VERTEILTE SYSTEME

Distributed Systems

... a collection of independent computers that appears to its users as a single coherent system

... a collection of autonomous computers linked by a computer network and supported by software that enables the collection to operate as integrated facility

Types:
- object- / component based: CORBA, EJB
- file based: NFS
- document based: WWW, Lotus Notes
- coordination- / event based: Jini, Java Spaces
- resource oriented: GRID
- service oriented: web services

- distributed computing: computing grid
- distributed information systems: SOA
- distributed pervasive: P2P

Reasons: connecting users to resources and services
dependability and security
performance

Wrong assumptions: network is reliable
latency is zero
bandwidth is infinite
transport cost is zero
topology doesn't change

→ Quality of Service: client can indicate the level of service (SLA) they require
→ can't have it all → trade offs

– Transparency: hide different aspects of distribution from client
→ provide lower level services which are used by client
↳ service layer provides service with certain QoS

(non functional requirement)
- Access: hide differences in data representation and how resource is accessed
- Location: hide where resource is located
- Migration: hide that resource may be moved to other location
- Relocation: hide that resource may be moved to other location while in use
- Replication: hide that resource is replicated
- Concurrency: hide that resource may be shared by several competitive users
- Failure: hide failure & recovery of system

→ do not blindly hide every aspect of distribution
↳ Trade Off: transparency / performance

- Openness: offer services according to standard rules
  (syntax and semantics: format, content, meaning)
  → formalized in protocols

  Interfaces: complete → interoperability (communications)
  neutral → portability (different implementations)

  Flexibility: composition, configuration, replacement, extensibility

  Separating policy from mechanism: component interaction and composition standards

  → different web servers & browsers
  → plugin interface for browsers
  → caching algorithms

- Scalability: ability to grow to meet increasing demands along
  - size (users / resources)
  - geographically (topology)
  - administratively (independent organizations / domains)
    → conflicting policies: mail servers

  → System remains effective without need to change
  → Trade Off: scalability / security

  Performance bottlenecks: centralized services / data / algorithms

  → decentralized algorithms: no machine has complete system state information
  machine make decision based on local info
  failure of one machine does not ruin algorithm
  → no assumptions about global clock

  Techniques: hiding communication latencies
  - asynchronous communication
  - reduce overall communication

  distribution: hierarchies, domains, zones

  replication: availability, load balancing
  → caching (→ consistency issues)

Dealing with complexity:
- abstraction: interface vs implementation
- information hiding
- separation of concerns: components
  client / server

Communication models:
- Multiprocessors: shared memory
  → semaphores, monitors
- Multicomputers: message passing
  → blocking

Architectural styles: layered architecture: request flow
object based architecture: method calls
data - centered architecture: shared data space
event - based architecture: event bus

Application layering: user interface level
processing level
data level

Multitiered architectures: client / server
→ split application levels between tiers
↔ vertical distribution
→ multi tier systems

Processes and Communication

⇒ Processor: Provide set of instructions along with ability to execute series of instructions
- context: collection of values in registers of processor used for executing instructions

⇒ Process: Software (virtual) processor in whose context threads can be executed
- context: collection of values in memory and registers for executing thread

⇒ Thread: Minimal software processor in whose context a series of instructions can be executed
- context: collection of values in memory and registers for executing instructions

⇒ Threads can share same address space: cheaper + creating and destroying
⇒ Process switching involves asking OS in the loop: expensive

Benefits of multithreaded processes: parallelism
blocked threads for blocked tasks

⇒ User-space thread solution
⇒ Kernel-space thread solution

Virtualization: abstract view on IT resources
→ pooling of resources

Benefits: resources are shared: more efficient
consolidation: different desires of applications
fault tolerance: isolation of failures

⇒ mimicking different underlying hardware

Architecture of VMs: virtualization can happen at different levels
- process VM: programm compiled to intermediate code
- VM monitor: software layer mimicing instruction set of hardware

Clients: major part focused on user interface
but: eg client side request replication

distribution transparency:
- access transparency: client stub
- location / migration transparency: keep track of server location
- replication transparency: handled by client stub
- failure transparency

Server: process waiting for incoming requests at specific transport address
→ superservers, iteration vs concurrent servers

