Zusammenfassung Verteilte Systeme

von Joshua Oblong, 01631789

Inhaltsverzeichnis

Verteilte Systeme

Internet of Things

Alles ist über das Internet verbunden, nicht nur Menschen, sondern auch Dinge wie Licht, Klimaanlagen, Aufzüge, etc.

Nachteile:

- Verteilte Systeme werden größer (nicht mehr wartbar)
- Menschen und Maschinen miteinander koordinieren
- Maschinen müssen mit uns arbeiten

Was ist ein verteiltes System?

„Eine Ansammlung an unabhängigen Computern, die dem Benutzer als einziges zusammenhängendes System vorkommen."

Es können aber auch mehrere Systeme in einem System sein.

„Eine Ansammlung an autonomen Computern, die durch ein Computer-Netzwerk verbunden sind und von einer Software, die es ermöglicht, diese Ansammlung als integrierte Einrichtung operieren zu lassen, unterstützt wird."

„Du weißt du hast ein verteiltes System, wenn der Absturz eines Computers, von dem du noch nie gehört hast, dich daran hindert deine Arbeit fertig zu bekommen."

Arten von Verteilten Systemen

- Objekt-/Komponentenbasiert (CORBA, EJB, COM)
- Dateibasiert (NFS)
- Dokumentenbasiert (WWW, Lotus Notes)
- Koordinations-/Eventbasiert (Jini, JavaSpaces, publish/subscribe, P2P)
- Ressourcenorientiert (GRID, Cloud, P2P, MANET)
- Serviceorientiert (Web Services, Cloud, P2P)

Insbesondere:

- **Distributed Computing** (Cluster, GRID, Cloud)
- **Distributed Information Systems** (EAI, TP, SOA)
- **Distributed Pervasive** (oft P2P, UPnP in Heimsystemen, Sensornetzwerke, …)

Konzepte von Verteilten Systemen

- Kommunikation
- Parallelität und OS Support (kompetitiv, kooperativ)
- Namensgebung und Finden von Ressourcen
- Synchronisation und Vereinbarung
- Konsistenz und Replikation
- Fehlertoleranz
- Sicherheit

Warum verteilen wir überhaupt?

- Benutzer mit Ressourcen und Services verbinden
    - Grundlegende Funktion von Verteilten Systemen
- Zuverlässigkeit und Sicherheit
    - Verfügbarkeit, Fehlertoleranz, Eindringtoleranz
- Performance
    - Latenz, Datenrate, etc.

Designziele

- Teilen von Ressourcen (kollaborativ, kompetitiv)
- Transparenz
- Verstecken interner Strukturen, Komplexität
    - Offenheit
    - Portabilität, Interoperabilität
- Standardregeln/Protokolle bereitstellen
- Skalierbarkeit
- Möglichkeit das System einfach zu erweitern
- Parallelität
    - Von Natur aus parallel, nicht nur simuliert
- Fehlertoleranz, Verfügbarkeit

Die 8 Irrtümer von Verteilten Systemen

1. Das Netzwerk ist zuverlässig
2. Keine Latenz
3. Bandweite ist unendlich
4. Das Netzwerk ist sicher
5. Topologie (Struktur des Netzwerks) ändert sich nicht
6. Es gibt nur einen Administrator
7. Keine Transportkosten
8. Das Netzwerk ist homogen

Benutzer mit Services verbinden

- Zugang und Teilen von Ressourcen
- Wirtschaft und Richtlinien
- Zusammenarbeit durch Informationsaustausch
- Kommunikation (Convergence, VoIP)
- Groupware und virtuelle Organisationen
- Elektronischer und Mobiler Handel
- Sensor-/Akteur-Netzwerke in autonomem und allgegenwärtigem Rechnen
- Kann Sicherheit und Privatsphäre kompromittieren

Quality of Service (QoS)

Quality of Service ist ein Konzept, mit dem Clients das benötigte Level of Service (SLA) angeben können.

Beispiele:

Videokonferenz: Client bevorzugt zuverlässige Übertragungszeiten über garantierter Übertragung
Finanz-Apps: Client möchte lieber verschlüsselte Kommunikation als schnelle Kommunikation

Man kann nicht alles haben! (**Trade-offs**)

Transparenz

- Konzept: Verstecken von verschiedenen Aspekten einer Verteilung vor dem Client. Es ist das ultimative Ziel von vielen verteilten Systemen.
- Es kann durch das Bereitstellen von Lower-Level (System) Services erreicht werden
- Der Client benutzt die Services statt der hardgecodeten Informationen
- Der Service Layer stellt einen Service mit einer gewissen Quality of Service zur Verfügung

Transparenz	**Beschreibung**
Zugriff	Verstecken, wie auf die Ressource zugegriffen wird
Standort	Verstecken, wo sich die Ressource befindet
Migration	Verstecken, dass die Ressource sich evtl. zu einem anderen Standort bewegt
Umzug	Verstecken, dass die Ressource sich zu einem anderen Standort bewegt während sie benutzt wird
Replikation	Verstecken, dass die Ressource repliziert wird
Parallelität	Verstecken, dass die Ressource evtl. von mehreren Benutzern geteilt wird
Versagen	Verstecken des Versagens und der Wiederherstellung einer Ressource

Grad der Transparenz

- Nicht blindlings versuchen jeden Aspekt der Verteilung zu verstecken
- Performance-Transparenz ist schwierig (LAN/WAN)
- Trade-off Transparenz/Performance
    - Verstecken von Fehlern
    - Replikatkonsistenz
- → Transparenz ist ein wichtiges Ziel, muss aber mit allen anderen nicht-funktionalen Anforderungen zusammen und mit Respekt zu besonderen Forderungen berücksichtigt werden

Offenheit

- Services gemäß Standardregeln anbieten (Syntax und Semantik: Format, Inhalt, Sinn)
- Formalisiert in Protokollen
- Schnittstellen (IDL): Semantik oft informell
    - Vollständig → Interoperabilität: Kommunikation zwischen Prozessen
    - Neutral → Portabilität: Verschiedene Implementationen einer Schnittstelle
- Flexibilität: Komposition, Konfiguration, Ersatz, Erweiterbarkeit (CBSE)

Richtlinien von Mechanismen trennen

- Granularität: Objekte oder Applikationen?
- Komponenteninteraktion und Kompositionsstandards (anstatt geschlossen/monolithisch)
- Z.B.: Webbrowser erleichtert es gecachete Dokumente zu speichern, aber Richtlinien zu cachen kann willkürlich vorgenommen werden.

Wie erreicht man Offenheit?

Beispiele im Web:

- Verschiedene Web Server und Web Browser können zusammenarbeiten
- Neue Browser können eingeführt werden, um mit vorhandenen Servern zu arbeiten
- Plugin-Schnittstelle erlaubt es neue Services hinzuzufügen

Skalierbarkeit

- Die **Fähigkeit zu wachsen**, um über mehrere Dimensionen hinweg den steigenden Anforderungen gerecht zu werden
    - Größe (Benutzer und Ressourcen)
    - Geographisch (topologisch)
    - Administrativ (unabhängige Organisationen/Domains)
- System bleibt effektiv
- System und Anwendung sollten sich nicht ändern müssen
- Trade-Off Skalierbarkeit/Sicherheit

Herausforderungen Skalierbarkeit (Größe)

- Steuern der Kosten von physischen Ressourcen: Die benötigte Quantität sollte O(n) sein
- Steuern des Verlustes an Performance: In hierarchischen Systemen sollte es nicht schlechter als O(*log* n) sein
    - O(log n) bedeutet, dass die Zeit sich linear erhöht während n sich exponentiell verringert. Also wenn es 1 Sekunde dauert, um 10 Elemente zu berechnen, wird es 2 Sekunden dauern um 100 Elemente zu berechnen, 3 Sekunden zum Berechnen von 1000 Elementen, usw.
- Verhindern, dass Softwareressourcen ausgehen. Überkompensation könnte aber umso schlimmer sein: z.B. Internetadressen oder Oracle7 2TB Beschränkung
- Performance Bottlenecks vermeiden (Zentralisierte Services, Daten oder Algorithmen)

Performance Bottlenecks

Konzept	Beispiel
Zentralisierte Services	Nur ein Server für alle Benutzer
Zentralisierte Daten	Nur ein Online-Telefonbuch, Zentraler DNS
Zentralisierte Algorithmen	Routing basierend auf der ganzen Information

Dezentralisierte Algorithmen

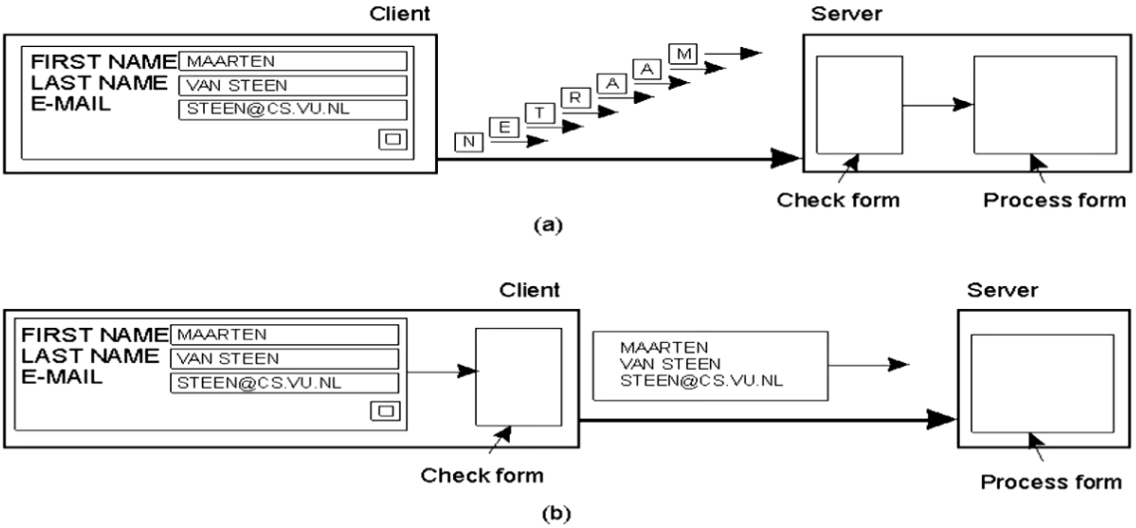
1. Keine Maschine besitzt die vollständigen Systemstatus Informationen
2. Maschinen treffen nur Entscheidungen basierend auf lokalen Informationen
3. Ausfall einer Maschine ruiniert nicht den Algorithmus (Kein Single Point of Failure)
4. Keine implizite Annahme, dass eine globale Uhr existiert

Geographische Skalierbarkeit

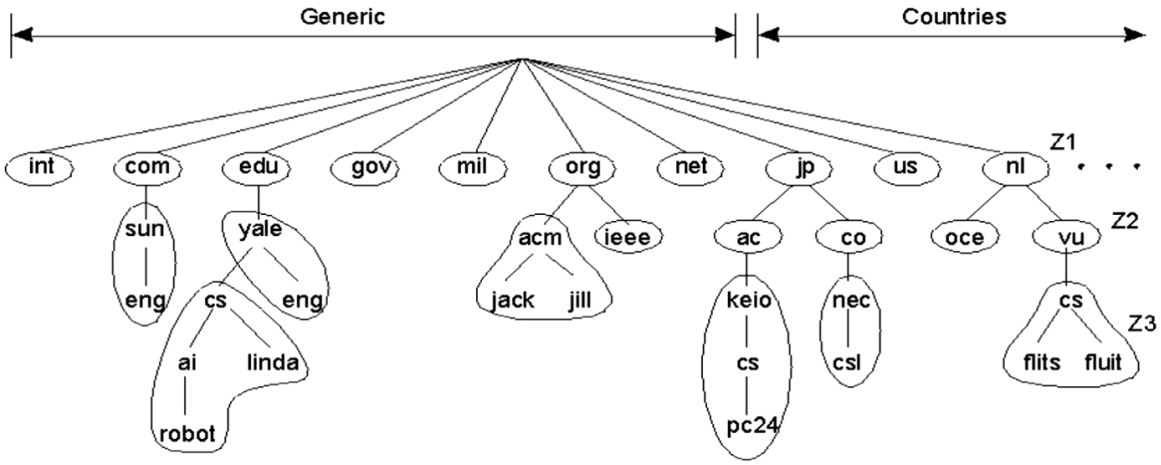
- LAN
    - Synchrone Kommunikation
    - Schnell
    - Broadcast
    - Sehr zuverlässig
- WAN
    - Asynchrone Kommunikation
    - Langsam
    - Point to Point (bspw. Probleme mit Standortservices)
    - Unzuverlässig

Techniken Skalierbarkeit

- Verstecken von Kommunikationslatenzen
  - Asynchrone Kommunikation (Batch Processing, Parallele Anwendungen)
  - Verringern der insgesamten Kommunikation (HMI)
- Verteilung
  - Hierarchien, Domains, Zonen, etc.
- Replikation
  - Verfügbarkeit, Lastausgleich, Verringern der Kommunikation
  - Caching: Nähe, Entscheidung des Clients
  - Konsistenzprobleme können Skalierbarkeit beeinträchtigen



Der Unterschied zwischen **(a) den Server** oder **(b) den Client** das Formular überprüfen zu lassen während es befüllt wird.



Architekturstile

Umgehen mit Komplexität

- **Abstraktion (und Modellierung)**
  - Client, Server, Service
  - Schnittstelle vs. Implementation
- **Verstecken von Informationen (Kapselung)**
  - Schnittstellen-Design
- **Trennung der Verantwortlichkeiten (Separation of concerns)**
  - Schichtung (Beispiel Dateisystem: Bytes, Diskettenblöcke, Dateien)
  - Client und Server
  - Komponenten (Granularitätsprobleme)

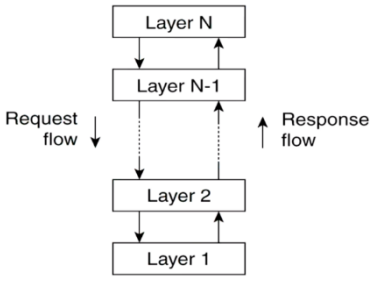
Kommunikationsmodelle

- Multiprozessoren: geteilter Speicher (benötigen Schutz vor parallelem Zugriff)
- Multicomputer: Überreichen von Nachrichten
- Synchronisation in geteiltem Speicher:
  - Semaphoren (atomare Mutexvariable)
  - Monitore (ein abstrakter Datentyp bei dem Operationen von parallelen Threads aufgerufen werden können; unterschiedliche Aufrufe sind synchronisiert
- Synchronisation in Multicomputern: Blockieren beim Überreichen der Nachrichten

Architekturstile

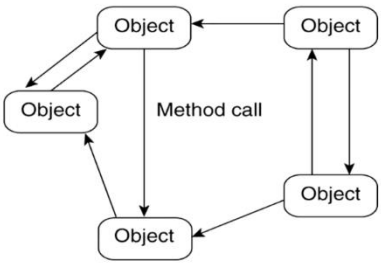
Wichtige Architekturstile für verteilte Systeme

- **Geschichtete Architekturen (Layered)**
- **Objektbasierte Architekturen**
- **Datenzentrierte Architekturen**
- **Eventbasierte Architekturen**



(a)

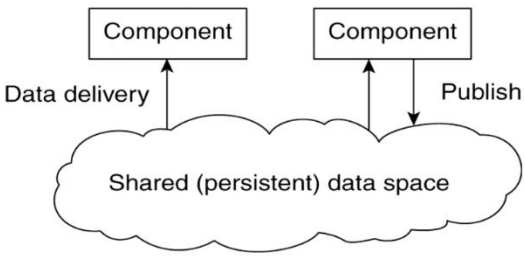
The **layered** architectural style

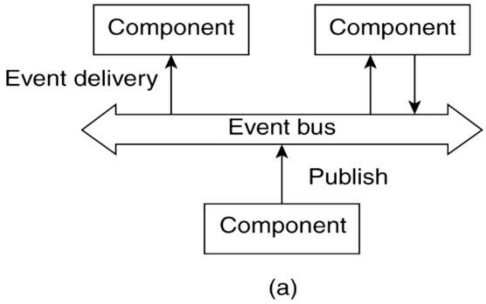


(b)

The **object-based** architectural style

The **shared data-space** architectural style.

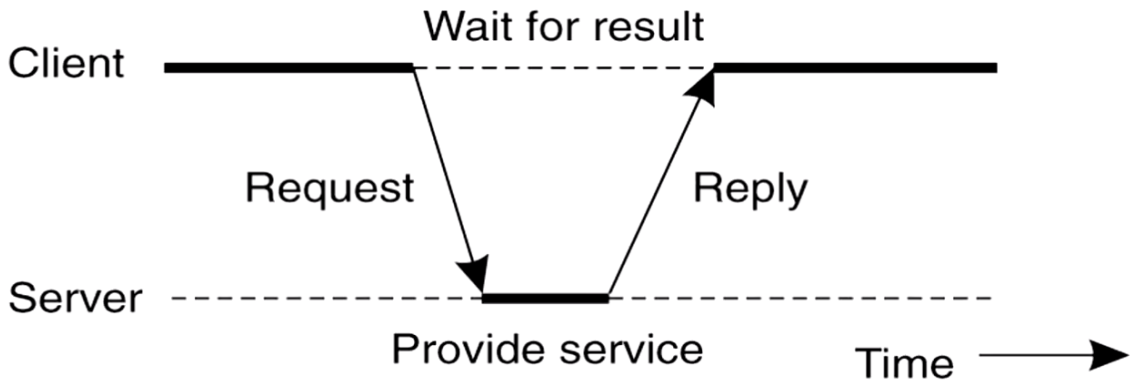




(a)

The **event-based** architectural style

Allgemeine Interaktion zwischen einem Client und einem Server:

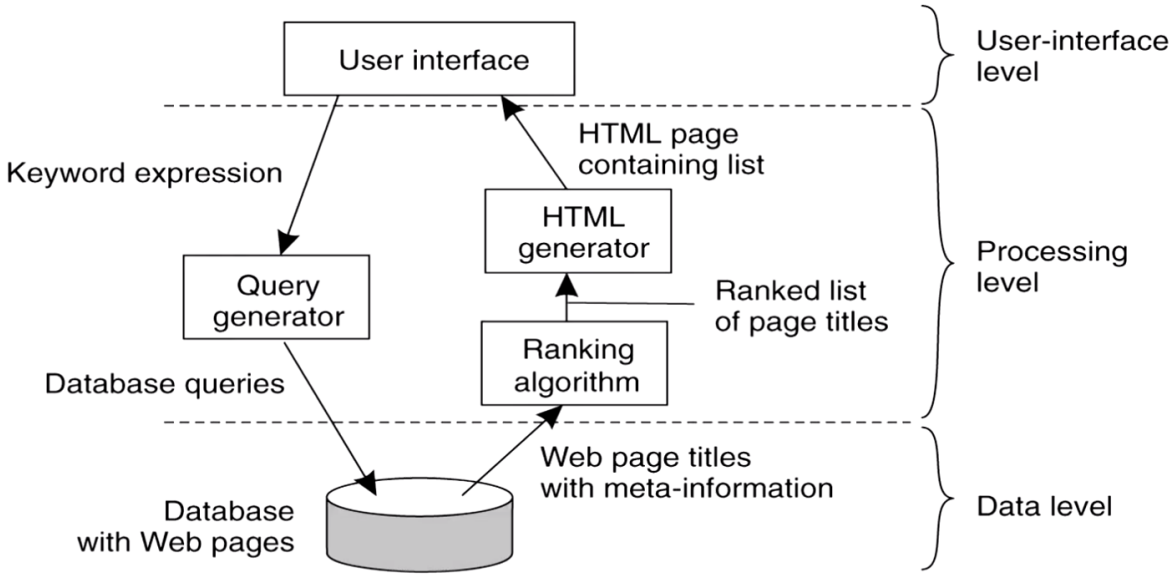


Schichtung der Anwendung

Einteilung der Anwendung in Schichten

- Benutzerschnittstellenschicht
- Verarbeitungsschicht
- Datenschicht

Die vereinfachte Organisation einer Internet Suchmaschine in drei verschiedene Schichten:

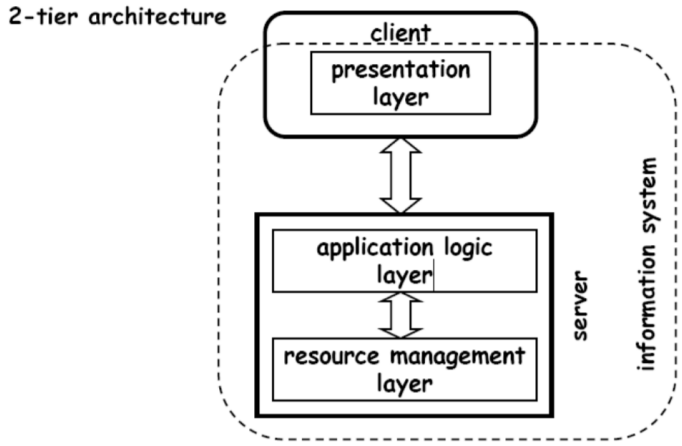


Mehrschichtige Architektur

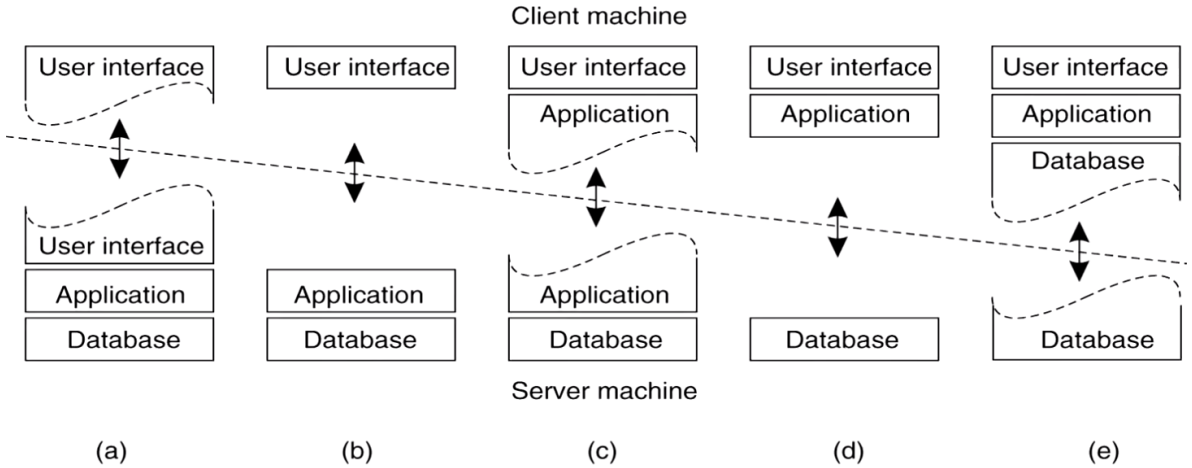
Die **einfachste Organisation wird mit nur zwei Typen von Maschinen erreicht:**

- Eine Client-Maschine, die nur die Programme beinhaltet, die die Benutzerschnittstellenschicht implementieren
- Eine Server-Maschine, die den Rest beinhaltet: die Programme, die die Verarbeitungs- und die Datenschicht implementieren

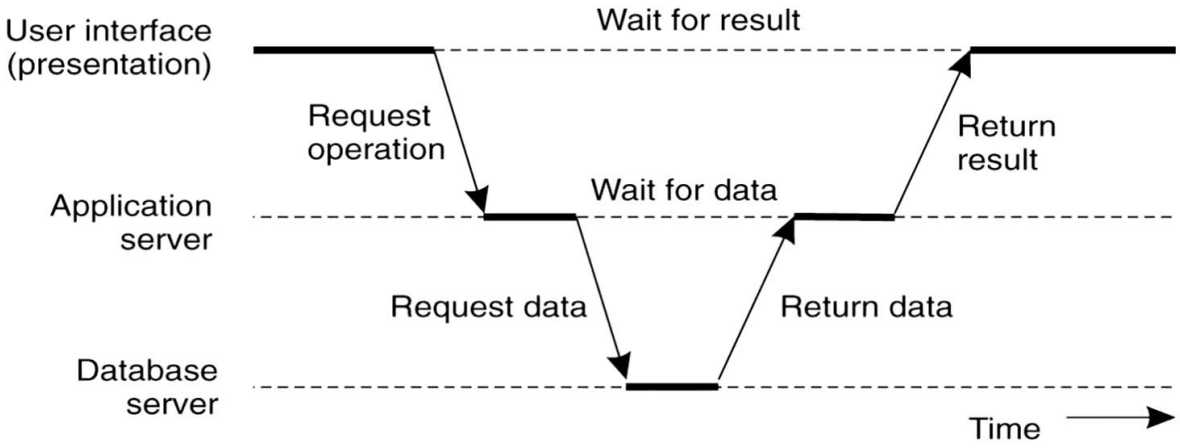
Zweischichtige Architektur:



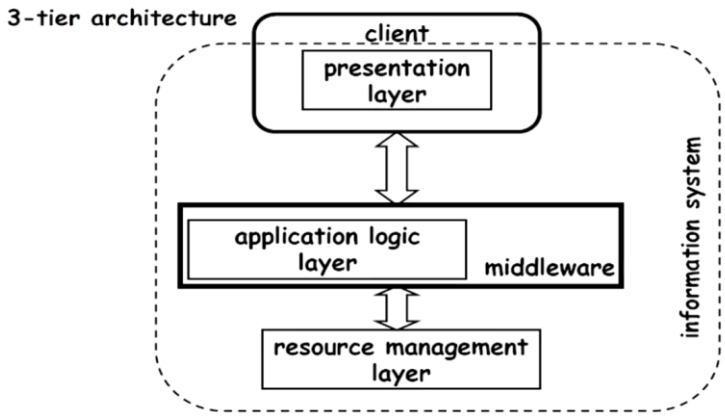
Vertikale Verteilung: Alternative Client-Server Organisationen (a)-(e).



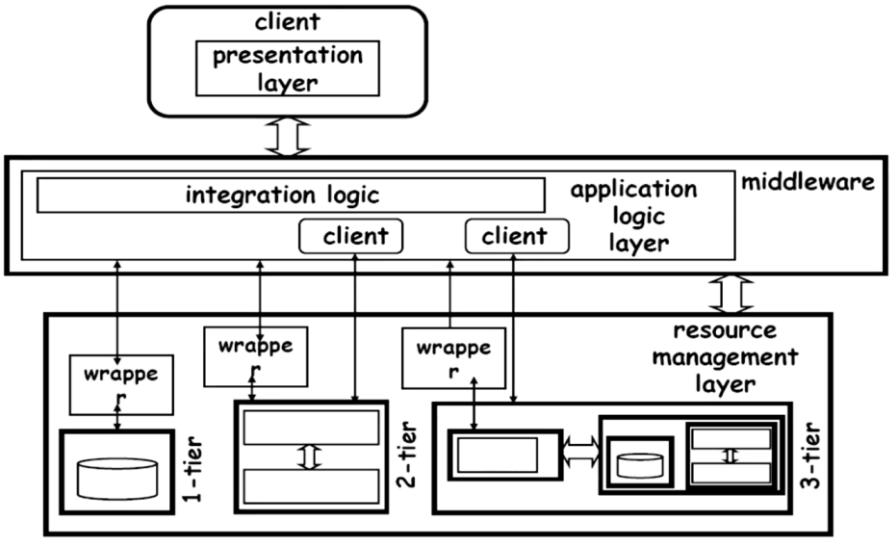
Beispiel eines Servers, der sich als Client ausgibt:



Dreischichtige Architektur:



Mehrschichtige Architektur:



Zusammenfassung

- Ein verteiltes System ist eine Ansammlung an Computern die nahtlos miteinander arbeiten
- Verteilte Systeme haben sich weiterentwickelt und sind allgegenwärtig
- Prinzipien und Techniken werden gebraucht, um mit der Komplexität von verteilten Systemen umgehen zu können (Offenheit, Skalierbarkeit, Architekturstile, etc.)
- Grundlegende Abstraktionen und Konzepte für verteilte Systeme: Client/Server, Schichtenmodell (Multitier), Middleware, Service, QoS, etc.

Prozesse und Kommunikation

Virtuelle Prozessoren – Grundsätzliche Idee

Wir bauen virtuelle Prozessoren in eine Software auf einen physischen Prozessor

- **Prozessor**: Stellt einen Befehlssatz zur Verfügung mit der Fähigkeit, automatisch diese Befehle auszuführen
- **Prozess**: Ein virtueller Softwareprozessor, auf dem einer oder mehrere Threads ausgeführt werden
- **Thread**: Ein minimaler Softwareprozessor, auf dem Befehle ausgeführt werden können

Kontextwechsel

- **Prozessorkontext**: Minimale Sammlung an Werten, die den Registern eines Prozessors gespeichert sind und für die Ausführung von Befehlen verwendet wird
- **Threadkontext**: Minimale Sammlung von Werten, die in den Registern und dem Speicher gespeichert sind und für die Ausführung von Befehlen verwendet wird
- **Prozesskontext**: Minimale Sammlung von Werten, die in den Registern und dem Speicher gespeichert sind und für die Ausführung von Threads verwendet wird

Kontextwechsel: Beobachtungen

- Threads teilen sich denselben Adressraum.
  Kontextwechsel bei Thread kann komplett unabhängig vom Betriebssystem durchgeführt werden
- Prozess-Kontextwechsel ist generell teurer, da das Betriebssystem involviert ist
- Herstellen und Zerstören von Threads ist weitaus günstiger als bei Prozessen

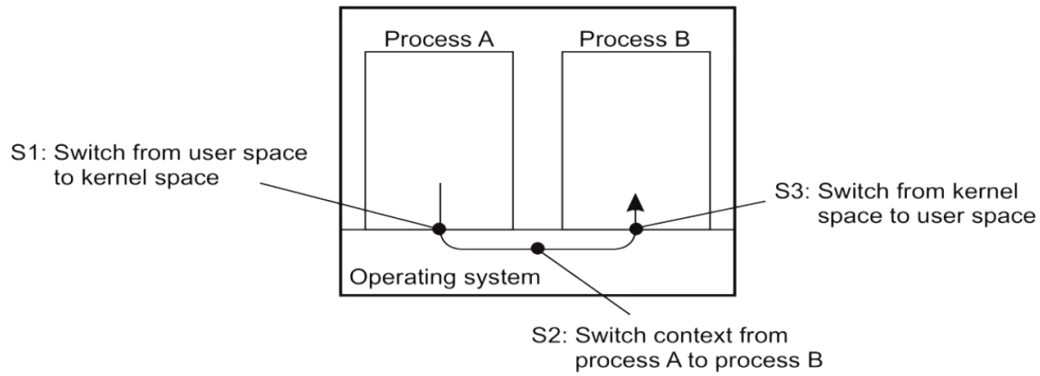
Threads – Allgemeine Informationen

- **Singlethreaded Prozess**: Wenn ein blockierender Systemaufruf ausgeführt wird, ist der komplette Prozess blockiert
- **Multithreaded Prozesse**:
  - Parallelität: Rechnen verschnellern, indem Threads auf verschiedenen CPUs laufen (während die geteilten Daten in einem geteilten Hauptspeicher sind)
  - Große Applikationen
  - Software Engineering: Dedizierte Threads für dedizierte Aufgaben

Warum Threads benutzen?

- **Unnötiges Blockieren verhindern**: Ein singlethreaded Prozess wird bei I/O (Ein- und Ausgabe) blockieren. In einem multithreaded Prozess können wir in diesem Prozess die CPU auf einen anderen Thread umschalten.
- **Parallelität ausnutzen**: Die Threads in einem multithreaded Prozess können so geplant werden, dass sie parallel in einer Multiprozessor oder Multicore Architektur laufen
- **Prozess-Kontextwechsel verhindern**: Große Applikationen nicht als Sammlung von Prozessen strukturieren, sondern durch mehrere Threads

Prozess-Kontextwechsel verhindern:



Trade-Offs:

- Threads nutzen denselben Adressraum: fehleranfälliger
- Kein Schutz von Betriebssystem oder Hardware, dass Threads den jeweils anderen Speicher verwenden
- Thread-Kontextwechsel kann schneller sein als Prozess-Kontextwechsel

Multithreaded Clients

Beispiel: Multithreaded Web-Client:

- Webbrowser überprüft eine eintreffende HTML-Seite und findet heraus, dass mehr Dateien heruntergeladen werden müssen
- Jede Datei wurde von einem eigenen Thread heruntergeladen, welcher jeweils eine (blockierende) HTTP-Anfragie durchführt
- Während die Dateien eintreffen, zeigt der Browser diese an

Threads serverseitig nutzen

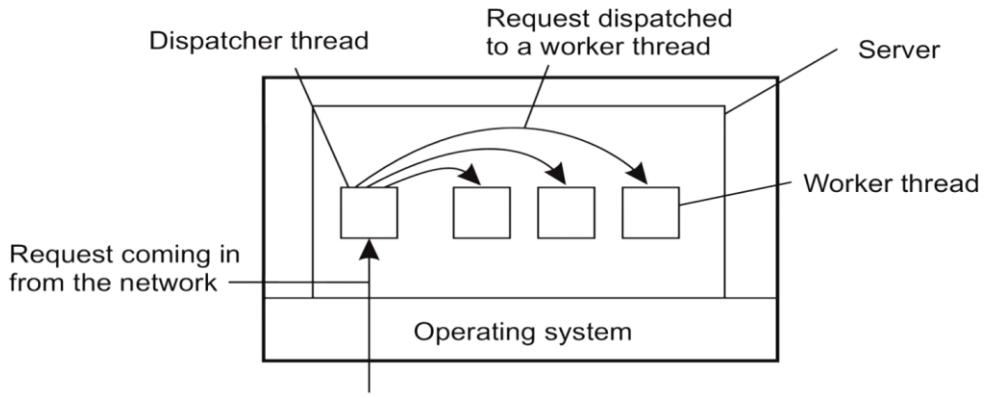
Performance verbessern:

- Einen Thread zu starten ist günstiger als einen neuen Prozess zu starten
- Einen singlethreaded Server zu haben, untersagt einem die einfache Aufrüstung zu einem Multiprozessor System
- Bei Clients: Netzwerklatenz verstecken, indem man auf die nächste Anfrage reagiert während auf die vorherige geantwortet wird

Bessere Struktur:

- Die meisten Server haben hohe I/O Anforderungen. Benutzt man einfache Blockieranfragen, wird die gesamte Struktur vereinfacht
- Multithreaded Programme sind meistens kleiner und verständlicher, aufgrund des vereinfachten Kontrollflusses

Multithreaded Server

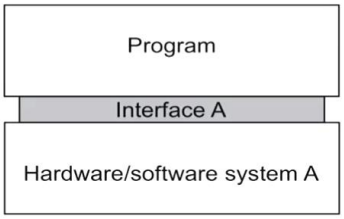


Virtualisierung als Basiskonzept

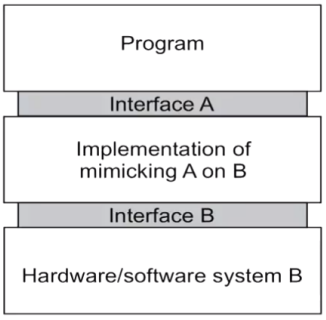
Grundsätzliche Idee: Abstrakte Sicht einer IT-Ressource

- Möglich auf verschiedenen Ebenen
  - Plattform
  - Speicher
  - HDD
  - Netzwerk

Ohne Virtualisierung:



Mit Virtualisierung:



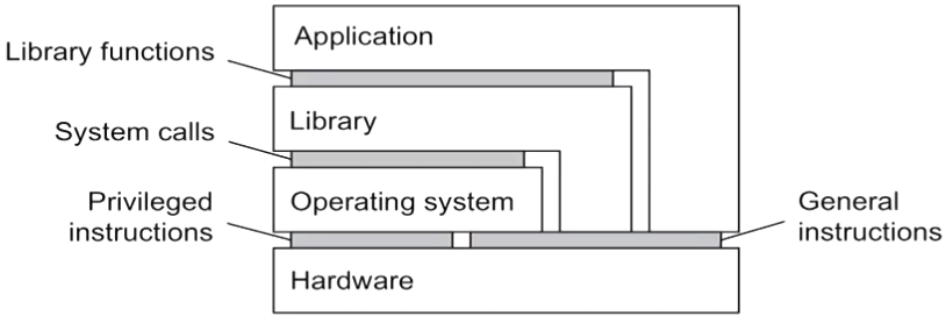
Schnittstellen imitieren

Vier Arten von Schnittstellen auf drei verschiedenen Ebenen:

1. **Befehlssatzarchitektur (ISA)**: Der Befehlssatz mit zwei Unterbefehlssätzen:
   a. Privilegierte Instruktionen: Dürfen nur vom Betriebssystem ausgeführt werden
   b. Generische Instruktionen: Können von jedem Programm ausgeführt werden
2. **Systemaufrufe**, die vom Betriebssystem angeboten werden
3. **Bibliotheksaufrufe**, auch bekannt als Application Programming Interface (API)

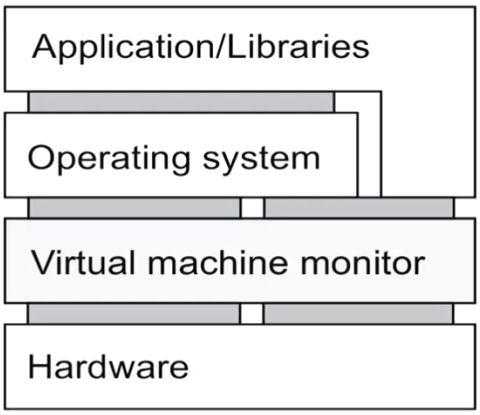
Prozess VMs

Ein Programm wird kompiliert, um Code zu vermitteln, welcher darauf ausgeführt wird.

