Introduction

Distributed Systems Definition:
Collection of independent computers that appear as a single coherent network.

Indepentend computers linked by a network aided by a software to work as an ntegrated facility.

When the crash of a computer you never heard of stops you from gettin shit done.

Types of distributed Systems:
Object/component based (CORBA, EJB, COM)
File based (NFS)
Document based (WWW, Lotus Notes)
Coordination (or event-) based (Jini, JavaSpaces, publish/subscribe, P2P)
Resource oriented (GRID, Cloud, P2P, MANET)
Service oriented (Web services, Cloud, P2P)

Distributed… Computing, Information System, Pervasive

Concepts:
Communication (E-Mail)
Concurrency and operating system support (competitive, cooperative) (Parallel Computing)
Naming and discovery (DNS)
Synchronization and agreement (TCP Handshake)
Consistency and replication (Netflix)
Fault-tolerance (Redundancy)
Security (XSS)

Why?
Connecting users to resources and services (Basic function of a distributed system)
Dependability and Security (Availability, Fault tolerance, Intrusion Tolerance)
Performance (Latency, throughput)

Design goals:
Resource sharing (collaborative, competitive)
Transparency
Hiding internal structure, complexity (Openness, Portability, interoperability)
Services provided by standard rules
Scalability
Ability to expand the system easily
Concurrency (inherently parallel (not just simulated))
Fault Tolerance (FT), availability

Eight wrong assumptions:
The network is reliable
Latency is zero
Bandwidth is infinite
The network is secure
Topology doesn't change
There is one administrator

Transport cost is zero
The network is homogeneous

Connecting Users and Services:
Access and share (remote) resources
Economics and policies
Collaboration by information exchange
Communication (Convergence, VoIP)
Groupware and virtual organizations
Electronic and mobile commerce
Sensor/actor networks in automation and pervasive computing (fine grained distribution)
May compromise security (tamper proof HW) and privacy (tracking, spam)

Quality of Service:
QoS is a concept with which clients can indicate the level of service (SLA) they require
Examples:
• For real-time voice communication, the client prefers reliable delivery times over guaranteed delivery
• In financial applications, a client may prefer encrypted communication in favor of faster communication
You can't have it all -> Trade-offs!

Transparency
•Hide different aspects of distribution from the client. It is the ultimate goal of many distributed systems.
Can be achieved by providing lower-level (system) services (i.e. use another layer).
The client uses these services instead of hardcoding the information.
The service layer provides a service with a certain Quality of Service.

- Access, hide differences in data representation and how a resource is accessed
- Location, hide where a resource is located
- Migration, hide that a resource may move to another location
- Relocation, hide that a resource may be moved to another location while in use
- Replication, hide that a resource is replicated
- Concurrency, hide that a resource may be shared by serveral competitive users
- Failure, hide the failure and recovery of a resource

→ Information hiding applied to distributed systems.

Do not blindly hide every aspect of distribution. Trade-off between transperency and performance, failure masking and replica consistency

→ Transpareny is an important goal, but has to be considered together with all other non-funtional requirements and with respect to particular demands

Openness
Offer services according to standard rules (syntax and semantics)
Formalized in protocols
IDL, semantics often informal (complete → interoperability, communication between processes; neutral → portability, different implementations of interface)

Felxibility: CBSE Component based software engineering; composition, configuration, replacement extensibility

Seperating Policy from Mechanism:
Granularity: object vs. Applications
Component interaction and composition standards
E.g. Web browser caching policy can be plugged in arbitrarily

Achieving openness examples
Different Web servers and browsers interoperate, new browsers made to work with existing servers, plugin interface allows new services to be added.

Scalability
→ ability to grow in size (users and resources), geographically (topologically), administratively (independent organizations and domains)
System remains effective, software should not need to be changed, tradeoff scalability/security

Scalability in size, challanges:
Cost of physical resources should be O(n)
Performace loss should be no worse than O(log n)
Preventing software resources form running out

Performance Bottlenecks:
Centralized services: single server for all users
Centralized data: single on-line telephone book, central DNS
Centralized algorithms: doing routing based on complete information

Decentralized Algorithms:
→ no machine has complete system state information
→ make desicions based only on local information
→ failure of one machine does not ruin the algorithms
→ no implicit assumption that a global clock exists

Geographoical scalability:
LAN synchronous, fast, broadcast, highly reliable
WAN asynchornous communication, slow, point to point, unreliable

Scaling techniques:
→ Hiding: communication latencies. Asynchronous comm., reduce overall comm.
→ Distribution: hierachies, domains, zones → split
→ Replication: Availability, load balance, reduce communication distance; caching (proximity, client descision)
Consistency issues may adverse scalability!
Let server or client check forms as they are being filled.

Achritectural Styles:

Abstraction and modeling: client – server – service, Interface vs. Implementation
Information hiding (encapsulation): interface design
Seperation of concerns: layering (e.g. FS bytes – disc blocks – files), client and server, components (granularity issues)

Communication Models
Multiprocessor: shared memory (protect concurrent access, synchronisation: semaphores, monitors)
Multicomputers: message passing (synchronisation: blocking in message passing)

Architectural Styles:
→ Layered
→ Object-based
→ Data-centered (shared (peristen) data space)
→ Event-based (Event bus, components, event delivery)

Centralized architecure: client – server

Application layering: user-interface level – processing level – data level

Simple two types: (2-tier architecure)
client: containing only the implementation of the user-interface
server: containing the programs implementing the processing and data level

3-tier architecture: client (presentation layer), middleware (application logic layer), resource management (persistence layer)
resource tier – business tier – presentation tier

Processes and Communication I

Virtual processors on top of physical processors
→ processor: provides set of instructions
→ process: virtual software processer in whichs context one or more threads are executed
→ thread: minimal software processor in whichs context a series of instructions can be executed

context switching:
→ processor context: registers used for the execution of a series of instructions
→ thread context: registers and memory for the execution of a series of instructions
→ process context: registers and memory for the execution of a thread

threads share the same address space, switching can be done independent from OS
process switching is more expansive as it involves the OS
→ creating and destroying threads is much cheaper than doing so for processes.

<u>Threads:</u>
→ singlethreaded: blocking system call blocks the whole processeor
→ benefints of multithreading:
  -parallelism: putting threads on different CPUs, data is in the shared main memory
  -large applications
  -software engineering: dedicated threads for dedicated tasks

Kernel vs. User-space threading:

User-level:
+ cheap, no OS involved only lease and release address space
- services provided by the kernel are done by the process.

Kernel-level:
+ operation that blocks thread no longer a problem
- loss of efficency, all thread operation has to be carried out by the kernel

→ mix user-level and kernel-leve threads into a single concept

Virtulization
...abstract view on IT resources, possible on different levels: platform, memory, HDD, network

Application
Library Functions
Library
system calls
Operating System
priveledged instructions
Hardware (ISA instruction set architecture, privileged vs. general instructions)

VM Monitor
Native: A separate software layer mimics the instruction set of hardware:
Hardware – VM Monitor – OS – Application/Libraries
Hosted: On top of existing OS:
Hardware – OS – VM Monitor – OS – Application/Libraries

Benefits:
- Higher degree of capacity utilization: resources are shared between users
- Consolidation: many different classes of applications on different virtualized assets, less energy consumption and space usage
- Fault tolerance: isolation of failures caused by errors or security problems

Clients
Application
Middleware
Local OS ------→ Application Independent Protocol to Server

Distribution Transparemcy:
- Acceess transperency
- Location/migration transperency
- Replication transperency
- Failure transperency

Client side handles request replication and sends request to multiple server

Server
A process waiting for incoming service requests, subsequently making sure that the request is taken care of, after which it waits for the next incoming request
Types: iterative, concurrent

Out-of-band Communication: a) use seperate port for urgent data (server has sperate thread/process for urgent messanges) or b) use out-of-band communication facilities of the transport layer

stateless server: keep no information on state of clients, clients and server completely independent, state inconsitencies due to client or server crashes are reduced, possible loss of performance

stateful server: maintain persistent information about its clients, performance gains possible since the server has additional information about its clients

3-tier server:
first tier: client requests → Dispatched requests
second tier: application/compute servers
third tier: distributed file/database system

TCP-handoff: client send request to swicht, switch hands off request to one of several availabe servers, server responds directly to client, logically all one single TCP connection

Distributed Servers
Addressing: Clients having Mobile IPv6 can transparently set up a connection to any peer:
- Client C sets up a connection to a IPv6 home address HA
- HA is maintained by a (network-level) home agent, which hands off the connection to a registered care-of address CA
- C can then apply route optimization by directly forwarding packets to address CA

Processes and Communication II

Communication in distributed systems
- between processes within a single application/middleware/service
- among processes belonging to different applications/middleware/services
- among computing nodes which have no concept of processes (e.g. sensors)

Application Layer HTTP, SMTP, FTP
Presentation Layer
Session Layer
Transport Layer TCP, UDP
Network Layer Ethernet
DataLink Layer LAN
Physical Layer LAN Cable

Headers added on each layering

Transport Layer provides actual communication facilities for most distributed systems.
TCP: connection oriented, reliable, stream oriented communication
UDP: unreliable (best effort) datagram communication

Interprocess communication based on low-level message passing offered by the underlying network. Communication entities: Processes

Most common protocolsuit in the internet: TCP/IP. Application Layer, Transport Layer, Internet Layer, Link Layer

Middleware Layer
provides common functionalities to different applications (e.g. fault tolerance, security, synchronization). It‘s a richt set of communication protocols, (un)marshaling of data necessary for integrated systems, naming protocols, security protocols, scaling mechanisms

Application ---- Application protocol ---- Applications
Middleware ---- Middleware protocol ---- Middleware
OS ---- host-to-host protocol ---- OS
Hardware ---- Physical/Link-level protocol ---- Hardware

Network

Types of communication

Persistent vs. Transient and Asynchronous vs. Synchronous
→ Transient: Communicating server discards message if it cannot be delivered at the next server, or at the receiver
→ Persistent: Message is stored at communicating server as long as it takes to deliver it

→ Synchronous: Sender is blocked until its request is known to be accepted
→ Asynchronous: No such functionality

Client/Server is usually transient and synchronous:
Client and server have to be active at the time of ommunication
Client issues request and blocks until it receives reply
Server essentially waits only for incoming requests, andsubsequently processes them

Drawbacks of synchronous communication:
→ Clients have to wait for reply (and do nothing else in the meantime)
→ Failures have to be handled immediately because the client is waiting

Remote Procedure Calls
A process calls/invokes a (remote) procedure in another process

1. Client call to procedure
2. Client stub builds message
3. Massege is sent across network
4. Server OS hands message to server stub
5. Server stub unpacks message
6. Stub makes call to „doit“

Generating Stubs:
Transport infromation (HTTP, TCP, UDP)
Interface description (IDL, XML)
Message format (XDR, XML)
→ Stubs: code for marshaling/unmarshaling

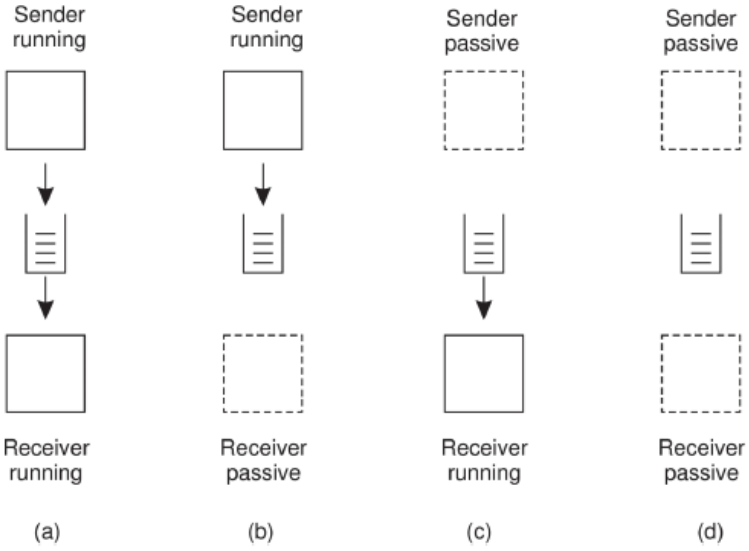
Asynchronous RPC: Client calls remote procedure and only waits for acceptance then returns to his business while the server processes the request / invokes the procedure

Implementations: XML-RPC: XML for messages, HTTP for transport
JSON-RPC: JSON for messages, HTTP and/or TCP for transport

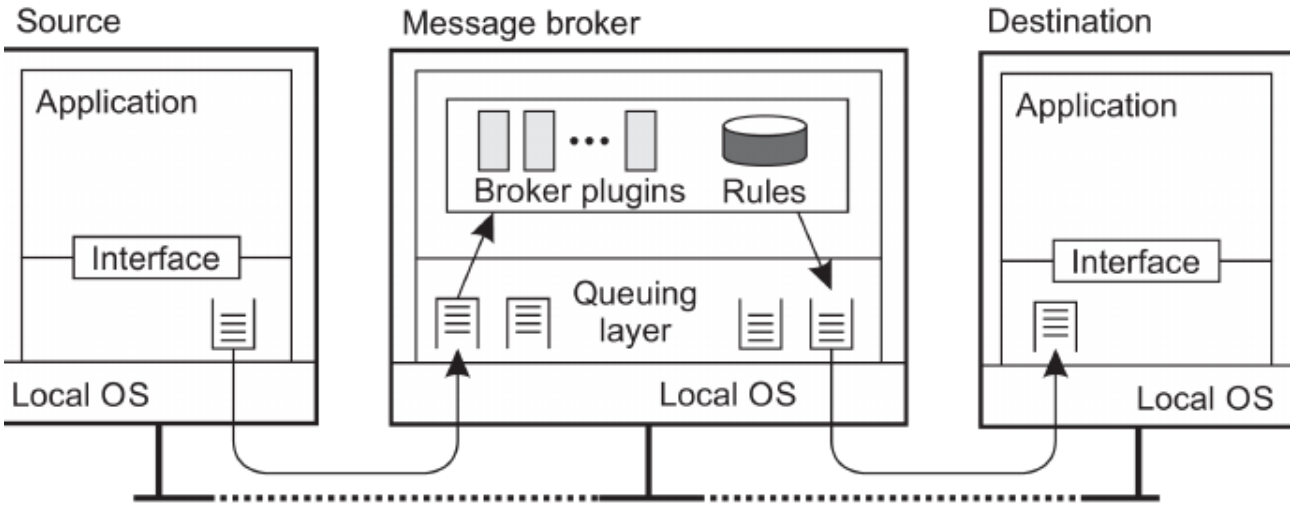
Message-Oriented Communication
Transient Messaging with Sockets: designed for low-level system, high-performance, resource-constrained communication
Socket interfaces: supported in almost all programming lanuages and OS
Client: socket, connect, send, receive, close
Server: socket, bind, listen, accept, receive, send, close

Messaging aims at high level persistent asynchornous communication:
processes send each other messages which are queued
sender need not wait for immediate reply but can do other things
middleware often ensures fault tolerance

Message-Oriented Middleware
Message-Queueing systems or MOM
well supported in large-scale systems for
- persistent but asynchornous messages
- scaleable message handling
- different communcation patterns



PUT, GET, PULL, NOTIFY



Multicast Communication
sending data to multiple receivers e.g. in peer-to-peer scenarios
- application-level multicasting
- flooding-based multicasting
- gossip-based multicasting

Multicasting: Sending a message to a group of nodes

Application-level multicasting
Based on overlay network:
- Composed of direct connections between peers
- Typically an “overlay“ network on top of a network (e.g., the Internet)
- But completely independent from physical network, due to abstraction of the TCP/IP layer
- Separate addressing scheme

Flooding-based Multicasting
- P sends a message m to each of its neighbours
- each neighbour will forward m, except to P and only if it has not seen m before
quite inefficient

Epidemic behaviour: Gossip-based Data Dissemination
- Peer P passes updates to a few neighbours Q
- If a neighbour Q_i already knows about the update, P stops contacting other servers with probility 1/k.
Gossip does not guarantee that every replica is informed! No eventual consistency!