Out of band communication: separate port
facilities of transport layer

Stateless vs. stateful server

Server clusters: different tiers: logical switch
application / compute servers
file / database system

Distributed servers: TCP handoff, IP06 spreading

Communication in distributed systems

ISO/OSI Modell: add headers on each layer
→ TCP: Physical, Data Link, Network, Transport

→ Middleware Layer: used for common functionalities in different applications: fault tolerance, security, synchronisation
↳ between Application and Transport

Types: persistent vs transient: communication server stores message until delivery
synchronous vs asynchronous
→ usually transient + synchronous

Remote procedure call: invoke procedure in another process

- Stub Generation: Transport information, Interface description, message format

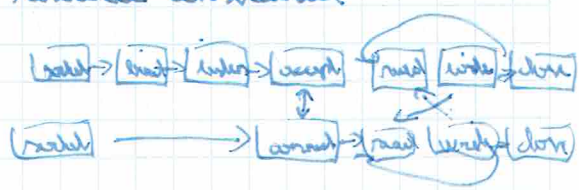
→ Asynchronous RPC
→ Remote Object call: RPC style in object oriented programming
↳ RMI client, RMI Registry, RMI Server

→ XML-RPC, JSOM-RPC

Message oriented communication

- Transport-level socket programming
→ low level systems, high performance, resource constrained

Server: bind, listen, accept, close
Client: connect, close



- Message-oriented middleware
↳ message queuing systems
↳ large-scale systems: persistent + asynchronous
scalable message handling
↳ PUT, GET, POLL, NOTIFY

- Message Broker: Transform incoming messages to target form
→ application gateway

Multicast Communication: sending data to multiple receivers

Application level multicasting: based on overlay network
- composed of direct connections between hosts
- independent from physical network

Gossip based Data Dissemination
- pass updates to a few neighbors
- if already known, stop passing with probability of $1/k$
- does no guarantee that everyone is informed!

Naming in Distributed Systems

... identification, naming, detailed description, functions for communication, security, auditing

→ Name: set of bits/chars used to identify an entity in a context
Identifier: name that uniquely identifies an entity
Address: name of an access point, location of an entity

data models: information about names
processes in naming services: creation, update, query, resolution activities

name space: contains all valid names recognized and managed by a service
→ not every name is bound to entity, alias: refers to another name
↳ naming domain: name space with a single administration authority

→ Naming design is based on specific system organizations and characteristics

- dynamic vs static binding (name to address binding mechanisms)
- distributed vs centralized management
- discovery / resolution protocol

Flat naming ... no structured description, just a set of bits,
do not contain information for understanding the entity
→ MAC address

→ broadcast based name resolution: ARP

→ Distributed Hash Tables: keyspace + management - range
→ Chord: ring network, nodeID = hash (IP)
↳ finger table pointing

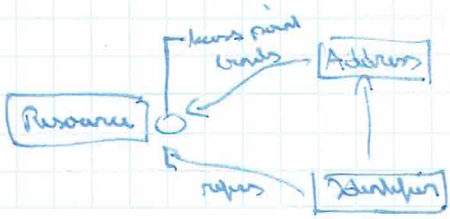
Structured naming ... organized in name spaces which can be modelled as graphs
→ leaf node vs directory node, → DNS

→ closure mechanism — multiple absolute path names referring same node, leaf node storing absolute path
→ Alias using links → hard link, symbolic link
→ Mounting: link local node to remote directory node

→ distributed name management: many distribution layers:
- global layer
- administrational layer: managed by single organization
- managerial layer: change regularly — higher load for servers, costumer, overall traffic — less overall cost for client

→ iterative vs recursive name resolution

→ DNS: map from domain name hierarchy to IP address
→ root name server, administered Zone name servers, authoritative names servers
primary & secondary servers, caching servers



Attribute-based naming ... use a set of tuples to describe entity
also called directory service
⇒ LDAP, RDF

Name resolution: querying mechanism

LDAP data model:
Object class: type of an entry, describes possible attributes
Directory entry: particular object, identified by DN (distinguished name)
⇒ alias entry + subentry possible
Directory Information Base: collection of all directory entries
Directory Information Tree: tree structure for entries in DIB