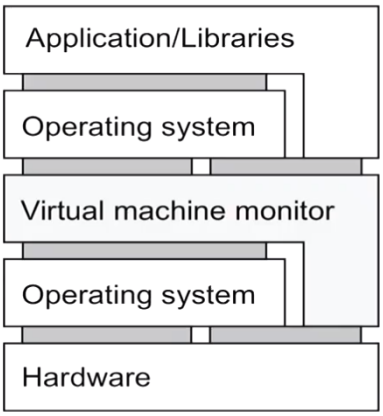


VM Monitore

Nativer VM Monitor:
Eine separate Softwareschicht imitiert den Befehlssatz der Hardware



Gehosteter VM Monitor:
Macht Gebrauch von existierendem Betriebssystem



Vorteil gehosteter VM Monitor:

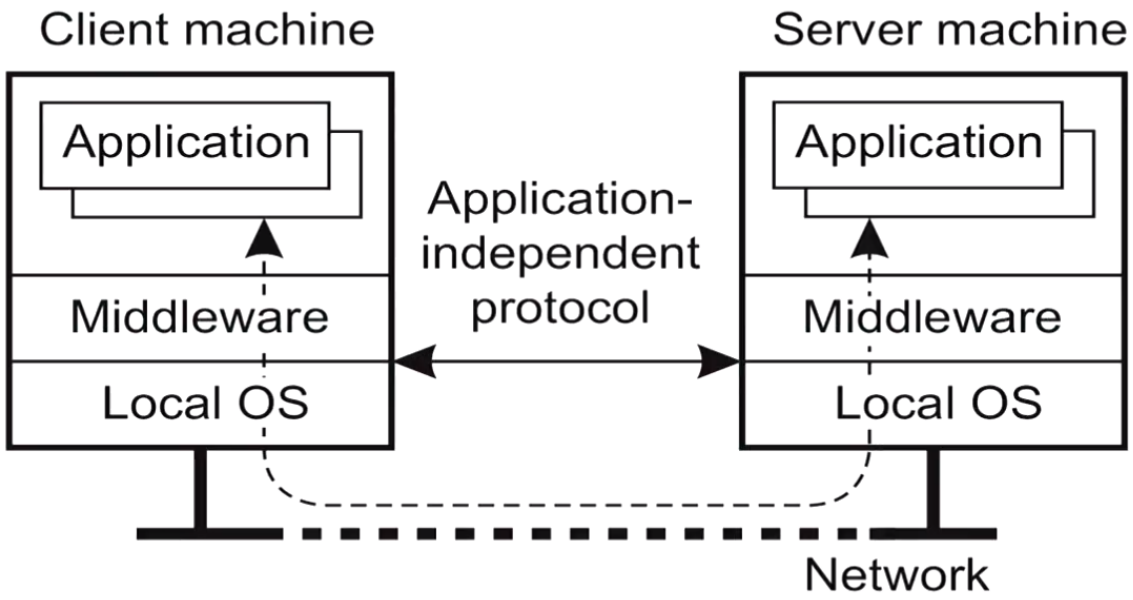
- Kann verschiedene Betriebssysteme laufen lassen

Vorteile Virtualisierung

- Sehr schnell in der Lage neue Ressourcen zur Verfügung zu stellen
- Hardware verändert sich schneller als Software
- Einfachheit der Portabilität und Code-Migration
- Fehlertoleranz: Isolation von Ausfällen entstehen durch Fehler oder Sicherheitsprobleme

Clients: Benutzerschnittstellen

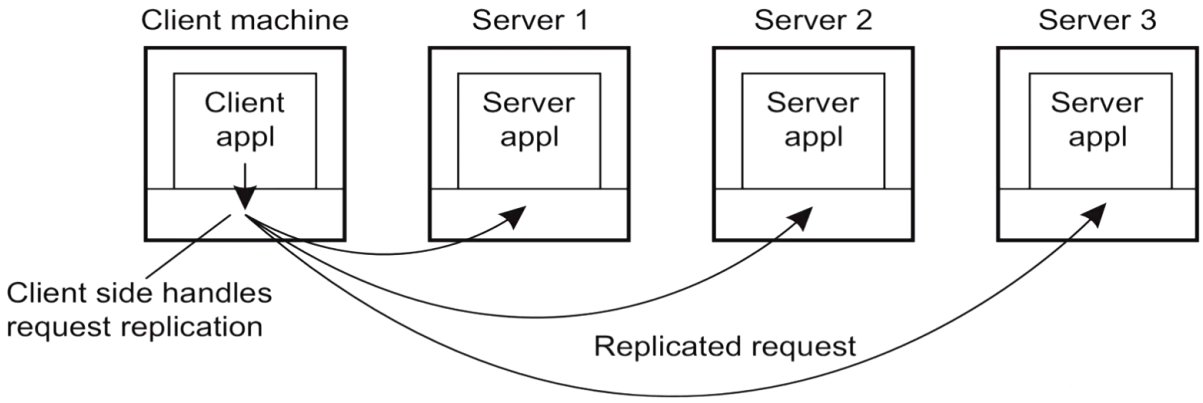
Ein großer Teil der client-seitigen Software ist auf Benutzerinteraktionen fokussiert



Client-Side-Software

Verteilungstransparenz:

- Zugriffstransparenz
- Orts-/Migrationstransparenz
- Replikationstransparenz
- Fehlertransparenz



Server: Generelle Organisation

Grundsätzliches Modell:

- Ein Server ist ein Prozess, der auf eingehende Serviceanfragen wartet
- Der Server stellt anschließend sicher, dass sich um die Anfrage gekümmert wird und wartet danach auf die nächste eingehende Anfrage

Arten von Servern:

- Iterative Server
- Nebenläufige Server

Out-of-band Communication

Problem: Ist es möglich den Server zu unterbrechen, sobald er eine Serviceanfrage akzeptiert hat?

Lösung 1: Nutzung eines separaten Ports für dringende Daten:

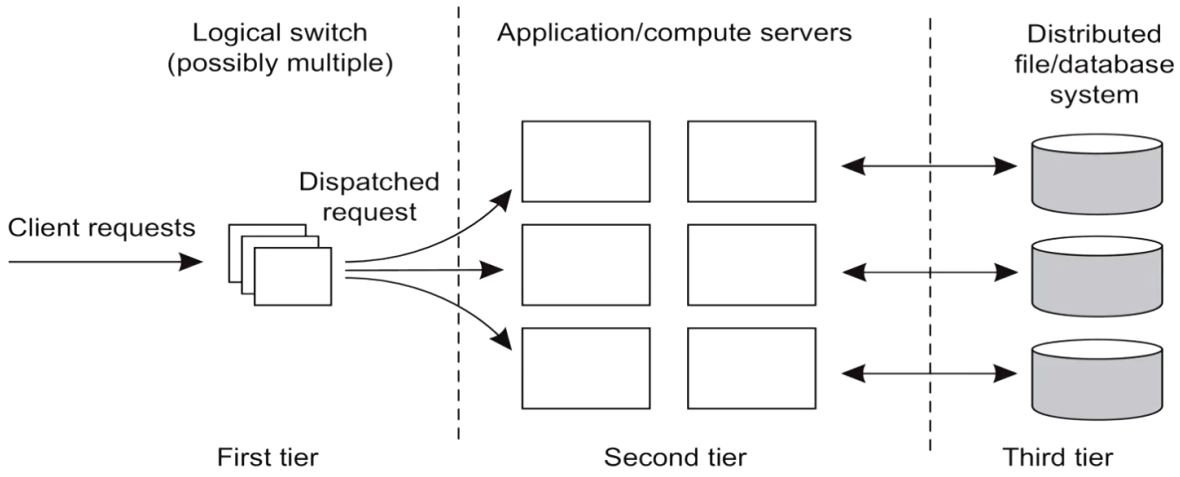
- Server hat einen separaten Thread/Prozess für dringende Nachrichten
- Nachricht kommt rein → Damit verbundene Anfrage wird angehalten

Lösung 2: Nutzung der Out-of-band Communication Befehle der Transportschicht

Statusbehaftete und Statuslose Server

- Statusbehaftete Server:
  - Behält persistente Informationen über seine Clients
  - Performanceverbesserungen möglich, da der Server zusätzliche Informationen über den Client besitzt
- Statuslose Server:
  - Behält keine Informationen über den Status seiner Clients
  - Clients und Server sind komplett unabhängig
  - Statusinkonsistenzen durch Client oder Server Abstürze sind verringert
  - Möglicher Performanceverlust

Server Cluster: 3 Verschiedene Stufen

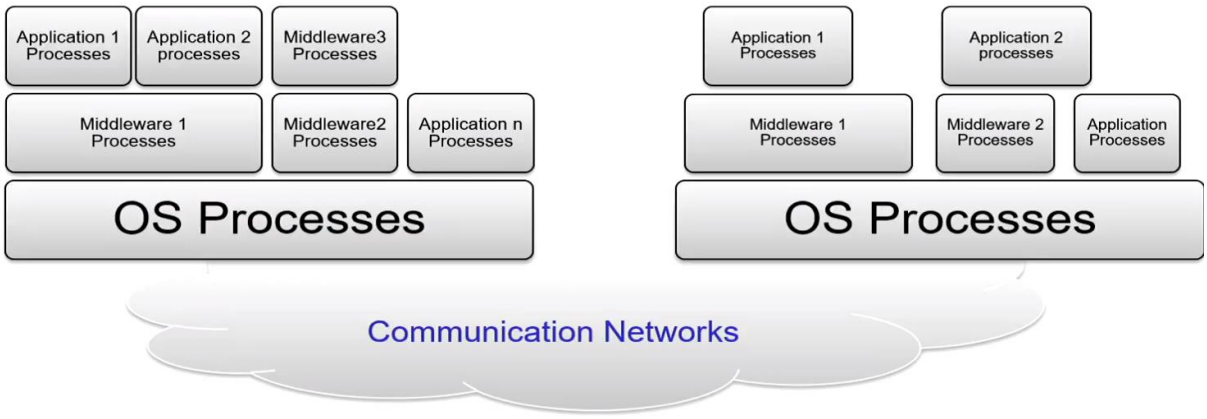


Zusammenfassung

Was bis jetzt gelernt wurde:

- Warum brauchen wir Threads?
- Warum brauchen wir Virtualisierung und was bietet es?
- Wie interagieren Clients und Server?

Kommunikationsentitäten



Kommunikation in verteilten Systemen

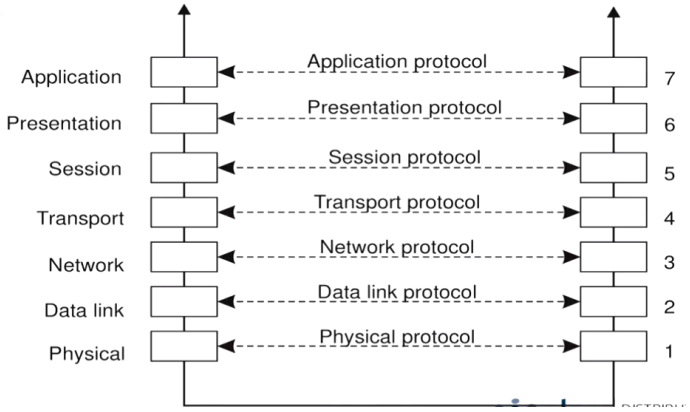
- Zwischen Prozessen innerhalb einer einzelnen Applikation/Middleware/Services
- Zwischen Prozessen die zu anderen Applikationen/Middleware/Services gehören
- Zwischen Knoten die das Konzept von Prozessen nicht kennen (z.B. Sensoren)

ISO/OSI Modell: Überblick

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)	

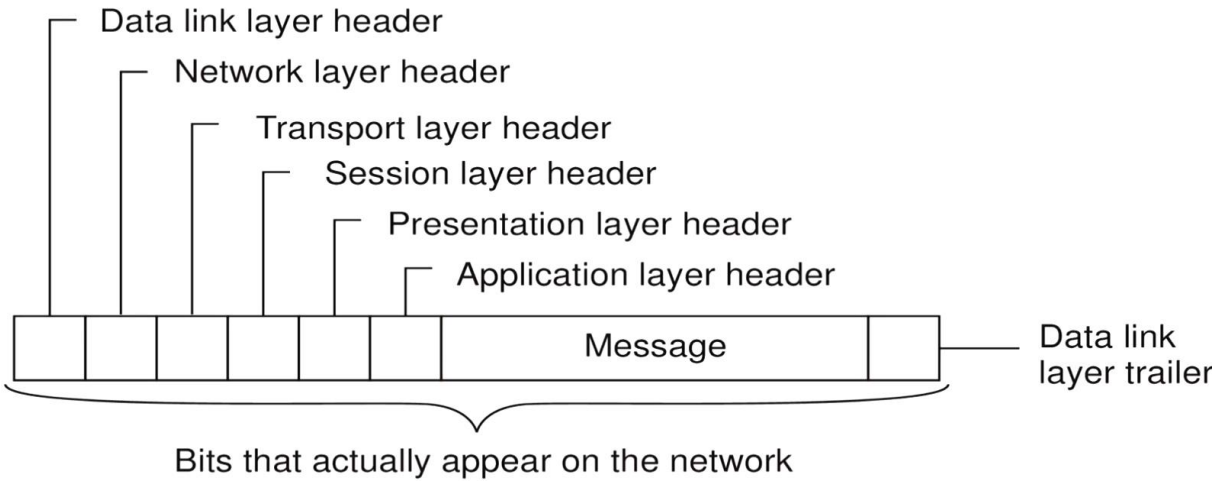
Interprozesskommunikation

- Basiert auf simpler Nachrichtenübertragung, die vom darunterliegenden Netzwerk angeboten wird
- Kommunikationsentitäten: Prozesse
- ISO/OSI Modell



ISO/OSI Modell: Header

- Headers werden zu jeder Schicht hinzugefügt

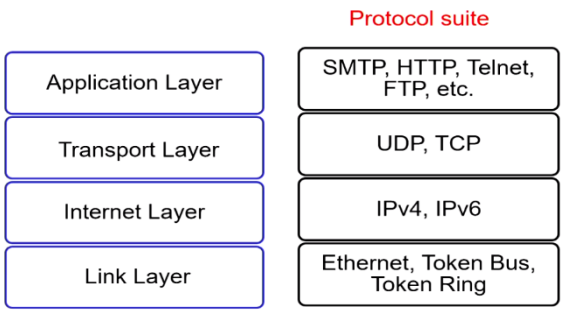


Transportschicht

- Bietet die eigentlichen Kommunikationsmöglichkeiten für die meisten verteilten Systeme
- Standardprotokolle:
  - TCP (Transmission Control Protocol): Verbindungsorientiert, zuverlässig, Stromorientierte Kommunikation
  - UDP (User Datagram Protocol): Unzuverlässige (best-effort) Datagrammkommunikation

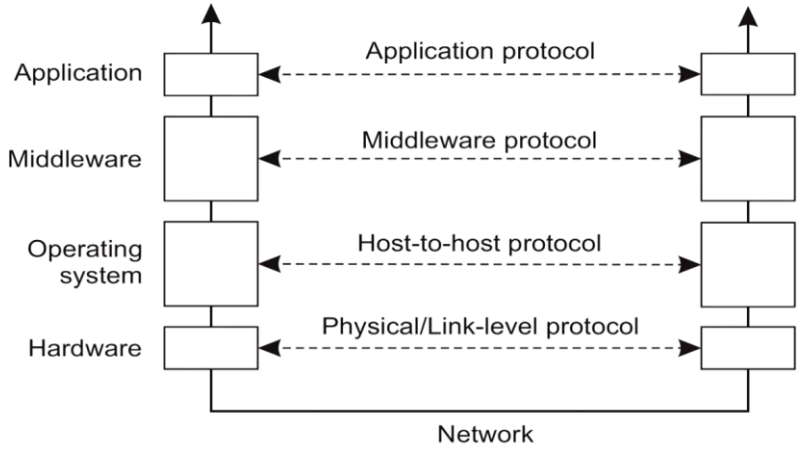
TCP/IP

- Das beliebteste Protokollsystem im Internet
- Vier Schichten



Middleware Schicht

- Was macht man, wenn verschiedene Applikationen gemeinsame Funktionalitäten brauchen (z.B. für Fehlertoleranz, Sicherheit oder Synchronisation)?
- Middleware Schicht:
    - Jede Menge an Kommunikationsprotokollen
    - Ordnen von Daten, notwendig für integrierte Systeme
    - Naming-Protokolle
    - Sicherheitsprotokolle
    - Skaliermechanismen



Kommunikationsarten

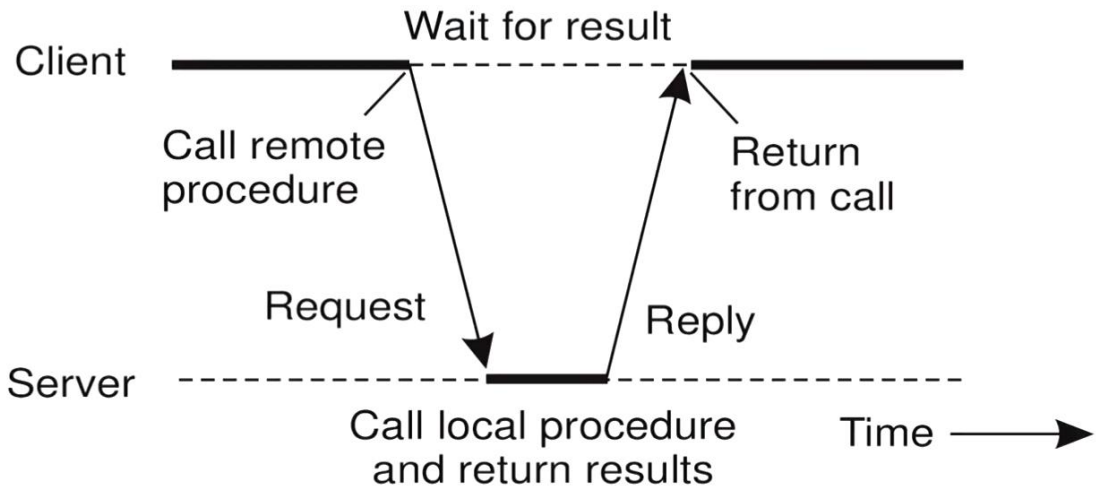
- Transiente vs. Persistente Kommunikation:
    - **Transient**: Kommunizierende Server verwerfen eine Nachricht, falls diese nicht zum nächsten Server oder zum Empfänger geliefert werden kann
    - **Persistent**: Die Nachricht wird, so lange es dauert die Nachricht zu liefern, im kommunizierenden Server gespeichert
- Synchrone und Asynchrone Kommunikation:
    - **Synchron**: Sender ist blockiert bis seine Anfrage wissentlich akzeptiert wurde
    - **Asynchron**: Hat nicht diese Funktionalität (Fire & Forget)

Client/Server

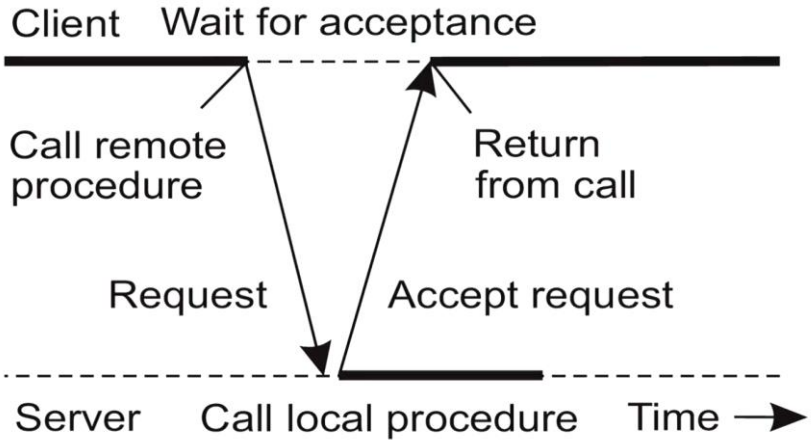
- Normalfall: transient und synchron:
    - Client und Server müssen über den gesamten Zeitraum der Kommunikation aktiv sein
    - Client sendet Anfragen und blockiert bis er eine Antwort erhält
    - Server wartet nur auf eingehende Anfragen und verarbeitet diese daraufhin
- Nachteile von synchroner Kommunikation:
    - Clients müssen auf Antwort warten (und machen nichts anderes in dieser Zeit)
    - Ausfälle müssen direkt behandelt werden: Client wartet

Remote Procedure Calls (RPC): Basics

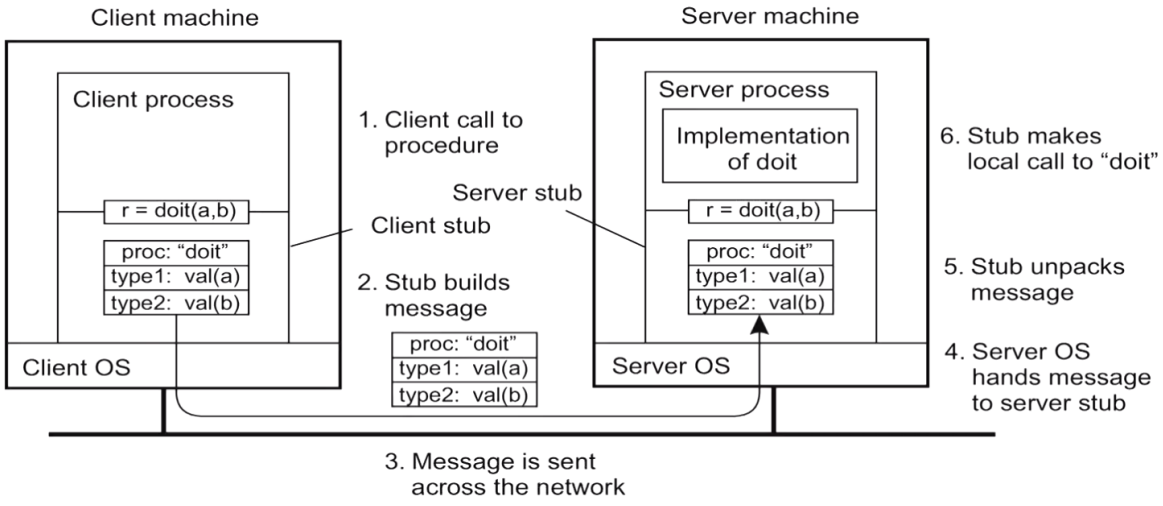
- Ein Prozess ruft eine (entfernte) Prozedur in einem anderen Prozess auf