Naming

For interaction with entities we need:
- a *name* to denote it
- an *access point* to access it
- access points a entities that are named by an *address*

- Name: set of bits/characters to identify/refeer to an entity (has no meaning, just a random string)
- Identifier: a name that uniquely identifies an entity, unique, refers to only one entity, each entity is refered to by at most one identifier
- Address: name of an accesspoint, the location of an entitiy

Flat naming: identifiers with no meaning at all
Structured naming: provides systematic way for naming
Attribute-based naming: based on attributes of an entity

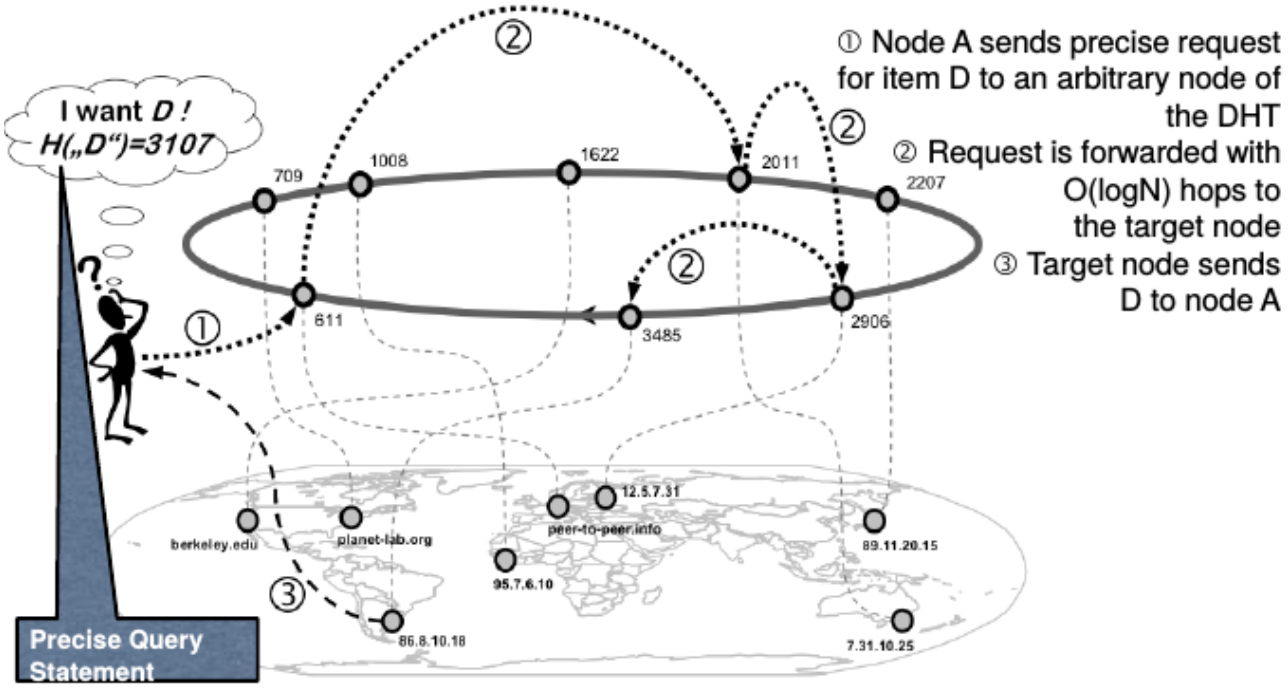
URL (Name) → DNS lookup → IP+Port (Identifier)→ ARP → Network address (Address)

Flat naming is a simple way to represent identifiers (MAC addresses)
Broadcasting (ARP)

Forward pointers: when an entity moves it leaves a pointer to its next location, dereferncing by following the chain of pointers

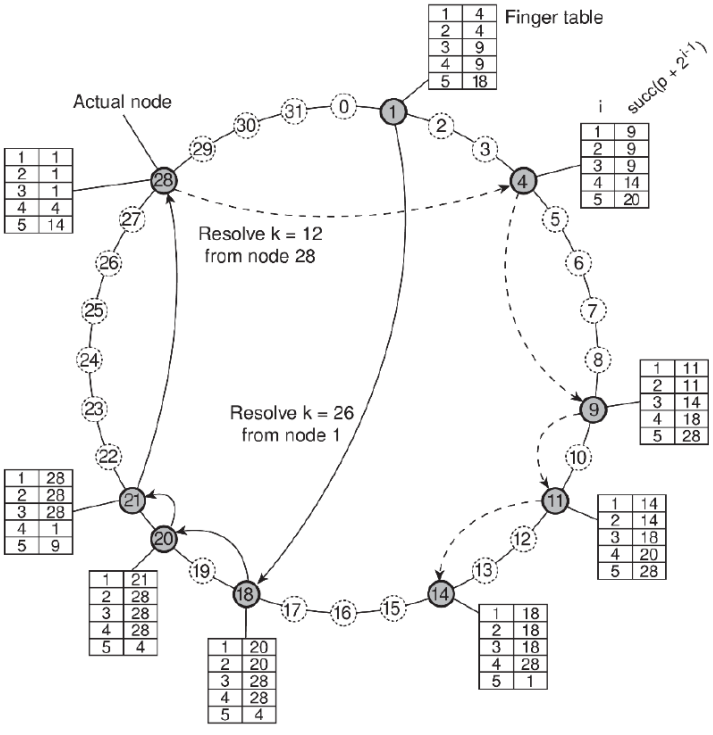
Dynamic systems:
nodes which have no centralized coordination (e.g. in an overlay network like large scale P2P systems - Chord)
nodes can join/leave/fail anytime
large number of node

Chord: well known peer to peer protocol
Circular linked list, each node has link to clockwise next node, a ring network with $0...2^m-1$ positions for nodes

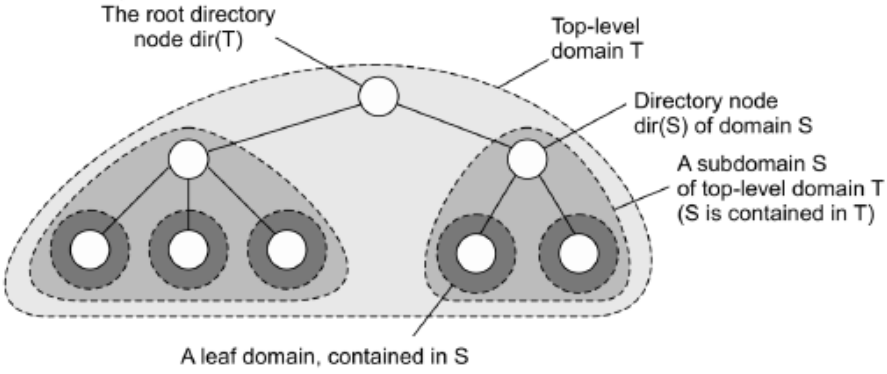


Distributed Hash Tables (DTH): m-bit is used for the keyspace for identifiers
→ Node identifier *nodeID* is one key in the keyspace
→ An entitiy is identified by a hash function k = hash(en)
→ A node with ID p is responsible for managing entities associated with a range of keys k
→ Nodes will relay messages till messages reach the right destination

Finger tables: reduce time to find responsible node using Finger Table pointing
Resolving at node p:
- keep m entries in a finger table FT
- $Ft_p[i] = successor(p+2^{i-1})$
- $q = Ft_p[j] <= k <= FT_p[j+1]$, otherwise $q = FT_p[1]$



Hirachical Location Services (HLS):
Build a large scale search tree for which the underlying network is devided into hierachical domains, each domain is represented by a sperate directory node

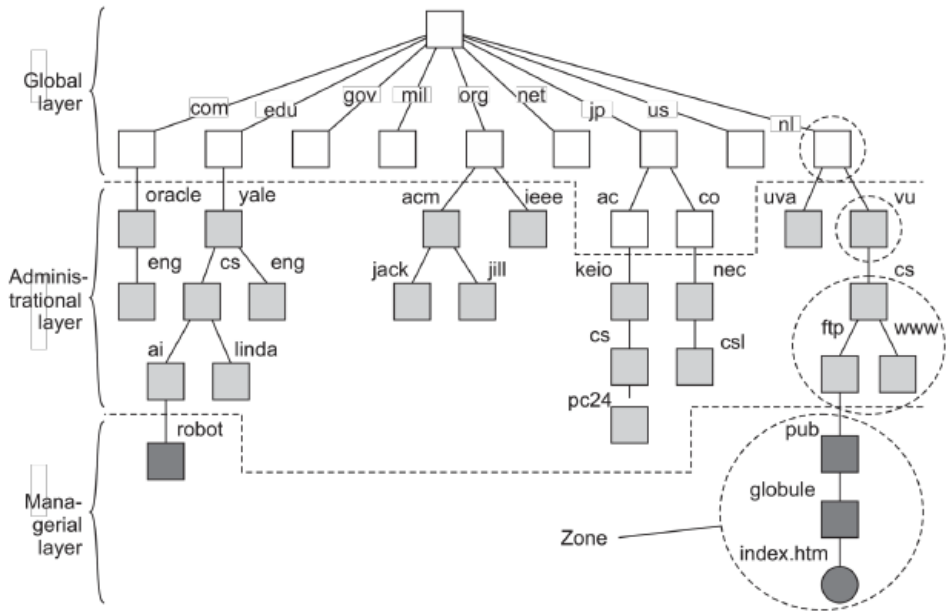


Structurd Naming
Naming desing principles:
→ Name Space contains all valid names recognized and managed by a service. A valid name might not be bound to any entity, Alias is a name that refers to another name.
→ Naming Domain: a name space with a single administrative authority which manages names for the name space
→ Name Resolution is a process to look up information/attributes from a name

Name spaces can be modeled as graphs, each leafe node represents an entity, nodes are also entities. Alias can be hard links (multiple absolute path names referring to same node) or symbolic links (leafe node storing absolute path name)

Mounting: a directory node in a remote server can be mounted into a local node (moint point)

Name Space Implementation
Distributed name management, several servers are used for manageging names in different layers
→ Global layer: root node and its closest nodes jointly managed by different administrations
→ Administrational layer: directory nodes managed within a single organization (Mid-Level directory nodes)
→ Managerial layer: Nodes typically change regularly (Low-Level directory nodes)



Iterative vs. Recursive Name Resolution

Domain Name Service (DNS)
Hierarchically organized name space with each node having exactly one incoming edge
Domain: a subtree
Domain name: A path name to a domain‘s root node
Representation: List its lables, sperated by a dot, root is expressed by a dot; e.g. flits.cs.vu.nl.
13 DNS Root Name Servers

DNS queries:
host name resolution: which is the IP of tuwien.ac.at?
email server name resolution
reverse resolution
host information
other services

Attribute-based Naming
Set of tuples (attribute, value) can be used to describe an entitiy
Attribute-based naming systems: (attribute, value) tuples for describing entities
naming resolution based on queriying mechanisms, queriying deals with the whole space
Examples: LDAP (Lightweight Directory Access Protocol), RDF (Resource Description Framework)

Directory Infromation Base: Collection of all directory entries in an LDAP service, each record is uniquely named as a sequence of naming attributes so it can be looked up

Performance, Dependability, and Fault Tolerance

Dependability:
In DS, components provide services to clients
Components may depend on other components correctness

Dependability attributes
→ Availability: Immediate readiness for correct services
→ Reliability: Continuity of correct services
→ Safety: Absence of catastrophic consequences
→ Integrity: Absence of inporper system alternation
→ Maintainability: Ability to undergo modifications

Fault (software bug, dormant)→ Error (method is called, calculation of wrong value) → Failure (if there is no mechanism to identify the error, incorrect service of the component calling the process)

Defect USB port → I/O operation with bit errors → not possible to correctly copy files to/from HD

Classes:
Development / Operational
Hardware / Software
Malicious / Accidential / Incompetence

Models:
Crash Failure: working correctly till it halts
Omission Failure: fails to repsond, recieve/send omission
Timing Failure: response outside a specified time interval
Response Failure: incorrect response, value/state-transition
Arbitrary Failure (aka Byzantibe Failure)

What to do?
Prevention
Forecasting
Tolerance ←
Removal

No Fault Tolerance without Redundancy
→ Information R. (Add a parity bit, Error Correcting Codes)
→ Time R. (Retransmission in TCP/IP)
→ Physical R. (Backup server, RAID 1)

Process Resilience
Organize several identical processes into a group
Flat group vs. Hierarchical group (with coordinator, not really fault tolerant or scalable but easy to implement)

Flat Groups
Good for fault tolerance as information exchange immediately occurs with all group members
May impose overhead as control is completely distributed, and voting needs to be carried out
Harder to implement

Groups and Failure Masking
k-fault tolerant group:
for crash/omission/timing failure: k+1 non-faulty processes are necessary
arbitrary/Byzantine failure model: 2k+1 non-faulty processes are necessary

Byzantine General Problem
Generals need to reach agreement: N processes, offering a value vi to each other
Each general builds a vector V from the values

Assumptions:
Unicast messages, Ordered message delivery, Synchronous processes, Bounded communication delay

Each process sends v_i to others, each process builds a vector V from the values.
If process is non faulty, $V[i] = v_i$
Every process passes it‘s vector V
Each process examines the i-th element of recieved vectors, if there is a majority value is put into resulting vector. No majority → element in result vector is marked unknown

So 2k+1 non faulty processes are required to detect k failures
One failure with 3 components can detect that there is a fault, but not who is causing it and what the correct values are.

Reliable Client-Server Communication

E.g. Connection between Process 2 and Process 1 fails

RPC what can go wrong?

1. Client cannot locate server

→ just report back to server, error handling by client

2. Client request is lost

→ resend request using some kind of timer

→ server needs to know difference between original and retransmission

3. Server crashes

Normal: Receive → Execute → Reply

Crash after execution: Receive → Execute → Crash (no reply)

Crash before execution: Receive → Crash (no reply)

→ client not able to tell the differences

Correct client behaviour depends on server:

- At-least-once-semantics: The Server guarantees it will carry out an operation at least once, no matter what
- At-most-once-semantics: The Server guarantees it will carry out an operation at most once.

Client can: (if not receiving reply but message that the server has rebooted)

- Always reissue a request
- Never reissue a requests
- Reissue only if no ACK has been received
- Reissue only if ACK has been received

→ 8 possible combinations of strategies, no combination of server and client stratigies will work correctly under all possible event sequences

Example print server:

(M) Send the completion message (ACK)

(P) Print the text

(C) Crash

Can occur in six different sequences:

M → P → C: crash after ACK and printing

M → C (→ not P) crash before the text was printed

P → M → C crash after sending printing and sending ACK

P → C (→ not M) printed but crash before sending ACK

C (→ not P → not M) crash before server could do anything

C (→ not M → not P) crash before server could do anything

Reissue Strategies:

Reissue strategy
Always
Never
Only when ACKed
Only when not ACKed
Client

Strategy M → P		
MPC	**MC(P)**	**C(MP)**
DUP	OK	OK
OK	ZERO	ZERO
DUP	OK	ZERO
OK	ZERO	OK
Server		

Strategy P → M		
PMC	**PC(M)**	**C(PM)**
DUP	DUP	OK
OK	OK	ZERO
DUP	OK	ZERO
OK	DUP	OK
Server		

M = Send the completion message, P = Print, C = Crash

Example: Client wrongly assumes Print hasn‘t been carried out

4. Reply massage from server is lost
How do we know that the server has not crashed?
Once again: Has the server carried out the operation?
Repeat request:
→ In case of real-world impact? Transfer from your banking account carried out twice?
No real solution! Except making operations idempotent, i.e., repeatable without any harm

5. Client crashes after request has been sent
Server executes requests anyway and sends response (called orphan computation)
Different Solutions:
1. Orphan is deleted by Client if it is received
2. Reincarnation: Client tells Servers that it has rebooted; Server deletes orphans
3. Expiration: Require computations to complete in T time units. Old ones are simply removed.

Time Synchronization and Coordination

Clock Snchronization
→ Physical Clocks
→ Logical Clocks
→ Vector Clocks

Distributed Coorination
→ Mutual exclusion algorithms
→ Leader election algorithms

Time synchronization for:
Achieving *fairness* in processing requests
Achieving *accountability* of processes
Maintaining *consistency* in processing messages
Establishing *validity* of messages

Physical Clocks
we need exact time for real-time systems, not just correct ordering of the events

UTC (Universal Coordinated Time)
→ Average of some 50 cesium clock around the world
→ Based on the number of transition per second of the cesium 133 atom
→ UTC is broadcast through short wave radio and sattelite
→ Satellites can give accuracy of about +-0.5 ms

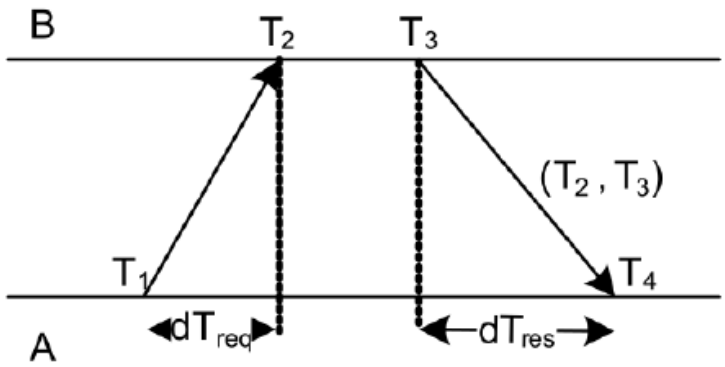
Problem:
Suppose we have a DS with a UTC-reciever somewhere in it → we still have to distribute its time to each machine.

Basic Principle:
→ Every machine has a timer that generates an interrupt H times per second
→ There is a clock in machine p that tickst on eacht timer interrupt. Denote the value of that Clock $C_p(t)$ where t is UTC time
→ Ideally we have that for each machine p, $C_p(t) = t$ or, in other words $dC/dT = 1$. unfortunatelly they drift
→ Keeping the drift under controll is achieved by synchronizating the clocks at least every delta/2roh seconds. rho is a constant given by the manufacturer e.g. 1/60 [1sec/min] and delta is the maximum time difference that the clocks should differ by.

<u>Clock Synchronization Principles</u>
→ Principle I: Every machine asks a time server for the accurate time at least once ever delta/2roh seconds (Network Time Protocol NTP)
But you need an accurate measure of round trip delay including interrupt handling and processing incoming messages.

NTP:

Network Time Protocol (NTP)



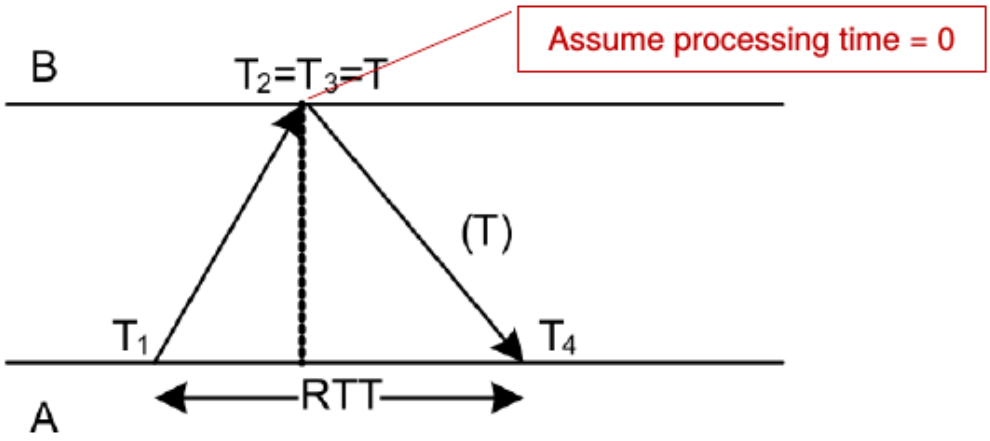
Final message to A contains timestamps T_1, T_2, T_3.

Offset Θ. Assumption: $t_{A\to B} \approx t_{B\to A}$

$\theta = T_3 + dT_{res} - T_4 \approx T_3 + RTT/2 - T_4 = ((T_2 - T_1) + (T_3 - T_4))/2$
If $\theta < 0$ the A's clock is running too fast.

