Distributed computing

WS 19/20

① Distributed Computing = DC

Definition 1: Wikipedia
- field of CS that studies distributed systems
- components interact to achieve a common goal
- use of distributed systems to solve computational problems
- overlaps parallel computing

Definition 2: Prof
- systems (hard- and software) that work together at geographical distance acting as a problem solving environment to benefit the user

Benefits:
- resources can be used independent of its location
  ↳ e.g. data can be produced in one, needed in another
- more cost efficient (e.g. some hardware not needed everywhere)
- higher reliability, availability → no single point of failure
- easier to expand

Architectures
- Client/Server: Client contacts server for work, pushes results back
- Peer-to-peer: no specialized machines, all responsibility distributed
  peers are both (server and client)

Metacomputing
- logical integration of several independent supercomputers coupled via high-performance connection
- aggregation of memory capacity and computing power
→ - solve problems too big for single supercomputer ("grand challenge problems")
  - solve faster than one supercomp.

- Coupled Simulations: run each sim. on most optimal platform

Grid Concept

- allows virtual organizations: scientific collaborations share resources on high scale and work together
- Idea: providing scalable, secure, high-performance mechanisms e.g. for discovering resources, using remote resources

Checklist

- coordinates resources that are not controlled centralized
- uses standard, open, general-purpose protocols and interfaces
  - Authentication, access, ...
- non-trivial qualities of service
  - response time, throughput, availability, security, ...
  - utility of the grid greater than sum of its parts

Example: CERN

Worldwide LHC computing Grid (WLCG)

- Storing, analyzing LHC data
- model: multi-tiered, hierarchical

Tier-0: CERN
- Data recording
- Initial data reconstruction
- Data distribution

Tier-1: 11 centres
- Permanent storage
- Re-processing
- Analysis

Tier-2: ~140 centres
- Simulation
- End-user analysis

Big Data

- 3 Vs: Volume, Velocity, Variety
- 5 Vs: + = Value, Veracity (Richtigkeit)
- Important: privacy, preservation and sustainability

Data Analysis

- goal: discovering useful information
- methods: inspecting, cleansing, transforming, modelling data

MapReduce (Google)

- process large scale data
  - automatic parallelization
  - scalable with TB/PB of data on thousands of machines

Cloud Computing

Def. 1: NIST
model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing ressources that can be rapidly provisioned and released with minimal management effort

Def. 2:
- illusion of infinite ressources on-demand
  elimination of upfront commitment by Cloud users
- ability to pay for use of resources on short-term basis as needed

⊕
- accessing resources (computer, storage, applications)
- available on-demand
- customized environment
- self-service without administrator
- cost-effective

Cloud vs. grid:

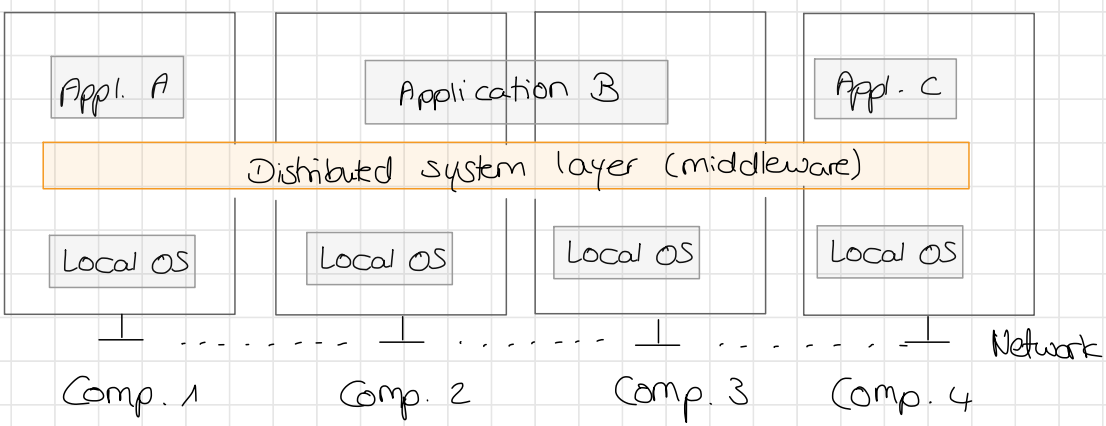
- Cloud:
  - centralized control
  - proprietary protocols and interfaces
  - typically "simple services"

Distributed Systems

- collection of autonomous computers
  - → connected through network and distribution middleware
  - → coordinate activities, share resources
  - ~→ user sees system as single, integrated computing facility

Characteristics

- autonomy of individual computers
- single-system view
  - → hides differences between individual computers and their interaction from user (→middleware)
- interaction user <-> application of distributed system: uniform and consistent
- scalable, extensible, fault tolerant



Centralized system	vs.	Distributed system
one component with non-autonomous parts	⟷	multiple autonomous components
components shared by users all the time	⟷	components not shared by all users
all resources accessible	⟷	resources may not be accesible
software runs in single process	⟷	software runs concurrently on different processors
single point of control/failure	⟷	multiple points of control/failure

Goals of distributed systems

1) Connecting users and resources
   - provide access to remote resources, support collaboration, guarantee security

2) Transparency
   - Hide the fact that resources are physically distributed

3) Openness
   - services according to standard rules

4) Scalability
   - allow growth of # users / resources, geographic extend, ...

Connecting Users and Resources

- Sharing and accessing remote resources
  → Hardware (memory, comp, ...), Software (data, licenses, ...) and services
- billing and accounting mechanisms

⊕
- economical benefits
  ↳ sharing expensive resources (e.g. telescope)
- simplifies collaboration and information exchange

⊖
- requires (difficult) security mechanisms
  ↳ access only for trusted users
  ↳ user only accepts secure resources

Transparency

Access:
- hide differences in data representation and how resource is accessed, e.g. byte order (little/big endian)

Location:
- hide location of resources
  ↳ logical names (e.g. URL) instead of / in addition to physical

Concurrency
- hide concurrent use of resources by several competitive users
  ↳ concurrent access has to ensure consistency!

Failure
- hide failure and recovery of resources
  ↳ need to differ between dead and slow resources

Openness

- offer functionality as specified in standard rules (how to access)
  - Protocols: specify format, contents, order, meaning of messages
  - Interfaces: specify syntax to access functionality
    ↳ often written in Interface Definition Language
  - Semantics: hard to formalize → often informal description (comments)

- **Interoperability:** 2 implementations co-exists, work together by relying on each others interface specifications

- **Portability:** implementation runs on different systems implementing the same interface

- **Flexibility:** system is easily configurable out of different components from different developers

Scalability

- Size: # of users / resources
  ↳ main bottleneck: centralization (e.g. single server for all users)

- Geographical extend:
  ↳ main bottleneck: latency of wide-area communication
  Problems:
  - synchronous communication (good in LAN, bad with WAN)
  - unreliable communication
  - no efficient broadcast possible
  - need for centralized components, services

- Number of independent administrative domains
  - no preexisting trust relationship btw different admin. domains
    ↳ e.g. sharing info between organizations?
  - requires mutual protection through restriction enforcement
    → increases system complexity
    Alternative: mechanism to establish trust relationship (certificates)

Problems

- developing distributed systems very complex

⚠ False assumptions:
- network is reliable
- network is secure
- network is homogenous
- topology does not change
- no latency
- bandwith is infinite
- no transport costs
- only one administrator

Types

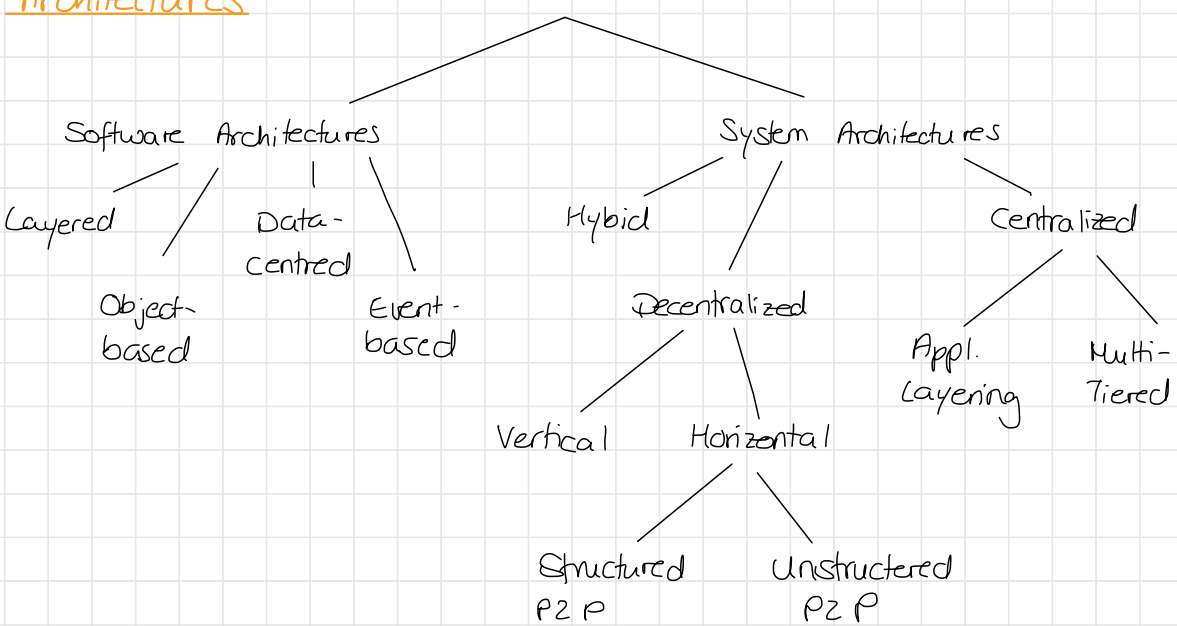
- Cluster Computing:
  Set of compute nodes connected with high-speed network
  → parallel high-performance computer

