Grundlagen

1. **Was ist ein distributed System?**

Ein Distributed System ist eine Ansammlung von Computern, welche durch ein Netzwerk verbunden und durch Software unterstützt sind was ihnen die Möglichkeit gibt als eine Einheit zu arbeiten.
Komponenten: Hardware, Software, Netzwerk(verbindung)

2. **Welche Typen von distributed Systems kennen Sie?**

•Object/component based (CORBA, EJB [Enterprise Java Beans], COM)
•File based (NFS)
•Document based (WWW, Lotus Notes)
•Coordination (or event-) based (Jini, JavaSpaces, publish/subscribe, P2P)
•Resource oriented (GRID computing, Cloud cleasomputing, MANET) -> Distributed Computing
•Service oriented (Web services, Cloud, P2P) -> Distributed Information Systems

Allgemeine Arten:
- Distributet Computing (cluster, GRID, cloud)
- Distributed Information Systems (EAI [Enterprise Application Integration], TP [Transaction Processing], SOA [Servive oriented Architecture])
- Distributed Pervasive (often P2P, UpnP [Universal Plug n Play] in home systems, sensor networks, ...)

Evolution of Distribution Technologies

- Mainframe computers
- Workstations and local Networks
- Client-server systems
- Internet scale systems and WWW
- Sensor/actor networks in automation
- Mobile, ad-hoc and adaptive systems
- Pervasive (ubiquitous [allgegenwärti]) systems
- Today, less than 2 % of processor go into personal computers!

Key Concepts of Distributed Systems
- Communication (How are the entities communicating over the network)
- Concurrency and operating system support (How does it work? Cooperative, competitive?)
- Naming and discovery (of the resources in the system)
- Synchronization (f.ex. of time) and agreement (based on time, role, ...)
- Consistency and replication (How important is it? How is data replicated?)
- Fault-Tolerance (Deal with it!)
- Security

3. **Welche Strategie ist für folgende Anwendungsszenarien besser geeignet (P2P oder Cloud Service): a. Zentralisiertes File Hosting System, b. Netzwerk um Medieninhalte unter Benutzern auszutauschen**

- P2P ist gut geeignet, um Dateien unter Usern auszutauschen, da User direkt mit anderen User kommunizieren kann
- Cloud ist gut geeignet, um ein Zentralisiertes File Hosting System zu betreiben

Why distribute?

Connecting users to resources and services
Dependability and security (Availability, fault tolerance, Intrusion tolerance, …)
Performance (latency, throughput)
Only distribute if necessary!

Design goals of distributed systems

Resource sharing
Transparency
Hiding internal structures, complexity (Openness, portability, interoperability, …)
Services provided by standard rules
Scalability
Ability to expand the system easily
Concurrency
Fault tolerance, availability

4. **The 8 Fallacies of Distributed Computing**

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

5. **Was ist QoS (Quality of Service)?**

QoS ist ein Konzept, in dem Clients angeben können, welches Level von Services sie benötigen. So ist für zB. Voice over IP wohl eher die schnelle als die korrekte Übertragung wichtig, beim Onlinebanking wohl eher andersrum. Alles zusammen ist jedoch nicht möglich.

BEN: Best Effort Network: Alles wird gleich behandelt. Grundlage für Net Neutrality!

6. **Was ist Transparency (In distributed systems)?**

Bestimmte Eigenschaften / Teile eines Systems werden vor dem Nutzer versteckt. Kann z.B. durch mehrere Layer erzeugt werden (Service Layer – Middleware – Data Layer). Nutzer verwendet dann Service Layer, welches gewisse QoS bietet.

Arten von Transparenz:

Transparency nur dann, wenn es sinnvoll ist, nicht zwingend überall einsetzen. Performance Transparenz schwierig, Trade-off transparency / performance

Transparency	Description
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 several competitive users
Failure	Hide the failure and recovery of a resource

7. **Was ist Openness?**
Openness ist ein Ziel, bei dem ich meine Services nach Standards anbiete, z.B. mit Interfaces (IDL: often informal; Complete → Interoperability: Communication between processes; Neutral → Portability: different implementations of an interface) ausstatte und möglichst flexibel konstruiere (Komposition, Konfiguration, Ersetzbarkeit, Erweiterbarkeit). Openness ist in Protokollen formalisiert. Somit können andere Services meinen Service einfacher und besser integrieren.
Beispiel: HTML - Durch das einigen auf HTML, können Webserver HTML produzieren und (relativ) sicher sein, dass die Seite richtig angezeigt wird. Durch ein Plugin Interface könen neue Services leicht hinzugefügt werden.
Beispiele: Unterschiedliche Webserver und Browser arbeiten miteinander
Neue Browser die bereits existierende Services nutzen möchten
Plugin Interfaces ermöglichen es neue Services hinzuzufügen

Welche Probleme kann die Heterogenität in VS bereiten?
In a heterogeneous distributed system, steps must be taken to make sure that all processes can communicate with each other. Typically whenever processes with different operating systems attempt to communicate, some type of conversion must take place. This greatly increases the complexity of the system and thereby the potential for faults.

Wie hilft die Middleware, um diese Probleme zu minimieren?
The middleware attempts to mask these differences and present all processes with a unified interface. The processes need not be aware that they are communicating with different types of machines and should be able to send messages, data, etc. as easily as possible.

Separating policy from mechanism

Granularity: Do you apply policies to objects or applications?
How do you interact betwen the components: Component interaction and composition standards (instead of closed / monolithic). Example: Web browser supports facility to store cached documents, but caching policy can be plugged in arbitrarily.

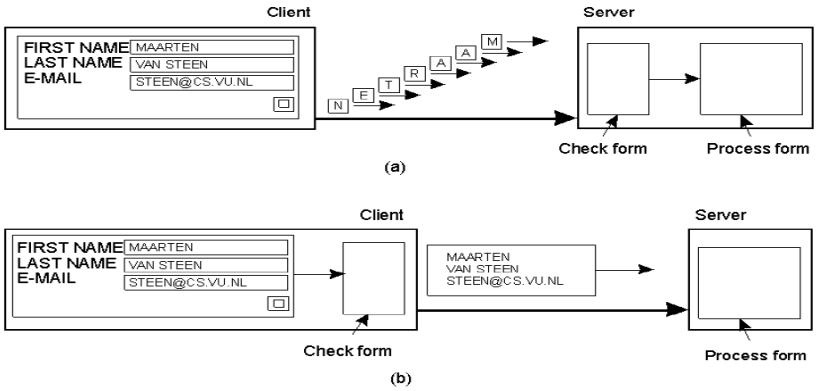
8. Was ist Scalability?
Scalability ist ein Ziel, das bedeutet, dass ein distributed System wachsen kann (In Größe, Geographie und Administration). Auch wenn das System wächst, bleibt es effektiv und das System und die Applikation sollten nicht geändert werden müssen. Je mehr Scalability, desto weniger Security und umgekehrt. Die physikalische Anforderung sollte nicht schlechter als O(n) sein (also linear wachsen), der Performance- Verlust sollte nicht schlechter als O(log n) (bedeutet, dass die Zeit linear erhöht wird und n exponential. Beispiel: Wenn es 1 Sekunde dauert 10 Berechnungen durchzuführen, dauert es 2 Sekunden für 100, 3 für 1000 etc) sein. Botlenecks vermeiden.

Techniken:
- Wartezeiten verbergen (Durch asynchrone, parallele Abarbeitung)
- Aufteilung (Hierarchien, Domains, Zonen, ...)
- kopierte Services (Aufteilung, z.B. durch einen load balancer)

Eine der verbreitetsten Techniken ist die Replikation von Entitäten. Beispiel: Durch das Aufstellen zusätzlicher physikalischer Server mit identischen Services, sowie dem Einsatz eines Load Balancers kann die Performance eines Systems/Services erhöht werden, da die Auslastung der einzelnen Server reduziert wird.

Geographic Scalability: LAN vs WAN?
Lan: Synchron, Schnell, Broadcast, hohe Verlässlichkeit.
WAN: Asynchron, Langsam, Point 2 Point, Nicht verlässlich.

Administrative Scalability:
Administrative Scalability beschreibt die Möglichkeit, ein verteiltes System unter einer wachsenden Anzahl von Benutzern verwalten zu können. Ein Beispiel wäre dafür eine Cloud, die Ressourcen on-demand bereitstellt (resource usage). Hier ist ein gut funktionierendes Abrechnungssystem (billing) eine Grundvoraussetzung für den Betreiber, um die Cloud vergrößern zu können. Auch wichtig: Management, Security: Schutz zwischen administrative domains – trusted domains – durchgesetzte Beschränkungen
Beispiel: Telefonnummer übernehmen



The difference between letting:

(a) a server or (b) a client check forms as they are being filled

9. Performance Bottlenecks: Centralized services, data, algorithms?
- Centralized services: Ein einzelnes Service, das alleine zuständig ist. Z.B.: Ein einziger Server für alle User – Bottleneck!

- Centralized data: Die Daten sind in einem einzelnen System gespeichert. Z.B.: central DNS - Bottleneck!

- Centralized algorithms: Ein zentraler Algorithmus sammelt alle Informationen und beginnt dann die Berechnung. Bottleneck - Es ist sehr aufwendig (und oft nicht sinnvoll) zuerst alle Informationen zu sammeln.

Decentralized Algorithms:
- Keine Maschine hat komplette Information über Systemzustand
- Maschine entscheidet nur aufgrund lokaler Informationen
- Fehler auf einer Maschine zerstört nicht Algorithmus (kein single point of failure)
- Keine Annahme dass globale Uhr existiert.

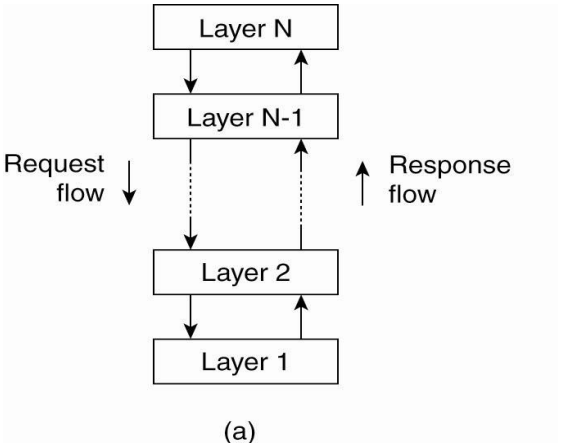
Dealing with complexity

Abstraction (unnötiges weglassen): Client, Server, Service; Interface vs implementation
Information hiding (encapsulation): Interface design
Separation of concerns: Layering, Client and server, Components

Communication models

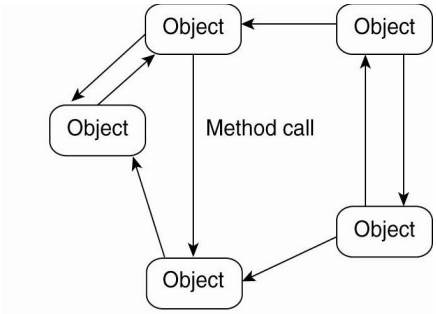
Multiprozessoren: geteilter Speicher (benötigt Schutz gegen fehlerhaften gleichzeitigen Zugriff)
Multicomputer: Nachrichten schicken
Synchronisation in geteiltem Speicher: Semaphores, Monitors („Aufpasser“)
Synchronisation bei Multicomputern: Blocken im Message Passing

11. Welche wichtigen Architekturen von verteilten Systemen gibt es?
- *Layered:* Anfragen werden von einem Layer zum nächsten weitergegeben und abgearbeitet.



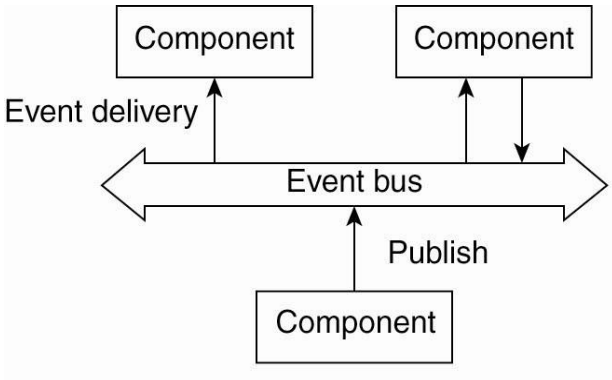
(a)

- *Object-based:* Objekte rufen sich gegenseitig auf, Anfrage wird von Objekt zu Objekt abgearbeitet.



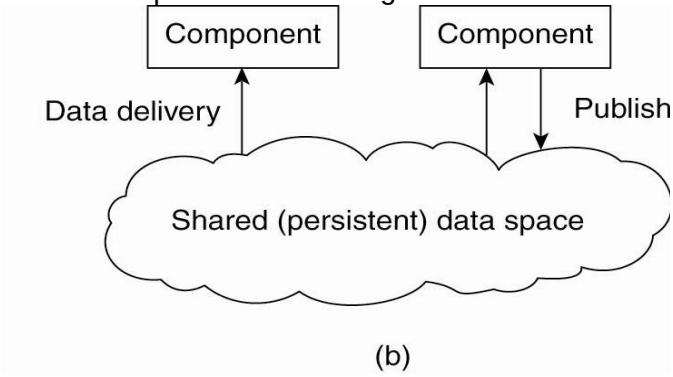
(b)

- *Data-centered*: Es gibt einen zentralen Datenspeicher (z.B. eine Datenbank), alle Komponenten greifen darauf zu.
- *Event-based:* Es gibt einen zentralen Event-Bus der an die verschiedenen Komponenten die Events leitet. Die Komponenten können Events an den Event Bus senden die entsprechend weitergeleitet werden

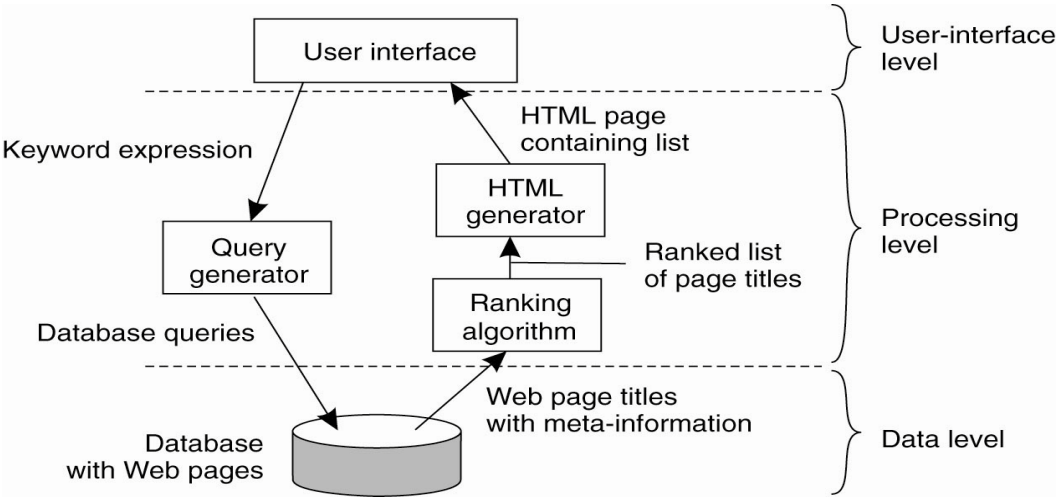


(a)

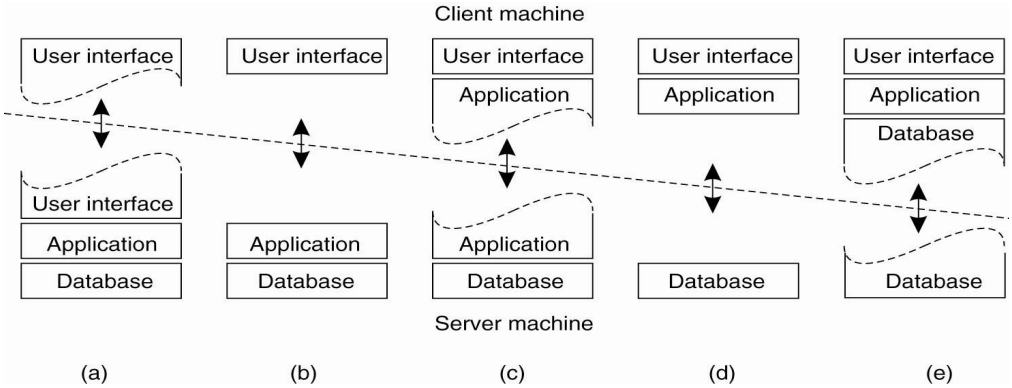
- *Shared data-space:* Mehrere Komponenten nutzen gemeinsamen Datenbereich

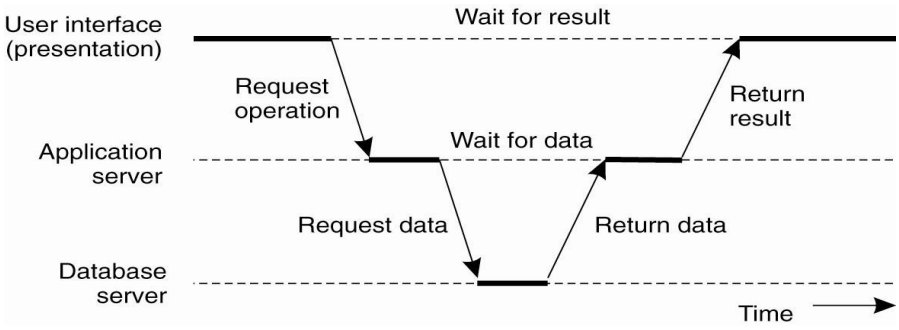


Beispiel Internet Search Engine (Layered):

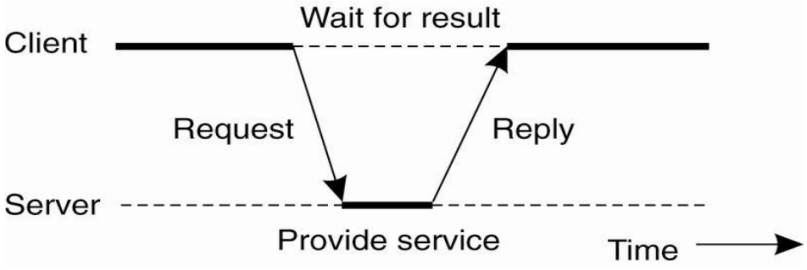


Was ist eine Two-, Three- bzw. N-tier (tier = Schichten) Architektur?:
- Die einfachste Variante ist eine Two-tier Architektur (Nur Client und Server).
- Eine Three-Tier Architektur ist z.B. die Internet Search Engine





12. Wie funktioniert die Kommunikation zwischen Client und Server allgemein?



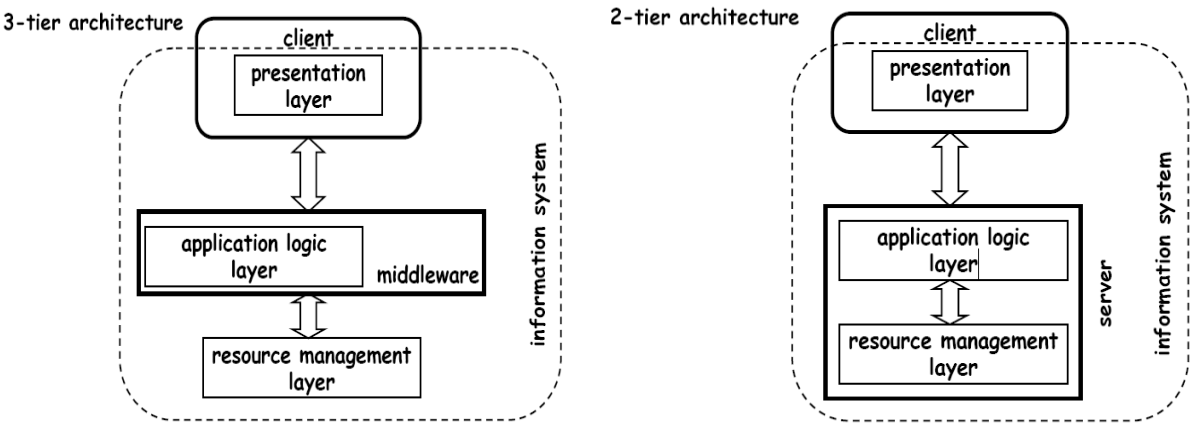
13. **Identify some concrete types of communication entities in real-world distributed systems (e.g., in a parallel cluster system)**
- Application processes
- Middleware and Runtime processes
- OS processes like Sockets, Routing, etc.
- Physical parts like Hosts
- Routers
- Switches etc

14. **Give an example of a distributed system. (1 point), b. Explain briefly its geographical distribution, communication, and naming in 3 sentences. (1 point)**
a) Facebook - Abstrakt gesehen gibt es einen Server, und fast 1 Milliarde Clients, die sich mit dem Server verbinden.

b) Der Server besteht aber selber wieder aus einem verteilten System von verschiedensten Front-End-Servern rund um die Welt, die die Anfragen entgegennehmen und auf gecachte Daten zugreifen, verschiedenen Data-Storage-Servern, die die Daten speichern und die Front-End-Server regelmäßig updaten und anderen Komponenten wie einem Chat-Server. Load-Balancer verteilen die Anfragen auf die Front-End-Server. Die Server sind über die ganze Welt verteilt.

Geographical distribution is the physical distribution of servers and clients across an area. Communication is the purposeful exchange of messages between at least two parties. Naming involves assigning resources/servers/clients in a system a unique identifier, which is supported by a system that allows the accurate discovery of a resource/server/client based on the name.

15. **Clientserver architecture: a. Draw one possible threetiered architecture. (1 point), b. Explain the placement of components in the threetiered architecture if you choose the vertical distribution style. (1 point)**

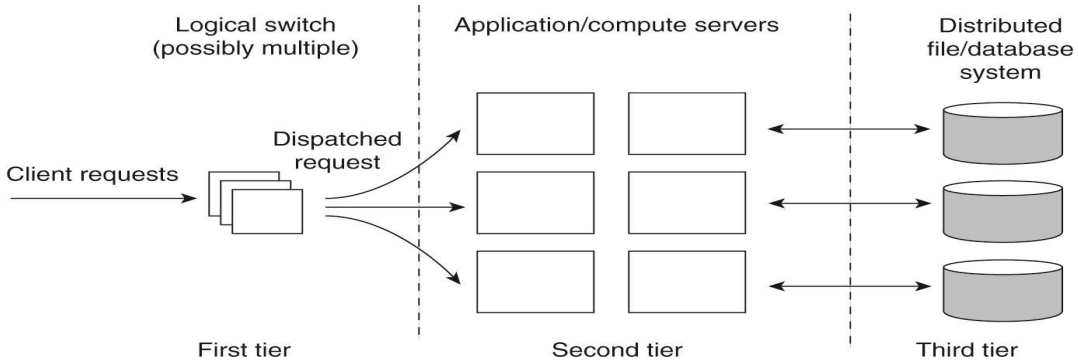


The logical switch handles all client requests and is the front-facing tier of a service. Client requests are passed to the second tier that contains the actual business logic of the service where computation and processing of the request takes place. The second tier accesses the third tier where the data that supports the entire service are stored.

16. **You are asked to explain the following architecture in which a service using several application/compute servers serves client requests**

a. How can this architecture help to improve the performance of the service? (1 p)

b. How can this architecture help to reduce the failure rate of the service? (1 p)



Dat sind Server Clusters: 3 different Tiers (so funzen Amazon / Google / Facebook)

a.: Anfragen werden an mehrere Worker (second tier) ausgelagert, so können mehrere Anfragen gleichzeitig abgearbeitet werden.

b.: Wenn ein Worker ausfällt, steht nicht das komplette System, da die anderen Worker weitermachen können.

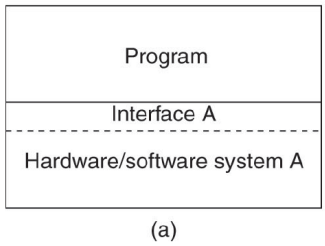
17. **Drei Nicht-funktionale Anforderungen von VS aufzählen und erklären.**

- Integrity: Die ungewollte Modifikation von Ressourcen muss verhindert werden.
- Security: Die Sicherheit in und für ein verteiltes System muss gegeben sein; schwerwiegende Konsequenzen bei Fehlern müssen ausgeschlossen sein.
- Availability: Das System muss die bestmögliche Verfügbarkeit aufweisen.

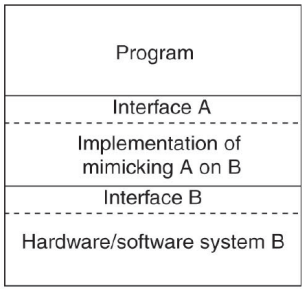
Prozesse & Kommunikation

18. **Was ist ein virtueller Prozessor? Virtualisierung allgemein**
Ein virtueller Prozessor ist ein Stück Software, das die Funktionen eines Prozessors simuliert. Sinnvoll, um z.B.: mehrere unabhängige Systeme auf einem Computer laufen zu lassen. Ein Prozessor kann grundsätzlich eine Serie von Instruktionen abarbeiten.
Durch Virtualisierung kann ich Ressourcen besser auf mehrere User aufteilen (in unterschiedlichen Bereichen, z.B. Speicher, Netzwerk, …) . Um physikalische Ressourcen virtuell aufzuteilen ist der sog. Hypervisor notwendig. Beispiel: User 1 und 2 brauchen normalerweise jeweils 10MB Memory, zu Spitzenzeiten aber 30MB Memory. Anstatt jetzt 2x 30MB Memory (30) zu verwenden, kann ich beiden einen virtuellen Memory von jeweils 30MB anbieten, aber nur einen echten Memory von 40MB verwenden. Sollte jetzt einer der User eine Spitze erreichen, geht sich das mit den 40MB noch aus, sollten wirklich beide gleichzeitig eine Spitze erreichen, muss ich spontan mehr Memory dazu nehmen. Ein anderer Vorteil: Ich kann unabhängig vom echten Betriebssystem ein individuell konfiguriertes System verwenden, das ich dann einfach 1:1 auf einen anderen Server übersiedeln kann. Weiters kann durch Virtualisierung Fehlerafälligkeit reduziert werden.
Virtualisierung kann auf verschiedenen Ebenen ansetzen:
- Ganzes Betriebssystem ist virtuell und wird von einer Virtual Machine ausgeführt
- Ein Runtime-System kann Plattformunabhängigen Code in plattform-spezifischen Code kompilieren (z.B.: Java)

Weiters:
Basic idea: Abstract view on IT resources
•Pooling of resources
•Possible on different levels:
•Platform (complete machine)
•Memory
•HDD
•Network

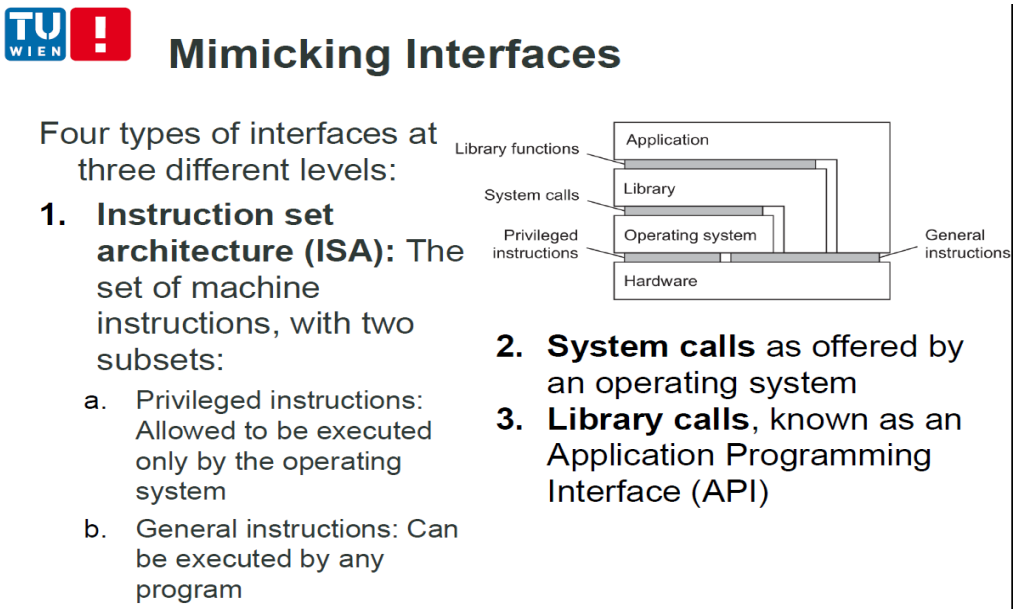


(a)



(b)

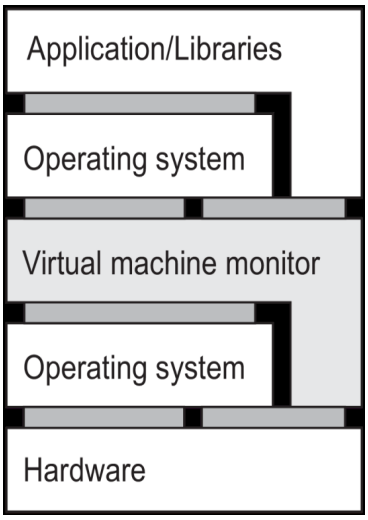
Architektur von virtuellen Maschinen:



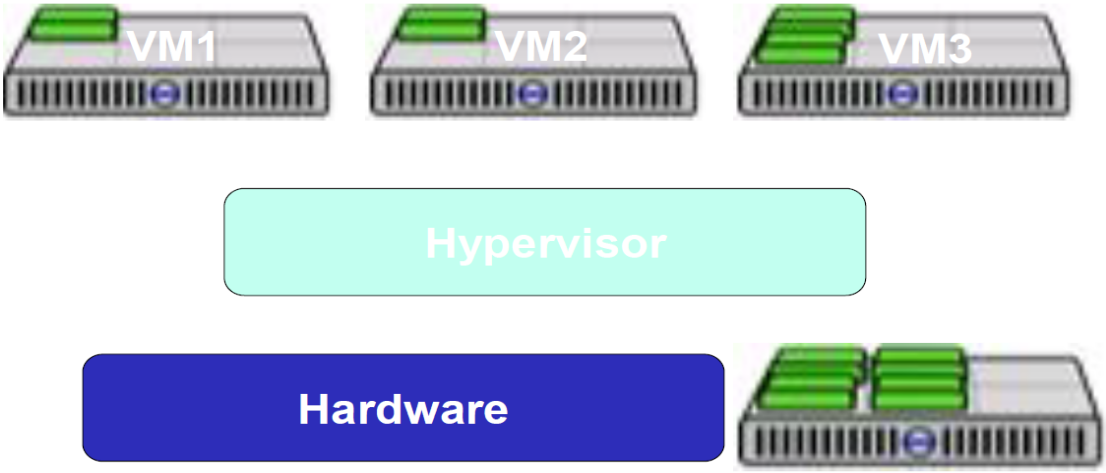
rechts: generelle (Maschinen) Instruktionen, links: privilegierte Instruktionen (Schnittstelle zwischen Betriebssystem und Hardware).

