Distributed Systems - Lecture Summary

📗 Infos: The notes in the book are not covered. The beginning of each chapter here has information on what parts of the book were relevant in WS22. You should definitely look at all the algorithms from the lecture, as I did not cover them either. You'll have to learn to deal with my stupid comments throughout this whole thing.

Introduction & Architectures

1 Introduction

1.1. What is a distributed system?

- A collection of computing elements (nodes) that can behave independently
- Users believe they are dealing with a single system

The nodes need to collaborate: That is the whole point of ✨distributed systems✨

1. Collection of autonomous computing elements

Nodes can have all types of different kinds (from small devices to high performance computers). They have common goals which they reach by exchanging messages. Each node has its own notion of time → not something like a global clock. The term "collection of nodes" implies that managing the membership and organization of that collection will be necessary (nodes have group chats, and smaller group chats with only the cool people). There are open groups and closed groups (open = everyone can communicate, closed = need a mechanism for joining and leaving groups). Group management can lead to a bottleneck. Theoretically, every node has to check if it is communicating with another group member and not an imposter. Members can communicate with nonmembers → confidentiality issue.

Distributed Systems are often organized as ***overlay networks.*** In this case, a node usually is a software process that has a list of processes it can directly send messages to (TCP/IP or UDP).

2 Types of overlay networks:

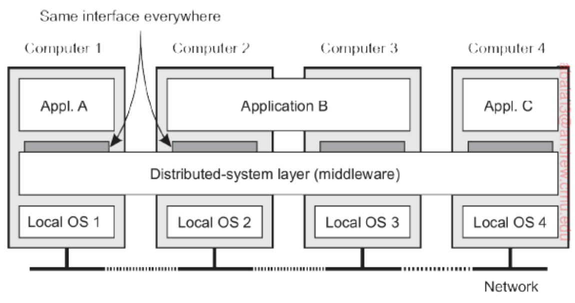
- Structured overlay (well defined set of neighbours, like a tree)
- Unstructured overlay (each node has references to randomly selected nodes)

A network should always be connected (There is always a way from A to B)

2. Single coherent system

In a single coherent system, the collection of nodes operates the same no matter where when and how the interaction between user and system occurs. The user should not be able to tell on which computer a process is executing, and where data is stored, or that the system might replicate data, shouldn't matter → ***distribution transparency.***
Any type of the system may fail at any time. (The failure of a computer you don't even know exists might fuck up your own computer)

Middleware and distributed systems



Distributed systems are often organized to have a separate layer of software → middleware.

Aims to hide the differences in hardware and operating systems from each application.

Middleware offers facilities for interapplication communication, security services, accounting services, masking of and recovery from failures

Difference with OS: middleware services are offered in a networked environment.

Typical middleware services:

- Communication: Remote procedure call (allows a function on a remote computer to be invoked locally)
- Transactions: if multiple services are involved, middleware makes it atomic (either all are invoked or none is)
- Service composition: Taking existing programs and just glueing them together is common practice. Web based middleware can help standardize everything
- Reliability: Ensure that in a group of processes either all or none receive a message

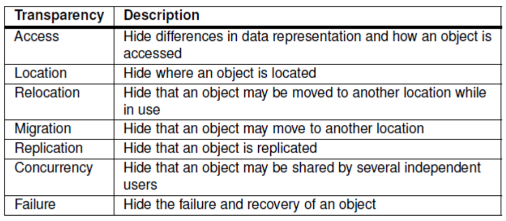
1.2. Design goals

A distributed system should make resources easily accessible, it should hide the fact that resources are distributed across a network, it should be open, and it should be scalable.

1. Support resource sharing

Resources can be virtually anything, typical examples are peripherals, storage facilities, data, files, services, and networks. Resource sharing is often cheaper (no shit sherlock) and makes it easier to collaborate and exchange information (✨the internet✨).

2. Making distribution transparent (as in invisible, not as in see-through)



Transparency	Description
Access	Hide differences in data representation and how an object is accessed
Location	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 move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Types of distribution transparency:

- access transparency: reach agreement on how data is to be represented by different machines and OSs (each may have their own file naming conventions, file operations or low-level interprocess communication)
- location transparency: users cannot tell where an object is located in the system. This can often be achieved by assigning names in which the location is not encoded (URL for example)
- Relocation transparency: important for cloud computing

- migration transparency: supports the mobility of processes and resources initiated by users without affecting ongoing communication and operations (talking on the phone while walking around)
- replication transparency: a system that supports replication transparency should also support location transparency as well. otherwise it would be impossible to refer to replicas at different locations
- concurrency transparency: consistency achieved through locking mechanisms, or use of transactions (difficult to implement, scalability issues)
- failure transparency: user or application do not notice some piece of the system fails, the system automatically recovers from the failure. Under realistic assumptions, this is often impossible. Problem is distinguishing between a dead process and a slow process

Degree of distribution transparency

Trade-off between a high degree of transparency and the performance of a system (masking a server failure before trying another server slows down the system). With smartphones, location awareness is important, it might be preferable to expose distribution rather than hide it.
Arguments against distribution transparency: full distribution transparency is impossible, so why even pretend? Users would better understand the behaviour of the distributed system.

3. Being open

Open distributed system offers components that can easily be used by, or integrated into other systems and it will itself probably consist of components that come from elsewhere.

Interoperability, composability, extensibility
Components should adhere to standard rules concerning syntax & semantics. General approach → define services through interfaces using an ***Interface Definition Language***. That makes defining the syntax easy, for semantics, a natural language approach is used (think a JavaDoc'd interface).
Interoperability: the extent by which two implementations of systems or components from different manufacturers can coexist and work together by relying on each other's services as specified by a common standard.
Portability: to what extent can an application developed for a distributed system ***A*** be executed without modification on a different distributed system ***B***, given that ***A*** and ***B*** implement the same interfaces.
It should be easy to configure the system out of different components (✨Mix-n-Match✨) and add or exchange components. An open distributed system should be ***extensible***. It should thus be easy to add parts that run on a different OS or replace an entire file system.

Separating policy from mechanism
The system should be organized of small, easily replaceable components (✨modularisierung✨). The stricter the separation between policy and mechanism. the more we need to ensure that an appropriate collection of mechanisms is offered.

4. Being scalable

Scalability can be measured along three different dimensions:

1. Size scalability
   More users and resources without noticeable loss of performance. Many services are implemented by a single server running on a specific machine in the distributed system, or a group of collaborating servers close together. The server(s) become a bottleneck with an increasing number of requests. Three root causes for a bottleneck: CPU limitation, storage capacity & I/O transfer rate, and the network between the user and the centralized service

2. Geographical scalability
   Users and resources may be very far apart, but communication delays are hardly noticed. Many systems designed for Local Area Networks are based on ***synchronous communication*** (aka blocking). Communication in WANs is less reliable than in LANs, and then bandwidth is limited. Thus, solutions for LANs can not always be ported to a wide area system (e.g. video streaming). Wide area systems also typically have limited facilities for multipoint communication, whereas LANs often support efficient broadcasting. In wide area systems, the need for naming- and directory services arises and now these need to be scalable as well and there are often no obvious solutions for this (✨ein rattenschwanz✨).

3. Administrative scalability
   Can easily be managed even if it spans many independent administrative organizations. This remains a difficult and often open question. Problem → conflicting policies regarding resource usage (&payment), management and security. Example: researchers sharing equipment in a ***computational grid***. If a distributed system expands to another domain, both domains need to protect themselves from malicious attacks from the other domain. Counterexample (no administrative scalability issues): file sharing peer-to-peer networks, skype, spotify. Their common part is that end users collaborate to keep the system running.

Scaling techniques
Increasing server capacity (more cpu, more memory) is referred to as ***scaling up. Scaling out*** refers to expanding the distributed systems (more machines). 3 Techniques:

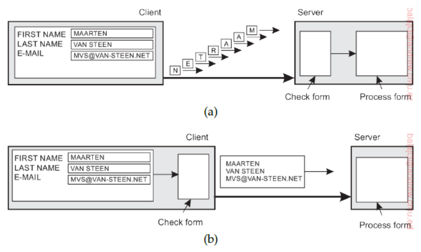


Figure 1.4: The difference between letting (a) a server or (b) a client check forms as they are being filled.

- Hiding communication latencies: applicable in the case of geographical scalability. Basic idea: avoid waiting for responses (aka asynchronous communication). This can often be used in batch-processing systems and parallel applications. Alternatively, start a new thread. Interactive applications often have nothing else to do while waiting → reduce overall communication (e.g. by moving some computation from the server to the client; ✨diy✨)
- Partitioning and distribution: take a component, split it into smaller parts, spread those parts across the system (e.g. DNS, the WWW)
- Replication: not only increases availability, but also helps with load-balancing. In geographically widely dispersed systems, replication can help with latencies. Caching is a special form of replication (distinction is often hard or artificial). In the case of caching, the replica is in the proximity of the client and it is a decision made by the client of a resource. Drawback to replication and caching: modifying a copy leads to consistency problems

Size scalability is the least problematic from a technical point of view. Combination of all techniques often leads to an acceptable solution.

Pitfalls

False assumptions everyone makes when developing a distributed application for the first time:

- the network is reliable
- the network is secure

- the network is homogeneous
- the topology does not change
- latency is zero
- bandwidth is infinite
- transport cost is zero
- there is one administrator

1.3 Types of Distributed Systems

High performance distributed computing

Cluster computing: underlying hardware consists of similar workstations connected by high speed LANs, each node runs on the same OS.

▼ Linux Beowulf

Each cluster consists of a collection of nodes controlled by a single master node (✨one node to rule them all✨). Master handles allocation of nodes to a particular parallel program, manages submitted jobs and provides an interface for the system users (master runs the middleware needed for execution of programs and management of the cluster, compute nodes have standard OS extended with middleware functions for communication, storage, fault tolerance...)

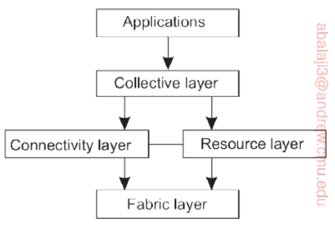
▼ MOSIX

MOSIX attempts to provide a single-system image of a cluster (ultimate distribution transparency)

▼ other attempts

middleware is functionally partitioned across different nodes (advantage. optimal performance for computationally intense applications, storage can likely be handled optimally...

Grid computing: distributed systems that are often constructed as a federation of computer systems, where each may fall under different administrative domain and may be different regarding hardware, software and deployed network technology. This is realized in the form of a ***virtual organization***. This means processes belonging to the same organization have access to all resources provided to that organization (e.g. compute servers, storage facilities, databases). Focus often lies on architectural issues.



- Fabric layer: provides interfaces to local resources at a specific site
- Connectivity layer: communication protocols for supporting grid transaction that require multiple resources, contains security protocols to authenticate users and resources
- Resource layer: managing a single resource. Uses the functions provided by the connectivity layer and calls the interfaces made available by the fabric layer. Is seen to be responsible for access control.

