Introduction to Distributed Systems

A broad overview of distributed systems!

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Folien: https://delors.github.io/ds-introduction/folien.en.rst.html
https://delors.github.io/ds-introduction/folien.en.rst.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues

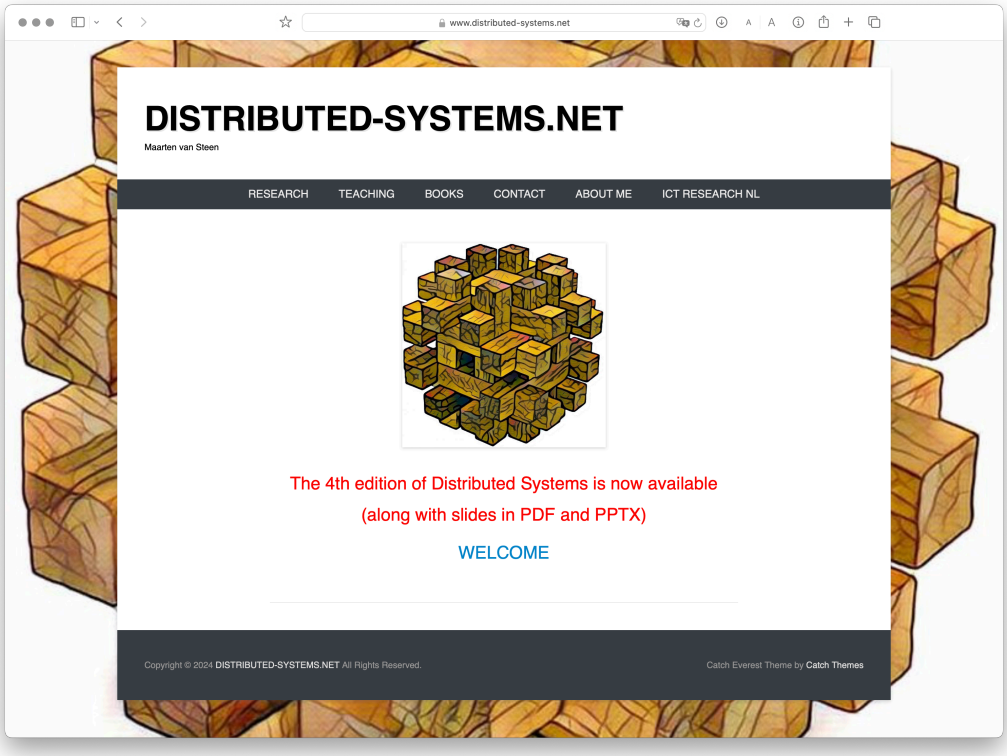
This slide set is based in parts on the following sources:

a. Maarten van Steen (material related to his book on *Distributed Systems*)
b. Henning Pagnia (based on his lecture *Verteilte Systeme*).

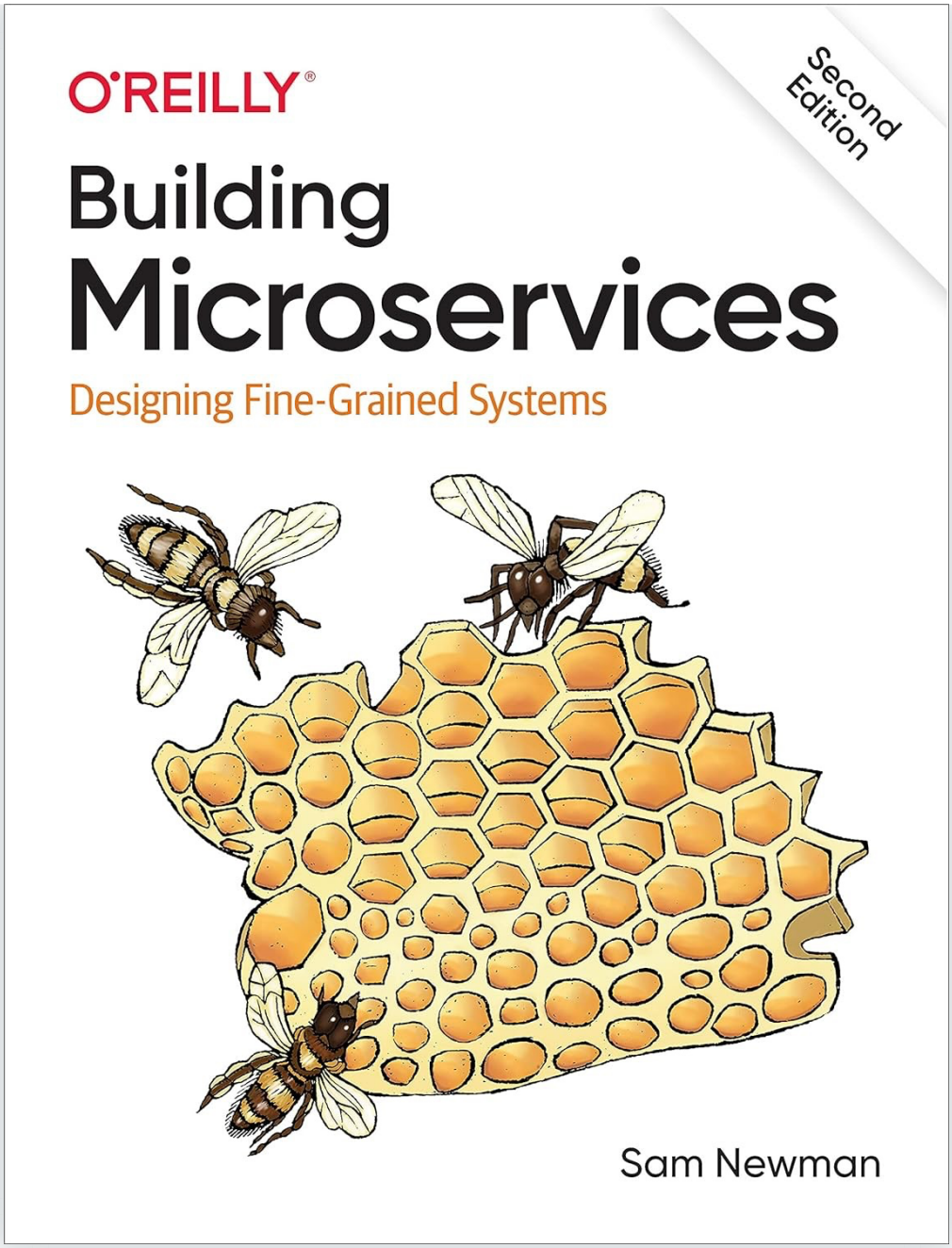
All errors are my own.


Agile
Microservices
CQRS
Distributed Applications
Clean Architecture
Unit Testing
SOLID
Container
Interface Segregation Principle
Dependency Inversion Principle
Message Queues
Layered Architecture
Orchestration
Changing Requirements
Hexagonal Architecture
Domain-driven Design
Command-Query Responsibility Segregation
Command-Query Separation
Ports & Adapters
Event Sourcing
RESTful
REST
EDA
DDD
CQS
3-Tier