Prozess VM vs. VM Monitors VS Hosted VM Monitor

- Process VM stellt Laufzeitsystem zur Verfügung, dockt beim Operating System und teilweise bei der Hardware an. Programm wird in Zwischencode kompiliert und der wird ausgeführt (Java VM). Verfügt über separates Befehlsset, Interpreter und läuft auf einem Betriebssystem.
- VM Monitor: Basieren nicht auf Betriebssystem, ist Zwischenschicht zwischen Hardware und Betriebssystem (Virtual Machines). Vorteil: kann unterschiedlichen Systemen angeboten werden. Bietet aber weniger Befehle an (das macht das Betriebssystem).
- Hosted VM Monitor bieten ihre Funktionalitäten einem Betriebssystem an, nutzen aber auch eines. Das heißt, er erweitert das Prinzip des VM Monitors, dass bestimmte Funktionalitäten des Betriebssystems übernommen werden. (wird zB von Amazon genutzt)

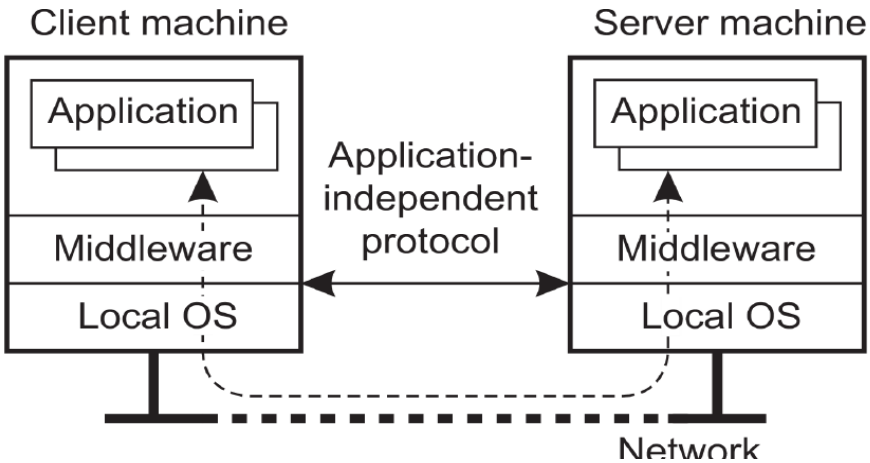


Platform Virtualization:



Vorteile:
Higher degree of capacity utilization:
•Resources are shared between users
•Consolidation:
•Put many different classes of applications onto different virtualized assets
•Positive side effects: energy consumption & space usage
•Fault tolerance:
•Isolation of failures caused by errors or security problems

Clients: User Interface



Client-Side-Software: zB Client sendet Request an unterschiedliche Server. Dadurch erreichte Transparenz: Access transparency, location / migration transparency, replication transparency, failure transparency.

19. **Was ist ein Thread? Was ist ein Prozess?**
Threads und Prozesse sind logische Einheiten, die eine Sammlung von Instruktionen abarbeiten können. Threads laufen auf Software-Prozessoren (virtuellen Prozessoren). Ein Prozess hat im Gegensatz zu einem Thread seinen eigenen Addressraum.

Processor: Provides a set of instructions along with the capability of automatically executing a series of these instructions. Also includes things like stackpointer, adress register
Thread: A **minimal** software processor in whose context a series of instructions can be executed. Includes parts of processor context (obviously).
Process: A software processor in whose context one or more threads may be executed. Auch Program in Execution genannt. Prozess verfügt im Gegensatz zu Threads über einen eigenen Addressraum und Speicher.

Context-Switchting:
Eine recht teure Aufgabe des OS ist es verschiedenen Prozessen abwechselnd den Prozessor zur Verfügung zu stellen. Threads können unabhängig vom OS in einem Prozess abgewechselt werden, was natürlich viel schneller geht als wenn das OS die Prozesse abwechselt. Genauso ist das Erstellen und Löschen von Threads viel schneller.
Processor context: Minimale Sammlung an Werten die in den Registern eines Prozesses zur Ausführung einer Reihe von Aufgaben gespeichert sind.
Thread context: Minimale Sammlung an Werten die in den Registern und Speichern zur Ausführung einer Reihe von Aufgaben gespeichert sind.
Process context: Sammlung an Werten die in den Registern und Speichern zur Ausführung eines Threads gespeichert sind.

- Single-Threaded: Ein Prozess hat nur einen Thread. Blockiert der, blockiert der Prozess
- Multi-Threaded: Ein Prozess hat mehrere Threads. Blockiert einer, kann der Prozess weitermachen + Threads werden auf mehrere Prozessors ausgelagert => parallel!
- Multi-Prozessor: Ein System mit mehreren Prozessoren

Wenn man einen Webserver schreibt, der viele Clients möglichst rasch bearbeiten soll, was/ warum am besten ist: Single-Threaded, Multi-Threaded, Multi-Prozessor:
Multi-threaded webserver with multiprocessors. There should be no blocking of clients and their requests, which is why each client should receive its own thread to interact with the server.
-> Ein dispatcher-Thread teilt die Anfragen auf die Worker-Threads auf, die dann parallel ablaufen.
Beispiel: Socket-Programmierung - ein Client soll möglichst schnell eine Liste von Dokumenten herunterladen, aber nicht die ganzen Dokumente, sondern nur den Namen der Dokumente in einer Liste - verwenden Sie dafür eine single-threaded, multi-threaded oder eine multi-process

Implementierung? Begründen, etc:
- Wenn der client nebenbei nix anders tun muss, ist ein single-threaded client am schnellsten.
Wie sieht das Kommunikationsmodell zu Ihrer vorherigen Implementierung aus, wenn Ihr Client mit mehreren Server kommuniziert?:
- Dann kann ich die Serveraufrufe mit einer multi-threaded Implementierung paralell ablaufen lassen und so schneller werden. Der Client startet für jeden Server-Aufruf einen eigenen Thread, der dann jeweils auf seine Server-Response wartet. Die Threads blockieren sich nicht gegenseitig, und können parallel ablaufen.

Vorteile von Multithreaded Processes:
Parallelism: Speedup computing by putting threads on different CPUs (while shared data is in the shared main memory)
Large applications
Software Engineering: Dedicated threads for dedicated tasks

Should an OS kernel provide threads, or should they be implemented as user-level packages?

User-space:
Alle Threads liegen auf einem Prozessor, Operationen sind aber viel schneller
Vorteile: Billig, da man nur Addressbereiche nehmen und freigeben muss, einfaches Context Switching
Nachteil: Alle vom Kernel angebotenen Services werden im Auftrag des Prozesses ausgeführt in dem sich der Thread befindet.

Kernel-space:
Threads können auf verschiedene Prozessoren aufgeteilt werden, Operationen (z.B. Switch) sind aber langsamer
Vorteil: Thread-blockende Operationen sind kein Problem.
Nachteil: Alle gethreadeten Aufgaben müssen vom Kernel ausgeführt werden

→ Kernel und User-level Threads zu einem Konzept mischen

20. **Stateless vs. Statefull Server**

Server ist grundsätzlich ein Prozess, welcher an einer bestimmten Addresse auf Requets wartet.
Arten von Servern: Superserver (bietet mehrere Services an), Iterative (nur ein Client gleichzeitig) / Concurrent Server (mehrere gleichzeitig)

- Stateless:
Hat keine Information über Zustand des Clients
Aufrufe des Clients sind unabhängig voneinander (z.B. HTTP)
Zustandsinkonsistenz durch Absturz von Client oder Server wird reduziert
Möglicher Performanceverlust

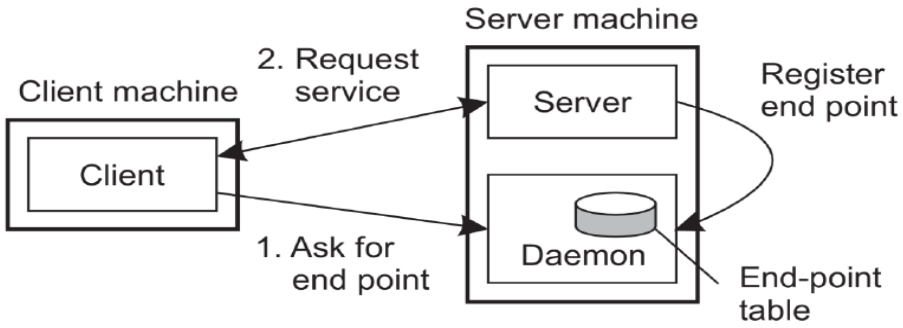
- Statefull:
Hat persistente Informationen über Zustand des Clients
Höhere Performance möglich, da Server zusätzliche Informationen über Client hat
Es wird eine Verbindung zwischen Client und Server aufgebaut, über die dann in beide Richtungen kommuniziert werden kann (z.B. Websocket)

Is it possible to interrupt a server once it has accepted a service request?

Möglichkeit 1: Separater Port / Thread für wichtige Daten
Möglichkeit 2: Out-of-band Communication: Bestimmte Befehle / Daten / Anfragen die vom Server bevorzugt behandelt werden.

Contacting a server:

Entweder über vordefinierten Port oder, falls Port unbekannt:



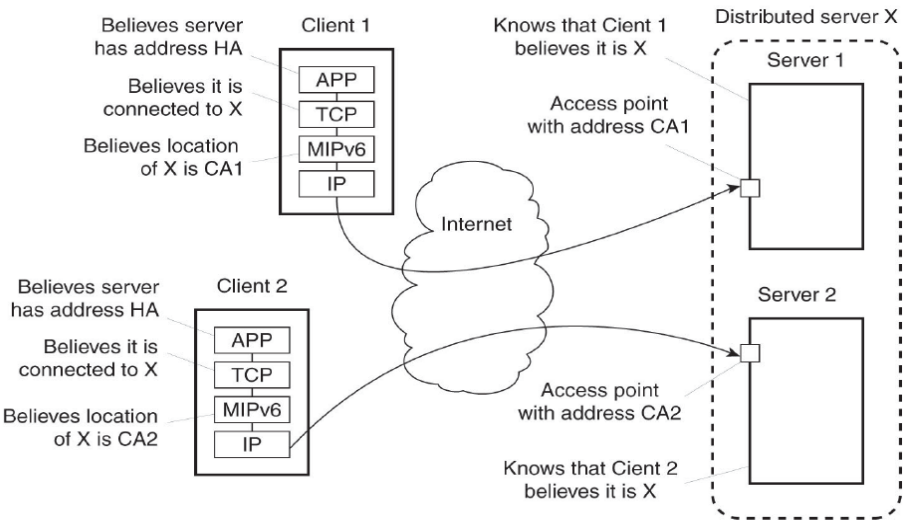
21. **Was ist Mobile Ipv6?**

Mobile IP is an Internet Engineering Task Force (IETF) standard communications protocol that is designed to allow mobile device users to move from one network to another while maintaining a permanent IP address.

21. **Distributed Server Addressing und Route optimization (aka. MIPv6 skizzieren ?)**

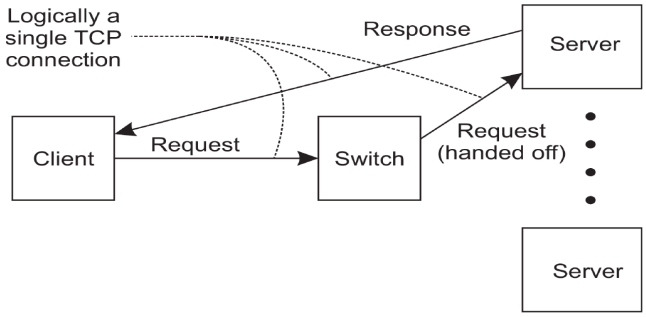
Clients glauben (durch Route optimization) mit nur einem Server zu kommunizieren, in Wirklichkeit kommunizieren sie aber mit verschiedenen Servern eines distributed Servers.

Clients having Mobile IPv6 can transparently set up a connection to any peer:
Route optimization:
Client *C* sets up a connection to 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* (Handoff (Umweg) über HA nicht mehr notwendig)



Was ist TCP Handoff?

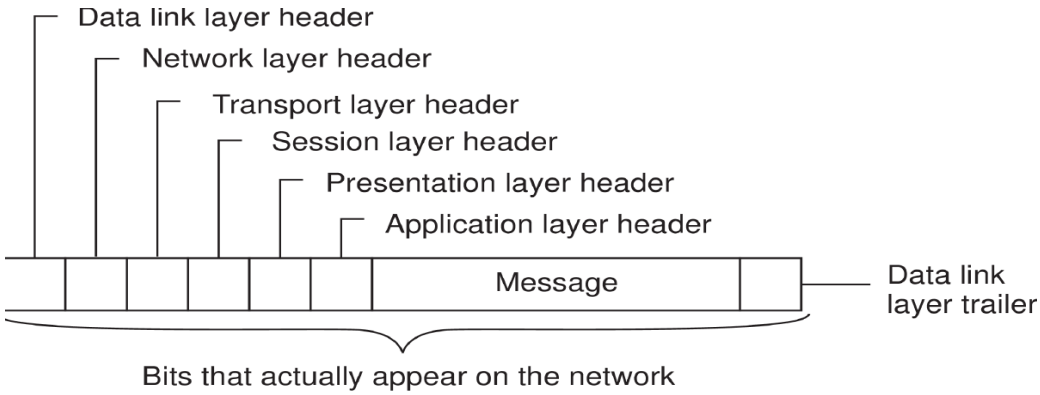
Wird für Request handling / dispatching verwendet. Client sendet Request and Switch, Switch leitet Anfrage an geeignetsten Server weiter, Server sendet Antwort direkt an Client. Vorteil: Das ist alles nur eine TCP Connection



ISO / OSI Modell

7	**Application Layer** (Anwendungsschicht)	**HTTP, HTTPS, SMTP, FTP**
6	**Presentation Layer** (Darstellungsschicht)	
5	**Session Layer** (Sitzungsschicht)	
4	**Transport Layer** (Transportschicht)	**Internet: UDP, TCP**
3	**Network Layer** (Vermittlungsschicht)	**Internet: IP (v4, v6)**
2	**Data Link Layer** (Sicherungsschicht)	**LAN, MAN High-Speed LAN**
1	**Physical Layer** (Bitübertragung)	

Wenn jetzt zB 2 Einheiten über dieses Modell kommunizieren, müssen gewisse Metadaten ausgetauscht werden um bestimmte Eigenschaften festzulegen. Kommt dann etwas völlig anderes an als erwartet bzw ausgemacht, stimmt etwas nicht. Bei Verteilten Systemen Schicht 3 oft unterste Schicht (1 und 2 ignoriert). Am wichtigsten für VS ist Transport Layer



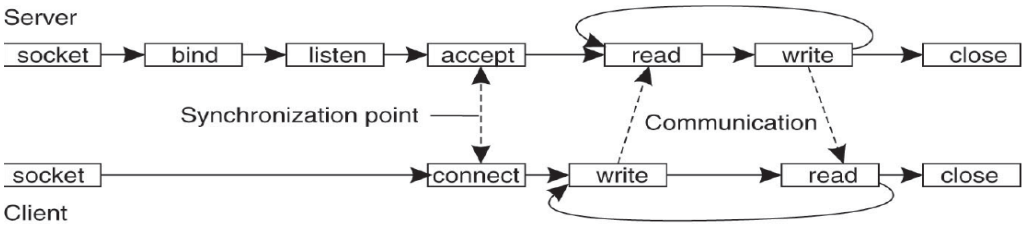
Nicht zwingend notwendig gesamten Stack zu nutzen, Middleware kann Teile davon übernehmen!

22. **Die Idee der Sockets skizzieren**

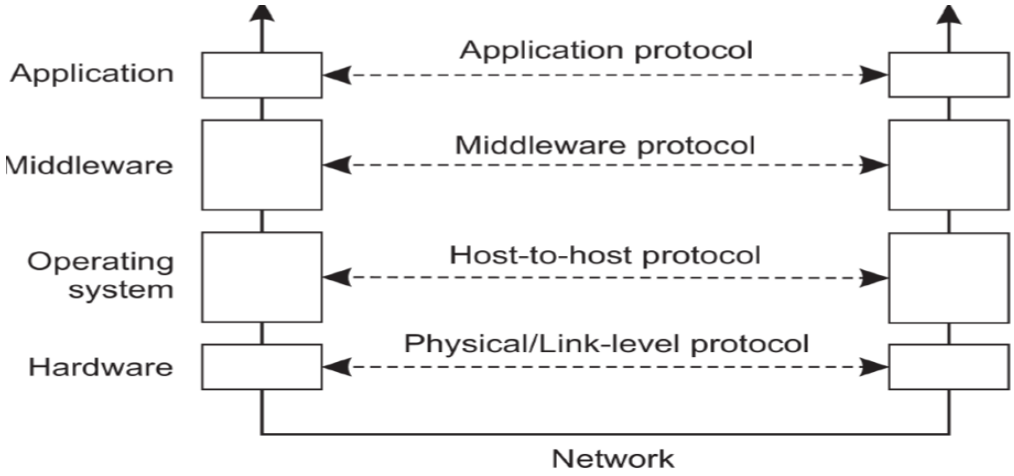
Sockets provide a communication point at a resource through which messages can be sent. This type of communication interface is universal across all systems and therefore masks any differences between the actual networking technology of two different machines. Furthermore, a socket provides a channel (or in Java a stream) to receive data from the socket/ communication partner and also send data. Sockets are in the Transport Layer.

Transport-level socket programming via socket interfaces:
•Designed for low-level system, high-performance, resource-constrained communication
•Socket interfaces:
•Very popular, supported in almost all programming languages and operating systems
•Client: Connects, sends, and then receives data through sockets
•Server: Binds, listens/accepts, receives incoming data, processes this data, and sends the result back to the client.

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection



23. **Was ist eine Middleware?**
Häufig verwendete Funktionalität wird in eine Middleware-Schicht ausgelagert und kann von verschiedenen Applikationen verwendet werden. Ziel: Konsistenz und Fault Tolerance. Es kann gleichzeitig mehrere Instanzen einer Middleware geben.

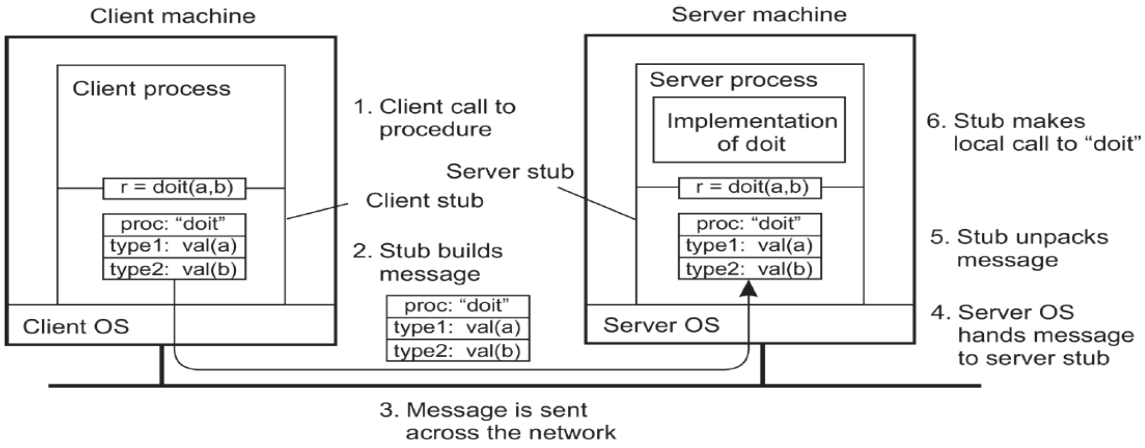


24. **Was ist der Unterschied zwischen „persistent“ und „transient“ communication?**
- Persistent: Nachricht wird am Server solange gespeichert, bis sie zugestellt wurde
- Transient: Nachricht wird vom Server gelöscht, wenn er sie nicht an einen weiteren Server leiten kann.

Eine typische Client/Server Architektur ist transient und synchron:
- Client und Server müssen gleichzeitig online sein
- Client schickt Request ab und wartet auf Antwort (Blockiert)
- Server wartet „nur” auf eingehende requests und arbeitet diese ab.

Nachteile: Clients müssen auf reply warten, Fehler müssen sofort behandelt werden

25. **Was sind RPCs (Remote Procedure Calls)?**
Ein Client ruft eine remote procedure beim Server auf und wartet bis der Server mit dem Ergebnis des Aufrufes antwortet. Wird oft von Middleware übernommen. Asynchroner RPC: Client wartet nicht auf Antwort sondern macht weiter sobald Server Empfang bestätigt.

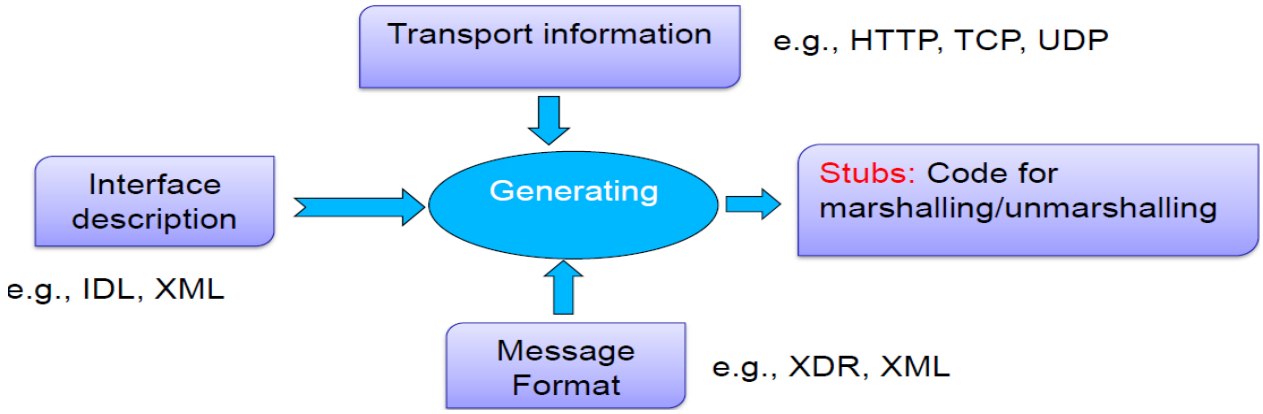


RPC-Beispiele:
XML-RPC: XML für Nachrichten, HTTP für Transport
JSON-RPC: JSON für Nachrichten, HTTP und / oder TCP/IP für Transport, weniger Overhead

26. **Was ist Marshalling / Unmarshalling?**
- Marshalling: Die Umwandlung von strukurierten oder elementaren Daten in ein Format für die Netzwerkübertragung (z.B. JSON: json_encode())
- Unmarshalling: Die Rückführung von Netzwerkdaten in strukturierte oder elementare Daten (z.B. json_decode())

27. **Was ist ein Stub?**
Ein Stub ist ein Programmcode der an Stelle eines anderen Programmcodes steht (Ein Ersatz). In verteilten Systemen kann mit einer Stub-Komponente sowie mit einer normalen lokalen Komponente gearbeitet werden (z.B. stellt sie Methodenaufrufe zur Verfügung). Anstatt die Anfrage aber abzuarbeiten, wird sie in einen Netzwerkaufruf (Marshalling) umgewandelt.
Ein Stub kapselt weiters Informationen bez. der Übertragung und meistens Interface Beschreibung.

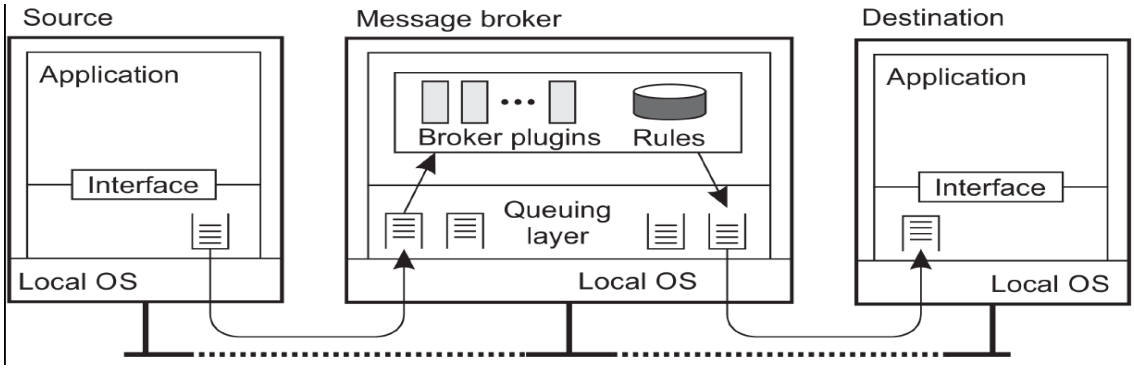
Stub-Generierung:



RMI (Remote Method Invocation) und Remote Object Call
RMI Client sucht in RMI Registry nach entsprechenden Objekten /Funktionen und kann dieses am RMI Server aufrufen. Ohne RMI registry praktisch normaler RPC.

Message Broker:

Zentralisierte Komponente / nodes:
transformiert eintreffende Nachrichten in entsprechendes Format für Empfänger, Anwendungs-Gateway:



28. **Multicast erklären**
Multicast bedeutet, dass ich Daten zu mehreren Empfängern schicke.
Publish/Subscribe - Konzept:
- Subsciber hören auf Veröffentlichungen eines bestimmten Types X (subscribing(X))
- Wenn ich jetzt eine Multicast-Nachricht des Types X publishe (publish(X, message), wird sie dem Message-Broker übergeben, der wiederum alle Subscriber informiert. Stichwort: Entkopplung von Sender und Empfänger.
Application-Level Multicast:
basiert auf überliegendem Netzwerk, aufgebaut aus direkten Verbindungen zwischen Peers, ein „Overlay-Netzwerk“ auf einem bestehenden Netzwerk (z.B. Internet) vom physikalischen Netzwerk unabhängig, da vom TCP/IP Layer abstrahiert, separates addressing Schema
z.B. Peer2Peer Systeme
Was hat das mit Multicasting zu tun? Overlay-Netzwerk wird für Multicasting verwendet
Flooding-based Multicasting:
Nachricht m wird von P an seine Nachbarn gesendet, diese senden Nachricht weiter, aber nicht wieder an P, und nur wenn Nachbar selbst die Nachricht noch nicht erhalten hat

29. **Man musste sagen, was aus {RPC (Remote Procedure Call), Publish/Subscribe, noch irgendwas} geignet ist, wenn man (i) über den Kommunikationspartner überhaupt keine Infos braucht und (ii) wenn der Empfänger/Sender nicht gleichzeitig online sein muss oder so**
Das Publish/Subscribe-Pattern wäre hier das richtige, da weder der Publisher nähere Infos über den Empfänger braucht, noch der Subscriber den Sender kennen muss (i).
Des Weiteren kann bei diesem Pattern, das eine Art Queue besitzt, der Empfänger die Nachrichten auch später abholen (ii).