Naming services in the web

Web Services: described by WSDL
↳ contain addresses

↳ Service Requester, Service Provider, UDDI registry

Open ID: people identifier in the web
⇒ identifier that can be accepted by several service providers
↳ Open ID identifier is an URL

⇒ Open ID interaction flow

Fault Tolerance

→ in DS components depend on each other as the use their services (inter-component)
↳ clients require correct services regarding to functional and non-funct. properties

Dependability:
- Availability
- Reliability — continuity of correct service
- Safety — absence of catastrophic consequences
- Integrity — absence of improper system alterations
- Maintainability — ability to undergo modifications

→ Fault: Cause of an error (bug)
Error: Deviation of the actual system state from the prescribed one (wrong value)
Failure: Delivered service deviates from correct service (return wrong value)

→ Failure Models:
- Crash Failure
- Omission Failure
  - Timing Failure
- Response Failure → reproducible
  - Arbitrary Failure

→ Fault prevention: inspection, static code analysis
→ Fault forecasting
→ Fault tolerance: keep service provision
→ Fault removal

Fault Tolerance: no fault tolerance without redundancy

→ Failure masking by redundancy (add)
- Information redundancy: add extra information → parity bit, error correcting codes
- Time redundancy: repeat request → retransmission TCP
- Physical redundancy: add additional components → backup servers, different implementations in different processes

Process resilience ... how to tolerate faulty processes
→ organize several identical processes in a group

Hierarchical Groups: single coordinator, single point of failure
Flat Groups: control completely distributed, voting needed → overhead

k-fault tolerant group:
- crash / performance failure: $k+1$
- arbitrary / byzantine failure: $2k+1$

Byzantine Agreement Problem:
- send own result
- send individual vectors
- send result if majority

Reliable Client Server Communication

Client cannot locate server
→ report back to client → exception handling

Client request is lost
→ resend request (→ server needs to remember if retransmission)

Server crashes
Server behaviour: At least once
At most once
Client behaviour: Always reissue request
Never reissue request
reissue only when received ack
reissue only if not received ack

→ 8 possible combinations
↳ no combination correct for all sequences
ack message print crash
→ Sequence: M → P → C

Server response is lost
→ repeat request (only if idempotent operation)
↳ else no solution (how to we know server did not crash?)

Client crashes (after request has been sent)
– server executes request and send response (orphan computing)
↳ Orphan killed by client if it is received
↳ Reincarnation: Client tells server about reboot, server kills orphan
↳ Expiration: Require computations to complete in T time units
Old ones are simply removed

Time Synchronization: achieving accountability of processes
maintaining correctness in processing messages
establishing validity of messages
achieving fairness in processing requests

Physical Clocks: Universal Coordinated Time: average of various clocks
→ broadcast through short wave radio & satellite

Distributed time in network: update time on interrupt
→ drift possible → synchronize at least every $\delta / 2\rho$ seconds
δ.. allowed difference, ρ... drift given by manufacturer

Cristian's algorithm: synchronization over external time-server
ask time to answer time + RTT/2 (client side computation)

Round Trip Time

Berkeley algorithm: synchronization by external time server (no reliable time source needed)
server initiates sync-process
- ask clients for local time (Cristians alg., broadcast)
- calculate average, calculate relative adjustment
- send relative adjustment

NTP: protocol for time propagation
hierarchical organization → stratum levels

Logical Time: local or global time wasn't important → make order of events understandable
→ order of events is important → no absolute time needed

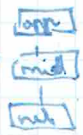
Happened before relationship: $a \to b$ in same process, a before b: $a \to b$
a is sending message, b is receiving message: $a \to b$
$a \to b$, $b \to c$: $a \to c$
→ partial ordering of events
→ concurrent events: $a \| b$

Lamports logical clock: set of logical clocks, one per process
→ different increase-rate for each process
attach timestamp to each event, increase timestamp before executing event
→ if $a \to b$: $C(a) < C(b)$
adjust timestamp on receipt of message
→ in middleware layer
then increase, then execute

