Dependability in Distributed Systems

Distributed Systems

Andrea Omicini
andrea.omicini@unibo.it

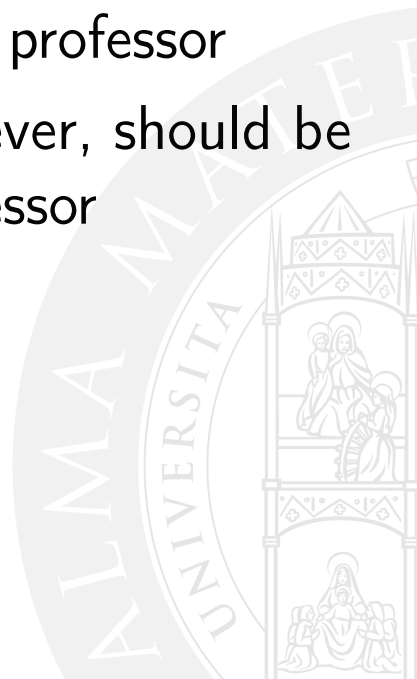
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2025/2026

Next in Line...

Disclaimer

- the following slides mostly borrow from a couple of books[Tanenbaum and van Steen, 2017, Zhao, 2014]
- material from there has been re-used in the following, and integrated with new material according to the personal view of the professor
- every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of this course's professor



Premise I

Dependability matters

- distributed systems are playing an ever-increasingly important role in all aspects of our society, governments, businesses, and individuals alike
  - they are behind many services on which we depend on a daily basis, such as
    - financial—e.g., online banking and stock trading
    - e-commerce—e.g., online shopping
    - civil infrastructure—e.g., electric power grid and traffic control
    - entertainment—e.g., online gaming and multimedia streaming)
    - personal data storage—e.g., cloud services such as Dropbox, Google Drive, and SkyDrive
  - their *dependability* matters to businesses as well as to every individual

Premise II

The cost of failures

- the cost of system *failures* is mostly and typically enormous
  - e.g., if a data centre is brought down by a system failure, the average cost for downtime may range from *tens* to *hundreds* of *thousands* of euros per hour, by summing up wasted expenses and loss of revenue

Premise III

The cost of dependability

- however, also dependability is *not cheap* at all
  - which is possibly the reason why nearly 60% of Fortune 500 companies suffer from more than one hour and a half of downtime per week[a]
- e.g., the cost of data centres is huge, and mostly depends on the level of availability required
  - from 450$ per square foot for 99.671% availability (i.e., 28.8 hours of downtime per year) to 1,100$ per square foot for 99.995% availability (i.e., 0.4 hours of downtime per year)[b]

? how can we reduce the cost of dependability?

---

[a] https://www.thefreelibrary.com/Assessing+the+financial+impact+of+downtime.-a0227813183

[b] https://www.datacenterjournal.com/the-price-of-data-center-availability/

Premise IV

"Engineers against downtime"

- in order to reduce the cost of building and maintaining highly-dependable systems, we could try and *train* more *experts* that know how to design, implement, and maintain dependable distributed systems
- let's try this here.

Next in Line...

Dependability in the Dictionary

Oxford Learner's Dictionaries

the quality of being able to be relied on to do what somebody wants or needs[a]

synonym reliability

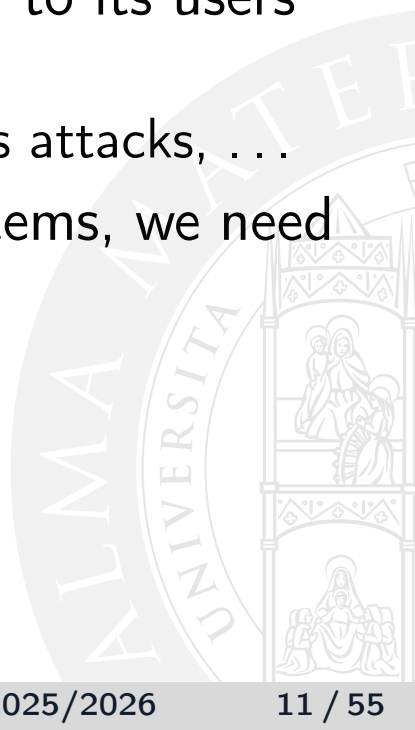
[a] https://www.oxfordlearnersdictionaries.com/definition/english/dependability