30. **Connection-oriented und Connection-less Kommunikation erklären + jeweils ein Beispiel**
- Connection-Oriented: Baut auf dem Prinzip von Sockets auf und verlangt einen Verbindungsaufbau, allerdings ist die Kommunikation bei bestehendem Kommunikationskanal dann einfacher (früher bei herkömmlicher Telefonie wurde durch den Verbindungsaufbau zusätzlich auch noch die Leitung reserviert). Beispiel: Socket basierter Chat
- Vorteile: steht eine Verbindung, ist der Datenaustausch einfach: Empfänger gefunden, Reihenfolge bleibt erhalten, Ressourcen reserviert, etc.
- Nachteile: der Aufbau einer Verbindung ist komplex und zeitintensiv, insbesondere wenn viele Stationen miteinander kommunizieren

- Connectionless: Spart sich den Aufbau einer Verbindung, jedoch ist die Adressierung der einzelnen Pakete dafür schwieriger. Beispiel: Normaler HTTP Request
- Vorteile: kein Verwaltungsaufwand durch Verbindungsaufbau
- Nachteile: Adressierung der Daten komplizierter, da Pakete unabhängig voneinander durch das Netz befördert werden.

31. Skizzieren eines Send-Vorgangs mit blockierendem Socket bei dem Client nach dem send Weiterarbeiten will (Auslagerung des send()-Befehls in anderen Prozess)
1. Client startet Prozess t1 und arbeitet normal weiter
2. t1 startet den Request zum Server und wartet auf dessen Antwort
3. Server arbeitet Request ab
4. Server antwortet dem Client-Prozess t1
5. t1 informiert Client, dass eine Antwort da ist.

32. Was ist Gossip-based Data Dissemination?
In einem Netzwerk teile ich Updates nur meinen Nachbarn mit. Wenn mein Nachbar das Update bereits kennt, hört er auf den „Gossip“ weiterzuerzählen mit einer Wahrscheinlichkeit von 1/k. => „Gossiping” garantiert also nicht, dass alle Nodes immer über alle Updates bescheid wissen.

33. **Can a service have multiple servers placed in different machines?**
Yes. A DS can be transparent regarding the location of its (sub-) processes and resources. Also, in multi-tier architectures, parts like application and database are often split and located on several machines.

34. **What are the benefits of group communication? Give some concrete examples (e.g., in P2P and social networks).**
With group communication a set of entities (nodes, users, or hosts) can be aggregated as a new entity group and be identified by a name/id/address entity. Every host can now send messages to the whole group instead of having to remember each node's name. Examples: "groups" or "circles" in social network apps like google+ or facebook.

Messaging:

Zielt auf high-level persistente, asynchrone Kommunikation ab:
- Prozesse senden sich gegenseitig Nachrichten, welche gequeued werden
- Absender muss nicht auf Antwort warten
- Middleware sorgt für Fault Tolerance

35. **Communication programming: consider Message oriented transient communications using Socket and using Message Passing Interface: a. Which layers are they designed for? (1 p), b. Which types of application styles/models are they suitable for? (1 p)**

- Message oriented: Nachrichten werden asynchron zwischen Client und Sever übertragen, ist grundsätzlich persistent. Operationen: PUT, GET (blocking), POLL, NOTIFY.
- Transient: Nachrichten werden gelöscht, wenn sie nicht weitergeleitet werden können
- MPI: a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation." MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.

- Berkeley Sockets
    - Layer: Transportlayer (TCP/IP)
    - Normale Sockets, sind ggf. langsamer als MPI
    - ggeignet für Client to client chat

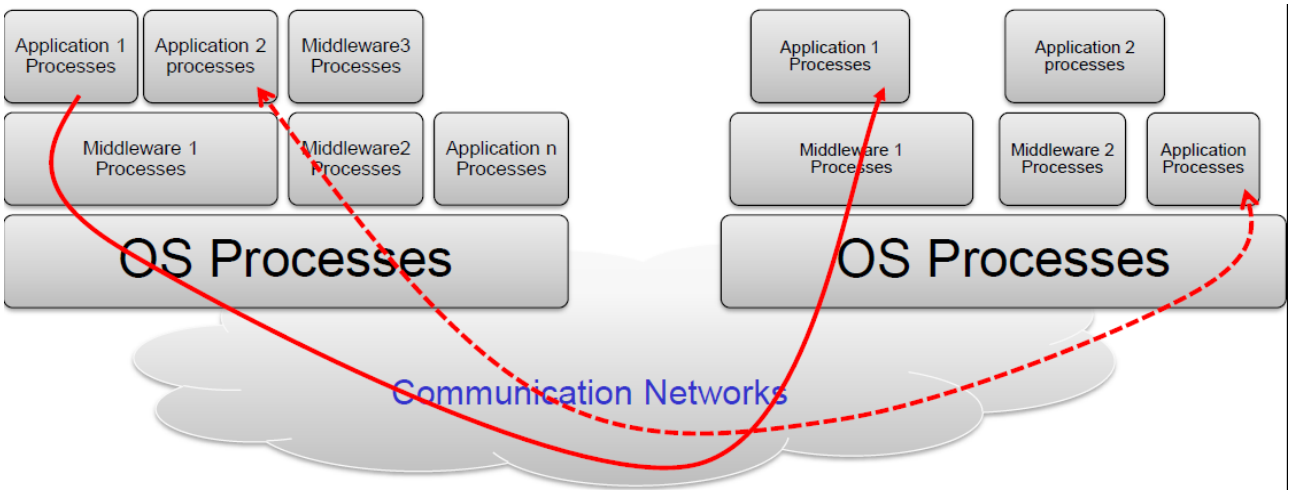
- Message Passing Interface:
    - Layer: Unterschiedlich, eher Session Layer, kann aber auch genauso Sockets verwenden
    - Einsatz: High Speed Interconnection Networks (Cluster)
    - geeignet für Kommunikation mit vielen Clients

36. **Mehrere Erzeuger möchten Produktinformationen an mehrere Verbraucher übermitteln, was ist besser geeignet MOM oder RPC und warum, plus Skizze des Ablaufs**

- MOM: Message Oriented Middleware oder Message Queuing Systems:
Asynchrone persistente Messages zu denen sich Clients subscriben können
Skalierbares message handling
unterschiedliche Kommunikationsmuster
- RPC: Client ruft Remote Procedure am Server auf, der antwortet dann mit dem Ergebnis.
- Ich würde MOM nehmen, da einfacher und schneller. Erzeuger publishen Informationen für Produkt p1, p2, p3, … . Verbrauche subscriben zu den Produkten p1, p2, p3, … und bekommen die Updates nur für ihr(e) Produkt(e).

37. **Was hat es mit Marshalling und Unmarshalling auf sich, wozu braucht man es**
Daten in Netzwerk-Nachricht umwandeln (Marshalling), Unmarshalling: Netzwerk-Nachricht wieder in Daten umwandeln.
Beispiel: $a = new Class(„Franz“, „Wilding“);
Marshalling $a -> JSON: ‚{„Franz“, „Wilding“}‘
Unmarshalling: JSON -> $a: new Class(„Franz“, „Wilding“);

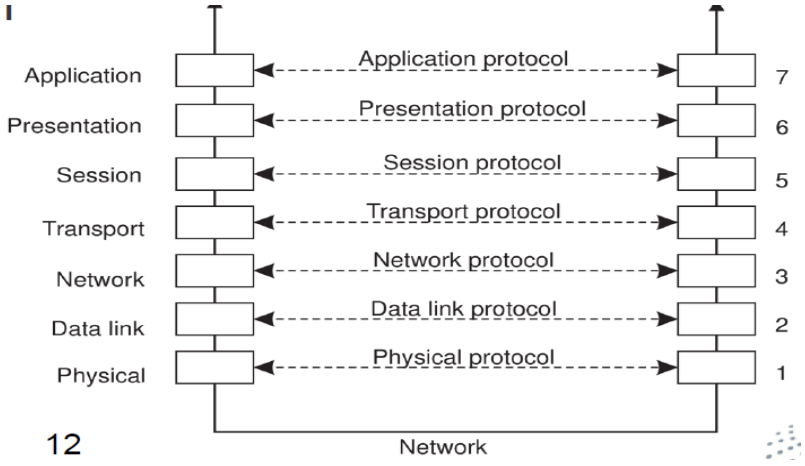
Communication Entities und allgemeine Kommunikation in VS



Kommunikation in verteilten Systemen:
- zwischen Prozessen mit einer einzelnen Anwendung / Middleware / Service
- zwischen Prozessen die zu unterschiedlichen Anwendung / Middleware / Service gehören
- zwischen Knoten, die kein Prozess-Konzept haben (z.B. Sensoren)

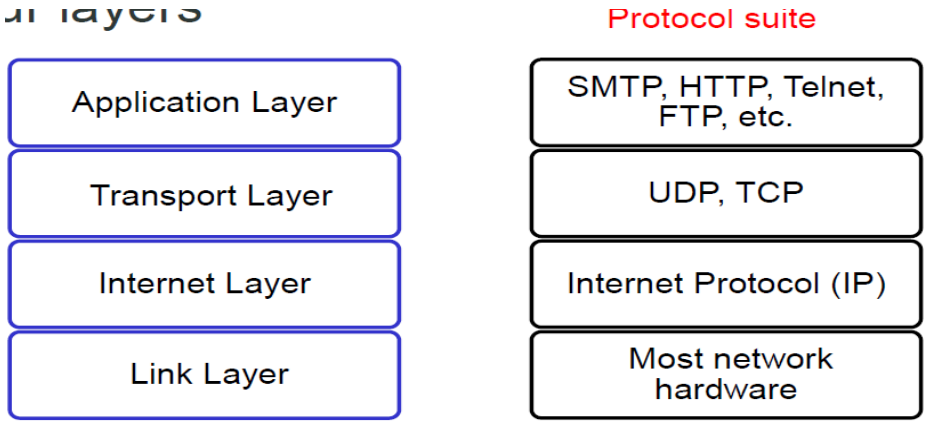
Interprocess Communication:

Based on low-level message passing offered by the underlying network
Communication entities: Processes
ISO/OSI Modell:



TCP/IP

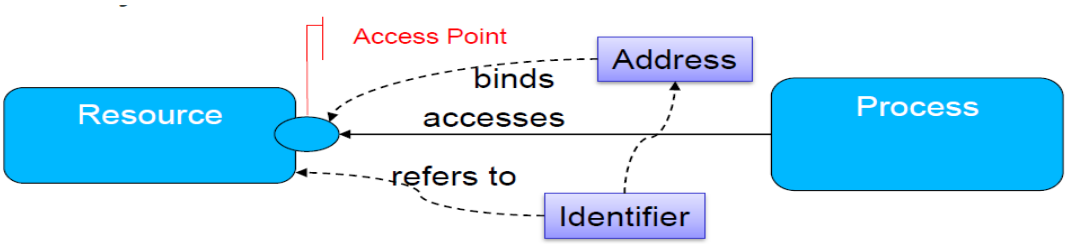
bekanntestes Protokoll im Internet. 4 Schichten:



Naming

38. **Was ist ein(e) „Entity“? Was ist ein Name? Was ist ein Identifier? Was ist eine Address?**
Entity: Irgendein Objekt (in einem distributed system), eine Datei, Drucker, Host, Endpoint, User, …
Name: Set von Bits / zeichen, der eine Entity, oder eine Gruppe von Entities in einem gewissen Kontext identifiziert, hat an sich selbst keine Bedeutung
Identifier: einzigartige Bezeichnung, der eine Entity grundsätzlich identifiziert
Address: Konkreter Ort, wo eine Entity gefunden werden kann, Name eines Access Points
Access Point: notwendig um eine Entity anzusprechen, access points sind entities die mit einer Addresse benannt sind

39. **Wieso sind Naming Systems so komplex?**
- Es gibt die unterschiedlichsten Typen von Entities
- Es gibt Abhängigkeiten auf unterschiedlichen Ebenen zwischen diesen Entities (Network, Data Link)
- Es gibt so extrem viele Entities (User, Hosts, …) aber wir müssen sie alle identifizieren



40. **Was ist ein Name Space?**
- Beinhaltet alle Namen, die bei einem Service verwaltet werden. Es kann auch validate Namen in einem Name-Space geben, die keiner Entity zugewiesen sind.
- Ein Alias ist ein Name, der auf einen anderen Namen zeigt.
- Eine Naming domain ist ein Name-Space mit einer einzelnen Administrations-Authoritiy für ihre Namen (Beispiel: nic.at ist für alle *.at Domains zuständig)

Name Space: Namen werden in einem Verteilten System mithilfe eines Namespace organisiert. Ein Name identifiziert ein Objekt. Zur eindeutigen Zuordnung ist jedoch der entsprechende Kontext – eben der Namensraum zu beachten. Hierarchische Namen können als beschriftete gerichtete Graphen dargestellt werden.

Durch '**Aliases** können mehrere Namen auf die gleiche Entität zeigen.
Hardlinks: mehrere absolute Pfade verweisen auf den selben Knoten im Graphen
Symbolic Links: speichern absoluten Pfad zu einer Entität

Durch Mounting können Namensräume miteinander kombiniert werden. (benötigt: access protocol, server, mounting point).
Closure Mechanismus: zu wissen, wie und wo die Namensauflösung beginnen soll. Auswahl des ersten Knotens.
Bsp. Unix: um absoluten Verzeichnispfad (/users/home) aufzulösen, muss dem Dateisystem der Root-Knoten „/“ bekannt sein. Der tatsächliche Offset des Root-Knotens ist im Superblock des logischen Laufwerks kodiert.
Bsp. DNS: Der Closure Mechanismus bei DNS sind die IP-Adressen der 13 DNS-Root-Server
Leaf nodes: repräsentieren Entität, Directory Node: „Verzeichnisknoten“; Nodes sind auch Entitäten

Naming Design Principles

Data models/structures for naming services
- information about names
- Strongly related to database topics
Processes in naming services
- E.g., Creation, management, update, query, and resolution activities (for example: how do we map a name to an address)
- Interaction protocols

Name space
- Contains all valid names recognized and managed by a service
- A valid name might not be bound to any entity

Naming domain
- Name space with a single administrative authority which manages names for the name space
Name resolution
- A process to look up information/attributes from a name

Naming design is based on specific system organizations and characteristics.
Structures and characteristics of names are based on different purposes.

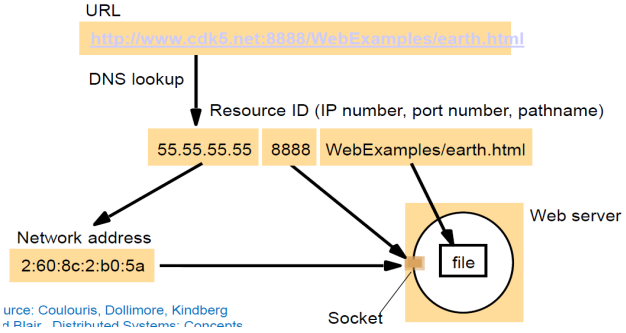
Beispiele:
- Netzwerk ← → Ethernet: Identifier: IP und MAC Addresse. Name resolution: Netzwerkaddresse zu Data Link Addresse
- P2P System: Identifier: m-bit key, Name resolution: Distributed hash tables.

Data structure:
- Can be simple, no structure at all, e.g., a set of bits:
  $ uuid
  bcff7102-3632-11e3-8d4a-0050b6590a3a
- Can be complex
- Include several data items to reflect different aspects on a single entity
 - Names can include location information / reference or not

Readability:
- Human-readable or machine-processable formats

Diverse name-to-address binding mechanisms
    - How a name is associated with an address or how an identifier is associated with an entity
    - Names can be changed over the time and names are valid in specific contexts
        - Dynamic or static binding?
- Distributed or centralized management
    - Naming data is distributed over many places or not
- Discovery/Resolution protocol
    - Names are managed by distributed services
    - None/single system can have a complete view of all names

Naming scheme example:



41. **Was ist Name resolution?**
Der Prozess, bei dem ich Informationen/Attribute von einem Namen nachschauen gehe. Z.B.: Die IP-Adresse zu einer Domain.

Broadcast based Name Resolution:
1. Ich sende eine Broadcast Nachricht an alle Nodes um den Access Point der entity „en“ zu finden: broadcast(ID(en));
2. Alle Nodes bekommen die Nachricht, aber nur einer wird mir antworten
=> So funktioniert zum Beispiel ARP, um die MAC-Adresse zu einer IP-Adresse in einem Netzwerk zu bekommen.

Dynamic systems:
- Ein Set von Nodes verwaltet entities, es gibt keine zentrale Koordination
- Neue Nodes können jeder Zeit dazu kommen, Nodes können auch jeder Zeit ausfallen
- Viele Nodes, aber eine Node kennt nur einen Teil der anderen Nodes.
=> So funktionieren z.B. große P2P Systeme, Chord, CAN (Content Addressable Network)

Mounting:
Ein Verzeichnis eines remote-Servers kann an einer gewissen Stelle als lokale Node eingehängt (gemounted) werden

42. **Was sind distributed Hash Tables?**
- m-bits werden für den keyspace verwendet
- Node identifier nodeID ist ein key im keyspace
- Eine Entity en wird durch die Hashfunktion k = hash(en) identifiziert
- Eine Node mit ID p verwaltet entities, deren Schlüssel in der Range von p sind
  If (k=hash(en) $\in$ range(p)), then put (k, en) will store en in p
- Nodes leiten Nachrichten weiter bis sie am richtigen Node ankommen.
2^m Entities können mit einem m-bit keyspace verwaltet werden.
(Bei 8 Computern brauchen wir also m=3 --> Weil 2^3 = 8)

CORD + Finger Table

Verringert notwendige Zeit um einen verantwortlichen Node zu finden mithilfe von „Finger-Table“ pointing

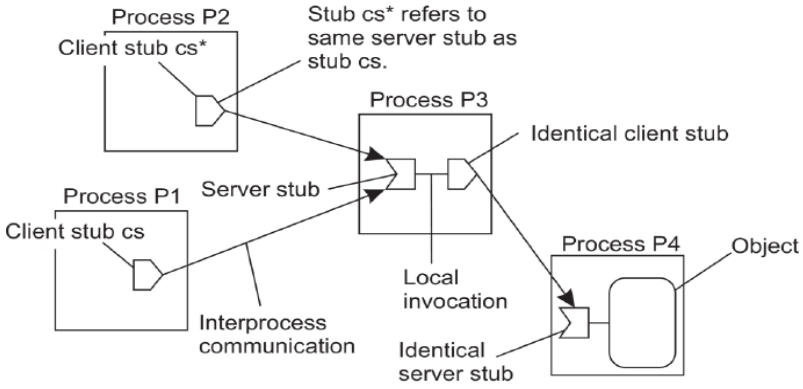
43. **Was ist Flat-Naming vs. Structured Naming?**
- Flat: Der Name ist einfach nur eine Abfolge von bits. Keine Struktur drinnen (Domains sind nicht flat). Beispiel: MAC Addresse, m-bt Nummern in distributed hash tables
- Structured: Der Name kann als Graph angesehen werden mit normalen Nodes und Leaf-Nodes.
Flat ist wann sinnvoll:
Für kleine Systeme, wo die Auflösung der Namen (Name Resolution) sehr trivial verläuft. (Keine verschiedenen Arten von Name Resolution notwendig)
Closure Mechanism:
Finde heraus, von wo die name resolution gestartet wird

Broadcasting

Es wird der Identifier einer Entität an alle Nodes geschickt, Nodes sehen nach ob sie diese Entity haben, wenn ja, schicken sie die Addresse (z.B. Address Resolution Protocol ARP

Forward Pointer

Bewegt sich eine Entität, lässt sie einen Pointer zum neuen Ort zurück. Client merkt davon nichts da Kette solcher Pointer einfach durchgegangen wird



Dynamic systems

Nodes formen ein System ohne zentrale Koordination. z.B. Overlay Network. Nodes können jederzeit dazukommen, wegfallen, ausfallen. Große Anzahl Nodes, aber jeder Node kennt nur eine gewisse Anzahl anderer Nodes. z.B. in großen P2P Systemen

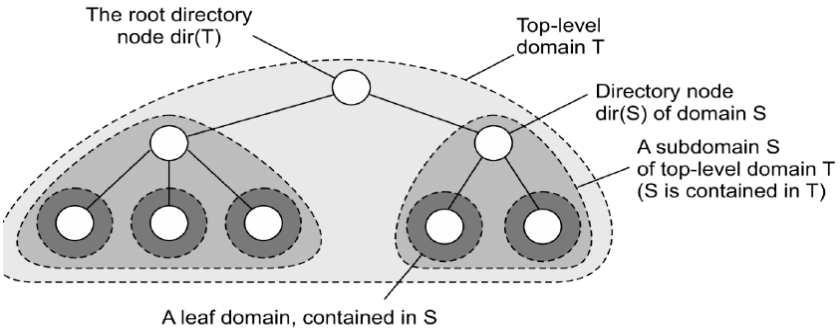
Beispiele:
Chord: Bekanntes P2P Protokoll. Circular linked list (jeder Node hat Link zu nächstem Node im Uhrzeigersinn). Ring Netzwerk mit 0 bis 2^m-1 Nodepositionen.

Distributed Indexing and Resolving

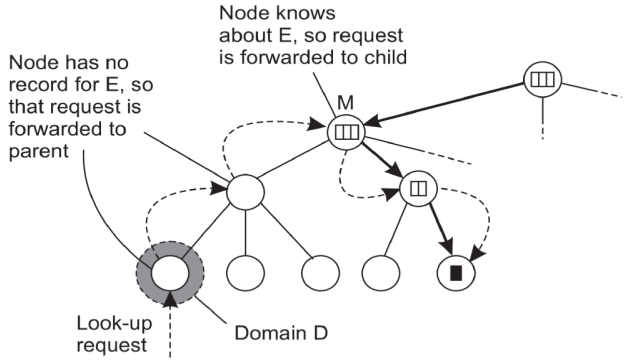
Node A sendet request für Gegenstand D an Knoten im DHT (distributed Hash Table). Request wird mit O(logN) Sprüngen zu Zielnode weitergeleitet. Ziel sendet D zu A.Directory-Node respräsentiert.

Hierarchical Location Services (HLS)

Grundidee: Bau einen großen Suchbaum dessen unterliegendes Netzwerk in hierarchische Domänen unterteilt ist. Jede Domain wird von einem separaten Directory-Node repräsentiert.



Lookup Operation:



44. **Hard links vs. symbolic links?**
- Hard links: mehrere absolute Pfade zeigen zur selben Node
- Symbolic links: Leaf-Node speichert einen anderen absoluten Pfad, auf den dieser Leaf-Node zeigt

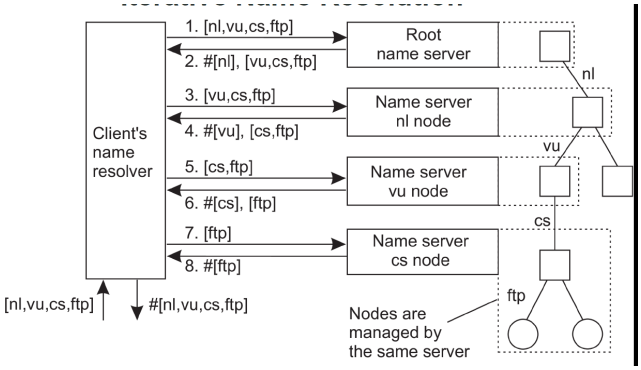
45. **Was ist distributed name management?**
Mehrere Server sind für das Name-Management zuständig
- Global Layer: Root-Nodes

- Administration Layer: Directory-Nodes

- Managerial Layer: Nodes, die sich oft ändern

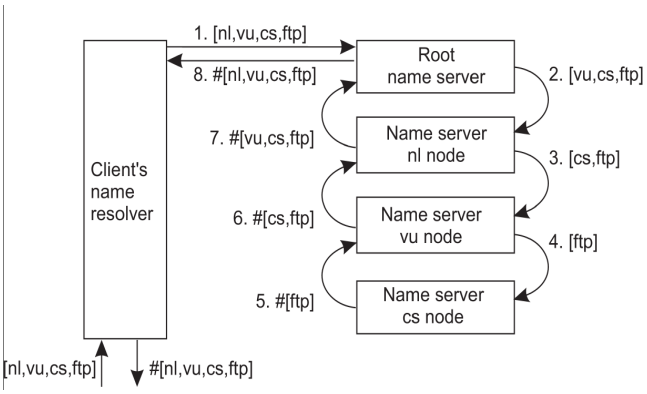
Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

46. **Iterative Name Resolution vs. Recursive Name Resolution?**
Beispiel Auflösung: „www.fwilding.at“:
- Iterative Name Resolution client side:
1. Client ruft NS_ROOT mit „at“ auf, der antwortet mit „NS2 verwaltet *.at domains“.
2. Client ruft NS2 mit „fwilding“ auf, der antwortet mit: „NS3 verwaltet *.fwilding.at domains“.
3. Client ruft NS3 mit „www“ auf, der antwortet mit „10.11.12.13“



- Iterative Name Resolution server side:

1. Client ruft NS_ROOT mit „www.fwilding.at“ auf.
2. NS_ROOT ruft NS2 mit „fwilding“ auf, der antwortet mit: „NS3…“.
3. NS_ROOT ruft NS3 mit „www“ auf, der antwortet mit: „10.11.12.13“.
4. NS_ROOT antwortet Client mit „10.11.12.13“.



- Recursive Name Resolution:
1. Client ruft NS_ROOT mit „www.fwilding.at“ auf.
2. NS_ROOT ruft NS2 mit „www.fwilding“ auf.
3. NS2 ruft NS3 mit „www“ auf.
4. NS3 antwortet NS2 mit „10.11.12.13“.
5. NS2 antwortet NS_ROOT mit „10.11.12.13“.
6. NS_ROOT antwortet Client mit „10.11.12.13“.

Vorteil rekursiver Namesauflösung
1) Durch Caching ist es möglich, dass weitere Anfragen gleich an den entsprechenden Server weitergeleitet werden. Zum Beispiel wenn jemand institut.tuwien.at auflösen möchte, dann Schickt der .at Nameserver eine Anfrage auf .tuwien und diese wiederum auf .institut. Wenn ein zweiter Client dann nach test.institut.tuwien.at sucht, weiß der Nameserver, dass er diese Anfrage direkt auf .institut weiterleiten kann.
2) Kommunikationskosten werden reduziert. Angenommen Client ist in den USA. Die Nameserver für test.institut.tuwien.at alle in Österreich. Client schickt eine Anfrage an Nameserver und dieser schickt ihm die komplette Auflösung zurück. Bei der iterativen Resolution müsste der Client mehrmals zwischen USA und den einzelnen Nameservern in Österreich kommunizieren.

Nachteil rekursiver Namensauflösung
Hoher Aufwand für einen Nameserver, der einen absoluten Pfad auflösen muss. Wird im Global Layer gar nicht verwendet.
Beispiel: (genaue Bezeichnung der einzelnen Elemente wichtig!)

Domain Name System (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

String representation of a pathname:
•List its labels
•Separate thelabelsbya dot
•Root isrepresentedbya dot
•Example: flits.cs.vu.nl.

47. **DNS Information Records**

Type of record	Associated entity	Description
SOA	Zone	Holds information on the represented zone
A	Host	Contains an IP address of the host this node represents
MX	Domain	Refers to a mail server to handle mail addressed to this node
SRV	Domain	Refers to a server handling a specific service
NS	Zone	Refers to a name server that implements the represented zone
CNAME	Node	Symbolic link with the primary name of the represented node
PTR	Host	Contains the canonical name of a host
HINFO	Host	Holds information on the host this node represents
TXT	Any kind	Contains any entity-specific information considered useful

48. **Attribute-Based Naming erklären + ein Beispiel**

Auch directory services genannt, Beim Attribute-based Naming wird eine Eigenschaft einer Entität durch ein Key-Value-Paar
beschrieben. Eine Entität kann in Folge durch ein Set von solchen Attributen beschrieben werden. Beispiel: Entität AustriaInfo. Beispiel: LDAP, RDF (Resource Description Framework)

Attribute	**Value**
CountryName	Austria
Language	German
MemberOfEU	Yes
Capital	Vienna

Warum geht structured / flat naming hier nicht (DNS System)? Kontext.

