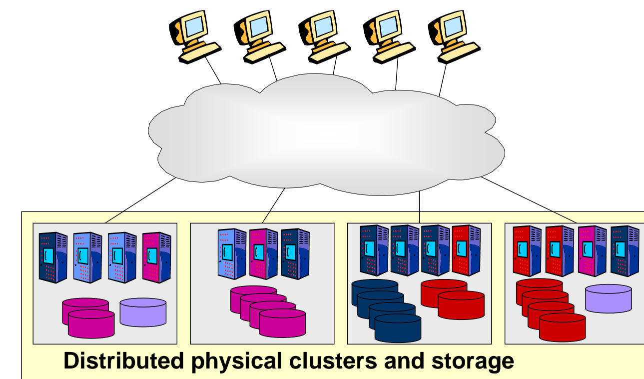




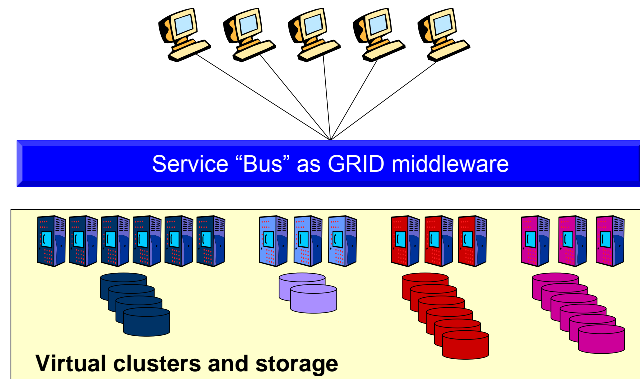
(a) A process virtual machine, with multiple instances of (application, runtime) combinations. E.g., JVM.

(b) A virtual machine monitor, with multiple instances of (applications, OS) combinations. E.g., VMware, Xen

Heterogeneous Resources



The Grid: Virtualizing Resources



Cloud Computing

amazon web services™ SIMPLE MONTHLY CALCULATOR

Computing Power as a configurable, payable Service

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

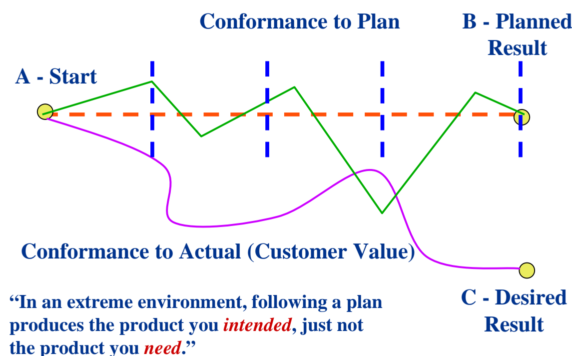
Requirements...

- ... do change – continously!
- ... 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

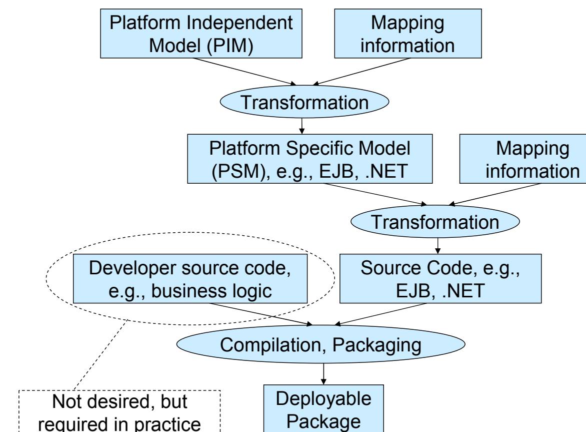
Answer on the process level

- Design for change in highly volatile areas!
- Heavy weight (CMM) → light weight (ASD) processes
- Development in-the-small: Component, service,...
  → agile development (ASD, XP), MDA, AOP, ...
- Development in-the-large: Procurement/discovery, re-use, composition, generation, deployment, ...
  → Product line, EAI, CBSE, (MDA), SOA, ...