→ Totally ordered multicast: send timestamped message to all others
put messages in priority queue ordered by timestamp
→ execute message, if smallest timestamp of all process queues
→ guarantees that concurrent updates are seen in the same order everywhere

$C(a) < C(b) \not\Rightarrow a \to b$: we miss causality information

→ adjustments in middle layer



Vector clocks: each process maintains vector VC_i
→ $VC_i[i]$ ... number of events happened in process i
→ $VC_i[k] = n$ .. process i knows that n events occured in process k that might have causal relation with process k

→ causally ordered multicast: increment $VC_i[i]$ only on send
receive message when
$ts[i] = VC_k[i] + 1$ .. received all preceding messages from i
$\forall x \neq i, ts[x] \leq VC_k[x]$ .. already aware of all other messages which the sender knows

Mutual Exclusion: exclusive access to some resource

centralized: single coordinator, request + release messages, queue at coordinator (→ only OK message)
→ single point of failure → synchronized

decentralized: resource replicated n-times, each replica has own coordinator
access requires majority vote $> n/2$ oks
OK + rejection message → vague
→ many competing nodes: no one gets access

distributed: assume totally ordered events (Lamport)
send request to all other processes
OK message or queuing, block when in use
→ n points of failure: no rejection message

token ring: organize processes in logical ring
pass token in ring, access resources when holding token
→ ring topology required, lost token situation

Election algorithms: often we need a leader / coordinator
pre chosen coordinator: single point of failure
→ select coordinator dynamically

Assumptions: processes uniquely identified, process know group membership
anyone notice failure of coordinator, anyone triggers election

Election by bullying: process sends election-msg to all processes
if no one with higher PID responds, process becomes coordinator and notifies everyone
if someone with higher PID responds, he starts with his part

Election in a ring: send election-msg to neighbor, sender adds itself to list when back at initiator; pass list with other processes
highest PID elects itself

Election in wireless networks: understanding current network topology (spanning tree)
respond with highest PID of own broadcast

Consistency and Replication

Replication: process of maintaining several copies of a data item at different locations

Consistency: process of keeping data item copies the same when changes occur

→ Reasons: performance and scalability: serve more client requests replicas close to client
fault tolerance by redundancy

→ Drawback: keeping replicas up-to-date consumes bandwidth
replicas not immediately propagated → stale data

→ Trade Off: Scalability
Performance
Consistency

Consistency model: contract between data store and processes, in which data store specifies what the results of read and write operations are in the presence of concurrency

Data-centric consistency models: guarantee consistency of entire data store

Sequential consistency: result of any execution is as if the operations of all processes were executed in some sequential order.
operations of each process appear in the global sequence in correct order.

Causal consistency: Writes that are potentially causally related must be seen in the same order by all processes
Concurrent writes may be seen in different order by different processes

FIFO consistency: Writes by P1 must be read in correct order by every other process.
Writes from two different processes can be read arbitrary

Client-centric consistency models: no concurrent writes / updates
→ relaxe consistency model

→ eventual consistency: after a write, replicas will gradually become consistent
problem: clients connect to other replicas
↳ client centric consistency models: guarantee for a client, that his accesses to data will be consistent

Monotic read consistency: successive read operations will always return same or more recent value → calendar reading

Monotic write consistency: write operation by client will always be completed before successive write by same process (→propagated)
→ maintaining version of files
→ central logging server

Read your writes consistency: effect of a write operation by a process will always be seen by a successive read operation by same process -> update weppage

Writes follow reads consistency: write operation by a process following read operation will always take place on same or more recent value which was read. -> adding comment

Replica Management: what is the best strategy of placing replicas to make an efficient use of replication and be able to maintain consistency among

-> if access-to-update ratio is high, replication pays off
-> if update-to access ratio is high, many updates are never read
↳ update replicas which are accessed

Permanent replicas: initial distribution of data
-> replicated website servers for load balancing

Server-initiated replicas: server decides when and where new replicas are created.
· different algorithms using monitoring and access-statistics determine optimal placing

-> Web hosting companies with many servers
each server keeps track of access counts per file, aggregated by domain server to client
replication threshold + deletion threshold
-> migration / replication also depends on cost of operation

Client-initiated replicas:
-> Cache: local storage facility managed and used by client to temporary store data and improve access times -> cache hit

-> useful if read-to-write ratio is high
-> data fetched and stored in cache upon request
-> stored for limited time only

- hardware caches: modern CPUs
- Software based solutions: middle ware based DS

Content Distribution: strategies and models of distributing content (files, DB, ...)