- in systems engineering, too, *dependability* is closely related to *reliability*

! not the same thing, however

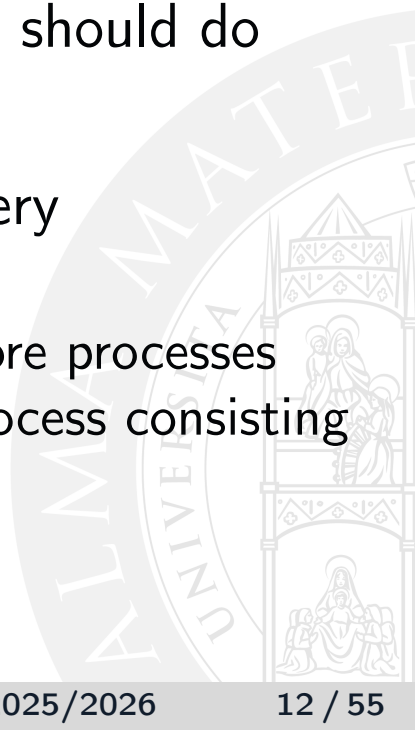
Basic Concepts and Terminologies

- the term "dependable systems", along with the related notions, belongs to the general scope of systems engineering, not just to computational systems—let alone distributed ones
- in the context of *distributed computing*, dependability refers to the ability of a distributed system to provide *correct services* to its users despite the many different threats
  - undetected software defects, hardware failures, malicious attacks, ...
- in order to reason about dependability of distributed systems, we need
  - a way to *model* a distributed system appropriately
  - a way to *model* the threats to the system



System Models I

- a (distributed) system is designed to provide a set of services to its users
  - often referred to as clients
- each service has an interface that a client could use to request the service
- a *functional specification* typically defines what a service should do
- at each moment in *time*, a system is in a given state
- what precisely is the *state* of a distributed system is a very complicated issue
  - e.g., a distributed system typically consists of one or more processes spanning over one or more network nodes, with each process consisting of one or more threads



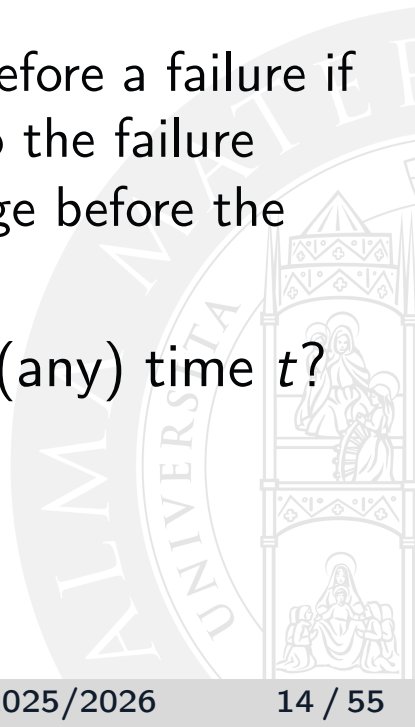
System Models II

- the state of a distributed system is determined *collectively* by the state of the processes and threads in the system
  - it typically consists of the values of its registers, stack, heap, file descriptors, the kernel state, ...
- part of the state might become either implicitly or explicitly visible through user interaction
  - which we call external state, or, *observable state*
  - basically, an abstract state as defined by the functional specification
- the remaining part of the state that is not visible to users
  - which we call internal state

System Models III

- generally speaking, the state of a system at time t is represented by the (minimum amount of) information that, along with the knowledge about the dynamics of a (deterministic) system, allows an observer to completely describe the future system behaviour—from time t on
- state can be used for recovery after a failure
  - a system can be recovered to the "place" where it was before a failure if its state was (consistently) captured and not lost due to the failure
  - e.g., if the state is serialised and written to stable storage before the failure, and accessible as intact after the failure

? how can we capture the state of a distributed system at (any) time t?



