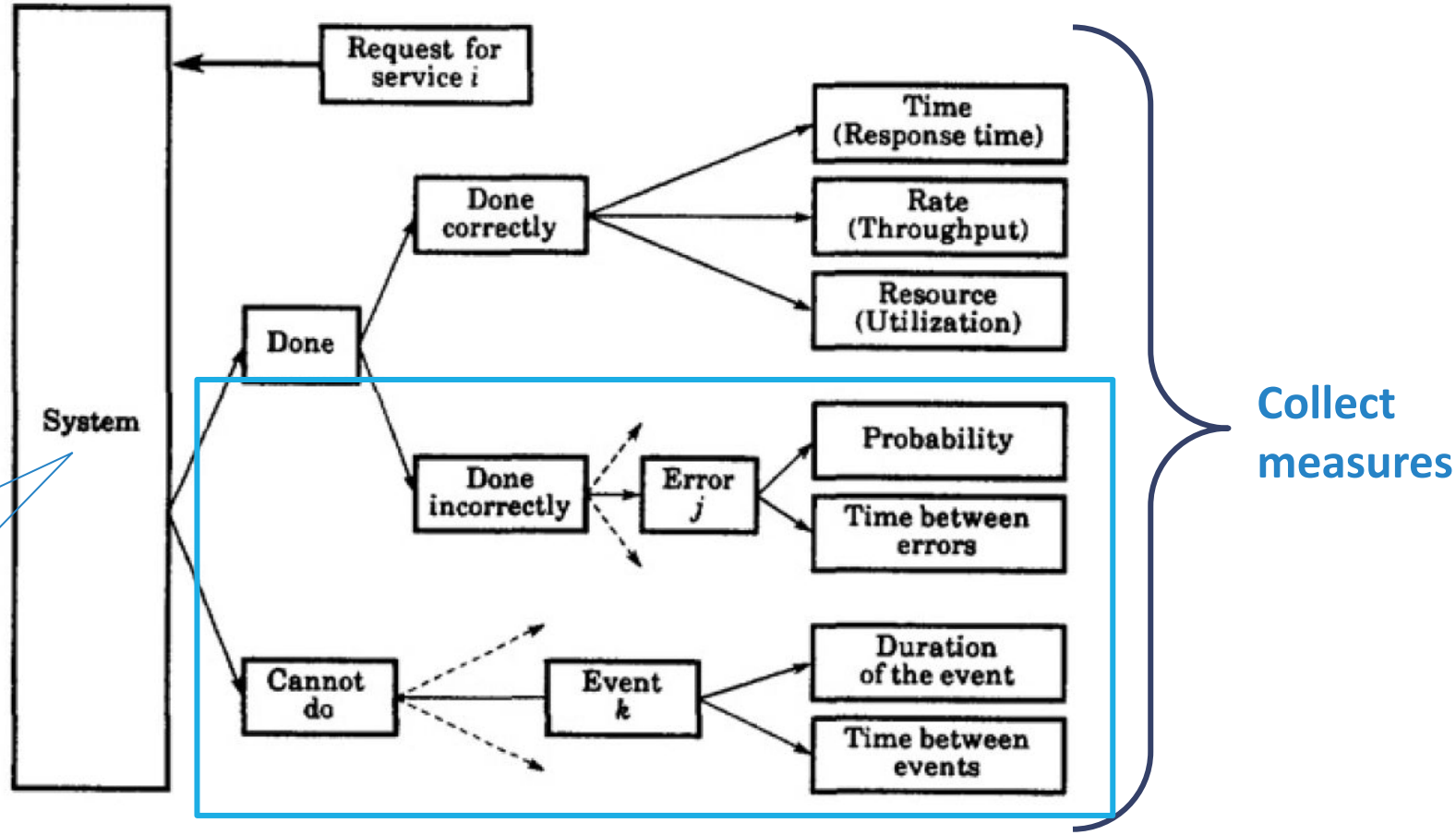
Dependable Distributed Systems
Master of Science in Engineering in Computer Science

AA 2022/2023

LECTURE 24 : DEPENDABILITY EVALUATION

Schena

Recall: Very Basics for Dependability Evaluation



Note that it is assumed that the system is available for measurements!

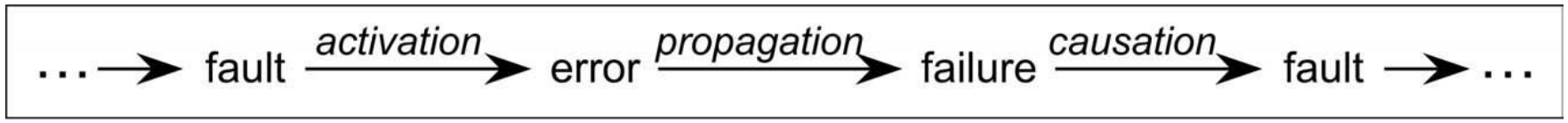
Recalls: Dependability

Dependability is the ability of a system to deliver a service that can justifiably be trusted, it is **the ability to avoid service failures** that are more frequent and more severe than is acceptable.

- **Availability**: readiness for correct service
- **Reliability**: continuity of correct service
- **Integrity**: absence of improper service
- …

Robustness: Dependability with respect to external faults which characterizes a system reaction to a specific class of faults