Asynchroner RPC

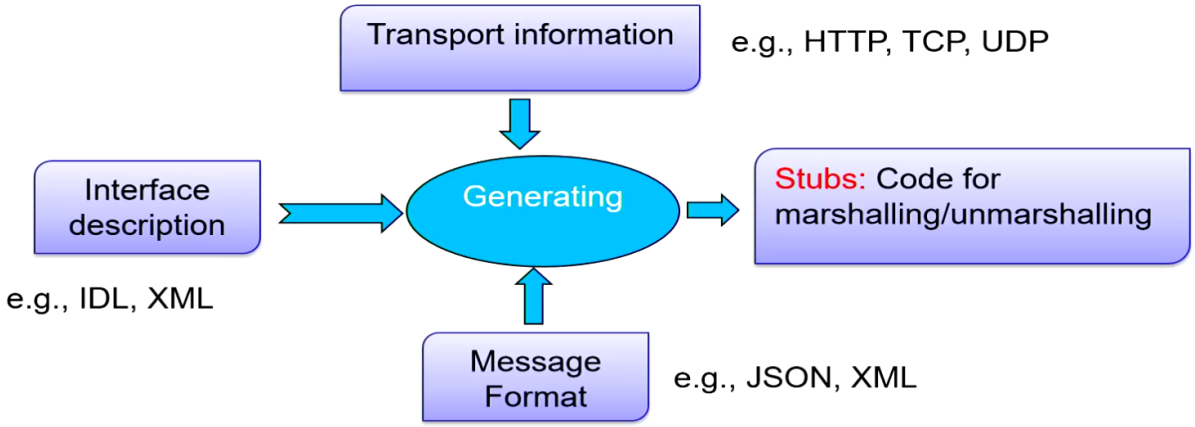


RPC: Interaktionen



Stub: Hilfsprogramm, das Anfrage in richtiges Format umwandelt für Kommunikation

Stubs erstellen

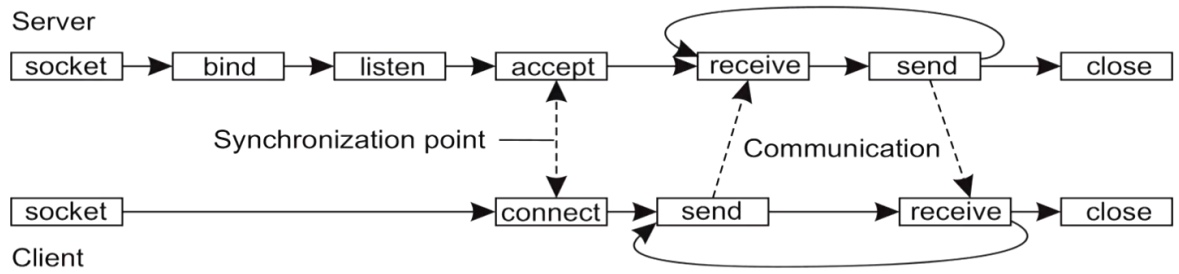


Flüchtige (Transiente) Nachrichtenkommunikation: Sockets

- Socket-Programmierung mittels Socketschnittstellen auf der Transportschicht:
  - Entworfen für low-level Systeme, hohe Performance, ressourcenbeschränkte Kommunikation
- Socketschnittstellen:
  - Sehr beliebt, wird in fast allen Programmiersprachen und Betriebssystemen unterstützt
  - Client: Verbindet sich, sendet und empfängt Daten via Sockets
  - Server: Verknüpft sich, wartet/akzeptiert, empfängt eingehende Daten, verarbeitet diese Daten und sendet ein Ergebnis zurück an den Client

Berkeley Sockets

- Socket: Stellt neuen Kommunikationsendpunkt zur Verfügung
- Bind: Fügt neue lokale Adresse zum Socket hinzu
- Listen: Zeigt Bereitschaft Verbindungen zu akzeptieren
- Connect: Aktiver Versuch eine Verbindung herzustellen
- Accept: Akzeptiert eingehende Anfrage
- Send: Sendet Daten über die Verbindung
- Receive: Empfängt Daten über die Verbindung
- Close: Schließt die Verbindung/den Socket



Nachrichtenorientierte Kommunikation

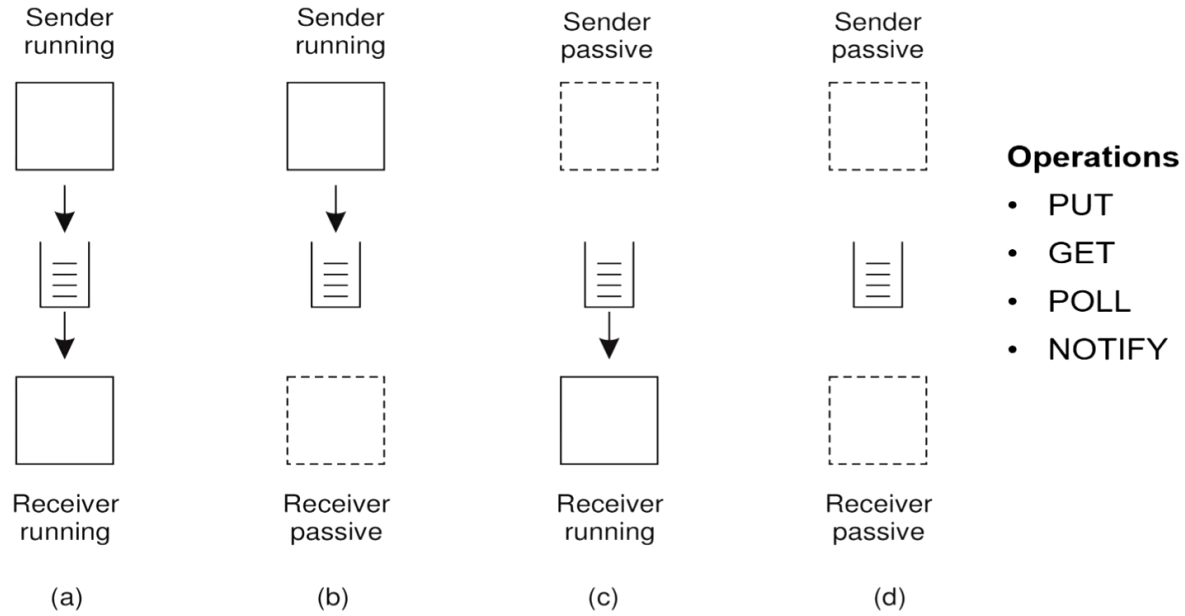
Zielt auf high-level persistente, asynchrone Kommunikation hinaus:

- Prozesse senden sich Nachrichten, welche in eine Warteschlange kommen
- Sender muss nicht auf sofortige Antwort warten und kann andere Sachen machen
- Middleware gewährleistet oft die Fehlertoleranz

Nachrichtenorientiere Middleware (MOM)

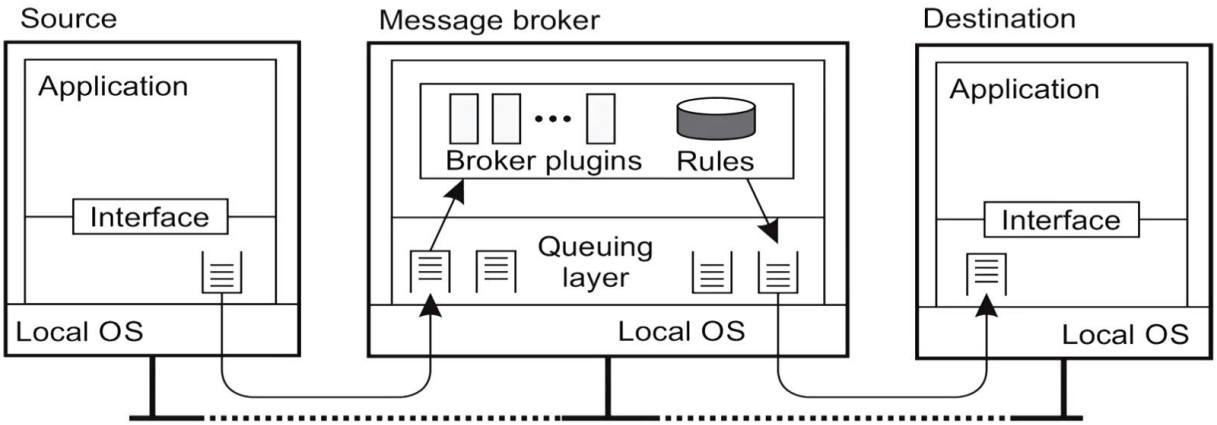
- Message-Queuing System oder Nachrichtenorientiere Middleware
- Unterstützt in großen Systemen für
    - Persistente, aber asynchrone Nachrichten
    - Skalierbarer Umgang mit Nachrichten
    - Verschiedene Kommunikationsmuster

Warteschlangenmodelle



Message Broker

- Zentralisierte Komponente/Knoten
    - Ändert die eingehenden Nachrichten ins Zielformat
    - Weitere Funktionalitäten

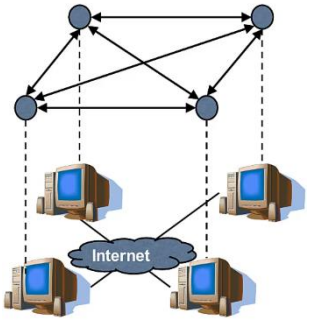


Multicast – Basics

- Multicast = Daten zu mehreren Empfängern senden
- Bspw. in Peer-to-Peer Szenarien
- Verschiedene Herangehensweisen:
    - Multicasting auf Anwendungsebene
    - Flutbasiertes Multicasting
    - Gossipbasierte Datenverbreitung

Multicasting auf Anwendungsebene

- Basiert auf einem Overlay-Netzwerk
    - Zusammengesetzt aus direkten Verbindungen zwischen Peers
    - Normalerweise ein „überliegendes“ Netzwerk auf einem anderen Netzwerk (z.B. das Internet)
    - Aber komplett unabhängig vom physischen Netzwerk aufgrund der Abstraktion der TCP/IP Schicht
    - Separates Adressierungsschema



Flutbasiertes Multicasting

- Multicasting: Senden einer Nachricht an eine *Gruppe von Knoten*
- Fluten:
    - *P* sendet eine Nachricht *m* an jeden seiner Nachbarn
    - Jeder Nachbar leitet *m* weiter, außer zu *P* und auch nur wenn m noch nicht gesehen wurde
- Ziemlich ineffizient…

Epidemisches Verhalten: Gossipbasierte Datenverbreitung

- Peer *P* sendet Nachricht an einige Nachbarn *Q*
- Wenn ein Nachbar Q_i Nachricht schon kennt, hört *P* mit einer Wahrscheinlichkeit von 1/k auf andere Server zu kontaktieren
- Gossiping garantiert nicht, dass jeder Peer informiert wird! Keine letztendliche Konsistenz!

Zusammenfassung

- Wie wird die Kommunikation zwischen Clients und Servern realisiert?
    - ISO/OSI Modell
    - Middleware Kommunikation
    - Remote Procedure Calls
    - Nachrichtenorientierte Middleware
    - Multicast Kommunikation

Benennung

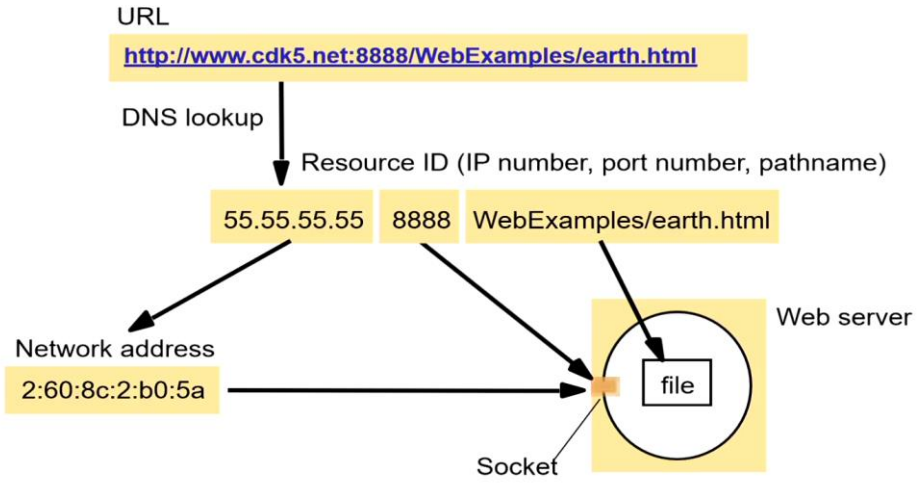
Benennung – Grundlegendes Konzept

- **Entität**: Jede Art von Objekt das wir in verteilten Systemen sehen: Prozess, Datei, Drucker, Host, Webseite, Kommunikationsendpunkt, etc.
- Wie interagiere ich mit einer Entität?
    - Wir müssen es benennen: Die Entität braucht einen **Namen**
    - Wir müssen darauf zugreifen: Die Entität braucht einen **Zugriffspunkt**
    - Zugriffspunkte sind Entitäten, die mittels einer **Adresse** benannt werden

Namen, Identifikatoren und Adressen

- (Reiner) **Name**: Abfolge von Bits/Charactern, die dazu benutzt werden, eine Entität oder eine Sammlung an Entitäten innerhalb eines bestimmten Kontextes zu identifizieren
    - Ein Name hat keine Bedeutung, es ist nur ein zufälliger String
- **Identifikator**: Ein Name der eine Entität *eindeutig* identifiziert
    - Der Identifikator ist eindeutig und bezieht sich auf nur eine Entität
    - Jede Entität wird bezeichnet von mindestens einem Identifikator
- **Adresse**: Der Name eines **Zugriffspunkts**, der Ort einer Entität

Beziehungen zwischen Namen/Identifikatoren



Flache Benennung

Unstrukturierte/Flache Namen: Identifikatoren haben keine strukturierte Beschreibung, bspw. nur eine Abfolge an Bits

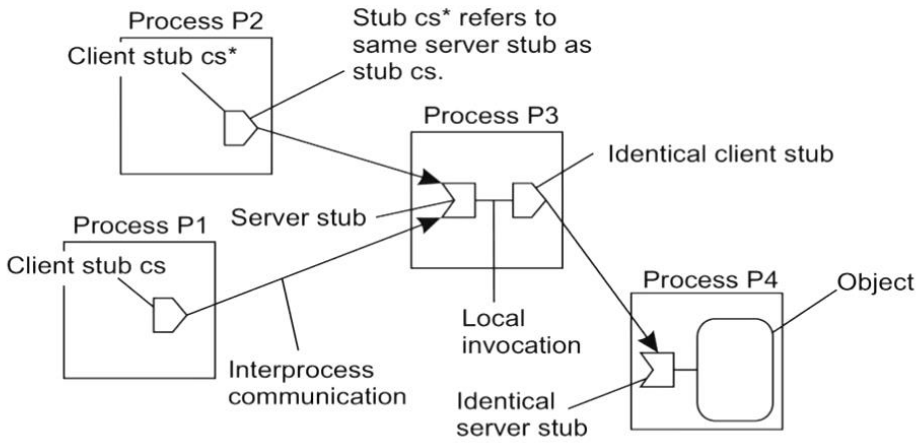
- Einfacher Weg, um Identifikatoren zu repräsentieren
- Beinhalten keine zusätzliche Information, um die Entität zu verstehen
- Beispiele
    - Internetadressen auf der Sicherungsschicht (MAC-Adresse), z.B. 00:14:a5:41:61:6b
    - Nummern in verteilten Hashtabellen, z.B. 0100

Broadcasting

- Identifikators einer Entität broadcasten
- Knoten schauen, ob sie diese Entität anbieten und geben die Adresse zurück
- Z.B. *Address Resolution Protocol (ARP)*

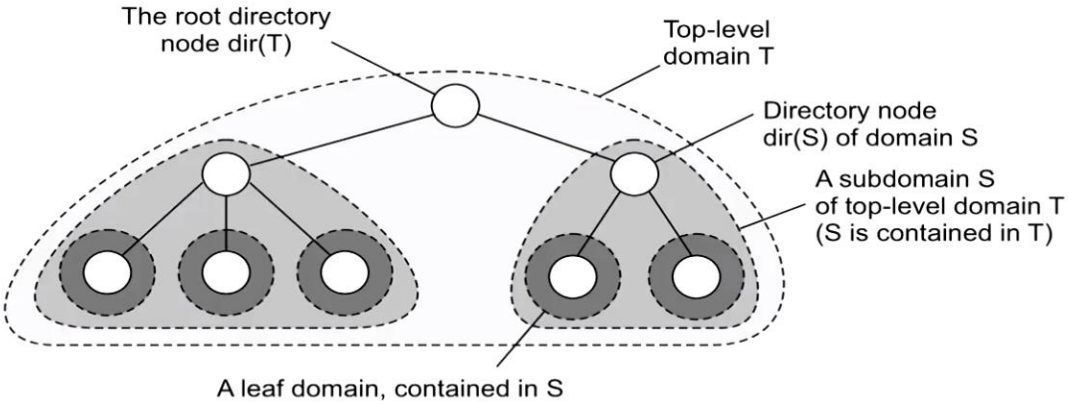
Vorwärtszeiger

- Wenn eine Entität umzieht, hinterlässt sie einen Zeiger zum nächsten Ort
- Dereferenzierung wird transparent für Clients, indem einfach der Zeigerkette gefolgt wird

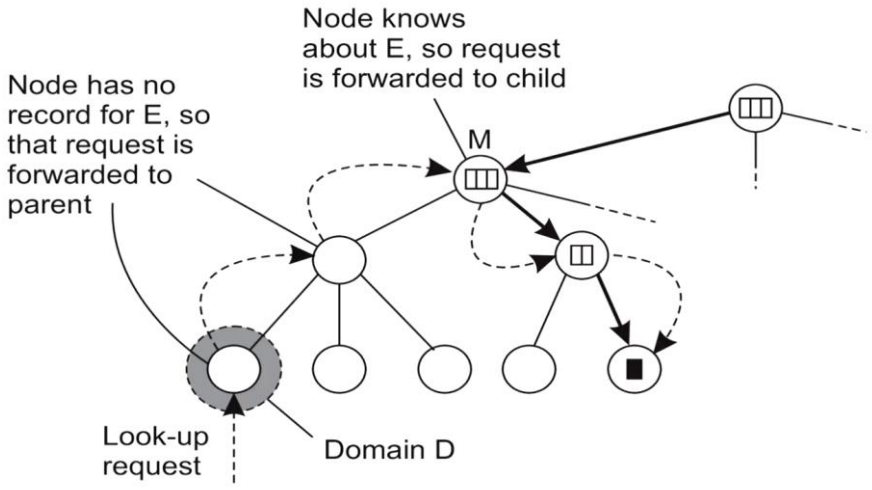


Hierarchische Lokationsdienste (HLS)

- Grundidee: Erstellen eines umfangreichen Suchbaums, für den das zugrunde liegende Netzwerk in hierarchische Domänen aufgeteilt ist
- Jede Domäne wird von einem separaten Verzeichnisknoten repräsentiert