System Models IV

- first of all, we should define the boundary of a system
  - that is, the physical / logical line that separates the system from its surrounding environment
    - inside the boundary, the system's components
    - outside the boundary, the system's environment
- the term "environment" refers here to all other systems that the current system interact with—in the very general sense that they affect the system in any possible way

! every definition of component vs. system is determined by the level of abstraction chosen—e.g.,

- at one given level of abstraction, we could observe a system (with its components) interacting with other systems
- at another given level of abstraction, the same system could be instead described as a component of a larger system

System Models V

- however, this is not enough to assess that we know how to *capture the state* of a distributed system
  - mostly because we do not know (yet) what "at time t" would mean there
- ↑ the complex issue of time in distributed systems
  - *we will talk about that in the next weeks*

Threat Models I

- when a system is not compliant with its functional specification, we say that a (*service*) failure has occurred
- the failure of a system is caused by (part of) its state having wrong values—i.e., errors in its state
- the causal model assumes that errors are caused by some faults

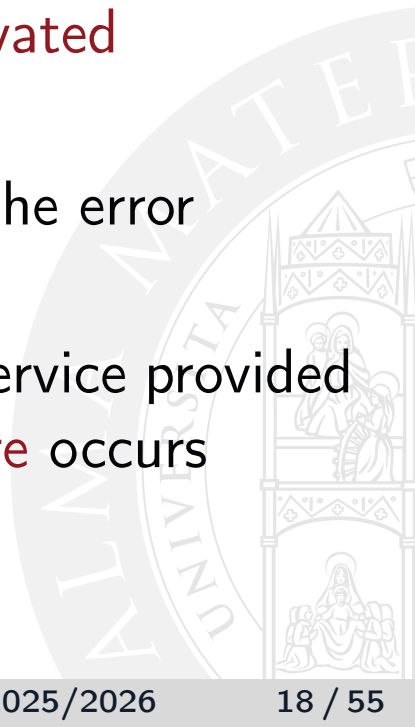
→ *threats* to dependability are modelled in terms of (different sorts of) faults

In other terms... [Tanenbaum and van Steen, 2017]

- a system is said to *fail* (failure) when it does not behave as promised
- an *error* is a part of a system state that might have caused a failure
- the cause of an error is a *fault*

Threat Models II

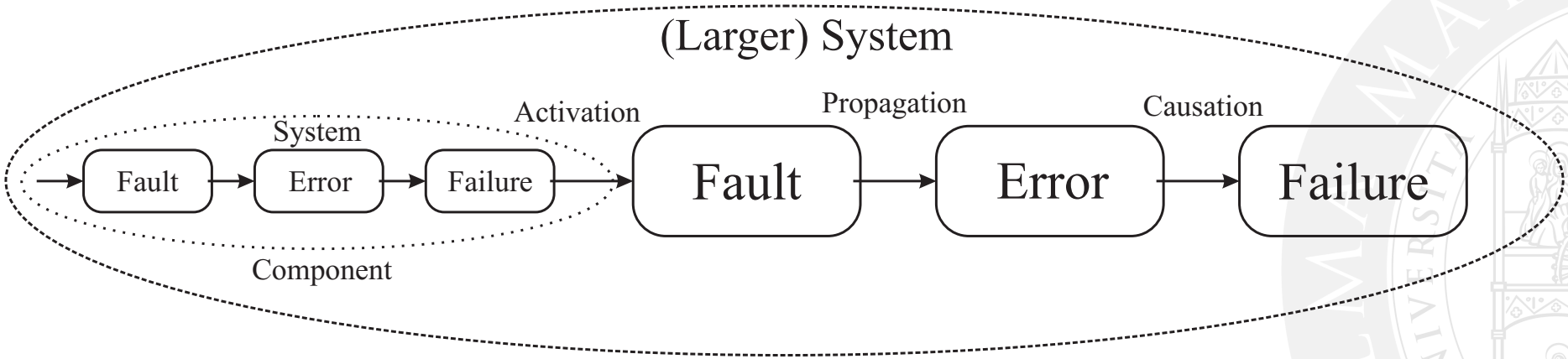
- a fault might be dormant
  - that is, it could not immediately show itself and cause error—e.g.,
    - a bug in the software not causing problems until the corresponding code is executed
    - a shared variable not lock protected in a multithreaded application until two or more threads try to update it concurrently
- when the specific condition is met, the fault will be activated
  - causing an error in the component
- when the component interacts with other components, the error propagates through the system
- when the error propagates to the interface, making the service provided to a client deviate from the specification, a *service* failure occurs