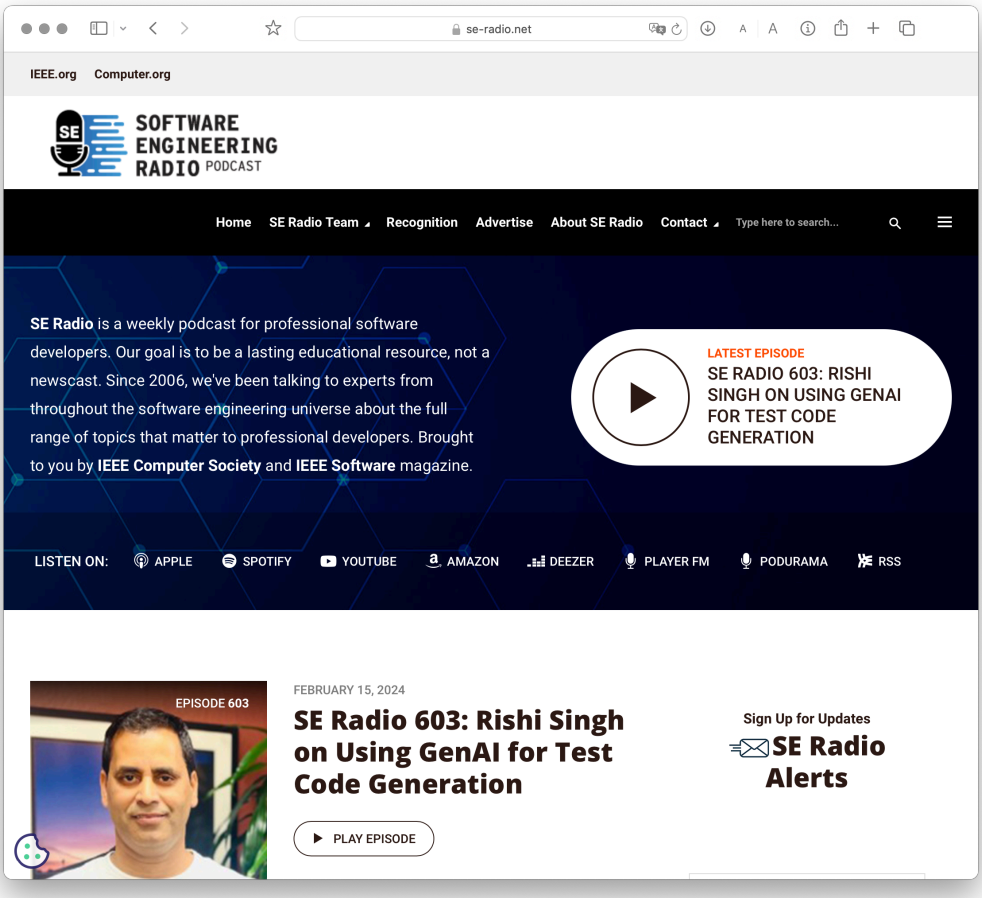
Recommended Literature



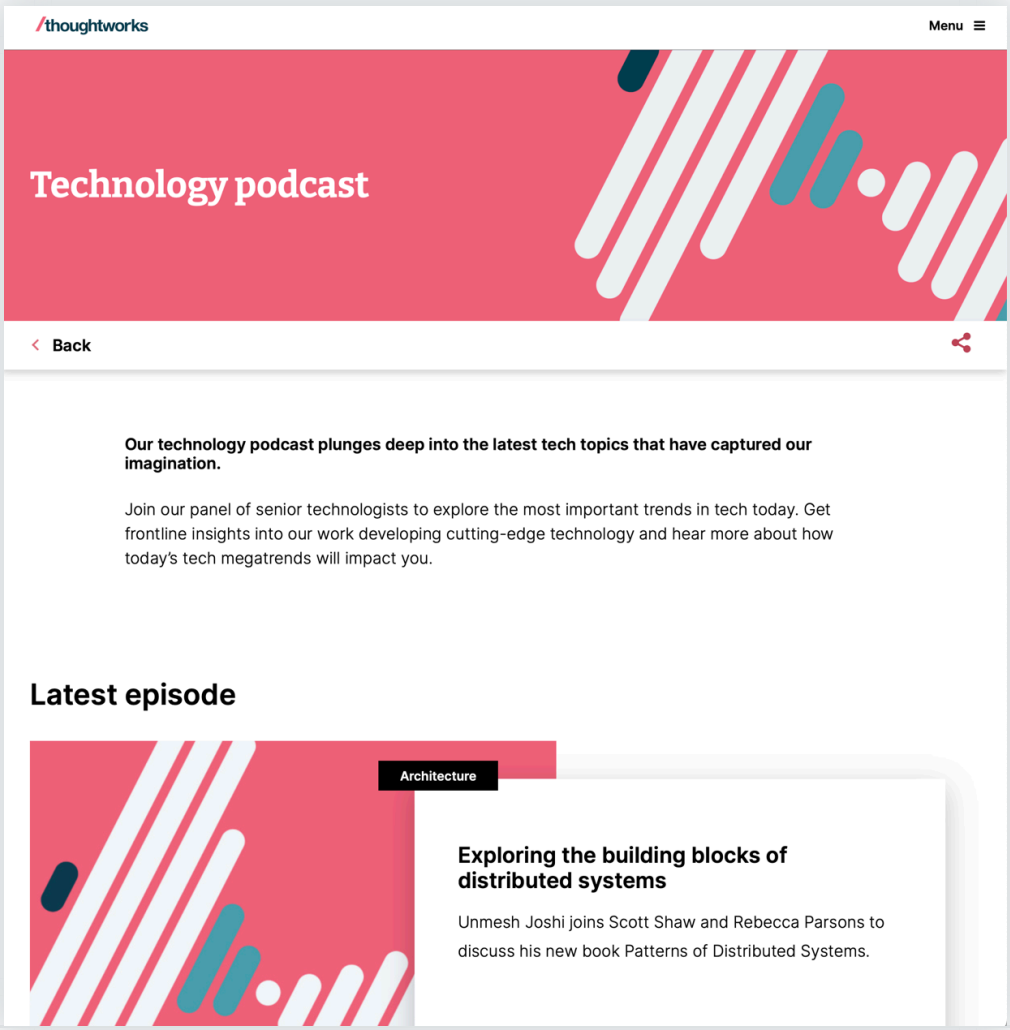
Supplemental material for interested students:


O'REILLY
Second Edition
Building
Microservices
Designing Fine-Grained Systems
Sam Newman

Recommended Podcast: SE-Radio

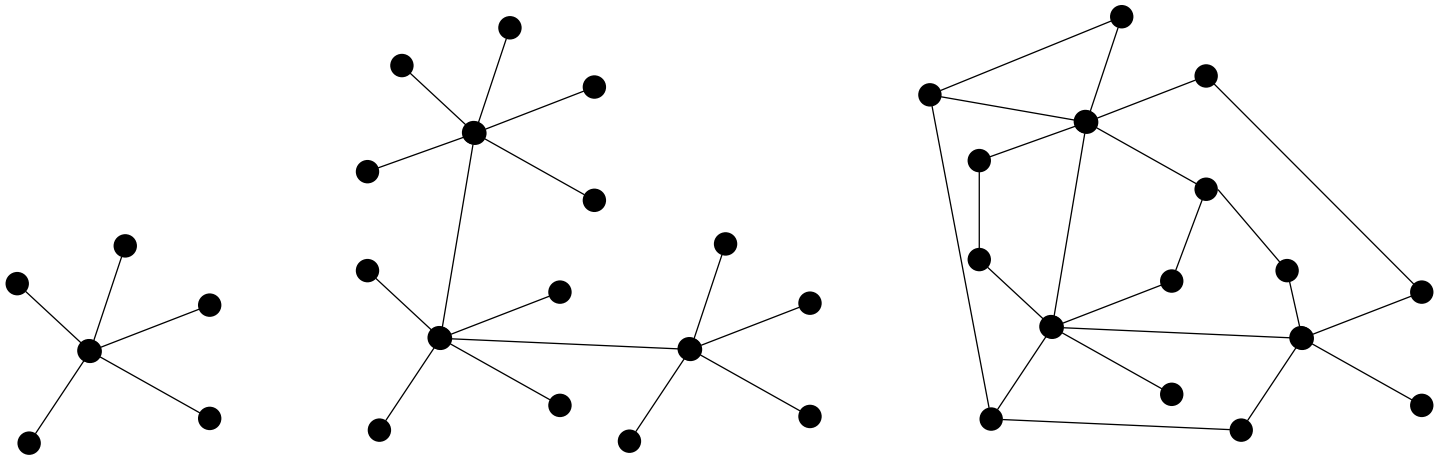


Recommended Podcast: Thoughtworks Technology Podcast



1. Distributed Systems - Definition and Properties

Distributed vs. Decentralized



Zwei Ansichten zur Realisierung verteilter Systeme

- **Integrative view**: Connection of existing (locally) networked computer systems to form a larger system.
- **Expansive view**: an existing networked computer system is expanded to include additional computers.

Definition

- A **decentralised system** is a networked computer system in which processes and resources are *necessarily* distributed across multiple computers.
- A **distributed system** is a networked computer system in which processes and resources are *sufficiently* distributed across several computers.

Common misunderstandings regarding centralised systems

1. **Centralized solutions do not scale**

   A distinction must be made between logical and physical centralization.

   Example

   The *Domain Name System (DNS)*:

   - logisch zentralisiert
   - dezentralisiert über mehrere Organisationen
   - physisch (massiv) verteilt

2. **Centralized solutions have a single point of failure**

   Generally not true (e.g. DNS).

   A single possible source of error is often...

   - easier to manage
   - easier to make more robust

Warning

There are many, poorly founded misconceptions about, for example, scalability, fault tolerance or security. We need to develop skills that make it easy to understand distributed systems in order to avoid such misunderstandings.

Perspectives on Distributed Systems

Distributes systems are complex.

Architectures:	What architectures and "architectural styles" are there?
Processes:	What kind of processes are there and what are their relationships?
Communication:	What options are there for exchanging data?
Coordination:	How are the involved systems coordinated?
Naming:	How do you identify resources?
Consistency and replication:	What trade-offs need to be made in terms of data consistency, replication and performance?
Fault tolerance:	How can operations be maintained even in the event of partial failures?
Security:	How can authorized access to resources be guaranteed?