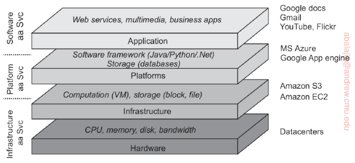
- Collective layer: handles access to multiple resources, typically consists of services for resource discovery, allocation, scheduling tasks onto multiple resources, replication… collective layer may consist of many different protocols (as opposed to the other layers) reflecting the wide range services it may offer to a virtual organization

- Application layer: consists of the applications

Collective, connectivity and resource layer form the heart of a “grid middleware layer”.

Shift towards a service-oriented architecture, in which sites offer access to the various layers through a collection of web services → lead to the definition of an alternative architecture (***Open Grid Services Architecture***)

Cloud computing: take it one step further than grid computing and outsource the entire infrastructure. Cloud computing was developed from utility computing, where one would upload tasks to a data center. It is based on an easily usable pool of virtualized resources. Resources can be configured dynamically (✨yayy scalability✨).



- hardware: literally hardware
- infrastructure: backbone of most cloud computing platforms. Deploys virtualization techniques.
- platform: platform layer provides to a cloud computing customer what an OS provides to application developers (aka the means to develop and deploy applications that need to run in a cloud. Provides higher level abstractions for storage and such
- application: actual applications run in this layer, offered to users for more customization (think MS Office like)

Three different types of services:

- Infrastructure as a Service (IaaS): covering hardware and infrastructure layer
- Platform as a Service (PaaS): covering platform layer
- Software as a Service (SaaS): applications are covered

Obstacles include: security concerns, provider lock—in, privacy issues, dependency on availability of services

Distributed information systems

A lot of middleware resulted from working with an infrastructure where it was easier to integrate applications into an enterprise-wide information system. This could take place at several levels. Lowest level: client wraps a number of requests (possibly even for different servers) and it is executed as a transaction. However, integration should also take place by letting applications communicate directly with each other → huge industry created (Enterprise Application Integration)

Distributed transaction processing

Referring to the “execute as a transaction” part. Types of transaction primitives depends on the context. Might be read and write, ordinary statements, procedure calls. Remote procedure calls are often encapsulated in a transition (transitional RPC).

Transactions adhere to the ACID properties: Atomic, Consistent (transaction does not violate system invariants), Isolated (concurrent transactions do not intefere with each other), Durable (Changes from a transaction are permanent).

Of course, there is also the abomination called a ✨nested transaction✨. Problem: if one of multiple parallel subtransaction commits (making the results visible to the parents) but the parent then aborts, the subtransaction must be undone, thus affecting the performance. In the early days the component that handled nested transactions formed the core for integrating applications at the server or database level and was called transaction processing monitor. It coordinated the commitment of subtransactions following the so-called distributed commit protocol.

Enterprise application integration
Interapplication communication being needed led to many communication models (like RPCs: send a request by calling a local procedure). Remote Method Invocations are basically RPCs with objects instead of functions. Disadvantages: both communication partners need to be up and running and need to know how to refer to each other. This has lead to message-oriented middleware (MOM). Applications send messages to logical contact points and they can also indicate their interest for a specific type of message (publish-subscribe system).

Pervasive systems

So far: distributed systems are characterized by their stability. Nodes are fixed and have permanent high quality connections to a network. This stability is realized via distribution transparency (hide failure, hide location of a node).

Things have changed, creating the notion of ***pervasive systems***. They are intended to naturally blend into our environment. With pervasive systems, the separation between users and system components is more blurred. There is often no single interface, instead it often has many sensors picking up the user's behaviour and feedback. Many of these devices are small and mobile with a wireless connection (not restrictive characteristics). These devices need special solutions to make pervasive systems transparent and inconspicuous.

Ubiquitous computing systems
The system is pervasive and continuously present (interaction is taking place without the user even realizing).
Four requirements for a ubiquitous computing system

- Distribution: Devices are networked, distributed, and accessible in a transparent manner
- Interaction: Interaction between users and devices is highly inconspicuous
- Context awareness: The system is aware of a user's context in order to optimize interaction
- Autonomy: Devices operate without human intervention (self-managed)
- Intelligence: The system as a whole can handle a wide range of dynamic actions and interactions

Mobile computing systems
Mobility is an important component of pervasive systems. The devices that are part of a distributed mobile system may vary widely (smartphone, tablet, GPS enabled devices). Mobile computing assumes the location of the devices changes over time, and some services may only be locally available, thus making dynamically discovering services and letting them announce their presence necessary. Knowing the actual coordinates of a device or detecting its network position may also be important.
Disruption tolerant networks: networks in which connectivity between two nodes can not be guaranteed. Trick for getting messages across: flooding based techniques and intermediate nodes storing received messages (like a courier).

Sensor networks
Sensor nodes often collaborate to efficiently process the sensed data. Nodes often act as actuators (e.g

automatic sprinklers). Problems: limited resources, restricted communication capabilities, constrained power consumption (battery powered little thingies), thus it must be really efficient. Two extremes to organizing a sensor network as a distributed database: all sensors send data to the operator, or the operator asks each sensor for the data. Both suck. Needed: facilities for in-network data processing. This can be done by aggregating the sensor results in a tree like structure while propagating to the operator (How do we dynamically set up an efficient tree in a sensor network? How does aggregation of results take place? What happens if network links fail?).

2 Architectures

- Software architecture: Concerned about the logical organization of the software; how components interact, in what ways can they be structured, how can they be made independent...
- System architecture: where are the components that make up a distributed system placed across the various machines

2.1 Architectural styles

Components can be replaced, even while the system is running, because it is often not possible to shut down a system for maintenance.

Connector: a mechanism that mediates communication, coordination, or cooperation among components. It allows for the flow of control and data between components.

Using components and connectors leads to various configurations (architectural styles). The most important are: Layered architectures, object based architectures, resource centered architectures, event based architectures.

Layered architectures

Components are organized in a layered fashion (✨no shit sherlock✨), where the layers can make a ***downcall*** to a lower layer, which in turn responds. ***Upcalls*** are only made in exceptional cases (e.g. the OS signals the occurrence of an event and calls a user-defined operation for which an application had previously passed a handle).

Layered communication protocols
Very common layered architecture: communication-protocol stacks. Each layer implements communication services and an interface (which should completely hide the actual implementation) specifying the functions that can be called. (There’s a part about TCP here in the book, but nothing special)

Application layering
Distinction between 3 logical levels: application-interface level, processing level, data level. Applications usually consist of roughly 3 different pieces: a part that handles interaction with a user/external application, a part that operates on a database/file system, and a middle part with the core functionality. (Examples on page 60).

Data level is responsible for persisting data and keeping it consistent across different applications.

Object-based and service-oriented architectures

Each object corresponds to a component and they are connected through a procedure call mechanism (can also take place over a network). We may place an interface on one machine and the corresponding object on another, which is referred to as a ***distributed object***. When a client binds to a distributed object, a proxy is loaded into the client's address space. The proxy's task is turn messages into method invocations and vice versa. Fun fact: a counterintuitive feature of most distributed objects is that their state is not distributed, only the interfaces implemented by the object are made available on other machines (***remote objects***).

In service oriented architecture, a distributed application/system is constructed as a composition of many different services, which may not belong to the same administrative organization. The problem of developing a distributed system is partly one of service composition and making sure those services operate in harmony. It is crucial that each service offers a well-defined interface.

Resource-based architectures

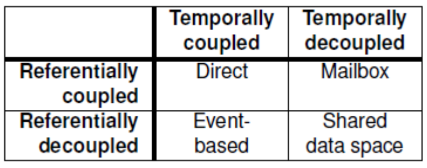
The distributed system is viewed as a huge collection of resources which are individually managed by components. Resources may be added or removed by applications and can be retrieved or modified. (REST).

REST characteristics:

- Resources are identified through a single naming scheme
- All services offer the same interface (put, get, delete, post)
- Messages sent to or from a service are fully self-described
- After executing an operation at a service, that component forgets everything about the caller (stateless execution)

Publish-subscribe architectures

Dependencies between processes must be as loose as possible. Separation between processing and coordination. The idea is to view a system as a collection of autonomously operating processes.



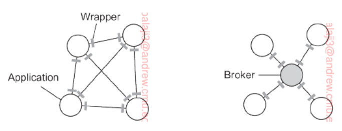
	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

- direct: both must know each other, both must be up and running (e.g. phone call)
- mailbox: do not need to run at the same time
- event-based: do not know each other, but both receive a notification they have subscribed to.
- shared data space: process subscribes to certain tuples via a search pattern and is notified when matching tuples are inserted into the data space

Important aspect of publish-subscribe systems: Communication takes place by describing the events a subscriber is interested in, thus naming is really important. A description of an event usually consists of (attribute,value) pairs in ***topic-based publish-subscribe systems*** and (attribute,range) pairs in ***content-based publish-subscribe systems***. When there's a match, the middleware either forwards the notification along with the data, or it only forwards the notification and the subscribers can execute a read operation.

2.2 Middleware organization

Wrappers



The interfaces offered by the legacy component are probably not suitable for all applications. The wrapper (adapter) is a special component that makes the interfaces compatible (examples on page 72). Wrappers are crucial when it comes to extending systems with existing components. This leads to a quickly rising number of wrappers. Solution: implementing a broker, which handles all accesses between different applications.

Interceptors

An interceptor breaks the usual flow of control and allows other code (application specific) to be executed.

Modifiable middleware

Wrappers and interceptors offer means to adapt the middleware. The need for adaptation arises because the environment in which distributed applications are executed changes continuously. Middleware may not only need to be adaptive, we should be able to purposefully modify it without bringing it down. Popular approach: dynamically constructing middleware from components.

2.3 System architecture

Centralized organizations

Simple client server architecture (page 76 if you dont know how that works)
Connectionless protocols are more efficient, but transmission failures are a problem. An operation is ***idempotent*** if it can be done multiple times without any harm.
Connection-oriented protocols are reliable (e.g. TCP/IP). Con: setting up and tearing down a connection is costly, especially if the messages are only small.

Multilayered architectures
Distributing a client-server application over multiple machines with two types of machines:

- A client machine containing only programs implementing part of the user-interface level
- A server machine containing the rest (programs implementing the processing and data level)

The client is basically just a dumb terminal.

Distinction beetween client machines and server machines → two-tiered architecture. Sometimes servers need to act as clients → three-tiered architecture.

Decentralized organizations: peer-to-peer systems

Organizing a client-server application as a multitiered architecture into interface, processing and data level, is often referred to as ***vertical distribution***. Logically different components are placed on different machines. In modern architectures, the distribution of clients and servers is often more imporrant, which is referred to as ***horizontal distribution***.

In a p2p system, much of the interaction is symmetric, meaning that each process will act as client and server at the same time (”servant”). Given this symmetric behaviour, the idea is organizing the processes in an overlay network.

structured and unstructured overlay networks are described in detail on page 82, Hypercube example

Unstructured overlay:

- flooding: a node passes a request for data to all its neighbours. The neighbour either forwards the request or sends the requested data back. Can be very expensive
- Random walks: ask a random neighbour that asks a random neighbour...