Client-Server combination:

Invalidation: send notification on update (no data)
low read-to-write ratios

Passive replication: transfer data from one copy to other
consumes bandwidth
high read-to-write ratio → bundle updates

Active replication: send operations instead of data

Push-based (server-based) protocols: server needs to know status of clients
propagate updates to other copies without need of request
→ ensuring consistency better.
→ inefficient if no efficient multicast implementation

Pull-based (client-based) protocols:
client polls server to check for update, then asks for update
→ most used by client caches
→ increased response time on cache miss

Lease: hybrid solution to switch between pulling and pushing,
contract in which the server promises to push updates until
lease is expired

→ minimize load on server and speed up updating by
dynamically choosing different lease durations

- expire based leasing
- renewal frequency based leasing
- state (server) based leasing

Blocking: propagate changes to other replicas, then answer request
→ synchronous, weigh

Non-Blocking: update one copy, then answer request, then propagate
changes
→ asynchronous, lazy

Consistency Protocols

Primary-Backup Protocol: implements sequential consistency model

- → Primary server for every file
  - all writes are executed on the primary server and propagated to other servers → synchronous on write
  - changes are atomic, no inconsistencies
- → fast reading, slow writing
- → can't write if one node is not available

- → non-blocking where possible: ack when primary got update
  - → avoid DoS because of node failure
  - → inconsistencies can occur (→ node not up to date)

Primary-Backup with local writes: at least FIFO consistency

- → change primary to server with last write
- → writing is fast (when there are not many concurrent writes)
- → asynchronous → propagate changes after primary change
  - ack on primary change

Quorum-based protocols: read quorum + write quorum

- → client has to write to count of write quorum replicas
- → need to contact two nodes, data has timestamp to compare newest

- → Configure WQ and RQ as needed
  - optimize read: $W=N \quad R=1$
  - optimize write: $W=1 \quad R=N$
  - avoid write conflicts: $W \geq (N+1)/2$
  - strong consistency: $W+R > N$

Distributed File Systems: make filesystem transparently available to remote clients

Remote access model: File stays on server
Upload / Download model: File moved to client, then accessed, then returned to server

NFS is implemented using VFS (Virtual File System) abstraction:
→ VFS: layer between system calls and local file system interface
used in a lot of modern file systems
→ provides standard file system interface and therefore allows to hide difference between accessing local or remote file system

NFS v4: Open file and Close file operations possible
(ursprünglich über UDP, jetzt über TCP → nur ein Port (Firewalls))

Cluster-based file systems: very large data collections cannot be handled by server-client approach

→ split files in blocks/chunks and distribute over different servers
↳ enable parallel fetching

Google file System: divide files in 64MB chunks and replicate chunks across many servers

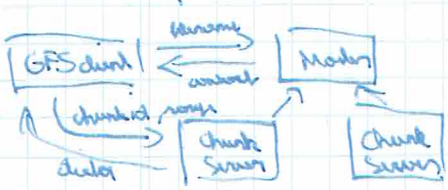
master maintains file-name/chunk server table in main memory → minimal I/O
files are replicated using a primary-backup scheme
→ master is kept out of loop

File Sharing Semantics: define ordering / effects of concurrent read / write operations

UNIX semantics: read returns effect of last write
→ only possible if only single copy of file
Transaction semantics: system support transactions on single file
→ how to allow access to physically distributed file?
Session semantics: effects of read or write operation only seen by client who has opened (a local copy of) the file
→ what happens when file is closed? only one client can win

→ File Sharing in Coda: assume transactional semantics without full-fledged capabilities of real transactions
↳ improve performance

ensure that concurrent updates are detected:
Version Vector: CVi(f)[k] = j ... Si knows, that Sk has seen version j of f



Current Trends in Distributed Systems

Internet of the Things: physical objects are seamlessly integrated in the information network

→ Physical objects become active participants in business processes
↳ become „smart"
- Technologies: RFID, sensor networks, IPv6