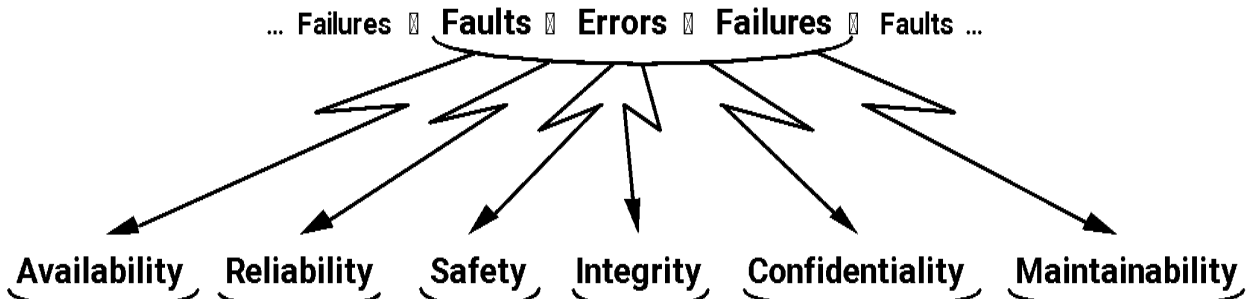
Recalls: Threats



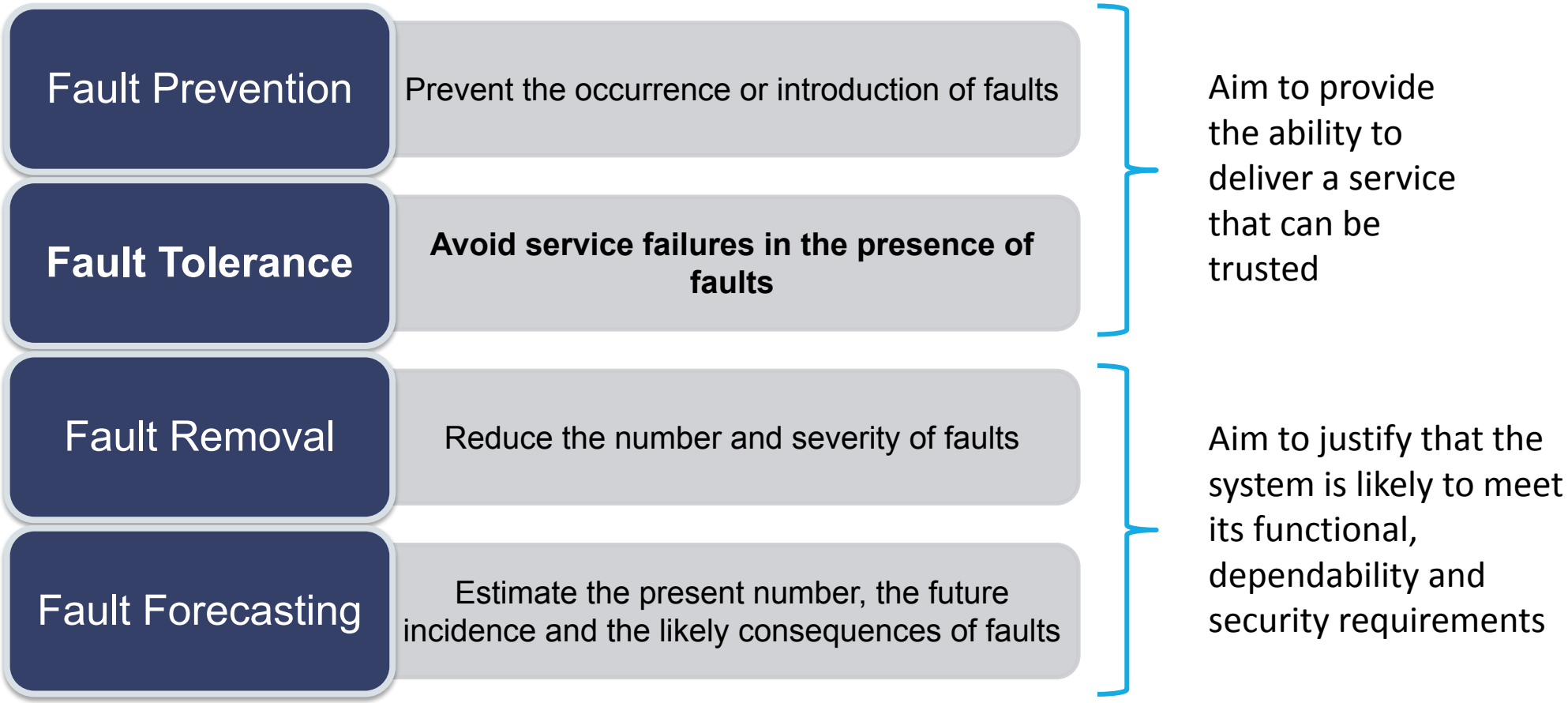
Having a service **failure** means that there exists at least a deviation of the system behavior from the correct service state

The deviation from the correct state is called an **error**

The adjudged or hypothesized cause of an error is called a **fault**



Recalls: Means



Example: Registers

Which means are applied (Prevention, Tolerance, Removal, Forecasting)?

(1,N) regular register

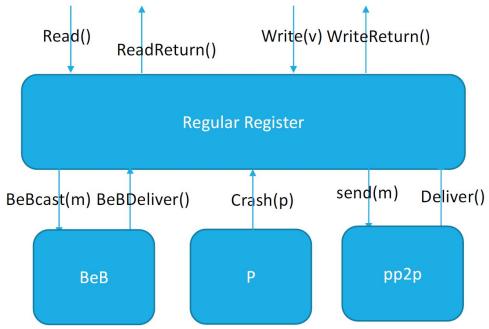
Algorithm 4.1: Read-One Write-All

Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
$val := \perp$;
$correct := \Pi$;
$writeset := \emptyset$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
$correct := correct \setminus \{p\}$;



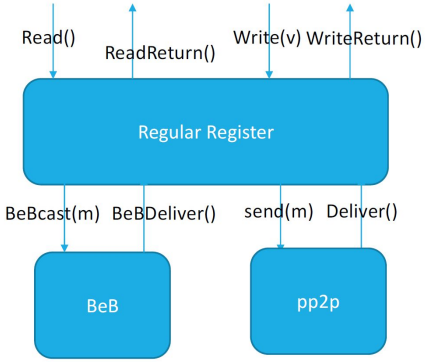
(1,N) regular register

Algorithm 4.2: Majority Voting Regular Register

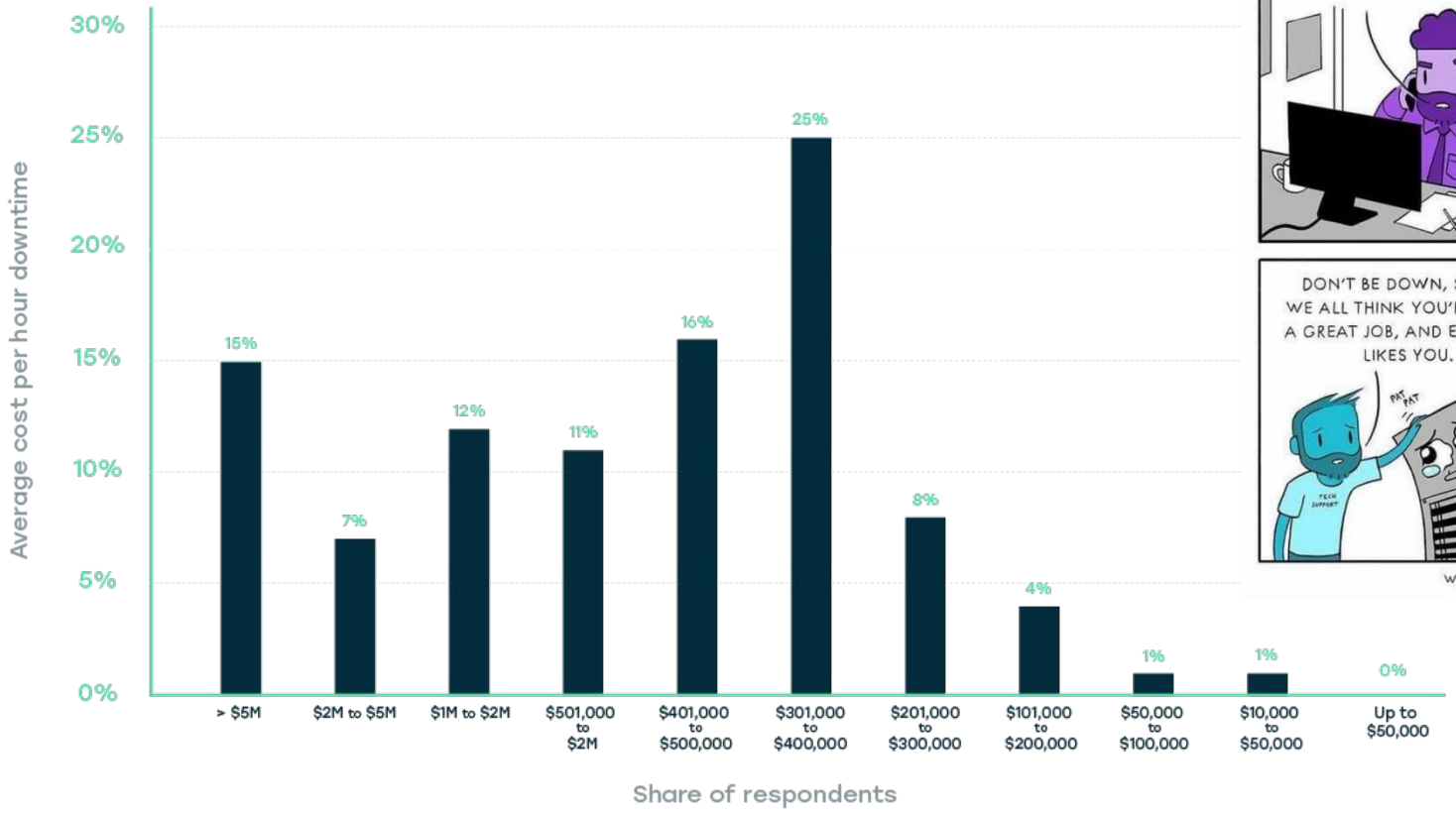
Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.