Threat Models III

- given the typical recursive nature of system composition, the failure of one system may cause a fault in a larger system when the former constitutes a component of the latter
- such a relationship between fault, error, and failure is referred to as *chain of threats*

! which makes the terms "fault" and "failure" often used interchangeably in the literature—somehow improperly, anyway



Sorts of Faults

- different sorts of faults require different treatment
- faults can classified according to different criteria
  - the *source* of the fault
  - the *intent* of the fault
  - the *duration* of the fault
  - the *manifestation* of the fault
  - the *reproducibility* of the fault
  - the *relationship* of the fault with other faults

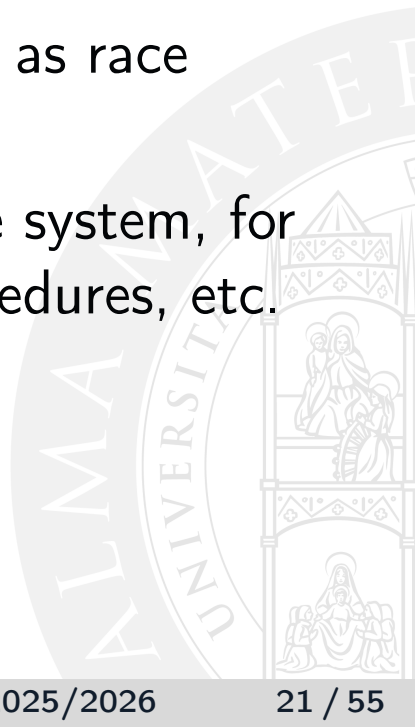
Source of Faults

Based on their *source*, faults can be classified as

hardware faults if the faults are caused by the failure of hardware components such as power outages, hard drive failures, bad memory chips, etc.

software faults if the faults are caused by software bugs such as race conditions and no-boundary-checks for arrays

operator faults if the faults are caused by the operator of the system, for example, misconfiguration, wrong upgrade procedures, etc.



Intent of Faults

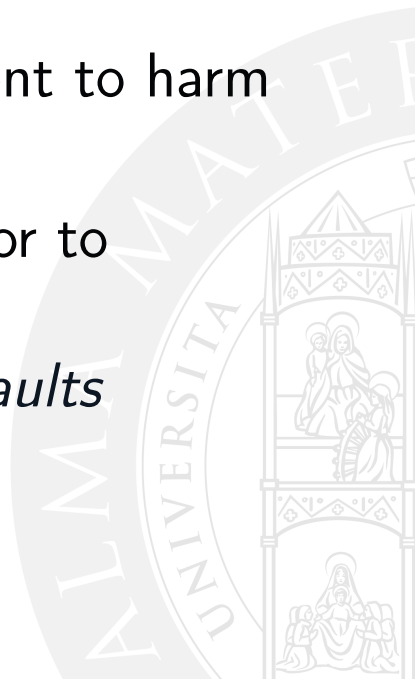
Based on their *intent*, faults can be classified as

non-malicious faults if the faults are not caused by a person with malicious intent

- e.g., a naturally-occurred hardware fault, or, some unintended software bugs

malicious faults if the faults are caused by a person with intent to harm the system

- e.g., to deny services to legitimate clients or to compromise the integrity of the service
- see also *commission faults*, or *Byzantine faults*



Duration of Faults

Based on their *duration*, faults can be classified as

transient faults if the fault is activated momentarily then goes dormant again

- e.g., when a power spike affects a hardware component then disappears

intermittent faults if the fault occurs, vanishes of its own accord, then reappears, and so on

- e.g., race condition, occurring when and only when two threads access the same shared variable at the same time

permanent faults if, once activated, the fault stays such unless the faulty component is repaired or the source of the fault is addressed

- e.g., a power outage is considered a permanent fault because a computer system will remain powered off unless / until power is restored
- or, a (process) crash fault

Manifestation of Faults

Based on their *manifestation*, faults can be classified as

content faults if the faults cause the values passed on to other components to be wrong

- a faulty component may always pass on the same wrong values to other components, or, it may return different values to different components that it interacts with
- where the latter case is specifically modelled as *Byzantine faults*

timing faults if the faulty component either returns a reply too early, or too late after receiving a request from another component