Design-goals of Distributed Systems

- Shared Usage of Resources
- Distribution Transparency
- Openness
- Scalability

Shared Usage of Resources - Examples

- Cloud-based shared storage and files
- Peer-to-peer supported multimedia streaming
- Shared email services (e.g. outsourced email systems)
- Shared web hosting (e.g. *content distribution networks*)

Definition

Definition

Distribution Transparency

Transparency describes the property that a distributed system attempts to hide the fact that its processes and resources are physically distributed across multiple computers that may be separated by large(r) distances.

The distribution transparency is realized by many different techniques of the so-called *middleware* - a layer between applications and operating systems.

Aspects of Distribution Transparency

Data access	hide differences in data representation and the type of access to a local or remote object
Location of data storage	hide where an object is located
Relocation	hide that an object may be moved to another location while in use
Migration	hide that an object may be moved to another location
Replication	hide that an object is replicated
Concurrency	hide that an object may be shared by several independent users
Fault transparency	hide the failure and recovery of an object

Datendarstellung: Big-Endian vs. Little-Endian; ASCII vs. Iso-Latin 8859-1 vs. UTF-8

Degree of achievable Distribution Transparency

Observation

Complete distribution transparency cannot be achieved.

However, a high level of distribution transparency can result in high costs.

- There are communication latencies that cannot be hidden.
- It is (theoretically and practically) impossible to completely hide network and node failures.
- You cannot distinguish a slow computer from a failed computer.
- You can never be sure that a server was actually performing an operation before it crashed.
- "Complete transparency" costs performance and exposes the distribution of the system.
  - Keeping the replicas exactly on the same level as the master takes time
  - Write operations are immediately transferred to the hard drive for fault tolerance

Disclosing Distribution can bring Advantages

- Use of location-based services (E. g. to enable finding friends nearby.)
- When dealing with users in different time zones
- When it is easier for a user to understand what is going on

  (E.g. if a server does not respond for a long time, it can be reported as down).

Observation

Distribution transparency is a noble goal, but often difficult to achieve and frequently not worth striving for.

Open Distributed Systems

Definition

An open distributed system offers components that can easily be used by other systems or integrated into other systems.

An open distributed system itself often consists of components that originate from elsewhere.

Open distributed systems must be able to interact with services of other (open) systems, regardless of the underlying environment:

- they should implement well-defined interfaces correctly
- they should be able to interact easily with other systems
- they should support the portability of applications
- they should be easily extensible

Authentication services are one example. They can be used by many different applications.

Policies vs. Mechanisms

🇺🇸 *Policies vs. Mechanisms* ≙ 🇩🇪 *Vorgaben/Richtlinien vs. Umsetzungen*

Policies when implementing openness

- What level of consistency do we need for data in the client cache?
- What operations do we allow downloaded code to perform?
- Which QoS requirements do we adapt in the presence of fluctuating bandwidths?
- What level of secrecy do we need for communication?

Mechanisms to support openness

- Enabling the (dynamic) setting of caching policies
- Support of different trust levels for mobile code
- Provisioning of adjustable QoS parameters per data stream
- Provisioning of various encryption algorithms

The hard coding of policies often simplifies administration and reduces the complexity of the system. However, it comes at the price of less flexibility.

Security in Distributed Systems - Security Objectives

Observation

A distributed system that is not secure is not reliable.

Foundational security objectives

Confidentiality: Information is only passed on to authorized parties.

Integrity: Changes to the values of a system may only be made in an authorized manner.

Together with the third security objective: **availability**, these three protection objectives form the CIA triad of information security: Confidentiality, Integrity, and Availability.

Security in Distributed Systems - Authorization, Authentication, Trust

Authentication: Process for verifying the correctness of a claimed identity.

Authorization: Does an identified unit have the correct access rights?

Trust: A component can be certain that another component will perform certain actions in accordance with expectations.

Security - Encryption and Signatures

It is essentially about encrypting and decrypting data (X) with the help of keys.

$E(K,X)$ means that we **e**ncrypt the message X with the key K.

$D(K,X)$ denotes the inverse function that **d**ecrypts the data.

Symmetric Encryption

The encryption key is identical to the decryption key; the same key K is used for both operations.

$X = D(K,E(K,X))$

Asymmetric Encryption

We distinguish between private (PR) and public keys (PU) ($PU \neq PR$). A private and a public key always form a pair. The private key must always be kept secret.

Encrypting Messages

Alice sends a message to Bob using Bob's public key.

$$Y = E(PU_{Bob}, X)$$
$$X = D(PR_{Bob}, Y)$$

Signing Messages

Alice signs (S) a message with her private key.

$$Y = E(PR_{Alice}, X)$$
$$X = D(PU_{Alice}, Y)$$

Security - Secure Hashing

A secure hash function $Digest(X)$ returns a character string of fixed length (H).

- Any change - no matter how small - to the input data results in a completely different character string.
- With a hash value, it is mathematically impossible to find the original message X based on $Digest(X)$.

Signing Messages

Alice signs a message X with her private key.

$$\text{Alice:} [E(PR_{\text{Alice}}, H = Digest(X)), X]$$

Bob checks the message X for authenticity:

$$\text{Bob:} D(PU_{\text{Alice}}, H) \stackrel{?}{=} Digest(X)$$

🇩🇪 *Sicheres Hashing* ≙ 🇺🇸 *Secure Hashing*

Question

1.1. Encryption with Public-Private Keys/Asymmetric Encryption

If Alice sends Bob a message encrypted with Bob's public key, what security problem could arise?

25

Scalability in Distributed Systems

We can distinguish at least three types of scalability:

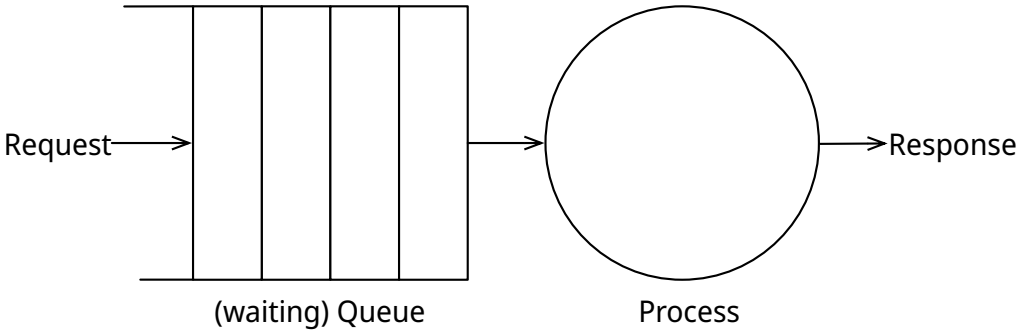
- Number of users or processes (size scalability)
- Maximum distance between nodes (geographical scalability)
- Number of administrative domains (administrative scalability)

Scalability in terms of size can often be achieved by using more and more powerful servers that are operated in parallel.

Geographical and administrative scalability is often a greater challenge.

Analysis of the Scalability of Centralized Systems

A centralized service can be modelled as a simple queuing system:



Assumptions

The queue has an infinite capacity, i.e. the arrival rate of requests is not influenced by the current length of the queue or by what is currently being processed.

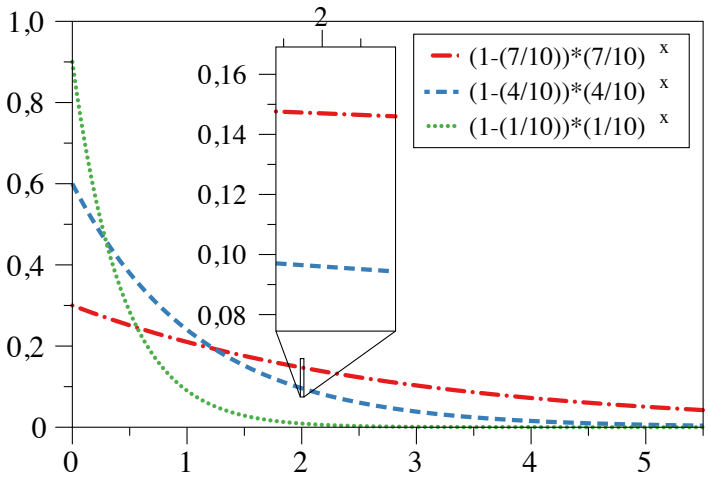
- Arrival rate of requests:

  λ *(requests per second)*
- Processing capacity of the service:

  μ *(requests per second)*

  Proportion of time with x requests in the system:

$$p_x = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^x$$



 # Requests in process and in queue

For example, the proportion of time in which the computer is *idle* (i. e. p_0) is : 90 %, 60 % and 30 %.