49. LDAP

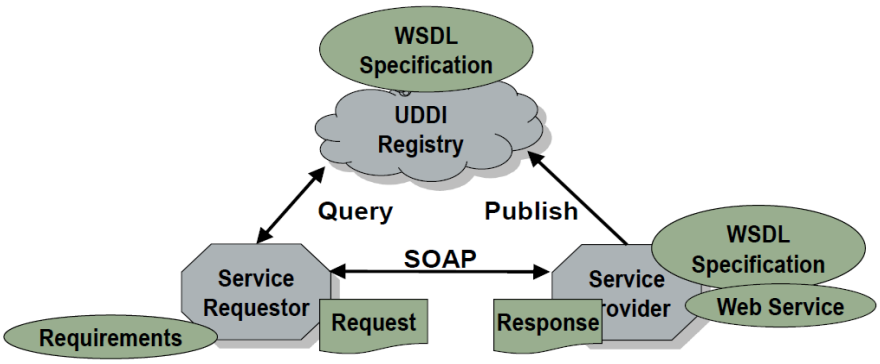
- Lightweight Directory Access Protocol
- Object class: Informationen zu einer Entity werden als tupel(attribute, value) gespeichert
- Directory entry: Entry einer bestimmten Object class
- Directory Information Base (DIB): Collection aller Directory entries
- Directory Information Tree (DIT): Baumstruktur oder Naming-Graph für DIB-Entries
- Jeder Eintrag ist einzigartig als Sequenz aus von naming Attributen benannt

Naming resolution basiert auf querying-Mechanismus
Ablauf:
1. Client schickt query an LDAP-Server
2. LDAP-Server leitet Frage an DIB-Fragment weiter
(3. Antwort ggf. ein Verweis auf einen anderen LDAP-Server, dann query an den anderen Server)
5. LDAP-Server antwortet mit results.

Beispiel: Bei auftretenden Ereignissen sendet ein Türsensor Daten an einen Datenserver. Der Datenserver muss dann die Tür-ID und Gebäude-ID dazu speichern. Welches Naming-Service bietet sich an? LDAP, DHT (distibuted hash table), DNS?
-> Lightweight Directory Access Protocol (LDAP)

50. Was ist ein Webservice - Was ist WSDL?

- Ein Webservice bietet Funktionalität über ein genau definiertes Interface an und ist über das Netzwerk zu erreichen.
- WSDL (Web Service Description Language) - Beschreibt einen Web-Service, z.B: welche Adressen wie aufgerufen werden sollen um welche Funktion zu erreichen.



51. Was ist OpenID?

- Ein Standard um einer Entity „Mensch“ eine eindeutige ID (Als eine URL) für verschiedenste Webservices zu geben.
- Wird von verschiedenen OpenID-Providern angeboten
- ID ist unique, da auch der Provider-Service in den key einbezogen wird.

Wie funktioniert der Login?
1. Client-Login bei Website gestartet
2. Website schickt ein redirect zum OpenID-Provider
3. Client loggt sich auf der OpenID-Provider-Seite ein
4. OpenID-Provider schickt zurück zur Website mit dem authentifizierten User
5. Website kann dann Authorisierung anhand des Users vornehmen
6. Website antwortet Client

52. **Drei wesentliche Eigenschaften von Peer2Peer Systemen.**

- Die Entitäten eines P2P-Systems kommunizieren untereinander durch direkte Verbindungen, nicht über eine zentrale Steuereinheit.
- Alle Entitäten eines P2P-Systems sind gleichgestellt, es gibt keine höherwertigeren Entitäten.
- Jede Entität konsumiert und stellt die gleichen Services bereit (Beispiel: Torrent).

53. **Global Taxi Identifier entwerfen, bei dem es einen Global Server gibt, der den Name-Server des Taxi-Unternehmers liefert und dieser Taxi-Server kann durch die Taxi-ID den Standort des Taxis bestimmen. Welche Dinge müssen in den Global Taxi Identifier?**

Frage 2: Wie sieht hier das Name Resolving aus?

55. **Firma möchte ein Naming System für Kühlschränke einführen (Hersteller, Land, Name, etc.) welche Lösung ist am besten geeignet (LDAP, DHT, oder noch irgendetwas) und warum, plus beispielhafte Namensauflösung skizzieren**

=> Attribute Based?

2. **Entwerfen sie einen globalen Identifier, der auf die Location von Produkten auflösen lässt. Es gibt einen globalen Naming Dienst und jede Company hat einen eigenen Name Server, der zu einem Produkt die Location hat. Jedes Produkt und jede Firma hat einen einzigartigen Identifier. Wie soll also der globale Identifier gewählt werden?**

=> ?

3. **Bei auftretenden Ereignissen sendet ein Türsensor Daten an einen Datenserver. Der Datenserver muss dann die Tür-ID und Gebäude-ID dazu speichern. Welches Naming-Service bietet sich an? LDAP, DHT, DNS?**

Time Synchronisation

56. **Welche generellen Gründe gibt es für Synchronisation?**

- Achieving Accountability: Verantwortlichkeit von Prozessen
- Maintaining Consistency: Konsistente Verarbeitung von Nachrichten
- Establishing Validity: Gültigkeit der Nachrichten schaffen
- Achieving Fairness: Alle Prozesse werden fair behandelt (kommen z.b. ähnlich oft dran)

57. **Physical clocks**

Manchmal brauchen wir exakte Zeit, nicht nur korrekten Ablauf von Ereignissen:

- UTC (Universal Coordinated Time) -> Atom-Uhr sendet Zeit via Radio und Satellit
(Genauigkeit Satellit: +/- 0.5ms.)

Zeitverteilung in verteiltem System mithilfe von UTC:
- Jeder Rechner p hat einen Timer der alle H Sekunden Interrupt generiert. Bei jedem Timer Interrupt tickt die Uhr in p. Die aktuelle Zeit wird dann als Cp(t) (wobei t = UTC-Time) bezeichnet. Im besten Fall haben wir Cp(t) = t, also keine Abweichung von UTC, ist aber fast nicht möglich. Fehler wird mit der Abweichung von dC/dt von 1 (da Idealfall wäre dC/ct = 1 wäre) bezeichnet. Richtlinie: 1-p <= dc/dt <= 1+p (p wird von Hersteller vorgegeben)
- Ziel: Keine zwei Cp1 Cp2 in all meinen Rechnern dürfen sich um mehr als Wert x unterschieden (z.B. x = 3s.) Also muss ich x/2p, (in diesem Fall alle 90s) synchronisieren

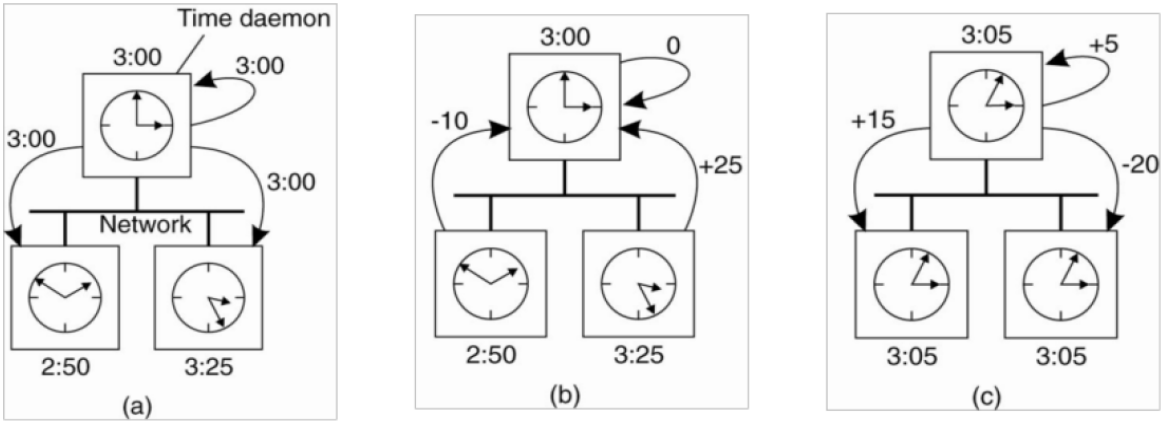
- Synchronisation der Zeit übers Netzwerk kann auf zwei Arten passieren:

- Network Time Protocol (NTP): Server A schickt eine Nachricht an den Timeserver B, bekommt 4 Zeiten zurück (T1 abschicken, T2 beim Server ankommen, T3 vom Server zurückschicken, T4 beim Client ankommen), RTT = Round Trip Time.
Theta = ((T2-T1)+(T3-T4))/2
Ist Theta < 0, läuft die Uhr von A schneller. Bei Theta > 0 ist B schneller.
- Cristian's Algorithm: Ich nehme an, dass die Berechnungszeit des Time-Servers = 0 ist. Dann kann ich Zeit berechnen (Anmerkung: T2=T3=T) : Theta = T + RTT / 2 -T4 = T + RTT/2 – T4, also so lange wie das Packet ca. vom Server zurück braucht. Ich kann den Vorgang öfters wiederholen und dann den Wert mit dem geringsten RTT nehmen.

Symmetric Time Propagation in NTP:
- Je genauer ein Time-Server ist, desto kleiner bezeichne ich sein Stratum. Atom-Uhr: Stratum-0, je weiter der Wert abweicht, desto weiter zähle ich rauf (Stratum-1, Stratum-2, …)

Falls keine zuverlässige reale Uhrzeit vorhanden, wird lokal ein Timeserver erstellt welcher die Synchronisation durchführt. Beispiel:

Berkeley Algorithm:
- Time-Server scannt regelmäßig alle Rechner, berechnet den Durchschnittswert und teilt jedem Rechner mit, wie weit er daneben liegt
- Zeit darf niemals zurückgesetzt (= reset, != rückwärts korrigiert) werden!
- Beispiel:



$$\bar{\delta} = \frac{-10+25+0}{3} = 5 \qquad T_i' = T_i + \delta_i' \qquad \delta_i' = \bar{\delta} - \delta_i$$

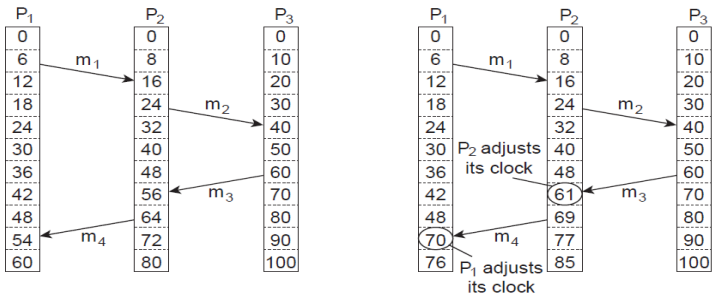
Also: Time daemon errechnet Durchschnitt aus zeitlichen Unterschieden gegenüber allen Teilnehmern (bezieht dabei seine eigene Zeit mit Differenz 0 ein). Der daraus resultiernde Wert (kann positiv oder negativ sein) wird an die aktuelle Zeit des Time Daemons hinzugefügt (bei negativer Zeit wird Zeit natürlich zurückgestellt). Nachdem der Time Daemon dadurch eine neue „Haupt-Uhrzeit“ berechnet hat, teilt er allen anderen im Netzwerk mit, wie weit diese ihre Uhr vor bzw. zurückstellen müssen.

Diese Methode ist nur innerhalb eines LAN Netzwerks sinnvoll, da ansonsten zu viele Unterschiede in der Zeit auftreten (unterschiedliche Zeitzonen, Übertragungsdauer bei großer Distanz, …)

58. **Logische Uhren (Lamport Clocks)**

- Sind einfache event-counters

basieren auf **Happened-before relationship:**
- a → b (a kommt vor b)
- Wenn a das Senden einer Nachricht ist, und b das empfangen dieser Nachricht, gilt a → b.
- Wenn a → b und b → c dann a → c



Problem: bei Synchronisation zwischen mehreren Prozessen nicht ausreichend!→ Lamport Clocks:

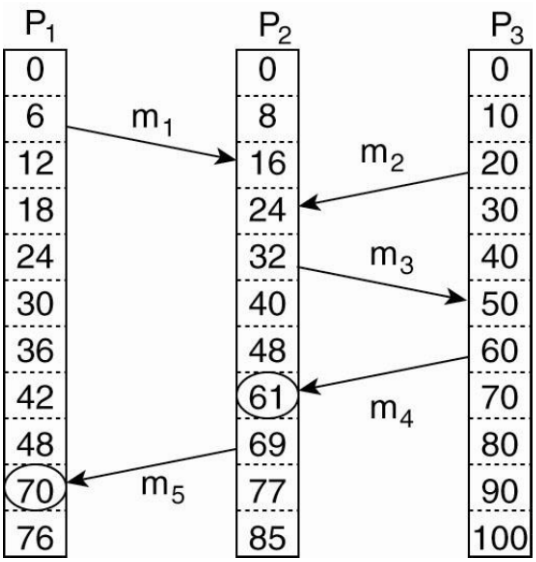
Jeder Prozess hängt an eine Nachricht einen Zeitstempel C(e), mit folgenden Bedingungen:
- dass der Zeitstempel C(a) < C(b) ist, wenn a und b 2 Eereignisse im selben Prozess sind und a → b gilt.
- Wenn a eine Nachricht schickt, die b empfängt, dann auch a -> b und daher C(a) < C(b)

Problem: Wie soll das ohne globale Uhr gehen Antwort: Jeder Prozess hat einen counter Ci (logische Uhr)

1) Bevor ein Ereignis in Pi (ein Prozess) ausgeführt wird, wird Ci um 1 erhöht
2) Immer wenn eine Nachricht in Pi gesendet wird, bekommt sie (vor dem Senden) den Timestamp ts(m) = Ci.
3) Immer wenn eine Nachricht in Pj empfangen wird, setze den Cj auf max{Cj, ts(m)} und führe Schritt 1 vor der Verarbeitung der Nachricht aus.

Die Ci-Anpassung erfolgt in der Middleware-Layer (Zwischen Application-Layer und Network-Layer)

Wichtig: C(a) < C(b) bedeutet nicht unbedingt, dass a -> b gilt! Beispiel:



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

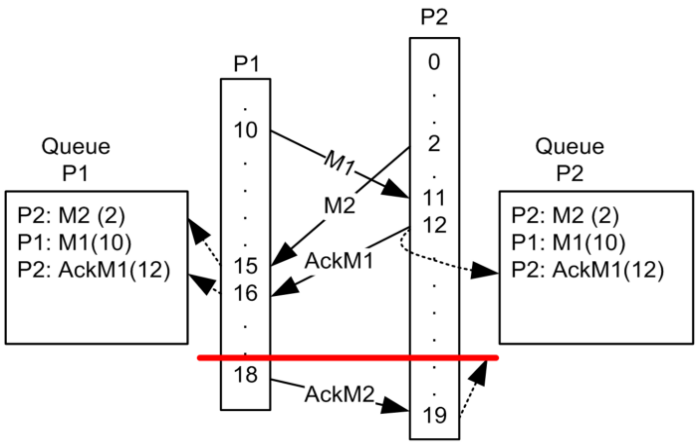
recv (m1) < send (m2):
We do not know their relationship just by comparing clock values!

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

59. **Totally ordered multicast**

- Bei „gleichzeitigen“ updates muss die Konsistenz der Reihenfolge immer gewährleistet sein
- Jeder Prozess hat eine eigene Queue und sendet timestamped Messages an alle anderen Prozesse, die diese Timestamps in ihre Queue speichern. Die Queue ist nach den timestamps sortiert. -> Ricart & Agrawala ist so ähnlich

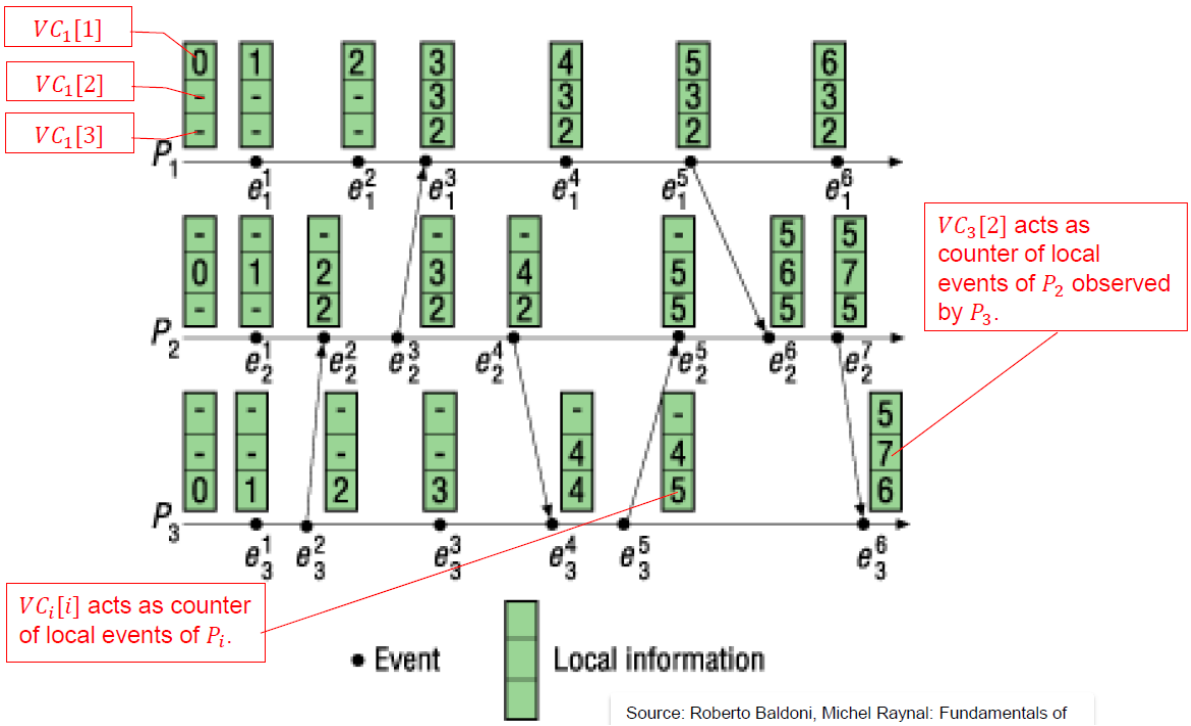
Also leitet Pj eine Nachricht msg i erst zu seiner Anwendung weiter, wenn:
- msg i an der Spitze der queue von j steht
- es für jeden anderen Prozess Pk bereits eine msg k in der queue von j mit einem größeren Zeitstempel gibt.



Vector Clocks

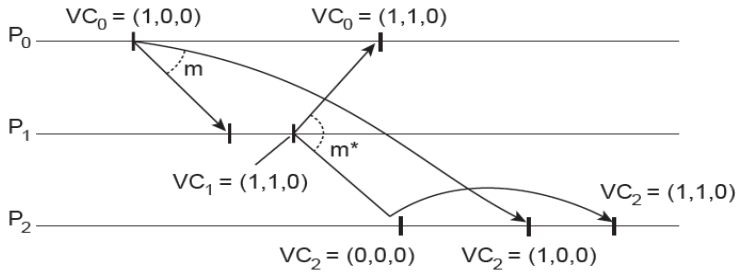
- Jeder Prozess Pi hat eine Vektoruhr VCi. VCi[i] ist die Anzahl der Ereignisse die in Pi vorgekommen sind. VCi[j]=k bedeutet, dass Pi weiß dass k Ereignisse die im kausalen Zusammenhang mit Pi stehen können bereits in Pj geschehen sind.

Anmerkung: Wird z.B von P1 eine Nachricht an P2 geschickt, ist das für P2 ein Ereignis bei dem er seine Vektoruhr erhöht.



60. **Causally ordered multicast**

- Zu jeder Nachricht wird ein Vektor gesendet. Der Sender-Prozess pi erhöht NUR beim Senden den Wert an Stelle VCi[i] +1. Der Empfänger-Prozess pj updatet nicht sofort seine Vektoruhr und leitet die Message auch nicht weiter, sondern erst, wenn gilt:
1) ts[i] = VCj[i] +1 (Das ist die nächste Nachricht die Pj von Pi erwartet)
2) für alle k != i gilt: ts[k] <= Vcj[k] (Pj kennt bereits alle anderen Nachrichten die Pi von anderen Prozessen erhalten hat)
Sind diese Bedingungen erfüllt wird die Nachricht geliefert und Vector Clock wird geupdatet:
Vcj[k] = max(VCj[k],ts[k])



Weiteres Beispiel:
Prozess 2: lokale Vektoruhr VC2 = [0,2,2], erhält ts(m) = [1,3,0] von P0.

Nachricht ist von P0 und besagt, dass P0 3 Ereignisse von P1 und kein Ereignis von P2 gesehen hat. P2s Uhr besagt, dass P2 0 Ereignisse von P0, 2 von P1 und 2 von P2 gesehen hat. Da dies die erste Nachricht von P0 zu P2 ist, ist die erste Bedingung erfüllt. P2 muss jedoch noch auf die Nachricht 3 von P1 warten, die P0 bereits erhalten hat. Daher wird die Nachricht verschoben bis VC2[1] den Wert 3 erreicht.

61. **Für was benötigt man Time-Sync? Einen Grund, + Erklärung**

Für Konsistenz und die Korrektheit von Prozessen und deren Logs wird Zeitsynchronisation benötigt, wenn mehrere physikalische Maschinen interagieren sollen. Andernfalls könnten Inkonsistenzen auftreten, die zu Softwarefehlern führen oder die Nachvollziehbarkeit (Logs) beeinträchtigen könnten (Beispiel: Maschine A führt Transaktion mit Maschine B durch; A loggt Startzeitpunkt, B den Endzeitpunkt; Uhr von B geht vor und Endzeitpunkt liegt laut Logs schlussendlich vor Startzeitpunkt).

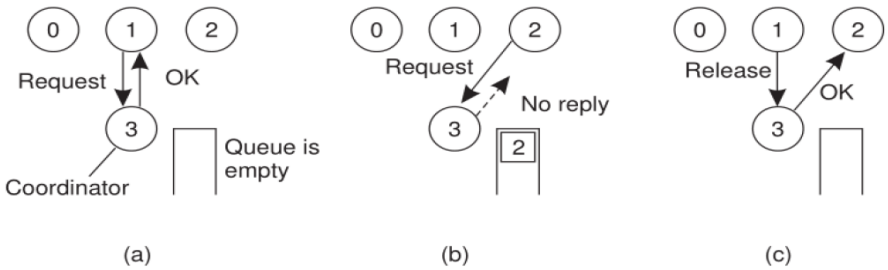
Mutual Exclusion allgemeine Ansätze:

Token basiert: Token (besondere Nachricht) kann nur immer von einem Prozess besetzt werden, wird immer weitergegeben. Vorteil: Starvation / Deadlocks werden vermieden.

Permission-based: Bevor Prozess Zugriff auf Ressource bekommt muss er zuerst Anfrage stellen und Berechtigung bekommen.

62. **Zentralisiertes Modell für Mutual Exclusion**

Beim zentralisierten Modell gibt es eine Koordinatoreinheit (zentraler Server/Prozess), welche die Berechtigungen zum Zugriff auf kritische Ressourcen vergibt und wieder einsammelt.
Vorteile: fairer Algorithmus.
Nachteil: Wenn der Koordinator crasht, dann kann im Prinzip niemand mehr auf die Ressource zugreifen (Prozess kann nicht zwischen denied und gecrashtem Koordinator unterscheiden). In einem großen System kann ein single-Koordinator auch zum bottleneck werden.

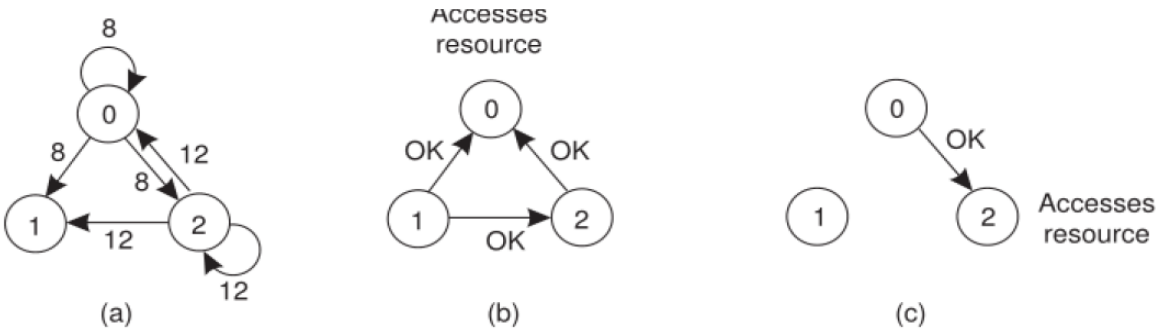


Dezentralisiertes Modell für Mutual Exclusion

Annahme: Jede Resource ist n mal repliziert und jedes Replikat hat einen eigenen Koordinator. Für Berechtigung muss Anfrage an mehrere Koordinatoren gesendet werden und mehr als n/2 müssen zustimmen. Problem: Der Koordinator kann sich zwar schnell wieder neustarten, verliert aber jegliche Information darüber, wer wann auf die Ressource vorhin zugegriffen hat bzw. wer in der Queue wartet. Somit kann es dann passieren, dass es zu einem unerlaubten Zugriff kommt, so dass der Koordinator einem anderen Prozess Zugriff gewährt, weil dieser es vorhin vergessen hat. Weiteres Problem: Viele Knoten fragen und keiner bekommt eine Mehrheit → keiner bekommt Zugriff.

Distributed Modell für Mutal Exclusion (Ricard Agrawala)

Prozess schickt Request an alle anderen Prozesse. Wenn Empfänger kein Interesse an Ressource hat, antwortet er mit OK.
Wenn Empfänger an Ressource arbeitet, keine Antwort und stellt Request in Queue.
Wenn Empfänger auch Anfrage gestellt, wird Timestamp verglichen, niedrigere gewinnt. Wenn Empfänger gewinnt schickt er keine Antwort, sonst OK.
Nachteile: Nicht sehr effizient, 2(n-1) (Anforderungen und Bestätigungen) Nachrichten pro Eintrag. Größtes Problem: n points of failure: Reagiert ein Knoten nicht können andere Knoten nicht weitermachen (wird als „denied“ identifiziert)! Kann verbessert durch konkrete „reject“ Nachrichten. Jeder Prozess muss zudem immer wissen, wer zur Gruppe gehört (Group membership management oder multicast benötigt)



a) Prozess 0 und 2 wollen (fast) gleichzeitig zugreifen. 1 Hat kein Interesse, gibt an beide OK.
b) 2 checkt eigene Timestamp, sieht dass 0 niedriger ist (also „gewinnt“), sendet OK. 0 sieht dass er gewinnt, antwortet nicht und stellt Anforderung von 2 in Warteschlange.

c) 0 ist fertig, hat 2 oben in Queue, sendet OK an 2

Token Ring Probleme

Token kreist herum wenn niemand Zugriff braucht → Starvation wenn Token nicht ankommt.
Nicht reagierende Prozesse werden übersprungen, alle müssen Ring-Topologie beibehalten.
Erhalten von token muss mit akn bestätigt werden (lost token detection)
Wichtig: Jeder Prozess muss Reihenfolge der Prozesse kennen (z.B. für Überspringen bei gecrashtem Prozess)

63. **Why is understanding time and space uncoupling important for implementing communication in distributed systems?**

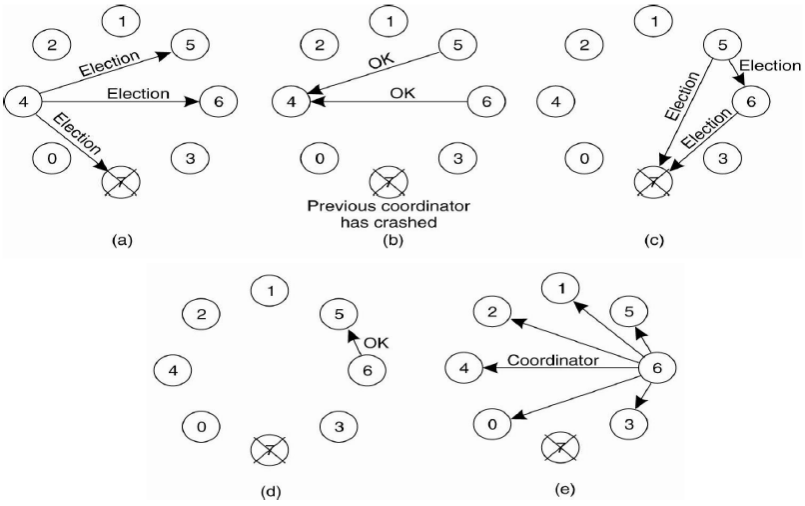
In a distributed system you cannot rely on a same understanding of time and time may differ between long distances. Also, delay and transmission time can be huge around long distances. Two nodes cannot rely on the physical time matching each other.

65. **Election algorithms**

- Viele Algorithmen brauchen einen coordinator, die Frage ist, wie aus allen beteiligten Prozessen der coordinator dynamisch bestimmt werden kann.
- In vielen Systemen passiert die Auswahl einfach per Hand -> centralized solution

Annahmen:
- Jeder Prozess hat unique ID. (grundsätzlich wird Prozess mit höchster ID Koordnator)
- Jeder Prozess kennt alle anderen Prozesse und deren Details (ID, group membership?, aber nicht ob sie zur Zeit aktiv sind.
- Jeder kann mitkriegen wenn der Koordinator weg ist und eine Election starten.