Agile Development (ASD)



Model-Driven Architecture (MDA)



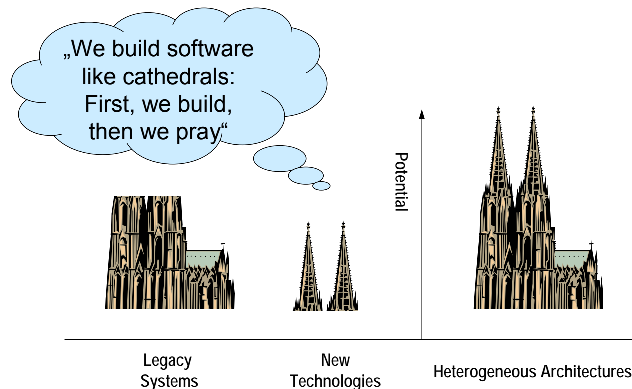
Dependability arguments for MDA

- Verification of system properties at PIM level
  - Formal verification
  - Testing (?)
- Verification of system properties at PSM level
  - Formal verification
  - Testing
  - Required platform specific properties
- In theory no component testing at code level necessary
  - Only System Test
- Documentation always up-to-date

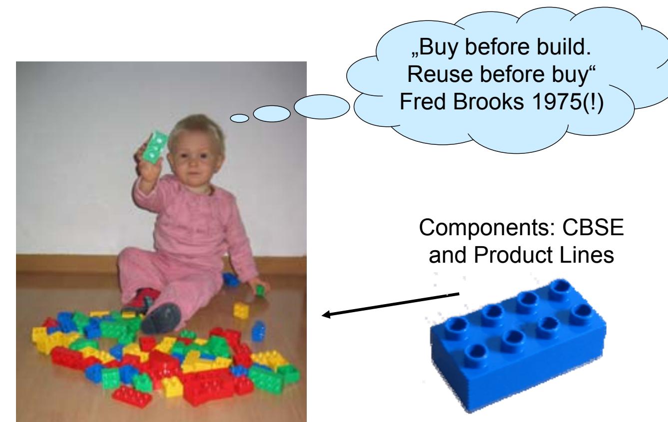
EAI: Software Cathedral

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

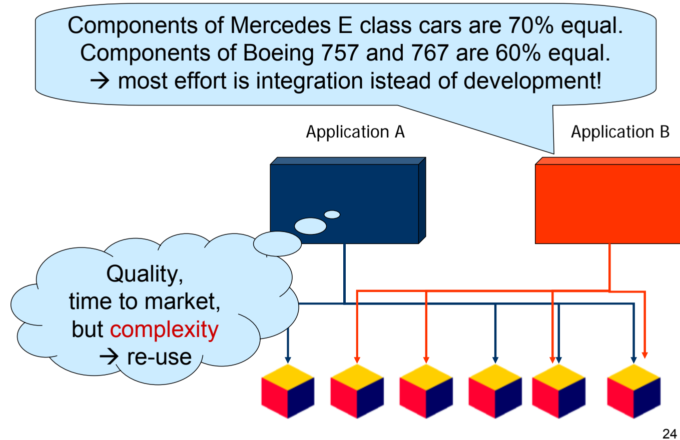
Heterogeneous Architectures



Component-based Software Engineering



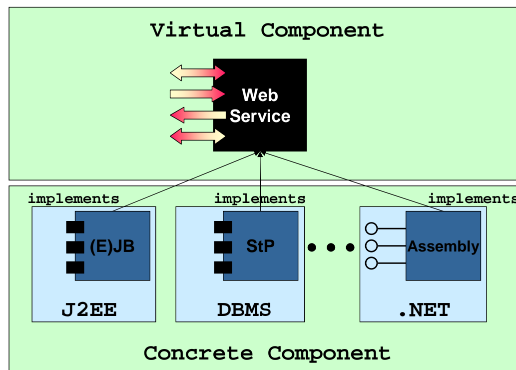
Product Line



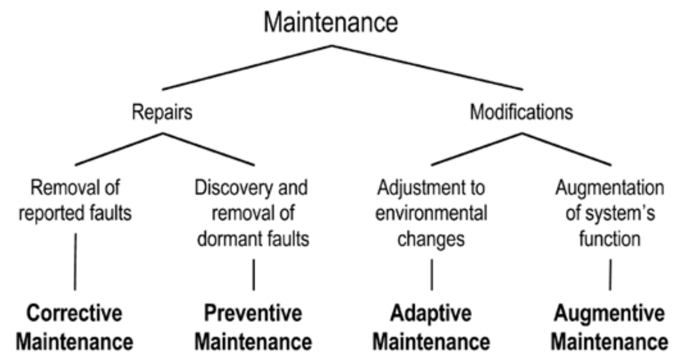
SOA is an evolution, not a revolution

- 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")