HLS: Suchvorgang

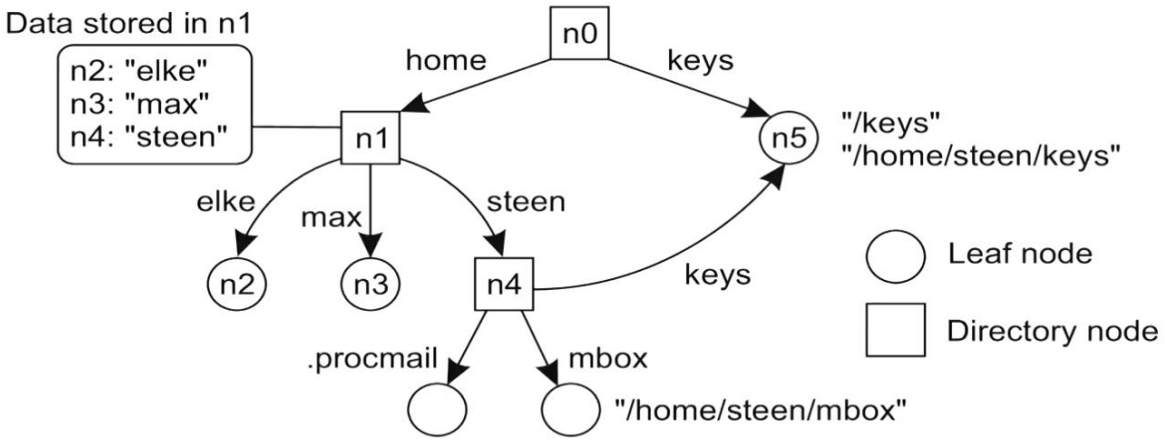


Benennung Designprinzipien

- **Namensraum**
  - Beinhaltet alle validen Namen, die innerhalb eines Systems anerkannt und gemanaged werden können
    - Ein valider Name muss nicht unbedingt zu einer Entität zugeordnet sein
    - Alias: Ein Name verweist auf einen anderen Namen
- **Namensdomäne**
  - Namensraum mit einer einzelnen administrativen Autorität, welche die Namen für den Namensraum verwaltet
- **Namensauflösung**
  - Ein Vorgehen, um Informationen/Attribute basierend auf einem Namen aufzurufen

Namensräume

- Namen werden in Namensräumen organisiert, welche als Graphen modelliert werden können:
  - Blattknoten vs. Verzeichnisknoten
- Jeder Blattknoten repräsentiert eine Entität; Knoten sind auch Entitäten

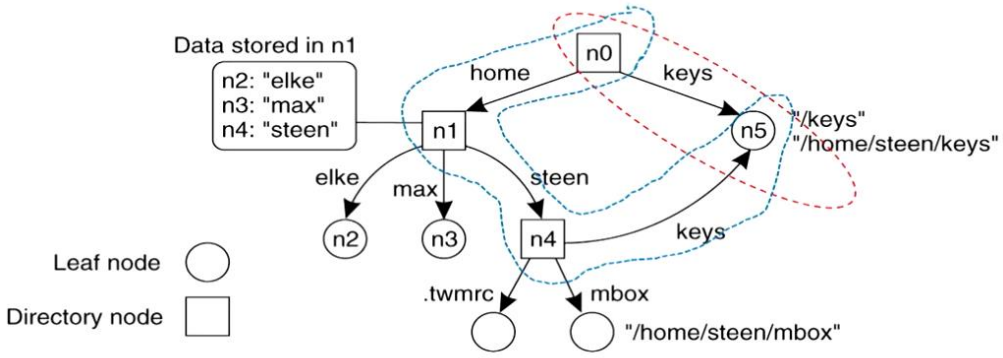


Namensauflösung – Schließmechanismus

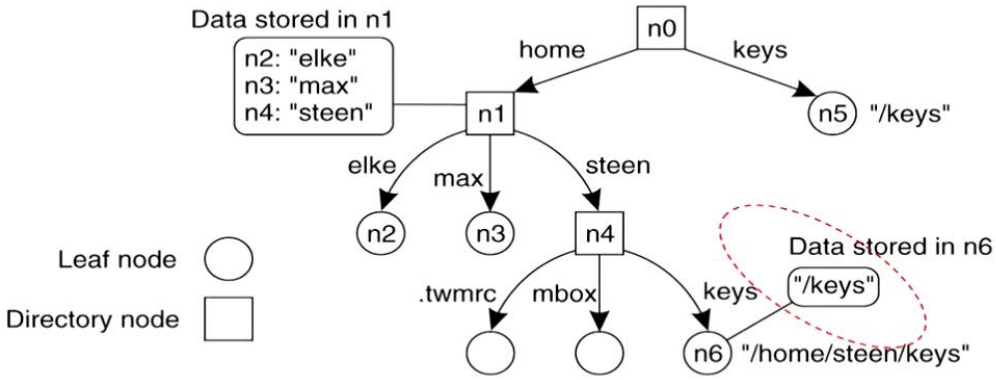
- Namensauflösung: Pfad p:
  <label1, label2, label3, …, labelp>
  - Start bei Knoten N
  - Suche (label1, identifikator1) in N's Verzeichnistabelle
  - Suche (lavel2, identifikator2) in identifikator1's Verzeichnistabelle
  - Usw.
- Schließmechanismus: Herausfinden, wo und wie die Namensauflösung gestartet wird
  - Z.B.: Namensauflösung für /home/dustdat/ds.txt
  - Oder für https://me.yahoo.com/a/

Aliase erlauben mittels Links

Hard Links: Mehrere absolute Pfadnamen verweisen auf denselben Knoten

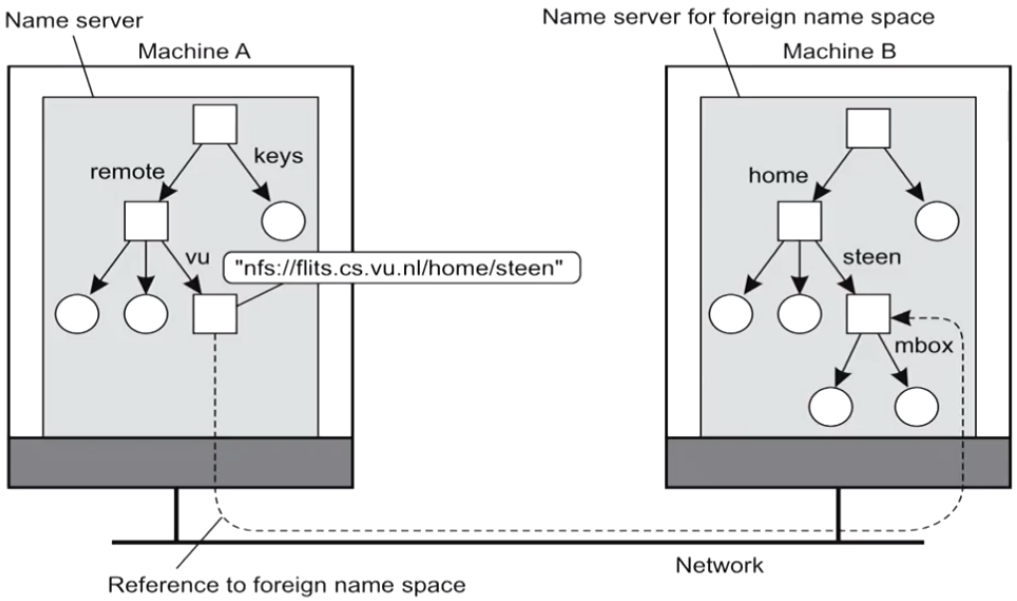


Soft/Symbolische Links: Blattknoten speichert den absoluten Pfadnamen



Namensauflösung – Mounting

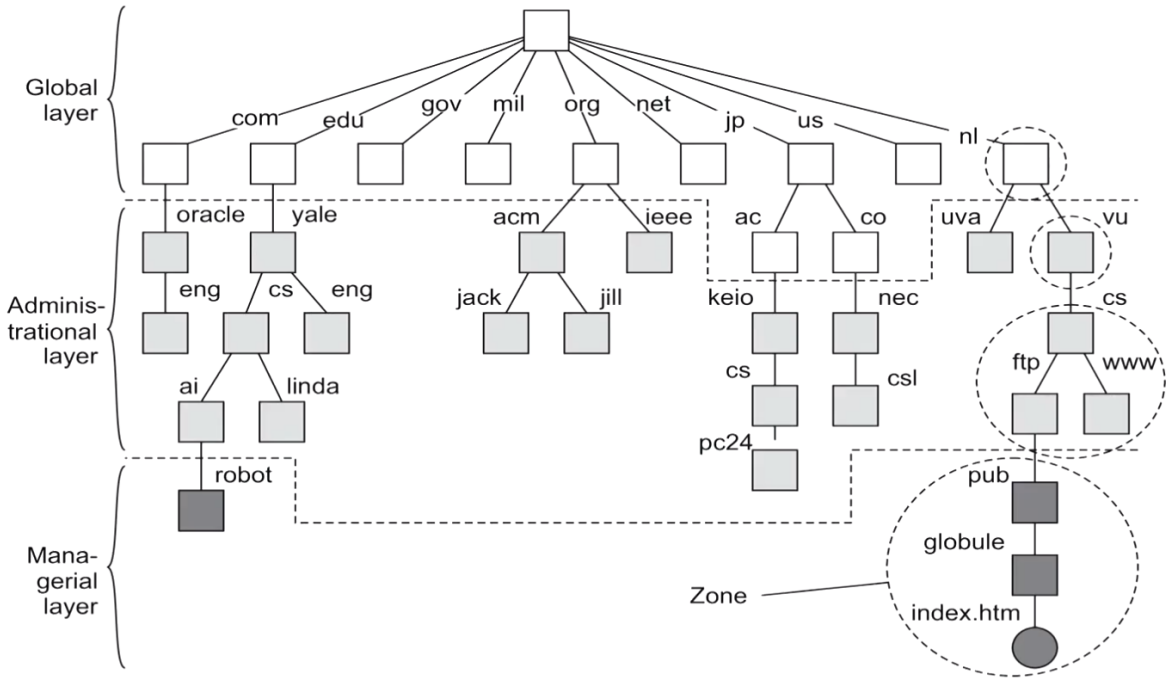
Ein Verzeichnisknoten (Mountingpunkt) in einem entfernten Server kann in einen lokalen Knoten (Mountingpunkt) gemounted werden



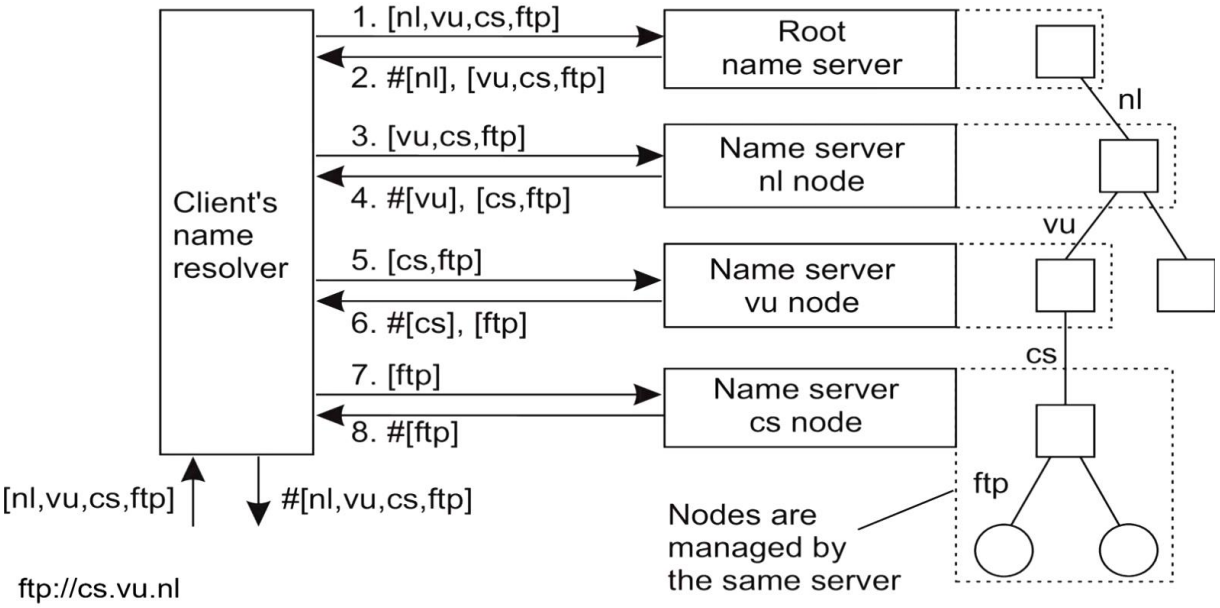
Namensraum Implementation

- Verteilte Namensverwaltung
  - Mehrere Server werden für die Verwaltung von Namen benutzt
- Viele Verteilungsschichten
  - **Globale Schicht**:
    - Der Root-Knoten und dessen nächste Knoten
    - Gemeinsam verwaltet von mehreren Administrationen
  - **Administrative Schicht**:
    - Verzeichnisknoten, die von einem einzelnen Unternehmen verwaltet werden
    - „Mid-Level“ Verzeichnisknoten
  - **Managementschicht**:
    - Knoten verändern sich häufig
    - „Low-Level“ Verzeichnisknoten

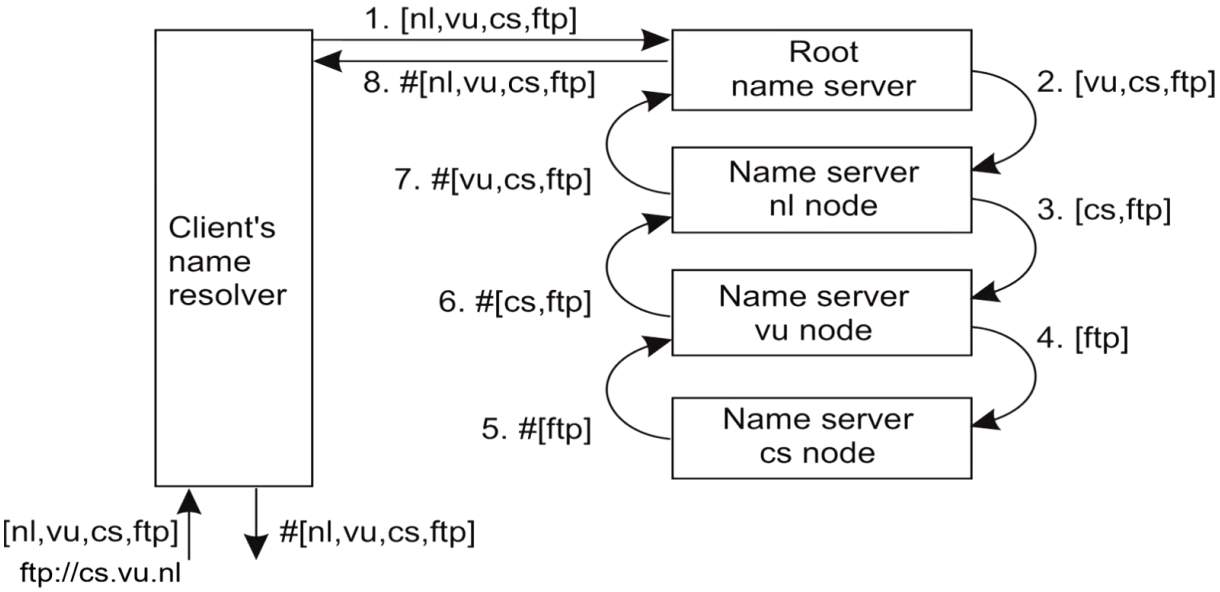
Beispiel in einem Domain Name System



Iterative Namensauflösung



Rekursive Namensauflösung



Nachteil rekursiv:

- Weitaus höherer Aufwand auf der Serverseite

Vorteile rekursiv:

- Serverseitiges Caching
- Kommunikationswege

Das Domain Name System (DNS)

- Hierarchisch organisierter Namensraum, wobei jeder Knoten genau eine eingehende Kante hat
- Domäne: Ein Teilbaum
- Domänenname: Ein Pfadname zu einem Root-Knoten einer Domäne
- String-Repräsentation eines Pfadnamens:
    - Beschriftungen auflisten
    - Beschriftungen mit einem Punkt trennen
    - Root wird mit einem Punkt dargestellt
    - Beispiel: flits.cs.vu.nl.

DNS-Abfragen

- **Einfache Host-Namensauflösung**
    - Was ist die IP von www.tuwien.ac.at?
- **E-Mail Server Namensauflösung**
    - Was ist der E-Mail Server von dustdat@dsg.tuwien.ac.at
- **Rückwärtsauflösung**
    - Von IP zu Hostname
- **Hostinformationen**
- **Andere Dienste**

Attribute/Werte

- Ein Tupel (Attribut, Wert) kann benutzt werden, um ein Objekt zu beschreiben
    - Z.B.: („country“, „Austria“), („language“, „German“)
- Eine Reihe an Tupeln (Attribut, Wert) kann benutzt werden, um eine Entität zu beschreiben

AustriaInfo

Attribute	Value
CountryName	Austria
Language	German
MemberofEU	Yes
Capital	Vienna

Attributbasierte Naming Systeme

- Verwenden von (Attribut, Wert) Tupeln, um Entitäten zu beschreiben
- Auch Verzeichnisdienste genannt
- Namensauflösung
    - Normalerweise basierend auf Abfragemechanismen
    - Das Abfragen benutzt meistens den gesamten Raum
- Beispielimplementationen
    - LDAP (Lightweight Directory Access Protocol)

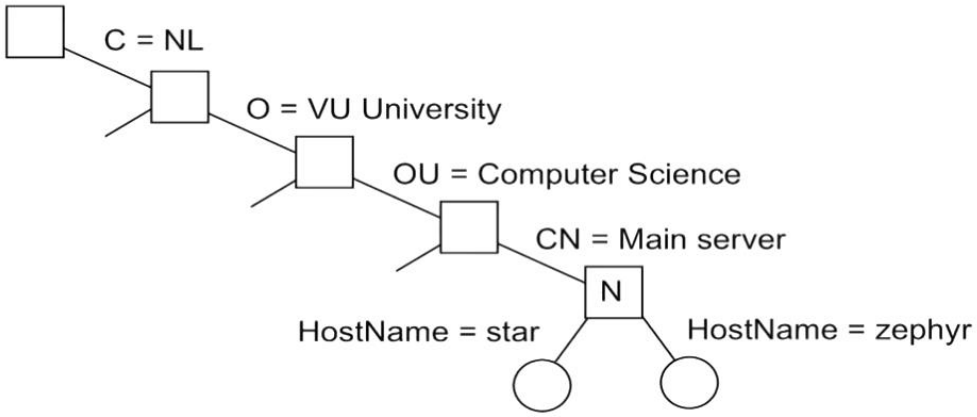
LDAP – Lightweight Directory Access Protocol

- Verzeichnisinformationsgrundlage: Sammlung aller Verzeichniseinträge in einem LDAP Dienst

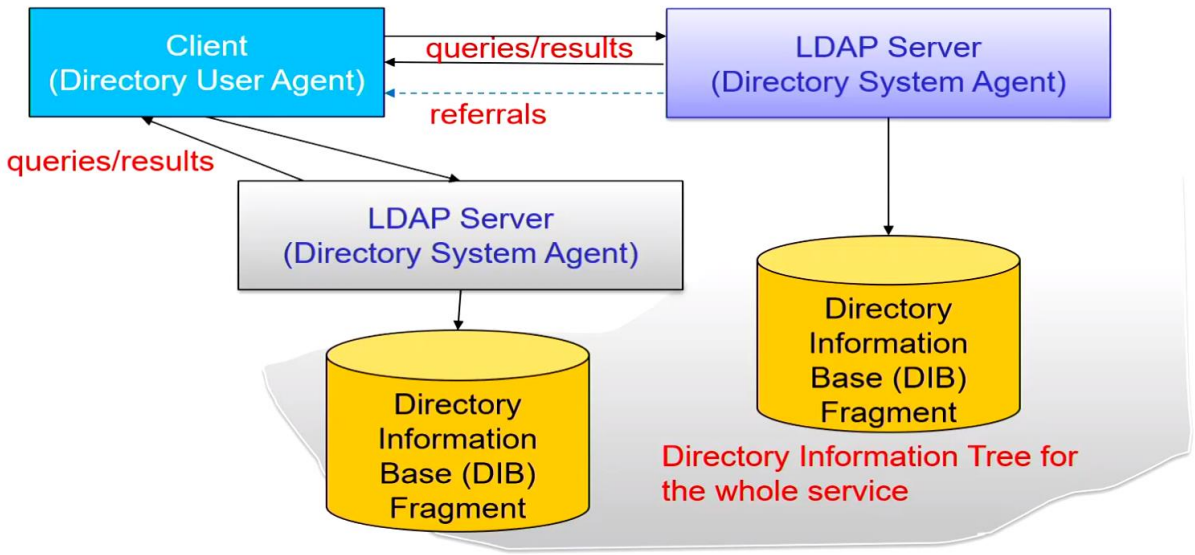
Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	—	130.37.20.20
WWW_Server	—	130.37.20.20