Bullying
- Jeder Prozess hat ein Gewicht; Der schwerste Prozess soll der Koordinator sein
- Jeder Prozess kann eine Election starten: Er schickt sein Gewicht an alle anderen Prozesse.
- Wenn ein anderer Prozess schwerer ist, schickt er eine „take-over“ Nachricht an den herausfordernden Prozess, und sendet sein Gewicht an alle weiteren Prozesse.
- Wenn kein Prozess mit einer „take-over“ Nachricht antwortet, gewinnt der herausfordernde Prozess und schickt eine „victory“ Nachricht an alle Prozesse



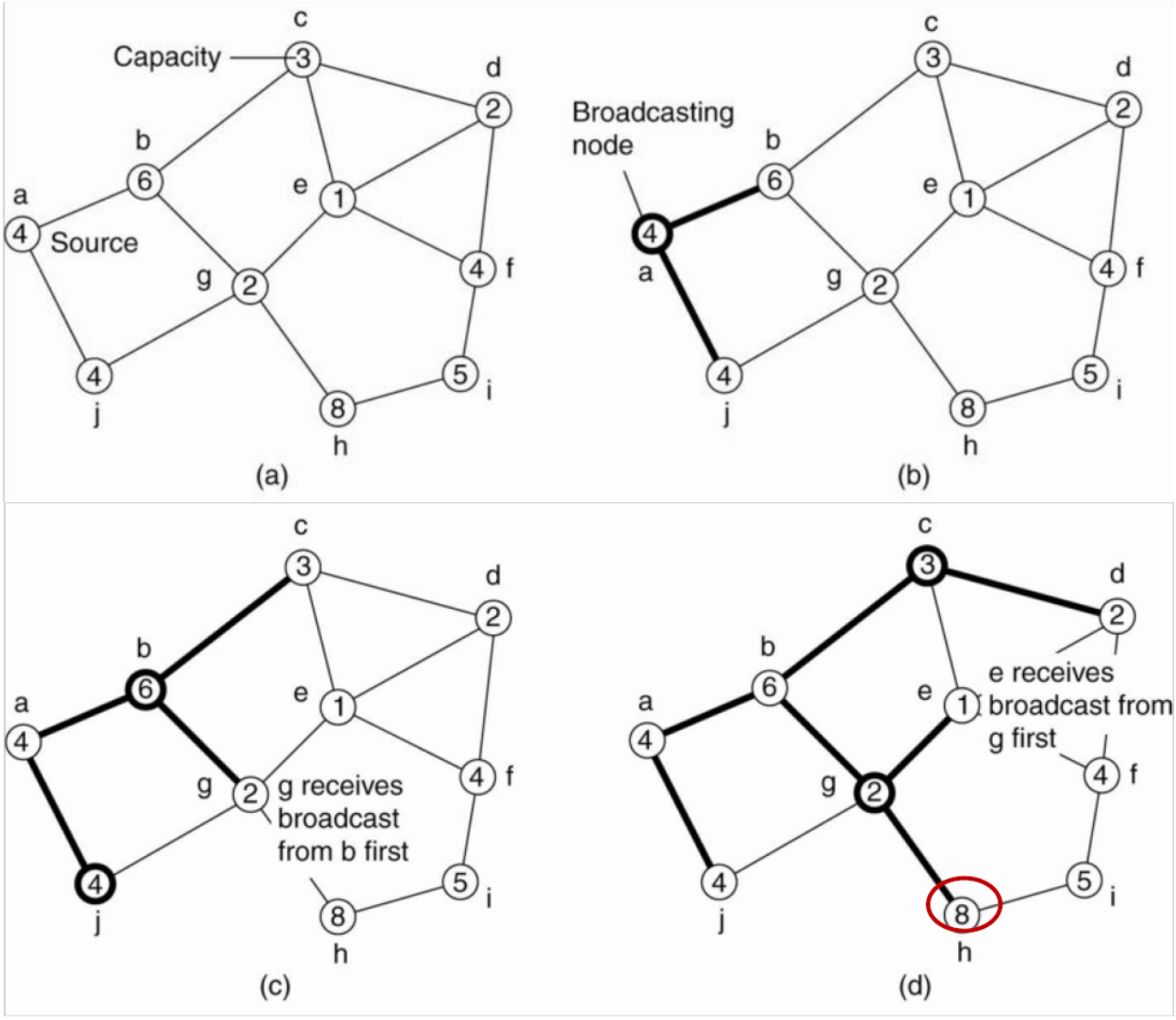
: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

Im Ring
- Wie bei Bullying, nur wird die Nachricht von Prozess zu Prozess weitergegeben und jedes mal wird das Gewicht des Prozesses dazu genommen. Wenn die Nachricht einmal durch den Kreis gelaufen ist, haben alle ihr Gewicht bekannt gegeben. Der herausfordernden Prozess kann dann den Koordinator bekannt geben.

In wireless environments

Notwendig weil bei traditionellen Election Algorithmen zuverlässige Nachrichtenübertragung angenommen wird, und dass sich die Netztopologie nicht verändert und diese allen Knoten bekannt ist. Bei wireless environments muss also bei einer Election auch die aktuelle Netzwerktopologie festgestellt werden

Knoten a baut Stammbaum auf, erhält Details aller Knoten, gibt am Schluss Knoten mit höchster Kapazität Auftrag, Koordinator zu werden.



Fault Tolerance

66. Means for Dependability erklären.

Dependability bedeutet: Wenn meine Komponente Services von anderen Komponenten verwendet, hängt die korrekte Ausführung meiner Komponente von den anderen Services / deren Korrektheit ab.
Attribute der Dependability:
- Availability: Service ist sofort korrekt verfügbar
- Reliability: Service ist durchgehend korrekt verfügbar
- Safety: Keine katastrophalen Folgen
- Integrity: Keine unpassenden Systemänderungen, die Integrität des Systems bleibt erhalten
- Maintainability: System lässt sich modifizieren
- Confidentiality

67. Was ist „Failure“, „Error“, „Fault?

- Fault: Der Grund für den Error
- Error: Die Abweichung eines Services (Der Grund für den Failure)
- Failure: Ein verwendeter Service weicht von seiner Korrektness ab / Ist nicht mehr erreichbar

Fault -> (führt zu) Error -> (führt zu) Failure

Beispiel:
1. Fault: Ein Software-Bug tritt auf, solange aber niemand den entsprechenden Code ausführt, wird der Fault nicht aktiv.
2. Error: Der Code wird ausgeführt und der Fault wird aktiv. Auswirkung: Zum Beispiel eine falsche Berechnung.
3. Failure: Wenn der Error nicht identifiziert wird, führt das zu einem inkorrekten Service, da ja der Wert falsch berechnet wird.

68. Welche Fault Klassen gibt es?

- Development faults
- Operational faults
- Hardware faults
- Software faults
- Malicious faults
- Accidental faults
- Incompetence faults
…

69. Welche Failure Models gibt es?

- Crash Failure: Service stoppt, funktioniert aber bis zum Stop korrekt
- Omission Failure: Service reagiert nicht mehr
- Timing / Performance Failure: Antwort auf Request kommt zu spät
- Response / Common-Mode Failure: Reproduzierbarer Fehler mit korrektem Input, aber falschem Output (Value failure oder State-transition failure)
- Arbitrary Failure: Willkürlicher Fehler, die unvorhersehbar auftreten (Manchmal funktioniert es nicht)

70. Gegenmaßnahmen zu Faults

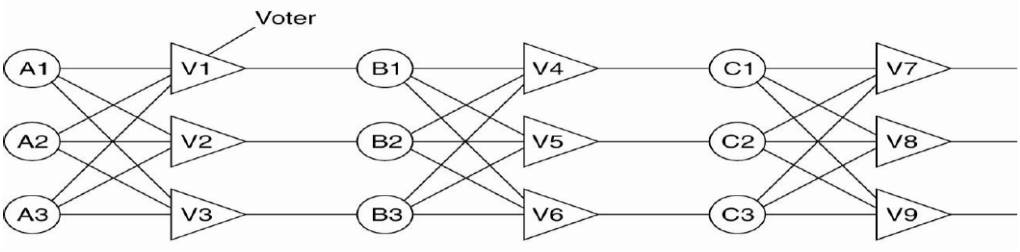
- Fault Prevention: Auftreten von Faults verhindern
- Fault Forecasting: Schätze aktuelle und zukünftige Fehler und ihre Konsequenzen

- Fault Tolerance: WICHTIG!: Verhindere, dass ein Fault zu einem Failure führt, verberge das Auftreten von Faults
- Fault Removal: Reduziere die Schwere von Faults
Wie kann ich Fault Tolerance erreichen?

- Durch Redundanz: Siehe Frage 71

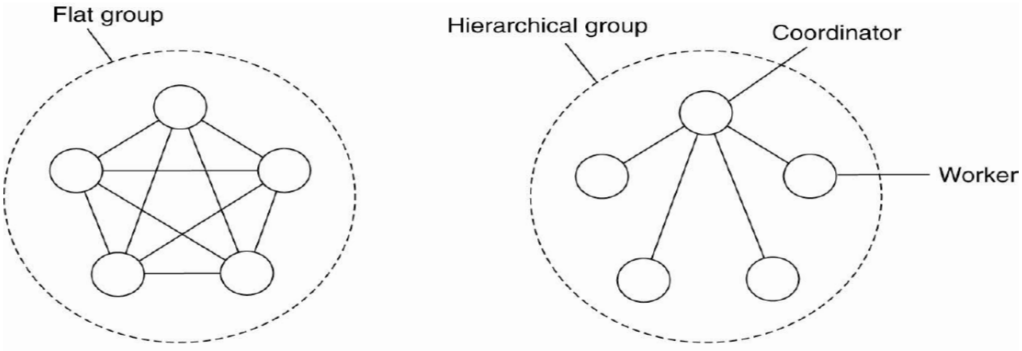
71. Die 3 Typen der Failure Masking by Redundancy angeben

- Information: Extra bits (parity bit) are added to a message to allow a reconstruction of the message in the event that bits become corrupted during transmission (Hamming code)
- Time: Sending messages should repeat in the event that a previous attempt failed
- Permanent / Physical Redundancy: There are backups of data that can be accessed in the event that the primary copy becomes corrupted or lost. Or different implementations of same functionality in different processes
Beispiel permanente Redundanz durch Voter (Am häufigsten vorkommendes signal ist das korrekte und wird weitergeleitet, funktioniert hier nur wenn nur je eine Komponente defekt ist) :



72. **Die zwei Arten, wie Prozesse in Gruppen organisiert werden können + Vorteile/Nachteile**

- Flat: All processes are equal.
Vorteil: the crash of one process does not affect others. Good for fault tolerance.
Nachteil: Communication and coordination is more complicated however, as voting among processes is required.
- Hierarchical:
Vorteil: A coordinator exists within the group and handles balancing the work load among members of the group.
Nachteil: The coordinator presents a single point of failure. This is also not scalable!



73. **Gruppen und das Verstecken von Fehlern**

- k-fault tolerant group: Die Gruppe kann trotz gleichzeitig auftretenden Fehlern in k Gruppen weitermachen (Annahme: Alle Member sind und funktionieren gleich.
- Bei Crash/Omission/Timing failure models muss so eine Gruppe k+1 Member haben
- Bei Arbitrary/Byzantine failure models muss so eine Gruppe 2k+1 Member haben

74. **Byzantine Agreement Problem + Ablauf**

- Byzantine failure: System kann sich beliebig und willkürlich falsch verhalten. Beispiel: Server schickt manchmal falsche Nachricht.
- Idee: Ich habe einen Befehlshaber, der alle Member meiner Gruppe nach ihren Aktionen fragt. Wenn mindestens die Hälfte +1 der Member eine korrekte Aktion vorschlagen, so kann das System weiterhin korrekt funktionieren, da der Befehlshaber annimmt, dass dies die richtige Aktion ist. Deswegen brauchen wir auch 2k+1 Member (2k = korrekt funktionierende Member, daher mit 3 Membern nicht möglich!) für k-fault tolerance.

Beispiel (4 Komponenten, 3. Komponente ist falsch, er sendet x, y, z aus):
1. Alle Nachrichten werden gesendet:
    Member 1 bekommt: (1, 2, x, 4)
    Member 2 bekommt: (1, 2, y, 4)
    Member 3 bekommt: (1, 2, 3, 4)
    Member 4 bekommt: (1, 2, z, 4)
2. Jeder Member gibt seine Vektoren weiter:
    Member 1 bekommt: Von 2: (1, 2, y, 4), Von 3: (a, b, c, d), Von 4: (1, 2, z, 4)
    Member 2 bekommt: Von 1: (1, 2, x, 4), Von 3: (e, f, g, h), Von 4: (1, 2, z, 4)
    Member 4 bekommt: Von 1: (1, 2, x, 4), Von 2: (1, 2, y, 4), Von 3: (i, j, k, l)
3. Wenn es eine Mehrheit zu einem Wert gibt, dann wird dieser in den Result-Vektor übernommen, wenn nicht wird UNKNOWN gesetzt. Result in dem Beispiel: (1,2,UNKNOWN,4)

Mit 3 Komponenten geht's nicht. Beispiel (3 Komponenten, 3. Komponente ist falsch, sendet x, y aus):
1. Alle Nachrichten werden gesendet:
    Member 1 bekommt: (1, 2, x)
    Member 2 bekommt: (1, 2, y)
    Member 3 bekommt: (1, 2, 3)
2. Jeder Member gibt seine Vektoren weiter:
    Member 1 bekommt: Von 2: (1, 2, y), Von 3: (a, b, c)
    Member 2 bekommt: Von 1: (1, 2, x), Von 3: (d, e, f)
3. Es gibt keine Mehrheiten für Werte, Result: (UNKNOWN,UNKNOWN,UNKNOWN)

75. Reliable Client-Server Communication (oder auch: Was kann falsch laufen bei RPC)

- Eine Komponente kann nicht nur falsche Werte schicken, es kann auch die Verbindung zwischen zwei Komponenten ausfallen. Das hat zum Beispiel folgende Gründe + Lösungen:

- Client kann den Server nicht finden -> Einfach dem Client mitteilen, der kümmert sich drum
- Client-Request geht verloren -> Request nochmal senden, Server muss die Differenz der beiden Requests kennen (z.B. Banküberweisung 2mal senden, aber nur 1mal verwerten)

- Server stürzt ab -> Client kann nicht unterscheiden, ob der Server vor oder nach der Abarbeitung des Requests abgestürzt ist. Client bekommt nur die Nachricht, dass Server neugestartet ist. Es kann aber ausgemacht werden, dass Client dann immer den Request nochmal sendet, oder nie. Da muss aber für jede Anwendung eine eigene Lösung gefunden werden.
At-least-once-semantics: The Server guarantees it will carry out an operation at least once, no matter what
2.At-most-once-semantics: The Server guarantees it will carry out an operation at most once.

Was macht der Client wenn er keine Antwort erhält, aber eine Nachricht dass der Server gerebootet hat?

- Immer request erneut senden
- Niemals request erneut senden

- Request nur erneut senden falls kein ACK von Server erhalten
- Request nur erneut senden falls ACK von Server erhalten

- Server-Antwort geht verloren:
Ist der Server gecrasht? Hat er die Aufgabe ausgeführt? Keine wirkliche Lösung außer Operationen idempotent machen, also die doppele Ausführung einer Operation ohne Probleme zu ermöglichen.

- Client stürzt ab (Nachdem sein Request gesendet wurde) -> Server sendet den Response trotzdem (orphan computation), dann gibt es 3 Möglichkeiten:
- Orphan wird vom Client verworfen wenn Response erhalten wird.
- Reincarnation: Client berichtet Server über seinen neustart -> Server verwirft Orphan.
- Expiration: Orphan die älter als T sind werden automatisch verworfen.

76. **Nennen und erklären eine Methode zur Failure-Detection in Verteilten Systemem.**

Fehler sind unvermeidbar, deshalb sollte man einen Weg finden, damit umzugehen, ohne dass das komplette System (bzw. der Service) fehlerhaft wird. Ein gängiger Weg ist das verschleiern eines Fehlers bei der Antwort an den aufrufenden Prozess (Exception abfangen und etwas "Normales" zurück geben, aber keinen Stacktrace o.Ä.).

Distributed File Systems

77. **Was ist das Ziel von Distributed File Systems?**

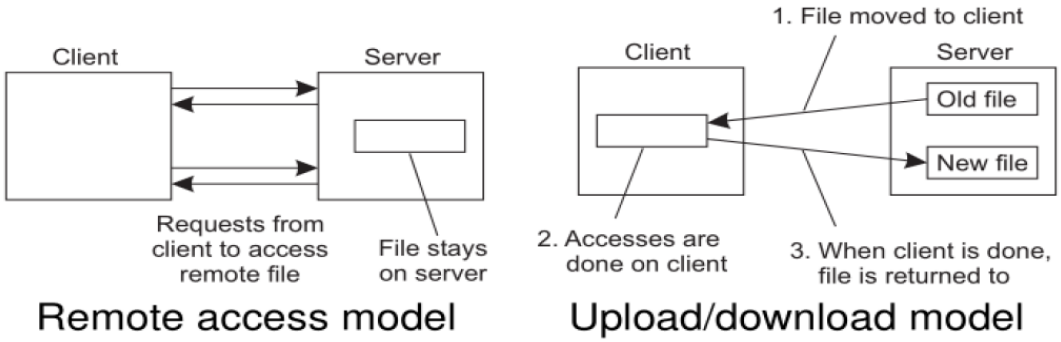
- Versuche das Dateisystem für remote-clients transparent verfügbar zu machen.

Frage: Welche zwei grundsätzlichen Modelle gibt es bei Distributed File Systems und mit welcher kann man am besten Client-side caching implementieren?

Remote access model
- Client ruft Datei am Server auf, die Datei bleibt am Server
- *„Die angeforderte Datei wird nicht kopiert, es kann jedoch auf ihr operiert werden, als wäre sie eine normale Datei auf dem lokalen Rechner (open, close, read, write, ...). (Beispiel: NFS)“*

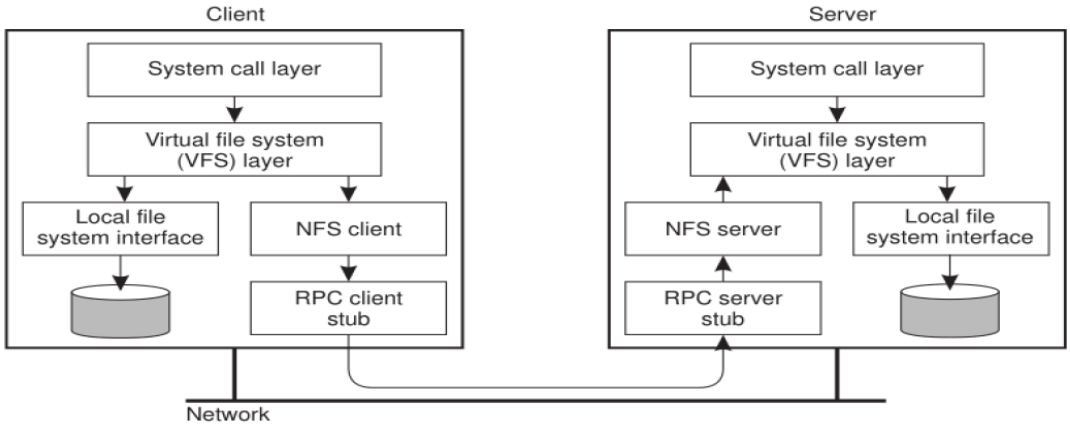
Upload/Download model
- Client lädt Datei vom Server herunter, ruft sie lokal auf und speichert dann die neue Datei wieder auf den Server
- *„Die angeforderte Datei wird heruntergeladen (=kopiert), kann dort modifiziert und wieder hochgeladen werden. Sie wird dann auf dem Server einfach ersetzt. (Beispiel: FTP)“*

=> Zum Cachen ist das Download/Upload Modell besser geeignet, da dort relativ leicht mit Versionierung/Bearbeitungsdatum der Zustand geprüft werden kann.



78. **Was ist die NFS (Network File System) Architektur?**

- NFS implementiert Virtual File System, das eine einheitliche Schnittstelle zu lokalen Dateien sowie zu Remote Dateien bietet
- NFS-Client ruft RPC-Stub-Client auf
- RPC-Stub-Client kommuniziert mit dem Server:
- RPC-Stub-Server, der wiederum mit dem NFS-Server kommuniziert ist Teil des VFS (Virtual File System) des Servers



Ein paar NFS Operationen:
- Create, Symlink, Mkdir, Remove, Open, Close, Lookup, Read, Write, …

What is the main difference between NFS v3 and v4? (1 point), What benefit does this difference provide? (1 point)

- NFS4 ist stateful
- In NFS3 gab es kein Open / Close
- Locking & Mounting ist in NFS4 direkt eingebaut
- NFS4 kann eine Gruppe von Operationen (COMPOUND) gemeinsam an den Server schicken

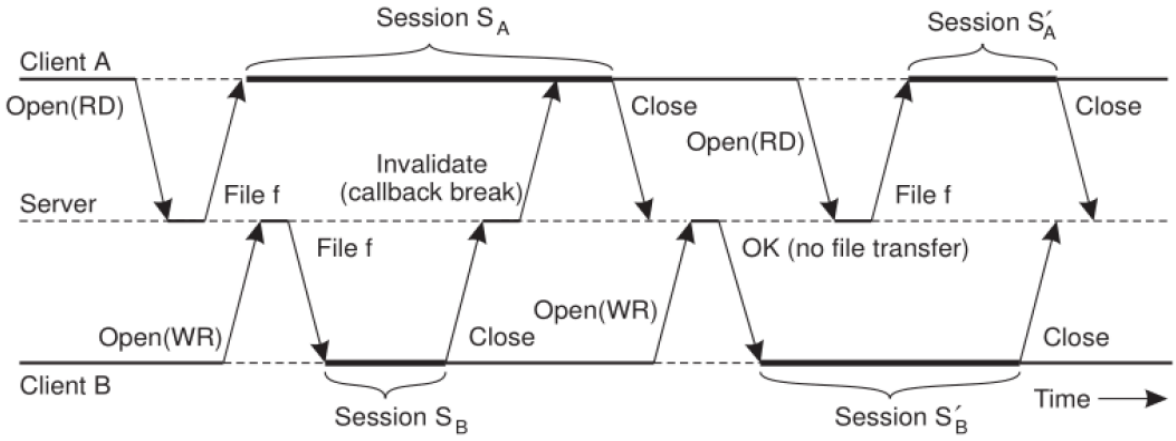
79. **File sharing semantics + Coda**

- Problem: Bei parallelem Read/Write auf Dateien, spielt die Reihenfolge der Operationen eine große Rolle für die Konsistenz.
- Unix semantics: Jede Read-Operation liefert den Effekt der letzten Write-Operation zurück. Geht nur für Remote-Access-Models mit nur einer Dateikopie
- Transaction semantics: Dateisystem erlaubt transactions an einer einzelnen Datei. Problem: Wie wird der nebenläufige Zugriff gemanaged?
- Session semantics: Die Read/Write Auswirkungen werden nur von dem ausführenden Client gesehen

Beispiel: Coda:
- Coda verwendet Transaction semantics
- Coda erlaubt es mehreren Rechnern gleichzeitig mit dem gleichen File-System zu arbeiten

- Ablauf Beispiel:
1. Client 1 öffnet Datei zum lesen Open(RD)
2. Server antwortet mit der Datei
3. Client 2 öffnet Datei zum schreiben Open (WR)
4. Server antwortet mit der Datei
5. Client 2 ändert Datei und speichert sie zurück auf den Server
6. Server benachrichtigt Client 1, dass die Datei invalide geworden ist
7. Client 1 schließt die Datei

- Mit Client-Side-Caching gehts weiter:
8. Client 2 öffnet Datei zum schreiben noch einmal
9. Server antwortet mit OK anstatt der Datei, da Client 2 die aktuelle Version im Cache hat
10. Client 1 öffnet die Datei zum Lesen noch einmal
11. Da sich die Datei durch Client 2 geändert hat, schickt der Server die neue Datei



Caching:
- Client-Side-Caching ist die moderne Variante um Performance zu erhöhen, Server benachrichtigt Client, wenn Cache invalide ist

Replication:
- Server-Side Replication um Fehlertoleranter zu sein: Mehrere Server haben die Datei
- Volume Storage Group (VSG): Gruppe an Servern die meine Datei haben
- Accessible Volume Storage Group (AVSG): Die Server in VSG, auf die ein Client Zugriff hat
- Read-One (nur ein Server aus AVSG lesen). Write-All (Auf alle Server in AVSG schreiben)
- Wenn es Server in VSG gibt, die nicht AVSG sind (also nicht erreichbar sind) kommt es zu unterschiedlichen Version einer Datei in den VSGs
- Lösung dafür: Version Vector CVVi(f) für jede Datei f. CVVi[j](f) bedeutet: Server i weiß, dass Server j zumindest die Version k der Datei f hat.

Beispiel: Drei Server: S1, S2, S3, Zwei Clients C1 und C2, eine Datei f

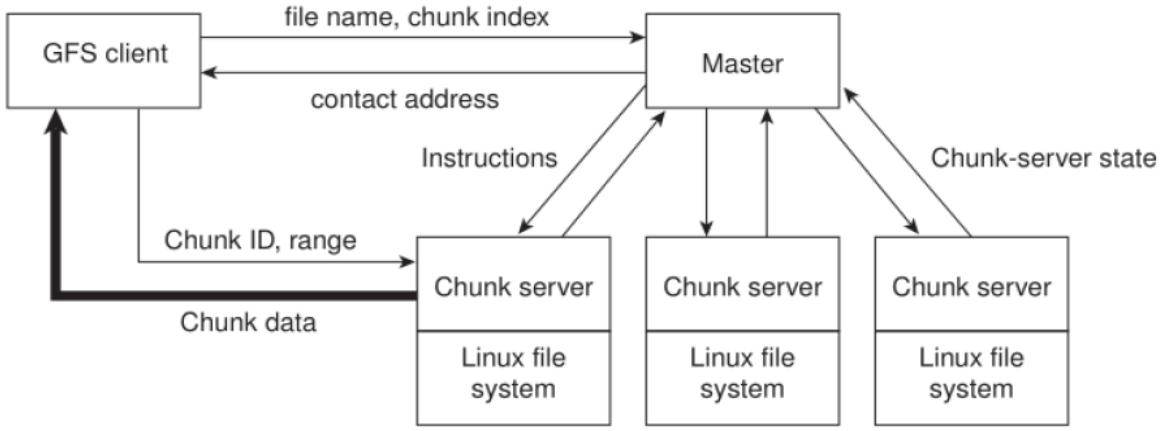
1. Startvektor: CVV1(f) = CVV2(f) = CVV3(f) = [1,1,1]
2. C1 ändert Datei f und speichert auf seine AVSG (S1, S2): Da S3 nicht erreichbar ist, wird der Vektor nur für S1 und S2 aktualisiert: CVV1(f) = CVV2(f) = [2,2,1]. CVV3(f) = [1,1,1]
3. C2 ändert holt sich auch die Datei f von S3 und aktualisiert sie. Da er aber nur S3 erreichen kann, wird auch nur S3 aktualisiert: CVV3(f) = [1,1,2]
4. Wenn sich die Server wieder alle erreichen können, kommen sie drauf, dass es einen Konflikt gibt, da CVV1(f) = CVV2(f) = [2,2,1] != CVV3(f) = [1,1,2]

Ob Coda die UNIX Semantics unterstützt. Warum/Warum nicht
- Coda unterstützt nicht UNIX-Semantics, aber Transaction-Semantics.
- Warum? Weil Coda die Arbeit mit ausgefallenen Servern unterstützt, daher kann es vorkommen, dass ein Client nicht die Version nach dem letzten Schreiben einer Datei verwendet. Das aber wird von Unix-Semantics verlangt.

80. **Cluster-Based File Systems anhand des Google File System**

- Cluster-Based?: Eine einfache Client/Server Architektur funktioniert für sehr große Datenmengen nicht mehr. Daher brauchen wir Techniken um Dateien parallel aufrufen zu können.

Google File System:
- Dateien werden in 64MB chunks unterteilt und auf Chunk-Server verteilt und gespiegelt
- Der Master-Server verwaltet nur den Datei-Namen + dazugehörigen Chunk-Server
- Google File System Client ruft den Master-Server mit dem File-Name und Chunk-Index auf, dieser sucht den Chunk-Server raus
- GFS-Client kann dann direkt den Chunk-Server aufrufen und den Chunk bekommen
- Daten werden mittels Primary-Backup Schema repliziert; der Master ist nicht in dieser Schleife.