Both of these can be stopped with a TTL value (time-to-live).

Between those two methods lie ***policy-based search methods***. Based on previous answers, a node picks its favourite neighbours and these are flooded.

Hierarchically organized peer-to-peer networks
Abandoning the symmetric nature of p2p sometimes makes sense. In a collaborative ***content delivery network*** (CDN), nodes may offer storage for hosting copies of Web documents allowing Web clients to access pages nearby quickly. A broker is employed for collecting data on resource usage and availability for a number of nodes. Nodes that maintain an index or act as broker are referred to as ***super peers***. Regular peers are called weak peers. If a super peer leaves the network, a new super peer must be chosen (leader-election problem, example page 88).

Hybrid architectures

Specific cases of distributed systems in which client-server solutions are combined with decentralized architectures.

- Edge-server systems: servers are placed at the edge of a network
- Collaborative distributed systems (example BitTorrent filesharing system page 91)

2.4 Example architectures

The network file system

Every file system is different → NFS makes it heterogeneous.
The client is unaware of the actual file location and is offered an interface with various file operations but the server is responsible for implementing the operations. This model is referred to as ***remote access model***.
Client accesses a file locally after downloading it from the server, after usage it is uploaded again → ***upload/download model***

The web

- simple web based systems: communication between browser and webserver is standardized (HTTP)
- multitiered architectures: web started out as simple client-server system

Processes & Communication

📗 CH 3
CH 4

- Section 4.1: Completely discussed.
- Section 4.2: Mostly covered, but not all topics are discussed in detail. In particular Notes 4.2-4.6 are not regarded, but taking a look would help your understanding. Multicast RPC and DCE RPC are also not discussed.
- Section 4.3: Partially covered. Notes 4.7-4.12 are not discussed. While simple transient messaging with sockets is covered, advanced transient messaging is not. Message-oriented persistent communication is part of the lecture, but IBM's WebSphere and AMQP are not.
- Section 4.4: All subsections are covered, but not in much detail. Again, the notes are not discussed.

3 Processes

3.1 Threads

Introduction to threads

To execute a program, an OS creates a number of virtual processors, each one for running a different program. The OS keeps a process table storing CPU register values, memory maps, open files… these entries form a ***process context***. A process is defined as a program in execution. Independent processes can not maliciously affect each other (concurrency transparency). Threads do not attempt to achieve concurrency transparency. Information that is not strictly necessary to manage multiple threads is generally ignored.

📗 more info on threads in the book

Many-to-one threading model: Multiple threads mapped to a single schedulable entity

One-to-one threading model: every thread is a schedulable entity. Con: every thread operation has to be carried out by the kernel.

Threads in distributed systems

Multithreaded clients
To achieve high distribution transparency, long interprocess message propagation times need to be concealed, usually something else is done during the waiting time (think async methods in JS)

Multithreaded servers
Main use of multithreading in distributed systems is found at the server side. The dispatcher reads incoming requests, an idle worker thread handles it.

3.2 Virtualization

The separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as ***resource virtualization***.

Principle of virtualization

Virtualization basically deals with extending or replacing an existing interface in order to mimic the behaviour of another system.

Virtualization and distributed systems
Historical context: virtualization important for allowing legacy software to run on expensive mainframe hardware.
The diversity of platforms and machines can be reduced by letting each application run on its own virtual machine. This provides a high degree of portability and flexibility.

Types of virtualization
Computer systems generally offer four different types of interfaces at three different levels:

- Interface between the hardware and software, called the ***instruction set architecture***, forming the set of machine instructions. It is divided into two subsets: privileged instructions (only OS can execute) and general instruction (everyone can execute)
- Interface consisting of ***system calls*** as offered by an operating system
- Interface consisting of library calls, generally forming what is known as an application programming interface (API).

The essence of virtualization is to mimic the behaviour of these interfaces.
Virtualization can take place two different ways:

- Build a runtime system that provides an abstract instruction set that is to be used for executing applications → ***Process virtual machine***
- Provide a system that is implemented as a layer shielding the original hardware but offering the complete instruction set of that same hardware as an interface → ***native virtual machine monitor.*** Access to various resources (storage, networks) has to be provided, a hosted virtual machine monitor will make use of existing facilities of the host operating system for that purpose. (cue the 5 page long note page 120)

Application of virtual machines to distributed systems

Most important application of virtualization is cloud computing. Virtualization plays a huge role in Infrastucture as a Service. Instead of renting out a physical machine, it's just a virtual machine. Insert 20 page long example of Amazon once again (page 122).

3.3 Clients

Networked user interfaces

Two ways to provide user-remoteserver interaction:

- for each remote service, the client machine will have a separate counterpart that can contact the service over the network (smartphone calendar needing to sync with a remote calendar)
- Provide direct access to remote services by offering only a user interface (thin client approach)

Example on page 125

Client side software for distribution transparency

Often, parts of the processing and data level in a client-server application are executed on the client side as well (Special case: embedded client software like ATMs, cash registers, barcode readers). Client software also comprises components for achieving distribution transparency. Access transparency: generation of a client stub from an interface definition of what the server has to offer, stub hides the actual communication and possible differences in machine architectures. Several ways to handle location, migration and relocation transparency, convenient naming system (and in many cases cooperation with the client side software) is crucial. Replication transparency: often implemented by means of client-side solutions (forwarding each request to all replicas and collecting all responses and only passing back a single one). Failure transparency: usually done through client middleware, Concurrency transparency: can be handled through special intermediate servers (transaction monitors).

3.4 Servers

General design issues

- Iterative server: receives request, handles it, responds if necessary
- concurrent server: passes the request to a separate thread or process and waits for the next request (e.g. multithreaded server, or fork a new process for each request). The thread that handles the request is responsible for the response.

Each server listens to a specific end point (port). Either globally assign end points for well-known services (HTTP port 80, FTP port 21), or look up the endpoint.

How to interrupt a server: sometimes only solution is exit the client application and restart it. Better solution: have the server listen to a separate control end point to which the client sends out-of-band data, or send out-of-band data to the same endpoint where the normal data is being sent. TCP has possibility for sending urgent data.

Stateless vs stateful servers: Stateless server does not keep information on the state of its clients and can change its own state without informing the client. A stateful server maintains persistent information on its clients (e.g. fileserver that allows a client to keep a local copy of a file and update it). Con: if the server crashes, information might be lost.
Difference between session state and permanent state: session state lost? just repeat the last request.

Being stateful or stateless should not affect the services provided by the server.

Object servers

The object server by itself does not provide a specific service. The specific services are implemented by the objects that reside in the server. The server only provides means to invoke local objects. Changing services can easily be done by adding or removing objects (different approaches discussed page 133, mentions object wrappers and activation policies, followed by 4 pages of notes).

Example: The Apache Webserver

> 📗 watch the lecture, I have no idea whats relevant here, sorry

Server clusters

Local-area clusters
Server cluster is a collection of machines connected through a network where each machine runs one or more servers. Server clusters are generally organized in 3 tiers: a logical switch through which client requests are routed, application processing, and data-processing (aka database and fileservers). Not all server clusters follow this strict separation.

Wide-area clusters
Usually, one would have to deal with multiple administrative organizations, but this can nowadays be worked around by using the facilities of a single cloud provider. Request dispatching: dispatcher has to estimate the latency between the client and several servers. A server is selected by the dispatcher (might be a DNS) based on the redirection policy.

Case Study PlanetLab page 149

3.5 Code migration

Reasons for migrating code

Moving a running process from one machine to another is difficult and costly. Aim: load balancing.
Now: use each node so efficiently, that the least possibly number of nodes is used. Migrating virtual machines is less intricate than migrating a process.

Migrating parts of the server to the clients is also an approach.

Migration in heterogeneous systems

With homogeneous systems, the migrated code can easily be executed at the target machine. Distributed systems are however, usually heterogeneous. Either you basically use an adapter-strategy, or you migrate not only a process but an entire computing environment.

Real-time migration of a virtualized operating system in a cluster of servers where a tight coupling is achieved through a shared LAN. Two major problems:

- Migrating the entire memory image. How to handle:
  - pushing memory pages to the new machine and resending the ones modified during the migration process
  - stopping the current VM, migrating memory, starting the new VM
  - letting the new VM pull in pages as needed
- Migrating bindings to local resources

4 Communication

4.1 Foundations

Layered Protocols

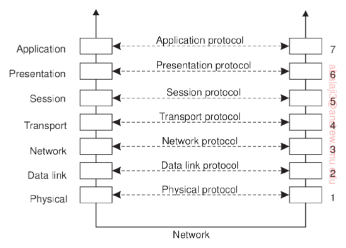
Absence of shared memory → all communication in distributed systems is based on sending and receiving messages.

OSI reference model

OSI reference model is designed to allow open systems to communicate. An open system communicates with another open system by using standard rules regarding format, content and meaning (communication protocols). A protocol provides a communication service. There are two types of communication services.

- Connection-oriented service: before exchanging data, a connection needs to be established
- Connectionless service: basically like dropping a letter in a mailbox

In the OSI model, communication is divided into seven layers. Each layer provides an interface to the one above it, this way getting a message from A to B is split into little chunks.



- Physical layer: standardizes how two computers are connected and how 0s and 1s are represented
- Data link layer: detect and maybe correct transmission errors, protocols to keep a sender and receiver in the same place
- Network layer: contains protocols for routing a message through a computer network, protocols for handling congestion
- Transport layer: protocols for directly supporting applications (establish reliable communication, support real-time streaming of data)
- Session layer: Provides support for sessions between applications
- Presentation layer: how data is represented in a way that is independent of the hosts on which communicating applications are running
- Application layer: everything else (e-mail protocols, web access protocols, file transfer protocols…)

> 📗 5 page note about different protocols page 167 (HTTP, FTP, UDP, TCP/IP, real-time transport protocol RTP)

Middleware protocols

> 📗 not sure which parts are relevant, watch the lecture

Types of Communication

Persistent communication: a message that has been submitted for transmission is stored by the communication middleware as long as it takes to deliver it to the receiver, so the sending application does not need to continue execution after sending, and the receiving application does not need to execute when the message is sent.

Transient communication: messages are stored by the communication system only as long as the sending and receiving application are executing.

Asynchronous communication: sender continues immediately after submitting a message

Synchronous communication: sender blocks until the request is accepted (three possible points of synchronisation: when middleware takes over, until the request has been delivered, waiting until the request has been fully processed and the recipient returns a response).

