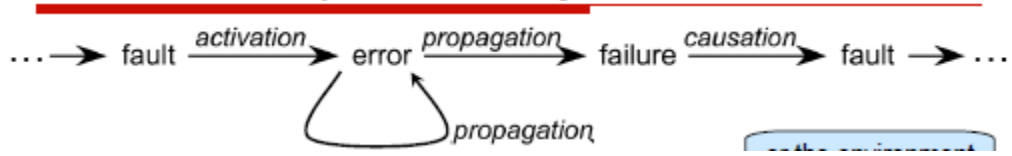
Question1: Explain the basic terms of Dependability: Name the five attributes (requirements) of a "dependable system". What is the difference between availability und reliability? Explain the "dependability threats" Failure, Error and Fault and the relation between them. Explain "permanent", "transient" and "intermittent" faults and provide examples.

Definition: (Slide 6)
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.

Attributes of a dependable system: (Slide 7)
- 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. **Difference to Availability**: A system which is maintained for 2 weeks a year but never fails has a high reliability but only an availability of 96%
- 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.

Dependability Threats: (11-13)
- Failure: Event that occurs, when the delivered service deviates from correct (expected/useful) service.
- Error: 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.
- Fault: A (design, programming, manufacturing) defect, that has the potential to generate errors

Relation between the above threats: (Slide 14)



Types of faults + Examples (Slide 46)
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

Why do we need a failure model? Provide different failure models for a "fail-controlled systems" and discuss them with respect to the required effort for masking faults. Why is it awkward to specify a system as "k-fault-tolerant"?

Book p. 324-326; Slides p. 51, 61

Why

Failures are divided into classes (failure models) for a better estimation of their consequences.

Failure models (by Tanenbaum) (slides p. 51)

- **Crash failure:** A server crashes and doesn't give any answers until it is rebooted but the server worked fine before the crash. The failure can be masked with the help of multiple redundant servers, because it is easy to detect, but as a result the traffic goes up and you have to face new problems like consistency.
- **Omission failure:** A server doesn't reply to requests. This is easy to mask.
  The failure can have following reasons:
  - **receive omission:** The server never received a request.
  - **send omission:** The server received the request, did the processing but wasn't able to send the reply.
- **Timing failure:** The response takes longer then a defined maximum response time. The masking effort depends on the defined response time. Too short: Too many failures occur. Too long: Failures are detected too late.
- **Response failure:** The reply from the server is incorrect. There are two kinds of response failures:
  - **Value failure:** The server replies a wrong answer. E.g. a search engine provides websites that were never searched for.
  - **State transition failure:** The server acts in an unexpected way to a request, i.e. he departs from the program flow. E.g. A server receives a request which he can't understand and starts an action that should never happen. This failure can't be masked most of the time because it is very hard to detect (mostly only by the user).
- **Inconsistent/arbitrary/byzantine failure:** A server may answer in a different way to the same request. If the failure cannot be detected internally by the system, it is called a byzantine failure. This failures cannot be masked at all, because they can't be detected.

K-fault-tolerance (slides p. 61)

A k-fault-tolerant system is a system, where k components can fail without any interference to the system and the system has to be possible to respond correctly.
Problem:

- The faulty processes can possibly run on and produce wrong data (byzantine failure).
- 3k+1 processes are necessary for a k-fault-tolerant system.

Summary

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Q 3:

Slide: 8 p. 56 Book: 326f

Why do we need redundancy for masking faults? What kinds of redundancy do you know?

Redundancy is the key for fault-tolerance. There can be no FT without redundancy!
If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy.

Three kinds are possible:

- information redundancy
- time redundancy
- physical redundancy

information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

time redundancy, an action is performed, and then if need be, it is performed again. Transactions use this approach. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

physical redundancy, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. Physical redundancy can thus be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of them crash, the system can still function correctly.

Exercise 04: Two army problem (slide 08, p. 68 – 69)

Question: Explain the proposition of the "two-army" problem.

Problem:
Two processes, using unreliable a communication, can't agree to a consensus.
(Example on slide p. 68)

Situation:

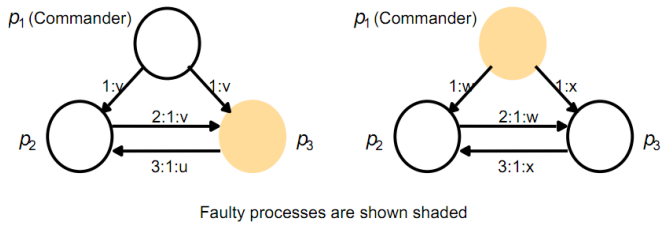
- Asynchronous messaging-system:
  Two nodes have to coordinate each other → agree to a consensus
- The processes are reliable
- Messages aren't tampered (gefälscht)
- Message loss is possible, because the used channel is unreliable
- It is not enough to just acknowledge messages
- Evidence of FLP: It is impossible to design a deterministic consensus algorithm in an synchronous distributed system subject to even a single process crash failure *(slide p. 69)*.

Example:
Meeting for dinner, sent per mail → It is impossible to know if the message has ever arrived yet!