Virtualizing Components



Maintenance



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

Challenges of today's applications

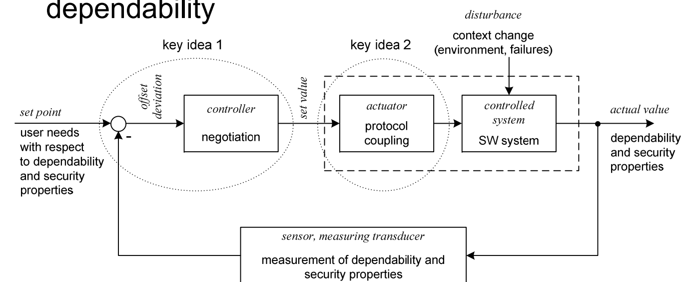
- heterogeneity (SOA, GRID)
- large-scale (pervasive, GRID, P2P, ULS)
- dynamic (MANET, SOA)
- run continously (24*7)
- time to market
- cost pressure
- → Human maintenance and repetitive software development processes
  - error-prone and costly
  - slow, sometimes prohibitively
  - BUT: self-learning and highly adaptive ;-)
- → dependability degradation / dependability gap

System Life Cycle – revisited

- Development Environment: physical world, human developers, development tools, production and test facilities → development faults.
- Use Environment: physical world, administrators, users, providers, infrastructure, intruders.
- Use phase: service delivery, service outage, service shutdown.
- Maintenance: repairs and modifications (iterative development process).
- Design-time/run-time convergence

Adaptivity

- Answer on the process level does not address run-time needs sufficiently
  → run-time adaptivity is needed for dependability



Adaptive Coupling

- Complexity theory demands focus on structure and interaction rather than properties of the individual constituents
- Relationships of differing strengths → mixture of tightly and loosely coupled parts
- → overall system properties are also determined by the strength of coupling
- → inner loop provides adaptivity by controlling the strength of coupling

Long-term evolution

- regulate emerging behaviour (policies)
- evolvement of user needs and context

- → change the system's design while running!
- requires run-time accessible and processable requirements and design-views, e.g.
  - constraints
  - models („UML virtual machine")
  - (partial) architectural confgurations
- requires methods to balance and change design and implementation during run-time

Run-time software development

- requires middleware support
  - stored in repositories
  - accessed via reflection
  - aspect-oriented programming (dynamic aspects)
  - protocols for meta-data exchange
  - „APIs" to change software artifacts
- → convergence of software development tools with middleware services („re-engineering running software")
- → new challenges: e.g., run-time testing and verification

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

Lecture overview

- Basic concepts
- Dependability and fault tolerance
- Group membership and atomic multicast
- Replication
- Peer to peer systems
- Multiplayer Online Games
- Case study: DeDiSys project
- Emerging techniques (→ seminar)

36

30

Dependability

- Motivation and lecture overview
- Fault tolerance
- Reliable client server
- Agreement (consensus)

„Classical“ fault tolerance

- 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

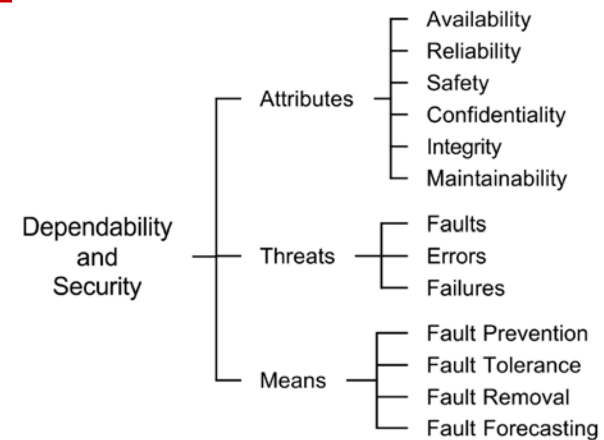
39

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.