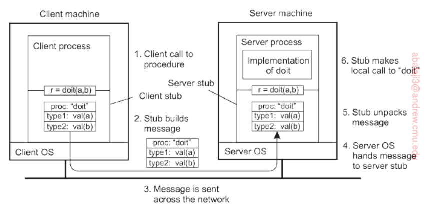
4.2 Remote procedure call

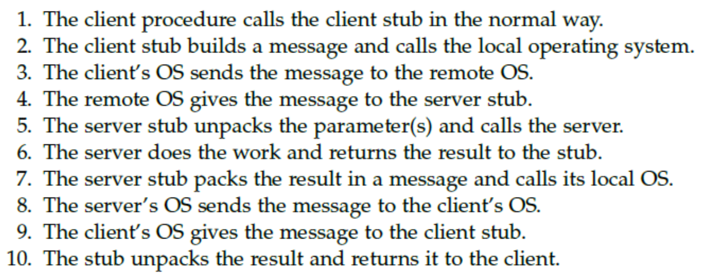
Allow programs to call procedures located on other machines. When A calls B, A is suspended and the execution of the called procedure takes place on B. Information can be transported in the parameters/the procedure result.

Problems: they execute in different address spaces, parameters and results have to be passed, either or both machines can crash.

Basic RPC operation

Goal: make RPC transparent. Insanely detailed descriptions page 175ish.




1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameter(s) and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs the result in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns it to the client.

Parameter passing

Packing parameters into a message is called ***parameter marshaling***. Byte order (little endian vs big endian) is important here. Usually, big endian is used for transferring bytes across a network.

How are pointers, or references in general, passed: With great difficulty. Since a pointer is referring to come local address, it would not make sense on the other machine. One solution: Just dont do it. Other solution: just copy the data over (for arrays, strings and such), and if it will not be read again, we do not need to copy it back. Other solution: Global references (if both processes share a file → pass a file handle instead of a pointer)

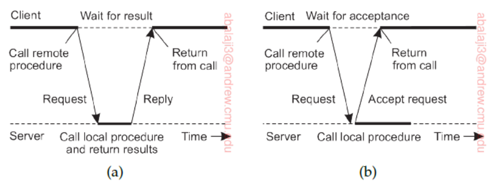
RPC-based application support

Two ways to support RPC-based application development (Described in detail page 183):

- let a developer specify exactly what needs to be called remotely, from which complete client-side and server-side stubs can be generated
- embed remote procedure calling as part of a programming language environment (like Java RMI)

Variations on RPC

Asynchronous RPC



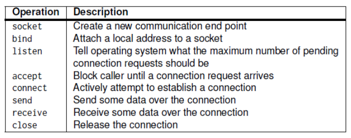
For situations where there is no result to return to the client (or the client can do something else in meantime). The server just quickly acknowledges that it has received the request, the client can then simply continue. Or it doesn't even wait for the acknowledgement (one-way RPC)

In cases where the client can do something else in the meantime, a callback makes sense (deferred synchronous RPC).

4.3 Message-oriented communication

Simple transient messaging with sockets

Socket offers an abstraction over the actual port that is used (details in the book)



Operation	Description
`socket`	Create a new communication end point
`bind`	Attach a local address to a socket
`listen`	Tell operating system what the maximum number of pending connection requests should be
`accept`	Block caller until a connection request arrives
`connect`	Actively attempt to establish a connection
`send`	Send some data over the connection
`receive`	Receive some data over the connection
`close`	Release the connection

Message-oriented persistent communication

Also referred to as message-queueing systems. They provide support for persistent asynchronous communication. Essence of the system: they offer intermediate-term storage without requiring the sender or receiver to be active during message transmission. As opposed to sockets and MPI (not discussed in the lecture), the message transfers are usually allowed to take minutes.

Message-queueing model: applications communicate by inserting messages in specific queues. Either specific queue per application, or multiple applications share a queue. No guarantees about the message actually being read are given. 4 different combinations: sender/receiver running/passive. Only if the case where both are passive is supported, the system is truly persistent.

General architecture of a message-queueing system
Queues are managed by a queue manager (✨Oh wow I was not expecting that ✨). It is either a separate process or implemented by a library that is linked with an application. Generally, an application can put messages only into a local queue and getting a message is only possible by extracting it from a local queue. The queue manager QMA that handles application A's queue, will be places on the same machine or at the very least in the same LAN.
How is the destination queue addressed: Logical, location-independent names preferred to enhance location transparency. Each name should be associated with an address (like a (host,port) pair).
How is the name-to-address mapping made available to a queue manager: implement the mapping as a lookup table, copy that table to all managers (every time a queue is added or named, all tables need to be updated)

Efficiently maintaining name-to-address mappings: Special queue managers that operate as routers (forward messages to other queue managers). The message queueing system may grow into a complete, application-level overlay network.

Message brokers
Important application area of message-queueing systems is integrating existing and new applications into a single, coherent distributed information system. The book mentions here, for the 10000th time that sender and receiver need to adhere to the same format. If not, we need protocol converters. N*N in a system with N application. Agreeing on the same protocol will not work for message-queueing systems (problem with the abstraction level). Solution: lift the level of abstraction (like an XML message), so that messages carry information on their own organization and what has been standardized.
Common approach: learn to live with the differences and make conversion as simple as possible. Conversions are handled by special nodes in a queueing network, known as ***message brokers***. A message broker can be as simple as just reformatting messages. Or, it may act as an application-level gateway. For each pair of applications, there is a separate subprogram that converts messages between two applications (like a plugin for a broker).

4.4 Multicast communication

Aka sending data to multiple receivers. With p2p technology and structured overlay management, setting up communication paths became easier.

Application-level tree-based multicasting

Two approaches for nodes to organize themselves:

- into a tree → unique path between every pair of nodes
- mesh network → multiple paths, thus more robust concerning connection failure

  Tree: might be really inefficient if nodes that frequently "talk" are far apart. Three different metrics for measuring quality:

  - Link stress (how often does a packet cross the same link)
  - Stretch (ratio in the delay between two nodes)
  - Tree cost (construct minimal spanning tree)

  Assume a multicast group has a known node that keeps track of the nodes that have joined the tree. A new node contacts this node to find which member would be best as a parent. (selection based on: which node has less than k neighbours, details page 225)

Flooding-based multicasting

Construct an overlay network per multicast group, now multicasting to the group is the same as broadcasting to the group (Con: a node belonging to multiple groups needs to maintain a separate list of neighbours per group).

Flooding: each node forwards a message to each neighbour, except the one it received the message from. If a node keeps track of received messages, it can ignore duplicates. Roughly twice as many messages as there are links in the networks are sent → inefficient (calculations page 226). Solution: probabilistic flooding (only forward a message to a neighbour with a certain probability). Many approaches and heuristics to optimize this (e.g. number of neighbours influences probability), see page 227.

Gossip-based data dissemination

Based on how epidemics spread.

If a node holds data it is willing to spread, it is called ***infected.*** If it holds data, but won't spread it, it is called ***susceptible***. If an updated node is neither willing nor able to spread its data, it is said to have been ***removed.***

Anti-entropy: Propagation model where a node picks another node and they subsequently send each other updates (only pull in updates, only push out updates, or do both).

Rumor spreading: Specific variant of epidemic protocols. If a node has been updated, it tries to push the update to an arbitrary other node. If that other node already knows the news, the original node is discouraged, and becomes removed.

Directional Gossiping: a node has its favourite other nodes it updates.

Removing data
Spreading the deletion of a data item is hard. Solution: record the deletion as just a normal update and keep a record of the deletion, thus old copies will not be interpreted as something new. This is called

✨spreading a death certificate✨. After some time, the death certificate is removed, or we would have a growing database of useless history. Few nodes still keep them just in case.

5 Naming

📗 CH 5

- Section 5.1 is covered completely, apart from the Notes.
- Section 5.2 is mostly covered, apart from "Home-based approaches" and Notes.
- Section 5.3 is mostly covered, but the Notes are not discussed at all; DNS is discussed in the lecture, but not as extensively as in the textbook; NFS is not discussed in the lecture at all.
- Section 5.4 is partially covered. The basics of directory services and LDAP are covered, but we do not discuss decentralized implementations or the notes.

Implementation of a naming system is itself often distributed across multiple machines. How the distribution is done largely affects efficiency and scalability.

5.1 Names, identifiers and addresses

To operate on an entity (hosts, printers, files, mailboxes, webpages… literally everything) we need to be able to access it. The name of an access point is ***address***. There can be multiple access points. An address could be used as a name, but that is human unfriendly, and as soon as an access point changes, the reference is invalid. A name for an entity, which is independent from the address is often better and more flexible (***location independent***). An identifier uniquely identifies an entity. It has the following characteristics:

- Refers to at most one entity
- Each entity is referred to by at most one identifier
- An identifier is never reused

A naming system maintains a ***name-to-address-binding*** aka a table with (name, address) pairs. In distributed systems, a centralized table is not going to work. Page 240 has a brief description of recursive domain registration and root server.

5.2 Flat naming

Identifiers are often simple bit strings. How can we locate an entity when we only know the identifier?

Simple Solutions

- Broadcasting: broadcast a message containing the identifier of the entity, each machine checks whether or not it has the entity, if a machine has it, it responds with a message containing the address of the access point. This is used in the Internet Address Resolution Protocol (ARP). Broadcasting becomes inefficient when the network grows. Solution: switch to multicasting (details page 242).
- Forwarding pointers: Follow the chain of forwarding pointers from a to b, from b to c…. Two drawbacks: chain can become really long, all intermediate links in the chain have to maintain their

part of the chain as long as needed (broken links are a problem).

- Dynamic Systems (from the lecture): work better than the other two when it comes to very large systems. A node only knows a subset (can be dynamic) of nodes

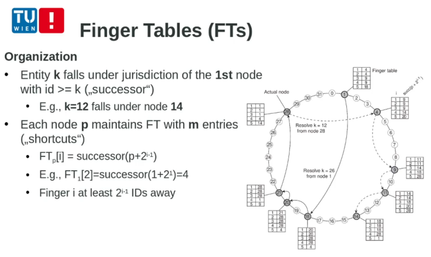
Distributed Hash Tables

M bit identifier space to assign identifiers to nodes and keys to entities. An entity with key k falls under jurisdiction of the node with the smallest identifier id ≥ k. This node is called ***successor*** of k. A ***finger table*** is used.

DHT explanation from lecture: user is connected to the nearest node. The request is forwarded with O(logN) hops to the target node.

How to look up nodes:

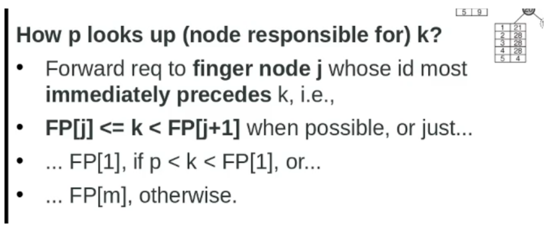
- naive way: propagate clockwise to successor until it is found
- cool way: instead of successor, try to just send it "closer" via a shortcut. Each node maintains a table with a number (m) of entries. Each entry is called a "finger" (→ finger table). The ith entry is at least a number of IDs away. Send the request to the node in the finger table that ***most immediately precedes*** k. If currentNode < key < finger[1], send it to finger[1]. Otherwise, send it to the last entry.