Service oriented computing:
Internet of Services: Software services provided through the internet [Rest, WSDL, SOAP]
foundation for cloud computing
Service oriented architecture: mainly concept to organize IT software in an organization

Motivation: cross organizational business process because of globalization
flexibility is a key success factor
→ flexible IT architecture required, reusable

Service oriented architecture:
IT architecture made up from single services → self contained, reusable software components with distinct functionality
Complex applications arise from coupling of single services

Roles

Cloud computing: on-demand self service: quick, automated rental of capacity using web interfaces
- resource pooling: use virtualization techniques
- rapid elasticity: virtually unlimited capacity + scalability
- measured service: pay as you go

→ demand for service varies with time
→ unknown demand: startups
→ Batch analytics

Service models: cloud
Infrastructure as a Service: Amazon EC2, IaaS
Cloud Platform as a Service: Google app engine
Cloud Software as a Service: Salesforce: ERP software
SaaS

Deployment models: Public Cloud, Private Cloud, Community Cloud, Hybrid Cloud

Cloud vs IoT: structural differences: distributed vs centralized
↓
Fog computing: push principles from cloud computing to use IoT
→ use IoT devices as virtualized assets (Sensor / actuator)
→ rapid elasticity by pooling local assets

Fog Cell: Software component running on virtualized IoT device
Fog Colony: Macro data centers made up from fog cells

Security: most important aspects: integrity, confidentiality
... overlaps with other wanted properties of distributed systems

Players: Subject, Channel, Object

Threats: Interception, Interruption, Modification, Fabrication

Security mechanisms: Encryption
Authentification
Authorization
Auditing

Cryptography: Symmetric System: encryption → prevention of interception
Asymmetric System: authentification → prevention of fabrication
Hashing System: integrity → prevention of modification

- essential properties of cryptographic functions: $E_K(m_{in}) = m_{out}$
  - make encryption method public, but as a whole parameterized by a key
  - one-way-function: given some output, it is hard to find input
  - weak-collision resistance: given input + output, hard to find input for same output
  - strong-collision resistance: hard to find 2 inputs with same output
  - one way key: given input + output + method: hard to find key which matches
  - weak key collision resistance: given input + key + method: hard to find key with same output
  - strong key collision resistance: hard to find two keys with same output

Secure Channel: authentification + integrity + confidentiality
(only one is useless)

Secret shared keys: secret keys reflection attack

Key distribution center: need to manage $N \cdot (N-1)/2$ keys → each subject knowing $N-1$ keys
→ use trusted key distribution center for generating key when necessary
→ each subject needs key for KDC

Needham-Schroeder Authentification
- use nonce against spoofed KDC response (message replay)
- use target in KDC response to avoid interception / tampering of request
- use nonce which is determined to prevent replay attack

Problem with keys: keys never end
damage of replay
compromised keys
} use session keys for one connection

Digital signatures: Authentification
Non repudiation
Integrity

→ use own private key before target public key (not recommended)
→ message digests: separate authentification + secrecy
→ send message in clear text, sign digest with private key

Security Management

Key establishment: Diffie Hellman

construct secret keys without third party
+ without need to send whole key ⇒ discrete logarithms

Key distribution: Secret keys: create own, exchange out of band
trust a KDC
Public keys: personally exchange out of band
trust a certification authority
↳ public key is put in certificate signed by CA
Certificate lifetime: certificate revocation list
expiration time

Access control: authentication vs authorization

access control matrix: contains allowed operations for subjects on object O

protection domains: groups: each group has associated rights
roles: role is determined at login, can be changed
⇒ use certificates providing info about group / role membership

Firewalls: check incoming packets before they reach destination
⇒ filtering rules } filtering routers
⇒ routing
⇒ packet inspection: interpret content } application level firewall

Secure mobile code: protecting host

sandbox model: allow only pre-defined resources and services
playground model: run code on separate machine
↑
untrusted

Common Attack Scenarios: can be compromised on any layer
large attack surface

- Stack Buffer Overflows
- SQL injection
- Cross-Site-Scripting (XSS)
- Distributed Denial of Service
- Side-Channel
- Social Engineering