- an extreme case is when the faulty component stops responding at all
  - i.e., it takes infinite amount of time to return a reply
- e.g., when the component crashes, or hangs due to an infinite loop or a deadlock

Reproducibility of Faults

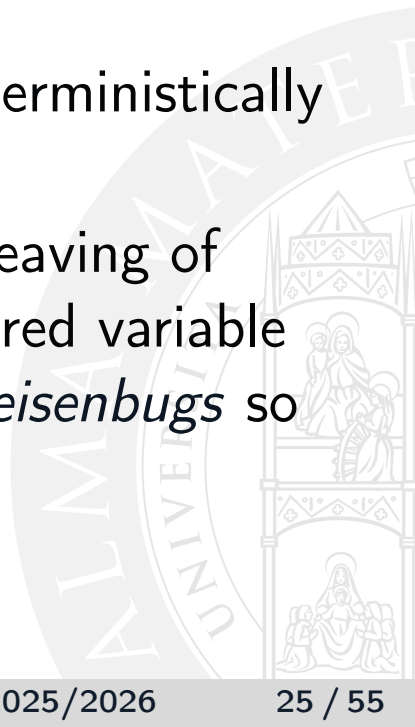
Based on their *reproducibility*, faults can be classified as

reproducible faults (*deterministic* faults) if the fault happens deterministically and can be easily reproduced

- e.g., accessing a null pointer
- this sort of faults are typically easily identified and repaired

nondeterministic faults if the fault appears to happen nondeterministically and is hard to reproduce

- e.g., if a fault is caused by a specific interleaving of several threads when they access some shared variable
- this sort of faults are also referred to as *Heisenbugs* so as to highlight their uncertainty



Relationship of Faults with Other Faults

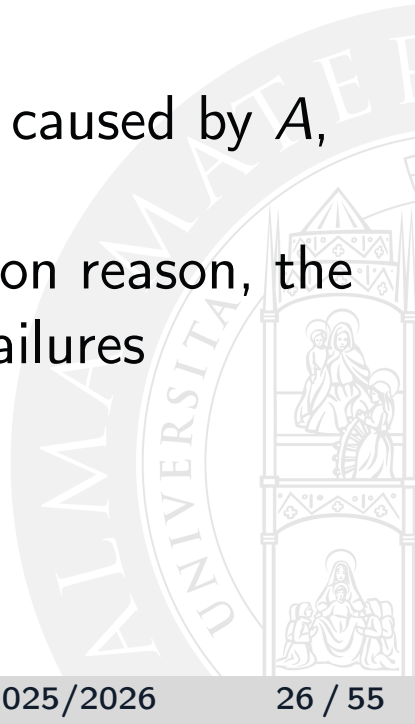
Based on their *relationship* with other faults, faults can be classified as

independent faults if there is no causal relationship between the faults,

- e.g., given fault A and fault B, B is not caused by A, and A is not caused by B

correlated faults if the faults are causally related

- e.g., given fault A and fault B, either B is caused by A, or A is caused by B
- if multiple components fail due to a common reason, the failures are referred to as *common mode* failures



Failure Models: Another Classification [Tanenbaum and van Steen, 2017]

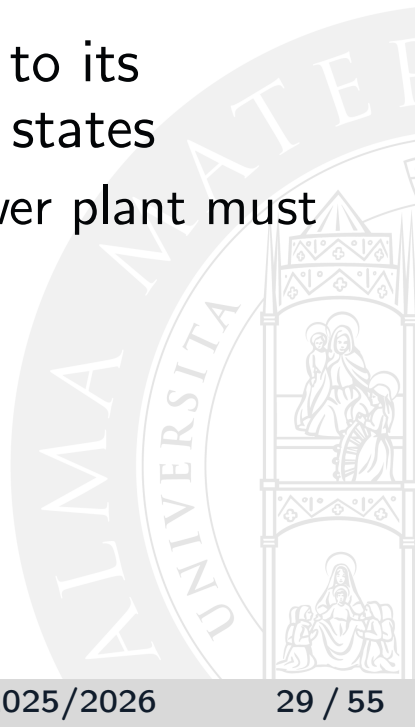
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response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

Fail-Stop Systems

- when the system fails, it is desirable to avoid *catastrophic consequences*
- the consequence of the failure of a system can be alleviated by enhancing the system with dependability mechanisms such that when it fails, it stops responding to requests
- such systems are referred to as fail-stop systems