Dependablity and security tree



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 continously 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.

Dependability and Security

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.

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, ...

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!

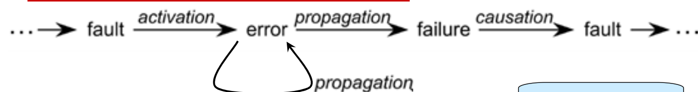
48

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)

49

Chain of dependability threats



or the environment

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

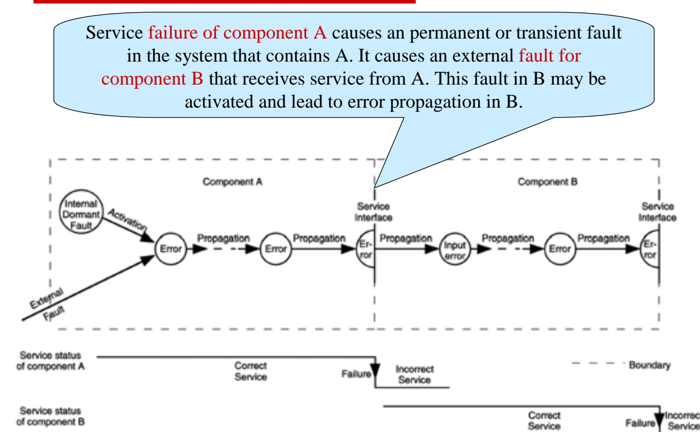
failure → fault

- 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

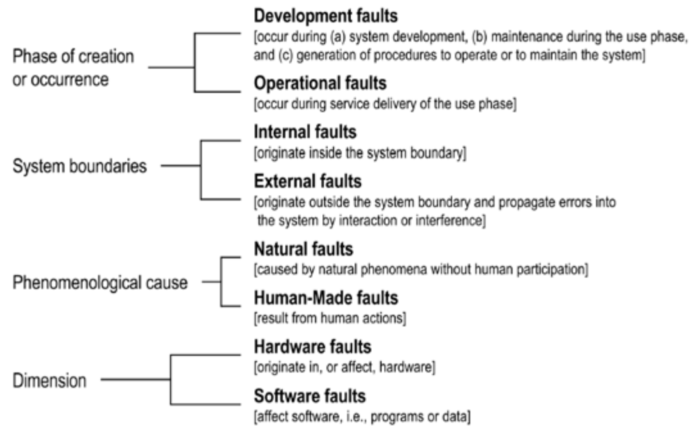
Propagation can occur via interaction, composition, creation, and modification

50

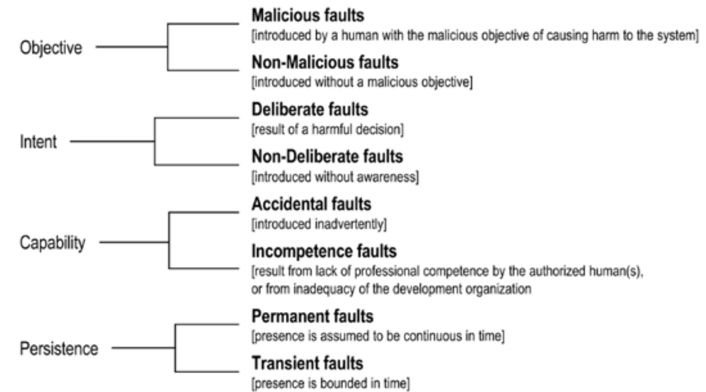
Error propagation



Fault classes (1)



Fault classes (2)