LDAP Verzeichnisinformationsbaum

Verzeichnisinformationsbaum: Der Benennungsgraph eines LDAP Verzeichnisdienstes; Jeder Knoten stellt ein Verzeichniseintrag dar



LDAP- Client-Server Protokoll



Zusammenfassung

- Flache Benennung
  - Broadcasting
  - Vorwärtszeiger
  - Hierarchischer Suchbaum
- Strukturierte Benennung
  - Einfacher zu verstehen für menschliche Nutzer
  - Pfadnamen nutzbar, um Adresse gut aufzulösen
- Attributbasierte Benennung
  - Suche nach Attributen statt „nur“ Namen

Fehlertoleranz

Der Fall von Ariane 5

Video: https://youtu.be/PK_yguLapgA?t=71

- Kontrollsoftware von Ariane 4 verwendet, ohne diese zu testen
  - Kosten: 290 Millionen Euro
- Ein „einfaches“ Problem mit Gleitkommazahlen hat das Problem verursacht
- Software war sogar überflüssig:
  - Beide Systeme hatten trotzdem den gleichen Softwarebug

Zuverlässigkeit

- Grundlagen: In Verteilten Systemen stellen *Komponenten Dienste* für *Clients* zur Verfügung
  - Um Services bereitzustellen, benötigt die Komponente evtl. Dienste von anderen Komponenten
  - Das heißt: Es ist *abhängig* von anderen Komponenten
  - Genauer: Die *Korrektheit* der Komponente hängt von der *Korrektheit* einer anderen Komponente ab
- *Abhängigkeit* ist deshalb ein Kernziel in verteilten Systemen: Clients brauchen korrekte Dienste in Bezug auf funktionale und nicht-funktionale Eigenschaften

Zuverlässigkeit: Attribute

Abhängigkeit	**Sicherheit**
Verfügbarkeit	
Verlässlichkeit	
Sicherheit	
	Vertraulichkeit
Integrität	
Wartbarkeit	

- Verfügbarkeit: Sofortige Bereitschaft für korrekte Dienste
- Verlässlichkeit: Beständigkeit eines korrekten Dienstes
- Sicherheit: Abwesenheit von katastrophalen Konsequenzen
- Integrität: Abwesenheit unzulässiger Systemveränderungen
- Wartbarkeit: Möglichkeit Änderungen vorzunehmen

Gefahren für die Zuverlässigkeit

- **Fehlschlag (Failure)**: Gelieferter Dienst weicht von korrektem Dienst ab, d.h. die Systemfunktionalität ist nicht mehr gegeben
- **Störung (Error)**: Abweichung des tatsächlichen Systems vom wahrgenommenen
- **Mangel (Fault)**: Grund für die Störung

Mangel → Störung → Fehlschlag
Fault → Error → Failure

Beispiel 1: Fehlschläge, Störungen, Mängel

- **Mangel**: Softwarebug in einer bestimmten Methode
  (bis jetzt ist der Mangel *schlafend*: Solange niemand die Methode aufruft, wird er nicht *aktiv*)
- **Störung**: Der Prozess, der die Methode anbietet, wird aufgerufen
  (Mangel wird *aktiv*), führt zur Berechnung eines falschen Wertes
- **Fehlschlag**: Wenn es keinen Mechanismus zur Identifikation der Störung gibt, wird die Komponente, die den Prozess aufruft, einen falschen Wert zurückbekommen

Beispiel 2: Fehlschläge, Störungen, Mängel

- **Mangel**: Fehlerhafter USB-Port einer externen Festplatte
  (Solange man die Festplatte nicht benutzt: Mangel ist schlafend, Computer funktioniert noch)
- **Störung**: I/O Vorgang gestartet; Bitfehler tauchen auf
- **Fehlschlag**: Es ist nicht möglich Dateien korrekt von/zu der externen Festplatte zu kopieren

Fehlschlagmodelle

- Crashfehlschlag: Komponente stoppt, funktioniert aber noch perfekt bis zu diesem Moment
- Versäumnisfehlschlag (Omission): Komponente kann nicht antworten:
  - Empfangsversäumnis: Kann eingehende Nachrichten nicht empfangen
  - Sendeversäumnis: Kann Nachrichten senden
- Timingfehlschlag: Antwort liegt außerhalb des bestimmten Zeitintervalls
- Antwortfehlschlag: Antwort ist inkorrekt:
  - Wertfehlschlag: Der Wert der Antwort ist falsch
  - Zustandsübergangsfehlschlag: Weicht von richtigem Kontrollfluss ab
- Beliebiger (Byzantinischer) Fehlschlag: Tritt auf oder auch nicht; Beliebiger Fehlschlag zu beliebigen Zeiten

Was macht man gegen Fehler?

- Fehlerprävention:
  - Vorkommen eines Mangels verhindern
- Fehlervorhersage
  - Annahme von gegenwärtigen und zukünftigen Mängeln und deren Konsequenzen
- **Fehlertoleranz**:
  - **Verhindern, dass Dienstfehlschläge aufgrund von Mängeln auftreten, d.h. *maskieren* der Präsenz von Fehlern**
  - **Bereitstellung des Dienstes wird fortgesetzt!**
- Fehlerentfernung:
  - Verringern der Anzahl und Schwere der Fehler

Ansatz zu Fehlertoleranz

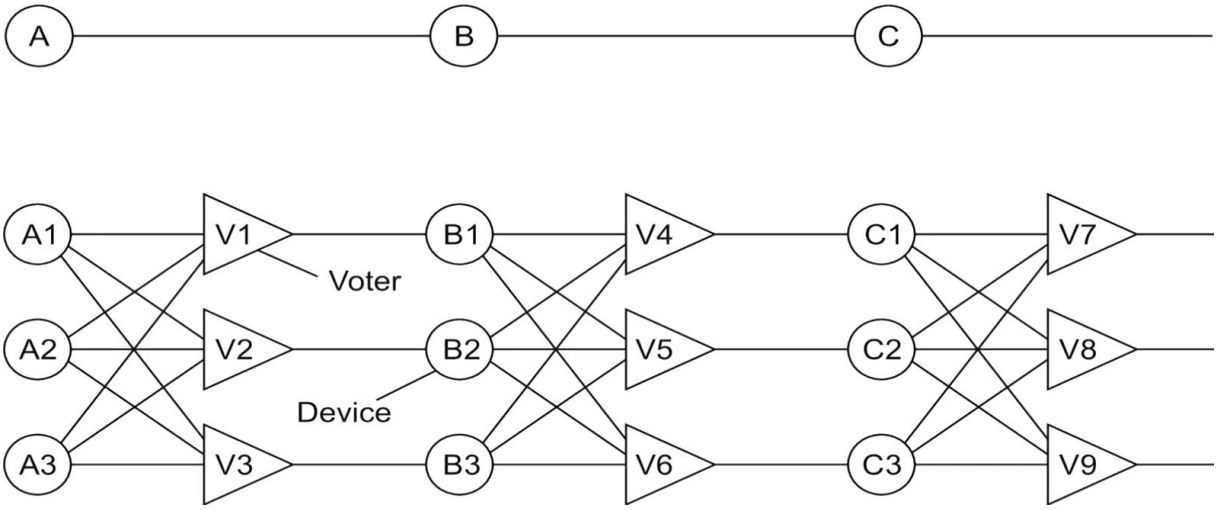
- „Keine Fehlertoleranz ohne Redundanz“ (Gärtner, 1999)
    - Redundanz benutzen, um Fehler zu maskieren, d.h. Vorkommen von Fehler verstecken
- Fehlermaskierung durch Redundanz
    - Informationsredundanz: Zusätzliche Informationen hinzufügen
    - Zeitredundanz: Anfrage wiederholen
    - Physische Redundanz: Zusätzliche Komponenten hinzufügen

Redundanz – Beispiele

- Informationsredundanz:
    - Paritätsbit hinzufügen
    - Error Correcting Code (Speicher)
- Zeitredundanz:
    - Neuübertragungen in TCP/IP
- Physikalische Redundanz:
    - Backup Server
    - Festplatten in einem RAID 1
    - Aber auch: Verschiedene Implementationen von der gleichen Funktionalität in verschiedenen Prozessen

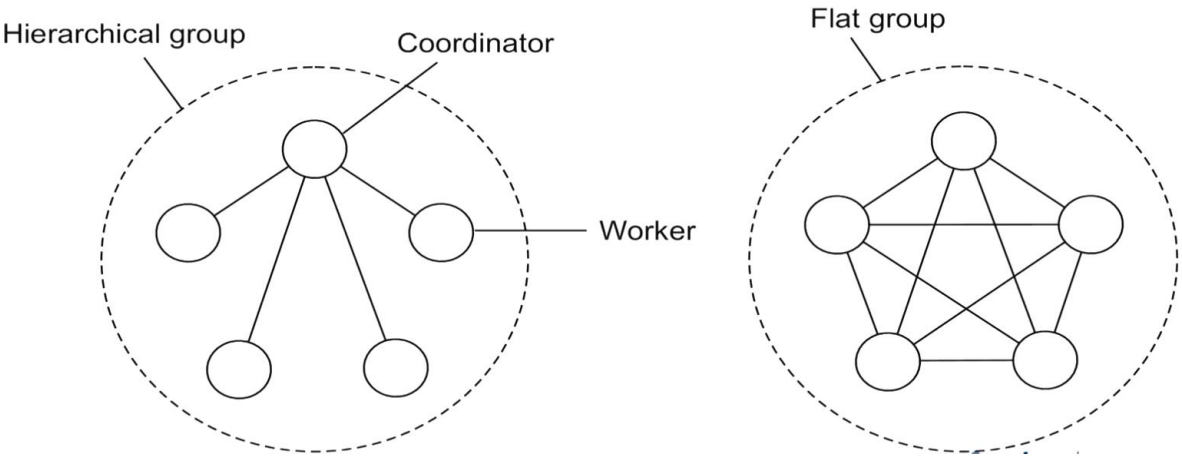
Physische Redundanz – Beispiel

Elektrischer Kreislauf mit dreifach modularer Redundanz:



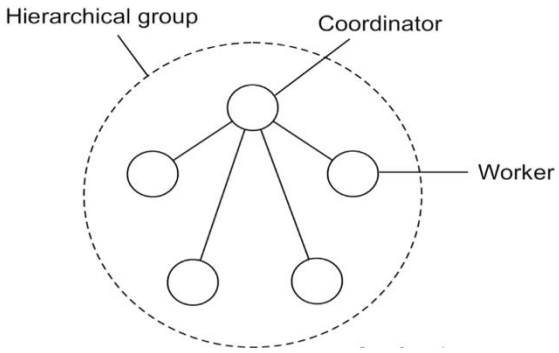
Prozesswiderstandsfähigkeit – Grundlagen

- Wie toleriert man fehlerhafte Prozesse?
    - „Keine Fehlertoleranz ohne Redundanz“
        - Mehrere identische Prozesse in eine Gruppe einordnen



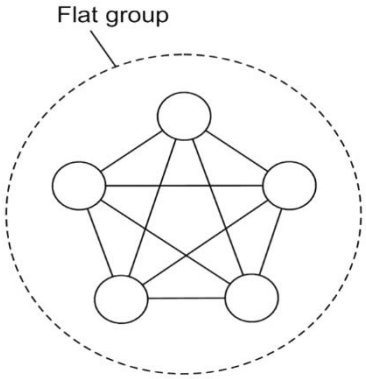
Kommunikation in Hierarchischen Gruppen

- Hierarchische Gruppen:
    - Kommunikation mittels eines einzigen Koordinators
    - Nicht fehlertolerant oder skalierbar
    - Aber einfacher zu implementieren



Kommunikation in Flachen Gruppen

- Flache Gruppen:
    - Gut im Hinblick auf Fehlertoleranz, da der Informationsaustausch sofort mit allen Gruppenmitgliedern stattfindet
    - Overhead kann entstehen, da die Kontrolle komplett verteilt ist und eine Abstimmung muss durchgeführt werden
    - Schwieriger zu implementieren

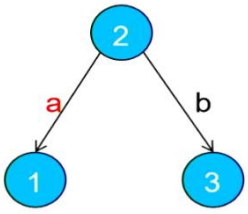


Gruppen und Fehlermaskierung

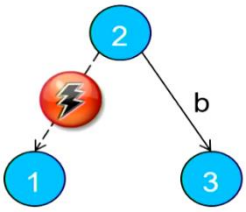
- *k*-Fehler Toleranzgruppe:
    - Gruppe ist in der Lage, gleichzeitig *k* Fehler zu maskieren
- Annahmen: Alle Mitglieder sind ident und verarbeiten alle Eingaben in derselben Reihenfolge
- Wie groß muss eine k-Fehler Toleranzgruppe sein?
- Crash-/Omission-/Timingfehlschlagmodelle (d.h. Komponenten antworten nicht mehr: k+1 sind notwendig
- Beliebiger/Byzantinisches Fehlschlagmodell: 2k+1 Komponenten sind notwendig

Zuverlässige Client-Server Kommunikation

- Bis jetzt: Prozesswiderstandsfähigkeit
- Aber was ist mit zuverlässigen Kommunikationskanälen?



Process 2 tells different things



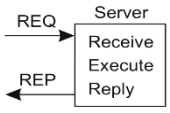
Connection between Process 2 and Process 1 fails

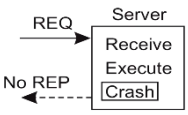
Remote Procedure Calls: Was kann schiefgehen?

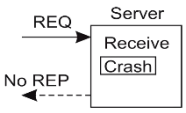
1. Client kann den Server nicht finden
2. Clientanfrage geht verloren
3. Server crasht nach Empfang einer Nachricht
4. Antwortnachricht vom Server geht verloren
5. Client crasht (nach Senden einer Anfrage)

Remote Procedure Calls: Lösungen

1. Client kann den Server nicht finden
   → Einfach dem Client Bescheid geben
   → Client muss sich darum kümmern (z.B. Exception Handling)
2. Clientanfrage geht verloren
   → Anfragenachricht nochmal senden (mittels einer Art von Timer)
   → Server muss den Unterschied zwischen Original und Neuübertragung wissen
3. Server crasht
   a. Normalfall
   b. Crash *nach* Ausführung
   c. Crash *vor* Ausführung







→ Client sieht nicht den Unterschied zwischen Crash vor oder nach Ausführung
→ Korrektes Verhalten des Clients kommt auf Verhalten des Servers an

1. *Mindestens*-Einmal-Semantik: Der Server garantiert die Durchführung einer Operation *mindestens* einmal, egal was passiert
2. *Maximal*-Einmal-Semantik: Der Server garantiert die Durchführung einer Operation *maximal* einmal

→ Und der Client? (wenn keine Antwort, aber eine Nachricht, dass der Server neustartet, empfangen wird

1. *Immer* die Anfrage nochmal senden
2. *Nie* die Anfrage nochmal senden
3. Nur dann die Anfrage nochmal senden, wenn der Server ein ACK empfangen hat

→ 8 Mögliche Kombinationen von Strategien
→ Beispiel ein Client sendet eine Druckanfrage and einen Druckserver
Drei Ereignisse können am Server passieren:
(M) Senden der Fertigstellungsnachricht (REP)
(P) Text drucken
(C) Crash
→ Es gibt keine Kombination von Server und Client Strategien, die bei allen möglichen Ereignissequenzen richtig funktionieren

→ Diese Ereignisse können in sechs verschiedenen Sequenzen auftauchen:

1. M → P → C: Nach Senden der Fertigstellungsnachricht und Drucken des Texts taucht ein Crash auf
2. M → C (→ P): Nach Senden der Fertigstellungsnachricht, aber bevor der Text gedruckt konnte, taucht ein Crash auf
3. P → M → C: Nach Drucken des Texts und Senden der Fertigstellungsnachricht taucht ein Crash auf
4. P → C (→ M): Der Text wird gedruckt und ein Crash taucht auf, wodurch die Fertigstellungsnachricht nicht gesendet wird
5. C (→ P → M): Ein Crash taucht auf bevor der Server irgendwas machen konnte
6. C (→ M → P): Ein Crash taucht auf bevor der Server irgendwas machen konnte

Reissue strategy	Strategy M → P			Strategy P → M		
	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	**Server**			**Server**		

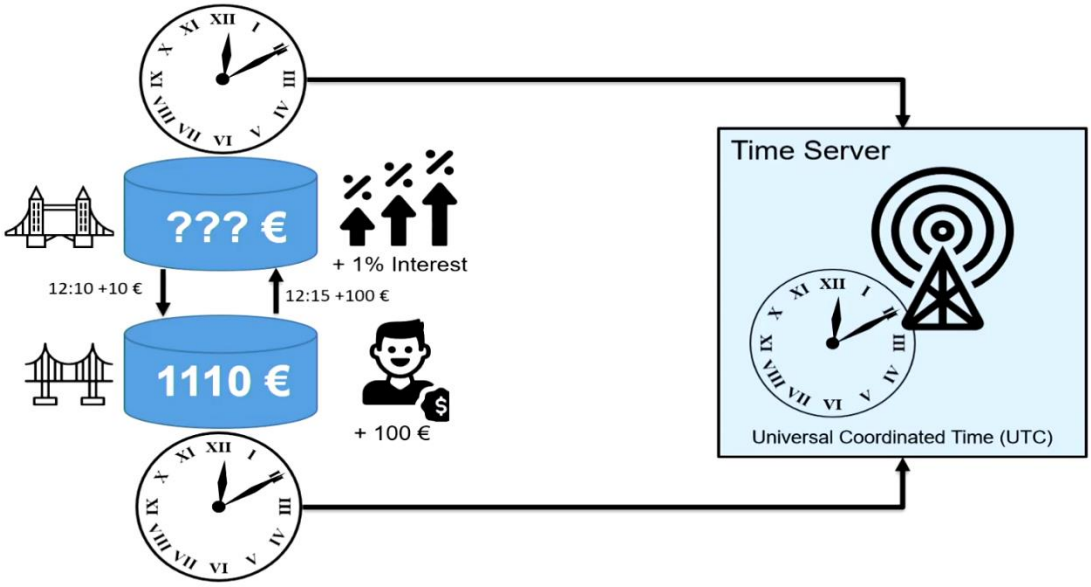
4. Serverantwort geht verloren
   → Woher wissen wir, dass der Server nicht gecrasht ist?
   → Wieder mal: Hat der Server die Operation durchgeführt?
   → Anfrage wiederholen:
   Im Falle von realen Auswirkungen? Überweisung deines Bankkontos zweimal durchgeführt?
   → Keine wirkliche Lösung! Außer Operationen *idempotent* machen (Wiederholbar ohne irgendeinen Schaden)
5. Client crasht (nachdem Anfrage gesendet wurde)
   → Server führt Anfrage trotzdem aus und sendet eine Antwort (auch Waisenberechnung genannt)
   → Verschiedene Lösungen:
   1. Waise wird vom Client gelöscht, falls es empfangen wurde
   2. Reinkarnation: Client sagt Servern, dass er neugestartet hat; Server löscht Waisen
   3. Ablauf: Berechnungen müssen in gewisser Zeit ausgeführt werden

Zusammenfassung

- Generelle Einführung zu Fehlertoleranz
- Widerstandsfähige Prozesse: Gruppen
- Zuverlässige Client-Server Kommunikation