- Grid computing:
  resources distributed accross geogr. distance
  accessible through middleware and web services

- Distributed information system:
  transaction processing systems with databases, Enterprise Application, Integration, ACID operations
  → very stable

- Distributed pervasive systems:
  mobil devices, embedded systems, electronic health care system, sensor networks

Architectures

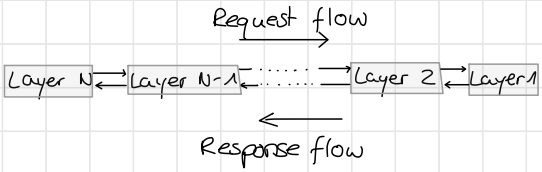


Software Architectures

Organization / interaction of various software components

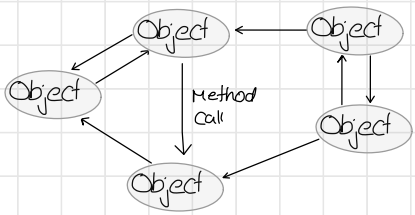
Layered

- Several layers stacked
- often used in network community



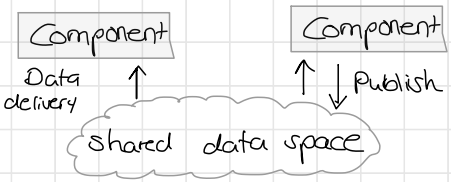
Object-based

- each component = object
- connected through RPCs
- ≈ client-server architecture



Data-centred

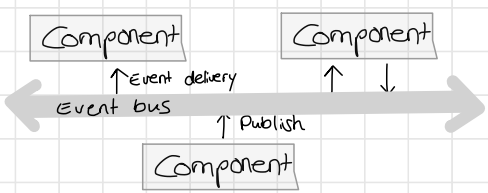
- components communicate through shared data space



Event-based

- components communicate through event-propagation
- publish / subscribe system

⊕ components loosely coupled



System Architectures

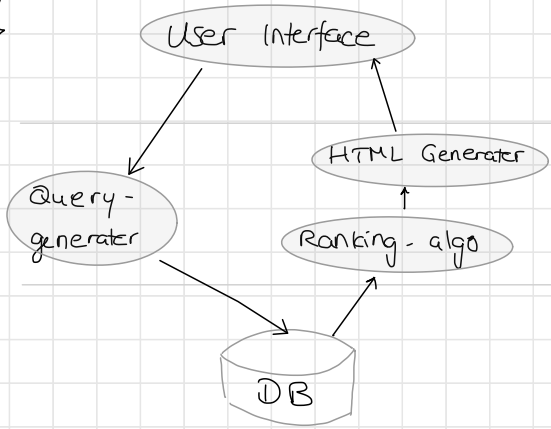
final instantiation of software architecture on real machines

Centralized

- basic client-server model
- communication: connectionless protocol
  - ↳ difficult: making protocol resistant to occasional transmission failure
    - ! re-send only for idempotent messages harmless
- Alternative: reliable connection-oriented protocol ⊖ costly
  - ↳ e.g. internet appl. based on TCP/IP

Application Layering

- 3 level:
  1) user-interface-level:
     - clients implement this level
     - simple program (GUIs, ...
  2) Processing-level:
     - contains application
  3) Data-level:
     - where data is located/processed
     - data often persistend
     - typically implemented on server side as database

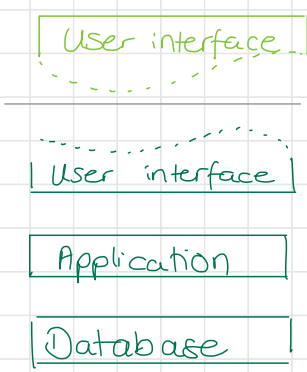


Multi-tiered

- distributing client/server application across several machines
  ~> distribute programs in application layers

Possibilities: (Client Server)

a)

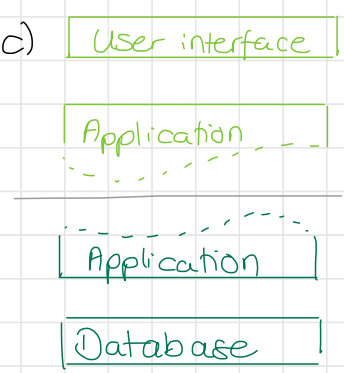


- only terminal dependent part of UI is on client machine

b)

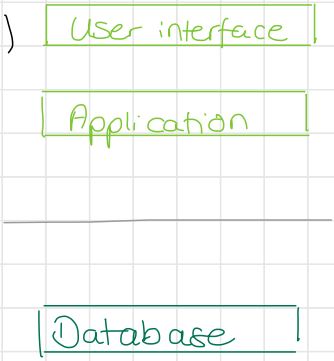
- entire user-interface on client side
- UI: only processing that's necessary for UI

c)



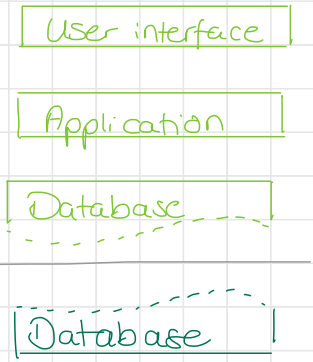
- move part of the application to the frontend

d)



- data ops running on server
- app mostly on client

e)



- clients lokal disk contains part of the data

used where client is PC / work station, connected to database / distributed file system

⇝ A-C desired / trending : easy client, complex server
- easier to handle
- fat clients more error prone, more dependent on underlying platform
- often: server seperated into 3-layers (user, app, db) using multiple servers

Dezentralized

Vertical Distribution

- splitting functions logically and physically across machines
  ↳ each machine for specific group of functions

program demo:

func 1, func 2 } → machine 1
func 3 } → machine 2
func 4 } ...
func n

Horizontal Distribution

- split clients, server physically into logically equivalent parts
- each part operates on own share of complete data set
  ↳ load balancing
- each component equal: most interaction is symmetric
- each part is client and server at the same time ⇝ servent
- communication through overlay network
  ↳ participating peers = nodes
  edge btw. 2 nodes if they know each other
  → classification based on how the nodes are linked:
  Structured / unstructured P2P overlay network

Structured P2P

- structuring based on deterministic algo
  ↳ e.g. distributed hash table
- data items, nodes get random key as identifier
- difficulty: implement efficient, deterministic mapping to map data items to nodes
  (⇝ lookup data item results in node address)

Unstructured P2P

- overlay network = random graph
  ⇝ constructed by randomized algorithm ← = 'partial view'
- each node contains list of neighbors, neighbor = random node from current set of nodes
- place data items randomly on nodes
- query for data needs to be flooded through network
  ↳ find as many peers that share data as possible
  ↳ popular content easily found
  ↳ rare - data maybe not found at all ⊖
  ↳ special nodes (Superpeers) maintain index of data items

⊖ flooding causes lots of traffic → poor search efficiency
⊕ easy construction
easy peer insertion

Hybrid: Bit Torrent

- collaborative distributed - system
- Joining:
  - Step 1: node joins using client-server-scheme
  - Step 2: after joining: node uses fully decentralized scheme for collaboration

Example: BitTorrent

- P2P file-sharing system
- files distributed as chunks across network
- download: get chunks from other users, assemble to complete file

Download in detail:

1) access global directory of website
   - ~> directory contains references to .torrent files
   - .torrent contains: · info needed to download file
     · tracker
   - tracker: tracks active nodes (currently downloading) that own chunks of requested file
   - one tracker (= server) per file
     - ⊖ system bottleneck
2) now: info containing nodes identified
   - ↳ download chunks of the files