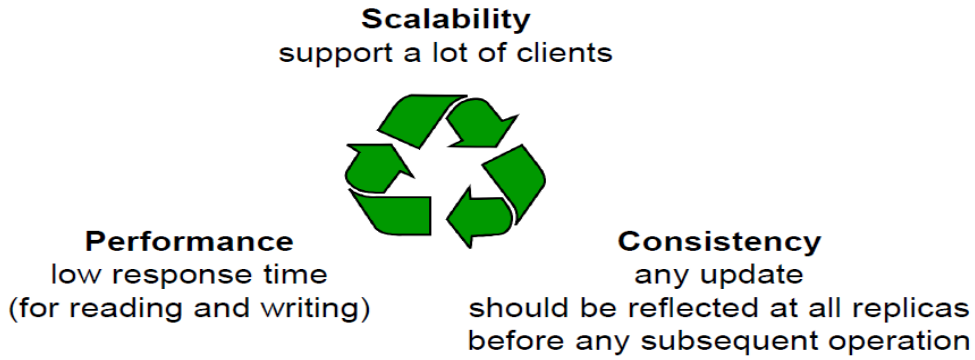


Consistency & Replication

81. **Was ist Consistency & Replication?**

- Replication: Verwaltung von mehreren Kopien der selben Datei an verschiedenen Orten
- Consistency: Die Kopien einer Datei konsistent also am gleichen Stand halten
- Gut: Durch Replication kann die Performance erhört werden (Mehr Client Requests durch verschiedene Server), Ein Server nahe am Client geht auch viel schneller
- Nicht Gut: **stale data**: Die Kopien am gleichen Stand zu halten braucht auch Resourcen, Änderungen werden nicht sofort auf alle Kopien angewandt => unterschiedliche/falsche Version einer Datei

Trade-off Performance vs Scalability
Es gehen nur 2 von 3 Dingen:



→ Kompromiss: Konsistenz auflockern und trotzdem konsistenten Service anbieten können (siehe Consistency models später).

Theoretisches Grundmodell: Distributed Datastore Model: Mehrere lokale Kopien hängen mit einem verteilten Datenspeicher zusammen, Jeder Prozess arbeitet mit einer der Kopien. Die Kopien werden konsistent gehalten.

Consistency models: Vertrag zwischen (distributed) data store und Prozessen, in dem präzise angegeben wird was die Ergebnisse von read und write Operationen bei Präsenz von Concurrency sind. Erlaub Erfüllung fundamentaler Annahme, dass die lokale Kopie bei einem read die aktuellste Version liefert.

Synchronous Replication issue
- Wenn ich ein Update aller Replications als atomare Aktion durchführen will (Um stale data zu verhindern), muss ich zuerst die Zustimmung aller Replications abwarten (vielleicht ist die Datei ja gerade wo anders geöffnet). Das wird sehr schnell sehr aufwendig!
- => Deswegen: Ich erlaube einen gewissen Grad an Inkonsistenz damit ich nicht bei jeder Operation alle anderen Replicates updaten muss.

Konflikte:
- Read-Write-Conflict: eine Read-Operation und eine Write-Operation gleichzeitig
- Write-Write-Conflict: Zwei Write-Operationen gleichzeitig
- Read-Read ist kein Konflikt
- Unser Ziel: Alle Konflikt-Operationen müssen bei allen Replications in der selben Reihenfolge erfolgen

Es gibt zwei grundsätzliche Gruppen von Consistency models: Data centric und Client centric. Auf den Folien werden für die Beispiele folgende Notationen verwendet:

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

82. **Was sind Data-centric Consistency models?**

- Eine Vereinbarung zwischen einem Data-Store und Prozessen, bei der der Data-Store festlegt, was die Ergebnisse von parallelen READ und WRITE-Operationen sind. Es wird erwartet, dass ein Prozess immer die aktuellste Version einer Datei erhält.

Degree of consistency (Der Grad der Konsistenz)
- Replicas können sich in **numerical value** unterscheiden
- Replicas können sich in ihrer **staleness (Verbrauchtheit)** unterscheiden
- Replicas können sich in der Anzahl und Folge der auf ihnen erfolgten Operationen **(performed update operations)** unterscheiden
Vector Clocks / performed update operations:
- Vector Clock Replicate A = (15,5) bedeutet, dass wir in Replicate A bei T=15 sind, in Replicate B bei mindestens 5, da unserer letztes Update zum Zeitpunkt 5 erfolgt ist
- Vector Clock Replicate B = (11, 0) bedeutet, dass wir in Replicate B bei T=11 sind, in Replicate A bei 0, da wir noch gar keine Updates von A bekommen haben
- Wenn ich sage, dass z.B: 20 Zeiteinheiten nicht aktualisiert werden muss, sind wir noch konsistent. Wenn wir aber sagen, dass nach 10 Zeiteinheiten überprüft werden muss, dann müssen wir aktualisieren!

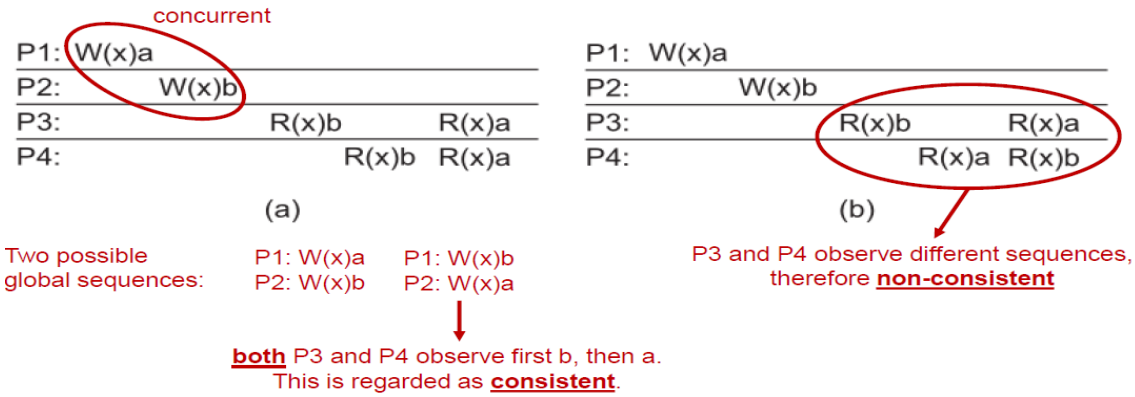
Numerical Deviation / numerical value:
- Wie weit ist der Wert x in A bzw. B von einander unterschiedlich
- Ich vergleiche also den in Replicate A bereits persistierten Wert von X mit dem in B durch seine noch nicht persistierten Änderungen möglichen Wert x und umgekehrt.
- Beispiel: Numerical Deviation = 5, wenn in A der Wert x = 3 gespeichert ist, und in B x = 0 ist und 8 Operationen mit x = x+1 möglich sind.
- Wenn ich bestimmt habe, dass erst ab einer Inkonsistenz von 10 aktualisiert werden muss, muss ich noch nicht aktualisieren, wenn z.B. der Wert 3 ist, muss ich updaten.*Conit: Consistency Unit*
- Legt fest, anhand welcher **data unit** wir die Konsistenz messen
- Beispiele: Webpage, Table entry, entire table in DB

Beispiel: Conit ist der Wert unserer Aktie:
- numeric value: Darf sich nicht mehr als 10 cents unterscheiden
- staleness: Meine lokale Kopie muss zumindest alle 10 Sekunden überprüft werden
- performed update operations: Es dürfen nicht mehr als 3 unbeachtete Operationen erfolgen

1. **Was ist das Sequential consistency model?**

- Jeder Prozess sieht die selbe Reihenfolge für alle Operationen.
- Nicht konsistent, wenn z.B.: Prozess 2 R(x)a, R(x)b und Prozess 3 R(x)b, R(x)a als

Abfolge hat, da diese Operationen nicht in der selben Reihenfolge auftauchen
- Um das zu erreichen, müssen alle Reader die Updates in genau derselben Reihenfolge bekommen
- => Alle Read müssen in der selben Reihenfolge sein
-=> Nicht erfüllt, wenn die Abfolge der READ in zwei Prozessen unterschiedlich sind, unabhängig von den WRITEs
- *„ Die Operationen von nebenläufigen Prozessen werden, auch wenn sie eigentlich simultan geschehen, in einer zeitlich geordneten Reihenfolge abgearbeitet - sowohl global gesehen, als auch für jeden Prozess selbst betrachtet. Das bedeutet, jeder Prozess bekommt die gleichen Updates in der gleichen Reihenfolge mitgeteilt. “*

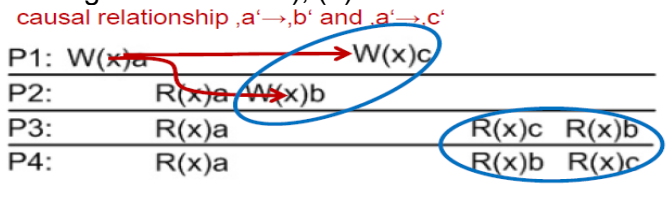


→ W(x)b und W(x)a kann als global logische Sequenz festgesetzt werden.

2. **Was ist Causal consistency?**

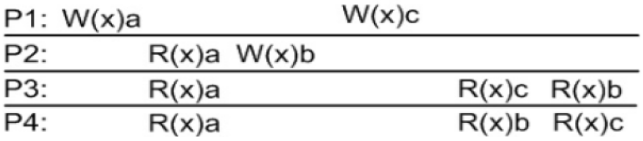
- Alle Read-Operationen, die von Write-Operationen abhängig sind, müssen relativ zu ihren Write-Operationen in der selben Reihenfolge sein
- Schwierig: Zu entscheiden, was kausal ist.
- => Es geht darum, wenn in einem Prozess ein READ vor einem WRITE kommt, müssen alle READ in der Reihenfolge von diesem READ->WRITE sein
- *„ Änderungen, die kausal zusammenhängend sind, müssen von allen Prozessen in derselben Reihenfolge gesehen werden. Andere Änderungen können auch simultan bearbeitet und nicht in der gleichen Reihenfolge gesehen werden. “*

Unterste Beispiele: (a) nicht konsistent (P1 beeinflusst P2, Schreiboperation müsste von allen Prozessen gelesen werden), (b) konsistent

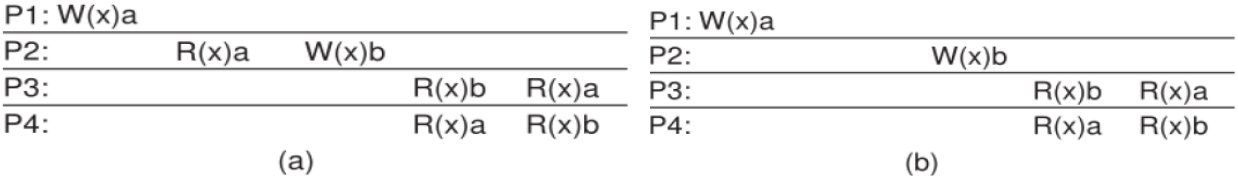


Both P3 and P4 observe ‚a' before ‚b' and ‚a' before ‚c'.
Therefore, causally consistent

Since W1(x)c and W2(x)b are concurrent, P3 and P4 can observe them in any order.

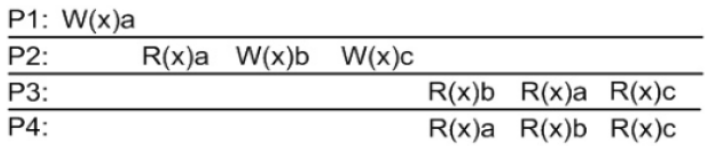


But not sequentially-consistent, since P3 and P4 observe different sequences of b and c.

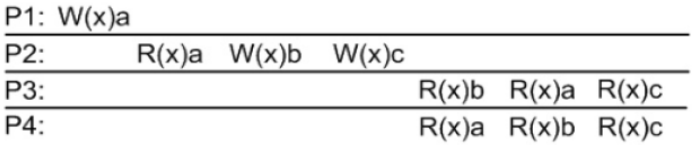


3. **Was ist FIFO-Consistency?**

- Wenn ein Prozess zwei Write in einer gewissen Reihenfolge durchführt, müssen alle Prozesse die Reads zu diesen Werten in der selben Reihenfolge durchführen.
- Writes aus verschiedenen Prozessen sind unabhängig von einander! ! !
- *„ Die Änderungen eines Prozesses sind bei allen anderen Prozessen in derselben Reihenfolge sichtbar; bei mehreren Prozessen und simultanen Änderungen kann dies allerdings nicht mehr garantiert werden. Klassisches First In - First Out Prinzip. “*



- 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.

4. **Welche Consistency – Beispiel**

A: Sequential Consistency? NEIN, da in P3 Reihenfolge a -> c -> b und in P4: c -> a -> b
B: Causal Consistency? JA, da in P2 Reihenfolge a -> b und auch in allen Prozessen a -> b
C: FIFO Consistency? NEIN, da in P1 Write a -> c, aber in P4 Read c -> a

5. **Grouping Operations / synchronisierte Variablen / lock**

- Zugriffe auf synchronisierte Variablen sind sequentiell Konsistent
- Zugriff auf eine synchronisierte Variablen ist nur erlaubt, wenn alle Write darauf fertig sind
- Data-Zugriff erst erlaubt, wenn alle vorherigen Zugriffe darauf fertig sind
- Alle haben immer die selbe Sicht auf eine synchronisierte Variable

Ablauf:
1. Ich setze ein Lock auf eine Variable
2. Ich bearbeite die Variable / oder ich lese die Variable
3. Ich entferne das Lock und liefere nur das Ergebnis der Operationen auf die Variable an alle aus, nicht die ganzen Operationen

ACHTUNG: Es wird auch ein Lock zum Lesen einer Variable gebraucht, da sonst das darunter liegende System nicht den Wert der Variable verfügbar macht. Erst durch das Lock bekomme ich überhaupt einen (aktuellen) Wert.

=> Data-centric consistency sollte vermieden werden, da es sehr viel Aufwand ist und recht komplex und teuer wird. (angeblich)

83. **Client-centric consistency models**

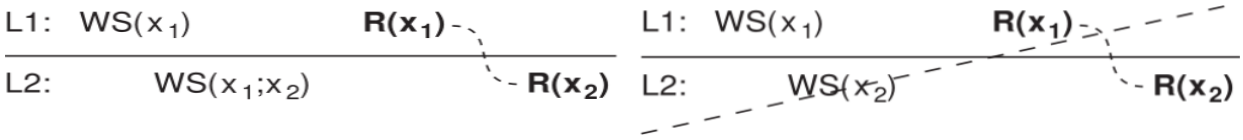
- Wir versuchen systemweite Konsistenz zu lockern, indem wir berücksichtigen, was die individuellen Anforderungen einzelner Clients sind

Beispiel: Konsistenz für mobilen User, der mit seinem Laptop auf eine Datenbank zugreift
- User beginnt seine Arbeit und macht ein Update der Datenbank am Arbeitsplatz A
- User geht zu Arbeitsplatz B und macht mit seiner Arbeit weiter
- In diesem Fall heißt „Konsistenz“ für diesen User lediglich, dass der Server am Arbeitsplatz B auf dem Stand ist, den er am Arbeitsplatz A zurückgelassen hat auch wenn er vom Arbeitsplatz B eine andere Replica der Datenbank aufruft.
- Für diesen User in diesem Beispiel spielt parallele Konsistenz keine Rolle
- Wir verwenden Eventual Consistency *(Client-centric Consistency models)*

Eventual Consistency
- Updates werden nicht sofort an alle Replicas geschickt, sondern die Replicas werden erst nach und nach aktualisiert. Irgendwann sind dann alle aktuell.

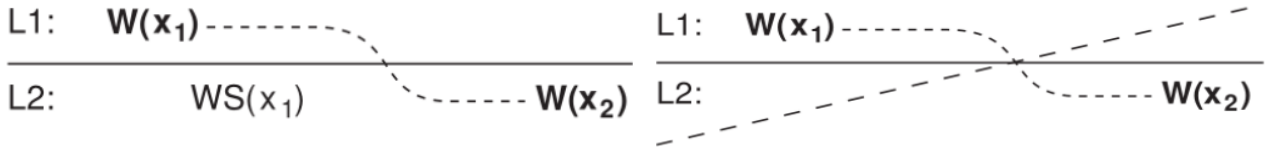
1. Write Sets WS
    - Ein Write-Set ist eine Gruppe von Operationen, die zu dem Ergebnis einer Variable führen.
    - WS(xi[t]) ist die Gruppe an Operationen in Location i, die zur Version xi der Variable x geführt habem
    - WS(xi[t1];xj[t2]) bedeutet, dass WS(xi[t1]) zu WS(xj[t2]) geführt hat (WS(xi[t1]) ist Subset von WS(xj[t2])

2. Monotonic Read Consistency
    - Nach einem Read auf eine Variable in einer Location, liefern alle weiteren Reads auf diese Variable entweder die selbe Version der Variable oder eine aktuelle Version, die auf die alte Version aufbaut.
    - Beispiel: Wenn Ich in L1 R(x1) und in L2 R(x2) habe (x2 ist die aktuellere Version von x1), dann muss ein WS(x1;x2) stattgefunden haben, also x2 ist die aktuellere Version von x1



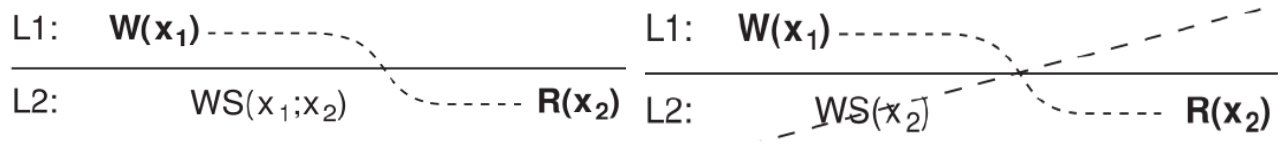
    - Beispiel: Wenn ich Emails von unterschiedlichen Email-Servern abrufe, bekomme ich sowohl die neuen Emails als auch die, die ich bereits bei einem anderen Server früher abgerufen habe.

3. Monotonic Write Consistency
    - Eine Write-Operation auf eine Variable x erfolgt erst nachdem alle anderen Write-Operationen auf diese Variable abgeschlossen sind.
    - Wenn ich also auf X schreibe: W(x2), dann muss mir der vorherige Schreibvorgang W(x1) bekannt sein, was durch WS(x1) angegeben wird.



4. Read-your-Writes

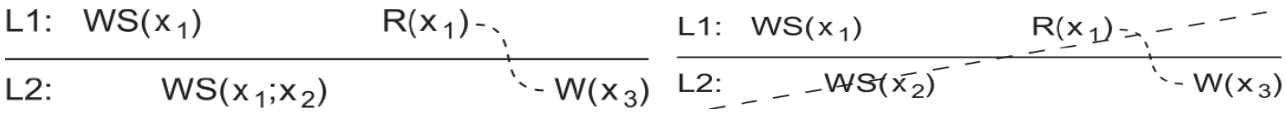
- Jede Read-Operation auf X R(x2) zeigt den Effekt einer vorhergegangen Write-Operation W(x1), was durch WS(x1;x2) angegeben wird
- Beispiel: Wenn ich eine Website aktualisiere und sie danach im Browser aufrufe wird die aktualisierte Version und nicht etwa eine gecachte angezeigt.



5. Write-follows-reads

- Jede Write-Operation, die auf eine Read-Operation folgt, passiert auf Basis der Version von X das gelesen wurde oder einer aktuelleren. Es muss also ein WS(x1;x2) zwischen R(x1) und W(x3) erfolgen

Beispiel: Kommentare auf einen Artikel werden nur angezeigt wenn man überhaupt mal den Artikel hat (das read „pullt“ die dazugehörige Write Operation.



84. **Replica Consistency Concerns & Management**

- Replicas müssen konsistent nach manchem Modell sein
- Wenn es kein Update gibt, gibt es auch kein Problem
- Wenn die *access-to-update ratio* hoch ist, bringen Replicas etwas (Wenn also öfters auf ein Item zugegriffen wird als es upgedatet wird),
- Wenn die *update-to-access ratio* hoch ist, werden die Updates überhaupt nicht verwendet
- Im Idealfall sollten nur die Replicas aktualisiert werden, auf die auch zugegriffen wird
- Replicas sollten sich in der Nähe der Clients befinden (CDN)

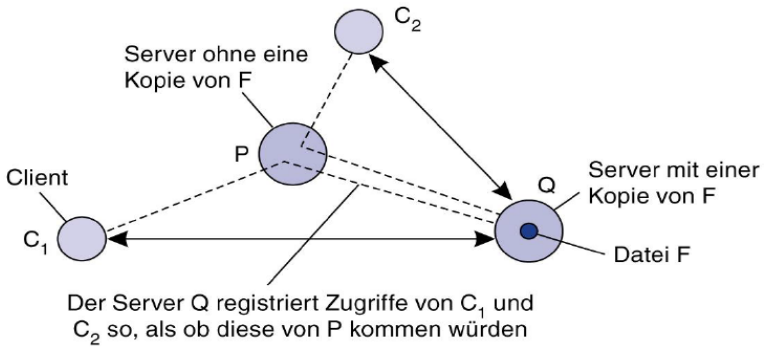
Replica placement

- Herausfinden, welche Plätze K die besten aus N verfügbaren sind
- K sind Plätze, für die die average distance to clients minimal ist -> computationally expensive
- K sind die am besten angebundenen Hosts in den größten autonomen Systeme -> c. expensive
- K sind die Plätze in einem geometrischen Raum, die an dichten Stellen stehen -> c. cheap
- => In der Praxis: Einfach zusätzliche Server mieten, wo immer ich sie brauche

85. **Content Replication**

- Permanent: Oft initiale Verteilung der Daten. Ein Server hat immer die aktuelle Replica (origin Server). Beispiele: replizierte website Server für load balancing, mirroring (client ist sich der replikas bewusst und wählt besten aus). Normalerweise nicht viele Kopien
- Server-initiated: Server entscheidet dynamisch wann und wo Replicas erstellt werden sollen. Basis für Web hosting und Cloud Services.

Beispiel:



System oft von Web Hosting Companies genutzt. Jeder Server weiß welcher Server der nächste zum Client ist (wäre). Jeder Server hat ein Protokoll zu den access counts per file, das gleichzeitig immer bedenkt, wo der nächstgelegenste Server zum Client ist, der die Zugriffe macht $cnt_Q(P,F)$.
Jeder Server hat einen replication threshold R und 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

- Client-initiated: Replica wird auf Anfrage des Clients erstellt (Client cache). Datei wird nur temporär behalten, befindet sich normalerweise auf Client Maschine (Oder in der Nähe bei gemeinsam genutztem Cache). Wenn Daten in Cache gefunden werden spricht man von „Cache hit“. Arten von Client Caches:
Hardware cache: verwendet in modernen CPUs und shared-memory multiprocessor systems
Software based solutions: Für Middleware-basierte Verteilte Systeme

86. **Content distribution**

Varianten:
- Nur notifications/invalidation von updates bekannt geben (Beispiel: Caches updaten). Da keine konkreten Daten gesendet werden, niedrige Bandbreite benötigt. Gut wenn viele Updates aber wenig Reads
- Passive Replication: Daten von einer Kopie zu einer anderen transferieren. Sinnvoll wenn read-to-write Ratio hoch ist. Mehrere Änderungen können gebündelt werden und einmal gesendet.
- Active Replication: Gebe deine Update-Operationen den anderen Kopien bekannt (z.B. komplexe Berechnungen: Rechnungsformel selbst benötigt zur Übertragung wenig Bandbreite, aber Ergebnis ist dasselbe wie bei direkter Datenübertragung)

-=> Es gibt keine beste Lösung, hängt von Bandbreite und read-to-write ratio ab

Push (server)-based protocols
- Ich aktualisiere andere Replicas ohne, dass sie mich darum gebeten haben
- Wird oft bei permanenten und server-initiated Servern verwendet
- Hoher grad an Konsistenz
- Multicasting
- (blocking/eager/synchronous): Replicas werden sofort aktualisiert und erst dann wird weitergemacht (schwierig, braucht viel Zeit, jeder Client kann failen (kann mit effizienter Multicast Implementierung verbunden werden)). Bsp: Flugticketkauf sollte sofort registriert werden.
- (non-blocking/lazy/asynchronous): Replicas werden erst nach und nach aktualisiert (z.B. diejenigen, die die Updates eher brauchen), es wird vorher schon weitergemacht

Pull (client)-based protocols
- Client fragt Server ob es Updates gibt, können dann abgerufen werden. Antwortzeit beim Client ist länger bei Cache miss
- Wird oft bei client caches verwendet
- Unicasting

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

Leases
- Leases ist ein Versprechen vom Server mir updates zu schicken, bis mein Lease abgelaufen ist. Also eine Kombination aus Push und Pull: Bis der Lease abgelaufen ist, bekomme ich Updates gepushed. Dann muss ich pullen.

- Age-based: Ein Object, das sich langer nicht geändert hat, wird das auch demnächst nicht tun, also lange leasen.
- Renewal-frequency-based: Je öfters ein Client nach einem Objekt fragt, desto länger wird seine expiration-time für dieses Objekt.
- State-based: Je mehr ein Server belastet ist, desto kürzer wird die Expiration-Time

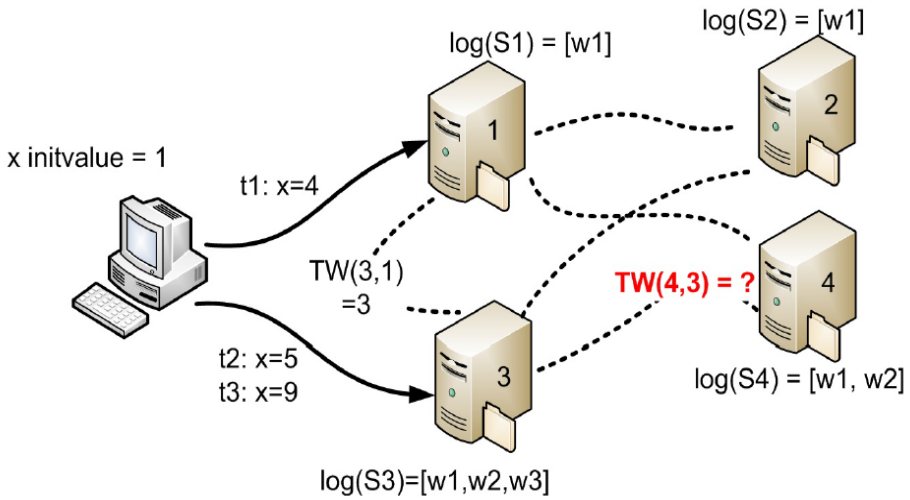
=> Ziel: Den Server entlasten (kleinerer server state, muss sich nur mit Clients beschäftigen die lease brauchen), aber trotzdem Konsistent bleiben! Außerdem melden sich nur Clients die hohe Konsistenz brauchen für nen Lease an, der Server kann diese also besser bearbeiten. Wenger unnötiger Traffic.

87. **MC-Frage zu wann Push einem Pull-based Protokoll vorzuziehen ist**
Bei permanenten bzw. serverseitig initiierten Replikas ist die Push-Variante zu bevorzugen. Pull wird hingegen gerne von Clients mit Cache verwendet.

88. **Continuous consistency**

- Ich hab mehrere Server Si, die unsere Replicas enthalten mit jeweils einem Logfile log(Si)
- Wenn ein replica aktualisiert wird, merke ich mir die „Schwere“ des Updates, die ich durch die Funktion weight(W) (W=Operation) bekomme. weight(W) > 0.
- Mit der Variable TW(S1, S2) wird angegeben, wie groß das **T**otale Ge**W**icht von Änderungen von S2 ist, die in den Logs von S1 gefunden wurde.
- Mit TW(S1,S1) bezeichne ich die lokalen Änderungen
- TW[Si,Sj] = SUM { weight(W) = Sj & W ELEMENT VON log(Si) } *// Summe aller Gewichte aller Operationen von Sj, die in den Logs von Si gefunden wurden*

Beispiel:

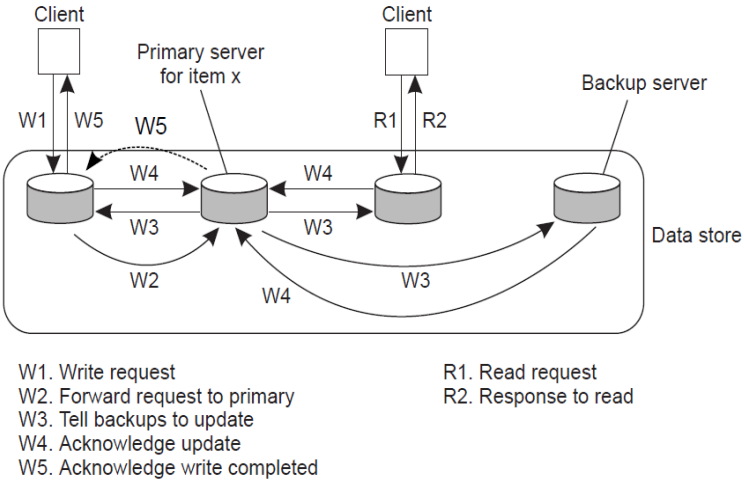


Welchen Wert hat TW(4,3)?.
- TW(3,1) bedeutet: Nach t1 teilt S1 S3 seine Änderungen mit. In seinem log steht w1, zu diesem Zeitpunkt ist im Log von S3 noch nichts, also ist der Unterschied zwischen [] und [w1] = 3, da der Unterschied zwischen dem Init-Wert (1) und dem Update durch w1 (4) = 3.
- TW(4,3) bedeutet: Nach t2 teilt S3 dem S4 seine Updates mit. Die Operationen in den Logs sind zu dem Zeitpunkt [w1, w2] und [w1], die Gewichte 5 (w1+w2) und 4 (w1), also ist der **Unterschied 1**.