Synchronisierung und Koordination

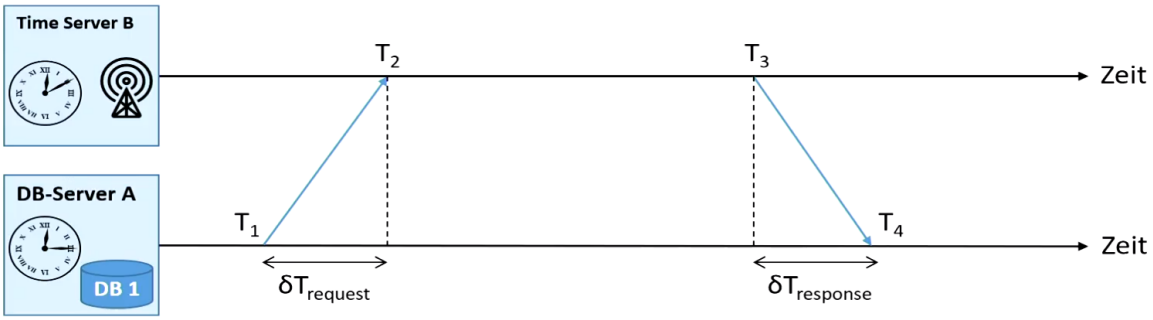
Motivation – Uhrensynchronisierung



Physikalische Uhren

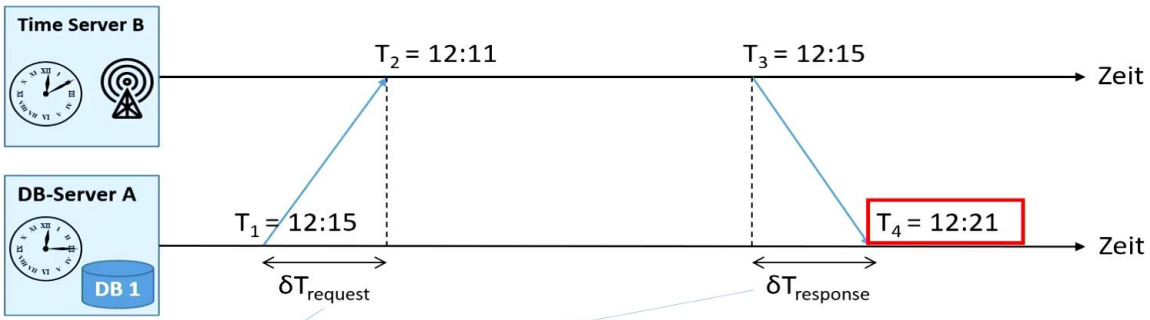
- Die interne Uhr deines Computers basiert auf einem Quarzkristall
- Verschiedene Taktwerte: Zeitversatz
- Lösungsansatz: Synchronisieren der lokalen Uhren mit Hilfe der Universal Coordinated Time (UTC)
- Ziele:
    - Präzision: Abweichungen in einem bestimmten Bereich zwischen Maschinen in einem verteilten System behalten (interne Synchronisierung)
    - Genauigkeit: Abweichungen mit einem Referenzwert in einem bestimmten Bereich behalten (externe Synchronisierung)

Network Time Protocol (NTP)



$$\delta T_{request} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{response}$$

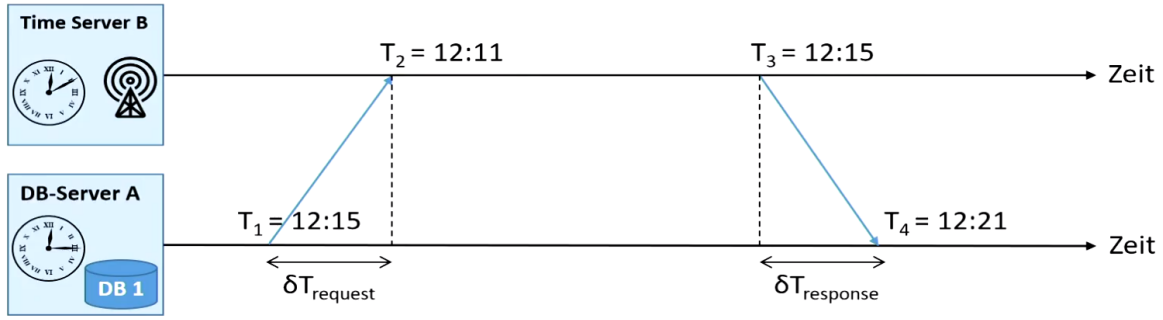
NTP: δ Berechnen



$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$
$$= \frac{(12{:}11 - 12{:}15) + (12{:}21 - 12{:}15)}{2}$$
$$= \frac{-4 + 6}{2} = 1$$

T_1	T_2	T_3	T_4	δ
12:15	12:11	12:15	*12:30*	5.5
12:15	12:11	12:15	*12:21*	1

NTP: Θ Berechnen



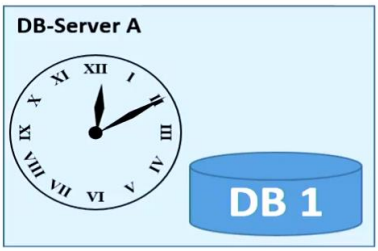
$$\theta = T_3 + \delta - T_4 = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4$$
$$= \frac{2 * T_3}{2} + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - \frac{2 * T_4}{2} = \frac{2 * T_3 + (T_2 - T_1) + (T_4 - T_3) - 2 * T_4}{2}$$
$$= \frac{T_3 + T_2 - T_1 - T_4}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$
$$= \frac{(12{:}11 - 12{:}15) + (12{:}15 - 12{:}21)}{2} = \frac{-10}{2} = -5$$

Merken:

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2} \qquad \theta = T_3 + \delta - T_4 \quad = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

NTP: Zeit anpassen

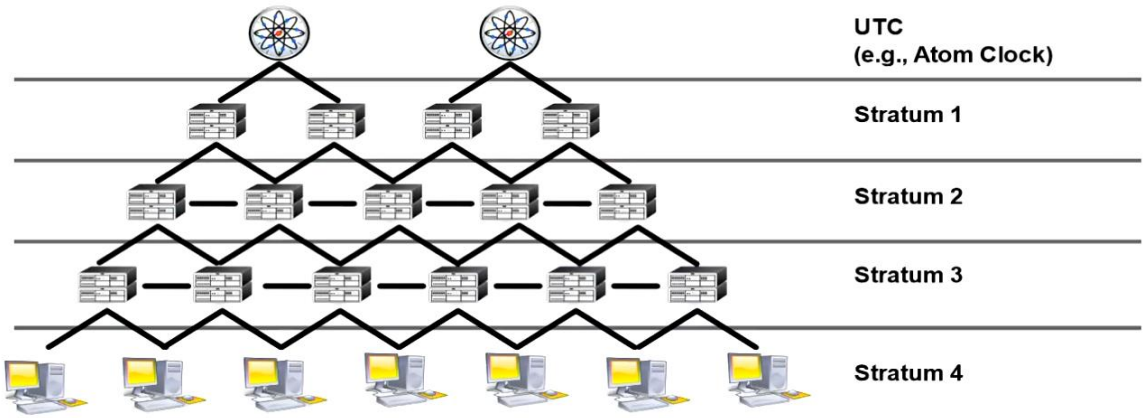
- 8 Paare buffern (δ, Θ)
  - Paar mit minimalem δ und dem dazugehörigen Offset Θ auswählen
- Was macht man, wenn Θ negativ ist?
  - Zeit darf nicht zurückgestellt werden
  - Also: Uhr langsamer laufen lassen, bis die Korrektur durchgeführt wurde



T_1	T_2	T_3	T_4	δ	θ
12:15	12:11	12:15	*12:30*	5.5	-9.5
12:15	12:11	12:15	*12:21*	1	-5

NTP: Symmetrische Zeitausbreitung

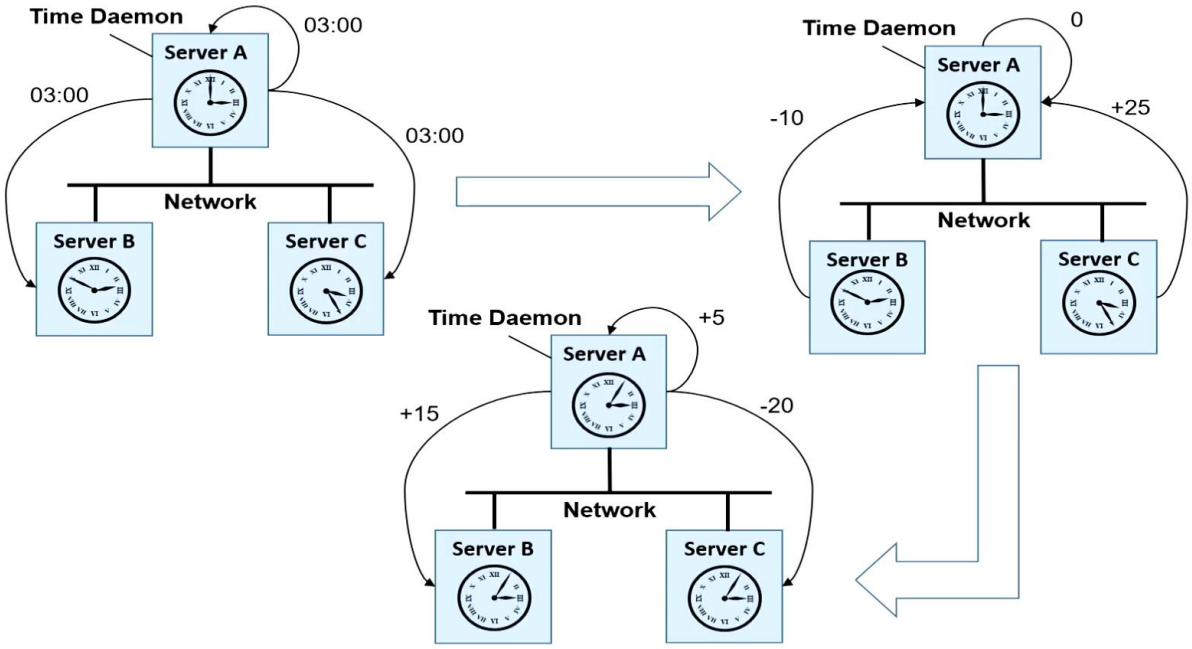
- NTP wird bidirektional verwendet, d.h. beide beteiligten Server erhalten das Delta und den Offset und passen ihre Uhren an



Berkeley Algorithmus: Grundidee

- NTP funktioniert am besten, wenn man eine sehr genaue Quelle hat
- Wenn man keine zuverlässige Zeitquelle hat: Ein lokaler Zeitserver soll alle Maschinen periodisch scannen, den Durchschnitt berechnen und die übrigen Maschinen darüber informieren, wie sie ihre Zeit *relativ zur jetzigen Zeit* anpassen sollen

Berkeley Algorithmus: Beispiel

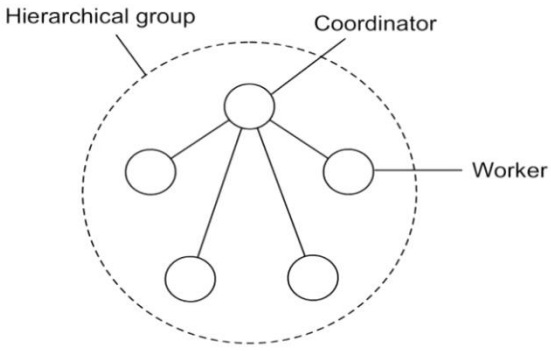


Berkeley Algorithmus: Übersicht

- Zeit Daemon ist pollingbasiert
- Vorteil: Maschinen werden ohne UTC synchronisiert
- Nachteil: Nur geeignet in kleinen Umgebungen

Auswahlalgorithmen – Motivation

- Wie können wir diesen speziellen Prozess *dynamisch* auswählen?
- Grundannahmen:
    - Alle Prozesse haben eindeutige IDs
    - Alle Prozesse kennen die IDs von allen Prozessen im System (aber nicht ob Prozess verfügbar ist)
    - Auswahl bedeutet Identifizierung des verfügbaren Prozesses mit der höchsten ID



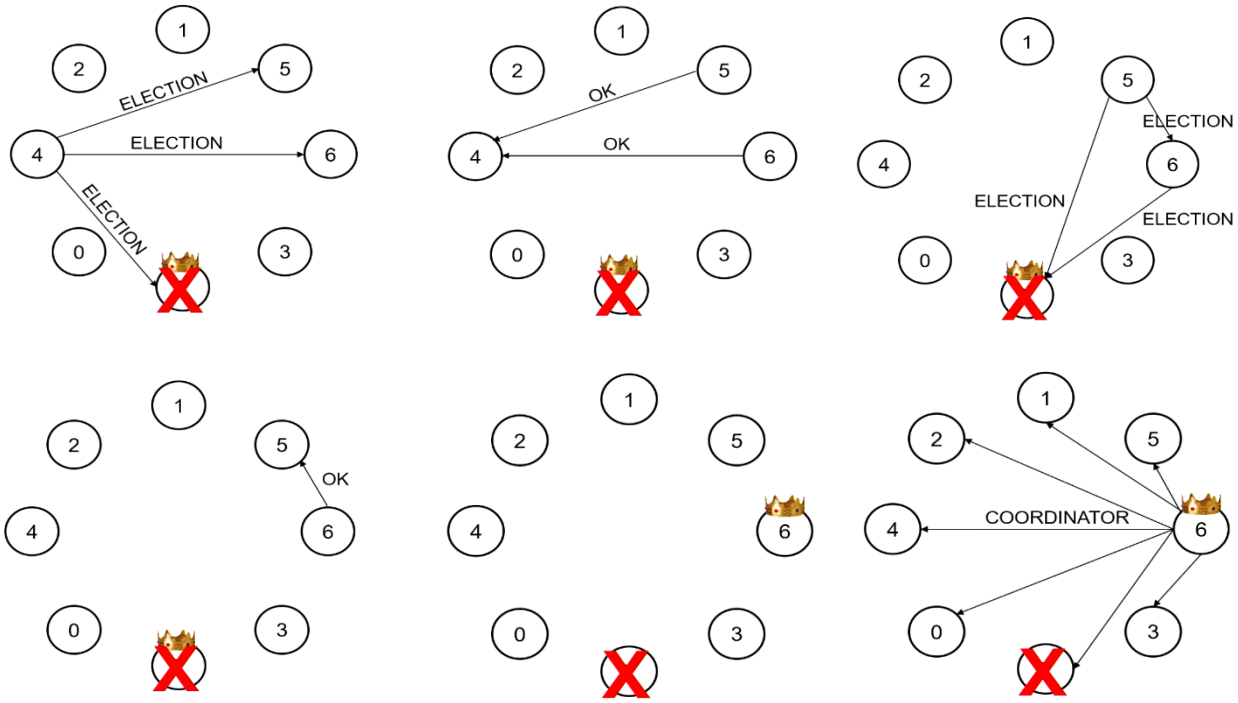
Auswahl durch Mobbing (Bullying)

Sehr einfacher Ansatz:

- Jeder Prozess kann Fehlschlag/Abwesenheit von Koordinator mitbekommen
- Jeder Prozess kann Auswahl auslösen

Auswahlverfahren:

1. P_k sendet eine ELECTION Nachricht an alle Prozesse mit höheren Identifikatoren (P_{k+1}, P_{k+2}, …)
2. Wenn keiner antwortet, gewinnt P_k
3. Wenn keine höheren Prozesse verfügbar sind, wird dieser übernehmen

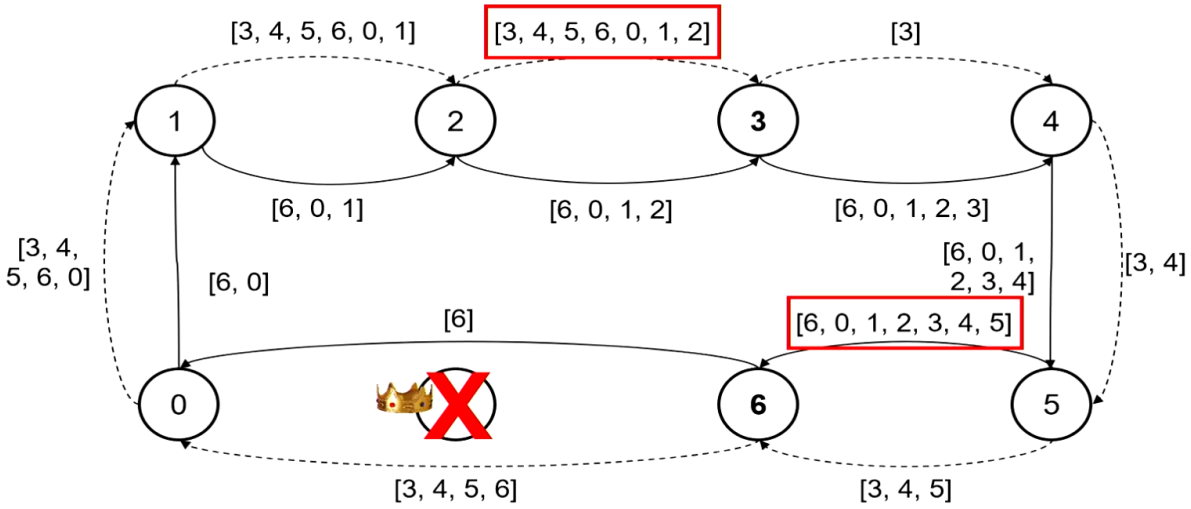


Auswahl durch Ring

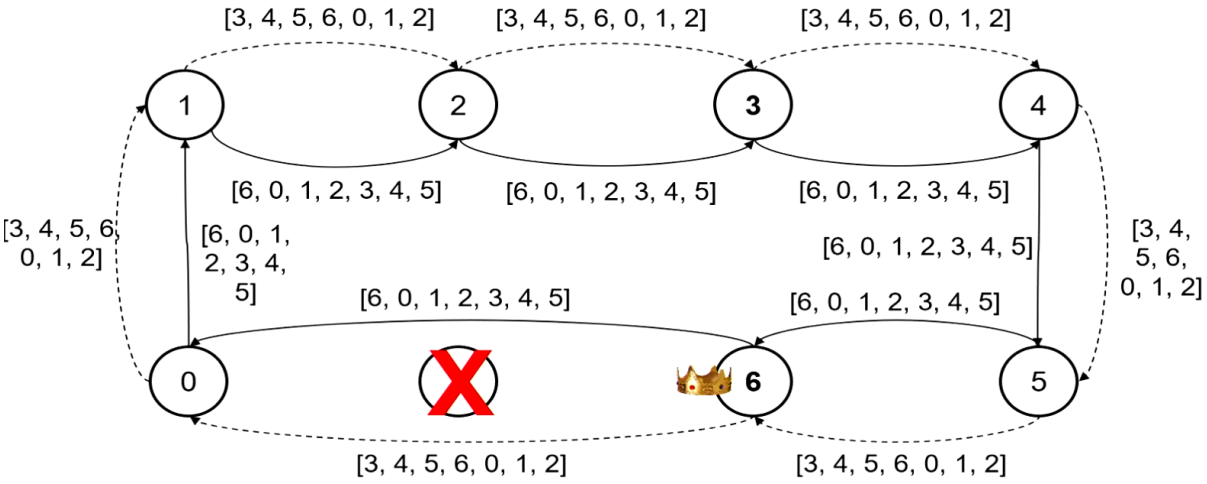
Prozesse befinden sich in einem logischen Ring, **geordnet** nach Prozesspriorität

- Jeder Prozess kann eine Auswahl starten, indem eine *election* Nachricht zum Nachfolger geschickt wird. Wenn der Nachfolger nicht erreichbar ist, wird die Nachricht weitergeleitet zum nächsten Nachfolger.
- Wenn die Nachricht weitergeleitet wird, fügt der Sender sich selbst zu einer Liste in der Nachricht hinzu. Wenn sie zurück zum Initiator kommt, hatte jeder eine Chance seine Anwesenheit bekannt zu machen.
- Der Initiator sendet eine *coordinator* Nachricht um den Ring herum, welche eine Liste mit allen verfügbaren Prozessen beinhaltet. Der eine Prozess mit der höchsten Priorität wählt sich selbst als Koordinator.