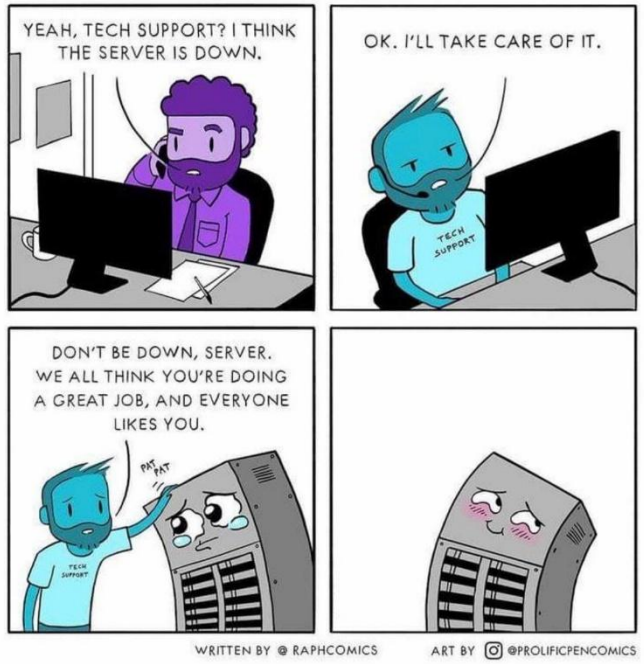
Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;



Downtime Cost





"Instead of minimizing the downtime cost, minimize the chance of it happening!"

https://medium.com/@FedakV/how-much-does-your-server-downtime-cost-it-outsourcing-company-it-svit-9cf2a206f28

Why (again) fault tolerance?

Should we necessarily care about making a system fault tolerant?

Think about the number of involved component and their scale

The probability of a single fault occurrence

X number of “components”

X dependencies (_> cascading effects)

“at Google’s scale failures with one in a million odds are occurring several times a second”

Redundancy

Fault tolerance in computer system is achieved through **redundancy** in hardware, software, information, and/or time.

Redundancy is simply the addition of information, resources, or time beyond what is needed for normal system operation.

Redundancy can provide **additional capabilities** within a system. But, redundancy can have very **important impact** on a system's performance, size, weight and power consumption.

Redundancy

- **Hardware** redundancy is the addition of extra hardware, usually for the purpose either detecting or tolerating faults.
- **Software** redundancy is the addition of extra software, beyond what is needed to perform a given function, to detect and possibly tolerate faults.
- **Information** redundancy is the addition of extra information beyond that required to implement a given function; for example, error detection codes.
- **Time** redundancy uses additional time to perform the functions of a system such that fault detection and often fault tolerance can be achieved. Transient faults are tolerated by this approach.

Redundancy *vs* Diversity

Redundancy contributes to improve the system dependability…

… but it **does not guarantee alone a higher level of security (robustness)**

- If you replicate using exact clones, the attacker will not have a harder time.

To improve security you need to apply diversity in

- Hardware
- Software
- Network configuration

Techniques for Dependability Evaluation

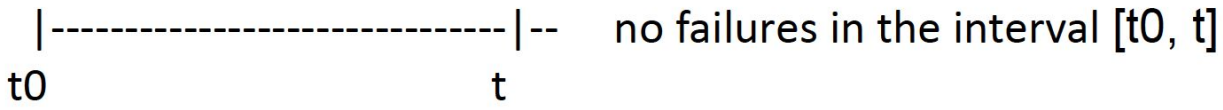
The dependability evaluation of a system can be carried out either:

- **experimentally (heuristic)**: a system prototype is built, and empirical statistical data are used to evaluate the system's metrics:
  - by far more expensive and complex than the analytic approach
  - building a system prototype may be impossible
  - experimental evaluation of dependability requires long observation periods
- **analytical**: dependability metrics are obtained by a mathematical model of the system:
  - mathematical models may not adequately represent the real system's structure or the behavior of its components
  - *simulation models* may be a complementary helpful tool

Definition of Dependability Attributes

Reliability - R(t)

conditional probability that the system performs correctly throughout the interval of time [t0, t], given that the system was performing correctly at the instant of time t0



Availability - A(t)

the probability that the system is operating correctly and is available to perform its functions at the instant of time t

Definition of Dependability Attributes

MTTF (Mean Time To Failure). Expected time before a failure, or expected operational time of a system before the occurrence of the first failure.

$\lambda(t)$ failure rate: the rate at which a component fails

$$\textbf{Failure Rate } (\lambda) = \frac{Number\ of\ Failures}{Operating\ Time}$$

$$\text{MTTF} = \theta = \frac{Operating\ Time\ (Cycle)}{Number\ of\ Failures}$$

$$\lambda = \frac{1}{\theta}$$

Example

correct
20 − 6 = 14 replicas

Let us assume we have **20 independent replicas of a component**

Let them **operate for a period of time**, let us say **1000 hours**

14 · 1000

During the period of 1000 hours, **6 of them fail**. NUMBER of FAILURES

Specifically, they fail at the following hours (assuming t0 = 0h):

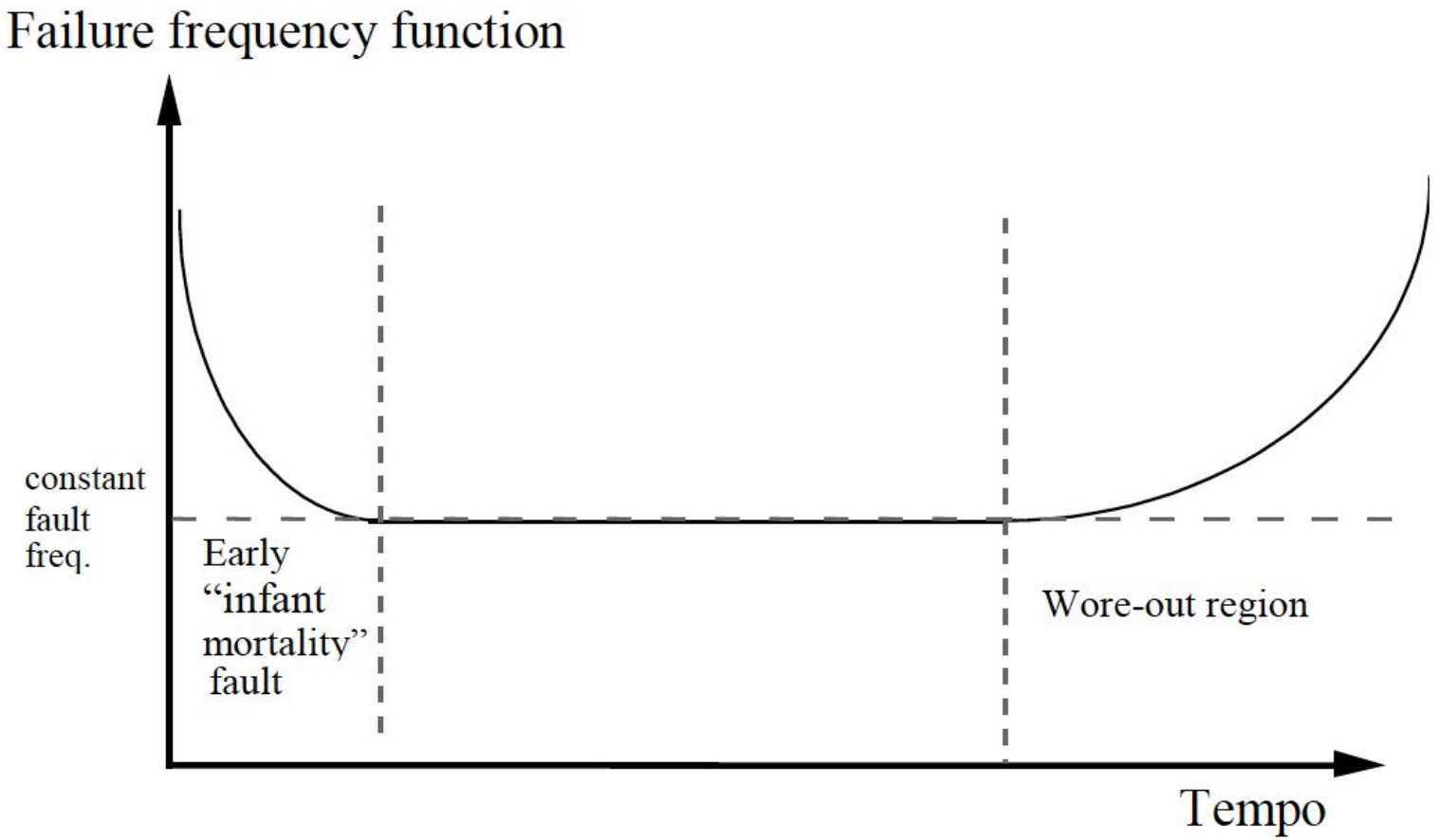
550, 480, 680, 790, 860, 620 OPERATING HOURS