Self-management

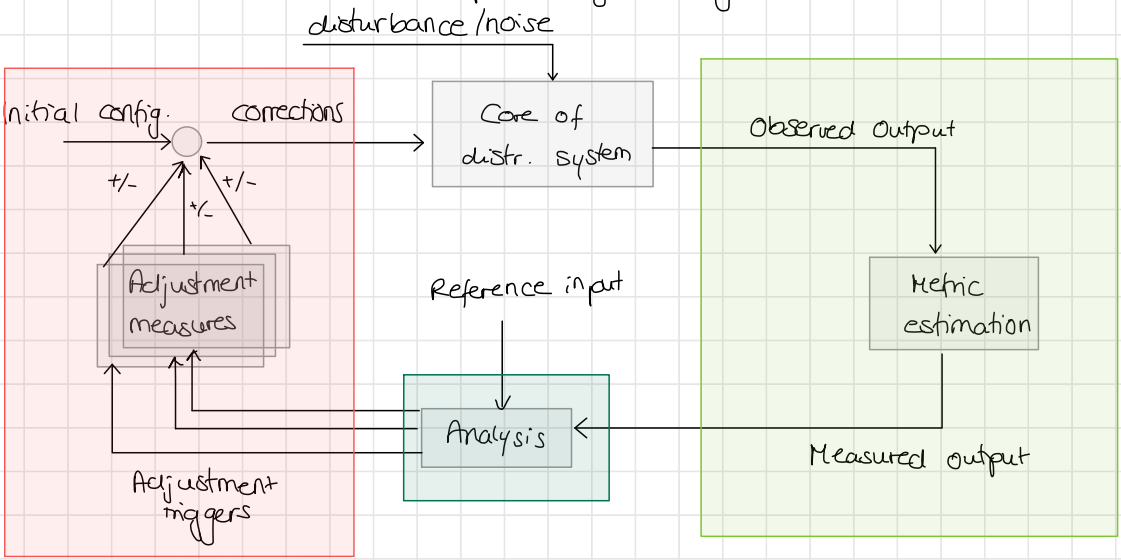
Requirement: provide general solution towards shielding undesirable features to support as many applications as possible

↳ Distributed systems should be adaptive
↳ requires strong interplay between system, software architectures

- Organize DS as high-level feedback control system
  → automatic adaptation to changes

Feedback control system

- ≥ 1 feedback control loop: (logical organization)



influence system behaviour by...
... placing replicas
... changing scheduling priorities
... switching services
... moving data
... redirecting requests
...

Analysis
- heart of the system
- analyse measurements, compare to references
- contains algo that decides on possible adaptations

↳ needs to be aware of these mechanisms and their effect

Monitoring
- measure system aspects

manual management: analysis replaced by human administrator
↳ supported by monitoring tools

Fault Tolerance

- DS contain many components, connections
  ↳ failures, faults occur everywhere at any time

Fault Tolerance Definition

= property that enables system to continue operating properly in the event of the failure of / faults within some components

→ important aspects: dependability

- Availability: system is ready to use immediately
- Reliability: system can run continuously without failure
- Safety: if system fails no catastrophy will happen
- Maintainability: when system fails it can be repaired easily + quickly (without user noticing failure)

Failure

System fails if it cannot meet its promises

- failure induced by existence of errors
- cause of an error = fault

fault → error → failure

Fault Types

- transient: appears once, then disappears
- intermittent: occurs, vanishes, reappears; follows no real pattern
- permanent: requires replacement/repair of faulty component

Failure Types

- Crash: server halts, works correctly until it halts
- Omission: server fails to respond to incoming requests
  - receive omission: server fails to receive incoming requests
  - send omission: server fails to send ack messages
- Timing: server's response lies outside specified time interval
- Response: server responds incorrect
  - Value failure: response value is wrong
  - State transition failure: server deviates from correct control flow
- Arbitrary: server produces arbitrary responses at arbitrary times

Failure Masking: Redundancy

- hide failure occurences from other processes using redundancy
  - Information Redundancy:
    add extra bits for error detection/recovery
  - Time Redundancy:
    perform operation, if needed: perform it again
  - Physical Redundancy:
    add extra hardware and/or software to the system

Reliable Client-Server Communication

Potential communication failures:

- Client cannot locate server -> request cannot be sent
- client's request to server is lost -> no response to waiting client
- server crashes after receiving the request -> request is acked, but left undone
- server's reply is lost -> server completes, client never gets reply
- Client crashes after sending -> servers reply not expected by restarted client

Lost replies difficult to handle

- reason?
  -> server dead (slow? reply went missing?
- repeating request not always solution (e.g. bank transfer)

Server crashes handle by implementation philosophies:

- at least once semantics:
  guarantee, that the communication occured at least once, maybe more
- at most once semantics:
  guarantee, that the communication occured at most once, maybe never
- no semantics:
  no guarantee for anything, client/server take their chances
- exactly once semantics: proved challenging

Client crashes

- old reply = orphan
- orphan arrives:
  - Extermination: kill orphan
  - Reincarnation: client sessions have epoch associated → orphans easy to spot
  - gentle reincarnation: new epoch: try to locate requests owner
    ↳ otherwise kill orphan
  - Expiration: if communication not completed within standard time: assume it expired

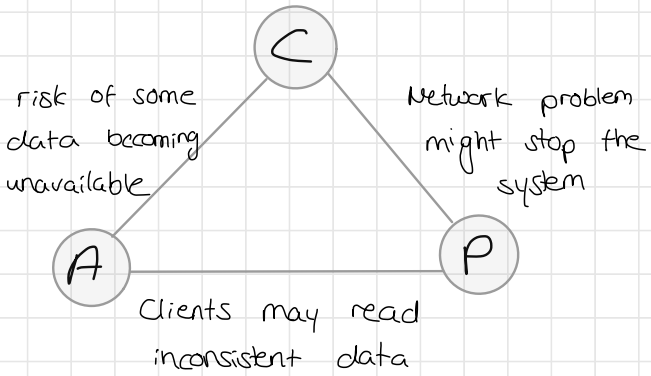
In practise: above methods not good for handling orphans

CAP - Theorem

by E. Brewer transformed to DS

- Consistency = all components are in the same state
- Availability = all components reply to requests as expected
- Partition tolerance = system as whole works as expected even with single components failing

→ DS can never guarantee more than two



- Domain Name System (DNS):
→ A, P
- robust
- low consistency → changes distributed slowly (up to hours)

- Relational Database Systems
→ C, A
  - partitioning tolerated: transaction journals

- Banking Applications:
→ C, P
  - C most important
  - at cost of communication failures

DISTRIBUTED COMPUTING

Definition 1: Wikipedia
- field of CS that studies distributed systems
- components interact to achieve a common goal
- use of distributed systems to solve computational problems
- overlaps parallel computing

Definition 2: Prof
- systems (hard- and software) that work together at geographical distance acting as a problem solving environment to benefit the user

Benefits:
- resources can be used independent of its location
  - ↪ e.g. data can be produced in one, needed in another
- more cost efficient (e.g. some hardware not needed everywhere)
- higher reliability, availability → no single point of failure
- easier to expand

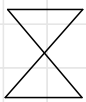
Grid Concept
- allows virtual organizations: scientific collaborations share resources on high scale and work together
- Idea: providing scalable, secure, high-performance mechanisms e.g. for discovering resources, using remote resources

Checklist
- coordinates resources that are not controlled centralized
- uses standard, open, general-purpose protocols and interfaces
  - Authentication, access, ...
- non-trivial qualities of service
  - response time, throughput, availability, security, ...
  - utility of the grid greater than sum of its parts

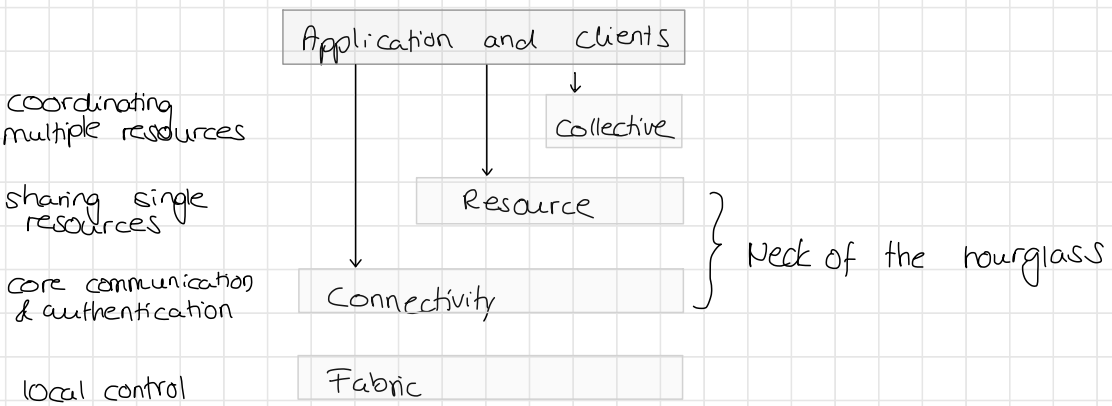
Grid Architecture

Anatomy

defines grid middleware as protocol stack
↳ hourglass model: small set of core abstractions allows mapping of diverse applications onto diverse set of resources



Grid Protocol Architecture



Fabric Layer

- Provides resources to be shared
  computers (clusters, ...), storage, network, sensors, databases, ...