U is the proportion of time in which a service is utilized:

$$U = \sum_{x>0} p_x = 1 - p_0 = \frac{\lambda}{\mu} \Rightarrow p_x = (1-U)U^x$$

Note

x = # Anfragen im Sys.

$$p_x = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^x$$

Average number of requests:

$$\begin{aligned} \bar{N} &= \sum_{x\geq 0} x \cdot p_x = \sum_{x\geq 0} x \cdot (1-U)U^x \\ &= (1-U)\sum_{x\geq 0} x \cdot U^x = \frac{(1-U)U}{(1-U)^2} = \frac{U}{1-U} \end{aligned}$$

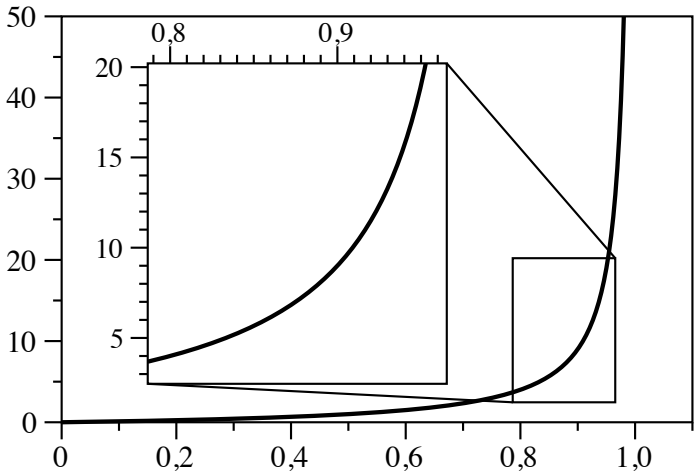
Average throughput:

$$X = \underbrace{U \cdot \mu}_{\text{utilized}} + \underbrace{(1-U)\cdot 0}_{\text{unused}} = \frac{\lambda}{\mu} \cdot \mu = \lambda$$

For an infinite geometric series with the quotient U applies:

$$\sum_{k\geq 0} k \cdot U^k = \frac{U}{(1-U)^2}$$

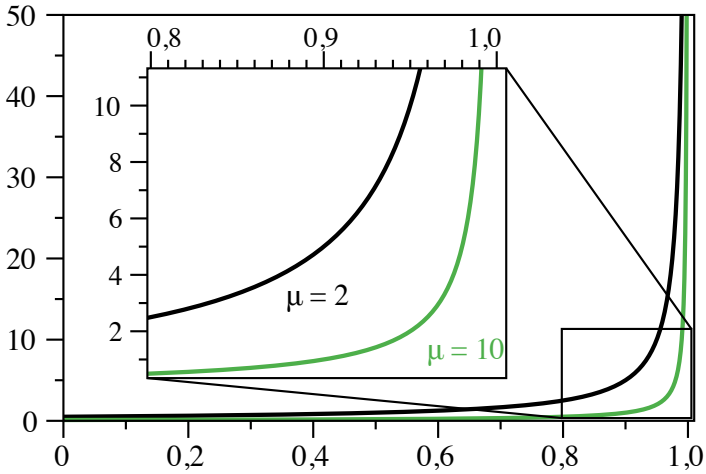
Representation of the average number of requests in the system depending on the utilization U:



The response time is the total time taken to process a request

$$\begin{aligned} R &= \frac{\bar{N}}{X} = \frac{S}{1-U} \\ &\Rightarrow \frac{R}{S} = \frac{1}{1-U} \end{aligned}$$

with $S = \frac{1}{\mu}$ for the average service time.



- If U is small, the response time is close to 1, i.e. a request is processed immediately.
- If U increases to 1, the system comes to a standstill.

Problems of Geographical Scalability

- Many distributed systems assume synchronous client-server interactions and this prevents a transition from LAN to WAN. Latency times can be prohibitive if the client has to wait a long time for a request.
- WAN connections are often unreliable by nature.

Problems of Administrative Scalability

Observation

Conflicting guidelines in terms of usage (and therefore payment), administration and security.

Example

- Grid computing: shared use of expensive resources across different domains.
- Shared devices: How to control, manage and utilize a shared radio telescope designed as a large-scale shared sensor network?

Exception

Various peer-to-peer networks [1] where end users collaborate rather than administrative units:

- File sharing systems (e.g. based on BitTorrent)
- Peer-to-peer telephony (early versions of Skype)

[1] Here, "peer" is to be understood as a network of equal computers.

Approaches to achieve Scaling

Hiding communication latencies through:

- Use of asynchronous communication
- Use of separate *handlers* for incoming responses

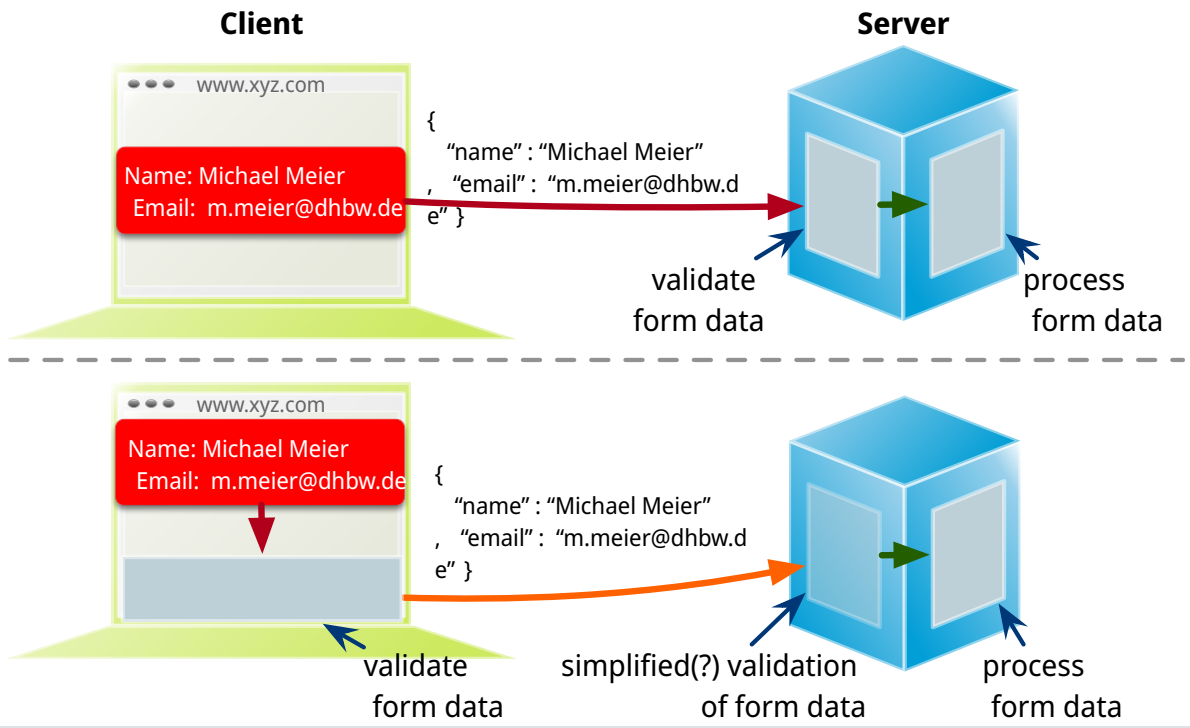
Observation

However, this model is not always applicable.

Partitioning of data and calculations across multiple computers.

- Relocation of calculations to clients
- Decentralized naming services (e.g. DNS)
- Decentralized information systems (e.g. WWW)

Shifting Calculations to Clients



Scaling via Replication and Caching

Use of replication and caching to make copies of data available on different computers.

- replicated file servers and databases
- mirrored websites
- Web caches (in browsers and proxies)
- File caching (on server and client)

Challenges of Replication

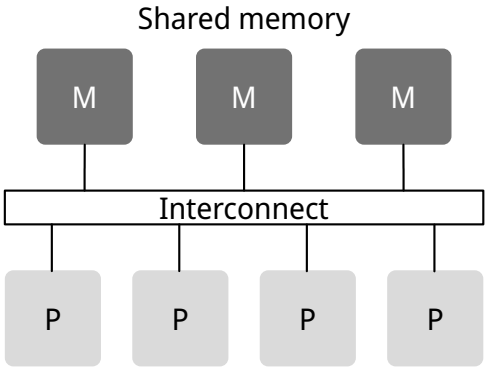
- Multiple copies (cached or replicated) inevitably lead to inconsistencies. Changing one copy means that this copy differs from the others.
- To achieve consistency, global synchronization is required for every change.

> **Attention!**
> Global synchronization rules out solutions on a large scale.