$$\textbf{Failure Rate } (\lambda) = \frac{6}{550 + 480 + 680 + 790 + 860 + 620 + (14 * 1{,}000)} = \frac{6}{17{,}980}$$

$$\textbf{Failure Rate } (\lambda) = 0.000334 \textit{ Failures Per Hour}$$

$$MTTF = \frac{17980}{6} \simeq 2996 \qquad \lambda = \frac{1}{2966} = 0{,}00033 \text{ failures/h}$$

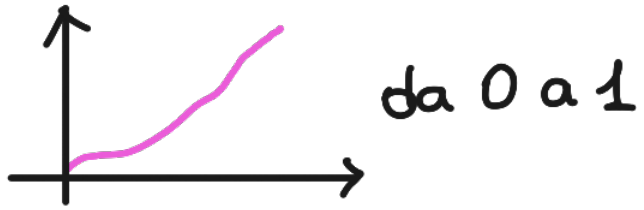
Bathtube Curve – Hardware Failure Frequency



Definitions

***Failure Function Q(t)*:**

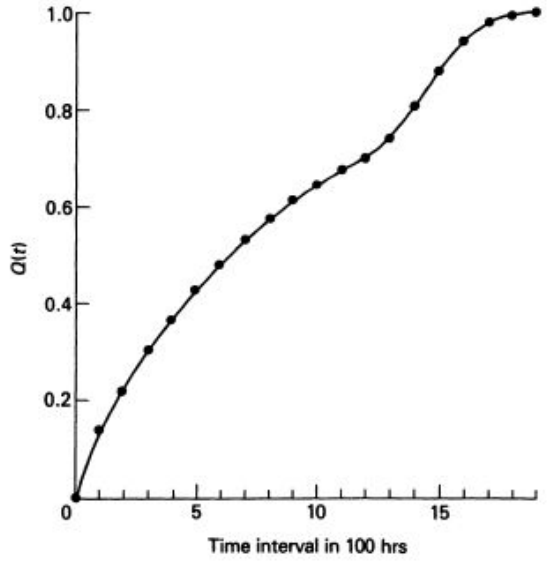
- **probability that a component fails for the first time** in the time interval *(0,t)*
- it's a *cumulative distribution function*:



$Q(t) = 0$ for $t = 0$

$0 \leq Q(t) \leq Q(t + \Delta t)$ for $\Delta t \geq 0$

$Q(t) = 1$ for $t \rightarrow +\infty$



Definitions: Reliability function

Reliability Function R(t):

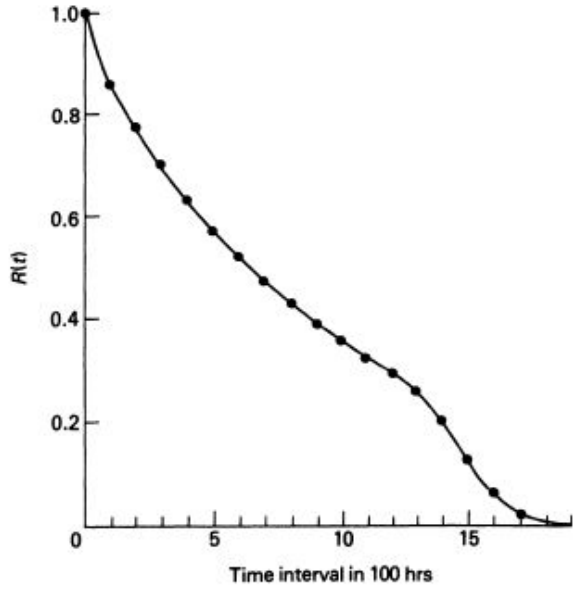
- **probability that a component functions correctly** in the time interval *(0,t)*

$$R(t) = 1 - Q(t)$$

$R(t) = 1$ for $t = 0$

$1 \geq R(t) \geq R(t + \Delta t)$ for $\Delta t \geq 0$

$R(t) = 0$ for $t \rightarrow +\infty$

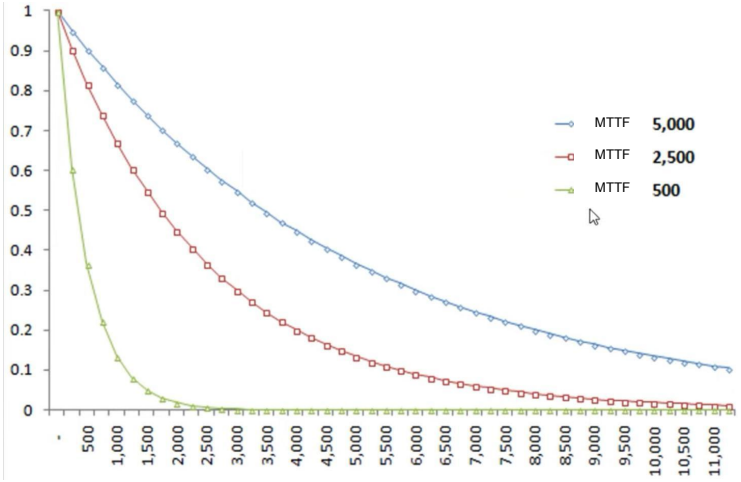


Definitions: Reliability function

$$R(t) = \exp\left[-\int_0^t \lambda(t)\,dt\right]$$

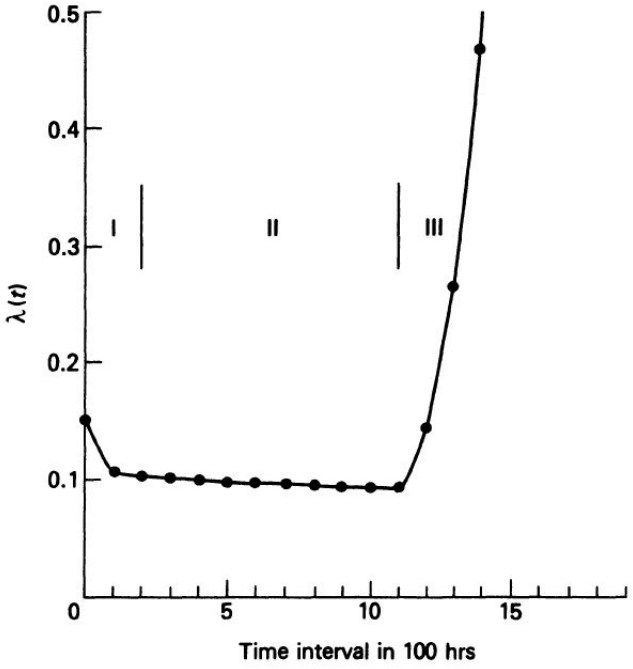
For the special case in which $\lambda(t)$ is a constant and independent of time