Cristian‘s Algorithm

Cristian’s Algorithm



Offset Θ. Assumption: $t_{A\to B} \approx t_{B\to A}$ **and** $T_3 - T_2 = 0$

$T_3 - T_2 = 0 \Rightarrow RTT = T_4 - T_1$

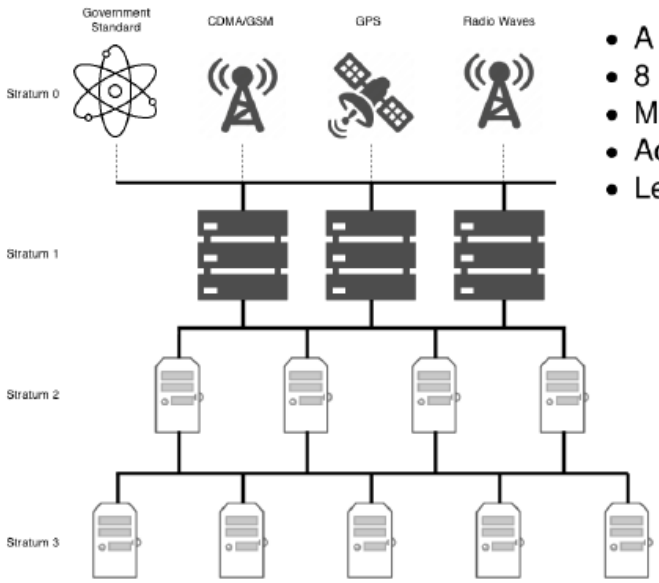
$\theta = T + RTT/2 - T_4 = T - (T_1 + T_4)/2$

Collect multiple measurements, take the one with smallest RTT.

The smaller the RTT, the more likely it is that $t_{A\to B} \approx t_{B\to A}$.

Symmetric Time Propagation in NTP

Stratum Approach

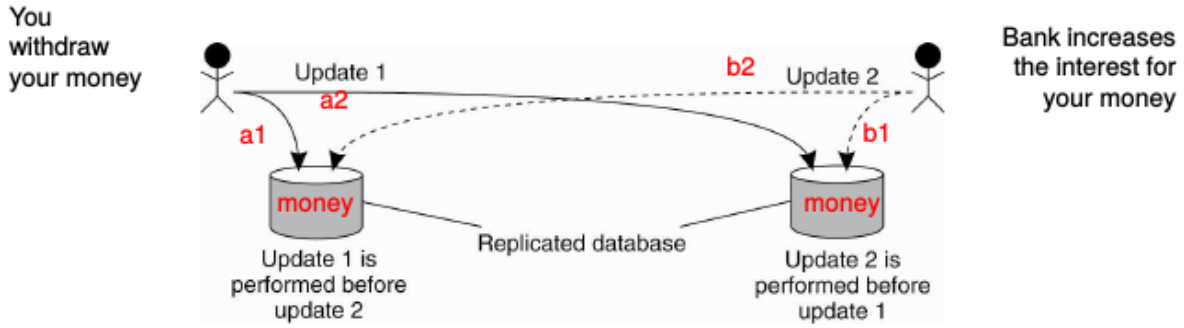


- A obtains time-offset from B and vice versa
- 8 measurements
- More accurate server determines time
- Accuracy determined by stratum level
- Less accurate server becomes stratum k+1

1-50ms worldwide accuracy

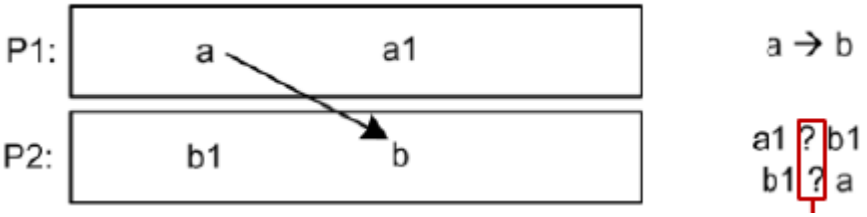
→ Principle II
When no reliable time source: Let a logical time server scan all machines periodically and calculate an average, and inform remaining machines how they should adjust their time relative to server‘s time
You will probably get every machine in sync, you don‘t even need to propagate UTC time.

In many occasions the absolute time (neither local nor global) is really important as long as we can establish the **order of events**, e.g.,:



We just need to establish (agree upon) what happened before:
(a1,a2) before **(b1,b2)** or vice versa. ⟶ Logical clocks

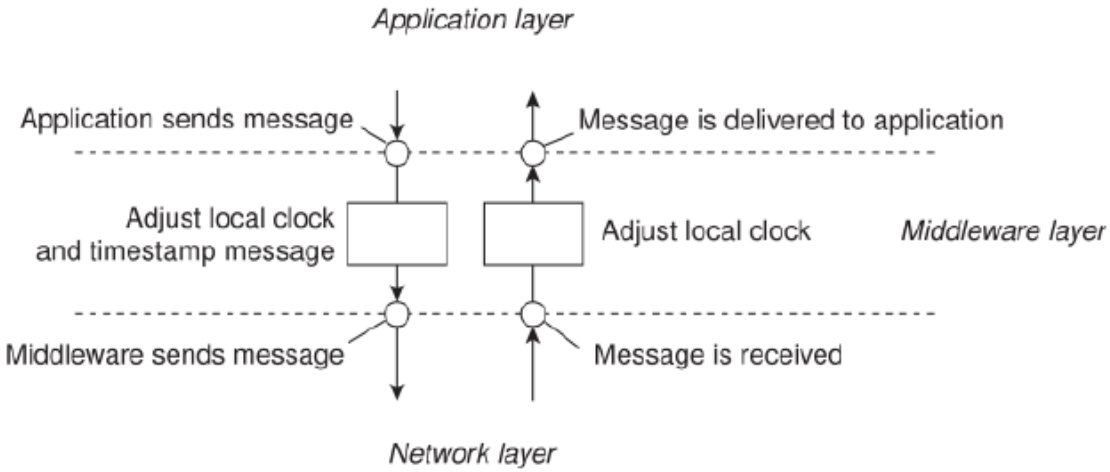
Happened-before relationship
If a and b are two events in the same process and a comes before b then a → b
If a is the sending of a message and b is the receipt of that message then a → b
If a → b and b →c then a → c
This introduces a partial ordering of events in a system with concurrently operating processess.
a→b possible that a affects b, concurrent events: a || b ↔ (a /- > b) and (b /- > a)



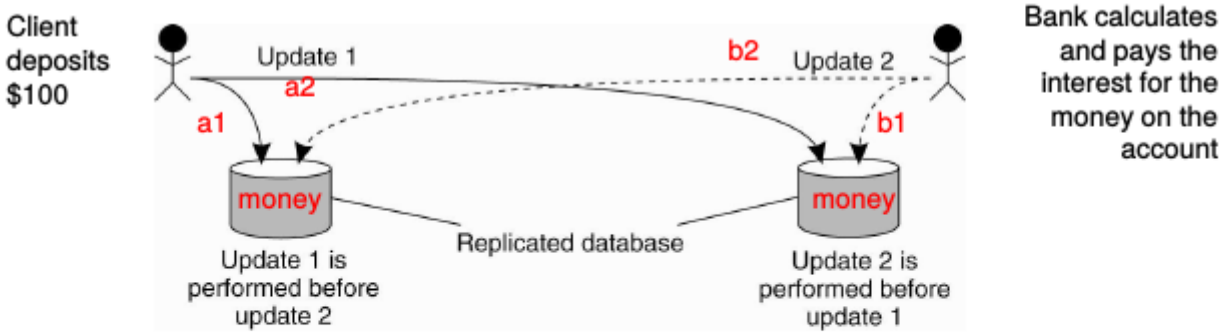
Lamport‘s logical clocks:
How do we acheive that distributed systems globally behaves consistently with the happened befor relation?
Attach a timestamp C(e) to each event e satisfying the following properties:
→ If a and b are two events in the same process and a → b then we demand that C(a) < C(b)
→ If a corresponds to sending a message and b to the reciept of that message, than also C(a) < C(b)

For attaching the timestamp you need to maintain a consistent set of logical clocks, one per process.

Adjustments take place in the middleware layer



Problem: we sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere: P1 adds $100 to an account (initial value 1000), P2 increments account by 1%
There are two replicas



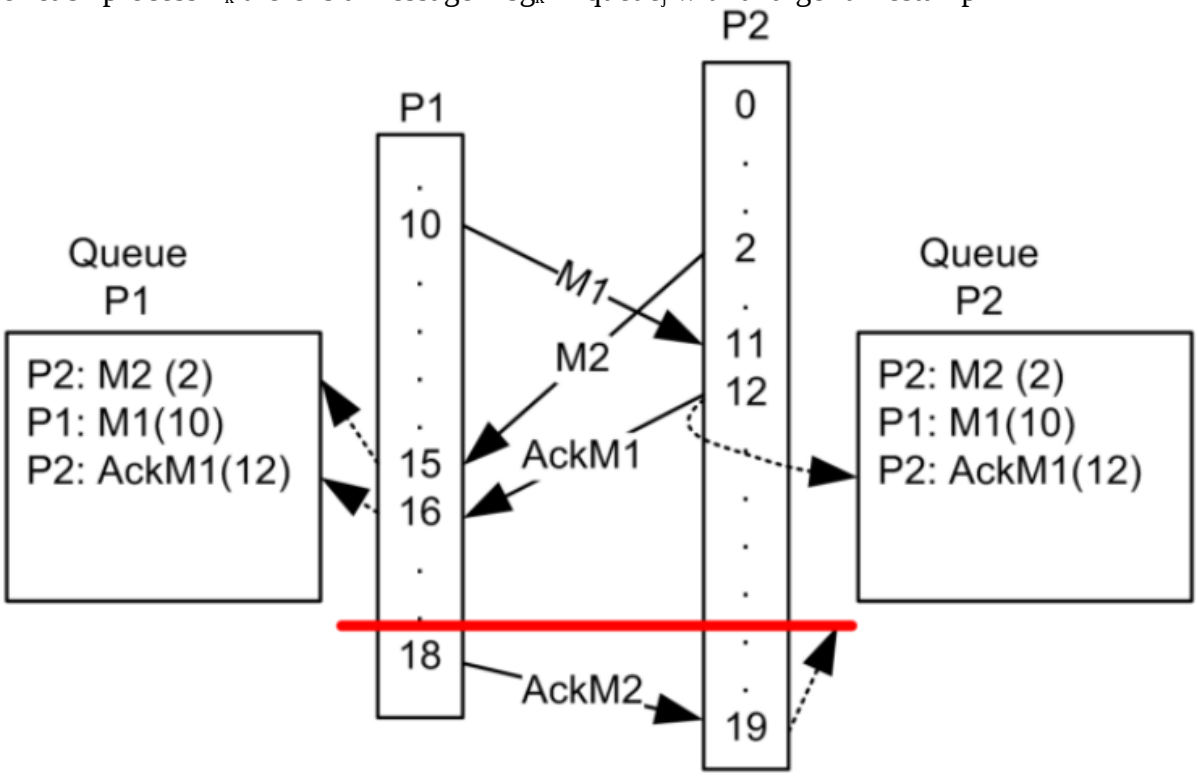
Result

In absence of proper synchronization:
replica #1 ← $1.111,00 while replica #2 ← $1.110,00

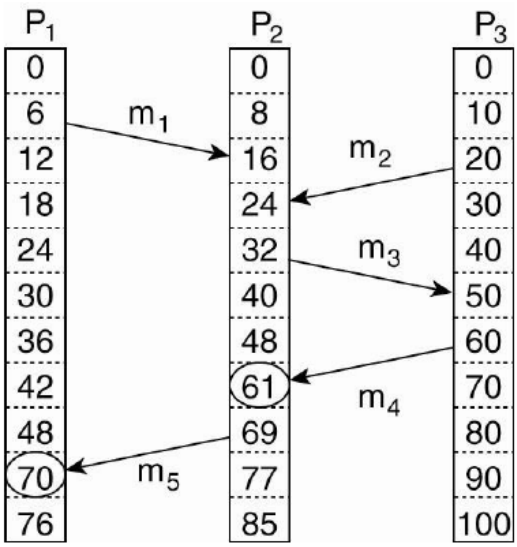
Application: Totally ordered multicast

Process Pi sends timestamped message msg_i to all others. The message itself is put in a local $queue_i$.
Local Queues are priority queues → dynamically sorted on sender timestamp.
Any incoming message at P_j is quede in $queue_j$, according to its timestamp and acknowledged to every other process.

P_j passes a massge msg_i to its application if:
(1) msg_i is at the head of $queue_j$
(2) for each process P_k there is a message msg_k in $queue_j$ with a larger timestamp



Limitations of Lamport's clocks



recv(m4) < send(m5):
Maybe m5 depends on m4 (**causality**)

recv (m1) < send (m2):
We do not know their relationship just by comparing clock values!

$C(a) < C(b) \not\Rightarrow a \rightarrow b$
We miss causality information!

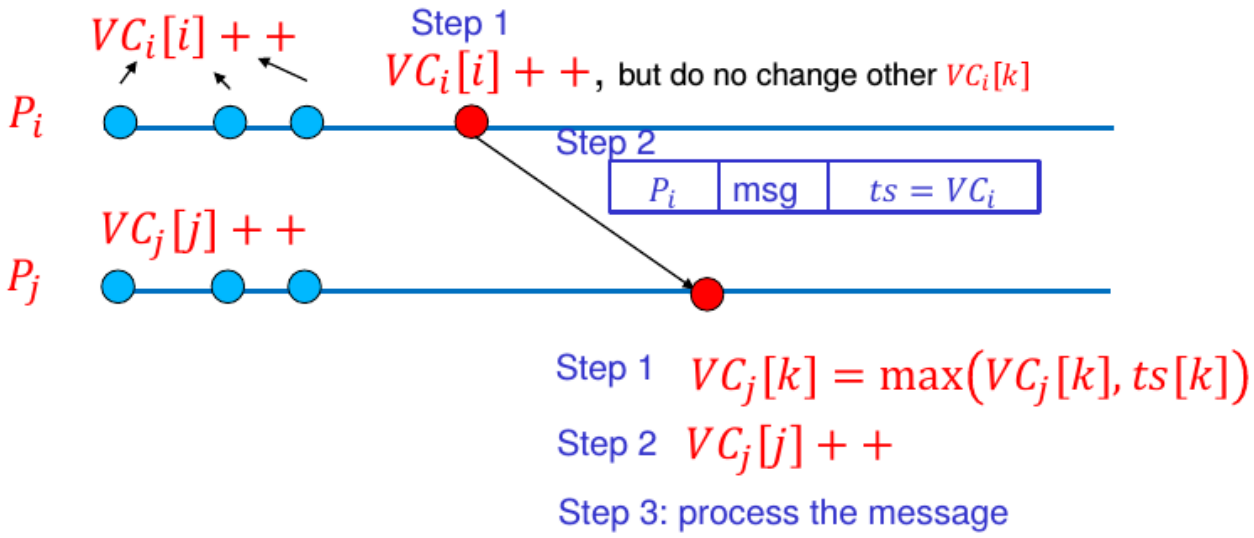
Vector Clocks

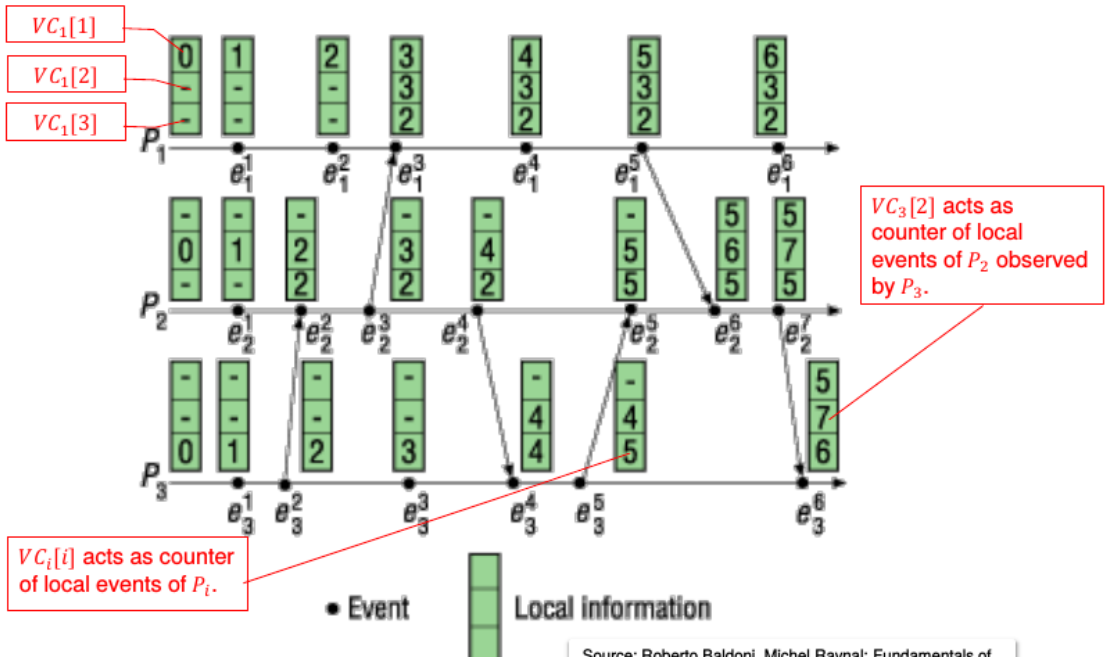
A vector clock (VC) allows us to know that: if VC(a) < VC(b) then a causally preceeds b

Each process P_i maintains a vector clock VC_i where:

$VC_i[i]$ is the number of events happend in P_i

$VC_i[j] = k$ means that P_i knows that k events that might have causal relation with P_i previously occured in P_j





Mutual Exclusion

A number of processes in a distributed system want exclusive access to some resource.

General Approaches:

→ Token-based

→ Permission-based

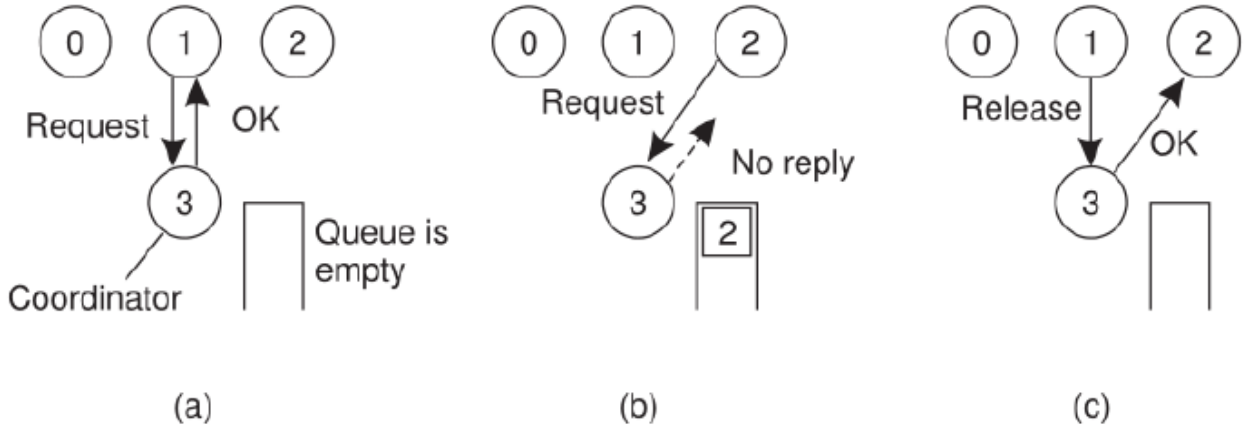
Implementations:

→ Centralized

→ Decentralized

→ Distributed

→ Token-ring



Coordinator as a single point of failure: requesting process cannot distinguish dead coordinator from permission denied. Performace bottleneck.