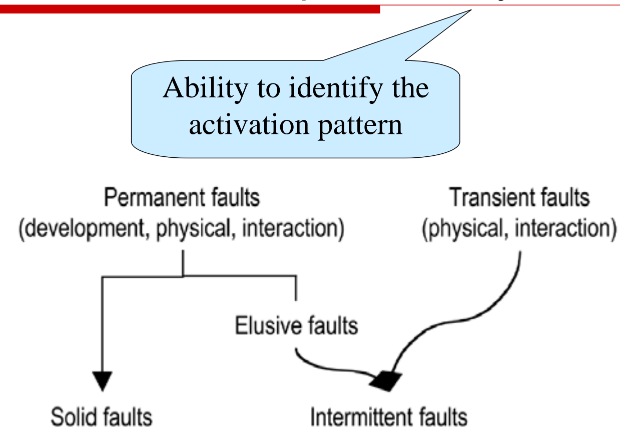
51-3

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

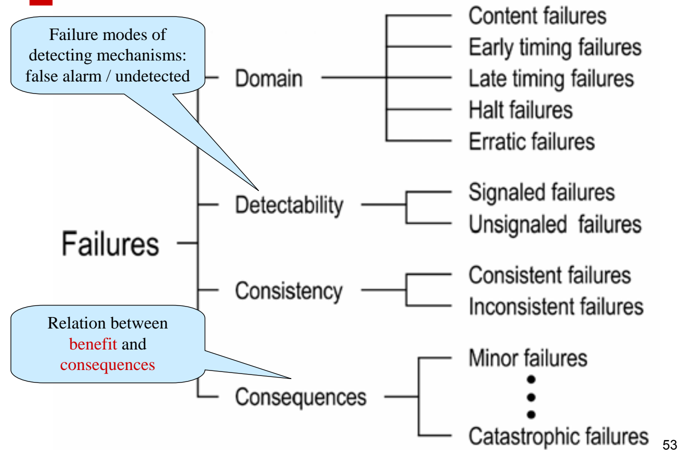
51-4

Fault activation reproducability



52

Service failure modes



53

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.

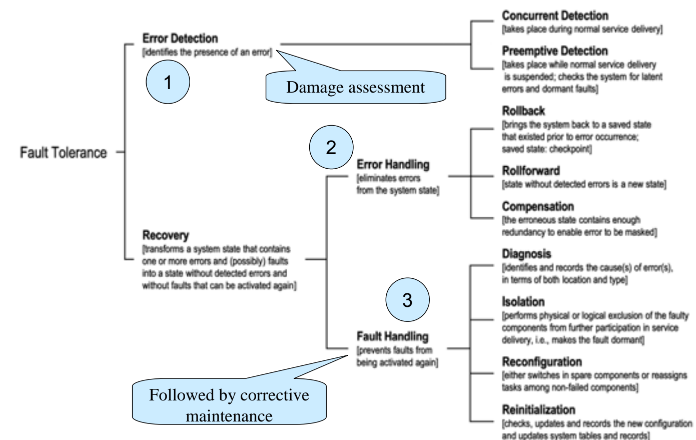
54

Fault Tolerance

- 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)

55

Fault tolerance techniques

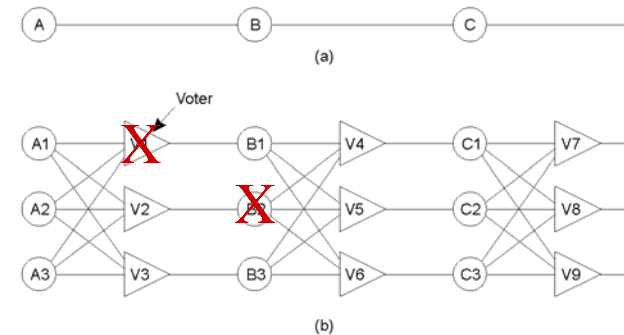


Failure masking by redundancy (1)

- 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?)

Failure masking by redundancy (2)

Triple modular redundancy (TMR)

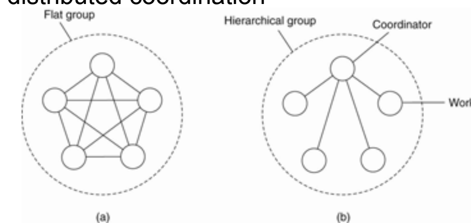


Process Resilience

- 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* replication is needed (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

- Motivation and lecture overview
- Fault tolerance
- Reliable client server
- Agreement (consensus)

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
   a. Before execution
   b. After execution
4. Reply message is lost
5. Client crashes after sending request

3. Server crash (1a)

- Primitive print service
- Three events that can happen at the server:
- Send the completion message (M),
- Print the text (P),
- Crash (C).