- Offers local, resource specific operations on specific resources
  ↳ result of operation sharing on higher level

Connectivity Layer

- Core communication protocols (transport, routing, naming)
  - ⇒ exchange data between fabric-layer resources

- Authentication protocols
  - Secure mechanisms to verify user identities / resource identities
  - access to multiple resources via single sign on
  - allow program to act on user's behalf using delegation
  - integration with local security solutions
  - user-based trust relationship:
    user is allowed to use sites A and B: using A, B together without requiring interaction of A's and B's security admins

Resource Layer

- sharing of single resources
- defines standard protocols for secure negotiation, initiation, monitoring, control, accounting, payment of sharing operations on individual resources
- implementations call fabric layer functions

Information protocols:
- structure, state of resource
- e.g. configuration, usage policy, cost

Management protocols:
- negotiate access to a shared resource, specifying
  - resource requirements
  - operations to be performed
- accounting, payment
- monitoring, controlling operations

Collective Layer

Coordinates multiple resources

- Directory services (discover existence, location, properties of a resource)
- Co-allocation, scheduling, brokering services
  - allocate multiple resources for simultaneous usage
  - schedule tasks on these resources
  - negotiate access to multiple resources
- Monitoring and diagnostics services
  - look for failure, adversarial attack, overload, ...
- Data replication
  - maximize data-access performance with respect to response time, reliability, cost

Physiology

Service-oriented architecture based on web services technology

↳ Open Grid Services Architecture (OGSA)

- standardized framework
- all resources represented by a service
- all components of the environment are virtual

Aligns grid technology with web services

- extends notion of web service to address stateful behaviour
  ↳ distinction between service and stateful resource

- Includes set of basic interfaces to access state of a resource

- Defines infrastructure for integrating, managing services within virt. organ.
  - execution management services
  - data, resource management services
  - security, self-management services
  - information services

Virtualization and Cloud

Expectations:

- Reduction of costs for operaters and users (pay-per-use)
- Reduced complexity for users (higher usability, shorter time to solution)
- easier resource access (web services)
- flexibility, user satisfaction (virtualization)
- "unlimited" resources (virtualization)
- increased freedom for users, e.g. free choice of OS
- ...

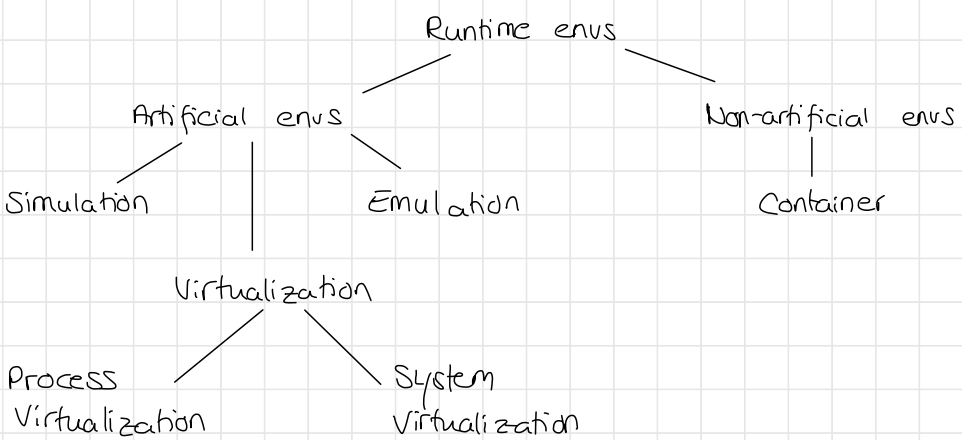
Virtualization

Can be done in Hardware or Software

! Simulation ≠ Emulation ≠ Virtualization

! Containers ≠ Virtualization

Hierarchy of runtime environments



- Simulation: models aspects, limits of target env, does not necessarily implement features or behaves exactly the same

- Emulation : reproduces target env as close as possible with all features

- Virtualization : creates env with components behaving like expected, not necessarily implemented exactly as original component

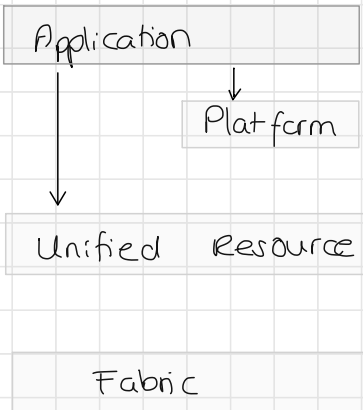
Cloud

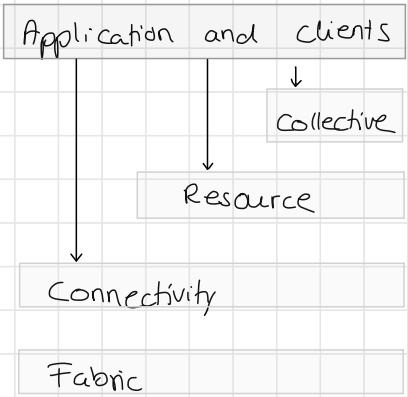
Def. 1 : NIST model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing ressources that can be rapidly provisioned and released with minimal management effort

Def. 2 :
- illusion of infinite resources on-demand
  elimination of upfront commitment by Cloud users
- ability to pay for use of resources on short-term basis as needed

⊕
- accessing resources (computer, storage, applications)
- available on-demand
- customized environment
- self-service without administrator
- cost-effective (shared costs / pay-per-use)

Cloud vs. Grid





Leasing desired comp. platform via network/web enabled services	Objective	sharing of resources
one/few data centres	Infrastructure	many sites, geogr. distributed
yes	central control	no
virtualization, web 2.0, web services	enabling technology	HPC, middleware, OGSA, WSRF
Propietary	Middleware	standards-based
easy to use/deploy; non-complex UI, browser-based	User interface	various types: portals, APIs clients, CLI, ...
academia, industry	Used by	academia
pay-per-use	Business model	use for free
commercial	Funding	public

NIST: Characteristics

1) On-demand self-service: Customers can acquire computing capabilities as needed (automatically, without human interaction)

2) Broad network access: Capabilities available over network, accessed through standard mechanisms, possible from various platforms

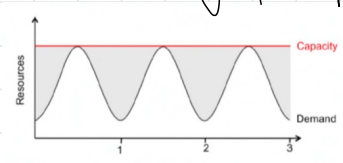
3) Resource pooling: providers computing services are pooled to serve multiple customers

4) Rapid Elasticity: can be rapidly and elastically provisioned to quickly scale out and rapidly released to quickly scale in

5) Measured service: resource use automatically controled, optimized usage can be monitored, controlled, reported

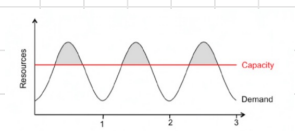
Elasticity

1) Provisioning for peak load → waste of resources



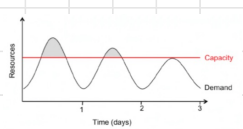
2) Underprovisioning

a)



- Some users not served

b)



- users experienced poor service → permanent loss

NIST: Deployment Models

Private Cloud

- solely for one organization
- owned, managed, operated by that orga / third party
- on or off premises
- apps, data are mission critical
- use case: strict governance, security, data protection required
  usually IT infrastructure already in place

Public Cloud

- available to general public
- owned, manged, operated by business / academic / government orga
- on premises (of the cloud provider), off premises of the user
- user, vendor belong to different orgas
  ↳ vendor typically runs business with cloud
- risk of vendor lock-in
- check for data security, privacy, protection, IPR
- standardized workload used by lots of people
- typically also offers SaaS

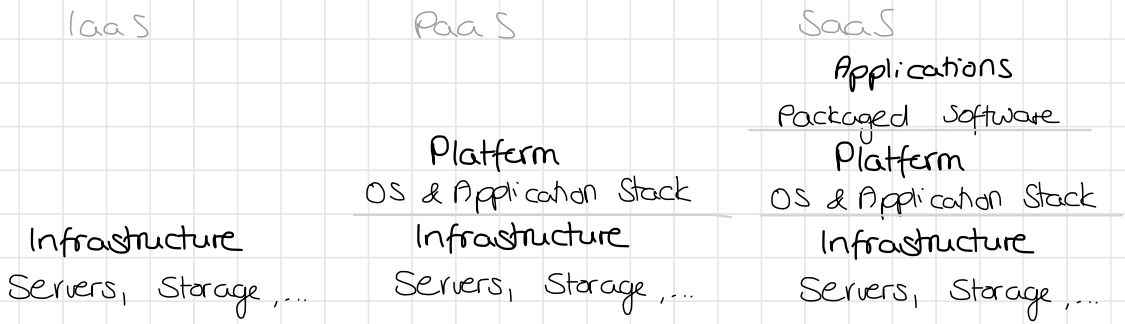
Community Cloud (rarely used)

- supports specific community with shared concerns
- owned, managed, operated by ≥1 orgas of this comm. / third party
- on or off premises

Hybrid Cloud

- comp. of ≥2 of above (e.g. private + public)
- parts remain unique entities, enable data/app portability
  ↳ "best of both worlds"
- different needs for different types of users / data

NIST: service models



Infrastructure as a service IaaS

- providing fundamental computing resources (processing, storage, ...)
- consumers can deploy/run arbitrary software (OS, apps, ...)
- underlaying cloud infrastructure is transparent to users
  - ↳ they don't manage/control cloud
- eg Amazon EC2, Flexiscale

Platform as a Service PaaS

- customers can develop their apps on cloud infrastructure
- programming tools, envs supported by provider
- users don't manage/control underlying cloud
- apps presented as web services
- eg Microsoft Azure, Google App Engine

Software as a service SaaS

- users can use provider's software on provider's infrastructure
- licensed software
- applications of public interest
- accesible through thin client interface, e.g. webbrowser
- related data stored remotely
- Customes don't manage/control underlying infrastructure
- e.g. Google docs, web-based emails