Decentralized Mutual Exclusion

Assume every resource is replicated n times, with each replica having its own coordinator → access requires a majority vote from m > n/2 coordinators. A coordinator always responds immediately to a request.

Assumptions: When a coordinator craches it will recover quickly but will have forgotten about permissions it has granted.

Typically quite robust, BUT for insufficient votes (less than m) back-off temporarily. With many competing nodes none gets sufficient votes. (There ar solutions for this)

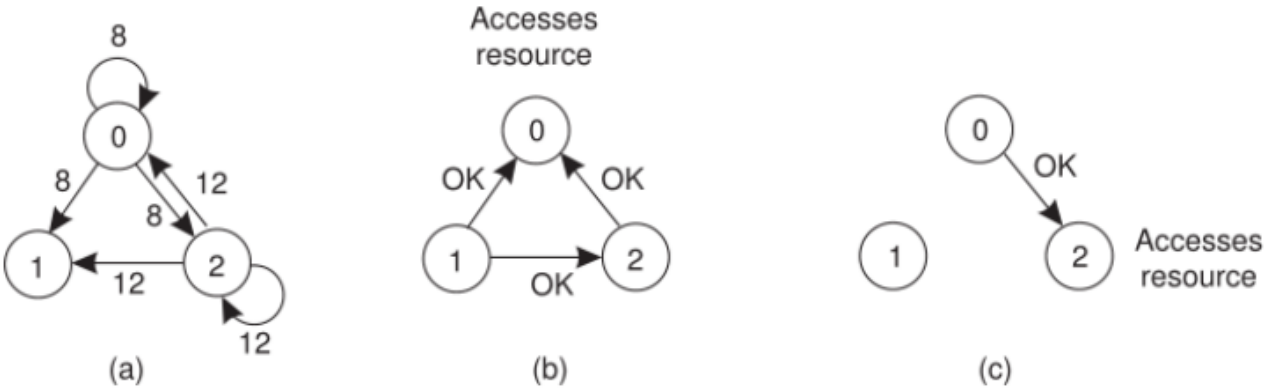
Distributed Mutual Exclusion

Assumes totally ordered events (Lamport), requester sends the request to all other processess. Receivers algorithm:

→ If the receiver has no interest in the shared resource sends OK in response.

→ If the receiver is working with the resource, doesn‘t answer but queues the request.

→ If the receiver wants to access the resource, it compares his timestamp with the received request. The lower wins. If he wins, he queues the request and does nothing. Otherwise he replies OK to the sender.



Issues:
Less eficient: 2(n-1) messages per entry
No single point of failure BUT worse N points of failures
A failing process is interpreted as „permission denied“, Blocks all subsequent requests (n times higher failure likelihood)
→ Introduce Rejection messages
In any case needs group membership management or multicasting

Token Ring mutual exclusion
Organize processes in a logical ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to)

Issues:
→ Token circulates when no one needs accesss
→ Lost token detection: need to akn token receipt
→ Unresponsive process is skipped
→ Requires everyone maintaining ring topology

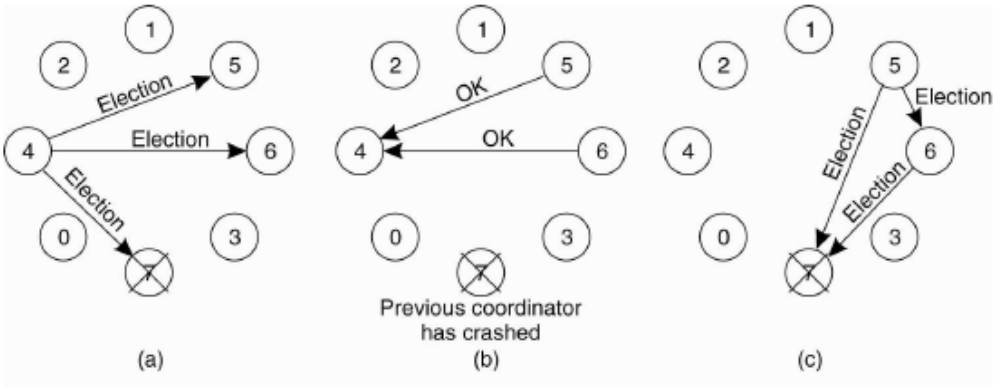
Election Algorithms
In many DS we need a leader/coordinator for all kinds of duties, selected from the set of all processess.
Often the coordinator is pre-chosen or implicitly determined (e.g. file servers). This leads to centralized solutions, also single point of failure.

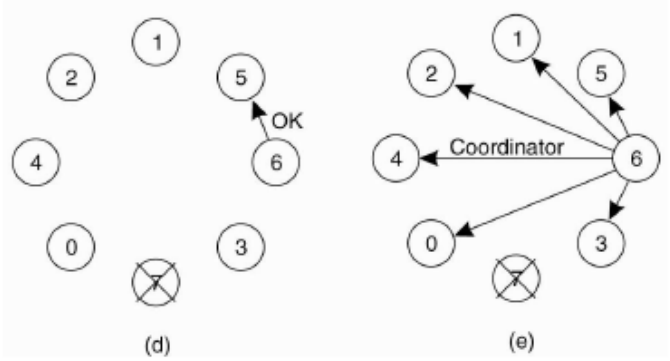
How can the coordinator be dynamically selected through consensus of processes?
→ Election Algorithms

Assumptions:
- Procesees are uniquely identifiable
- As a general rule the active process with the highest number is selected at the end of the algorithm.
- Each process knows in advance all other processes, their Ids and group memberships but not if they are active in the moment.
- Anyone may notice failure/absence of coordinator
- Anyone might trigger an election

Bully Algorithm

Process P sends an „Election“ msg to all processes with higher number.
If nobody responds, P becomes the coordinator and notifies everyone else.
If someone with a higher number responds (Q) P ends and Q starts executing from step 1.





: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

; 2017

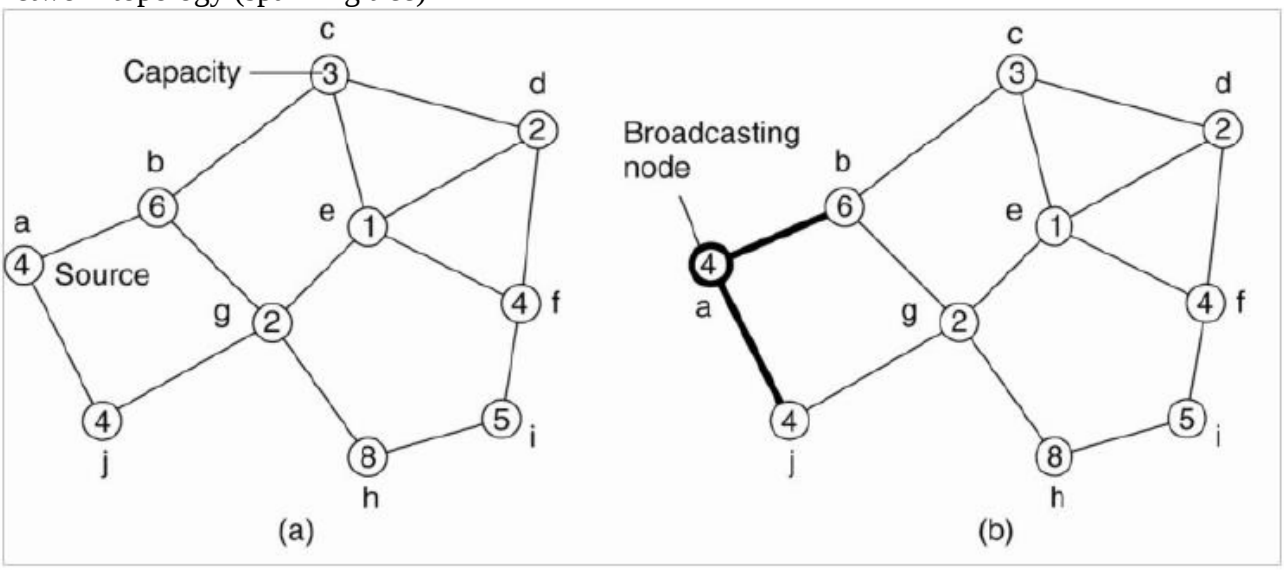
Election in a ring

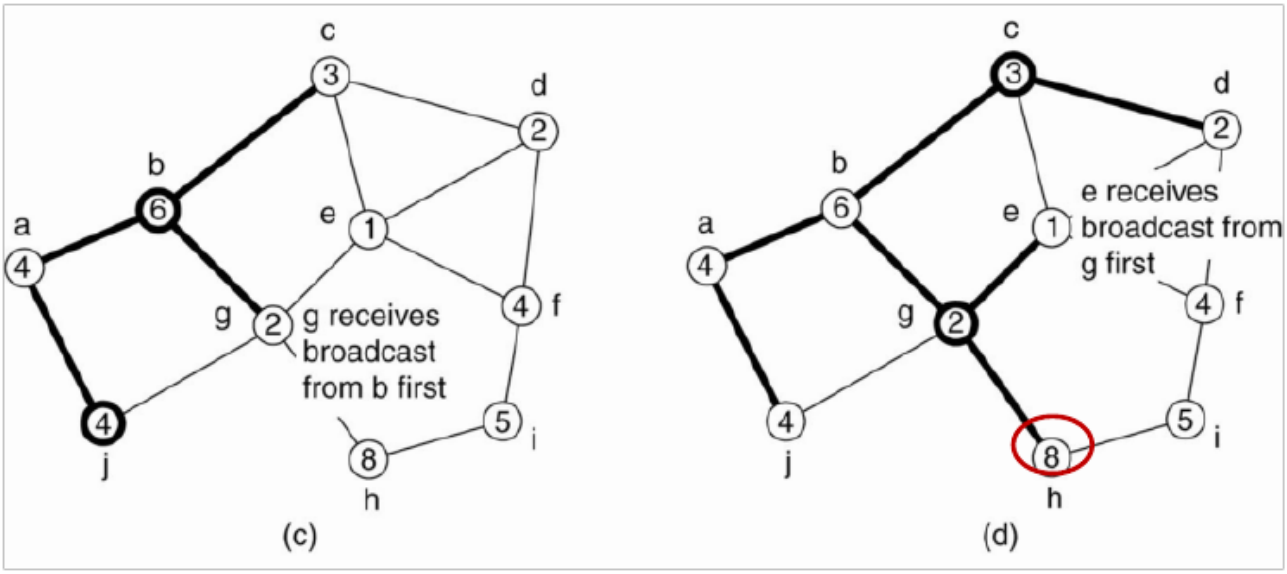
Process organized in a logical ring, ordered by process numbers.
1) Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
2) When the message is passed on, the sender adds itself to a list in this message. When it gets back to the initiator, everyone has had a chance to make its presence known.
3) The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority elects himself as coordinator.

Election in Wireless Environments

Traditional election algorithms are not appropriate for ad-hoc wireless networks.
- They assume that the message transmission is reliable
- and that network topology does not change when the nodes are active/reachable.
- furthermore, the nodes (processors) know the topology and each other node

In wireless ad-hoc networks the selection of coordinator implies also establishing the current network topology (spanning tree)





Consistency and Replication I

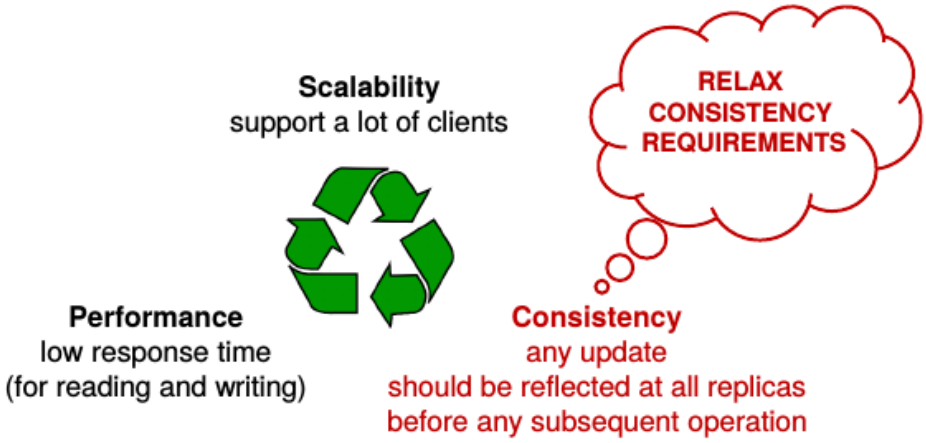
Replication is the process of maintaining several copies of an data item at different locations.
Consistency is the process of keeping data item copies the same when changes occur.

Reasons for replication:
-Performance and scalability
i) scale in numbers (more replicas can serve mor client requests) and ii) scale in geographic/topological complexity (replicas close to the client imporve response time)

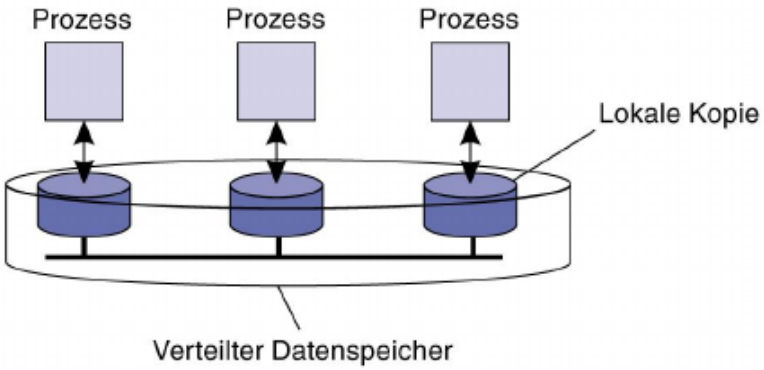
-Fault tolerance
i) switch-over in case of failures
ii) protection against corrupted data (majority vote)

Drawbacks
- keeping replicas up to date consumes bandwith
- updates to replicas are not immediately porpagated (stale data)

Performance vs. Scalability Tradeoff
We want to improve all three properties, but in reality can only pick two at expense of the third.



Distributed Datastore Model
Conceptual model representing different real distributed storage systems (dist. shared memory, distributed file system, distributed database).
Processes have access to the entire data store through the local copies.



Consistency Models

- A contract between a (distributed) data store and processes, in which the data store specifies precisely what the results of read and write operations are in the presence of concurrency.
- Fundamental expectation: A process that performs a read of element X from the local copy expects to see the latest value of X that was written by any other process into its local copy.
- In practice, we can relax this expectation, as long as the functioning of the overall system is not endangered.

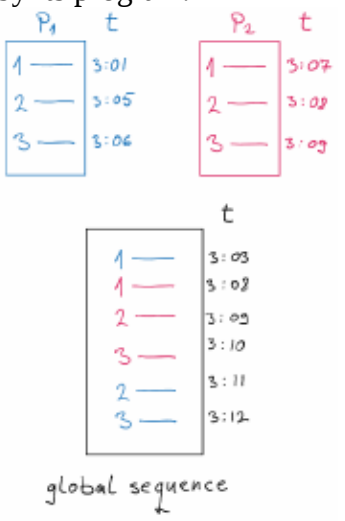
Two approaches:
- Data-centric: guarantee consistency of the entire data store
- Client-centric: guarantee consistency of data owned by a single client when the client accesses the data through different nodes

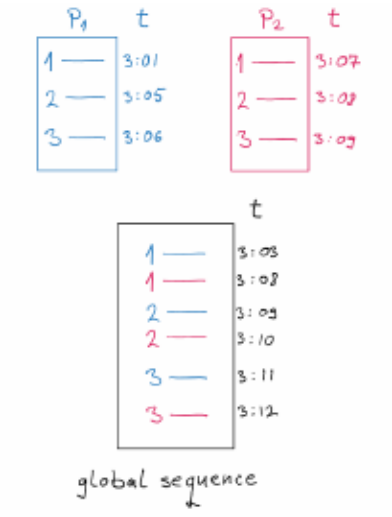
- We use the following notation:

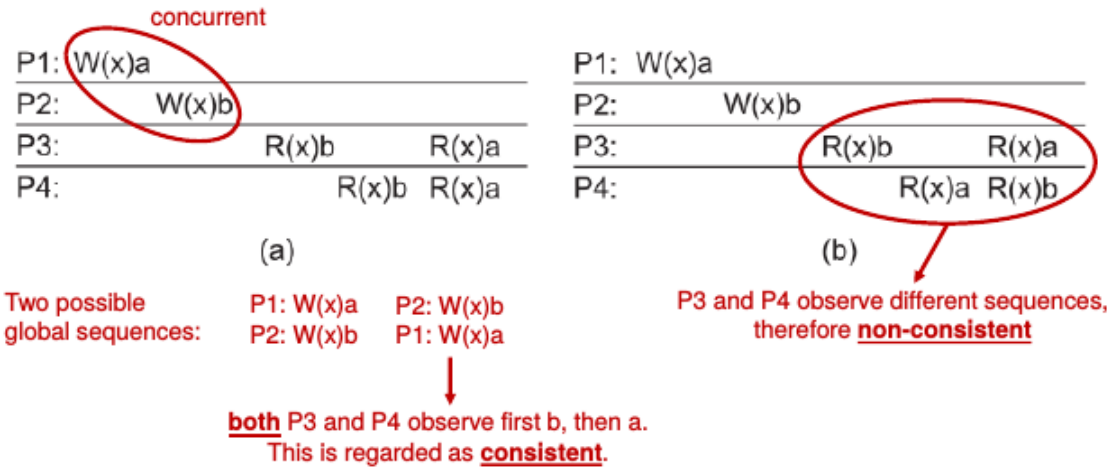
data-centric	P_n: R(x)a	Prozess P_n liest die Variable x und erhält den Wert a
	P_n: W(x)b	Prozess P_n überschreibt die Variable x mit dem Wert b
client-centric	L_n: R(x)a	Von der lokalen Kopie L_n wird aus der Variable x der Wert a ausgelesen
	L_n: W(x)b	Auf der lokalen Kopie L_n wird die Variable x mit dem Wert b überschrieben
	L_n: WS(x)	Auf der lokalen Kopie L_n wird ein Satz von Schreiboperationen auf x ausgeführt

Data-Centric Consistency Models

→ Sequential Consistency
The result of any execution is the same as if the operation of all processess were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.