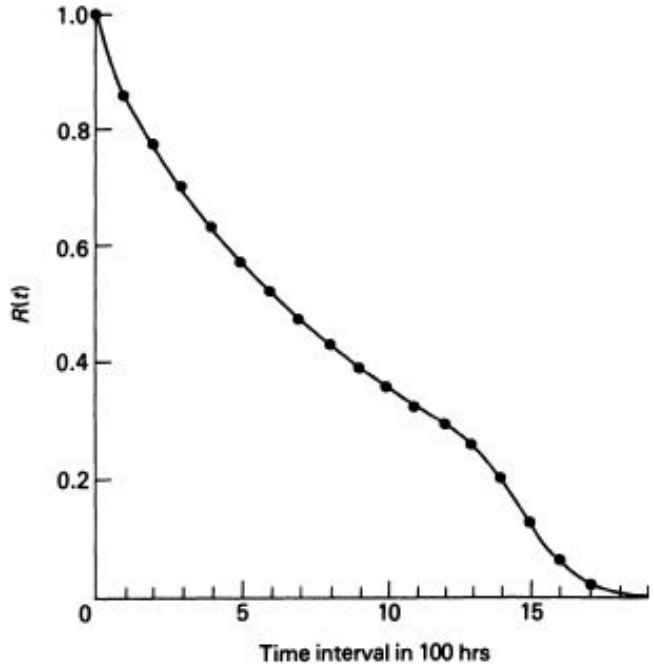
$$R(t) = e^{-\lambda t}$$



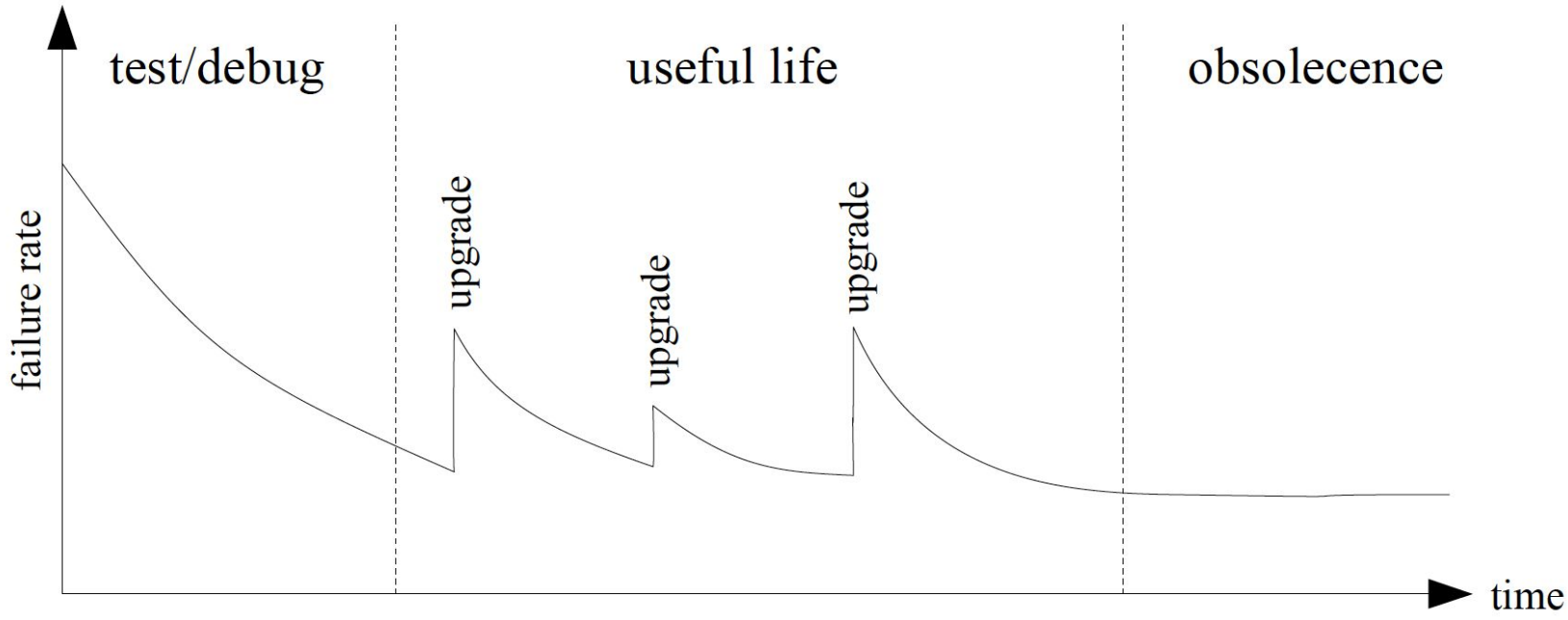
It is possible to evaluate the reliability of a component also cases where $\lambda(t)$ is not constant through the Weibull distribution

Example ([1] Section 6.4)





Software Failure Rate over Lifetime



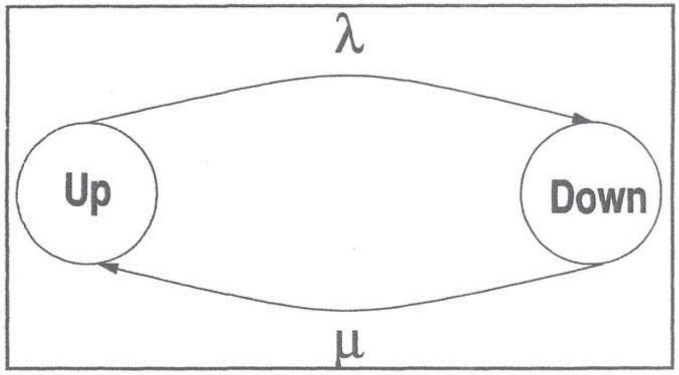
Software Failure Rate over Lifetime

Repairable Systems

In the case of repairable systems, besides the “fault occurrence” event, the event “repairing” or “replacement” of the faulty components has to be considered:

MTBF Mean Time Between Fault: Expected time before a new failure

MTTR (Mean Time To Repair): The average time to repair or replace a faulty entity



Availability

Point Availability, A(t)

Instantaneous (or point) availability is the **probability that a system** (or component) will **be operational** (up and running) at a specific time, t.

Steady State Availability, A(∞)

The steady state availability of the system is the limit of the availability function as time tends to infinity.

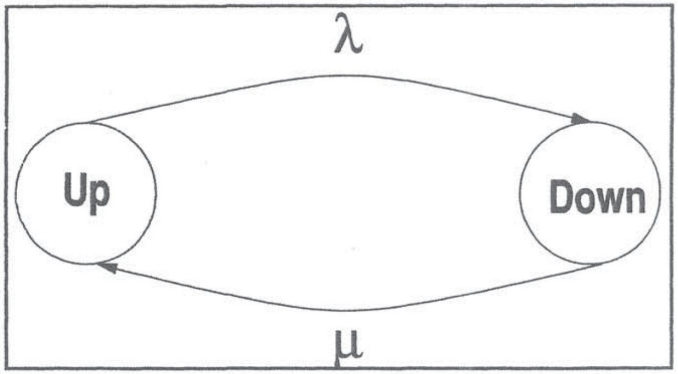
$$A(\infty) = \lim_{t \to \infty} A(t)$$

However, it must be noted that the steady state also applies to **mean availability**.