Everything as a service XaaS

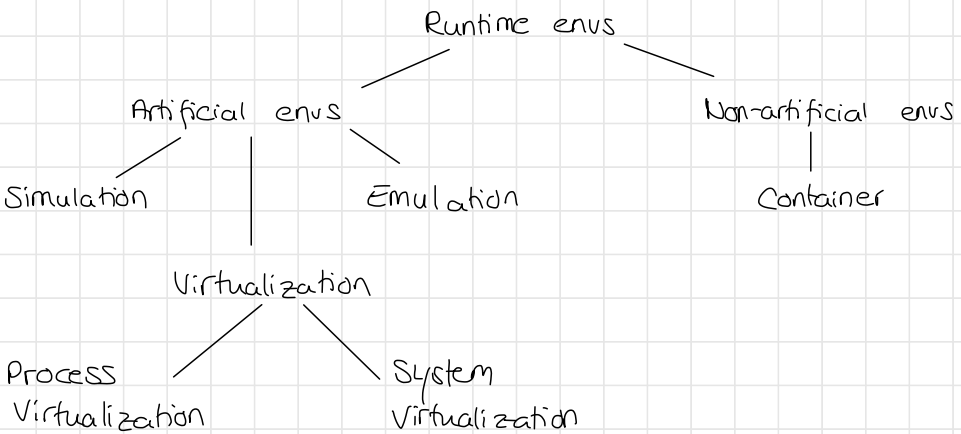
- Storage as a service
  - pay-per-stored volume, pay-per-transferred size
  - e.g. Dropbox
- Grid as a service
- HPC as a service

Function as a Service FaaS

- Scalability: Reaction to large # Requests
- Environment: Running the code on a platform
- Virtualization: Capsulation of running code
- event-driven, scalable, fast code deployment, payment per invocation

Virtualization

Hierarchy of runtime environments



Concepts

- virtual machine needs all components a real machine consists of:
  - CPU
  - memory
  - storage
  - peripheral devices

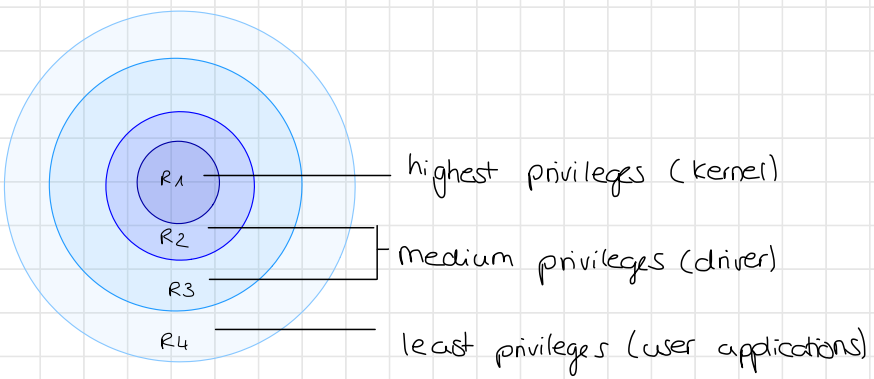
} must behave like a physical component

Hypervisor = Virtual Machine Monitor VMM
- component that enables virtualization
  ↳ provides, manages virtual components

Type-1-Hypervisor
- executed directly on hardware
- e.g. VMWare ESX

Type-2-Hypervisor
- Software running on top of OS
- e.g. VirtualBox, VMWare workstation

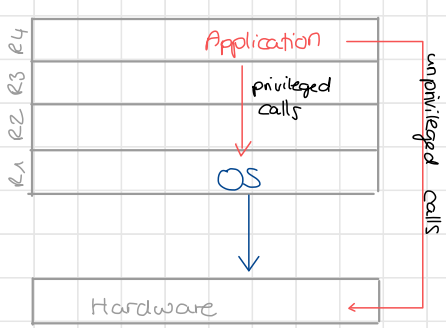
CPU Virtualization
- Protected mode: several protection domains = rings



- applications running on top of OS → most instructions can be executed directly on hardware
- no privileged calls (syscalls) in R4 → OS API calls required

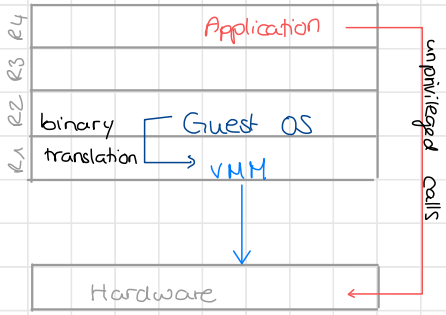
↳ Problem: virtualization requires OS to operate in less privileged rings

→ 3 solutions



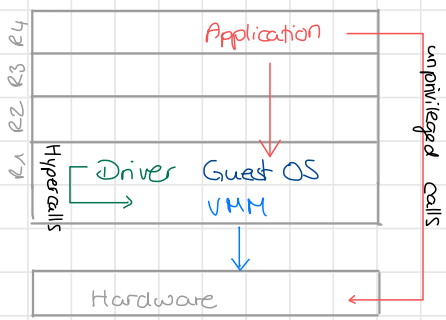
a) Full virtualization

- VMM is running in R1 privileges
- guest OS running in R2
- calls requiring high privileges translated "on the fly" (binary translation)
- high performance through optimization and caching
- majority of user applications unprivileged, can be executed directly
- Full decoupling: guest not aware it is running in virtual environment



b) Para-virtualization

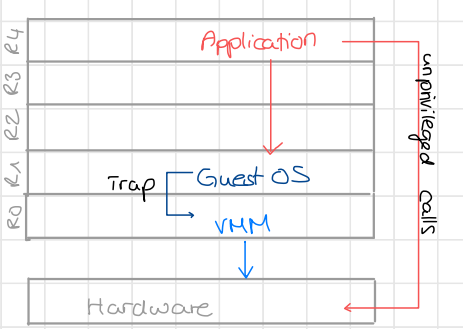
- Guest OS modified and non-virtualizable instructions replaced with hypercalls
- Hypercalls = instructions communicating directly with VMM
  ↳ implemented through drivers
- Hypercalls for:
  - memory management
  - interrupt handling
  - time keeping
- Guest OS is aware of it being virtualized



⊖ Portability, compatibility issues

c) Hardware-assisted virtualization

- CPU provides new execution mode: ring 0
- GuestOS in R1, privileged calls are intercepted using traps
  ↳ forwarded to VMM
- no translation / modification needed
- HW support necessary



Memory virtualization

- modern PCs have virt. mem. already
  - avoid mem. fragmentation
  - enable mem. protection, different mem. hierarchies

Memory virtualization

- each program has own virtual mem. address space
- OS operates on physical mem. space
- translation through lookup tables
  - ↪ slow, large mem. footprint
    Solution: paging
- hardware for mem. man.: MMU = memory management unit
- fast cache for commonly used addresses: Translation lookaside buffer

Process

1) VMM assigns each virtual machine a virtual mem. range
2) VM treats it as its own physical mem.
3) Guest OS provides virtual mem. space to running processes

→ Two translations:
  a) program's virtual adr. to VM's adr. space
  b) VM's adr. space to physical adr. space

→ Avoid this overhead: virtualized MMU uses directly hardware MMU

Storage virtualization

- VMM provides standardized interfaces
- VMM can use different storage types and make them accesible to VM
- VMM manages simultaneous access and isolates different VM domains

- Common: disk images = files → accessible to VM as disk drives
  ↪ I/O done to file instead of physical HW

Device virtualization

≈ Storage virtualization

- devices represented as commonly used devices
- emulation / abstraction of real hardware
- VMM manages I/O access
- all devices can be made available (network devices, sound cards, ...)

Examples

VMware

- full virtualization
- mechanisms for power management, resource scheduling

KVM = Kernel-based Virtualization Machine

- Linux as hypervisor
- standard linux kernel gets virtualization capabilities
- VM = regular linux process
- VMs scheduled by Linux scheduler
- new process mode:
  - guest mode: execute non-I/O guest code
  - kernel mode: handles exit from guest mode (I/O (special instructions)
  - user mode: perform I/O on behalf of guest

Xen

- Paravirtualization
- HW components not fully emulated
  ↪ I/O done by first guest kernel, further guests use this as well

Virtualization Benefits ⊕

- On-demand OS, resource customization
- Performance isolation → VMs isolated from each other
- Security → attacks only compromise one VM
- Availability → VMs easily migrated
- Easy management → each appl. domain manages its own env
- legacy support
- access "root" privilege

Virtual Appliance

- VM with pre-installed software → OS + software and env
- can be accessed via network interface once started

⊕ users don't need to maintain, deploy, install, configure software
SaaS for providers

Container

- running instance of a container image
- Container image = file containing all code, runtime, system tools, libraries a software needs to run
- container engines: docker, rocket, ..