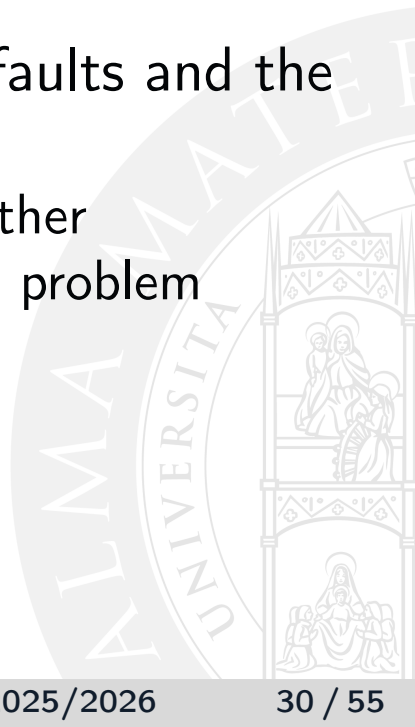
Fail-Safe Systems

- sometimes, a system is such that its failure does not cause great harm either to human life or to the environment
  - such systems are referred to as fail-safe systems
- usually, a fail-safe system defines a set of *safe states*
- when a fail-safe system can no longer operate according to its specification, it can transit to one of the predefined safe states
  - e.g., the computer system used to control a nuclear power plant must be a fail-safe system



Fail Fast

- it is often desirable for a system to halt its operation immediately when it is in an error state or encounters an unexpected condition
- the software engineering practice to ensure such a behaviour is called fail fast
- the fail-fast practice enables *early detection* of software faults and the diagnosis of faults
  - given that when a fault has been propagated to many other components, it way harder to pinpoint the source of the problem



Next in Line...

Features of Dependable Systems [Tanenbaum and van Steen, 2017]

Main features of dependable systems

- availability
- reliability
- safety
- maintainability

Availability

Definition

Availability refers to the property that a system is ready for immediate use

This means...

- ... that availability refers to the *probability* that a system is operating correctly at any given moment, ready to provide users with its functions
- so, a *highly-available system* is a system that is most likely to be ready and working at any given instant of time

Reliability

Definition

Reliability refers to the property that a system can run continuously without failure

This means...

- ... that reliability is defined in terms of a *time interval*, rather than of a instant (as instead in the case of availability)
- so, a *highly-reliable system* is a system that is most likely to keep on running for a long period of time

Safety

Definition

Safety refers to the situation that when a system temporarily fails to operate correctly, nothing *catastrophic* happens

This is...

- ... a very difficult property to be defined, and to be ensured as well
- where *catastrophic* basically refers to those states of affairs when there is no established path from fault back to normal operation

Maintainability

Definition

Maintainability refers to how easily a failed systems can be repaired

This means...

- ... that maintainability is closely related to availability
- so, a highly-maintainable system may also show a high degree of availability

Attributes of Dependable Systems[Zhao, 2014]

- a dependable system has a number of *desirable attributes*
  - availability
  - reliability
  - integrity
  - maintainability
  - safety
- some attributes can be used as *evaluation metrics* for the system
  - availability, reliability
- the others are difficult to quantify
  - integrity, maintainability, safety
- some attributes are *fundamental* to all distributed systems
  - availability, reliability, integrity
- the others are secondary or not applicable to all systems
  - maintainability, safety

Availability I

Informal definition

Availability is a measure of the readiness of a dependable system at a(ny) point in time, i.e., when a client needs to use a service provided by the system, the probability that the system is there to provide the service to the client