For ease of simplicity, **we assume that all faults occurrences and repairs are independent among them other**

(Steady-State) Availability Combinatorial Evaluation

Availability is the the fraction of time that a component/system is operational



$$\lambda = \frac{1}{\text{MTBF}} \qquad \mu = \frac{1}{MTTR}$$

$$\lambda \cdot p_{up} = \mu \cdot p_{down}$$

$$p_{up} + p_{down} = 1$$

$$A = p_{up}$$

$$A = \frac{\text{MTBF}}{\text{MTBF} + MTTR}$$

Reliability/Availability Combinatorial Evaluation - Interconnections

Most common component interconnections are:

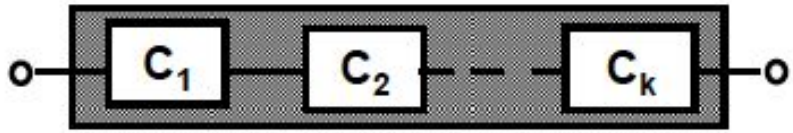
– *Serial*

– *Parallel*

– *Hybrid M out of N*

Serial Interconnection

K entities are serially inteconnected when the functioning of the system depends on the correct functioning of all the K entities.



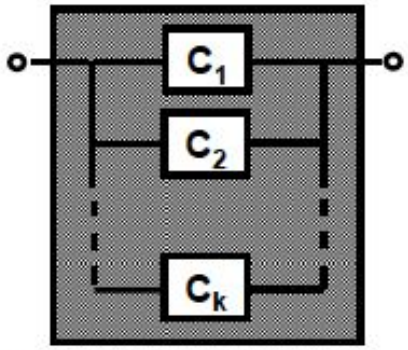
$$A_{serial} = \prod A_i$$

$$R(t) = \prod_{i=1}^{K} R_i(t)$$

"The probability of an event expressed as the intersection of independent event is the product of the probabilities of the independent events"

Parallel Interconnection

k entities are inteconnected in parallel when the functioning of the system is guaranteed even if just a single entity works.

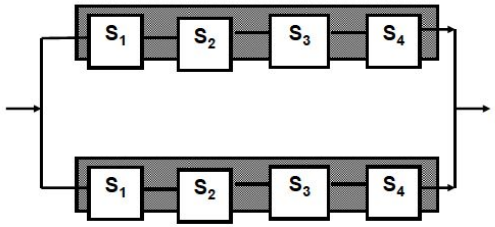


produttoria

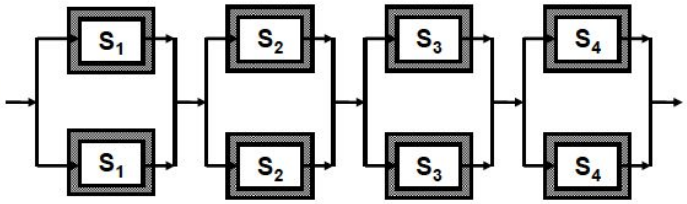
$$A_{parallel} = 1 - \prod(1 - A_i)$$

$$R(t) = 1 - (1 - R_1(t))(1 - R_2(t))...(1 - R_K(t))$$

Evaluation Example



$$A_{d1} = 1 - (1 - A_s)^2$$



$$A_{id} = 1 - (1 - A_i)^2$$

$$A_{d2} = A_{1d} A_{2d} A_{3d} A_{4d}$$

Which solution is more available?

$A_i = 0{,}9$ $A_s = 0{,}6561$ $A_{d1} = 0{,}8817$ $\boldsymbol{A_{d2} = 0{,}9606}$

Hybrid M out of N

The system works as long as there are at least *M* correct entities, namely at most $K = N - M$ entities fail.

availability

$$A = \sum_{i=0}^{K} \binom{N}{i} A_C^{N-i} (1 - A_C)^i$$

reliability

$$R(t) = \sum_{i=0}^{K} \binom{N}{i} R_C^{N-i}(t)(1 - R_C(t))^i$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

SLA – Classes of Availability

Availability Class	Availability	Unavailable (min/year)	System Type
1	90.0%	52,560	Unmanaged
2	99.0%	5,256	Managed
3	99.9%	526	Well-Managed
4	99.99%	52.6	Fault-Tolerant
5	99.999%	5.3	Highly Available
6	99.9999%	0.53	Very Highly Available
7	99.99999%	0.0053	Ultra Available

Standard Techniques to Increase a component Availability

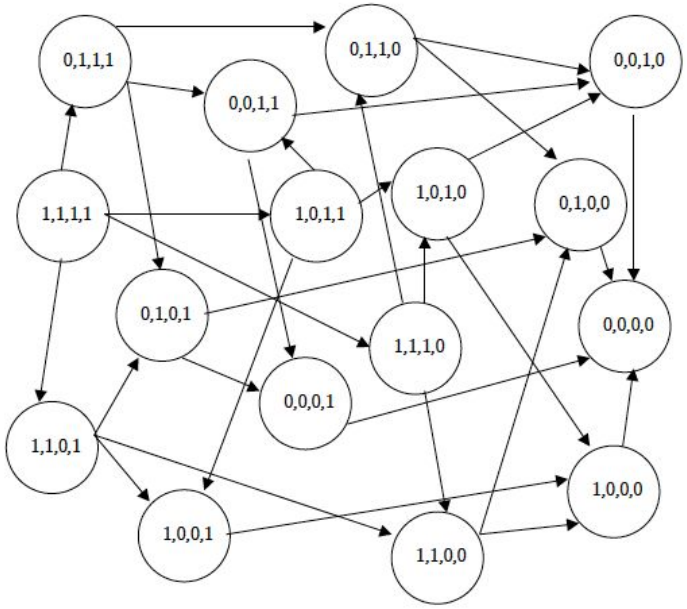
Hot-Standby: A working replica ready to be part of the system

Cold-Standby: A replica that can be started and, after a period of time, is ready to be part of the system.

Performability in a nutshell

Informally: **aggregated measure of availability and performances**

1. **Identify the states** in which the system can be found (with respect to possible faults)
2. **Compute the probability** of being in every of these states
3. **Compute the performance** metrics in every of these states
4. **Aggregate**



Example

A service is provided by 3 types of components (e.g., database, web server, and an application), referred to as A, B, and C. At least one instance of each component must be available to provide the service. Only one instance is deployed for components A and C, while two instances of component B are deployed.

Assuming that all failures and recoveries are independent of each other, that the failure rates of the three components A, B, and C are 0.2, 0.5, and 0.3 failures per day, respectively, and that the average time to repair components A, B, and C is 2 hours, 3 hours, and 1.5 hours, respectively, is the service available at least 98% of the time? If not, is the availability target met by adding an additional instance of one of the components?