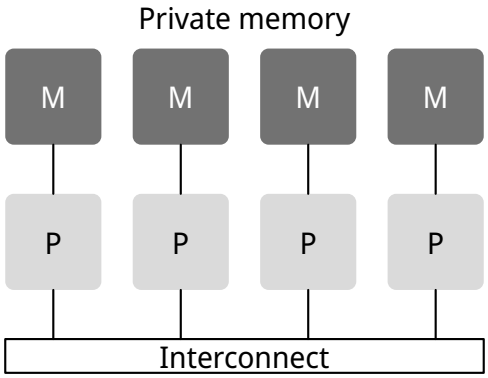
The extent to which inconsistencies can be tolerated is application-specific. However, if these can be tolerated, then the need for global synchronization can be reduced.

Parallel Computing

Multiprocessor



Multicomputer



Distributed high-performance computing began with parallel computing.

Distributed systems with shared memory (i. e. multi-computers with shared memory) as an alternative architecture did not fulfil the expectations and are therefore no longer relevant.

Amdahl's law - Limits to Scalability

- Solving **fixed problems** in the shortest possible time

  Example: Booting a computer. To what extent can more CPUs/cores shorten the time?

- It models the expected acceleration (*speedup*) of a partially parallelized/parallelizable program relative to the non-parallelized variant.

Definition

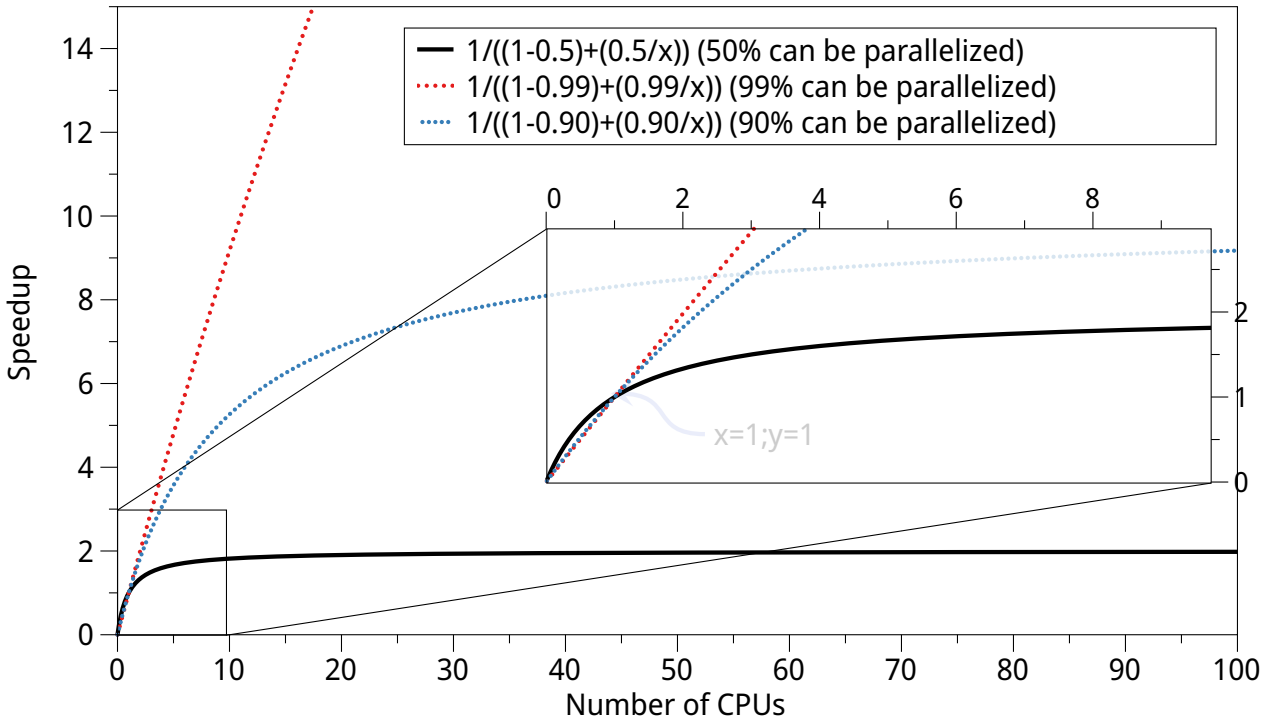
$$S(C) = \frac{1}{(1-P)+\frac{P}{C}}$$

Note

C = Number of CPUs

P = Degree of parallelisation in percent

S = Speedup



Gustafson's Law - Limits to Scalability

- Solving problems with (very) large, structurally repetitive data sets in **fixed time**; the serial part of the programme is assumed to be constant.

  Example

  Create the weather forecast for the day after tomorrow within the next 24 hours. To what extent can the precision of the forecast be improved by using more CPUs/computers?

- Beschleunigung (Speedup) eines parallelisierten Programms relativ zu der nicht-parallelisierten Variante:

  $S(C) = 1 + P(n) \cdot (C-1)$

Note

C: Number of CPUs

P: Degree of parallelisation as a function of the problem size n

S: Speedup

Example

Let the degree of parallelization for a relevant problem size n be $80\ \%$. This results in a speedup of $(1 + 0.8 \cdot 3) = 3.4$ for 4 CPUs, a speedup of 6.6 for 8 CPUs and a speedup of 13 for 16 CPUs.

Exercise

1.2. Compute Speedup

You are a pentester and you try to penetrate a system by attacking the passwords of the administrators. At the moment, you are using 2 graphics cards with 2048 compute units each. The serial part of the attack is 10 %. How high is the speedup you can expect, if you add two more comparable graphics cards with another 2048 compute units per GPU?

Background

The attacks are highly parallelizable and effectively depend on the number of CUs. The graphics cards are able to accelerate the attacks effectively.

2. Requirements on Distributed Systems

Dependability of Distributed Systems

Dependencies

A **component**[2] provides **services** to its **clients**. For that, the component may in turn require services from other components and therefore the component is dependent on another component (🇬🇧 *depend*).

Definition

A component C depends on C^* if the correctness of the behavior of C depends on the correctness of the behavior of C^*.

🇬🇧 *Dependability* ≙ 🇩🇪 *Verlässlichkeit*

[2] Components are processes or channels.

Requirements on the Reliability of Distributed Systems

Requirement	Description
Availability	The system is usable.
Reliability	Continuity of correct service provision.
Safety	Low probability of a catastrophic event.
Maintainability	How easily can a failed system be recovered?

Attention!

Security ≙ *Sicherheit*

Safety ≙ *Sicherheit*

Safety refers to the safety of people and property, while Security refers to the security of data and information.

Reliability vs. Availability in Distributed Systems

Reliability $R(t)$ of the component C

Conditional probability that C worked correctly during $[0, t)$ if C worked correctly at time $T = 0$.

Traditional Metrics

- Mean Time to Failure (MTTF):

  The average time to failure of a component.
- Mean Time to Repair (MTTR):

  The average time it takes to repair a component.
- Mean Time between Failures (MTBF):

  $\mathrm{MTTF} + \mathrm{MTTR} = \mathrm{MTBF}$.

Reliability: How likely is it that a system will work *correctly*?
Availability: How likely is it that a system will be available at a given time?

MTBF vs. MTTR

If the MTTF of a component is 100 hours and the MTTR is 10 hours, then the MTBF is $= \mathrm{MTTF} + \mathrm{MTTR} = 100 + 10 = 110$ hours.

MapReduce - Programming model and Middleware for Parallel Computing

- MapReduce is a programming model and a corresponding implementation (a framework originally developed by Google) for processing very large amounts of data (possibly TBytes).
- Programs implemented with the help of MapReduce are automatically parallelized and executed on a large cluster of commodity hardware.

  Responsibility of the runtime environment:

  Partitioning the input data and distributing it to the computers in the cluster.