Finger Tables (FTs)

Organization

- Entity **k** falls under jurisdiction of the **1st** node with id >= k („successor")
  - E.g., **k=12** falls under node **14**
- Each node **p** maintains FT with **m** entries („shortcuts")
  - $FT_p[i]$ = successor($p+2^{i-1}$)
  - E.g., $FT_1[2]$=successor($1+2^1$)=4
  - Finger i at least 2^{i-1} IDs away



How p looks up (node responsible for) k?

- Forward req to **finger node j** whose id most **immediately precedes** k, i.e.,
- **FP[j] <= k < FP[j+1]** when possible, or just...
- ... FP[1], if p < k < FP[1], or...
- ... FP[m], otherwise.

Hierarchical approaches

In a hierarchical scheme, a network is divided into a collection of domains. The top-level domain spans the entire network, and each domain is divided into multiple smaller subdomains. The lowest level is called a leaf domain and usually corresponds to a LAN in a computer network or a cell in a mobile telephone network. Delete and insert operations described on page 254.

5.3 Structured naming

aka human readable names

Name spaces

Names are organized into namespaces. Namespaces for structured names can be represented as labelled directed graphs with leaf nodes (named entities), and directory nodes (stores a table in which outgoing edges are represented as (node identifier, edge label) pairs → directory table). ***Absolute path name***: path starts at root, otherwise: ***Relative path name***. Global vs local names.

Name resolution

The process of looking up a name is called name resolution. Given a path name, it should be possible to look up anything stored in the node with that name.

Closure mechanism: knowing how and where to start name resolution. It deals with selecting the initial node.

Linking and mounting: An alias is another name for the same entity. Two approaches to implementing an alias: allow multiple absolute path names to refer to the same node in the naming graph (hard links), or represent an entity by a leaf node, but instead of storing the address or state of that entity, the node stores an absolute path name (symbolic links).
Name resolution can also be used to merge different namespaces. The directory node containing the node identifier from a node of a ***foreign name space*** is called a ***mounting point***. Mounting a foreign namespace in a distributed system requires the following information: name of an access protocol, name of the server, name of the mounting point in the foreign name space.

The implementation of a name space

In a large-scale distributed system, it is necessary to distribute the implementation of a name space over multiple nameservers.

Namespace distribution: usually organized hierarchically in three layers:

- Global layer: root node and other high-up nodes, everything in there is hardly ever changed
- Administrational layer: formed by directory nodes that together are managed within a single organization
- Managerial layer: nodes that typically change regularly like hosts in the local network.

Page 266 for availability and performance, but it’s not really interesting. Talks about caching and the frequency of updates.

Implementation of name resolution:
We assume (For the moment) that nameservers are not replicated and no client-side caches are being used. Each client has access to a local name resolver.

- Iterative name resolution: name resolver hands over the complete name to the root name server. The root server resolves the path as far as possible and returns the result to the client. The client contacts the next server…
- Recursive name resolution: Instead of passing the result back to the client, a nameserver passes the request to the next nameserver it finds. Drawback: higher performance demand per server. Advantages: caching is more effective, communication costs may be reduced (explanations for that page 269).

Example: The Domain Name System

Namespace organized as a rooted tree. Subtree is called ***domain***, path name to its root node is called ***domain name***.

Implementation: Global layer and administrational layer.

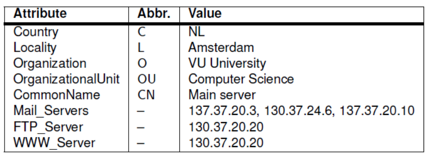
5.4 Attribute-based naming

Location independence and human friendliness are not the only criteria for naming entities. Sometimes, we require a description of what we’re searching for. These (attribute,value) pairs are referred to as ***attribute-based naming.*** It is up to the naming system to return one or more entities that meet the description.

Directory services

Attribute-based naming sytems are also known as ***directory services,*** whereas systems that support structured naming are called ***naming systems***. Designing an appropriate set of attributes is not trivial and often has to be done manually. In the Resource Development Framework, resources are described as triples (person,name,Alice) means a resource named person with the name Alice. Looking up values in an attribute-based naming system requires exhaustive search through all descriptors.

Hierarchical implementations: LDAP



Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	VU University
OrganizationalUnit	OU	Computer Science
CommonName	CN	Main server
Mail_Servers	–	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	–	130.37.20.20
WWW_Server	–	130.37.20.20

Common approach: combine structured and attribute-based naming. Many of these systems use the lightweight directory access protocol (LDAP). Details page 287ish.

8 Fault Tolerance

📗 CH 8

- Section 8.1 (we do not discuss the part of Note 8.1 where an expression of reliability is derived, but we discuss the fault-tolerance related metrics).
- Section 8.2: The first half (up to page 449), however "Consensus in faulty systems with crash failures" only implicitly by discussing Paxos (Note: Reading these two pages surely help to understand how Paxos works). Our discussion of Paxos in the lecture is more practice-oriented. "Consensus in faulty systems with arbitrary failures" (and the following subsections) are not discussed.
- Section 8.3: Completely covered.
- Sections 8.4-8.6: Not covered.

Partial failure: only part of the system fails. Goal: recover automatically without seriously affecting the performance.

8.1 Introduction to fault tolerance

Basic concepts

What does it mean for a distributed system to tolerate faults? Being fault tolerant is strongly related to ***dependable systems*** which describes a number of requirements, including the following:

- Availability: A system is ready to be used immediately. A highly available system is one that will most likely be working at a given instant in time.
- Reliability: a system can run continuously without failure. If a system goes down a millisecond every hour, it is still available more than 99% of the time, but not reliable.
- Safety: when a system temporarily fails, no catastrophic event happens.
- Maintainability: refers to how easily a failed system can be repaired.

A system is said to fail when it cannot meet its promises. An error is a part of a system's state that may lead to a failure. The cause for an error is called a fault.

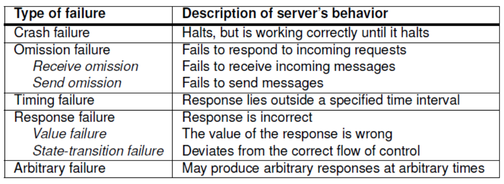
Transient faults occur once and then disappear.

Intermittent faults occur, vanish on their own accord, reappear…

Permanent faults exist until the faulty component is replaced.

Failure models

To determine how serious a failure is, there are several classification schemes.



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

- crash failure
- receive omission: maybe the server didn't get the request, maybe no thread was listening
- send omission: server did its work but does not respond, maybe the send buffer overflowed

- timing failure: e.g. streaming video and sending data too soon might lead to buffer overflow, late response leads to performance failure
- value failure: simply wrong reply
- state-transition failure: unexpected reaction
- arbitrary failure: maybe the server produced an unexpected result that is not detected as incorrect

In an asynchronous system, when Q crashed, P cannot simply assume that Q crashed. In a synchronous system, when Q shows no more activity, P can conclude that it crashed. Most distributed systems are partially synchronous. Most of the time it behaves synchronously, with occasional asynchronous behaviour.

- Fail-stop failures: crash failures that can reliably detected
- Fail-noisy failures: P will ***eventually*** find out that Q crashed
- Fail-silent failures: P cannot distinguish between crash failures and omission failures
- Fail-safe failures: arbitrary failures that do not do any harm
- Fail-arbitrary failures: Q may fail in any possible way, even unobservable and harmful

Failure masking by redundancy

- Information redundancy: something like hamming code
- Time redundancy: an action may be repeated (e.g. if a transaction aborts, you can redo it with no harm)
- Physical redundancy: extra equipment or processes can be added to make it possible for the system to tolerate the loss or malfunction of some components

8.2 Process resilience

Resilience by process groups

The key to tolerating a faulty process is organizing several identical processes into a group. Important: when a message is sent to the group itself, all members receive it. If one process fails, another can take over.

Group organization: In some groups, all processes are equal, while other groups have a hierarchy. If a hierarchical group loses its coordinator, the group comes to a grinding halt and a new leader needs to be elected.

Membership management: Method for creating and deleting groups as well as allowing processes to leave and join is necessary, for example via a group server. Disadvantage: single point of failure. Other approach: manage groups in a distributed way, through multicasting (”hi, I’m joining”, “bye, I’m outta here). Crashes can only be detected when it is discovered that a process does not respond to anything.
If many processes go down and the group can no longer function, a protocol is needed to rebuild the group.

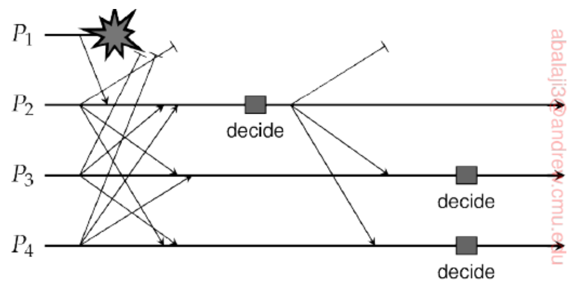
Failure masking and replication

Two ways to approach replication:

- Primary-based protocols: hierarchical group of processes, when the primary fails, the backups elect a new primary
- Replicated-write protocols: flat group (no hierarchy) → no single point of failure, con: cost of distribution coordination

How much replication is needed? k-fault tolerant: faults in k components can be survived and specifications can still be met. K-fault tolerant group needs k+1 members to tolerate crash/omission/timing failures and 2k+1 components for arbitrary (=byzantine) failures. (Math on page 435)

Consensus in faulty systems with crash failures

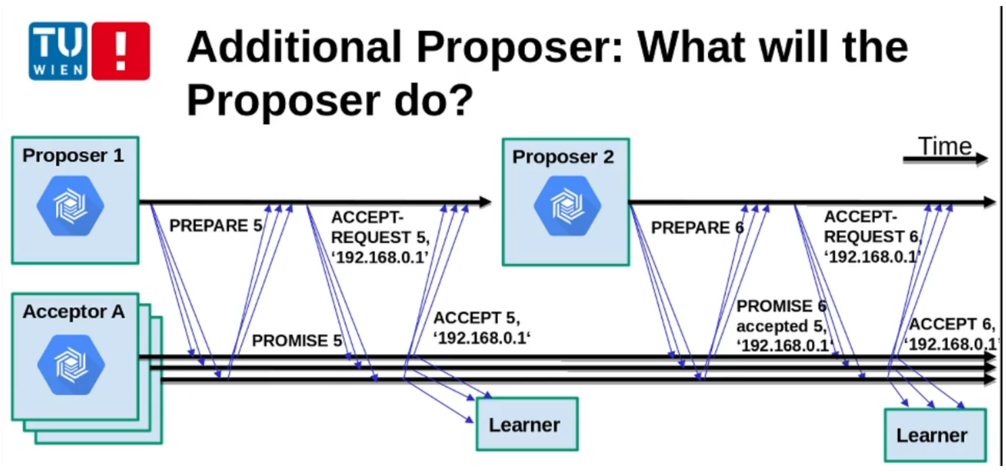


Flooding based consensus

In a fault-tolerant process-group, each nonfaulty process executes the same commands, in the same order, as every other nonfaulty process. The group members need to reach consensus on which command to execute.