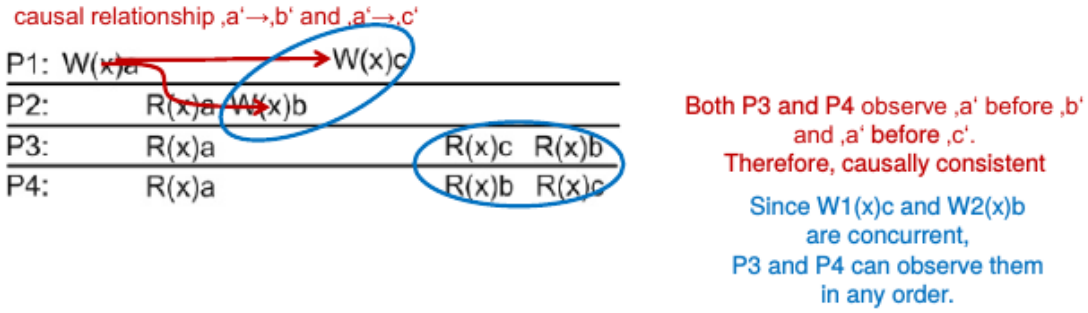






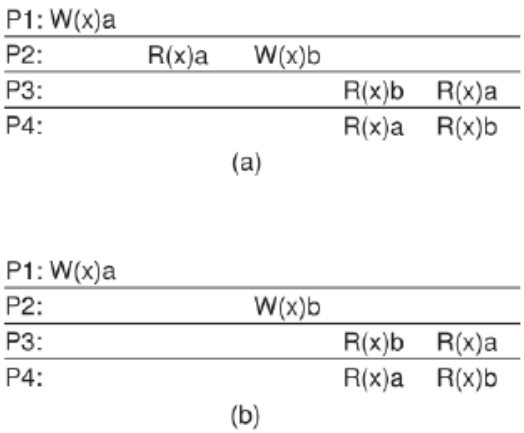
→ Causal Consistency

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



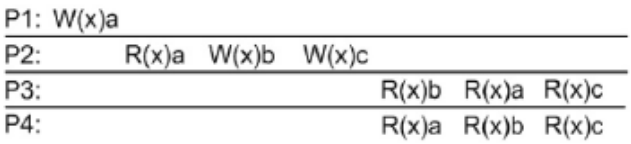
But not sequentially consistent since P3 and P4 observe different sequences of b and c.

More Examples:

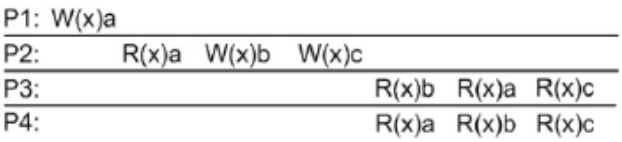


(a) causally inconsistent, (b) consistent

→ FIFO Consistency
Writes done by a process P1 must be read by any other process P2 in the order in which P1 wrote them.
Within a process, ordering between any two reads that are dependent on writes from two different originating processes can be arbitrary.



- In P1 only ‚a' is written
- In P2 ‚b' written before ‚c'
- So, in any process, ‚b' must be read before ‚c'
- ‚a' can be read independently of ‚b-c'
- So, **FIFO-consistent**!



Note, NOT causally consistent

Because causality relationship a→b→c is not properly observed by P3.

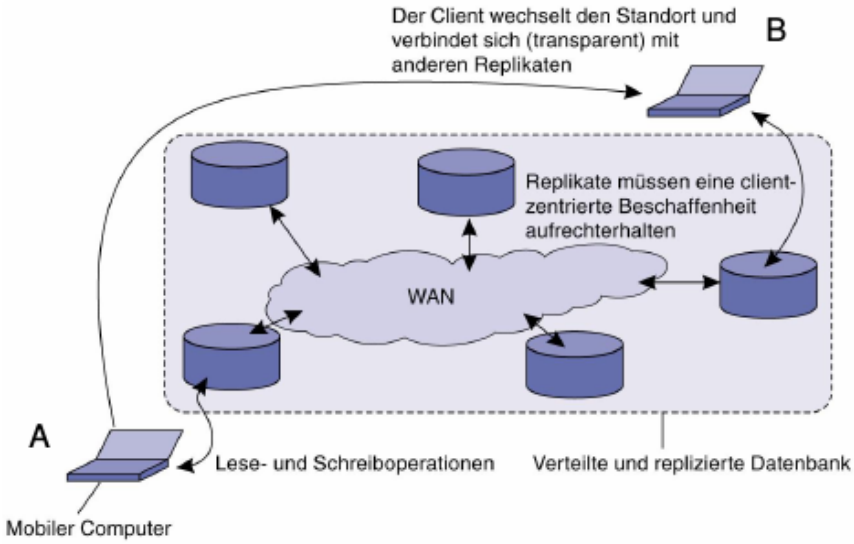
Client-Centric Consistency Models

→ Principle of Eventual Consistency:
So far (in Data-centric Consistency) we assumed that the system has to manage consistency during concurrent updates (writes).
Maintaining consistency is very expensive, as practical implementation requires synchronisation mechanisms.
But if the real system that we want to model is such that only a single entity can at a given time perform writes, then there can be no concurrent updates, and the consistency model can be relaxed.

In this case we talk about eventual consistency. Key porperty of eventual consistency:
After a write, the replicas will gradually become consistent. (It might take some time, but they will guaranteedly get updated at some point)

Many systems are not characterized by concurrent write and therefore can function well with eventual consistency. Example: Only webmaster updates static files, but many clients read them.

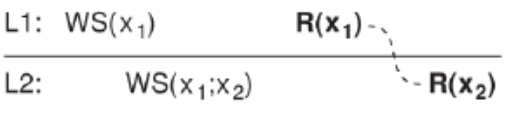
But there is a problem:

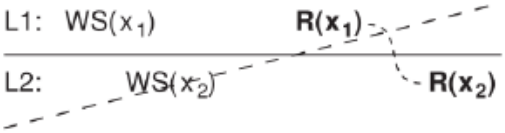


This problem can be alleviated by client-centric consistency models. They provide guarantees for a single client assessing a distributed file storage that his accesses to the data he owns will be consistent.

- $WS(x_i[t])$ is the set of write operations (at L_i) that lead to version x_i of x (at time t)
- $WS(x_i[t_1]; x_j[t_2])$ indicates that $WS(x_i[t_1])$ is subset of $WS(x_j[t_2])$.
- Note: Parameter t is normally omitted from figures.

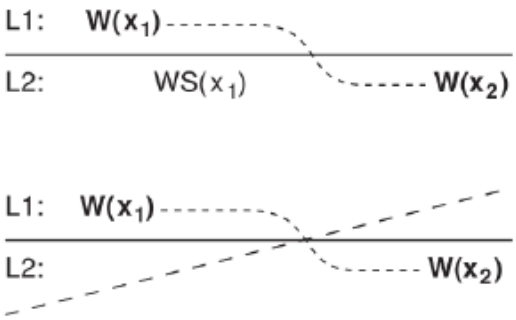
→ Monotonic Read Consistency
If a process reads the value of a data item x, any successive read operation on x by that process will always return that same or a more recent value.



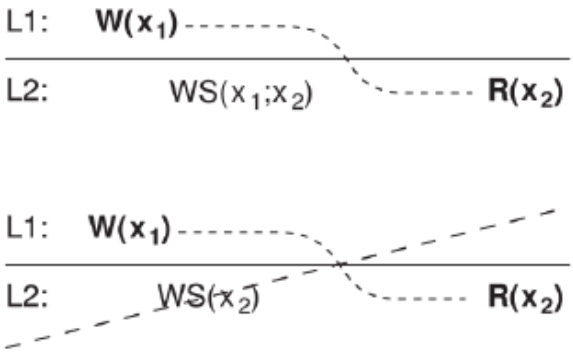


Automatically reading your personal calendar updates from different servers. Monotonic Reads guarantees that the user sees all updates, no matter from which server the automatic reading takes place.

→ Monotonic Write Consistency
A write operation by a process on a data item x is completed before any successive write operation on x by the same process. Maintaining versions of replicated files in the correct order everwhere (propagate the previous version to the server where the newest version is installed).

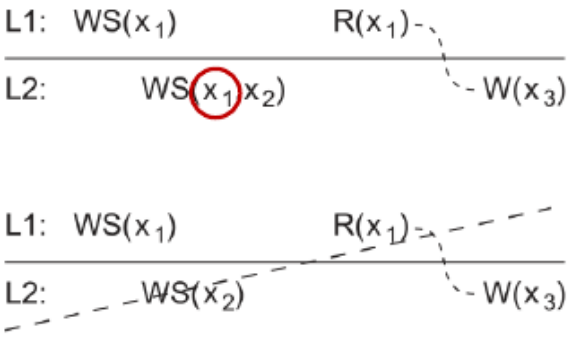


→ Read your Writes Consistency
The effect of a write operation by a process on data item x will be always seen by a successive read operation on x by the same process.



Example: Updating your web page and guaranteeing that your Web browser shows the newest version instead of its chached copy.

→ Writes follow Reads Consistency
A write operation by a process on a data item x following a previous read operation on x by the same process is guaranteed to take place on the same or a more recent value of x that was read.



Example: See reactions to posted articels only if you have the original posting (a read pulles in the corresponding write operation).

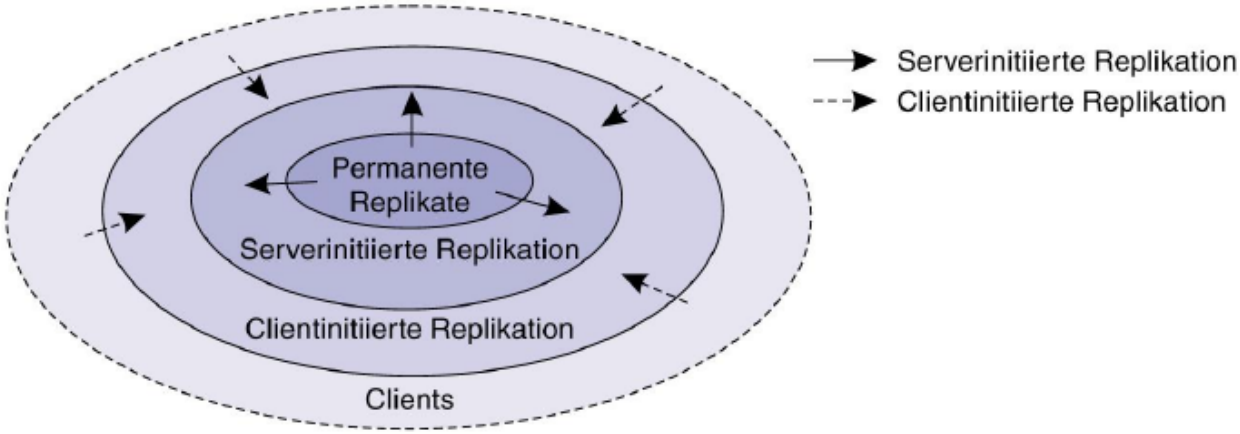
Consistency and Replication II

We learned, that there are different ways in which object replicas can be kept consistent (consistency models). But what is the best strategy of placing the replicas to make an efficent use of replcation and be able to maintain consistency as cheaply as possible?

From consistency models to replica management

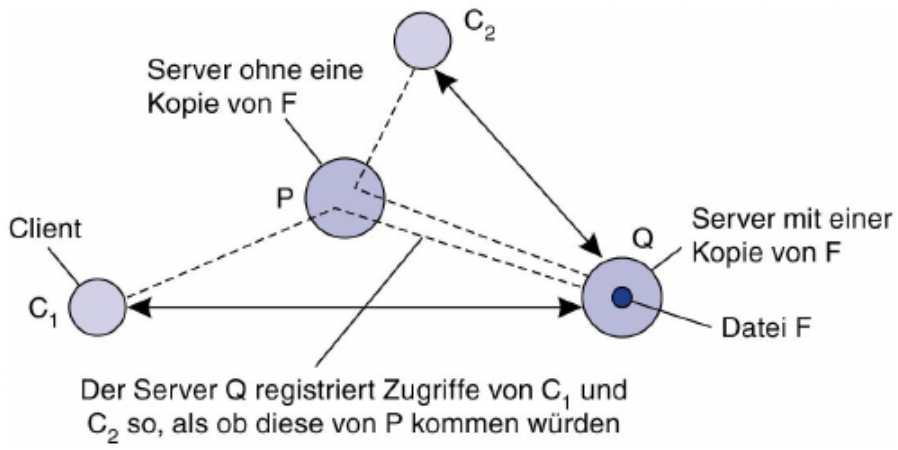
→ If no updates to the object → no consistency problem
→ If access-to-update ratio is high replication pays of
→ If update-to-access ratio is high many updates are never read
→ Ideally, update the replicas thar are going to be accessed
As a general rule we try to keep a replica close to it‘s clients

Challanges
→ Replica server placement, often a management or commercial issue
→ Content replication and placement
→ Content distribution:
state vs. operation
push vs. pull vs. lease
blocking vs. non-blocking (eager vs. lazy)
unicast vs. multicast (group communication)



Permanent Replicas
Basic replication
→ often initial distribution of data
→ supported by the (storage) system itself
Examples:
→ replicated website servers for load balancing (client sees no difference)
→ mirroring (client is aware and looks for specific server)
→ Usually not so many copies

Server initiated replicas
The server decides alone when and where more replicas should be created
Many different and complex algorithms
→ using monitoring and statistics od accesses to dertermine optimal placing
Basis for Web Hosting and Cloud services



Often used by Web Hosting companies
→ Each server knows what would be the closest server for each Client request.
→ Each server keeps track of access counts per file, aggregated by considering server closest to requesting clients $cnt_Q(P,F)$
→ Each server has a replication threshold R and deletion threshold D
Number of accesses drops below threshold D ⇒ drop file
Number of accesses exceeds threshold R ⇒ replicate file
Number of access between D and R ⇒ migrate file
→ Whether migration/replication occurs depends on the cost of operation
e.g., Is the target server‘s overall load too high? Does it have enough disk space?

Client-Initiated replicas
(Client) cache – local storage facility managed and used by client to temporarily store data and improve access times.
→ especially useful if datas read-to-write ratio is high, chache hit if data is found in cache
Data is fetched in cache upon clients requeste
Data stored for limited time only
Usually located on client machine or nearby (if shared by multiple clients)

Hardwar caches used in Modern CPUs, shared memory multiprocessor systems
Software-based solutions for middleware based distributed systems, we briefly discuss protocols later

Content Distribution
Discussing general strategies and models of distributing content (files, DB, memory elements) in distributed systems

Consider only a client-server combination:
1. Invalidation: propagate only notification of update, no actual data sent → low bandwith, good for low read-to-write ratios, when many updates happen that are only once in a while read-to-write
2. Passive replication: transfer data from one copy to another: makes sense when read-to-write ratio is high, consumes bandwith, multiple changes can be bundled toghether and sent as a single update
3. Active replication: Do not transfer the actual data but the operations that lead to the new values, send operations and parameters e.g. invert_matrix(A)
→ no single approach is the best, but depends highly on available bandwith and read-to-write ratio at replicas

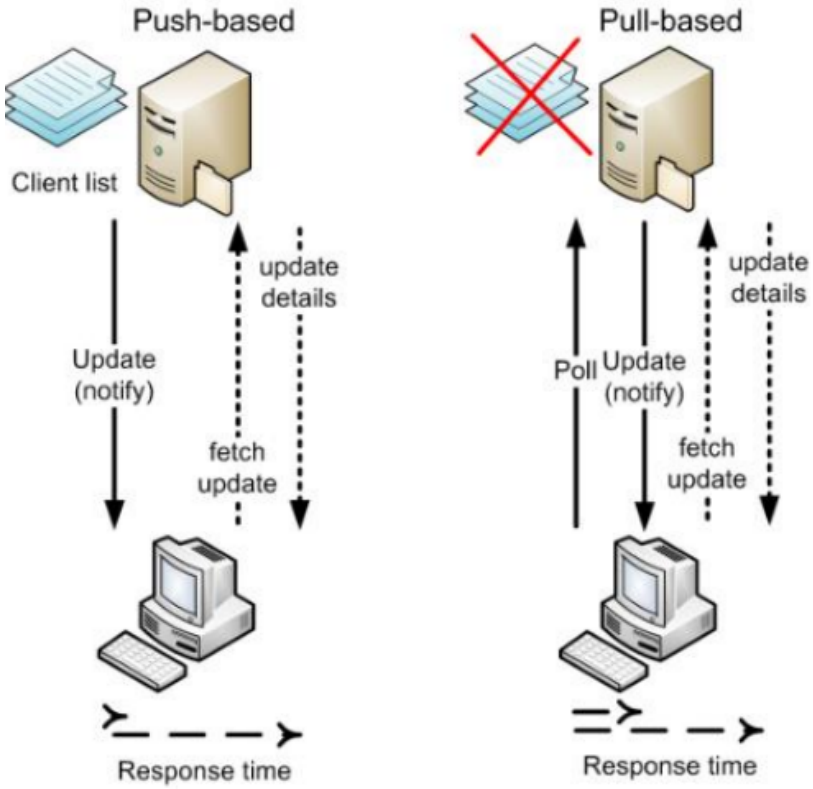
Push vs. Pull

Push-based (server-based) protocols
→ Updates are propagated to other replicas (clients) without them asking for updates, reaching consistency faster
→ If server wants to update all clients at once, this is not efficient (scalable): takes time, every client can fail, unless efficient multicast implementation is available. e.g., in LAN
→ Alternatively, update only clients that need the information: Server needs to know what each client has ⇒ stateful server, limited scalability and less fault tolerant

Pull based (client-based) protocols
→ Client polls server to check if updates are available, then asks for update
→ Response time for client increases in case of a cache miss
→ Most often used by client caches

Comparision:

Thema	Push-basiert	Pull-basiert
Zustand auf dem Server	Auflistung der Client-Replikate und Clientcaches	Keine
Gesendete Nachrichten	Aktualisieren (sowie später möglicherweise Abrufen der Aktualisierung)	Ständiges Abfragen und Aktualisieren
Antwortzeit für den Client	Unmittelbar (oder Zeitaufwand für den Abruf der Aktualisierung)	Zeitaufwand für den Abruf der Aktualisierung



Content Distribution – Leasing

Lease – A contract in which the server promises to push updates to the client until the lease expires.
→ A hybrid solution to dynamically switch between pulling and pushing.
→ When should a lease expire: Depends on system‘s behavior (adaptive leases).
→ By choosing different lease durations, we want to minimize the load on the server, server state and speed up the updating of clients (higher consistency level).