Question5: Explain the proposition of the "byzantine generals".

Problem: (Slide 65, 66)

- There is a system with n processes (nodes)
- Communication between nodes is synchronous and reliable
- All nodes communicate with each other
- k processes are erroneous and provide faulty data



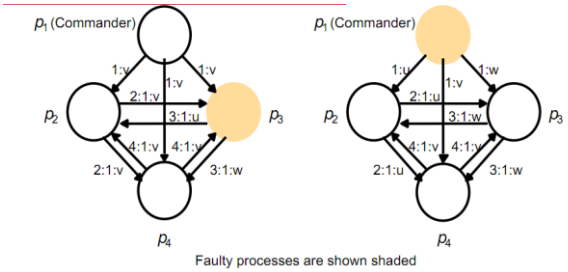
Faulty processes are shown shaded

(1:2:v means: process 1 says that process 2 says 'v')

Traitors may actively prevent loyal generals from reaching agreement by feeding incorrect and contradictory information

Solution: (Slide 67)

- n>=3k+1, i.e. a minimum of 3k+1 processes must exist in order to identify k wrong nodes
- This provides that the system runs correctly, i.e. k faulty nodes are compensated for. However, it is unknown which nodes are faulty



Faulty processes are shown shaded

3k+1 processes are needed for agreement with m faulty processes (using unsigned messages)

Explain the failure classes in client/server systems. What is the "lost reply" problem?

Book p. 336-342; Slides p. 73-80

Failure classes (slides p. 74)

1. **Client Cannot Locate the Server :** The client is unable to locate the server, e.g. the server is down, wrong client stub
2. **Lost request:** The request message from the client to the server is lost
3. **Server crashes:** The server crashes after receiving a request
   Problem: Client cannot detect if the server crashed before or after the processing of his request.

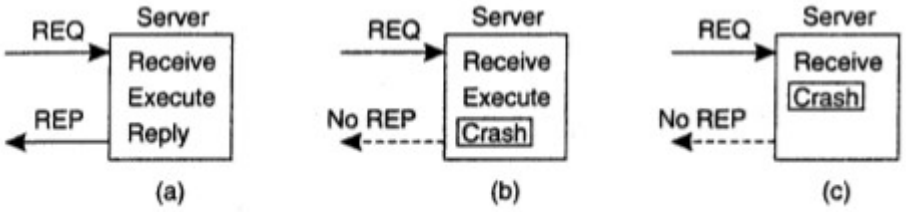


Figure 8-7. A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

4. **Lost reply:** The reply message from the server to the client is lost
5. **Client crashes:** The client crashes after sending a request

Lost reply (4th failure class) (slides p. 79)

The reply of the server gets lost. The client cannot detect whether the request got lost, the reply got lost or the server crashed in the meantime.

Solutions:

- **Repeat** the reply. Only useful with **idempotent** operations, i.e. operations which can be repeated (e.g. reading data; where a transaction from one bank account to another is not idempotent).
- Each request has a **sequence number**, so that the server can detect whether it is a new request or an already processed one. In this case there has to be a specified amount of time how long the server should remember the requests.
- The client could add a **flag** to the request, marking the request as new or repeated.

Q. 7

Slide: 8 p 82f book: 343f

What is reliable and ordered multicast (group communication) in static process groups? What has to be taken care of, when the groups change dynamically? Explain the concept of "atomic multicast" ("virtual synchrony").

Slide 8, p 82

Such services guarantee that messages are delivered to all members in a process group. Unfortunately, reliable multicasting turns out to be surprisingly tricky.

- reliable multicast guarantees that a message multicast to group view G is delivered to each non faulty process in G. If the sender of the message crashes during the multicast, the message may either be delivered to all remaining processes, or ignored by each of them.
- crashed replica, however, it may have missed several updates. At that point, it is essential that it is brought up to date with the other replicas. Bringing the replica into the same state as the others requires that we know exactly which
- transport layers offer reliable point-to-point channels, they rarely offer reliable communication to a collection of processes

slide p 93
Atomic multicast

In particular, what is often needed in a distributed system is the guarantee that a message is delivered to either all processes or to none at all. In addition, it is generally also required that all messages are delivered in the same order to all processes. This is also known as the atomic multicast problem.

slide p 97
virtual synchrony

Reliable multicast in the presence of process failures can be accurately defined in terms of process groups and changes to group membership. Virtual synchrony allows an application developer to think about multicasts as taking place in epochs that are separated by group membership changes. However, nothing has yet been said concerning the ordering of multicasts.

four different orderings:

- **Unordered multicasts:** nothing defined
- **FIFO-ordered multicasts:** the communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent.
- **Causally-ordered multicasts:** delivers messages so that potential causality between different messages is preserved. if a message m_1 causally precedes another message m_2, regardless of whether they were

multicast by the same sender, then the communication layer at each receiver will always deliver m_2 after it has received and delivered m_1

- **Totally-ordered multicasts:** means that regardless of whether message delivery is unordered, FIFO ordered, or causally ordered, it is required additionally that when messages are delivered, they are delivered in the same order to all group members