Example: Paxos

Widely adopted consensus algorithm. Clients request a specific operation to be executed. At the server side, each client is represented by a single proposer, which attempts to have a client’s request accepted.
A single proposer has been designated as the leader → drives towards consensus.
A proposed operation is accepted by an acceptor → majority vote


Additional Proposer: What will the Proposer do?

📗 Watch lecture for detailed explanation of PAXOS! For the exam it is important that you can also tell what is a valid state and what not

8.3 Reliable client-server communication

Point-to-point communication

Reliable point-to-point communication is usually established by using reliable transport protocols like TCP. Crash failures can only be marked if the system automatically attempts to set up a new connection.

RPC semantics in the presence of failures

Goal of RPC: hide communication by making remote procedure calls look like local calls. When an error occurs, it is not easy to mask the differences between local and remote calls.

Five classes of failures that occur in RPC systems:

- client is unable to locate the server: possible solution: throw an exception, drawback: transparency destroyed
- request message from client to server is lost: solution: start a timer when sending the request, resend the request if the timer expires before an acknowledgement or reply comes
- server crashes after receiving request: the timer expires, but we do not know if the request was lost or the server crashed.
  Solutions:
  - wait until the server reboots and try again (call-at-least-once semantics), this guarantees that the RCP has occurred at least once, possibly more times.
  - give up immediately and report a failure (call-at-most-once semantics)
  - guarantee nothing. The client gets no help and no promises about what happened. No one knows how many times the RPC was called. Pros: easy to implement
- reply from server to client is lost: rely on a timer again, resend the request. But the client does not know why no reply came back. Solution: try to structure every request in an idempotent way, so it can be resent infinitely. Other solution: have the server keep track of the most recently received sequence number, so that the server can differentiate between the original request and the retransmission, and can refuse the retransmitted request.

- client crashes after sending a request: orphan. Solutions:
    - ***Orphan extermination:*** log entry specifying what to do in case of an orphan, reboot, check the log, ***kill the orphan***. Con: expensive (disk record for every RPC). It may not even work, because orphans can also have RPCs (grandorphan created), furthermore if the network is partitioned, the orphan cannot be killed, even if it can be located.
    - ***Reincarnation:*** divide time into epochs, when a client reboots, broadcast a message to all machines declaring the start of a new epoch. This causes all remote computations to be killed. Orphans that still live because of partition can be detected from the invalid epoch number.
    - ***Gentle Reincarnation:*** When an epoch broadcast comes in, each machine kills its RPCs if their owners cannot be located.
    - ***Expiration***: each RPC is given a certain amount of time, if it cannot finish on time, it needs to ask for more time.

Killing an orphan can have consequences: if an orphan has locks on files or database records that remain forever, or made an entry to start another process in the future.

6 Synchronization & Timing (Called Coordination in the book)

📗 CH 6

- Section 6.1: Everything except for "Clock synchronization in wireless networks" and Notes 6.1 & 6.2.
- Section 6.2: Logical Clocks and Vector Clocks, but without the Example: Total-ordered multicasting and without Notes 6.3 & 6.4.
- Section 6.3: Not covered.
- Section 6.4: The bully algorithm and the ring algorithm are covered, but "Elections in wireless environments" and "Elections in large-scale systems" are not.
- Sections 6.5+6.6: Not covered.

6.1 Clock synchronization

Physical clocks

Time is relative and is not the exact same on different machines of a distributed system. The difference in time values is called ***clock skew.*** In real time systems, the actual clock time is important, not just the relative time. Thus, external physical clocks are needed. How do we synchronize with real clocks? How do we synchronize clocks with each other?

Universal Coordinated Time (UTC) is the standard and I hope someone got fired for that abbreviation.

Clock synchronization algorithms

If one machine has a UTC receiver, the goal is trying to synchronize the other machines to it (external synchronization). If no machine has a receiver, the goal is keeping the machines together as best as possible (internal synchronization). Page 303 has a description of how the clocks actually work.

Network Time Protocol
Let clients contact a time server and the delay it takes for the answer to get back is estimated. A server with a reference clock is referred to be a ***stratum-1-server*** and when A contacts B stratum-1, A only adjusts the time if its own stratum is higher, else B will adjust to A (aka lowest stratum wins).

Berkeley algorithm
Usually, the time server is passive. In berkeley unix, the timeserver (time daemon) is active and polls the machine and asks them what time it is. It then computes average time and tells the other machines to adapt their clocks. This works for systems without a UTC receiver, but the daemon's time must be set manually periodically. This is just an internal synchronization mechanism.

6.2 Logical Clocks

Logical clock: time does not even matter, two nodes just must agree on something by keeping track of each other's events.

Happens-before relation. If neither a → b nor b → a are true, then a and b are concurrent.

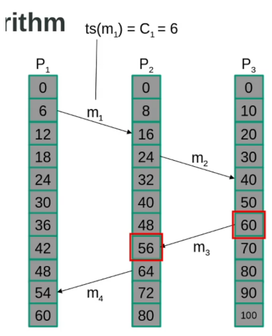
How do we maintain a global view on the system's behaviour that is consistent with the happened-before relation? Solution: attach a timestamp C(e) to each event e, satisfying the following properties:

- if a and b are two events in the same process and a → b, we demand that C(a) < C(b)
- if a corresponds to the sender of a message and b to the recipient of the same message, then C(a) < C(b)

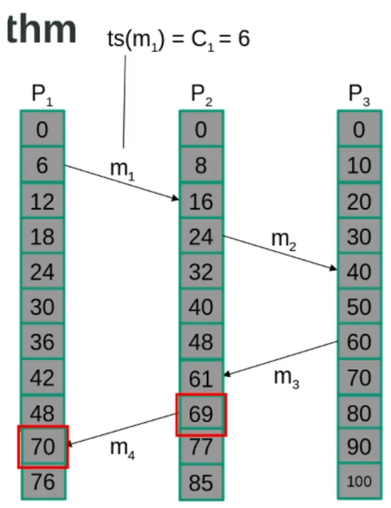
Lamport's logical clocks

Each process maintains a local clock which can be considered an event counter. The values of this counter grow monotonically. The clocks of all processes do not run at the same rate (e.g. P1: 0-6-12-18… P2: 0-8-16-24…).

Before an event takes place, the counter of the process is incremented by some value. E.g. if a process wants to send a message, it will attach its current time stamp to the message. If the receiver's timestamp is higher than the attached timestamp, the happened-before relation works. If the timestamp is lower than the attached timestamp, the receiver has to adjust its current timestamp to be higher. It advances its own timer, so that it is at least one unit bigger than the attached timestamp.



before

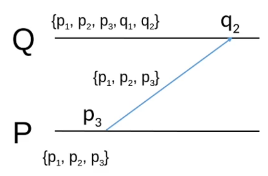


after Lamport's algorithm

Limitations: $a \rightarrow b \Rightarrow C(a) < C(b)$ but $C(a) < C(b) \not\Rightarrow a \rightarrow b$

Lamport's logical clocks lead to a solution where all events in a distributed system are totally ordered with the property that if A happened before B, then A will also be positioned in that ordering before B, that is C(A) < C(B). With Lamport clocks, nothing can be said about the relationship between two events by merely comparing their time values.

Vector clocks

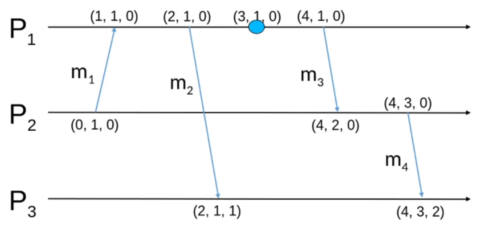


Allows us to know that if VC(a) < VC(b) $\Longrightarrow$ a causally precedes b

Approach: Assign each event a name (p_k is the k^{th} event that happened in process p). Causal history H(p2) of event p2 is {p1,p2}

If a message is sent, the causal history is sent as well. The causal history of the sender is merged into the causal history of the receiver

It would grow infinitely. Solution: we do not have to send the whole causal history, because p3 implies that p1 and p2 have already happened → only send the last event.



Basic Approach: Each process maintains a vector clock vc. VC has an element for each process in the system. VC_i[i] = the number of events that happened in P_i. VC_i[j] = k means that P_i ***believes*** that k events have previously occurred in P_j. (More could have occurred since the last communication). When sending a message, a process increments its counter and attaches it to the message as a timestamp. After receiving a message, the receiving process adjusts its vector clock to be the maximum of either the current value for each element or the elements of the vector attached to the message. Then it will increment its own counter.

Possible Conflict: We cannot say if a message causally precedes another if e.g. P1(4,1,0) and P2(2,3,0) because 4>2 but also 1<3. But we also do not know if it is a conflict. The operations could be two read operations as well.

6.4 Election algorithms

Many distributed algorithms need a leader/initiator/coordinator… It does usually not matter which process does it. We assume every process knows the identifier of every other process.

The bully algorithm

When a process notices that the coordinator is no longer responding to requests, it initiates an election. It sends an election message to all processes with higher identifiers than its own. If no one responds, this process is the new coordinator, if one of the higher-ups responds (holds its own election as well), it becomes the coordinator and announces its victory with a message to all processes. If a process was down and comes up again, it holds an election. If it has the highest number, it wins. The biggest guy always wins → Bully Algorithm.

A ring algorithm

We assume each process knows its successor. When a process notices the coordinator is not functioning, it sends an election message containing its own process identifier to its successor. If the successor is down, it goes to the successor's successor and so on. At each step, the process adds its own process number to the "candidate list". When the message comes back to the original sender, it picks the highest number and propagates the message who the coordinator is. If two processes send an election message, nothing bad happens, they both come to the same conclusion regarding the highest process number.

7 Consistency & Replication

📗 CH 7

- Section 7.1
- Section 7.2, except for "Continuous consistency" (incl. Note 7.1, 7.2) and "Grouping operations." We do not discuss Notes 7.3 and 7.4, but they might be interesting to take a look at.
- Section 7.3
- Section 7.4, except for "Managing replicated objects." We also did not cover the 3rd algorithm of Note 7.5 (Szymaniak et al., Fig. 7.20), nor Note 7.6. Both are worth reading, though.
- Section 7.5, except for "Continuous consistency", "Active replication," and "Cache coherence protocols"
- Section 7.6, except for the part on caching dynamic content
- Section 7.7

Quick overview:

📗 **DATA CENTRIC:**

- *Consistency Models*

- Sequential Consistency
    - Causal Consistency
    - Eventual Consistency
- *Consistency Protocols*

- Primary Based
    - Replicated Write (e.g., Quorum Based)

CLIENT CENTRIC:

- *Consistency Models*

- Monotonic Read
    - Monotonic Write
    - Writes Follows Reads
    - Read Your Writes
- *Consistency Protocols*

    - Provide a way to propagate write operations to replicas that a client visits
    - Based on the transfer of sets of write operations when needed

The read set for a client consists of the writes relevant for the read operations performed by a client. Likewise, the write set consists of the (identifiers of the) writes performed by the client.