Redundancy does not guarantee high level of robustness

↓

HW INFORMATION

SW TIME

Improve security → apply diversity

HW
SW
NET. CONF

Dependability evaluation ① experimentally ② analytical

Dependability attributes → RELIABILITY R(t)

↘ AVAILABILITY A(t)

↓

MTTF expected time before a failure

↓

$\lambda(t)$ failure rate →

$$\lambda(t) = \frac{\text{Number of failures}}{\text{Operating Time}} \qquad \lambda = \frac{1}{MTTF}$$

$$MTTF = \frac{\text{Operating Time}}{\text{\# of failures}}$$

Failure function $Q(t)$ → $= 0$ $t = 0$; $= 1$ $t \to +\infty$

Reliability function $R(t)$ → $= 1$ $t = 0$; $= 0$ $t \to +\infty$

$$R(t) = 1 - Q(t)$$

$$R(t) = \exp\left[-\int_0^t \lambda(t)\,dt\right]$$

$$R(t) = e^{-\lambda t} \longrightarrow \lambda(t) \text{ constant, independent time}$$

Availability

① Point availability $A(t)$

② Steady State Av. $A(\infty)$

$$A(\infty) = \lim_{t \to \infty} A(t)$$

↓

availability is the fraction of time that a component / system is operational

failure rate

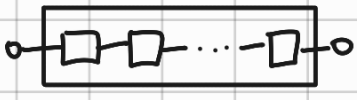
$$\lambda = \frac{1}{MTBF} \qquad \mu = \frac{1}{MTTR}$$

time

$$A = \frac{MTBF}{MTBF + MTTR}$$

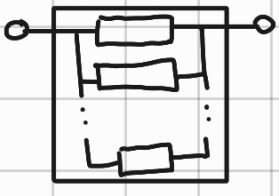
Component interconnections

① SERIAL



$$A_{serial} = \prod A_i$$

② PARALLEL INTERCONNECTION



$$A_{parallel} = 1 - \prod (1 - A_i)$$

③ HYBRID N OUT OF N

Techniques to increase a component availability

1. HOT STANDBY
2. COLD STANDBY

Extra Material

Intro to Chaos Engineering

Solutions to distributed system problems are designed over a set of assumptions (the system model)

_> their correctness rely on these assumptions.

What events can invalidate an assumption?

What is the probability of occurrence of such events?

What happens if part of them are not verified?

Intro to Chaos Engineering

Practically, **how dependability is evaluated and proven today** by the bigger tech companies:

Netflix, Google, Amazon, Microsoft, Dropbox, Yahoo!, Uber, cars.com, Gremlin Inc., University of California, Santa Cruz, SendGrid, North Carolina State University, Sendence, Visa, New Relic, Jet.com, Pivotal, ScyllaDB, GitHub, DevJam, HERE, Cake Solutions, Sandia National Labs, Cognitect, Thoughtworks, and O'Reilly Media …

It follows dependability evaluation and testing

Prerequisites for Chaos Engineering:

- **Be almost sure that the system is resilient to real-world events** such as service failures and network latency spikes
- **Have a monitoring system** that can be used to determine the current state of the system

Intro to Chaos Engineering

It is still a relatively new discipline within software development

“Over the past three years, the question has changed from “Should we do Chaos Engineering?” to “What’s the best way to get started doing Chaos Engineering?””

The story begins at Netflix in 2008,

moving from their datacenter to the cloud: necessitate horizontally scaled components + the single points of failure.

It took eight years to fully extract themselves from the datacenter.

AWS: instances occasionally blinked out of existence with no warning.

The streaming service was facing availability deficits due to the high frequency of instance instability events.

Intro to Chaos Engineering

Chaos Monkey, a very **simple app**: go through a list of clusters, **pick one instance at random from each cluster**, and at some point, **during business hours**, **turn it off** without warning (vanishing instances affected service availability).

Chaos Monkey gave them a way to **proactively test everyone's resilience to the failure and** do it **during business hours** so that people could respond to any potential fallout when they had the resources to do so.

Chaos Monkey forced everyone to be highly aligned toward the goal of being robust enough to handle vanishing instances, it doesn't suggest the solution.

Intro to Chaos Engineering

December 24, 2012, Christmas Eve. AWS suffered a rolling outage of elastic load balancers (ELBs). These components connect requests and route traffic to the compute instances where services are deployed.

Could something similar be built to solve the problem of **vanishing regions**?

AWS wouldn't allow Netflix to take a region offline (something about having other customers in the region) so instead this **was simulated**.

Chaos Kong were routinely conducted to verify that Netflix had a plan of action in case a single region went down.

2015: Netflix had **Chaos Monkey and Chaos Kong**, **working** on the small scale of vanishing instances and the large scale of vanishing regions, respectively.

Dependability of Complex Systems

Chaos Engineering was born of **necessity** in a **complex** distributed software system.

Simple systems	Complex systems
Linear	Nonlinear
Predictable output	Unpredictable behavior
Comprehensible	Impossible to build a complete mental model

A **simple system** is one in which a person can comprehend all of the parts, how they work, and how they contribute to the output.

A complex system, by contrast, has so many moving parts, or the parts change so quickly that no person is capable of holding a mental model of it in their head.

In simple systems, one person, usually an experienced engineer, can orchestrate the work of several engineers.

Birth of Chaos Engineering

Principles of Chaos Engineering https://principlesofchaos.org/

The discipline was officially formalized.

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production.

The facilitation of experiments to uncover systemic weaknesses

Principles lists

The Principles lists five advanced practices that set the gold standard for a Chaos Engineering practice:

- **Build a hypothesis around steady-state behavior**
- **Vary real-world events**
- **Run experiments in production**
- **Automate experiments to run continuously**
- **Minimize blast radius**

1) Hypothesize about steady state

Model with a **steady state** = **Expected values of the business metrics**

Balance for the choice of the metrics _> Relationship between the chosen metric and the impact on the clients

Engineering effort required to collect the data : measurability and accuracy

_> Latency between the metric and the ongoing behavior of the system (almost) real time

1. Start by defining "**steady state**" as **some measurable output of a system that indicates normal behavior**.

2. **Hypothesize that this steady state will continue** in both the control group and the experimental group.

_> focusing on the way the system is expected to behave and capturing that in a measurement.

2) Vary Real-World Events

Introduce variables that reflect real-world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.

From the distributed system perspective, almost all interesting availability experiments can be driven by **affecting latency or response type**.

Terminating an instance is a special case of infinite latency.

It follows that most availability experiments can be constructed with a mechanism to vary latency and change status codes.

_> Try to disprove the hypothesis by looking for a difference in steady state

3) Run Experiments in Production

Experimentation teaches you about the system you are studying.

If you are experimenting on a Staging environment, then you are building confidence in that environment.

This principle is not without controversy. Certainly, in some fields **there are regulatory requirements that preclude the possibility of affecting the Production systems**. In some situations, there are insurmountable technical barriers to running experiments in Production.

In most situations, it makes sense to start experimenting on a Staging system, and gradually move over to Production once the kinks of the tooling are worked out.

Experimentation, not testing

Testing, strictly speaking, **does not create new knowledge**.

Testing requires that the engineer writing the test knows specific properties about the system that they are looking for.

Complex systems are opaque to that type of analysis.

Tests make an assertion, based on existing knowledge, and then running the test collapses the valence of that assertion, usually into either **true or false**.

Experimentation, on the other hand, **creates new knowledge**.

Experiments propose a hypothesis, and as long as the hypothesis is not disproven, confidence grows in that hypothesis. If it is disproven, then we learn something new.

4) Automate Experiments to Run Continuously

To cover a larger set of experiments than humans can cover manually.

Automation provides a means to scale out the search for vulnerabilities that could contribute to undesirable systemic outcomes.

To empirically verify our assumptions over time, as unknown parts of the system are changed.

5) Minimize Blast Radius

By using a **tightly orchestrated control** experiments can be constructed in such a way that **the impact of a disproved hypothesis on customer traffic in Production is minimal**.