74-3

3. Server crash (1b)

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

74-4

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 failrure
- 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, alsways re-send, re-send only if ACK‘d, re-send only if *not* ACK‘d
- → 3 **crash orderings** per server strategy
- → 3*2*4 = 24 combinations to consider

74-5

3. Server crash (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

- 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.

74-6

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

Dependability

- Motivation and lecture overview
- Fault tolerance
- Reliable client server
- Agreement (consensus)

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

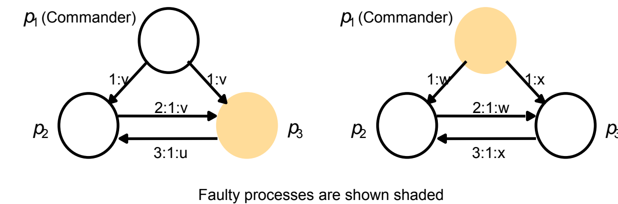
Synchronous vs. asynchronous

- 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 process crash detection while asynchronous model does not!

Agreement (consensus) problems

1. Synchronous, reliable communication, but processes exhibit arbitrary failure (including omission) → Byzantine generals problem.
2. Synchronous system, perfect processes, but communication unreliable → Two army (coordinated attack) problem.
3. Asynchronous communication, communication reliable, but arbitrarily slow (individual messages can be delayed indefinitely). At least one process *may* fail (silently). → FLP

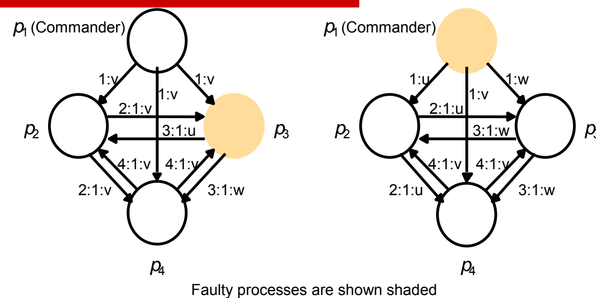
Three byzantine generals



Faulty processes are shown shaded

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

Four byzantine generals



Faulty processes are shown shaded

- 3m+1 processes are needed for agreement with m faulty processes (using unsigned messages)
- Recursive algorithm is quite expensive (m+1)

Two army problem

- Also known as „coordinated attack“
- Two armies have to coordinate their attack to succeed
- The processes (generals) are perfect
- However, the messanger can fail (the channel is unreliable)

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 (wrt time)
  - a degree of clock synchronization (long timeout helps with high probability, but slows down the system)
  - accurate enough failure detection

Jealous amazons problem

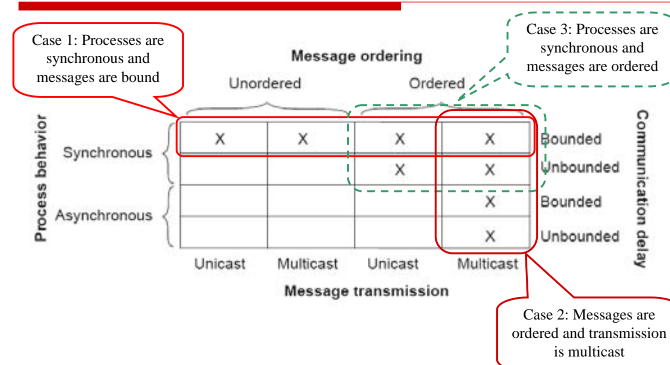
- Also known as „muddy children"
- State of being a cuckold lacks reflexivity
- Queen: „There are cuckolds. You are not allowed to cummunicate. When you are certain to be a cuckold, you shall shoot your partner at midnight that day"
- How does it work?

Bildzitat: Die junge Burra als Archetyp der Amazonen der Ophiswelt. Ausschnitt aus dem Titelbild von My56, von Nikolai Lutohin

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. →

Agreement in Faulty Systems (2)



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 practical cases!

71

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
- Lost-reply problem has to be dealt with in client/server architectures
- The many faces of consensus…

76