7.1 Introduction

Reasons for replication

- increase the reliability of a system
    - if one copy crashes, use another
    - protection against corrupted data
- performance
    - distribution copies over a wide area makes access faster
    - distribution copies over multiple servers helps with bottlenecks

Multiple replicas lead to consistency problems. When and how modifications (on all copies) need to be carried out determines the price of replication.

Replication as scaling technique

Placing copies of data close to the processes using them improves access times. Tradeoff: keeping replicas updated needs bandwidth. All replicas should be updated before another process reads/writes (tight consistency, as provided by synchronous replication). Keeping all replicas consistent requires global synchronization (costly in terms of performance). Solution: relax consistency constraints

7.2 Data-centric consistency models

Consistency model: a contract between processes and data store. Without a global clock, it is hard to define which write operation is the last one.

Consistent ordering of operations

- sequential consistency: the result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence order specified by its program. (Examples page 364)
- causal consistency: weaker than sequential consistency. Writes that are potentially causally related must be seen by all the processes in the same order. Concurrent writes may be seen in a different order on different machines. (Examples page 368)

Consistency vs coherence: a set is said to be consistent if it adheres to the rules described by the consistency model. Coherence models describe what can be expected to hold for only a single data item.

Eventual consistency

In many database systems, most processes hardly ever perform update operations, they mostly read. The question is: how fast should updates be made available to read-only processes. Usually very slowly, because we can assume that each process always accesses the same replica, so there will not be inconsistencies. This example (and the others on page 387) can be viewed as cases of (large-scale) distributed and replicated databases that tolerate a relatively high degree of inconsistency. If no updates take place for a long time, all replicas will gradually become consistent. This is called ***eventual consistency***. Write-write conflicts are handled by assuming that only a small group of processes can perform updates. In the case of conflicts, one specific write just wins → cheap to implement.

7.3 Client-centric consistency models

Look at this section in the book, there are many examples!

If the user accesses a database by connecting to a replica and then moves around and connects again, it might be a different replica. If the updates have not been propagated yet, an inconsistency occurs. That is why we need client-centric consistency. It provides guarantees for a single client concerning the consistency of accesses to a data store by that client. (No guarantees if another client modifies shared data → write-write conflict can occur).

Monotonic reads

A data store is said to provide monotonic-read consistency if it holds that if a process reads the value of data item x, any successive read operation on x by that process will always return that same value or a more recent value.

Monotonic writes

Often, propagating write operations in the correct order is important. In a monotonic-write consistent store, it holds that a write operation by a process on a data item x is completed before any successive write operation on x by the same process.

Read your writes

Read-your-writes consistency is provided if it holds that the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process. (example:

invalidate the cache after updating a page so that you actually see the new page and not the cached version).

Writes follow reads

Writes-follow-reads consistency is given if it holds that a write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.

7.4 Replica management

Where, when and by whom should replicas be placed, and which mechanisms are used for keeping them consistent.

Placement problem split into: placing replica servers (best location for a server) and placing content (on what server will the content be placed)

Finding the best server location

Book page 383. I have no fucking clue whats going on.

Content replication and placement

3 different types of replicas:

- permanent replicas: the initial set of replicas. Example: distribution of a website comes in two forms:
    - the files that constitute a site are repilcated across several servers at the same location. When a request comes in, it is forwarded to one of the servers
    - Mirroring: a web site is copied to a limited number of servers (mirror sites), which are geographically spread.
- server-initiated replicas: copies of a data store that exist to enhance performance, created by the server. As long as guarantees can be given that each data item is hosted by at least one server, it may suffice to use only server-initiated replication and no permanent replicas. However, these are often used as a backup.
- client initiated replicas: aka caches. Detailed explanation (cache hit, cache miss… page 387)

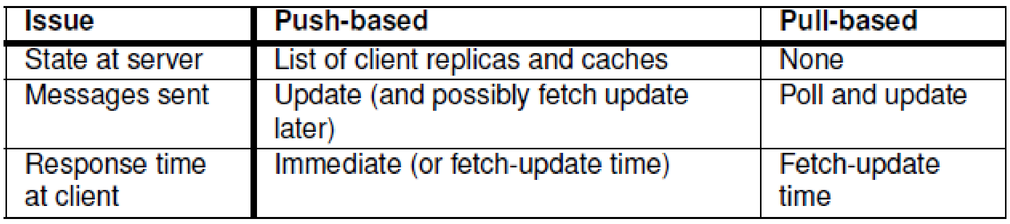
Content distribution

The propagation of updated content to the relevant replica servers

- state versus operations: what is actually to be propagated
    - only a notification of an update (e.g. invalidation protocols)
    - transfer data from one copy to another: useful when the read-to-write ratio is very high. Possibly only log the changes and transfer that, to save bandwidth. Multiple modifications packed into a single message → lowering communication overhead
    - propagate the update operation to other copies: do not send data modifications but tell each replica what update it should do and the parameters it will need (active replication)
- Pull versus push protocols:
    - push based (server based) approach: updates are propagated to other replicas without them asking. Often used between permanent and server-initiated replicas, can also be used to push

updates to caches. Usually applied when strong consistency is required. Efficient when read-to-update ratio at each replica relatively high.

- pull based (client based) approach: server or client requests another server to send any updates. Often used by caches. Efficient when the read-to-update ratio is rather low. Main drawback: response time increases in case of a cache miss.



Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

- Lease: promise by a server that it will push updates to the client for a specified time. When the lease expires, the client needs to request a new lease, or it needs to pull an update from the server. Convenient mechanism for dynamically switching between a push-based and a pull-based strategy. Three types:
  - Age based: given out on data items depending on the last time the item was modified. Data that has not been modified for a long time, is expected to stay the same for a long time.
  - Renewal-frequency based: how often a specific client requests its cached copy to be updated. The server will hand out a long-lasting lease to a client whose cache is often refreshed
  - State-based: When the server is gradually becoming overloaded, it lowers the expiration time of new leases.

- Unicasting vs multicasting
  - unicast: the server sends its update to each server with a separate message
  - multicast: the underlying network takes care of sending the message to multiple receivers.

7.5 Consistency Protocols

Primary-based protocols

Each data item x in the data store has an associated primary, which is responsible for coordinating write operations on x. The primary can be fixed at a remote server or write operations are carried out locally after moving the primary to the process where the write operation is initiated.

- Remote-write protocols: all write operations need to be forwarded to a fixed single server. Read operations can be carried out locally (”primary backup protocols”). A process wanting to perform a write operation on x forwards that operation to the primary. The primary performs the update on its local copy of x and forwards the update to the backup servers. Each backup server performs the update and sends an acknowledgement to the primary. When the primary has received all acknowledgements, it sends an acknowledgement to the initial process, which informs the client. Problem: takes really long. Solution: nonblocking approach, but that leads to problems with fault tolerance. Primary-backup protocols provide a straightforward implementation of sequential consistency.

- Local-write protocols: When a process wants to update data item x, it locates the primary and moves it to its own location. Pro: Successive writes carried out locally. This can be applied to mobile computers that are able to operate in disconnected mode. Before disconnecting, the computer becomes the primary for each data item it expects to update. When connecting again, updates are simply propagated. (Blocking, non blocking approaches page 400)

Replicated-write protocols

Writes can be carried out at multiple replicas.

- (Active replication: The operation is sent to each replica, and the replica carries it out. Problem: same order needed everywhere.) not relevant for the exam
- Quorum-based protocols: Clients need to request and acquire the permission of multiple servers before reading or writing a replicated data item (majority vote needed). Example: five servers, if three have version 8 of a file, the other two cannot have version 9. (Read-quorum, write-quorum page 417)

Implementing client-centric consistency

Naive implementation: Each write operation gets an identifier which is assigned by the server to which the write has been submitted. The read set for a client consists of the writes relevant for the read operations performed by a client. The write set for a client consists of the identifiers of writes performed by the client.

Monotonic read consistency implementation: when a client performs a read operation at a server, that server is handed the client's read set to check if all identified writes have taken place locally. If not, the other servers are contacted and it is brought up to dat before carrying out the read operation. Alternatively, the read operation is forwarded to an up-to-date server.

Monotonic write consistency implementation: analogous to monotonic read.

Read-your-writes consistency implementation: the server where the read operation is performed must have seen all the write operations in the client's write sets. Either fetch all writes before reading, or search for a server where the writes have already performed.

Writes-follows-reads consistency implementation: bring the selected server up to date with the write operations in the client's read set, later add the identifier of the write operation to the write set along with the identifiers in the read set (they are now relevant for the just-performed write operation)

7.6 Example: Caching and replication in the web

📗 read the book, this section is really long, caching dynamic content is not relevant

9 Security

📗 CH 9

- Section 9.1, except for Note 9.1
- Section 9.2, except for Note 9.3 and Kerberos (though worth taking a look)
- Section 9.3, except for Note 9.7. We did not discuss Denial of Service attacks in detail, though we mentioned them in our discussion on the security aspects of clock synchronization.
- Section 9.4
- Section 9.5, except for Note 9.10 and "Delegation"

9.1 Introduction to security

Security threats, policies, and mechanisms

Security is strongly related to the notion of dependability. Dependability includes availability, reliability, safety and maintainability. Confidentiality and Integrity should also be taken into account.

Confidentiality: Information is only disclosed to authorized parties.

Integrity: alterations to a system's assets can be made only in an authorized way. → Improper alterations should be detectable and recoverable.

Four types of security threats to consider:

- Interception: Unauthorized party has gained access to a service or data (e.g. communication between two parties overheard), or data being illegally copied (e.g. breaking into someone's directory in a file system).
- Interruption: Refers to data or services becoming unavailable, unusable, destroyed… e.g. DOS attacks
- Modification: Unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications.
- Fabrication: Additional data or activity are generated that would normally not exist.

Important security mechanisms:

- Encryption: Transforms data into something an attacker cannot understand. In addition, it allows us to check whether data have been modified.
- Authentication: Used to verify the claimed identity of an entity.
- Authorization: Checking whether a client is authorized to perform the action requested.
- Auditing: Auditing tools are used to trace which clients accessed what and in which way.