- Wieso wird t3 nicht in die Berechnung mit einbezogen? Weil t3 noch nicht weitergeleitet wurde. Es wurde also nur t2 von S3 auf S4 und t1 von S1 auf S4 weitergeleitet.

Verhältnis was ich weiß und glaube
- TWk(i, j) => Der Wert, den ich glaube dass TW(i,j) hat (gossiped weitergegeben)
- 0 <= TWk(i,j) <= TW(i,j) <= TW(j,j)
- // Der Wert, den ich für Si glaube ist kleinergleich dem Wert, den ich für Si weiß ist kleinergleich meinem lokalen Wert für das Total Gewicht.

89. **Primary-based protocols**



Synchronous

- Jede Variable hat einen primary Server, der die Schreibzugriffe koordiniert
- Keine Inkonsistenz (Da alle Kopien)
- Änderungen sind Atomar
- Langsam
- Nicht geschützt gegen Node-Fehler
- Ein lokales Read liefert immer die aktuellste Version

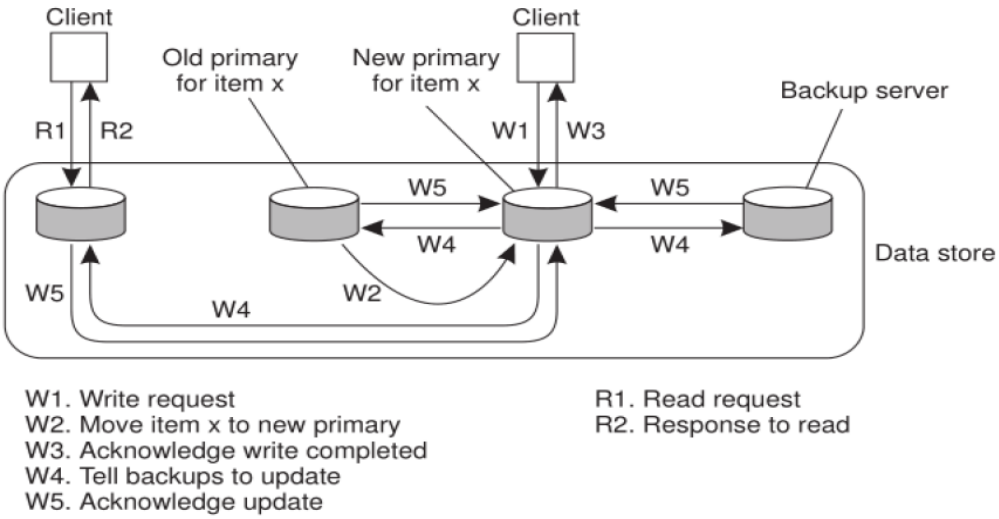
Verwendet sequential consistency model (synchronous Methode):
- Alle write Operationen werden durch den Primary Server geordnet und an alle anderen geschickt.
- Das Lesen einer lokalen Kopie liefert die aktuelle Version. Änderungen sind atomar, keine Inkonsistenz.
- Read schnell, Write langsam
- Ist ein Node nicht verfügbar, ist ein write nicht möglich → fehleranfällig bei Netzwerk oder Node Failure.

Asynchronous (non-blocking)
- Schneller, da nicht gewartet wird (ACK sobald primary server update erhält), write schneller
- Daten können aber Inkonsistent sein
- Ein lokales Read liefert nicht immer die aktuellste Version
- geschützt gegen Link oder Node Failure.

=> Verwendet in verteilten Datenbanken / Dateisystemem im LAN mit hoher Fehlertoleranz.

Beispiel: Asynchrones Primary Backup Protokoll: Bei einer Abbildung einer Schreiboperation auf einen Replika einzeichnen wie die Operation propagiert wird (an Primary, und Replikas)

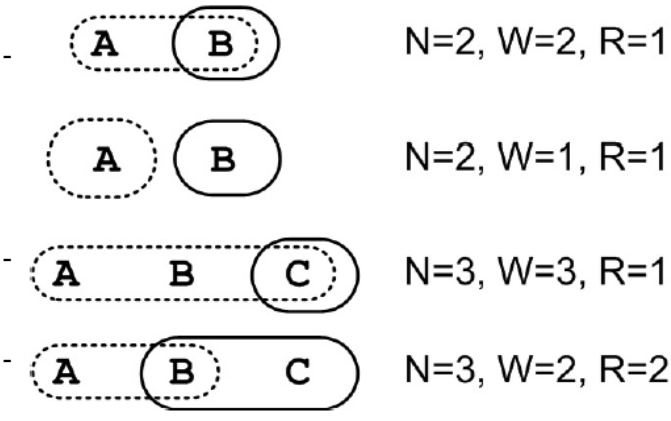
Primary-backup protocol with local writes:



Erlaubt wenigstens FIFO-Consistency. Wenn es nicht zuviele gleichzeitige Writes gibt, geht das schnell. Primary ist ein point of failure. Wenn er abstürzt, kann kein neuer Primary bestimmt werden. Bei Primary Transfer müssen Clients für Update warten, bis Transfer komplett ist.

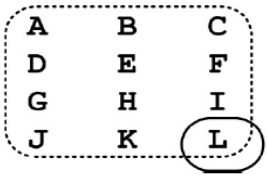
Beispiel: Mobile computing in disconnected mode (Primary sendet vor dem offline gehen alle relevanten Dateien an Clients. Geht Primary nachher wieder online, können Updates auf diese Daten wieder an alle gesendet werden und es kann weitergearbeitet werden.) z.B. Google Daten: Wenn ich offline auf meinen Daten arbeite und online gehe, müssen diese erst wieder an den online-Server geschickt werden, bevor ich auf ihnen weiterarbeiten kann.

90. **Replicated-write protocols (quorum-based protocols)**
- Basiert auf Read- und Write- Quorum, gibt an wie viele Replicas ich mind. Brauche
Grundidee: Wir haben z.B. 5 Nodes. Um auf das System ein Write auszuführen, muss ein read/write synchron auf mind. 3 Nodes (mehr als die Hälfte) ausgeführt werden. Daten haben Timestamp um zu sehen welche Version aktuell ist. Vorteil: Es müssen nicht alle Nodes kontaktiert werden. Gut z.B. wenn bestimmte Nodes für Client ohnehin nicht erreichbar sind.

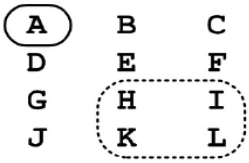


- Wenn Ich 2 Replicas (N=2) und RQ=1 und WQ = 2 habe, entspricht das Primary Backup Synchronous.
Warum? Beim Updaten muss ich alle Replicas (2/2) updaten, beim Lesen muss ich aber nur von einem Replica lesen.
- N2W1R1 entspricht Primary Backup Asynchronous, da ich beim Schreiben nur ein Replica updaten muss.
- N3W3R1 ist wieder Primary Backup Synchronous
- N3W2R2 „Tolerance only focus“.

=> Immer wenn sich R und W überschneiden (Wenn Also R+W > N), bekomme ich garantiert den aktuellsten Wert beim Lesen.



N=12, W=12, R=1

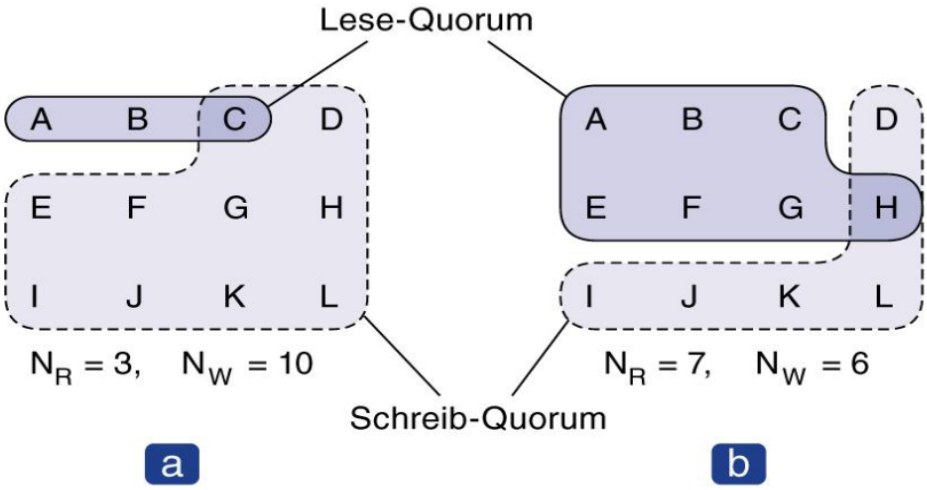


N=12, W=1, R=4

Focus on consistency + low write success probability

Focus on tolerance + eventual consistency

- Je Nach Anwendung kann ich meine Quorums so wählen, dass ich möglichst konsistent bin oder möglichst schnell.
- Operationen sollten so gewählt sein, dass ein Mehrheits-vote zustande kommt
- Im rechten Beispiel lege ich meinen Fokus auf Toleranz, bin aber vielleicht nicht immer konsistent



- Um Lesen zu optimieren setze ich R=1 und W=N,
- Um Schreiben zu optimieren setze ich R=N und W=1. (Ich muss alle lesen um den aktuellen Wert zu bekommen)
- Um Schreibkonflikte zu vermeiden sollte W >= (N+1)/2 sein. Also mehr als die Hälfte
- Für starke Konsistenz W + R > N

Beispiel: Ein Write-Quorum war gegeben - genau die Hälfte der Knoten. Warum ist das ein gültiges/ungültiges Quorum? Bei gültig: man musste ein gültiges Read-Quorum angeben, Bei ungültig: warum kann das zu Probleme führen?

Beispiel Given Quorum-based protocol example with 9 nodes (A-B-C-D-E-F-G-H-I) with read quorum = 4 nodes. Is the Write quorum (A-B-C-D-E) valid? With respect to answer: give the valid read quorum or explain why it would lead to problem.

Beispiel: Is the marked node set shown below a valid write quorum? (1 point)
i. [YES, because a write quorum of 6 and a read quorum of 9 > # total replica servers]
ii. [YES, because a read quorum of (A, B, C, D, E, F, G) overlaps with the write quorum]
iii. [NO, because a write quorum of (A,D,G, J, K, L) would not include C, F, or I]
iv. [NO, but it would be valid when we remove C from the set of replica servers]

Beispiel: Explain very briefly (1 point)
i. If YES [i,ii]: draw an example of a valid read quorum size and read quorum set.
ii. If NO [iii,iv]: draw a quorum (read or write set) that might lead to a problem.

Partitioning
- Ein Teil der Replicas wird unerreichbar, Partition mit W nodes nimmt weiter Updates an, selbiges gilt mit R set
- Problem ist wenn Lese/Schreib Zugriff in der unerreichbaren Partition angefragt wird, würde das verweigert werden
- Wenn das inakzeptabel ist, können Nodes zur anderen Partition hinzugefügt werden und ein application-assisted merge später durchgeführt werden.

X Security

91. **Was ist Security?**
Die Kombination aus:
- Confidentiality: Keine Einsicht in Informationen ohne Authorisierung
- Availability: Verfügbarkeit und Benutzbarkeit für authorisierte Entities
- Integrity: Keine unabsichtlichen oder bösartigen Änderungen der Informationen (Auch von authorisierten Entities)

92. **Security Threats**

Drei Hauptbestandteile:
Subject: Einheit die eine Anfrage für einen von Objekten angebotenen Service stellen kann
Channel: Träger von Anfragen und Antworten für Services die Subjects angeboten werden
Object: Einheit die Subjects Services anbietet

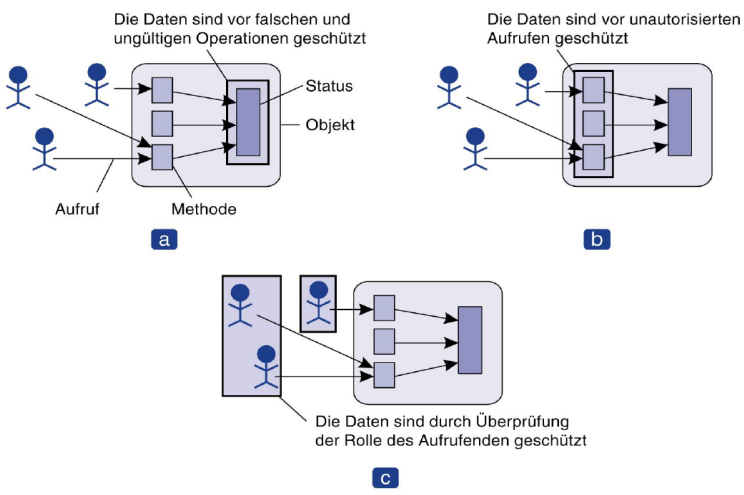
Threat	Channel	Object/Server
Interception	Reading the content of transferred messages	Reading the data contained in an object/server
Interruption	Preventing message transfer	Denial of service
Modification	Changing message content	Changing an object/server's encapsulated data
Fabrication	Inserting messages	Spoofing an object/server

93. **Security Mechanism**

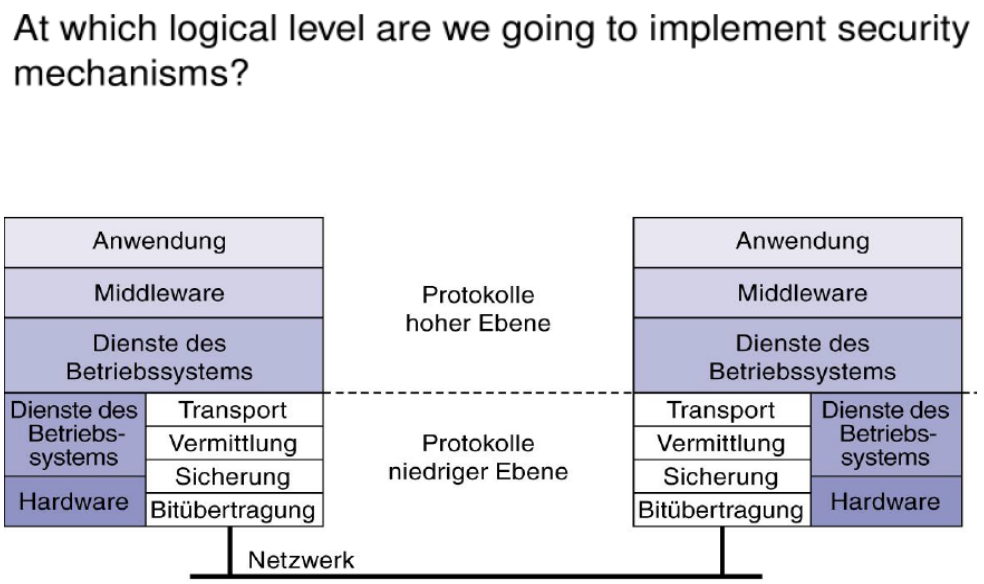
Allgemein auf niedrigen Übertragungsebenen angesiedelt

- Encryption: Daten verschlüsseln, damit Hacker damit nichts anfangen können (confidentiality) ODER Check ob Daten geändert wurden (Hash) (integrity)
- Authentication: Check der Identität eines Subjects (Alice behauptet sie ist Alice)
- Authorization: Check, ob ein Subject eine Aktion durchführen darf oder nicht
- Auditing: Verfolge, welches Subject worauf zugegriffen hat - Kann helfen (und nur sinnvoll um) einen Hacker zu schnappen

Focus of Control: Wer kontrolliert die Daten:
Objektstatus (Schutz vor ungültigen Operationen),
Objektmethoden (Schutz vor unauthorisierten Aufrufen),
Aufrufender (Überprüfung durch Rolle des Aufrufenden)

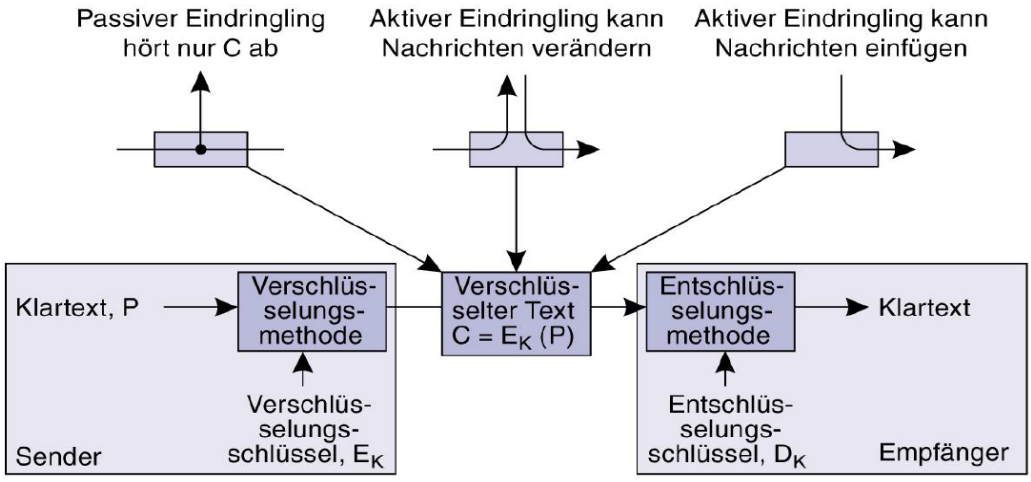


Auf welchen Ebenen können Sicherheitsmechanismen implementiert werden?



94. **Kryptography**

Grafische Darstellung für besseres Verständnis:



- Symmetric system (Use Case: Encryption): Nur ein Schlüssel zum Ver- und Entschlüsseln (Preventing: Interception)
- Asymmetric system (Authentication): Public + Private Key verwenden (Preventing: Fabrication)
- Hashing system (Integrity): Nur Verschlüsselung (z.B. MD5, SHA-1) (Preventing: Modification). Es gibt keine Entschlüsselung, nur Vergleich (der Hash-Werte) ist möglich

Kryptografische Funktion: $E_K(m_{in}) = m_{out}$
Verschlüsselungsmethode E öffentlich, aber Verschlüsselung im Ganzen mittels eines Key S parametrisieren.

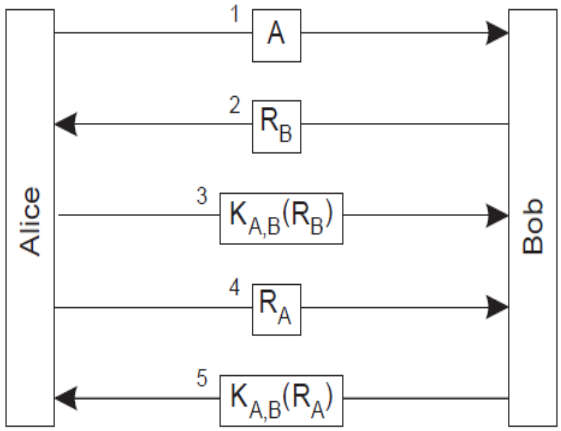
- One-Way-Function: Es gibt keine (effiziente) Umkehrfunktion. Für Keys heißt das: Es ist schwierig einen Key K zu finden sodass $m_{out} = E_k(m_{in})$
- Weak collision resistance: (Geburtstagsparadoxon) Haben wir ein Paar $<m,E_K(m)>$ ist es schwierig ein m* != m zu finden dass gilt $E_K(m) = E_K(m^*)$. Für keys heißt das: es ist schwierig ein K != K* zu finden, dass gilt: $E_{K^*}(m) = E_K(m)$

- Strong collision resistance: Wenn es allgemein schwierig ist zwei m* und m zu finden, so dass E(m) = E(m*), dann ist E strong collision resistance. Für Keys heißt das: es ist schwierig allgemein ein K und K* zu finden, dass gilt: $E_K(m^*) = E_K(m)$

95. **Was ist ein Secure channel?**
- Beide Seiten wissen, wer sich auf der anderen Seite befindet (Authentication)
- Beide Seiten wissen, dass Nachrichten nicht verändert werden können (Integrity)
- Beide Seiten wissen, dass keine Nachrichten geleakt werden können (confidentiality)

Authentication und Integrity hängen fest miteinander zusammen:
- Authentication without Integrity: Eine Nachricht wird abgefangen, der Inhalt wird verändert (Integrity zerstört), Authentication bleibt aber erhalten, wird dadurch jedoch nutzlos.
- Integrity without Authentication: Eine Message wird von A gesendet, von C abgefangen abgefangen und an B weitergesendet, welcher glaubt dass A der Sender ist (Authentication zerstört)

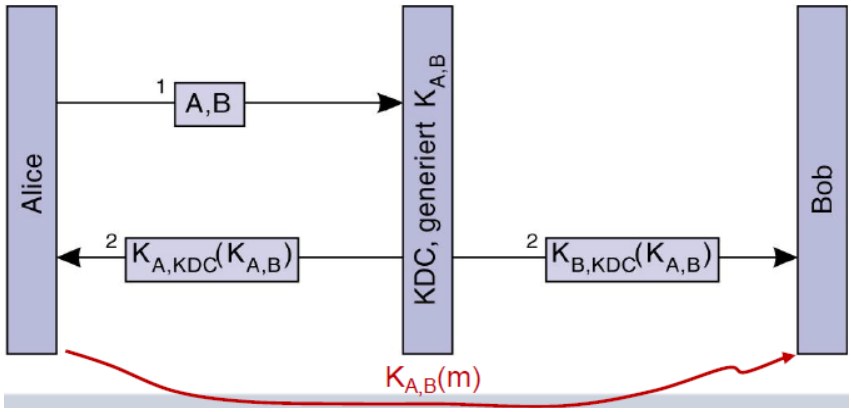
96. **Wie funktioniert shared key Authentication:**



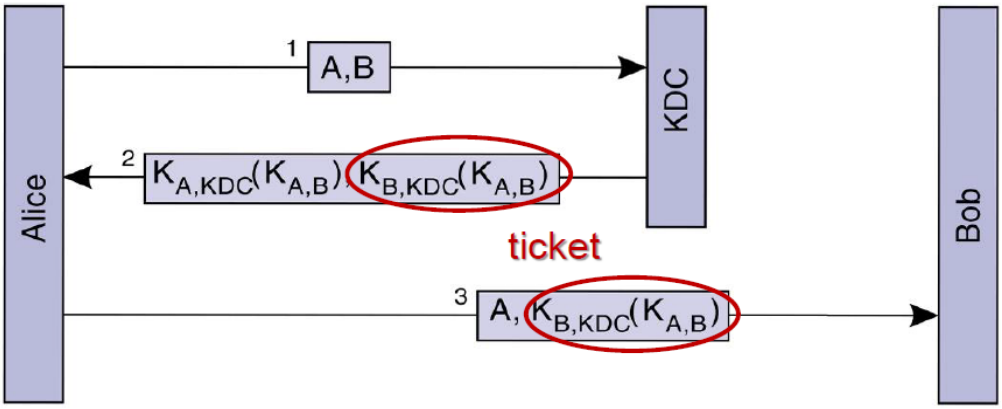
1. Alice sendet ihre ID A an Bob
2. Bob antwortet mit einer Challenge Rb
3. Alice verschlüsselt Rb mit dem gemeinsame Key Ka,b. Bob weiß jetzt dass er mit Alice redet.
4. Alice sendet eine Challenge Ra an Bob
5. Bob verschlüsselt Challenge Ra mit Ka,b. Alice weiß jetzt dass sie mit Bob redet.

=> A+B können sich jetzt ob der Identität der anderen Person sicher sein
-> Ich kann Step 1 & 4 und 2 & 5 kombinieren, verliere aber correctness.

-> Bei N Usern werden N * (N-1)/2 keys benötigt, jeder kennt N-1 keys => *Key Distribution Center* verwenden. Erzeugt Schlüssel je nach Bedarf



Optimiert: Alice bekommt von KDC beide Keys und Alice sendet dann Bobs Key selbst zusammen mit ihrer Identity.

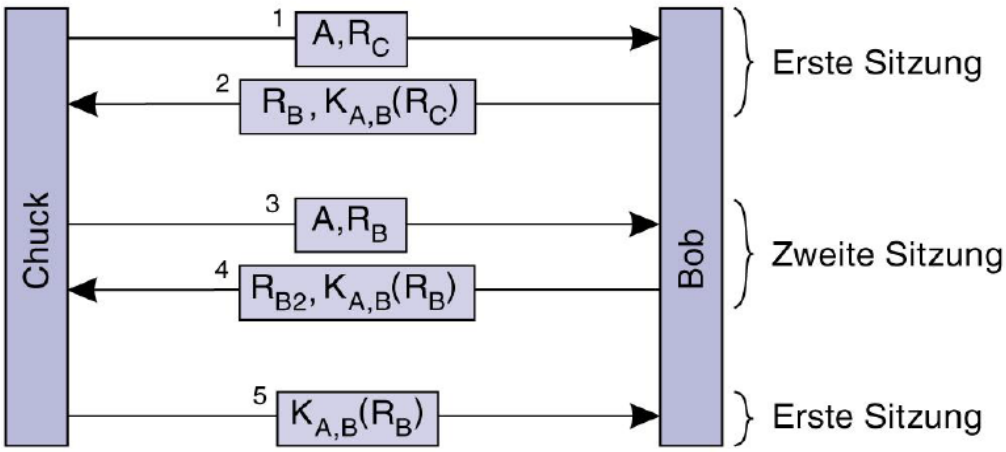


Frage: Wieso antwortet das KDC nach Alice' Anfrage Bob? Alice bekommt ein Ticket?
- Wenn Alice die Authentifizierung mit Bob startet, muss er schon den Key kennen. Also muss er vor der Anfrage von Alice den Key bekommen

Frage: Wieso muss das Ticket, das zurück an Alice geschickt wird, encryptet sein?
- Damit Alice sicher stellen kann, dass ihre Anfrage nicht verändert wurde

Beispiel: Skizzieren, wie die Authentifizierung zwischen zwei Clients mit pre-shared Secret Key funktioniert.

97. **Wie funktioniert key reflection attack:**

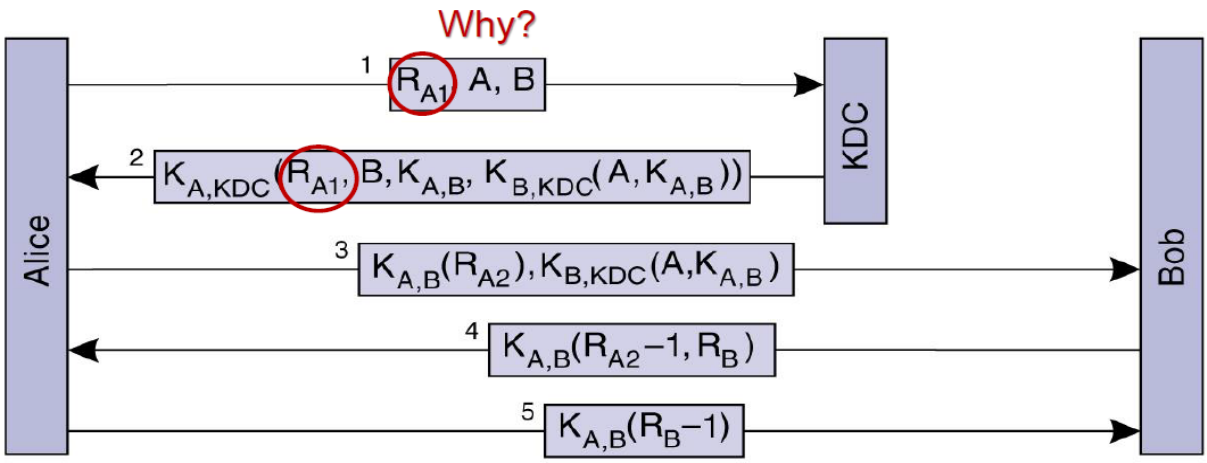


1. Chuck behauptet er ist Alice UND sendet eine Challenge Rc an Bob
2. Bob antwortet mit seiner Challenge Rb und der verschlüsselten Rc
3. Chuck startet eine zweite Session und behauptet wieder er ist Alice und sendet Bobs Challenge Rb
4. Bob antwortet wieder mit seiner Challenge Rb und der verschlüsselten Challenge Rb die Chuck gesendet hat.
5. Chuck sendet jetzt über die erste Session die von Bob mit dem gemeinsamen Key verschlüsselte Challene Rb aus der zweiten Session - die erste Session ist authorisiert (Die zweite Session wird einfach abgebrochen)

=> *Was hilft dagegen? z.B Needham-Schroeder Authentication:*

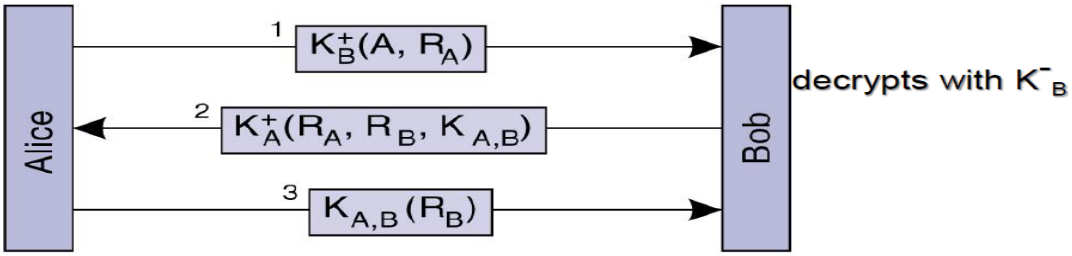
Needham-Schroeder Authentication

Verwende *nonce* (single use, random und unique large numbers) als Challenge um die Ordnung der Messages aufzubauen.