Lease Expiry
→ Age-based leases: An object that hasn‘t changes for a long time will not change in the near future so provide a long-lasting lease.
→ Renewal-frequency based lesases: The more often a client requests a specific object the longer the expiration time for that client (for that object) will be.
→ State-bases leases: The more loaded a server is the shorter the expiration times become.

What did we achieve with this?
→ Server‘s state is smaller, limited to clients and data under lease
→ Only those clients that actually need a higher consistency level (achieved by pushes) apply for a lease, so server can dedicate its resources to hem at the time.
→ Better utilization of server and network. Less unnecessary communication and data transfers.

Content Distribution – Blocking vs. Non-Blocking

A client wants to push an update to other clients via a server:
→ Synchronous (blocking, eager): All replicas are updated immediately, then reply to originating client.
→ Asynchronous (non-blocking, lazy): Update is applied to one copy then reply to client, then propagation to other replicas afterwards.

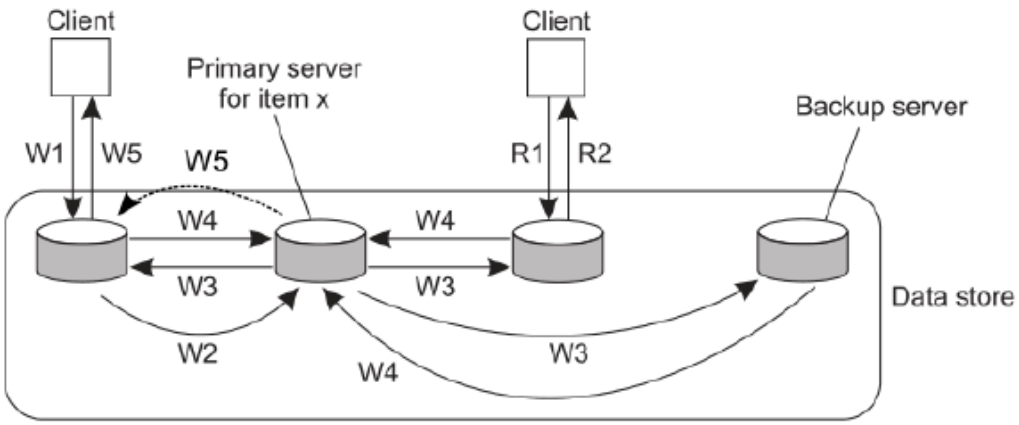
Consistency Protocols

Describe the implementation of a specific consistencs model.

Primary-based Protocols
→ Primary backupd protocol
→ Primary backup with local writes

Replicated-Write Protocols
→ Quorum based protocols

Primary-backup protocol:



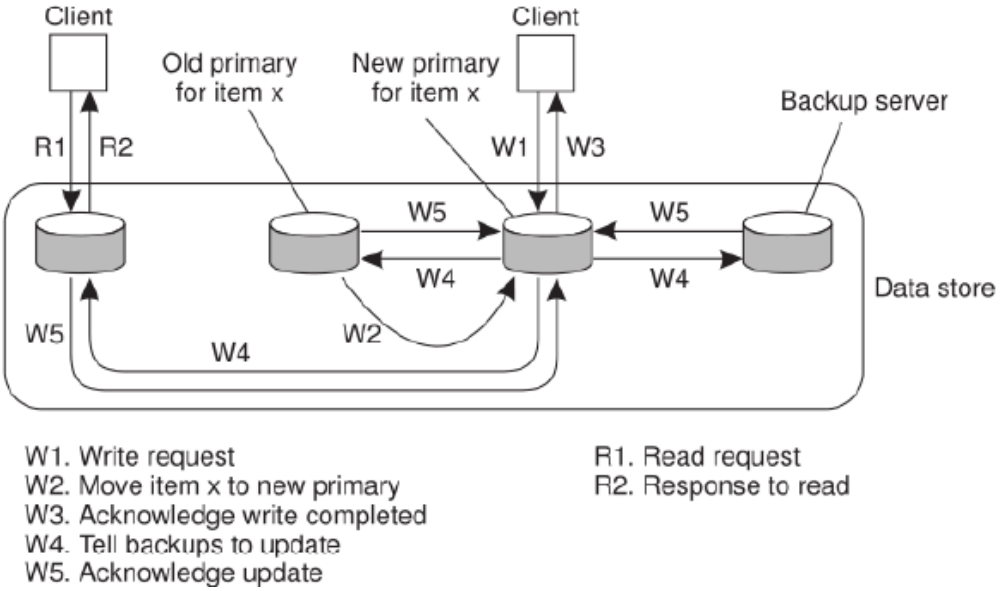
W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Implements the sequential consistency model.
→ All write operations are ordered through the primarŷ and delivered to the remaining servers.
→ Reading the local copy yields the most up-to-date value. Changes are atomic. No inconsistencies.
→ Reading fast. Writing is slow (blocking operation).
→ If one node not available, not possible to perform a write ⇒ not resilient against network or node failure

A non-blocking (asynchronous) scheme is also possible.
→ ACK as soon as Primary got the update
→ Speeds up the writing
→ Resilient against node and link failure
→ But, data inconsistencies can occur
→ a local read does not always return the most up-to-date value.

Primary-backup protocol with local writes:

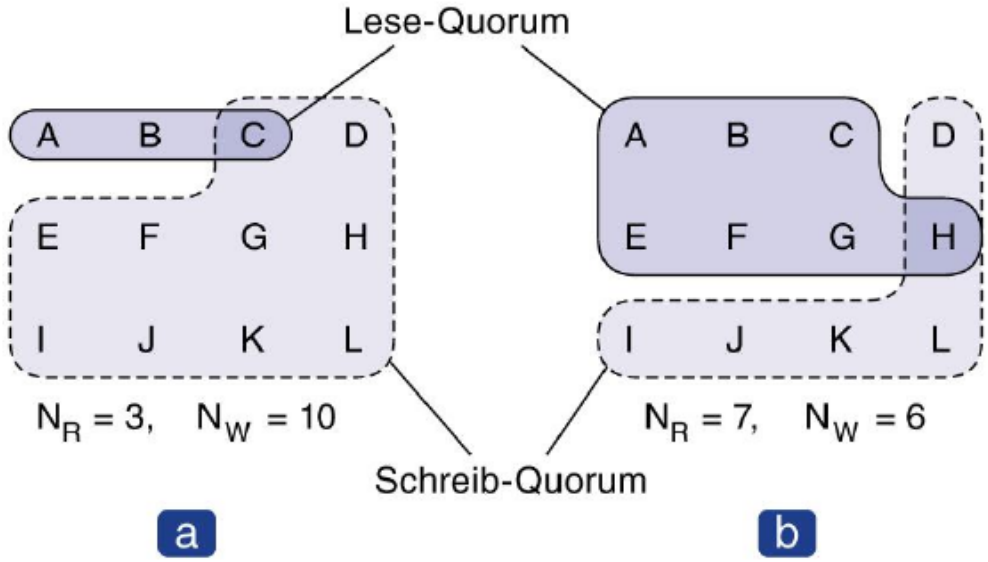


→ Allows at least FIFO consitency
→ If there are not too many concurrent writes, writing is fast.
→ Like with token-based synchronisation the primary is a point of failure. If it fails, no other node can become primary, or costly reconfiguration needed to make another node primary.

Example primary-backup protocol with local writes:
Mobile computing in disconnected mode (ship all relevant files to user before disconnecting and update later on)

Quorum-based protocolls

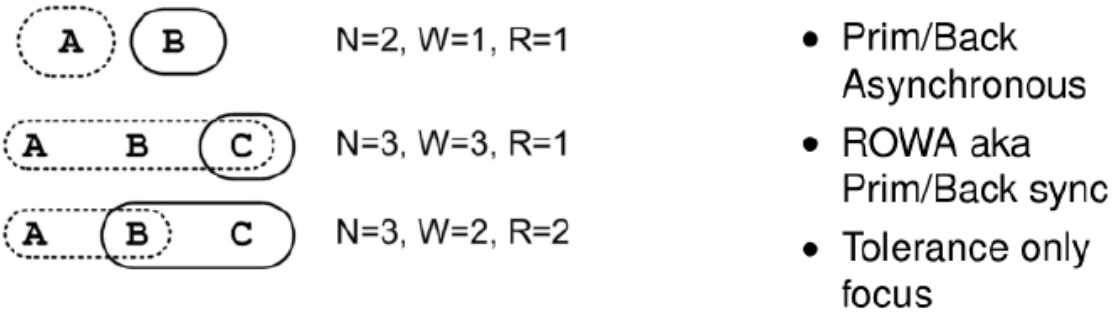
Ensure that each operation is carried out in such a way that a majority vote is established: distunguish read quorum and write quorum.



→ Basic idea: We have 5 nodes. To write to the system, the client has to synchrounously write/read to more than half of nodes (3).
→ Data have timestamps, so that we can establish which version is newer.
→ Advantage: Need to contact less nodes. Important if there are many nodes (shortens operation time) and/or some nodes are (often) inaccessible to the client.

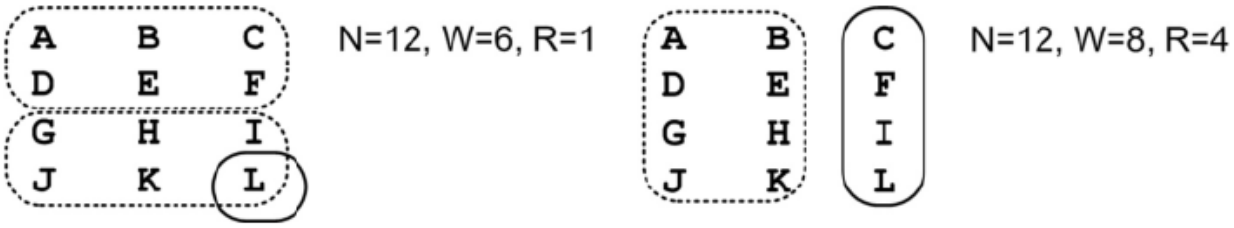
Quorum-based protocol examples

Configure N, W, R as needed



Quorum-based protocol configuration considerations

→ Optimize read: R=1, W=N
→ Optimize write: W=1, R=N (no durability guarantees)
→ Avoid write conflicts: W >= (N+1)/2
→ Strong consistency: W+R > N



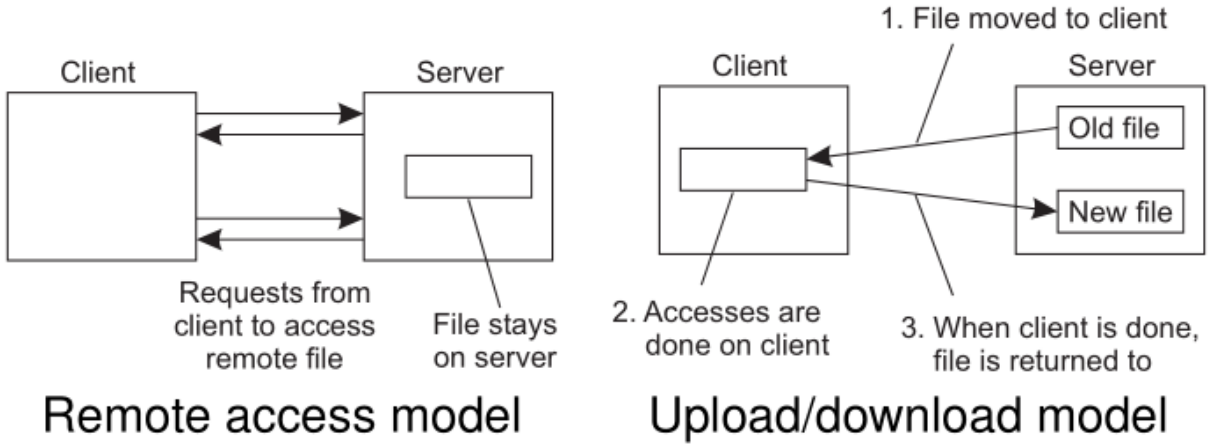
Quorum-based protocol partitioning impact

→ Partition with W nodes continues to take update, same for R set
→ Other partition becomes unavailable
→ if unacceptable: add nodes to other partition and applicaiton-assisted merger later (e.g. Amazon Dynamo)

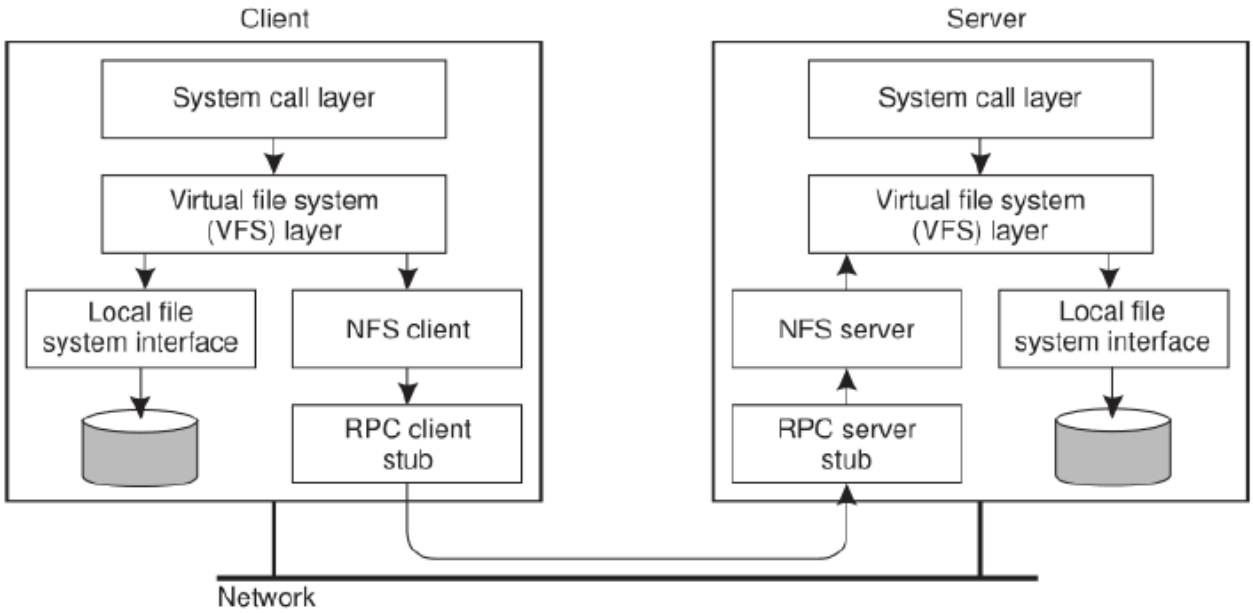
Conclusions:
→ Replication is a mechanism to improve performance (availability, scalability) and fault tolerance.
→ The big problem: consistency, for systems with different requirements we have defined different consistency models.
→ To implement consistency models, we need consistency protocols: Most importantly, we need to understand the implications of applying different protocols and techniques, make correct engineering trade-offs and design decisions to build an efficient system.

Distributed Fie Systems

Try to make a file system transparently available to remote clients.

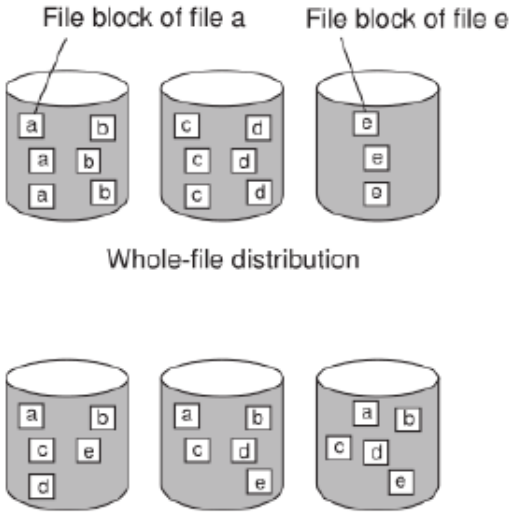


Network File System (NFS) Architecture
NFS is implement using the Virtual File System (VFS) abstraction which is now used for lots of different operating systems.

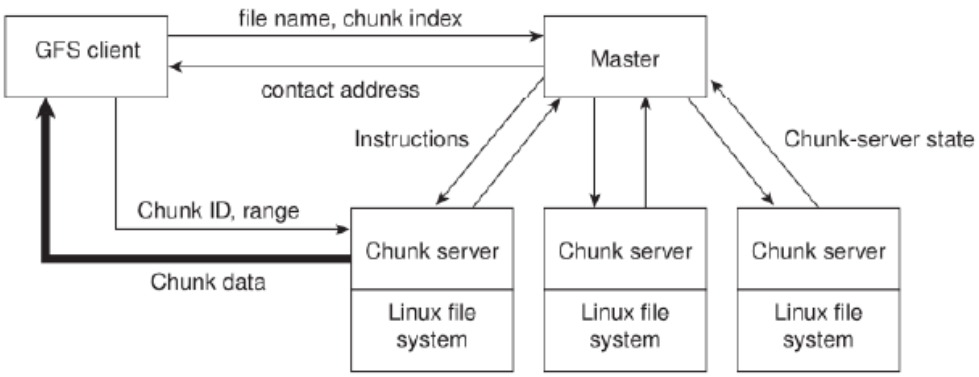


Essence:
VFS provides standard file system inteface, and allows to hide differnece between accessing local or remote file system.
NFS v3 does not, but v4 does support opening and closing files.

Cluster based file systems:
With very large scale data collections, following a simple client-server approach is not going to work → for speeding up file access apply stripping techniques by which files can be fetched in parallel.