About Minimize Blasting Radius

The **Chernobyl disaster** of 1986 is one of the most infamous examples of a catastrophic industrial failure. **Workers** at the nuclear power plant **were carrying out an experiment** to see whether the core could still be sufficiently cooled in the event of a power loss. Despite the potential for grave consequences, safety personnel were not present during the experiment, nor did they coordinate with the operators to ensure that their actions would minimize risk.

During the experiment, what was supposed to just be a shutdown of power led to the opposite; the power surged and led to several explosions and fires, leading to radioactive fallout that had catastrophic consequences for the surrounding area. Within a few weeks of the failed experiment, thirty-one people died, two of whom were workers at the plant and the rest of whom were emergency workers who suffered from radiation poisoning.

5) Minimize Blasting Radius

Try to **architect your experimentation** in a way that lets you have a **high level of granularity**, and target the smallest possible unit, whatever that is in your system.

You should always have an eye on your metrics and be able to quickly terminate an experiment if anything seems to be going wrong in unexpected and impactful ways.

“Hope is not a strategy”

Disaster testing helps prove a system’s resilience when failures are handled gracefully and **exposes reliability risks in a controlled fashion** when things are less than graceful.

Exposing reliability risks during a controlled incident allows for thorough analysis and preemptive mitigation, as **opposed to waiting for problems to expose themselves** by means of circumstance alone, when issue severity and time pressure amplify missteps and force risky decisions based on incomplete information

Becoming familiar with and **preparing for uncommon combinations of failures**.

“Hope is not a strategy”

As engineers we are frequently unaware of the implicit assumptions that we build into systems until these assumptions are radically challenged by unforeseen circumstances.

There are events purposefully imposed: software updates, OS patches, regular hardware and facility maintenance, and other threats that commonly occur.

Need to be open to unexpected consequences, and **practice recovery as best we can**.

One of the **most difficult aspects** of running a successful Chaos Engineering practice is proving that the **results have business value**.

Getting to Basic Fault Tolerance

First and foremost, **keep spare capacity online**.

Once you have spare capacity available, consider **how to remove malfunctioning computers from service automatically**. Don't stop at automatic removal,

Carry on to automatic replacement.

Replacement instances must enter service in less than the mean time between failures.

DiRT (Google)

DiRT tests must have **no service-level** objective breaking **impact on external systems or users**

Service levels should not be degraded below the standard objectives already set by the owning teams.

A key point of Google's approach to Chaos Engineering is that we prefer "controlled chaos" when possible.

When designing a test, you should **weigh the costs** of internal user disruptions and loss of goodwill carefully **against what you stand to learn**.

Example of Software Experiments

- Run at service levels
- Run without dependencies
- People outages
- Release and rollback
- Incident management procedures
- Datacenter operations
- Capacity management
- Business continuity plans
- Data integrity
- Networks
- Monitoring and alerting
- Reboot everything

Prioritize Experiments

There is **no way to identify all possible incidents** in advance, much less to cover them all with chaos experiments.

To maximize the efficiency of your effort to reduce system failures you should **prioritize different classes of incidents**.

How often does this happen?

How likely are you to deal with the event gracefully?

There will be events that you don't need to investigate, since you accept total loss. On the other hand, there will be events you certainly won't allow to take you down. Prioritize experiments against those to make sure your assumptions on reliability and robustness are being met conscientiously and continuously.

How likely is this to happen? once the event is imminent, you must prioritize testing against such events ahead of anything else.

Your answers to these questions should be based on the business needs of your prod- uct or service. The satisfaction of reliability goals with respect to business needs should match expectations set with your customers. It's always best to explicitly check with stakeholders whether your plans for reliability land within acceptable margins.

The Case for **S**ecurity **C**haos **E**ngineering

SCE allows teams to proactively, safely discover system weakness before they disrupt business outcomes.

There is no single root cause for failure

The reality is that **most of the malicious** code such as viruses, malware, ransomware, and the **like habitually take advantage of low-hanging fruit**.

This can take the form of weak passwords, default passwords, outdated software, unencrypted data, weak security measures in systems, and most of all they take advantage of **unsuspecting humans' lack of understanding of how the complex system actually functions**.

The Case for Security Chaos Engineering

Remove the Low-Hanging Fruit!

If we always operated in a culture where we expect humans and systems to behave in unintended ways, then perhaps we would act differently and have more useful views regarding system behavior.

Instead of simply reacting to failures, the security industry has been overlooking valuable chances to further understand and nurture incidents as opportunities to proactively strengthen system resilience.

The Case for **S**ecurity **C**haos **E**ngineering

The most common way we discover security failures is when a security incident is triggered.

Security incidents are not effective signals of detection, because at that point it's already too late.

SCE introduces observability plus rigorous experimentation to illustrate the security of the system. Observability is crucial to generating a feedback loop.

Injecting security events into our systems helps teams understand how their systems function as well as increase the opportunity to improve resilience.

By running security experiments continuously, we can evaluate and improve our understanding of unknown vulnerabilities before they become crisis situations.

The Case for **S**ecurity **C**haos **E**ngineering

SCE addresses a number of gaps in contemporary security methodologies such as Red and Purple Team exercises.

The goal of these exercises is the collaboration of offensive and defensive tactics to improve the effectiveness of both groups in the event of an attempted compromise.

These techniques remain valuable but differ in terms of goals and techniques. Combined with SCE, they provide a more objective and proactive feedback mechanism to **prepare a system for an adverse event** than when implemented alone.

SCE utilizes simple isolated and controlled experiments instead of complex attack chains involving hundreds or even thousands of changes.

Security Chaos Engineering Tool: ChaoSlingr

ChaoSlingr was originally designed to operate in Amazon Web Services (AWS).

It proactively introduces known security failure conditions through a series of experiments to determine how effectively security is implemented.

ChaoSlingr was developed to uncover, communicate, and address significant weaknesses proactively, before they impacted customers in production.

ChaoSlingr demonstrates how Chaos Engineering experiments can be constructed and executed to provide security value in distributed systems.

The majority of organizations utilizing ChaoSlingr have since forked the project and constructed their own series of security chaos experiments using the framework provided by the project as a guide.

Final Comments

All of these advanced **principles** are presented to **guide and inspire, not to dictate**.

It is currently under introduction of several no tech realities

Chaos Engineering serves as an instrument to **discover real systems behavior and real dependency graphs**.

Chaos Engineering is also applied extensively in companies and industries that aren't considered digital native, like large financial institutions, manufacturing, and healthcare.

Remember you're not doing anything that would not happen in real life.

Chaos Engineering References

_> Ali Basiri, Niosha Behnam, Rudd de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, Casey
Rosenthal, "**Chaos Engineering**," in IEEE Software, vol. 33, no. 3, pp. 35-41, May-June 2016,
https://doi.org/10.1109/MS.2016.60

_> Casey Rosenthal, Lorin Hochstein, Aaron Blohowiak, Nora Jones, Ali Basiri,
Chaos Engineering, O'Reilly Media, Inc., August 2017

_> Casey Rosenthal and Nora Jones,
Chaos Engineering – System Resiliency in practice, O'Reilly Media, Inc., April 2020,

_> https://www.gremlin.com/state-of-chaos-engineering/2021/

References

- Availability, MTTR, and MTBF for SaaS Defined https://medium.com/@daveowczarek/availability-mttr-and-mtbf-for-saas-defined-66b618ac1533

- A. Avizienis, J.-C. Laprie, B. Randell, C. E. Landwehr: *Basic Concepts and Taxonomy of Dependable and Secure Computing*. https://ieeexplore.ieee.org/document/1335465/

- Billinton, Allan - Reliability Evaluation of Engineering Systems

- D. A. Menascé, V. A. F. Almeida: Capacity Planning for Web Services: metrics, models and methods. Prentice Hall, PTR