Availability factors

MTTF *Mean Time To Failure* the average time until a system fails

MTTR *Mean Time To Repair* the average time needed to repair a system

MTBF *Mean Time Between Failures* just $MTTF + MTTR$

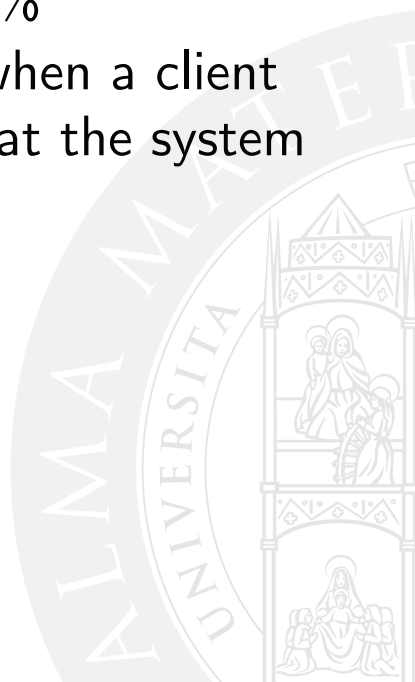
Availability II

Formal definition

$$Availability = MTTF/(MTTF + MTTR) = MTTF/MTBF$$

Availability III

- the availability of a system is typically represented in terms of *how many 9s*
  - e.g., if a system is claimed to offer *five 9s availability*, it means that the system will be available with a probability of 99.999%
  - i.e., the system has 10^{-5} probability to be unavailable when a client tries to access the service at any time—which means that the system may have *at most* 5.256 minutes of downtime a year



Reliability

Definition

Reliability is a measure of the system's capability of providing correct services *continuously* for a period of time

- it is often represented as the probability for the system to do so for a given period of time Δt
- reliability $= R(\Delta t) = e^{-\lambda \Delta t}$
  - where $\lambda \in [0, \infty)$ is the *failure rate*
- $\approx$ proportional to $MTTF$

! availability $\neq$ reliability

- e.g., a system failing frequently but recovering very quickly may have high availability, and at the same time very low reliability

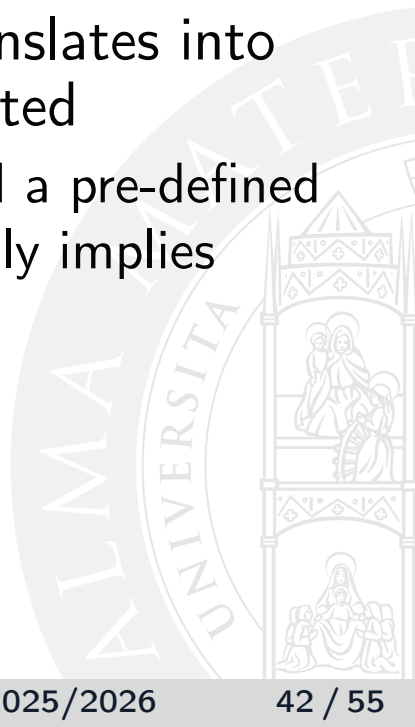
Integrity

Definition

Integrity refers to the capability of a system to protect its state from being compromised under various threats

! in dependable computing research, integrity typically translates into the consistency of server replicas, if redundancy is exploited

- as long as the number of faulty replicas does not exceed a pre-defined threshold, the consistency of the correct replicas naturally implies system integrity



Maintainability

Definition

Maintainability refers to the capability of a system to evolve after it is deployed

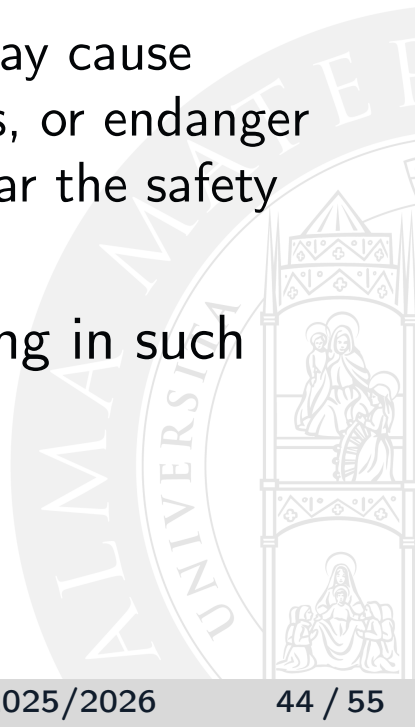
- once a software fault is detected, applying a repairing patch should not involve uninstalling the existing system and reinstalling an updated one
- the same patching/software update mechanism may be used to add new features or improve the performance of the existing system
- ideally, a highly-maintainable system is easy to repair / improve
  - *live upgrade*

Safety

Definition

Safety means that when a system fails, it does not cause *catastrophic consequences*

- i.e., the system must be fail-safe
  - e.g., systems that are used to control operations that may cause catastrophic consequences, such as nuclear power plants, or endanger human lives, such as hospital operation rooms, must bear the safety attribute
- safety attribute is not important for systems not operating in such environments, e.g., for e-commerce



Next in Line...