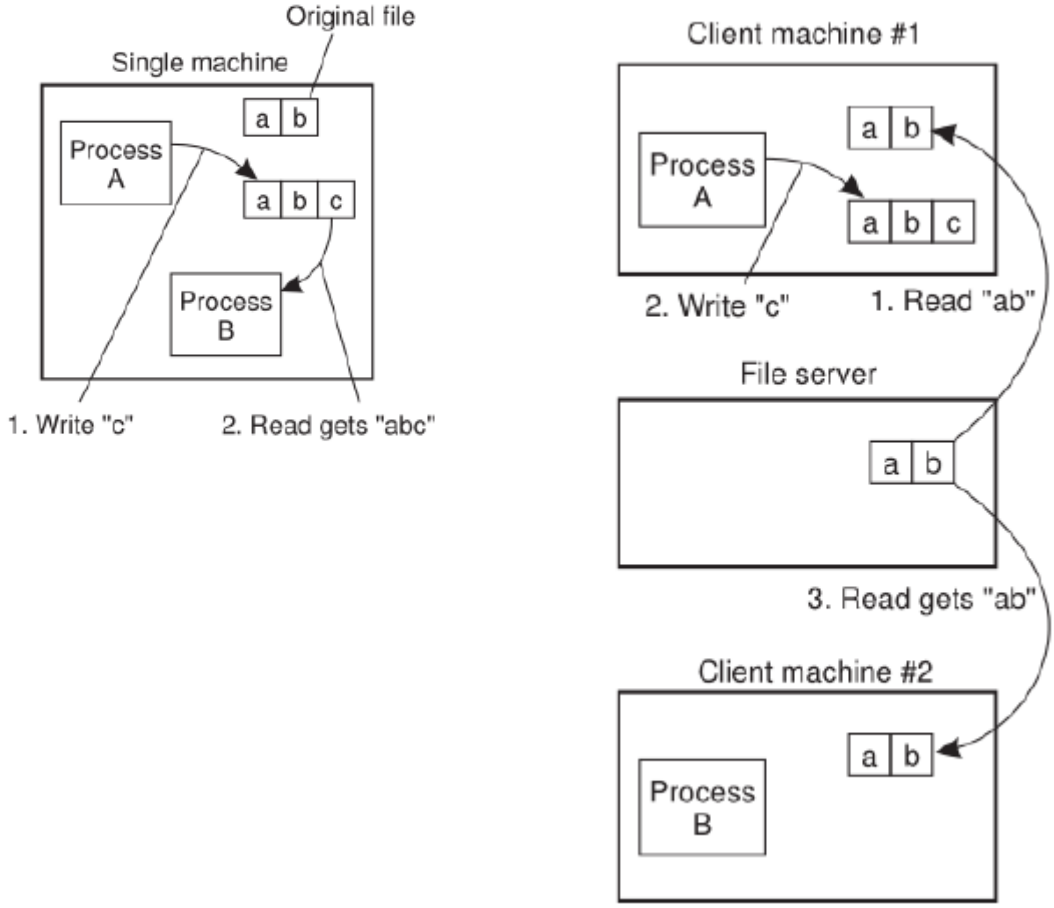


Example: Google File System
Devide files in large 64 MB chunks and distribute/replicate chunks accross many servers:
→ The master maintains only a (file name, chunk server) table in main memory → minimal I/O
→ Files are replicated using a primary-backup scheme, the master is kept out of the loop



File sharing semantics

When dealing with distributed file systems, we need to take into account the ordering of cuncurrent read/write operatins and expected semantics (i.e. consistency).



→ UNIX semantics: a read operation returns the effect of the last write operation → can only be implemented for remote access models in which there is only a single copy of the file
→ Transaction semantics: the file system supports transactions on a singe file → issue is how to allow concurrent access to a physically distributed file
→ Session semantics: the effects of read and write operations are seen only by the client that has opened (a local copy) of the file → what happens when a file is closed (only one client may actually win)

Security

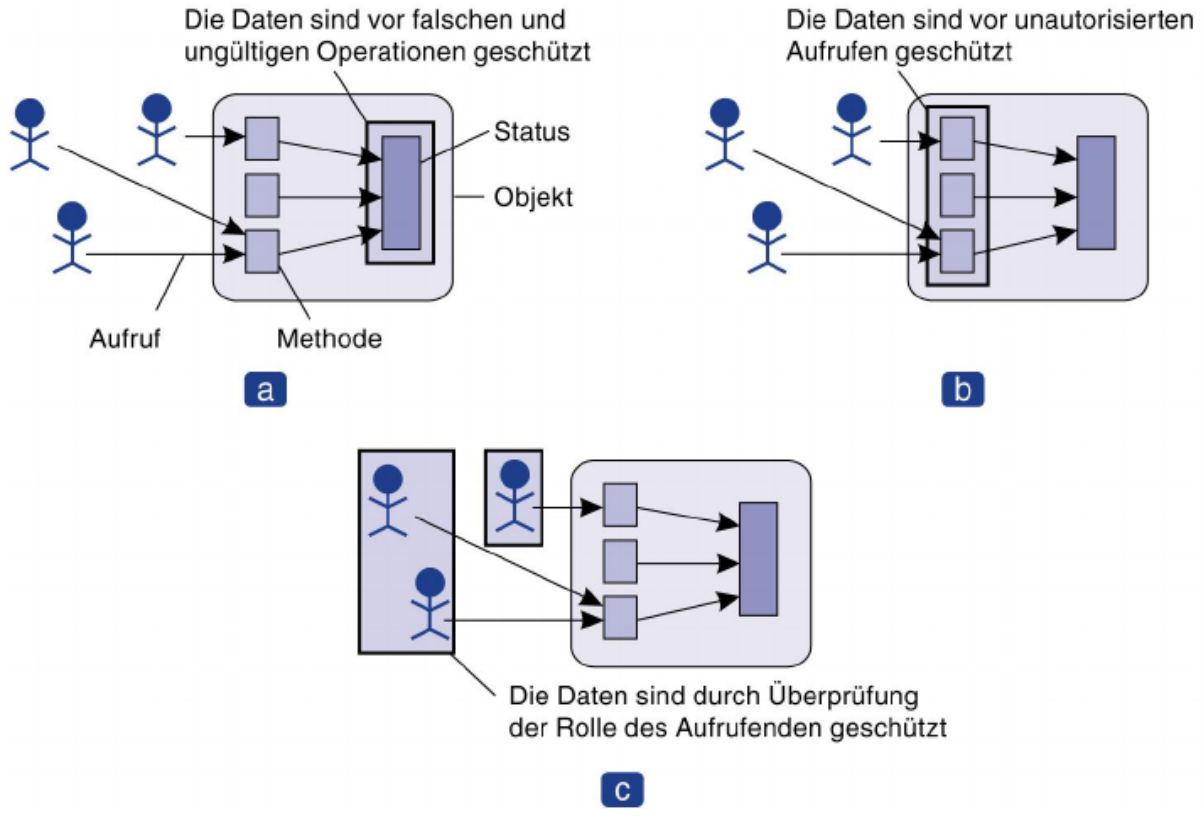
Notion of security overlaps with other wanted properties of DSs, Most important aspects are
→ integrity: no accidental or malicious alterations of information have been performed (even by authorized entities)
→ confidentiality: no unauthotized disclosure of information

Subject: Entitiy capable of issuing a request for a service as provided by objects
Channel: The carrier of requests and replies for services offered to subjects
Object: Entity providing services to subjects

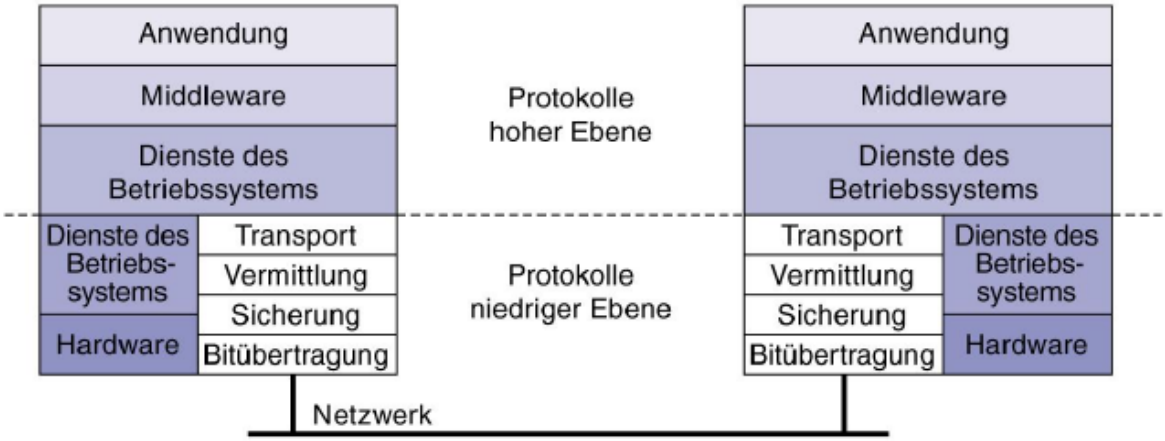
Threats
→ Interception: reading the content of transferred messages on the chanel, or reading the data contained in an object/server
→ Interruption: Preventing message transfer on the chanel, denial of service on the serveral
→ Modification: Changine message content on the chanel, changing an objects/servers encapsulated data
→ Fabrication: Inserting Messages in the channel, spoofing an object/server

To protect against security threats we have a number of security mechanisms:
→ Encryption: confidentiality, integrity
→ Authentication: verify identity
→ Authorization: who is allowed to do something
→ Auditing: trace accesses to help catching attackers

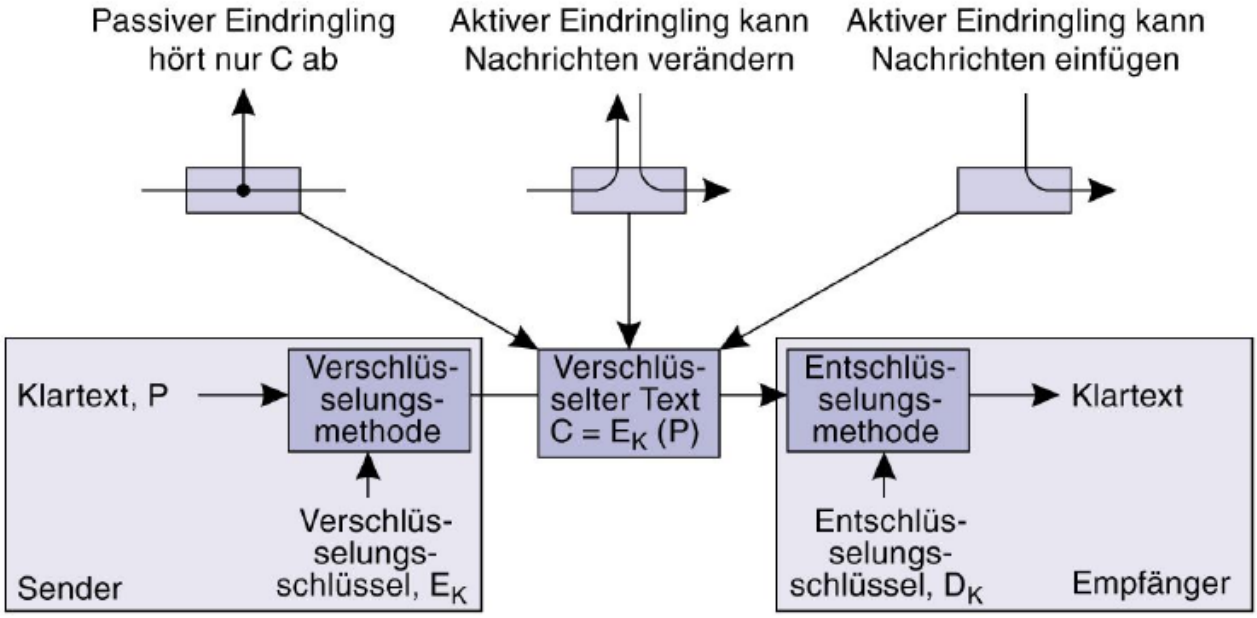
Design issue: focus of control



Design issue: Layering of mechanisms
At which logical level are we going to implement security mechanisms?



The securtiy is as exactly as good as its weekest component/link.



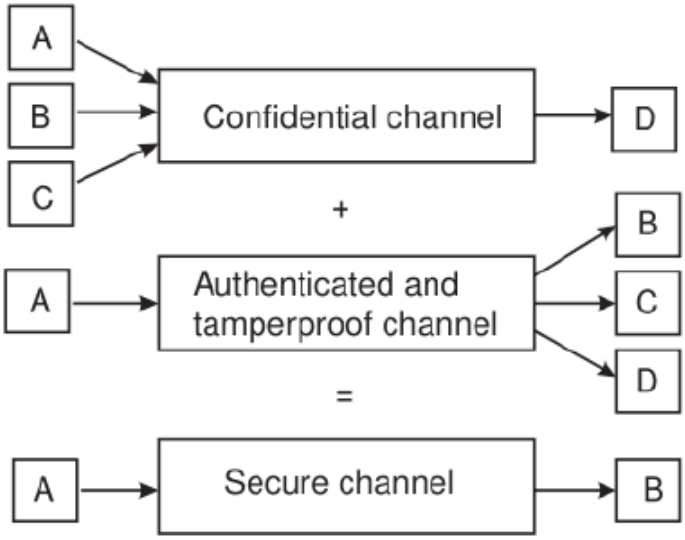
Cryptography Methods:
→ Symmetric system, DES, AES encryption (prevention of interception)
→ Assymetric system, RSA authentication (prevention of fabrication)
→ Hashing systems, MD5 integrity (prevention of modification)

Weak collision resistance: given (m, E(m)) infeasible to find m‘ != m such that E(m‘) = E(m)
Strong collision resistance: infeasible to find m‘ and m such that K(m‘) = E(m)

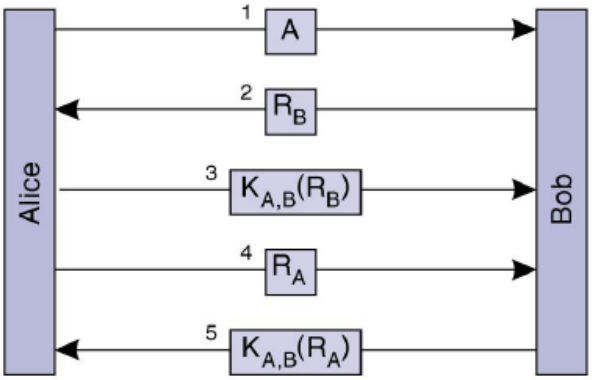
One-way key: cant find a key K so that $m_{out} = E(m_{in})$
Weak key collision resitance: given (m,K,E) it is infeasible to find an K‘ != K such that $E_{K\cdot}(m) = E_k(m)$
Strong key collision resistance: infeasible to find any two different keys K and K‘ such that for all m: E(m‘) = E(m)

Secure Channels

Both parties know who is on the other side (authenticated)
Both parties know that messages cannot be tempered with (integrity)
Both parties know messages cannot leak away (confidentiality)

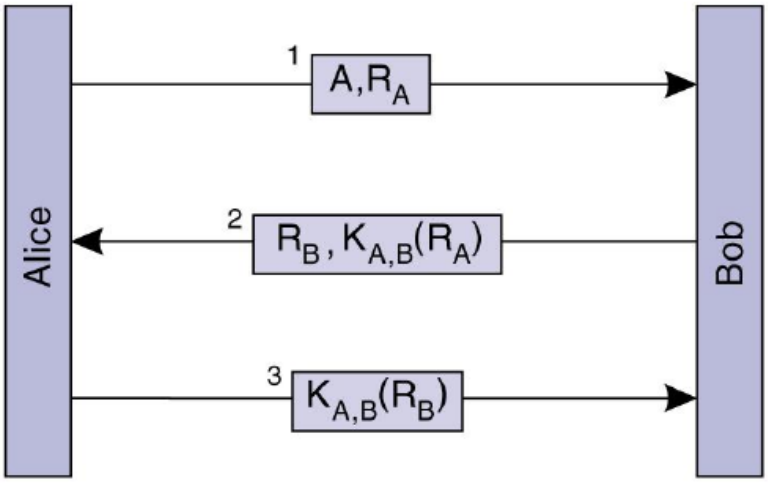


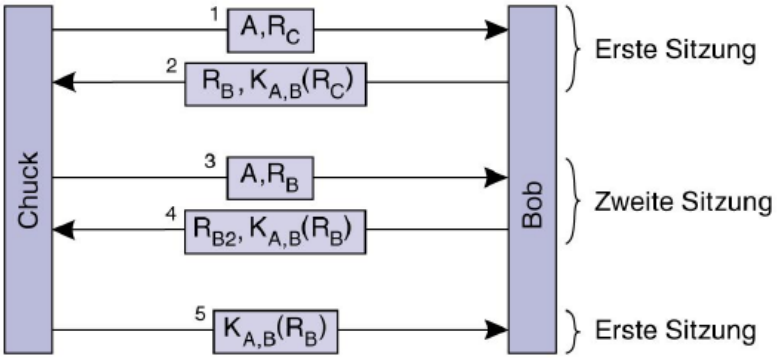
Authentication and data integrity rely on each other:
→ Authentication without integrity: Message is authenticated but intercepted and tempered with but authentication part is left as is. Authentication has become meaningless.
→ Integrity without authentication: Message is intercepted and then the interceptor send its, saying that the content was really sent by the sender. Integrity has become meaningless.



1: Alice sends ID to Bob
2: Bob sends challenge R_B to Alice
3: Alice encrypts R_B with shared key $K_{A,B}$. Bob now knows he is talking to Alice.
4: Alice sends challenge R_A to Bob
5: Bob encrypts R_A with $K_{A,B}$. Alice now knows that she is talking to Bob.