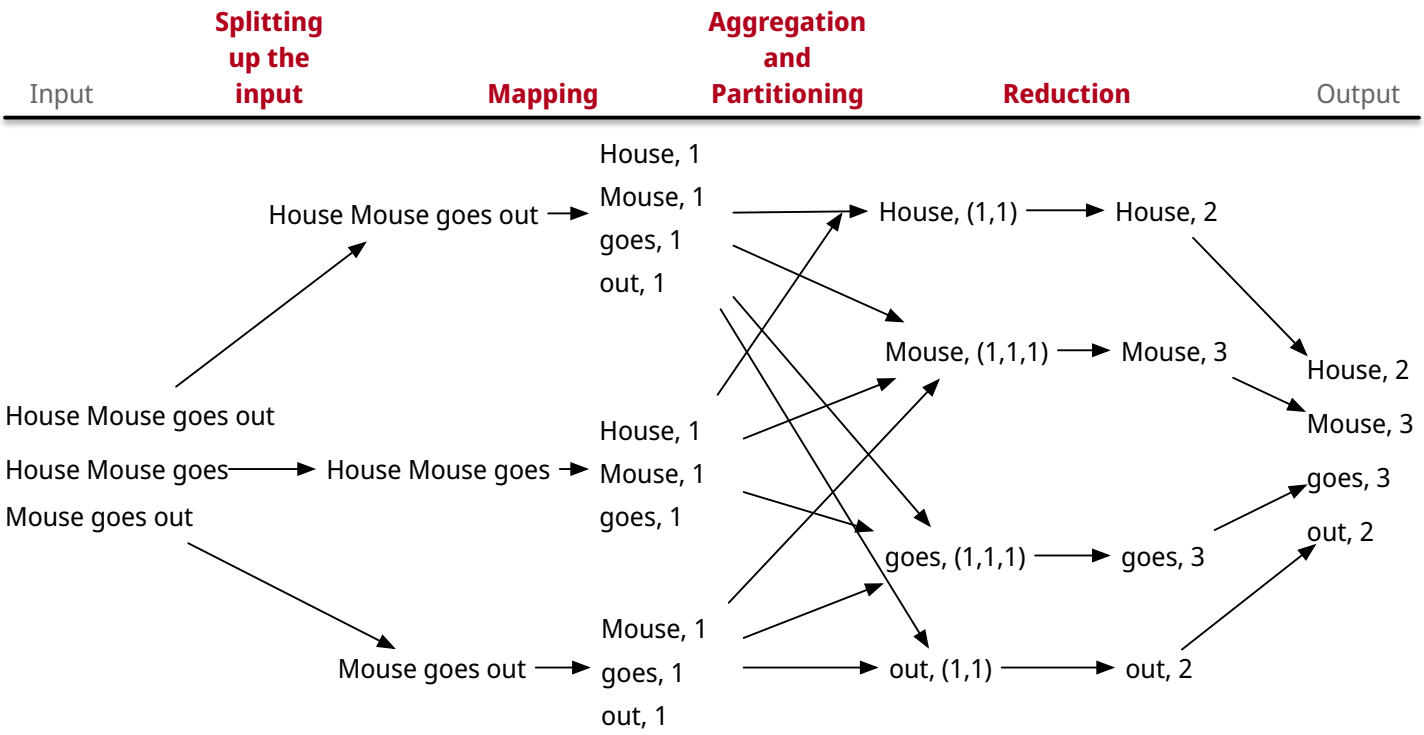
  Scheduling and execution of the `Map` and `Reduce` functions on the computers of the cluster.

  Error handling and communication between the computers.

Hint

Not all kinds of computations can be performed with the help of MapReduce.

MapReduce - Visualization of an Example



Here it is the calculation of the frequency of words in a very large data set.

Another canonical example is the calculation of an inverted index. I. e., the mapping of words to the documents/webpages in which they occur.

Exercise

2.1. Availability and Failure Probability

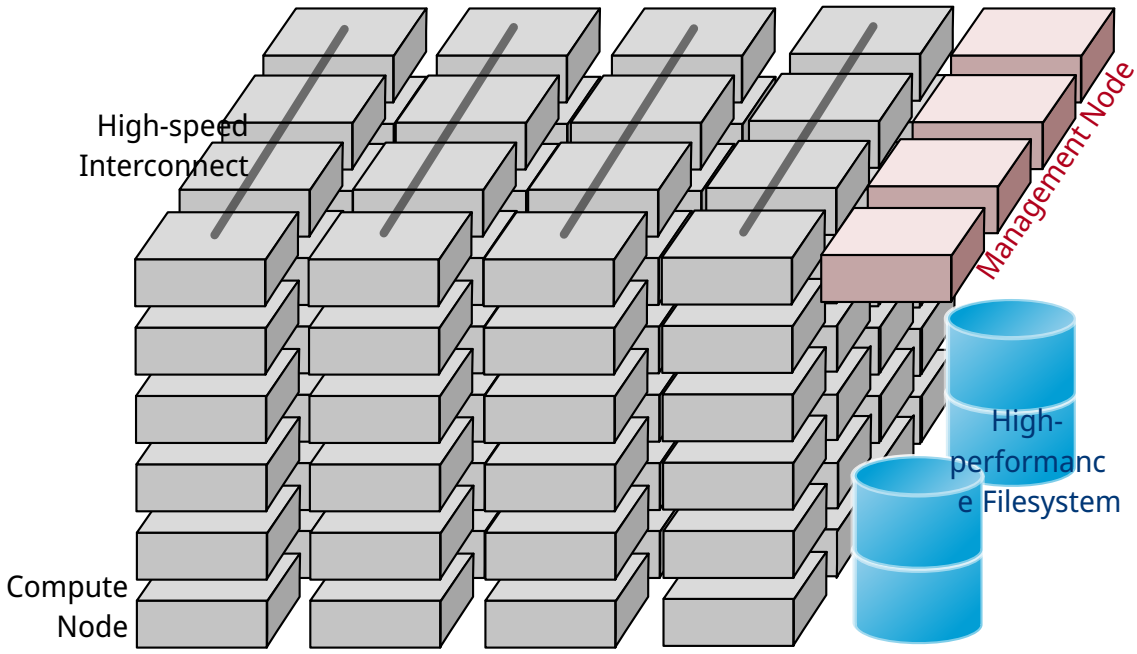
Consider a large distributed system consisting of 500 independent computers which fail independently of each other. On average, each computer is unavailable for twelve hours within two days.

a. Determine the intact probability of a single computer.
b. A data set is replicated on three computers for reasons of fault tolerance. What is its average availability when we try to access it?
c. On how many computers do you have to store this data set so that the average availability is 99.999%?
d. For how many minutes per year (with 365 days) is it *not possible to read the data set*, when we have an average availability of 99.999%?

3. Classification of Distributed Systems

Cluster Computing

A group of high-end systems connected via a LAN.



The individual computers/compute nodes are often identical (hardware and software) and are managed by a management node (*management node*).

Grid Computing

Continuation of cluster computing.

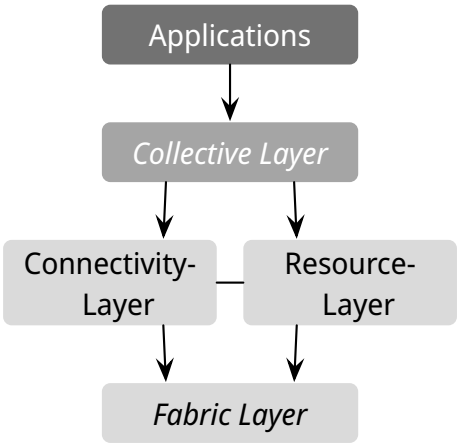
- Many heterogeneous nodes scattered over a wide area and across several organizations.
- The nodes are connected via the WAN.
- Collaboration takes place within the framework of a virtual organization.

(Volunteer) Grid Computing - Examples:

https://scienceunited.org

https://einsteinathome.org

Basic Architecture for Grid Computing



Fabric layer:
Provides interfaces to local resources (for querying status and capabilities, locking, etc.)

Connectivity layer:
Communication / transaction / authentication protocols, e.g. for transferring data between resources.

Resource layer:
Manages a single resource, e.g. creating processes or reading data.

Collective Layer: Manages access to multiple resources: discovery, scheduling and replication.

Applications: Contains actual grid applications in a single organisation.

🇩🇪 *Auffindung* ≙ 🇺🇸 *Discovery*

🇩🇪 *Einplanung* ≙ 🇺🇸 *Scheduling*

Peer-to-Peer-Systems

Vision:	"The network is the computer." There is a database that is always accessible worldwide.
Idea:	No dedicated clients and servers, each participant (peer) is both provider and customer.
	Self-organising, without a central infrastructure (coordinator, database, directory of participants).
	Each peer is autonomous and can be offline at any time, network addresses can change at will.
Main Application:	File-Sharing-Systems (in particular BitTorrent)

The peak of classic peer-to-peer systems was in the 2000s.

✓ Advantages of P2P systems are: cheap, fault-tolerant, dynamic, self-configuring, immensely high storage capacity, high data access speed.

! Problems of P2P systems are: start-up, poorly connected, low performance peers; *free riders*; copyright problems.

Cloud-Computing

Definition

Cloud computing refers to the provision of computing power, storage and applications as a service. It is the continuation of grid computing.

Variants

- Public Cloud (z. B. Amazon EC2, Google Apps, Microsoft Azure, ...)
- Private Cloud
- Hybrid Cloud
  (The private cloud is supplemented by a public cloud if required).
- Virtual Private Cloud

✓ Advantages of cloud computing: costs, up-to-dateness of data and services, no in-house infrastructure required, support for mobile participants

! Problems of cloud computing: security and trust, loss of in-house expertise, handling of classified data.

One way out could be homomorphic encryption, which makes it possible to perform calculations on encrypted data.