Design issues

Focus of control
Three approaches:

- concentrate directly on the protection of the data that is associated with the application (protection against invalid operations)
- concentrate on protection by specifying exactly which operations may be invoked and by whom
- concentrate directly on users by taking measures by which only specific people have access to an application, irrespective of the operations they want to carry out

Layering of security mechanisms
A system is either secure or not, but whether a client ***considers*** a system secure or not is a matter of trust. In which layer security mechanisms are placed depends on the trust a client has in how secure the services are in a particular layer. (page 506 for more details).
Distribution of security mechanisms: A trusted computing base (TCB) is the set of all security mechanisms in a system that are needed to enforce a security policy and thus need to be trusted. The TCB in a distributed system may include the local operating systems at various hosts. (Details on what to do if we do not trust something page 507ish)

Simplicity
Where to place security mechanisms in terms of simplicity. Example on page 508

Cryptography

Encryption and decryption are accomplished by using cryptographic methods parameterized by keys. The original form of the message is called ***plaintext,*** while the encrypted form is referred to as ***cyphertext***. An intruder that intercepts a message will see only unintelligible data (even so, they can draw conclusions based on the two communication partners). For modifying a message, the encryption would first have to be decrypted and encrypted again after modification. Third type of attack: Inserting an encrypted message.

- Symmetric cryptosystem: Same key is used to encrypt and decrypt. (”secret-key system / shared-key system”)
- Asymmetric cryptosystem: Different keys are used, but together form a unique pair. One is a private key, the other is a public key (”public-key systems”). If A wants to send a message to B, A uses B’s public key to encrypt the message, B decrypts it with its own private key. If B wants to ensure that the message came from A, A can keep its encryption key private, and B can decrypt the message with A’s public key.

Hash Functions have weak collision resistance: if you know an input m and its hash h(m), it is not possible to find an imput m’ that has the same hash, given m ≠ m’

Cryptographic hash functions have strong collision resistance, meaning that if you know the hash function H, you cannot generate two different inputs that have the same hash.

9.2 Secure channels

Two issues in making a distributed system secure: How to make communication between clients and servers secure, and once a server has accepted a request from a client, how can it find out whether that client is authorized to have a request carried out.

Authentication

Authentication and message integrity cannot do without each other. Assume A and B want to communicate. A sets up a channel and starts sending a message to B. Once the channel is set up, A and

B know they are talking to each other. Afterwards, secret-key cryptography is used (session key for as long as the channel exists).

Authentication based on a shared secret key
Challenge-response protocol: A and B challenge each other to a response that can only be correct if the other knows the shared secret key.

Authentication using a key distribution center
Shared key for authentication is leads to a scalability problem. Alternative: use a key distribution center (KDC). It shares a secret key with each of the hosts, but no pair of hosts have to share a secret key. If A and B want to talk, the KDC gives them both a key. (details page 516)

Authentication using public-key cryptography
If A and B have each others public key, authenticity is established and A or B generate a session key.

Message integrity and confidentiality

A secure channel should provide guarantees for message integrity and confidentiality in addition to authentication.

- Digital signatures: digitally sign the message in such a way that the signature is uniquely tied to its content and the signature can be verified. The validity of A’s signature holds as long as A’s private key remains secret.
- Session keys: The more often a key is used, the easier it becomes to reveal it. Protection against replay attacks by using a unique session key and a timestamp or sequence numbers.

Secure group communication

In distributed systems, connection channels between multiple parties often need to be set up.

- Confidential group communication: protect communication between a group of N users against eavesdropping. Solution: let all group members share the same secret key. All members need to be trusted to keep the key a secret. Alternative solution: separate shared key between each pair of group members. Public-key cryptosystem can improve matters. Each member has its own (public key, private key) pair.
- Secure replicated servers: If a client issues a request to a group of replicated servers it expects the response to be trustworthy. Solution: authenticate each server response. If the majority is authenticated, the client can trust the response (violates replication transparency).

9.3 Access control

A request from a client usually involves invoking an object. The request can be carried out only if the client has sufficient access rights for that invocation. Verifying access rights is referred to as access control, authorization is about granting access rights.

General issues in access control

- access control matrix: have each object maintain a list of the access rights of subjects that want to access the object or give each subject a list of capabilities it has for each object.

Firewalls

Cryptographic techniques and access control matrix only work if all communicating parties abide by the same rules. If they don’t, external access to any part of a distributed system needs to be controlled by a special kind of reference monitor (the firewall). Generally, there are two types of firewalls:

- packet-filtering gateway: operates as a router and makes decisions on whether or not to pass a network packet based on its source and destination (only inspects the header)
- application-level gateway: inspects the content as well
  - proxy gateway: only messages that meet certain criteria are passed

Secure mobile code

Once a malicious program has settled itself in a computer, it can easily corrupt its host. Protecting a host against downloaded malicious programs is not always easy and it is not about avoiding downloads, rather supporting mobile code that we can allow access to local resources in a flexible and fully-controlled manner. Approaches:

- construct a sandbox: a downloaded program is executed in such a way that each of its instructions can be fully controlled. Execution is halted if it does something forbidden. Approach: check the executable code and insert additional instructions for situations that can be checked only at runtime. (JVM example page 535).

Denial of service

Distributed denial of service: huge collection of processes jointly attempt to bring down a networked service.

- aimed at bandwidth depletion: send many messages to a single machine
- aimed at resource depletion

Protection: monitor network traffic.

9.4 Secure naming

when a client retrieves an object based on some name, how does it know that it got back the correct object. Three issues:

- validity: is the object a complete, unaltered copy of what was stored at the server?
- provenance: can the server that returned the object be trusted as a genuine supplier? Maybe the client is only given a cached version of the object
- Relevance: is what was returned relevant considering what was asked?

Solution: securely bind the name of an object to its content through hashing (self-certifying name). Problem: not human friendly.

Solution: have a self-certifying name and a human friendly label.

9.5 Security management

Key management

Establishing and distributing keys is not a trivial matter. Keys cannot distributed via an unsecured channel, and mechanisms for revoking keys are needed.

- key establishment: Diffie-Hellman key exchange: A and B want to establish a shared secret key. The need to agree on two large numbers n and g (their mathematical properties are not discussed). n and g can be made public. A picks a large random number x it keeps secret. B does the same (y). A sends g^x mod n to B, along with n and g. B subsequently calculates (g^x mod n)^y, which equals g^xy mod n. In addition B sends g^y mod n. A can then compute (g^y mod n)^x = g^xy mod n. Now both have the shared secret key g^xy mod n. Neither need to make their private number known to the other.

- key distribution: If no keys are available to A and B to set up a secure channel, the key needs to distributed out-of-band (aka A and B need to use another communication channel). Public-key distribution takes place by means of public-key certificates (the public key with a string identifying the entity the key is associated to). The public key and identifier have together been signed by a certification authority and the certificate is signed as well.

Secure group management

KDCs and CAs need to be trusted and they have to be available. Solution for availability is replication, but at the cost of vulnerability to security attacks. (Details page 546)

Authorization management

Managing security is also concerned with managing access rights. In nondistributed systems, when a new user is added to the system, it is given initial rights (specified in advance by the system administrators). In distributed systems, if this approach were to be followed, each user would need an account on each machine. Solution: A single account on a central server.

- Capabilities and attribute certificates: Capabilities are a much better approach. A capability is an unforgeable data structure for a specific resource, specifying exactly the access rights for the holder of the capability with respect to that resource. (Implementation Example page 548)
  Attribute certificate: A generalization of capabilities. They are used to list certain (attribute,value) pairs that apply to an identified entity. They can be used to list the access rights that the holder of a certificate has with respect to the identified resource. They are handed out by attribute certification authorities.

Applications & Technology Trends

📗 this section was not in the book, the below information is taken from the lecture slides

Hype Cycle:

- Innovation trigger
- Peak of inflated expectation
- trough of disillusionment
- slope of enlightenment
- plateau of productivity

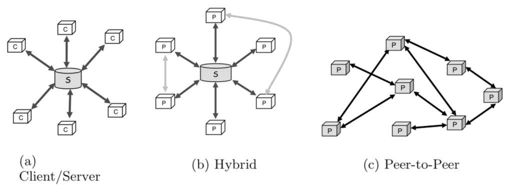
IoT: physical objects seamlessly integrated into the information network. Physical objects become “smart objects”

IoS (Internet of Services): software services are provided through the internet. Foundation for Cloud Computing and Edge Computing

IoS = global SOA?? (service oriented architecture)

SOA: originally mainly a concept to organize IT software architectures in an organization

Peer-to-peer



Key characteristics of a P2P system:

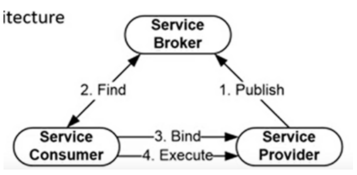
- Equality: all peers are equal
- Autonomy: no central control
- Decentralization: no centralized services
- Self-organization: no coordination from outside
- Shared resources: peers may use resources provided by other peers

Reasons for Application of P2P:

- costs: computing/storage can be outsources
- high extensibility: easy to add further resources
- fault tolerance: if one peer fails, the system will still work
- resilience to lawsuits (like those "legal" streaming sites)

SOA (Service Oriented Architectures

Roles in a Service oriented Architecture



Overview

- IT architecture made up from single services (aka self-contained software components with a distinct functionality)
- Complex applications arise from the coupling of single servers (e.g. service based workflows, mashups)
- It is however also possible to invoke single services

Cloud Computing

✨don't buy the cow if you want a bottle of milk✨

NIST: 3 Service Models: IaaS, PaaS, SaaS

NIST: 4 Deployment Models:

- Private Cloud: Operated solely for one single organization
- Community Cloud: Shared by several organizations
- Public Cloud: Open to general public, owned by an organization selling cloud services
- Hybrid Cloud: Composition of two or more Cloud deployment models

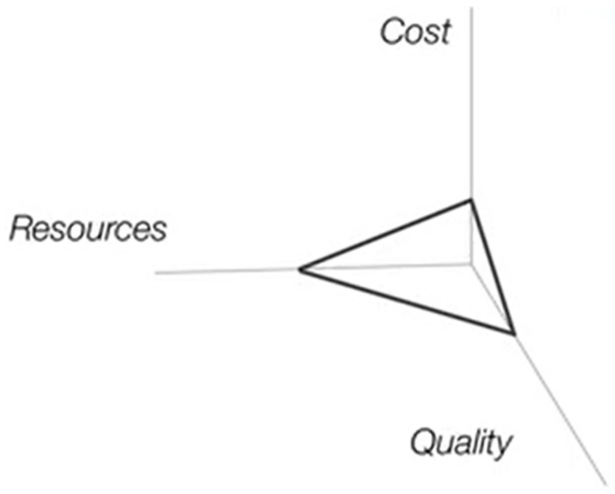
Elastic (Edge+Cloud) Computing

Intelligence shifted towards the edge of the network

Used if something is very sensitive to the latency of the network

Cloud-centric perspective:

- Assumptions: Cloud provides core services, edge provides local proxies for the cloud (offloading parts of the cloud's workload)
- Edge Computers:
    - play supportive role for the IoT services and applications
    - Cloud computing-based IoT solutions use cloud servers for various purposes including massive computation, data storage, communication between IoT systems, and security/privacy
- Missing: In the network architecture, the cloud is also located at the network edge, not surrounded by the edge. Computers at the edge do not always have to depend on the cloud, they can operate autonomously and collaborate with one another directly without the help of the cloud



Elasticity is 3-dimensional