Improvement: Combine 1&4 and 2&5:

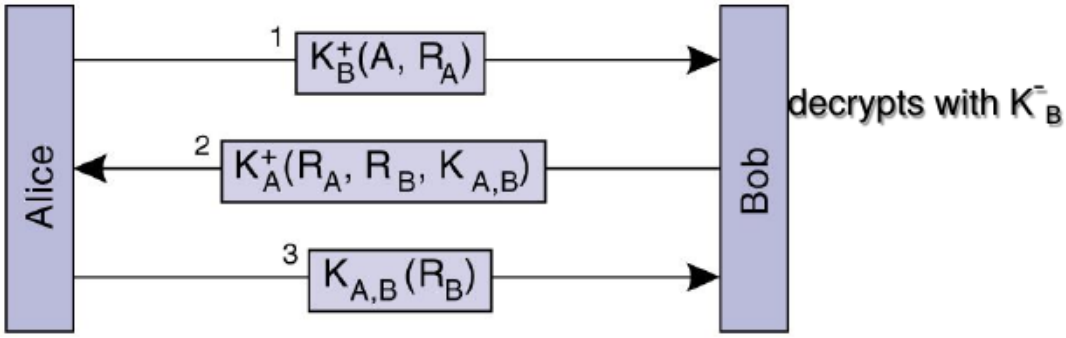




1: Chuck claims he's Alice, and sends challenge R_C
2: Bob returns a challenge R_B and the encrypted R_C
3: Chuck starts a second session, claiming he is Alice, but uses challenge R_B
4: Bob sends back a challenge, plus $K_{A,B}(R_B)$
5: Chuck sends back $K_{A,B}(R_B)$ for the first session to prove he is Alice.

→ Never give (first) valuable information to unknow parties.
→ Even if Alice used even and Bob odd challenges, the protocol would still be susceptible to the man-in-the-middle attacks.

Public-Private Key Authentication



1: Alice sends a challenge R_A to Bob, encrypted with Bob's public key K_B^+.
2: Bob decrypts the message, generates a secret key $K_{A,B}$ (session key), proves he's Bob (by sending R_A back), and sends a challenge R_B to Alice. Everything's encrypted with Alice's public key K_A^+.
3: Alice proves she's Alice by sending back the decrypted challenge, encrypted with generated secret key $K_{A,B}$

Confidentiality
→ Secret key: Use a shared secret key to encrypt and decrypt all messages sent between Alice and Bob
→ Public key: If Alice sends a message m to Bob, she encrypts it with Bobs public key.

Problems with keys:
→ wear out, dont use keys too often
→ danger of replay, dont use keys for different sessions

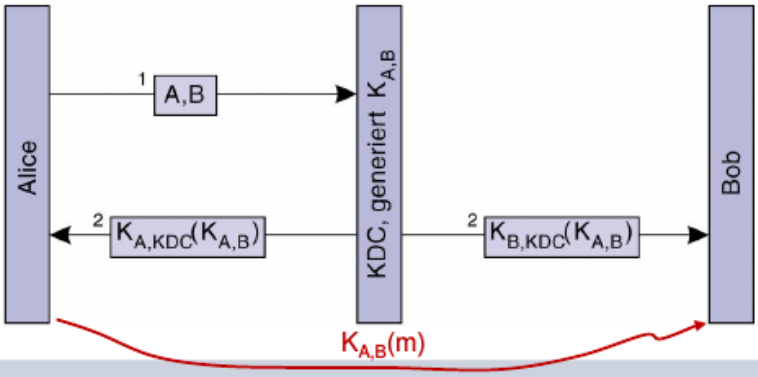
Compromised keys: you can never use it again, dont use the same key for different things.
Temporary keys: make keys disposable

Don‘t use valuable and expensive keys for all communcation, but only for authentication purposes.
→ Include a cheap session key that is used only during one single conversation.

Key Distribution Center

Problem

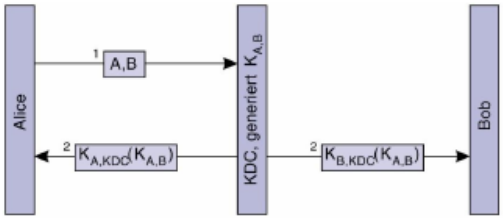
With N subjects, we need to manage $N(N-1)/2$ keys, each subject knowing $N-1$ keys ⇒ use a trusted Key Distribution Center that generates keys when necessary.



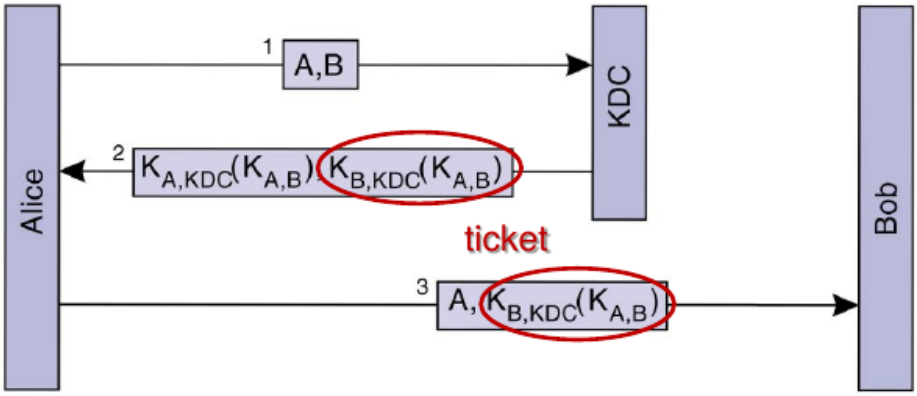
Question

How many keys do we need to manage?

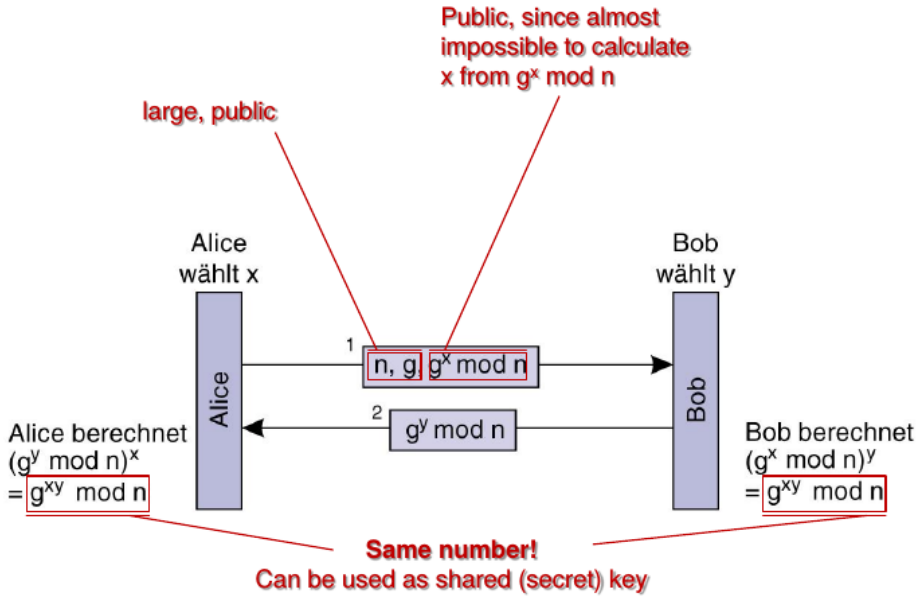
KDC with Tickets



- Optimization:
  Let Alice deliver the session key $K_{A,B}$ to Bob, encrypted with $K_{B,KDC}$



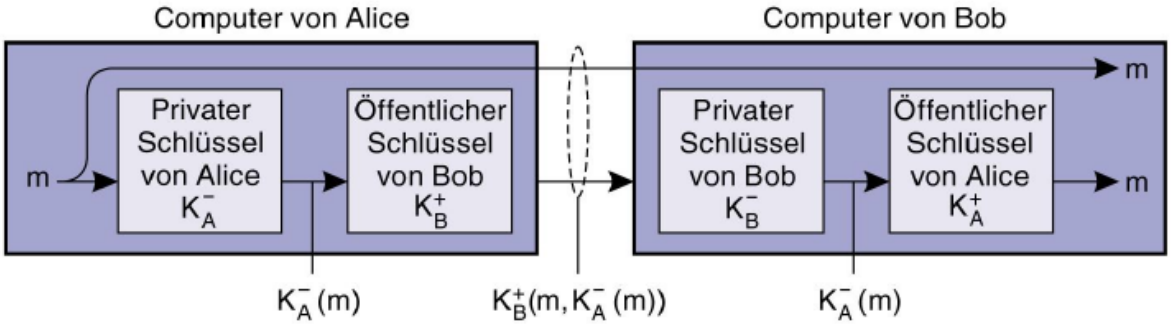
Key establishment: Diffie-Hellman:
We can construct secret keys in a safe way without having to trust a third party (i.e. a KDC)
→ Alice and Bob have to agree on two large numbers, n (prime) and g. Both numbers may be public.
→ Alice chooses large number x and keeps it to herself. Bob does the same, say y.



Key distribution:
Authentication requires cryptographic protocols, → require session keys to establish secure channels. Who‘s responsible for handing out keys?
→ Secret keys: Create your own and exchange it out of band, or trust a KDC and ask it for a key.
→ Public keys: How to guarantee that A‘s public key is actually from A?
→ personally exchanged out of band
→ use a trusted CA to hand out public keys. A public key is put into a certificate, signed by a CA.

Digital signatures:
→ Authentication: Reveiver can verify the claimed identitiy of the sender
→ Nonrepudiation: The sender can later not deny that he sent the message
→ Integrity: The message cannot be maliciously altered during or after receipt

Let a sender sign all transmitted massages in such a way that → the signature can be verifyed and
→ message and signature are uniquely associated

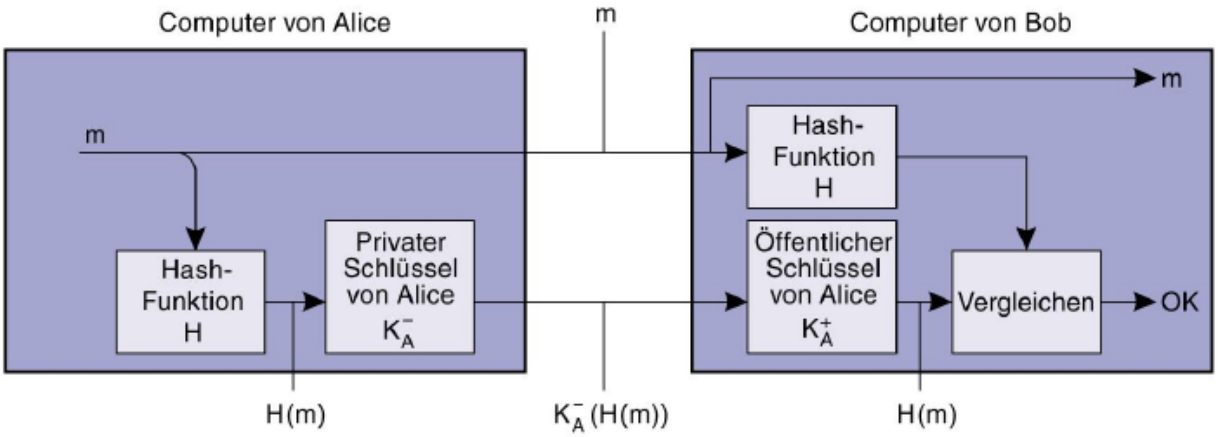


Not recommended

Don't sign the message, rather hash it and then sign it. Subject to attacks.

Message Digests:

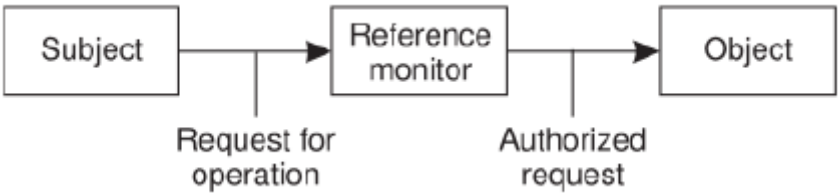
Don‘t mix authentication and secrecy. Instead it should also be possible to send a message in the clear, but have it signed as well → take a message digest and sign that.



Access Control

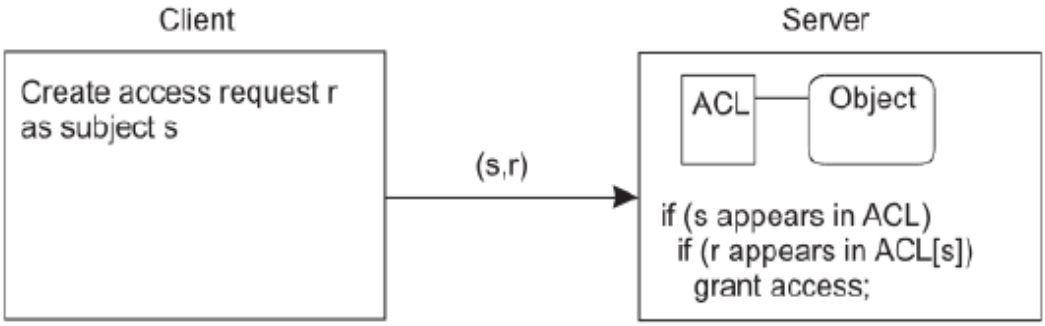
Authorization vs. Authentication
→ Authentication: verfiy identity of a subject
→ Authorization: derermining if subject is permitted to do something

Authorization only makes sense if the requesting subject has been authenticated.



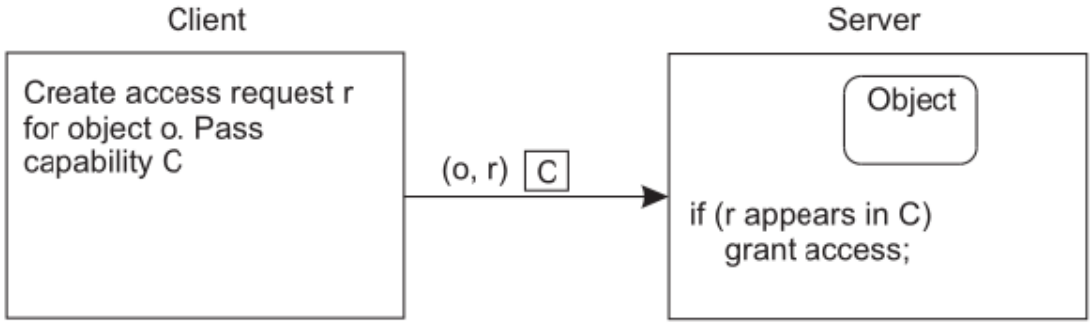
Maintain an access control matrix *ACM* in which entry *ACM[S,O]* contains the permissible operations that subject *S* can perform on object *O* (often R/W/+/- read/write/administer/none) .

	Object X	Object Y	Object Z
Alice	rw+	rw+	rw
Bob	rw	r	rw+
Chuck	-	-	-



(a)

Each object O maintains an access control list (ACL): ACL[*,O] decribing the permissable operations per subject (or group of subjects).
e.g. [Object X] Alice rw+; Bob rw; Chuck -



(b)

Each subject S has a capability: ACM[S,*] descirbing the permissible operations per object (or category of objects. e.g. [Alice] Object X rw+; Object Y rw+, Object Z rw

Protection Domains
→ Groups: Users belong to a specific group; each group has associated access rigths
→ Roles: Don‘t differentiate between users, but only the roles they can play. Your role is determined at login time. Role changes are allowed.

Decreasing time to lookup users to group/role mapping
→ Certificates: Users provide certificates which groups/roles they belong to

Firewalls

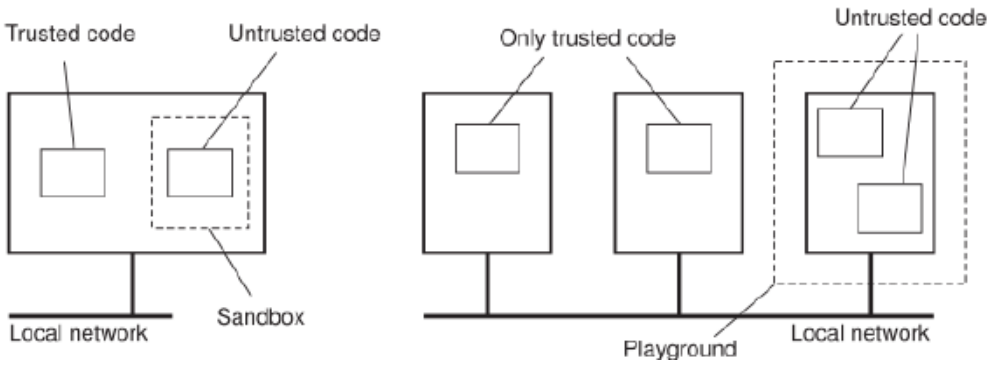
Filtering routers
→ Rules: specify action (allow/deny), source addr/port pattern, destination address/port pattern, +/- flags
→ Matching: apply rules in ordered sequence, execute upon match or default action

Application level gateway
→ Packet inspection: inerpret content based on application semantics; Mail example: drop attachments with .exe files, Web example: filter out scripts or applets

Secure mobile code: protecting agent
Fully protecting an agent almost impossible
→ Always depends on trusting in what host promises to do (or not to do)
→ Focus thus on detecting tampering while on the move
→ Encryption and Signatures

Enforce a (very strict) single policy, and implement that by means of a few simple mechanisms:

→ Sandbox model: Policy → Remote code is allowed access to only a pre-defined collection of resources and services; Mechanism → Check instruction for illegal memory access and service access.
→ Playground model: Same policy, but mechanism is to run code on seperate „unprotected“ machine.



Common Attack Secnarios
Any security breach potentially renders the entire system insecure

Buffer Overflows, SQL Injection, XSS, DDoS, Sidechannel Attacks, Socail Engineering

Stack Buffer Overflow: commong in unmanaged programming languages like C/C++
Allows attacker to overwrite return address and thus execute arbitrary code.

SQL Injection: Some web applications do not sufficently check data received from users befor issuing SQL queries.

```
select * from users where user = $username
                      and pw = md5($pw)
```

now assume following input:

```
$username = '1 or 1=1; drop table users; --'
```

and you get:

```
select * from users where user = 1 or 1=1;
                      drop table users; --
```

XSS: Some web applications do not sufficiently check data received from users; similar principle to SQL injection; Allows attakcer to inject arbitrary scripts into a legit (trustable) website.
Example: blog with commentary function that accepty arbitrary HTMl code

DDoS: Attacker uses a network of hacked machines; Bots/Zombies overload resources of target, difficult to protect against (needs to be done on ISP level), difficult to identify the attacker.

Sidechannel Attacks & Social Engineering
Ignore technical security mechanisms, but find out the secret that the mechanism was based on. Phising for PW or keys; NSA demanding private keys form certificate authorities; Reversee engineering keys in embedded devices by meassuring energy consumption

Sidechannel attakcs on humans behing the „secure“ technical system: usual assumption: people are easily manipulated. e.g. incoming call „Hey I‘m from IT, we have a problem with your account here… TBC