Wozu R_{A1}? Ohne kann Antwort von KDC von Chuck gestohlen und mehrfach verwendet werden um bei Alice zu behaupten, er ist Bob.
Wozu B in erster und zweiter Nachricht? Wenn B in Nachricht zurück an Alice ist (statt z.B. C), weiß Alice, dass die Nachricht nicht geändert wurde.
Warum antwortet Bob mit R_{A2}-1 ? Dadurch kann Bob erweisen, dass die verschlüsselte Nachricht korrekt dekodiert wurde. Hilft auch dabei, die Antwort mit der korrekten Anfrage zu verbinden.

98. **Wie funktioniert public-private key authentication?**
1. Alice schickt ihre Challenge Ra an Bob, die mit Bob's public key Kb+ verschlüsselt ist
2. Bob entschlüsselt die Nachricht mit seinem private key und generiert Ka,b
2a. Bob schickt Ka (um zu beweisen, dass er Bob ist) und seine Challenge Rb mit Alice's public key Ka+ an Alice zurück.
3. Alice entschlüsselt die Challenge mit ihrem private key und kann nun auch Ka,b generieren
3a. Alice sendet die encrypted Challenge an Bob zurück um zu beweisen, dass sie Alice ist.



=> Lebt davon, dass vom Public-Key nicht auf den Private-Key geschlossen werden kann

99. **Beispiel: A = ID von Alice, B = ID von Bob. K = gemeinsamer geheimer Schlüssel.**
Angabe: Ablauf:
1. A wird von Alice an Bob gesendet
2. B wird von Bob an Alice gesendet
3. K(A) wird von Bob an Alice gesendet
4. K(B) wird von Alice an Bob gesendet

Frage: Warum ist das nicht sicher? Welche Angriffe sind möglich?
- Mögliche Antwort: Es werden keine Challenges, sondern nur IDs verwendet, die sich nicht ändern. Es kann also eine alte ID wiederverwendet werden. Angreifer muss nur einmal K(A) und/oder K(B) abfangen und kann ab jetzt immer mit Alice bzw. Bob kommunizieren ohne K zu kennen

Frage: Was muss korrigiert werden, um die Authentifizierung sicher zu machen?
- Anstatt K(A) und K(B) sollte K(Ra) und K(Rb) Ra, Rb = Neu generierte Challenge ausgetauscht werden.

100.**Was für Probleme können mit Schlüsseln auftauchen?**
- Keys wear out: Je mehr Nachrichten mit dem selben Schlüssel verschlüsselt werden, desto leichter kann der Schlüssel herausgefunden werden => Der selbe Schlüssel sollte nicht zu oft verwendet werden

- Danger of replay: Wenn der selbe Schlüssel für verschiedene Sessions verwendet wird, können alte/andere Nachrichten in die aktuelle Session eingeschleust werden

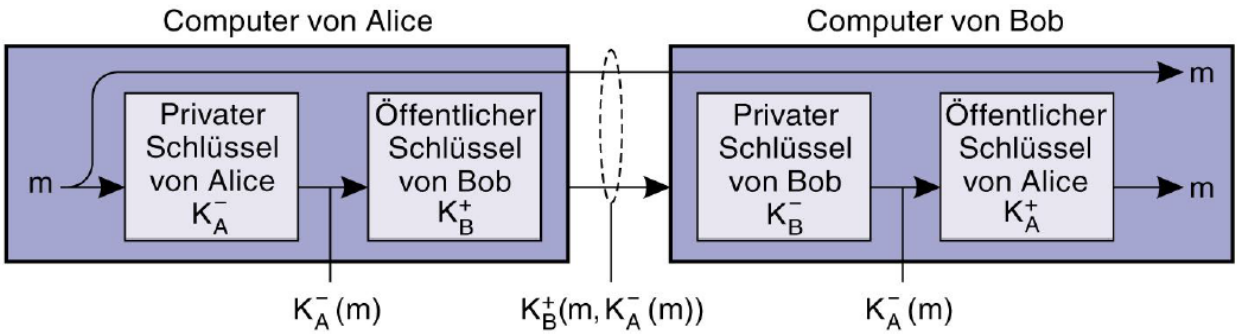
- Compromised keys: Wenn ein Schlüssel kompromittiert ist, kann er nie wieder verwendet werden. Blöd ist das dann, wenn die komplette Kommunikation zwischen A und B den Key verwendet. => Den selben Schlüssel nie für unterschiedliche Dinge verwenden!

- Temporary keys: Du möchtest einen Schlüssel oft nur temporär vergeben => Schlüssel sollten wegwerfbar sein, also: ich kann einen Schlüssel ungültig machen

101.**Digitale Signaturen**

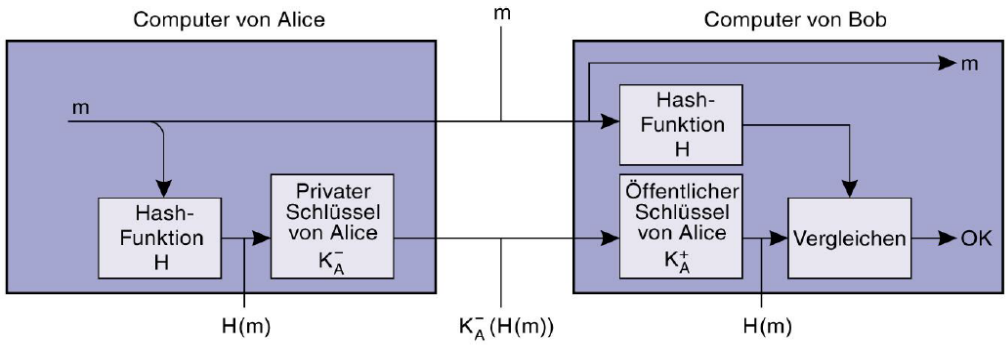
Voraussetzungen:
- Authentication (Beide wissen mit wem sie kommunizieren)
- Nonrepudiation (Nachricht kann im Nachhinein nicht behaupten, die Nachricht nicht gesendet zu haben)
- Integrity (Nachricht kann nicht geändert werden)

Lösung: Sender soll alle gesendeten Nachrichten signieren, damit 1: die Signatur verifiziert werden kann und 2: die Nachricht und Signatur einzigartig assoziiert sind.



Schlecht, weil: Nachricht sollte zuerst gehashed und dann signiert werden, wird aber hier zuerst signiert.

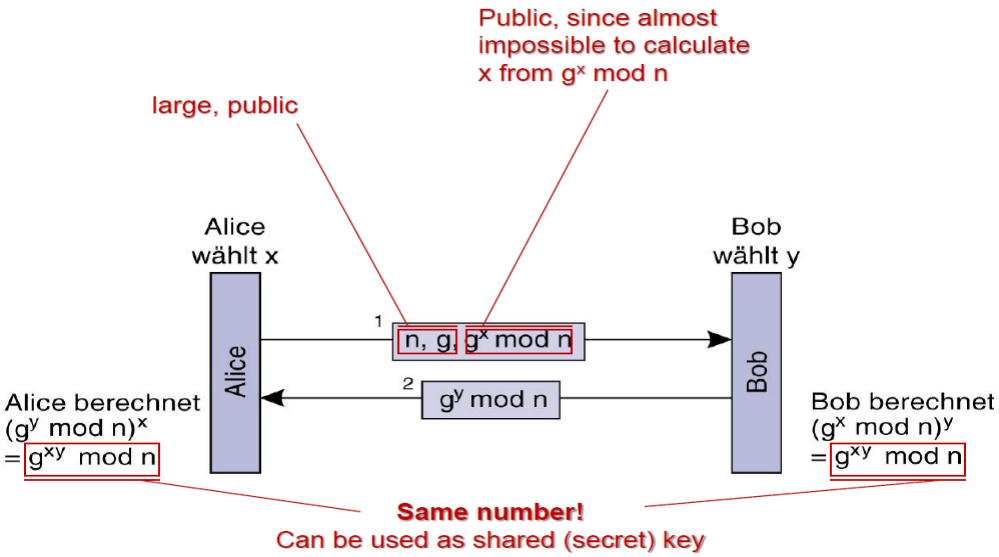
Besser: Message digest: Authentication und Secrecy nicht vermischen. Es sollte möglich sein, Nachricht parallel sowohl unverschlüsselt und gehasht als auch verschlüsselt senden. Danach selbst erstellten hash von unverschlüsselter Nachricht und entschlüsselten hash vergleichen.



Anforderungen an eine Hash-Funktion
A hash function must be *easy to compute* the hash function for a given message, it is *computationally infeasible to generate a message given the hash function* and it is *collision resistant* meaning that two different messages should always result in different hashes.

102. **Key establishment: Diffie-Hellman**

- Alice und Bob einigen sich auf 2 große Zahlen n (Primzahl) und g. Können öffentlich sein.
- Alice wählt große Zahl x und hält sie privat, Bob wählt große Zahl y (auch privat).



103.**Key distribution**

- Secret keys: Alice erstellt einen Key und bringt ihn (über eine sichere Verbindung [out of band]) zu Bob.
Oder verwende KDC

- Public keys: Wie kann ich garantieren dass A's public key wirklich von A ist?
Auch über gesicherte Verbindung übertragen oder: Verwende eine Certification Authority (CA) um public keys auszugeben. Public
keys werden in einem Zertifikat bei der CA signiert.
- Trust Hierarchy: Ich kann mich von einem Zertifikat in einer trust hierachy hocharbeiten bis ich zu einem Zertifikat komme, dem ich vertraue.

104. **Wie können Zertifikate ungültig gemacht werden?**
- Certificate Revocation Lists (CRL) werden von CA regelmäßig veröffentlicht
- Expiration Time (Ungültig nach einer gewissen Zeit), kombiniert mit CRL
- In Practice: Zertifikate mit begrenzter Lebenszeit

105. **Access Control**

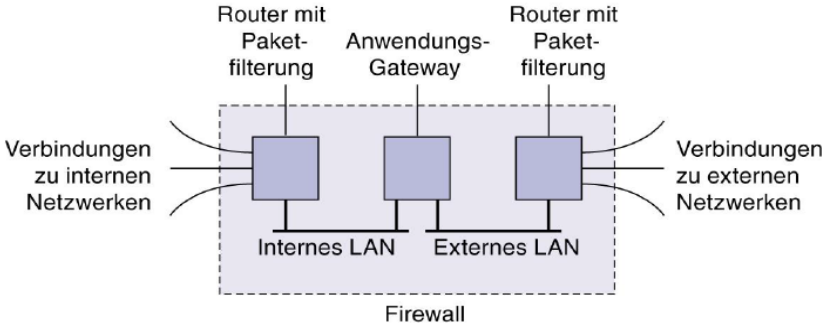
- **ACM (Access Control Matrix):** Eine Matrix in der für jedes Subjekt und jedes Objekt gespeichert ist, welche Operationen er_sie auf das Objekt ausführen kann
-**ACL (Access Control List) :** Jedes Objekt O hat eine ACL, in der gespeichert ist, welches Subjekt welche Operation auf diesem Objekt ausführen kann
Capabilities: Jedes Subjekt S hat eine Fähigkeit: ACM[S,*] beschreibt die erlaubten Operationen für jedes Objekt (oder Kategorie von Objekten)

=> Kann schnell groß werden. Deswegen verwende protection domains

106.**Was sind Protection Domains? Was ist der Unterschied zwischen Groups und Domains?**
- Bündelung von Zugriffsrechten zu häufig verwendeten Bündel.
- Group: Ein User gehört zu einer speziellen Gruppe, die gewisse Rechte hat
- Roles: Ein User hat eine oder mehrere Rollen, die gewisse Rechte haben
- User haben Zertifikate die zeigen welche Gruppen / Rollen sie haben

=> Unterschied: Gruppen ändern sich normalerweise nicht (z.B. „Student_in“ in TISS). Rollen können sich schon häufig ändern.

107.**Was ist eine Firewall?**
- Firewall checkt auf Netzwerkpaketebene ob irgendwas komisch ist und filtert gewisse Pakete raus.
- Größtes Problem von Firewalls: „Bring your own Devices" (z.B.: USB-Stick)



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.

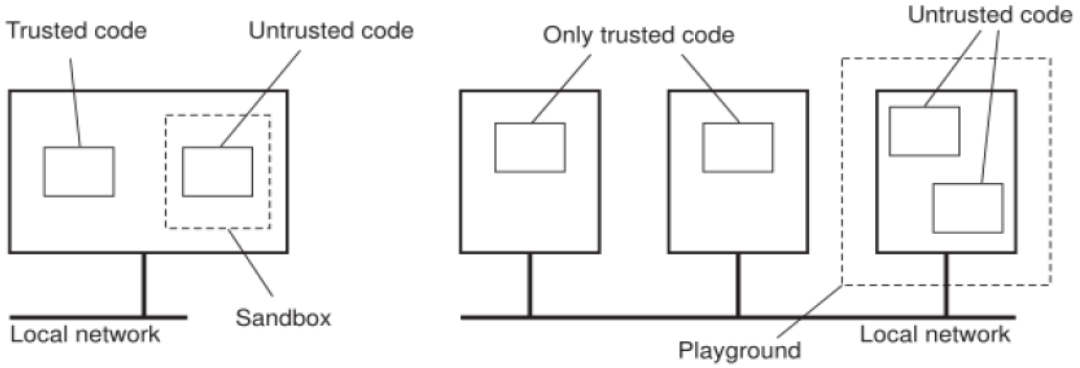
Application-level gateway

- Packet inspection: interpret content based on application semantics
- Mail example: drop attachments with exe files
- Web example: filter out scripts or applets

Secure mobile code: Protecting agent

Allgemein: Code auf anderen Maschinen ausführen.
Hängt davon ab, dass wir dem vertrauen, was der Host anbietet. Also konzentrieren wir uns mehr auf Kontrolle, ob Veränderungen auftreten. Lösung: EINE sehr strenge policy basierend auf ein paar einfachen Mechanismen implementieren. Weiterführung siehe Frage 108.

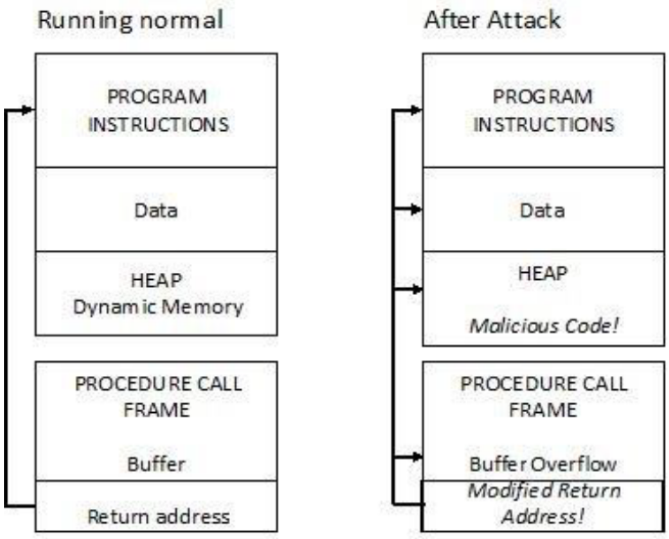
108.**Was ist der Unterschied zwischen einem Sandbox und einem Playground model?**
- Sandbox: (Remote)-Code darf nur ein gewisses Set an Befehlen ausführen und nur auf eine gewisse Menge an Ressourcen zugreifen. Beispiel: Browser-Tab
- Playground: Funktioniert ähnlich, aber der Code wird auf einem eigenen ungeschützten System ausgeführt. Beispiel: Virtual Machine



Stack Buffer Overflow

Häufiges Sicherheitsproblem in nicht verwalteten Programmiersprachen (C, C++)

- Daten eingeben die größer sind als der reservierte Platz am sTack
- Ermöglicht es einem Angreifer den existierenden return address pointer des Procedure calls mit einer anderen Addresse zu überschreiben, welche auf den bösartigen Code zeigt den der Angreifer davor am heap gespeichert hat.
  - Erlaubt es Angreifer beliebigen Code auszuführen



109.**Was ist ein Side-Channel Angriff?**
- Finde das Geheimnis / den Schlüssel eines Mechanismus heraus. Beispiel:
  - Passwort herausfinden
  - NSA verlangt Private Keys von CAs
  - Reverse-Engineering

Oft durch Dummheit möglich (Social Engineering): „Hallo ich bin von der IT Abteilung. Gib mal deine Accountdaten!“

110.Cross-Side Scripting Attack (XSS)

Some web applications do not sufficiently check data received from users

- Similar principle to SQL injection
- Allows attacker to inject arbitrary scripts into a legit (trustable) web site
- Example: blog with commentary function that accepts arbitrary HTML code

110.**Warum kann ein DDoS Angriff auf einen Webserver zu Problemen bei den Nutzern führen?**

The server is too busy responding to requests and trying to set up connections that never materialize into actual connections. Since the server is too busy responding to these spam requests, it cannot respond to any (or only a few) legitimate requests. Furthermore, since the spam requests often look just like real traffic it is difficult to identify what is and is not legitimate.

Attacker uses a network of hacked machines

- Bots/Zombies overload the resources of the target with requests
- Difficult to protect against (needs to be done at ISP level)
- Difficult to identify the attacker (all request come from unassuming zombies)

Current Trends

Gartner Hype Cycle
Viele neue Technologien erhalten anfangs viel Aufmerksamkeit, danach stürzen sie meistens jedoch ab und verschwinden, ohne sich etablieren zu können. Phasen:
Innovation Trigger – Peak of Inflated Expectations – Trough of Disillusionment – Slop of Enlightenment – Plateau of Productivity

111.**Gründe warum in der heutigen Zeit die Verwendung von Webservices immer weiter verbreitet und wichtiger wird**
Web Services werden immer wichtiger, da sie von überall aus nutzbar sind und wohldefinierte Schnittstellen bieten, welche keine spezifische Implementierung (z.B. immer genau in C++) voraussetzen. Sie können dadurch relativ leicht durch andere Implementierungen ausgetauscht werden, falls das (irgendwann) notwendig werden sollte (z.B. aus Performancegründen bei starkem Wachstum). Ein weiterer Vorteil ist außerdem die starke Wiederverwendbarkeit.

112.**Was bedeutet der Begriff „Rapid elasticity“ im Kontext von Cloud Computing.**
Rapid elasticity beschreibt die mehr oder weniger unbeschränkte Kapazität von Ressourcen, also das Abrufen zusätzlicher Ressourcen "on demand". Wird mehr Leistung benötigt, wird einfach mehr Leistung angefordert (z.B. zu „Peak Loads“ Zeiten). Und das Ganze selbstregulierend und ohne Unterbrechungen. In Folge können natürlich auch wieder Ressourcen freigegeben werden, wenn kein Bedarf mehr besteht.

Elastic Computing System:

Reactive (konstante Interaktionen), Hybrid (kontinuierliche + diskrete Ereignisse), Real-time, Self-adaptive, Distributed (autonome Entitäten, kommunizieren & koordinieren).
Es gibt Resource Elasticity, Quality elasticity, Costs & Benefits Elasticity (= Elasticity Signature)
Elasticity (vom Kontext abhängig, dynamisch) != Scalability (vom Kontext unabhängig, statisch)!

Cloud Service Models
- Cloud Infrastructure as a Service (IaaS): Computer Infrastruktur als Service (Virtual Machines, Speicher, …) (z.B. Amazon S3)
- Cloud Platform as a Service (PaaS): Computing Plattform und Solution Stack als Service (execution environment / framework) (z.B. Google App Engine)
- Cloud Software as a Service (SaaS): Software wird angeboten, z.B. Google Calendar

113.**4 Cloud Deployment Models**
- Private Cloud: Anwendungen nur für eine einzige Organisation
- Community Cloud: wird von mehreren Organisationen geteilt
- Public Cloud: Kann öffentlich verwendet werden, gehört einer Organisation die die Cloud-Services verkauft (z.B. Amazon WS)
- Hybrid Cloud: Mischform aus min. 2 der oberen Models

114.**Cloud Computing**
On-Demand Services, automatische Miete von Resources, Sehr gut Netzwerkanbindung, Starke Verwendung von Virtualization Software (Resource Pooling), Rapid elasticity (virtuell unlimitierte Resourcen)

115.**Internet of Things erklären?**
Physikalische Objekte (Wie Kühlschrank) werden in ein IT-Netzwerk eingebunden und können kommunizieren. Sie werden „Smart Objects“. Technologien: RFID, Sensor Network, Ipv6
Kann in sehr großem Bereich eingesetzt werden.

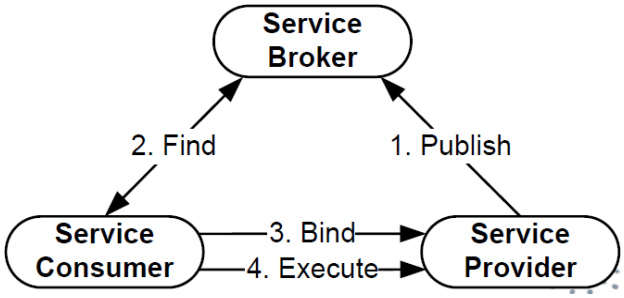
Fog Computing

Cloud Computing problematisch für Internet of Things: weit verteilte Geräte, während Cloud zentralisiert ist. Außerdem wollen viele keine Daten in die Cloud geben.
Fog Computing nutzt dezentral Kapazitäten von Geräten (z.B. Speicher / Rechenleistung von mobilen Geräten). → also praktisch eine dezentrale flexible Cloud mit selben Grundprinzipien (on-demand, lease/release).

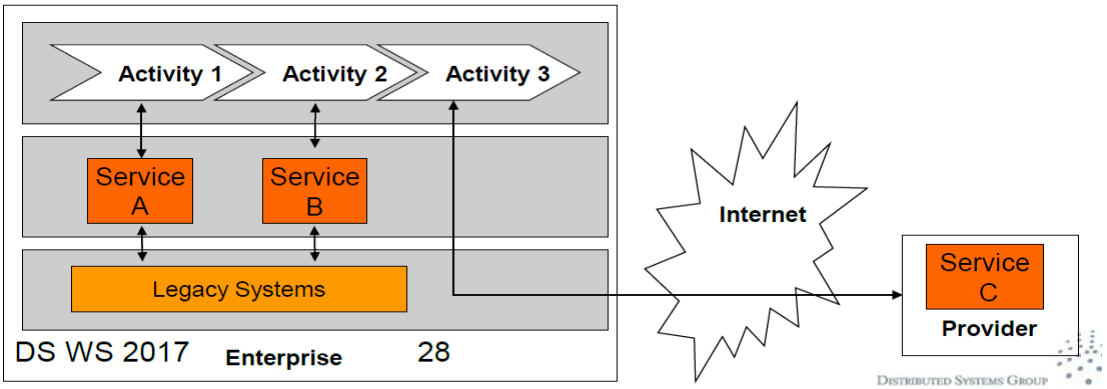
Fog Cell: Softwarekomponente welche auf einem virtualisierten IoT Gerät läuft.
Fog Colony: Mikro-Datenzentren aus einzelen Fog Cells aufgebaut.

Internet of Services und Service Oriended Architecture
IoS:
Software Services werden direkt im Internet angeboten (keine Softwareinstallation notwendig).
Technologien: REST, WDSL, SOAP, Microservices. Fundament für Cloud Computing
SoA:
Aus einzelnen Services gebaute IT Architektur, in einem einzelnen Gesamtsystem enthaltene Software Komponenten mit bestimmter Funktionalität. Viele einfache Anwendungen werden zu komplexem System zusammengeschlossen.

Rollen in SoA:



Workflows and Services: Workflows sind durch IT mögliche Geschäftsprozesse. Services können zu Workflows zusammengesetzt werden. Services verpacken Funktionalität von Legacy Services. Integration externer Services. → Services ermöglichen schnelle Komposition verteilter Workflows



Motivation / Gründe für SoA:
Globalisierung, Inter-organisatorische Zusammenarbeit, Business Process Outsorcing (BPO), Flexibilität für Geschäftsprozesse → Flexible, koppelbare IT Architekturen notwendig!
Unterschiede SoA vs IoS:
SOA ist ein Konzept um IT Software in einer Organisation zu verwalten
IOS Ist von Anfang dafür gedacht dass bestimmte Services im Internet gezielt genutzt werden

116.**Was bedeutet "measured service" im Kontext von Cloud Computing?**
Zu zahlst nur für das was du auch verwendest, also keine Anschaffung von Infrastruktur, sondern das dynamische Mieten von Ressourcen, die genau nach Verwendung (Teilweise im Minutenbereich oder per Request) verrechnet werden. (Beispiel: Kuh anschaffen oder Milch kaufen?)

117.**Welche Risiken bestehen bei Ressourcen-Underprovisioning und Ressourcen-Overprovisioning?**
- Underprovisioning: Benötigte Kapazität kann nicht erreicht werden
- Overprovisioning: Ungenützte Kapazitäten (darum häufig flexible Cloud statt festes Datenzentrum)
- Dagegen hilft: „stretch“ und „shrink“

Peer-to-Peer

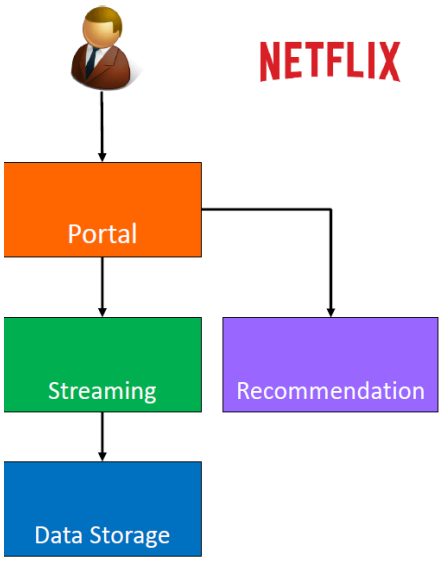
Komponenten interagieren direkt miteinander durch den Austausch von Services.
Alle Peers sind gleich viel wert.
Jede Peer Komponente bietet an / benutzt ähnliche Services

A Peer-to-Peer (P2P) system is„a self-organizing system ofequal, autonomous entities(peers) [which] aims for the shared usage of distributed resources in a networked environment avoiding central services.“
„A system with completely decentralized self-organization and resource usage.“

Key Characteristics: Equality, Autonomy, Decentralization, Self-Organization, Shared resources
Anwendung: VoIP, Media Streaming, File sharing etc
Gründe um P2P zu nutzen:
- Kosten: Computing / Speicher kann outsourced werden
- Hohe Erweiterbarkeit
- Hohe Scalability
- Fault Tolerance

Centralized P2P (zentrale Einheit bietet Services an) z.B. Napster, Pure P2P (alle unabhängig) z.B. Gnutella 0.4, Hybrid P2P (dynamic central entities) z.B. Gnutella 0.6, DHT-based (Verbindungen im Overlay sind „fix“) z.B. Chord

Microservices / Microservice-based interactions



Kleine Softwarekomponenten, die alle für sich laufen. Kleine Softwarecontainer irgendwo in der Cloud deployed. zB. Eine Instanz eines Containers in der Cloud, bei mehr Nutzern eine weitere Instanz dazuschalten. Wird für viele große IT-Plattformen verwendet. Siehe Beispiel Netflix:

- Benutzeranfragen werden durch Servicefolge bedient
- meisten Services nicht extren verfügbar
- viele Services liegen in mehreren Instanzen vor

Infrastruktur-Dienste:

- Container-Umgebung (docker: jede Instanz wird in docker-container gehostet. Docker können auf VM in Cloud gehostet werden).
- Cloud-Hosting
- Anfragenrouting
- Monitoring (z.B. hat eine Instanz Fehler mach ne neue und hau die alte ausm System)
- Ausfallsicherung & Trennschalter (Circuit Breaker)
- Service Discovery
- Lastausgleich (zuviel Last? Neue Instanz!)