Serverless Computing

Serverless Computing enables developers to create applications faster, as they no longer have to worry about managing the Infrastructure.

✓ The cloud service provider automatically provides, scales and manages the infrastructure required to run the code.

! Vendor-Lock-In

! Cold-boot latency

Time until the first code is executed can be longer, as the serverless functions are only instantiated when required.

! Debugging and Monitoring

Traditional tools and methods can no longer be used.

! Cost-transparency/-management

The costs of serverless computing are difficult to predict and control.

4. Challenges in Developing Distributed Systems

Application Integration

Typical enterprise applications in companies are networked applications and establishing interoperability between these applications is a major challenge.

Basic Approach

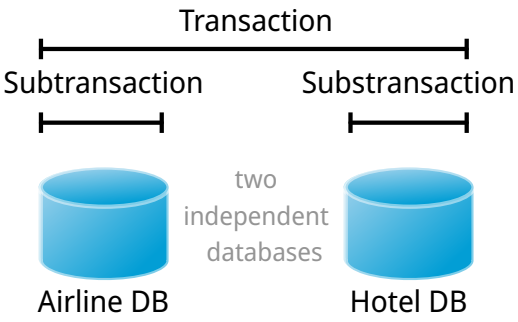
Clients combine requests for (different) applications, send them, collect the responses and present a coherent result to the user.

Modern Approach

Direct communication between applications leads to the integration of enterprise applications (Enterprise Application Integration (EAI)).

A networked application is an application that runs on a server and makes its services available to remote clients.

Transactions at Business Process Level



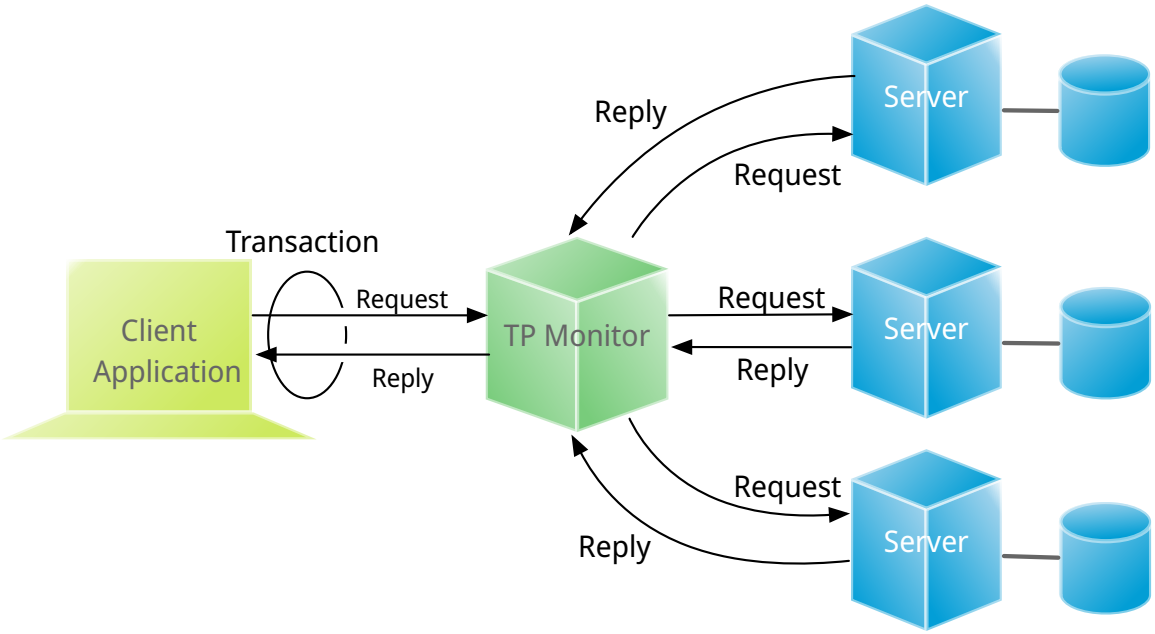
„All or nothing.“

Primitive	Description
BEGIN OF TRANSACTION	Indicates the start of a transaction.
END OF TRANSACTION	Completes the transaction with an attempt to COMMIT.
ROLLBACK OF TRANSACTION	terminate the transaction and restore the old status.
READ	Reading data from (e.g.) a file or a table.
WRITE	Writing data (e.g.) to a file or a table.

ACID-Properties:

Atomic: happens inseparably (seemingly)
Consistent: no violation of system invariants
Isolated: no mutual influence
Durable: after a commit, the changes are permanent

Transaction Processing Monitor (TPM)



Observation

The data required for a transaction is often distributed across several servers.

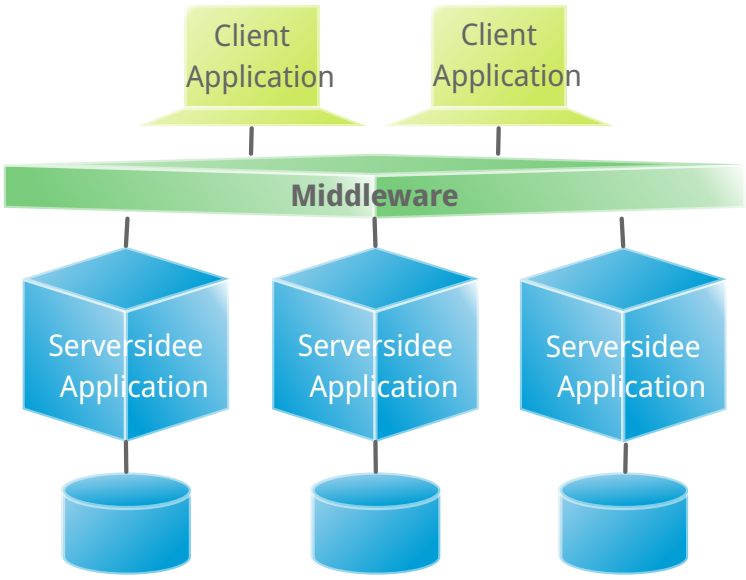
A TPM is responsible for coordinating the execution of a transaction.

When you implement microservices, the use of TPMs and 2PC for the purpose of coordinating business processes is often not the first choice.

Nevertheless, distributed transactions are an important part of distributed systems and Google, for example, has developed Spanner, a solution that enables transactions on a global scale (*Global Consistency*). (https://cloud.google.com/spanner?hl=en and https://www.youtube.com/watch?v=iKQhPwbzzxU).

Middleware and *Enterprise Application Integration (EAI)*

Middleware enables communication between applications.



Remote Procedure Call (RPC):
Requests are sent via a local procedure call, packaged as a message, processed, answered by a message and the result is then the return value of the procedure call.

Message Oriented Middleware (MOM):
Messages are sent (i. e. published) to a logical contact point (i. e. message broker) and forwarded to applications that subscribe to these messages.

How can application integration be achieved?

File transfer:

Technically simple, but not flexible:

- Determine the file format and layout
- Regulate file management
- Passing on updates and update notifications

Shared database:

Way more flexible, but still requires a common data schema in addition to the risk of a bottleneck.

Remote Procedure Call (RPC):

Effective when execution of a series of actions is required.

Messaging:

Enables temporal and spatial decoupling compared to RPCs.

5. Modern Distributed Systems

Distributed Pervasive/Ubiquitous Systems

🇺🇸 *Distributed Pervasive/Ubiquitous Systems* ≙ 🇩🇪 *verteilte, allgegenwärtige/alles durchdringende Systeme*

Modern distributed systems are characterised by the fact that the nodes are small, mobile and often embedded in a larger system. The system embeds itself naturally in the user's environment. Networking is wireless.

Three (overlapping) subtypes

Ubiquitous computing:
ubiquitous and always present; i. e. there is constant interaction between the system and the user.

Mobile computing: *ubiquitous*; the focus is on the fact that devices are inherently mobile.

Sensor/Actuator Networks:
ubiquitous; focus is on actual (collaborative) sensing and actuation.

Ubiquitous Systems - Key Elements

Distribution: The devices are networked, distributed and accessible without barriers.

Interaction: The interaction between users and devices is highly unobtrusive.

Context awareness: the system knows the user's context in order to optimize the interaction.

Autonomy: The devices work autonomously, without human intervention, and manage themselves independently to a high degree.

Intelligence: The system as a whole can handle a wide range of dynamic actions and interactions.