- running a container creates a set of namespaces
  - ↳ provide a layer of isolation
  - ↳ each container aspect runs in separate namespace, access is limited to that namespace

namespaces (docker on Linux)

- pid : process isolation (process ID)
- net : managing network interfaces
- ipc : managing access to IPC resources (inter process comm.)
- mnt : managing filesystem mount points
- uts : isolating kernel and version identifiers (unix timesharing system)

Container vs VM

- both provide resources, process isolation
- both can be managed from outside

Differences:
- container provides no own OS → relies on host OS
- no virtual layer → no translation or supervision of instructions by a hypervisor
- no emulation of non-existing devices possible (requires additional software)

(+)
- more lightweight than VM
- higher execution speed
- no HW features required
- easy to setup, manage

(−)
- bound to host OS
- no HW emulation (no support of "legacy envs")
- less secure

Cloud Computing

- 5 characteristics:
  - on-demand self-service
  - broad network access
  - resource pooling
  - rapid elasticity
  - measured service

} enabled by virtualization:
- hypervisors controllable through APIs
- VMs can have network devices attached
- hypervisor can run multiple VMs on same HW
- VMs can be duplicated / moved from host to host
- hypervisors can monitor resource usage from "outside"

-> virtualization enables high flexibility regarding infrastructure

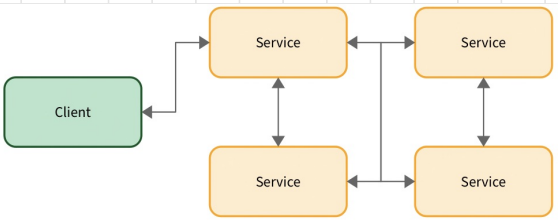
-> virtual infrastructures: elastic resources
↳ fine-grained, pay-as-you-go, resource specific accounting
- Computing
- Storage
- Communication

Elastic Resource

- capable of scaling on the fly to suit user's needs
- Scaling modes:
  - up/down: in/decreasing performance of resources
  - out/in: in/decreasing amount of resources
- Large scaling: distributet architectures, CAP theorem

Service Oriented Architecture SOA

- Atomic services provide single functionality
  → implementing applications = composing atomic services
- services communicated through standardized interface
- control mechanisms restrict access



(+)
- Modular approach: services = isolated entities
  ↳ upgrades, scaling possible on per-service basis
- complexity of service instances low
  ↳ small admin. overhead, codebase easy to maintain
- allows different communication mechanisms (direct, event-based, ...)
- metering per service possible

(−)
- security: multi-tenancy services incorporate risks
- communication: flexibility can become complex in large deployments
- different services with diff. infrastructures => admin. overhead

Cloud:

- cloud resources ≙ services
- inter-service-comm. through broad network
- payment based on service-specific metering, cost models

Services Elastic computing resources

Elastic Infrastructure

- Virtual CPUs (vCPU) incorporated in VMs
- dynamic resource distribution through virtualization
  - ↳ # of vCPUs, CPU cycles per vCPU adjustable
- Hypervisor measures resource usage without affecting machine's perform.
- virtually unlimited # VMs

⊕ maximum flexibility for user applications

⊖ large admin. overhead

- VMs bundle several resources (CPU, memory, storage, ...) → templates
- basic cost model (time-based)
- selecting # of VMs and right best-fitting is NP-hard

Elastic platforms

- Runtime environments on top of elastic infrastructure
- operated by cloud provider → less admin. overhead for users
- per-use configuration possible
- more detailed cost model (e.g. pay per request)

Specialized computing resources

- elastic MapReduce, Video encoding, Batch pipelines, ...
- cost models resource dependent
- used correctly: user can save money, time, effort

Elastic Storage

Block Storage

- emulate behaviour of physical disks:
  - data stored in addressable blocks
  - two atomic ops: read + write
- usually combined with overlaying file system (FS)
- no meta-data support (other than what FS provides)
- elasticity by scaling the adr. space → FS scaled separately
- max. performance: blocks distributed
- fast storage for systems with block-level access

File Storage

- higher-level storage, FS managed by provider
- access control + meta-data support
- single namespace for all files
- storage scalable up and out
  ↳ increased performance, capacity
- file-based locking, replication possible

Key-Value Storage

- store complete data object: data + meta-data
- adressing by user-defined key
- distributed among several backend storage servers
- key hashing to find storing server
- object-level access control, no locking
- flat-namespace: no folders
- highly redundant, replication inherent
- no underlying FS, no database → easily scalable

Consistent Hashing:

consistency problem through redundancy
↳ replica writing might fail

- on read/write...
  - ... wait for number of nodes > 1 to succeed
  - ... wait for one/all nodes to succeed

- Strong consistency if # of reader + # of writers > # of replicas

Comparison

Storage Type	Block	File	Object
Transaction units	Blocks	files	objects
Protocols	SATA, iSCSI, Fibre channel	SMB, NFS, FTP	HTTP
Meta data Support	System attributes	Fixed by FS	User-defined
Best suited for	frequently changing data	shared file data	static data
Biggest strength	Performance	Access, management	Scalability, distributed access
Biggest limitations	Scalability	Scalability beyond data centre	Frequent changing data
Scalability	data centre level	data centre level	global level

Elastic Communication

Virtual Networking

- Infrastructure level communication
- network interfaces emulated in SW / virtualized by hypervisor
- software-defined networks, VLAN tagging for network isolation, flow-control
- higher-level protocols required (OSI layers)

Message based mechanisms

- e.g. direct messaging (Remote Procedure Calls)
  message queuing (Advanced Message Queuing Protocol)
  event-driven messaging (Enterprice Service Bus)
  publish/subscribe

- different message queues depending on use case
  FIFO, Stacks, LIFO, Ring buffer, ...

- has to cope with:
  - adressing, routing
  - message formatting
  - delivery

- message-based architectures ...
  ... enable async app designs
  ... allow parallel processing of messages
  ... increase robustness, availability of the system

Security

- strategies similar to non-distributed systems
  ↳ more difficult to implement

Difficult to get it right, impossible to get it perfect

Types of threats

- Interception: unauthorized data access
- Interruption: service/data becomes unavailable, unusable, destroyed
- Modification: unauthorized changes to/tampering of data/service
- Fabrication: non-normal, additional activity

Security mechanisms

- Encryption : implements confidentiality, integrity
- Authentication: verifying identities Auth N
  - ↳ password, 2 factor, ...
- Authorization: verifying allowable operations Auth Z
  - ↳ is user allowed to do that?
- Auditing : who did what to what, when/how did they do it?

↳ Matching security mechanisms -> threats requires security policy

Example: Globus Toolkit Security Architecture

Globus Toolkit = Grid middleware system

Security Policy

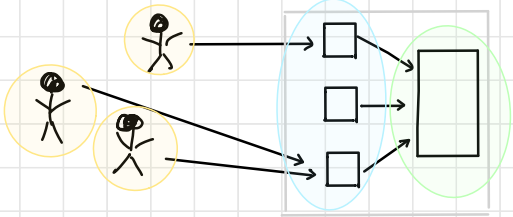
1) The env contains multiple administrative domains
2) Local operations handled by local domain security
3) Global operations: initiator needs to be known all invoked domains
4) Ops btw entities in different domains require mutual authN
5) Global authentication > local authentication
6) Resource access control handled by local security
7) Users can delegate rights to processes ("acting on behalf of user")
8) Group of processes in same domain can share credentials

Entities, Protocols

- users, user proxies, resource proxies, general processes
- entities located in domains
- entities interact
- Protocols:
  1) Creation of user proxy
  2) Allocation of remote resource
  3) Resource allocation from a process
  4) global-to-local mapping

Security Design Issues

1) Focus of Control



Protection against ...

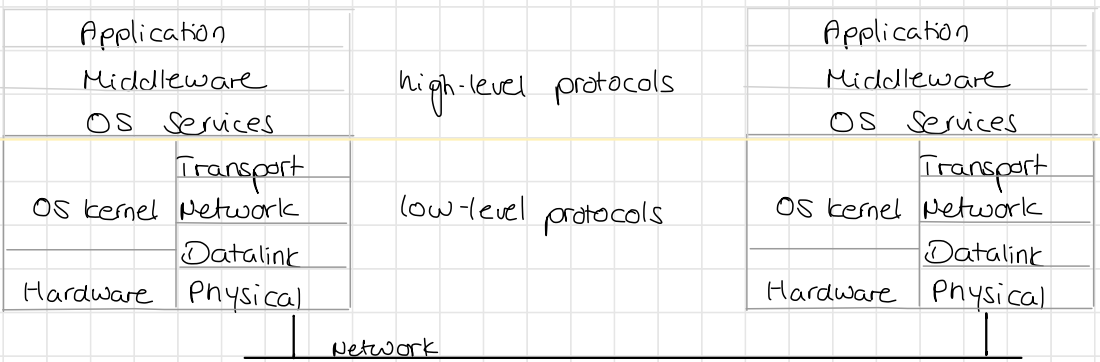
a) ... invalid OPs

b) ... unauthorized invocations

c) ... unauthorized users

2) Layering of security mechanisms

where place mechanisms?



3) Simplicity

- simple and secure difficult to achieve
  - ↳ complexity increases with complexity of system

Security in Grids

- Grid resources communicate via internet
- User, resource pool large + dynamic

U = user
R = resource

Possible threats: Intruder may...