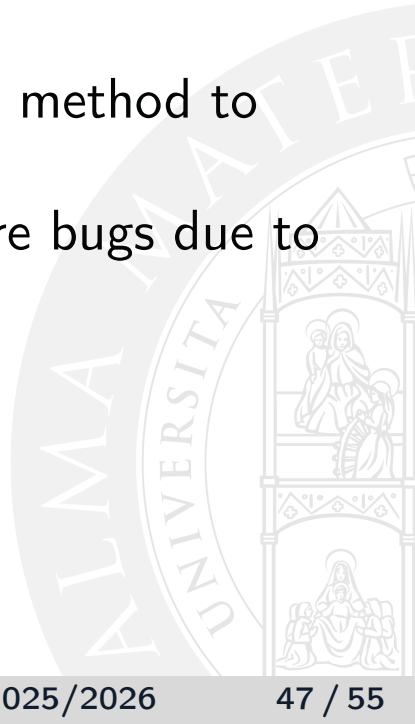
Improving Dependability[Zhao, 2014]

Approaches to improving the dependability of distributed systems

- fault avoidance
- fault detection & diagnosis
- fault removal
- fault tolerance

Fault Avoidance

- by building and using quality software components and hardware that are less prone to failures
- as far as software is concerned, fault avoidance aims at ensuring correct design specification and correct implementation before a distributed system is deployed
  - *rigorous software design*, possibly with the use of formal method to assess and verify system properties
  - *rigorous software testing*, to identify and remove software bugs due to design deficiency or introduced during implementation



Fault Detection & Diagnosis

- *detection* can be tricky
  - crash faults are trivial to detect: we can periodically probe each component to check on its health
  - however, components in a practical system might fail in various ways other than crash implementation, so probing does not always help
  - if a fault not detected, the integrity of the system cannot be guaranteed
- once the fault is detected, *diagnosis* is required to determine that a fault indeed has occurred, and to localise the source of the fault
  - i.e., pinpoint the faulty component
  - formal models and (statistical) tools are exploited to this end

Exception handling

- an example of fault detection and handling is represented by *exception handling* in modern programming languages

Fault Removal

- once a fault is detected and localised, it should be isolated and removed from the system
  - then the faulty component is either repaired or replaced, and reintroduced in the system
  - which typically requires reconfiguration
- in a distributed system, this typically requires a notion of *membership*
  - the faulty component is excluded from the system
  - the repaired component becomes part of the membership again

! special case: software update

Fault Tolerance I

- because of hardware failures, robust software itself is not enough to delivery high dependability
- unless a system is fully stateless, simply restarting after a failure would not automatically restore its state before the failure
- hence, fault tolerance techniques are essential to improve the dependability of distributed systems to the next level

Fault Tolerance II

Fault tolerance techniques vs. dependability requirements

- different fault tolerance techniques can be used to target different levels of dependability requirements
- e.g.,
  - for applications that need high availability, but not necessarily high reliability, logging and checkpointing, could be enough
  - more demanding applications could adopt recovery oriented computing techniques
  - both classes of techniques rely on rollback recovery—that is, back to the most recent recorded correct state

Masking Failure by Redundancy

Idea

- hiding failures from other processes
- the key technique for masking faults is redundancy

Three kinds of redundancy

- information redundancy
  - e.g., extra bits
- time redundancy
  - e.g., redos after transaction aborts
- physical redundancy
  - typical in biological systems

Next in Line...

Lessons Learnt

- dependability is an essential feature for distributed systems
- models for systems and threats are required to handle dependability
- techniques for handling failures determine the level of dependability of a distributed system

Dependability in Distributed Systems

Distributed Systems

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2025/2026

References

[Tanenbaum and van Steen, 2017] Tanenbaum, A. S. and van Steen, M. (2017).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, 3rd edition
https://www.distributed-systems.net/index.php/books/ds3/

[Zhao, 2014] Zhao, W. (2014).
Building Dependable Distributed Systems.
Wiley
DOI:10.1002/9781118912744