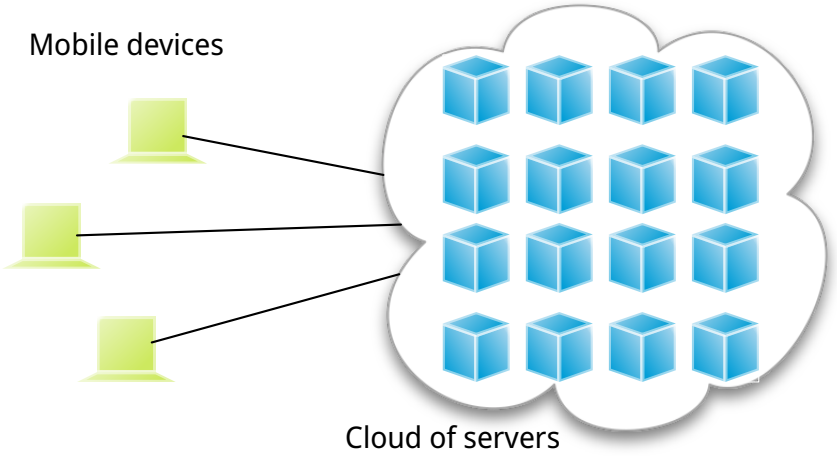
Mobile Computing - Characterizing features

- A variety of different mobile devices (smartphones, tablets, GPS devices, remote controls, active ID cards).
- Mobile means that the location of a device can change over time. This can, e. g., have an impact on local services or accessibility.
- Maintaining stable communication can lead to serious problems.

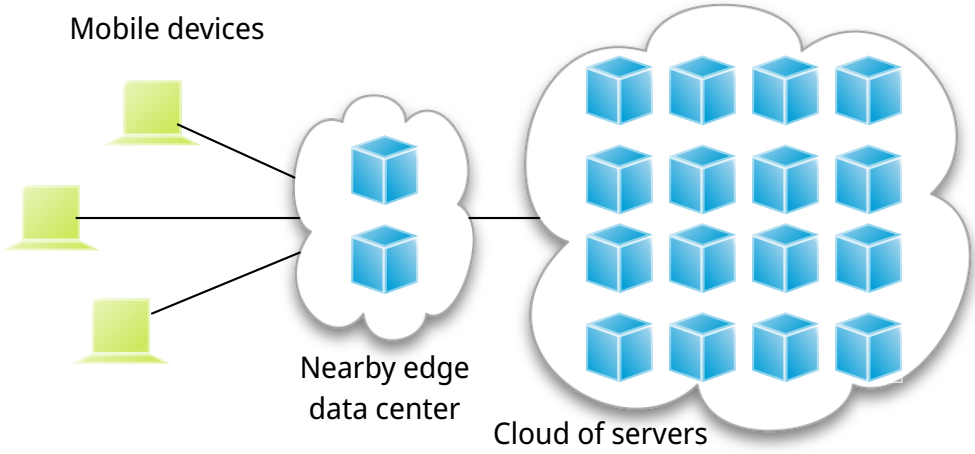
Observation

The current status is that mobile devices establish connections to stationary servers, making them in principle *clients* of cloud-based services.

Mobile Cloud Computing



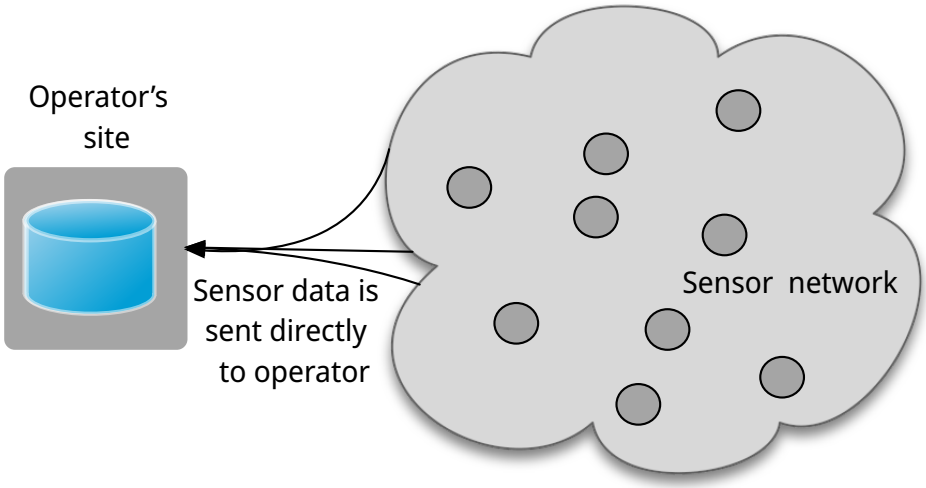
Mobile Edge Computing



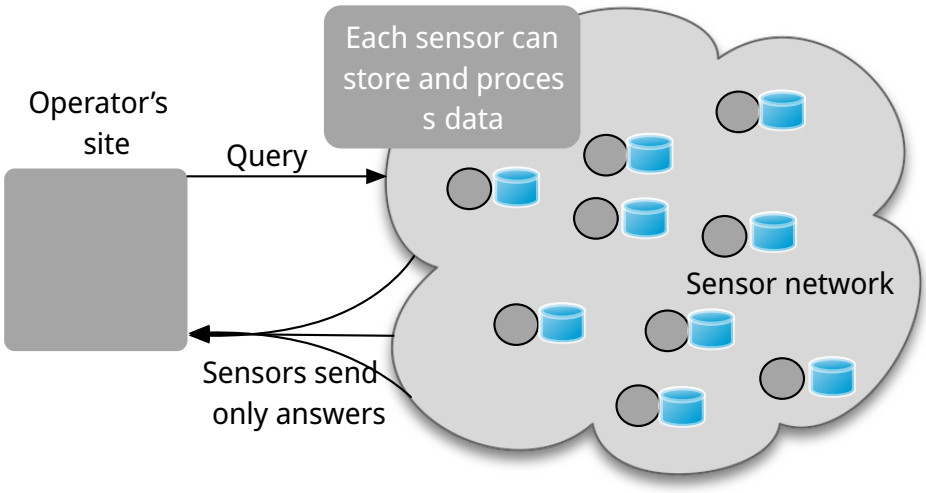
Sensor Networks

The nodes to which sensors are attached:

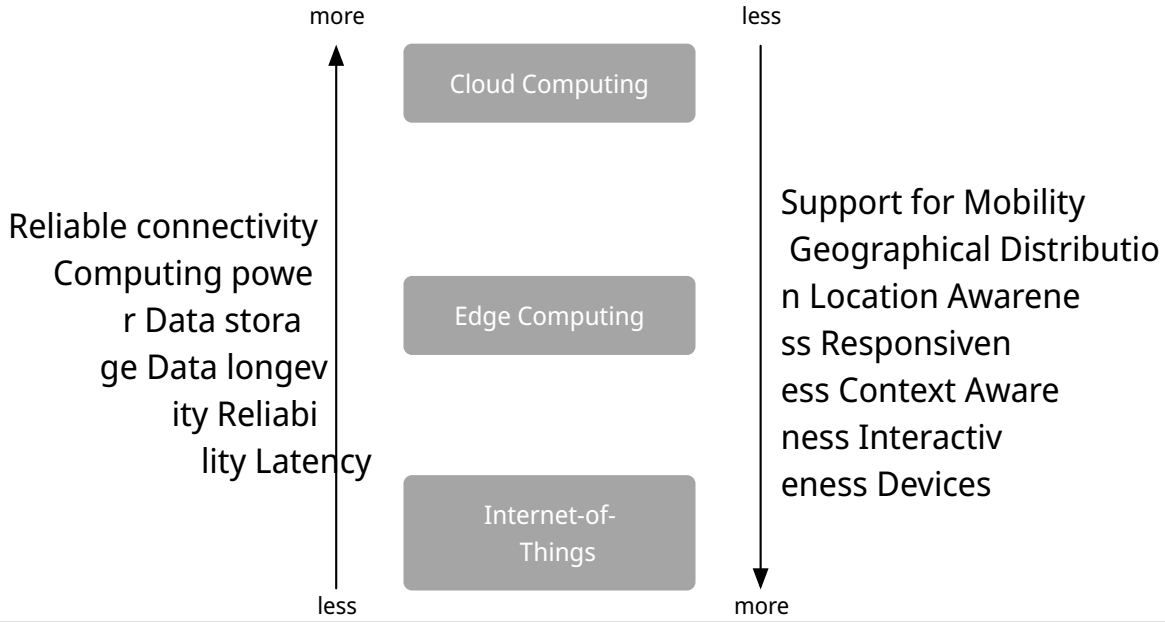
- "many"
- simple (low memory / computing / communication capacity)
- often battery-operated (or even battery-free)



Sensor Networks as Distributed Databases



The *Cloud-Edge Continuum*



Pitfalls in Developing Distributed Systems

Observation

Many distributed systems are unnecessarily complex due to incorrect assumptions and architectural and design errors that have to be rectified later.

Incorrect (and often hidden) assumptions

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- The latency is zero
- The bandwidth is infinite
- The transport costs are zero
- There is only one administrator