... read / alter messages between R+R / U+R

... connect two Rs and relay messages btw them (man-in-the-middle)

... flood resources with messages (requests (denial-of-service-attack)

↳ Security needs to be everywhere

Now: only user perspective

User security: Requirements

- Identification of individuals
  - ↳ private credentials need to be protected
- Single sign-on
  - ↳ provide pw once, use resources multiple times
- Uniform credentials / certification infrastructure
- Interoperability with local security solutions
  - e.g. different authN, authZ per resource
- Support for multiple implementations

Digital Certificates

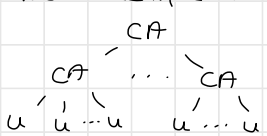
- requirement: trustworthy entity proves (by signature) that it checked the identity of the person and further attributes in the certificate

↳ entity = certification authority (CA)

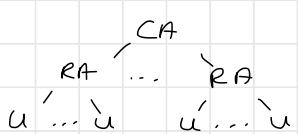
- certificates contain public key as attribute
- user authNs to other entity by presenting certificate + proving knowledge of private key

Public Key Infrastructure PKI

- one CA → bottleneck
  - ↳ chain of trust = PKI
- higher-level CAs certificate other CAs (distributed model of trust)



- one CA, Registration Authorities (RA) issue certificates in the name of CA (central model)



Certification Authority

- issues certificates for servers, users, ... on request
- defines policy for whom / under which conditions certificates are issued
- maintains Certificate Revocation List
- answers questions in certification issues

Registration Authority

- "field office" for CA
  accepts certification requests
- checks identity of a person
- issues no own certificates

Private Key

- needs to be protected → anybody with access can impersonate
- file permissions → only readable by owner

Short-Lived Credential Server SLCS

common certification standard: X.509

⊖ · certificates require a lot of knowledge from the user
· private keys must be secured

↳ Idea: use existing credentials at home institution for short-lived certificates in grid

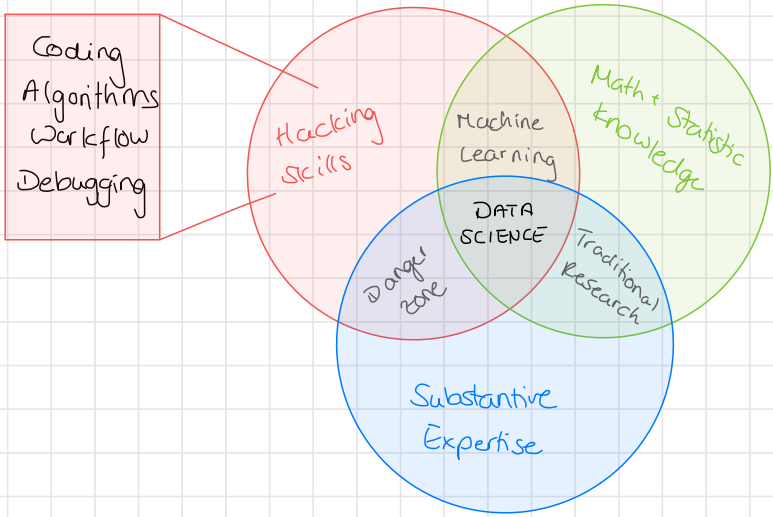
Shibboleth

- online-CA issuing X.509 short-lived certificates based upon Shibboleth identity provider
- lifetime of certificate < 11 days
  ↳ expire, user can reissue often

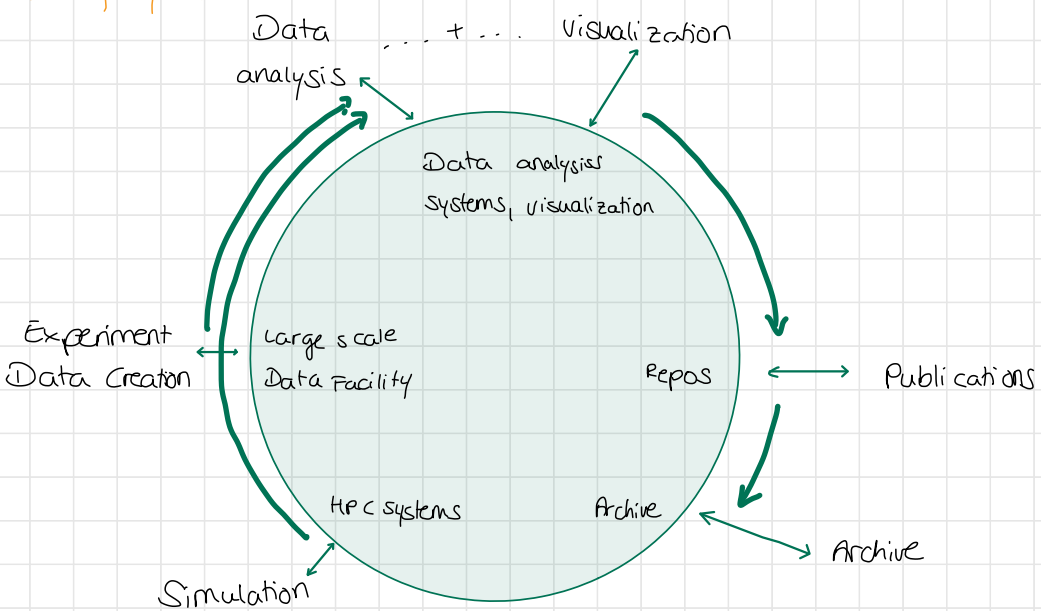
Big Data

Four pillars of science: Experiment, Data, Simulation, Theory

Data Science



Data Lifecycle



Big Data: Vs

Volume
- amount of data

Velocity
- Data Taking
- Ingest
- Analysis
- Transfer
- Visualization

Variety
- Data sources
- Data formats
- Workflows
- Analysis methods
- Interdisciplinary Approaches

Value
- non-reproducible data
- high costs of reproduction

Variability
- changing data
- changing models

Veracity
- systemic, statistical uncertainties
- biased sample

Data Stewardship
- responsible use, protection of digital assets through management, infrastructure support, sustainable practices

Tasks
- analyze data for quality, reconciling data issues
- creating + maintaining metadata
- ensure data preservation
- faciliate data curation

Data Preservation
- ensure long-term viability, availability of digital assets
  - Authentication
  - Reliability (trusted source)
  - Usability (formats, VMs, ...)
  - Integrity (records complete + unaltered)

Long-term:
- depends on data, may be forever
- changing technologies, new data formats, new user communities

Bit-Stream Preservation

- data preservation on bit-level

Increasing data reliability through...

- ... redundant storage
- ... heterogeneous, standardized storage technologies
- ... replicas at remote sites
- ... periodical refreshment, exchanging technologies
- ... security + disaster planning

Economical Aspects

- what to preserve
- how to preserve (metadata, formats, ...)
- paying the costs → who should pay
  - ↳ rule of thumb: lifetime cost is 1/2 ingest, 1/3 preservation, 1/6 access
    (ingest: paid upfront)
- long term preservation → who? (> 100 years)
- who can access
- who is responsible

Data Curation

- maintaining metadata

Metadata

= data about data

- creation date, purpose of the data, author, file size, ...

Types

- Administrative: Acquisition + location info
  documentation of legal access requirements
- Descriptive: title, abstract, data creator, keywords
  annotations by users
- Technical: formats, compression ratios, scaling routine
  AuthN, security data
- Preservation: physical condition of resources
- Use: use + user tracking, content re-use

Persistent Identifier (PID)

- long-lasting reference to document, file, web page, ...
- actionable PID = accessible over internet

PID Resolving

- forwarding users to correct location of PID's object
- e.g. DOI, EPIC

Open Access

- beneficiaries must ensure open access to all peer-reviewed scientific publications (EU H2020 program)

Scientific Publishing Process

1) Author submits article to conference / journal
2) Peer-review
3) publication of "corrected" article in conference / journal
   signature of copyright form
   payment of publication fee
4) final paper is published (book (electronically))
5) research institute subscribe to journals / publisher
   ↳ access to articles

payment twice?

↳ Open Access!

Green OA

- publish work + self-archive final version in repo
- responsibility: author

Gold OA

- publish work
- article processing charge → freely available online via publisher website

Hybrid OA

- business model of publisher partly based on subscriptions/fees
- Gold OA only for articles where author paid special fee

Hybrid OA: Double Dipping
= double payment to publisher (subscription fee + article processing charge)

↳ Solution: price reduction from publisher on subscription fee for OA articles

DATA ANALYSIS

massive amount of data → can't be processed on one machine

Option 1: Batch Processing

Unix command concept

⊕
- Immutable inputs: repetitive execution leads to same results
- no side-effects: inputs transformed into well-designed outputs
- arbitrary stops: pipelines can be ended at any point
- persistence: output can be written to files

⊖
- distribution: all commands locally executed

Option 2: Distributed Batch Processing

- Distributed Filesystem + MapReduce

Distributed FS

data source/sink for in/outputs of computing frameworks

⊕
- leight-weight
- resource localization: provide means to find block's host
- resource allocation: provide means to assign blocks to nodes

Functional programming

- paradigm: code based on evaluation functions
- Function: takes args, produces output
  - avoid side-effects: no changing state / mutable objects
    ↳ same call args => same output

Data Transformation Pipeline

- chain functions -> output of one is input of others
  - ↪ directed (acyclic) graphs : topological ordering
    - node = function
    - edge = dataflow
    - loops for iterative algos available

Transformation functions:

- fixed signature + implementation
- no side-effects : immutable inputs, outputs
- operate on same data formats
- may take + apply user defined callback functions

User defined callback functions:

- implement transformation-function-specific interface
- define application-specific tasks within TFs

Example function pipelining Java

```
String[] chapters = { "Opening,1", "Distributed systems,2", ... }

Stream.of(chapters)
        .distinct()
TFs     .map (s -> s.split(","))
        .filter (t -> Integer.valueOf(t[1]).intValue()>=1)
        .map (t -> t[0])                                    UDFs
        .sorted()
        .forEach(System.out::println);
```

Task parallelism

- execute consecutive functions in parallel
- work on same data / event
- start work before previous function finished

Data parallelism

- same function applied to different subsets of data in parallel

Map Reduce

= programming model (+ associated implementation) for processing large scale data
- automatic parallelization + scheduling
  - ↳ scalable with tera-/petabytes data on thousands of machines
- partitions input data
- handles machine failures + inter-machine communication
- works with procedural languages

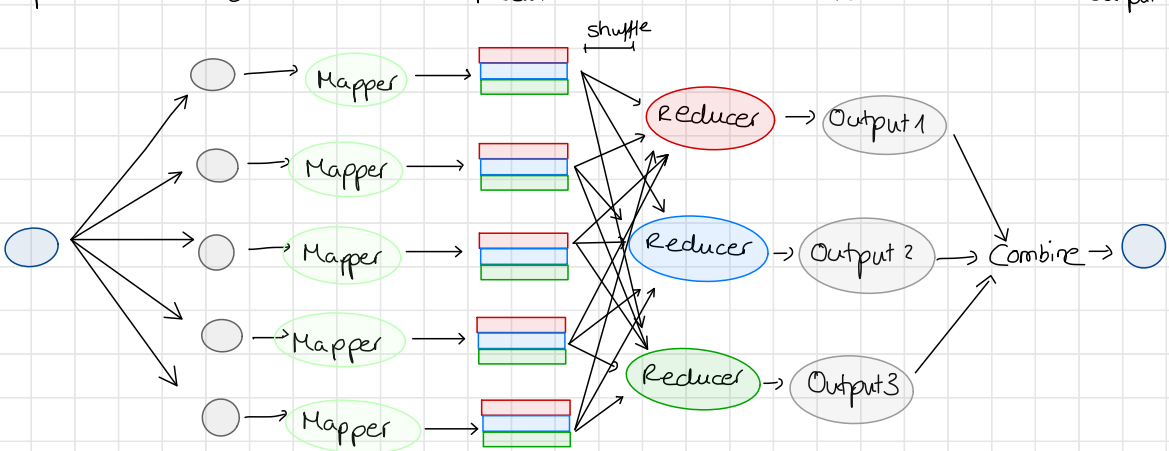
Data + Functions

- basic datatyp = key-value pair (k, v)
- 2 transformation functions

- MAPPERS: transform input record into intermediate record
  $map(k, v) \rightarrow ((k_1, v_1), (k_2, v_2), \dots, (k_n, v_n))$
  ↳ takes one pair, outputs $n \geq 1$ pairs

- REDUCERS: reduce set of intermediate values that share a key to a smaller set of values
  $reduce(k', (v_1', v_2', \dots, v_n')) \rightarrow (k', v'')$
  ↳ takes collection of key-value pairs with same key, outputs value(s)

Map Reduce Execution



Scheduler

Pipeline Startup:

- assigns TFs to nodes
- tries executing functions near their data

Shuffling:

- group key-value pairs by key
- distribute load evenly across reducer nodes

System view

- partitioned parallelisms in map and reduce phases
- distributed + scalable
- fault tolerant
- performance optimized

Hadoop

framework implementing Map Reduce on HDFS (= Hadoop distributed file system)

Limitations:

- disk-based messaging: intermediate results move via disk from mapper to reducer
- static job structure: always 1 map + 1 reduce
  ↳ no arbitrary chaining
- only 2 functions

Hadoop vs. Massively Parallel Processing Databases (MPP DB)

- = DB with distr. storage + parallel query processing

	Hadoop	MPP DB
Diversity of storage	data model independent	enforces one data model
Diversity of processing models	execute arbitrary UDFs	only SQL
design for frequent faults	tries to recover	abort failed queries
Reading workloads	answers queries	full data manipulation support

Algorithm Design

Controlable Aspects

- data structure as keys and values
- init + termination code for mappers, reducers
- preserve state across multiple input and intermediate keys in mappers, red.
- sort order of intermediate keys ~> order in which reducer will encounter keys
- partitioning of key space ~> which keys for which reducer

not controlable Aspects

- where mapper/reducer runs
- when mapper/reducer begins/finishes
- which input key-value pairs for which mapper
- which intermediate key-value pairs for which reducer

Option 3: Data Flow Engines

- distributed batch processing frameworks handling an entire workflow as one job
- Processing of Distributed Acyclic Graphs
  - Transformation functions = DAGs
    - ↳ no strict map/reduce alternation
  - output of one = input of other TF
    - ↳ no disk usage mandatory, TFs start executing when input ready
  - explicit sorting
  - scheduler: put subsequent tasks using same data on same machine
  - data exchange via local resources (shared mem, ...)

Fault Tolerance

- Checkpointing: write intermediate results to disk
  - ↳ start recomputation at latest checkpoint after crash
    - ! recomputation requires deterministic computions

Joins and Groupings

operations for connecting outputs to inputs

1) Repartitioning with sorting
   - Partitions records of two relations by key, sorts each partition
   - sort-merge join
2) Repartition without sorting
   - Partitions records of two relations by key
   - no sorting → important for non-blocking, streaming pipelines
   - hash join
3) Broadcast
   - send records of one relation to all partitions
   - broadcast hash join

Distributed Concurrent Programming

Concurrency

- decomposability property of a program, algorithm, problem into order-independent or partially ordered components or units
- environments for concurrent execution:
  - single-core processors
  - multi-core processors
  - many-core processors
  - distributed systems

⊕
- reduces latency
- hides latency
- increases throughput
- improves program structure

⚠ Concurrency != parallelism

Concurrency = conceptual program property

parallelism = runtime state → dependent on whether hw supports it

Programming Concurrency

Sequential Programming

- Deterministic programming model
- total order of all operations

Declarative Concurrency

- implicit control flow as result of computational logic of program
- allows multiple flows of execution
- concurrency data- / task- driven

Message-passing Concurrency

- (a)synchronous communication between concurent activities

Shared-state concurrency

- multiple activities access contended resources, states
- dedicated mechanisms for synchronization, coordination between activities

Coordination

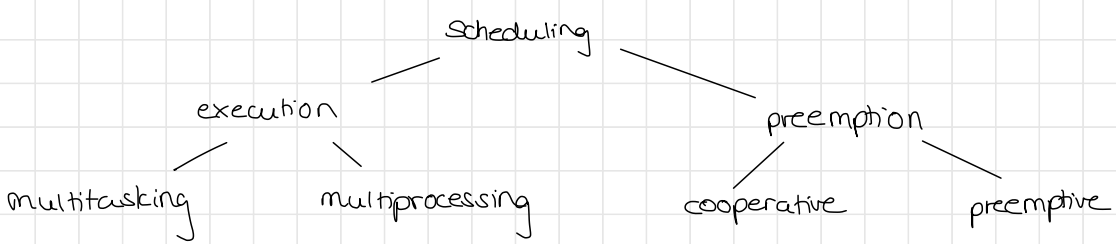
- A must wait for B to complete activity before A can continue

Synchronization

- ≥2 tasks must use same resource that cannot be used simultaneously

Concurrent Execution

- key concept: scheduling



⚠ deadlocks, starvation, indeterminacy

High Performance Computing / Parallelrechner

Parallelrechner = Höchstleistungsrechner = Supercomputer = HPC System

Anwendungsgebiet

Forschung + Entwicklung:
- große numerische Simulationsprogramme
- Datenanalyse
- verteilte Systeme

Industrie:
- große Simulationsaufgaben
- optische Bildverarbeitung - Schrifterkennung
- Banken
- Big Data Analysis
- KI, ML

Multiprozessorsysteme

Speichergekoppelt
- gemeinsamer Adressraum
- Kommunikation, Sync. über gemeinsame Variablen

a) Symmetrischer Multiprozessor: ein globaler Speicher

b) Distributed-shared-memory-System: gemeinsamer Adressraum, physik. verteilte Speichermodule

Nachrichtengekoppelt
- physikalisch verteilte Speicher
- prozessorlokale Adressräume
- komm. durch Nachrichten austausch

Leistungsfähigkeit

Maßzahl: Flop/s floating point operations per second

- nicht notwendigerweise prop. zur Taktgeschw.
  - ↳ Vektorprozessoren: gleichzeitige flop pro Takt

Cluster Systeme

Früher: COTS = commercial off-the-shelf

Interproz. komm. über Netzwerk + Nachrichtenaustausch

Heute: hybride Systeme

- gem. Speicher in einem Knoten (OpenMP)
- verteilter Speicher zw. Knoten (MPI)

↳ 2 Ebenen der Parallelität