Auswahl in einem Ring: Election Nachrichten

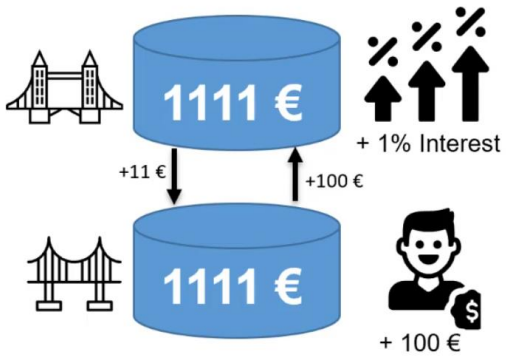


Auswahl in einem Ring: Coordinator Nachrichten

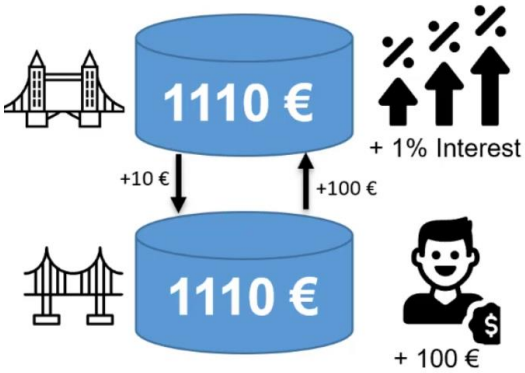


Logische Uhren: Motivation

Fall 1: Erst die Einzahlung, dann die Zinsen



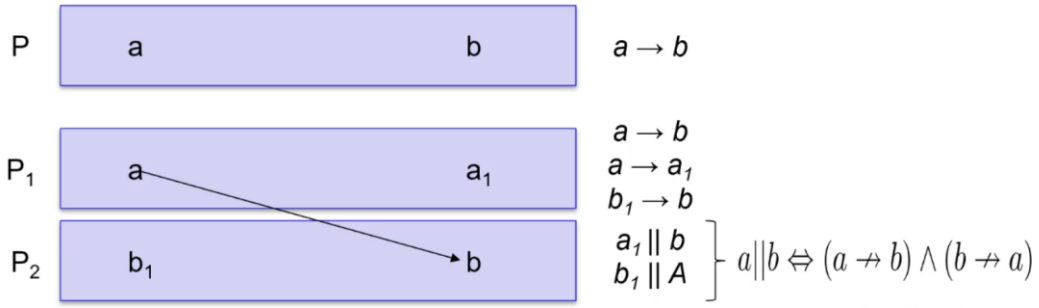
Fall 2: Erst die Zinsen, dann die Einzahlung



„Passiert-vorher“ Relation

- $a \rightarrow b$: „Ereignis a passiert vor Ereignis b“

1. Wenn a und b 2 Ereignisse im gleichen Prozess sind und a vor b kommt, dann $a \rightarrow b$
2. Wenn a das Senden einer Nachricht ist und b das Empfangen dieser Nachricht, dann $a \rightarrow b$
3. Wenn $a \rightarrow b$ und $b \rightarrow c$, dann $a \rightarrow c$



Logische Uhren: Globale Sicht

Frage: Wie warten wir eine Globale Sicht auf dem Verhalten des Systems, das konsistent mit der „Passiert-vorher“ Relation ist.

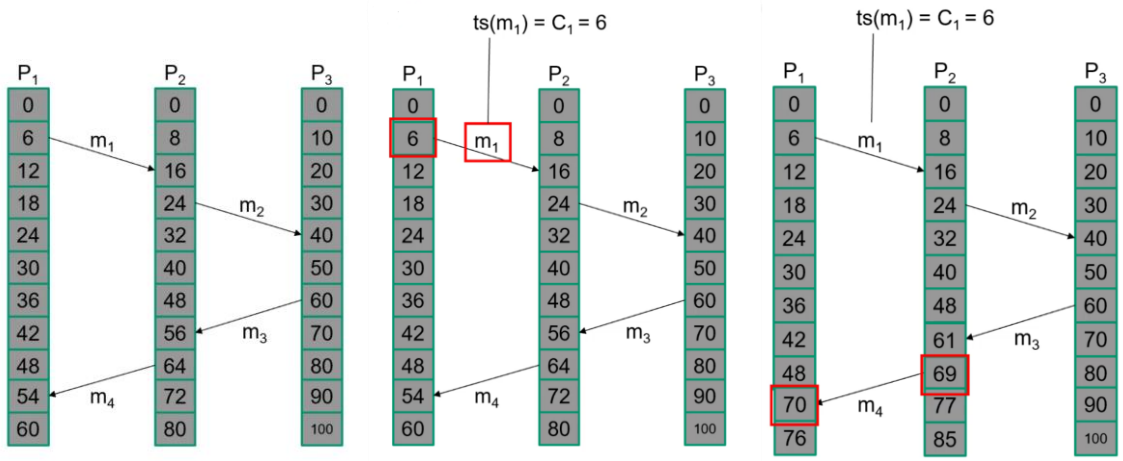
Lösung: Zeitstempel $C(e)$ zu jedem Ereignis e hinzufügen, um zwei Eigenschaften zu erfüllen:

1. Wenn a und b *2* Ereignisse im gleichen Prozess sind und $a \rightarrow b$, dann fordern wir, dass $C(a) < C(b)$
2. Wenn a dem Senden einer Nachricht m entspricht und b dem Empfang der Nachricht, dann auch $C(a) < C(b)$

Lamports Algorithmus

Jeder Prozess P_i hat einen lokalen Zähler C_i und passt diesen an:

1. Bevor jedes Ereignis, das in P_i stattfindet, wird C_i durch einen Wert erhöht, d.h.: $C_i \leftarrow C_i + n$
2. Bei jedem **Senden** einer Nachricht von Prozess P_i bekommt die Nachricht einen Zeitstempel $ts(m) = C_i$
3. Wann auch immer die Nachricht m vom Prozess P_j **empfangen** wird, ändert P_j seinen Zähler C_j zu $\max\{C_j, ts(m)\}$, führt Schritt 1 aus und übergibt m an die Applikation



Schwächen von Logischen Uhren

- Erfassen keine Kausalität
- $a \rightarrow b \Longrightarrow C(a) < C(b)$
  aber:
  $C(a) < C(b) \not\Longrightarrow a \rightarrow b$

Vektoruhren

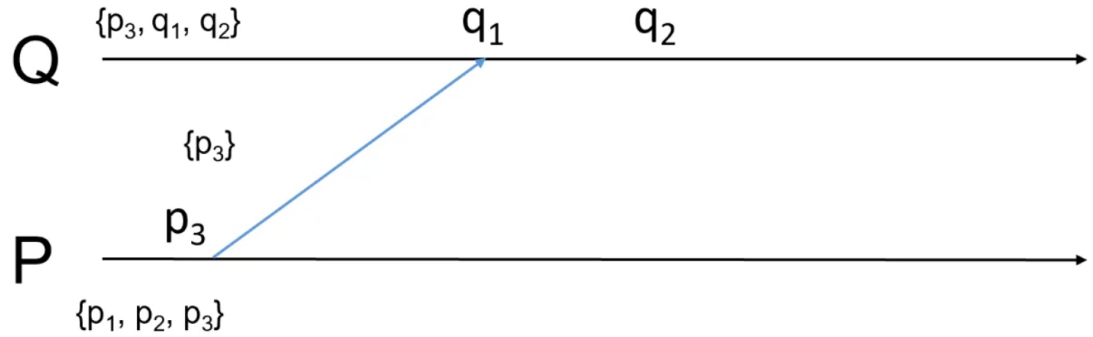
Kausale Abhängigkeit:

- Ermöglichen uns folgendes festzustellen: Wenn $VC(a) < VC(b)$, dann geht a kausal b voraus

Grundidee:

- Wir ordnen jedem Ereignis a einen eindeutigen Namen zu: p_k ist das k^{te} Ereignis, das bei Prozess P passiert
- Kausale Geschichte $H(p_2)$ von Ereignis p_2 ist $\{p_1, p_2\}$

Nachrichten senden



- Abgesehen vom eigentlichen Nachrichteninhalt wird auch die kausale Geschichte gesendet
- Kausale Geschichte des Senders wird in die kausale Geschichte des Empfängers integriert
- Kontrollieren, ob Ereignis p kausal einem Ereignis q vorausgeht

Vektoruhren: Grundidee

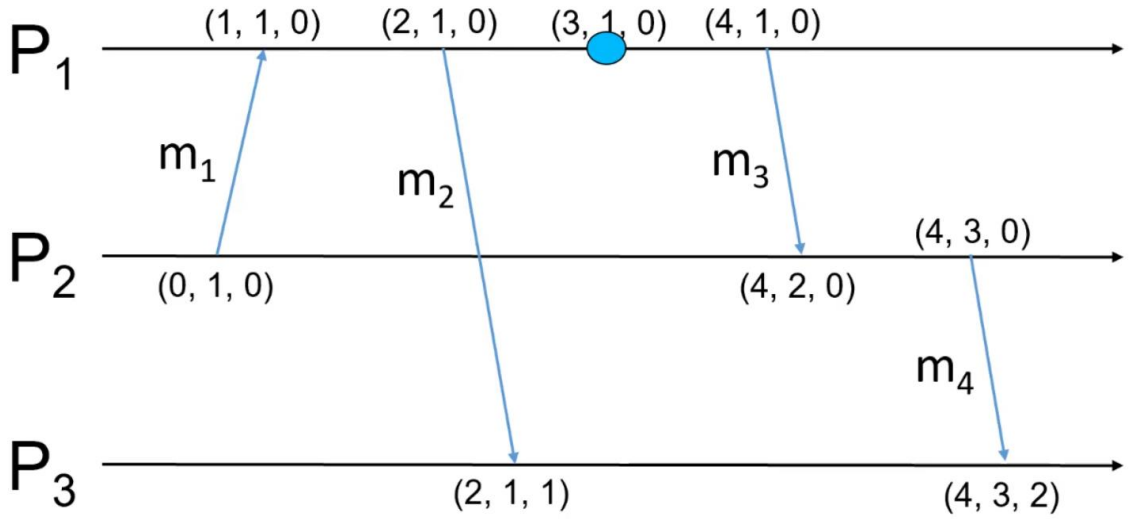
Jeder Prozess P_i beinhaltet eine Vektoruhr VC_i, wo:

1. $VC_i[i]$ die Anzahl der Ereignisse ist, die in P_i stattgefunden haben
2. $VC_i[j] = k$ bedeutet, dass P_i weiß, dass k Ereignisse schon einmal in P_j aufgetreten sind

Pflegen von Vektoruhren:

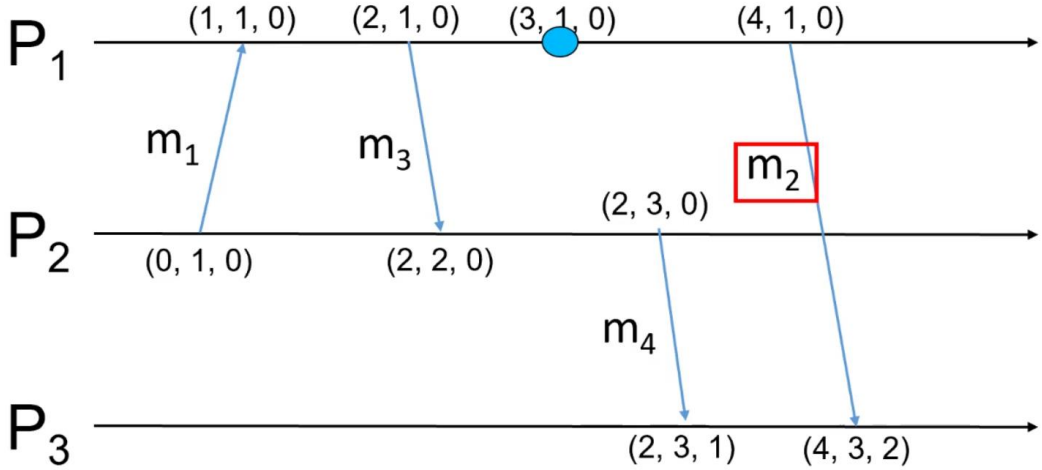
1. $VC_i[i]$ erhöhen bevor ein Ereignis ausgeführt wird
2. Beim Senden einer Nachricht m wird der (Vektor-) Zeitstempel $ts(m)$ von m gleich VC_i gesetzt nachdem Schritt 1 ausgeführt wurde
3. Beim Empfangen von m ändert P_j $VC_j[k]$ zu $max\{VC_j[k], ts(m)[k]\}$ für jedes k; danach wird Schritt 1 ausgeführt

Vektoruhren: Beispiel



Erinnerung: Beim Empfangen von *m* ändert P_j $VC_j[k]$ zu $max\{VC_j[k], ts(m)[k]\}$ für jedes *k*; danach wird Schritt 1 ausgeführt

Vektoruhren: Beispiel 2



	$ts(m_2)$	$ts(m_4)$	$ts(m_2) < ts(m_4)$	$ts(m_2) > ts(m_4)$	Conclusion
Last slide	(2, 1, 0)	(4, 3, 0)	Yes	No	m_2 may causally precede m_4
Here	(4, 1, 0)	(2, 3, 0)	No	No	m_2 and m_4 may conflict

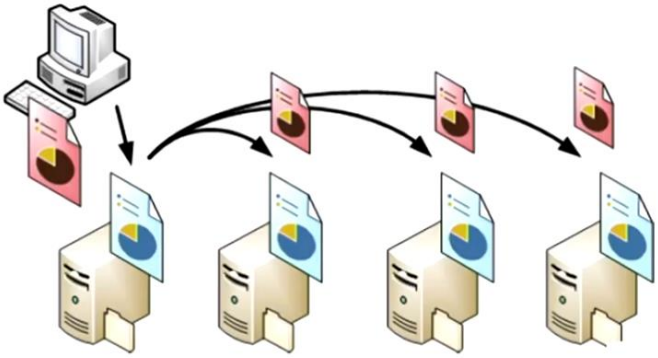
Zusammenfassung

- Uhrensynchronisierung: Network Time Protocol & Berkeley Algorithmus
- Auswahlalgorithmen
- Logische Uhren: Lamports logische Uhren und Vektoruhren

Konsistenz und Replikation

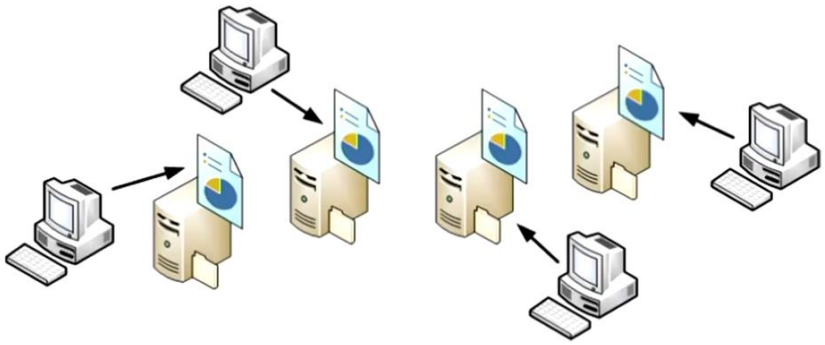
Ein paar Definitionen

- Replikation ist der Prozess, mehrere Kopien von Datenelementen an verschiedenen Orten aufrecht zu erhalten
- Konsistenz ist der Prozess, Kopien von Datenelementen gleichbleibend zu lassen, wenn es Änderungen gibt

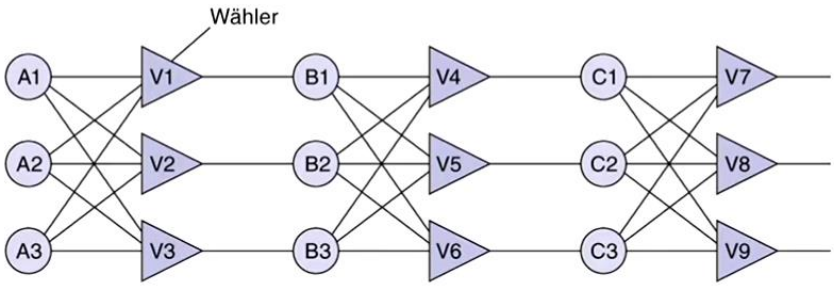


Gründe für Replikation

1. Performance und Skalierbarkeit
    a. Skalieren mit Anzahl
    (Mehr Replikas können mehr Clientanfragen bedienen)
    b. Skalieren mit geographischer/topologischer Komplexität
    (Replikas nahe zum Client können Antwortzeit verbessern)

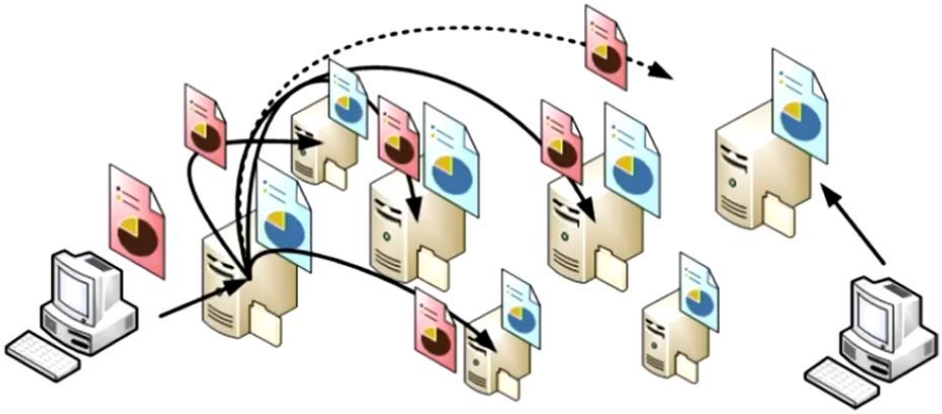


2. Fehlertoleranz durch Redundanz
    a. Umswitchen bei Ausfällen
    b. Schutz vor beschädigten Daten (Mehrheitsauswahl)



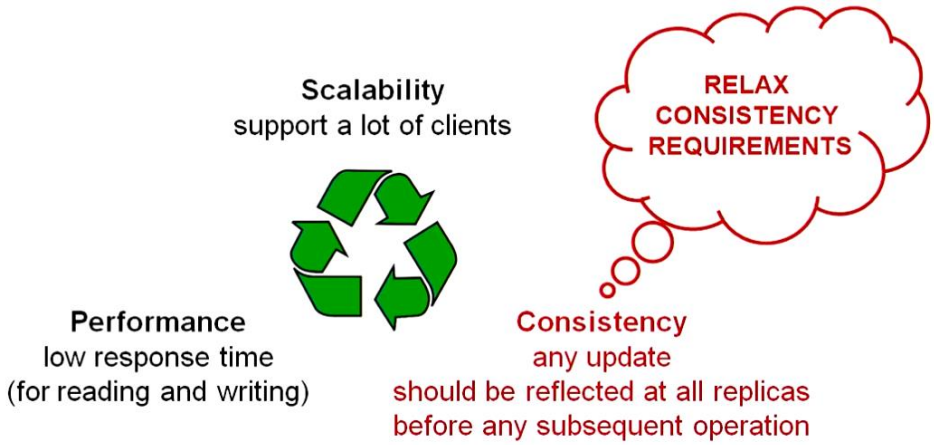
Nachteile von Replikation

- Replikate auf dem neusten Stand zu halten verbraucht Bandweite
- Aktualisierungen für Replikate werden nicht sofort verbreitet (veraltete Daten)



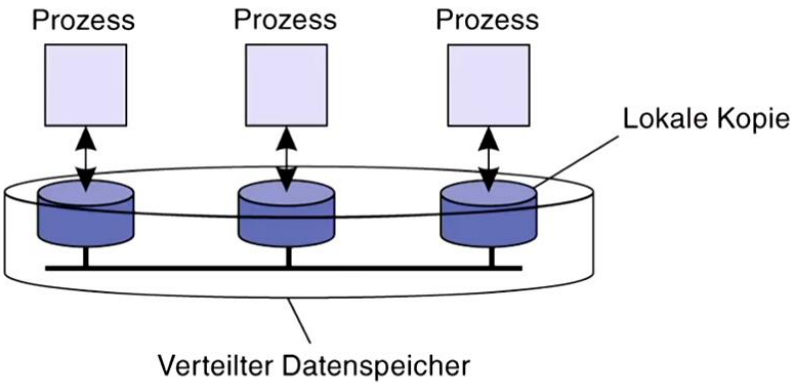
Performance vs. Skalierbarkeit Trade-Off

- Wir wollen alle drei Eigenschaften verbessern, aber in der Realität können wir nur zwei wählen und das Dritte verwerfen



Verteilter Datenspeicher Modell

- Konzeptuelles Modell, welches die verschiedenen echt verteilten Speichersysteme repräsentiert (entfernter geteilter Speicher, verteiltes Dateisystem, verteilte Datenbank)
- Prozesse haben Zugriff auf den gesamten Datenspeicher durch die lokalen Kopien



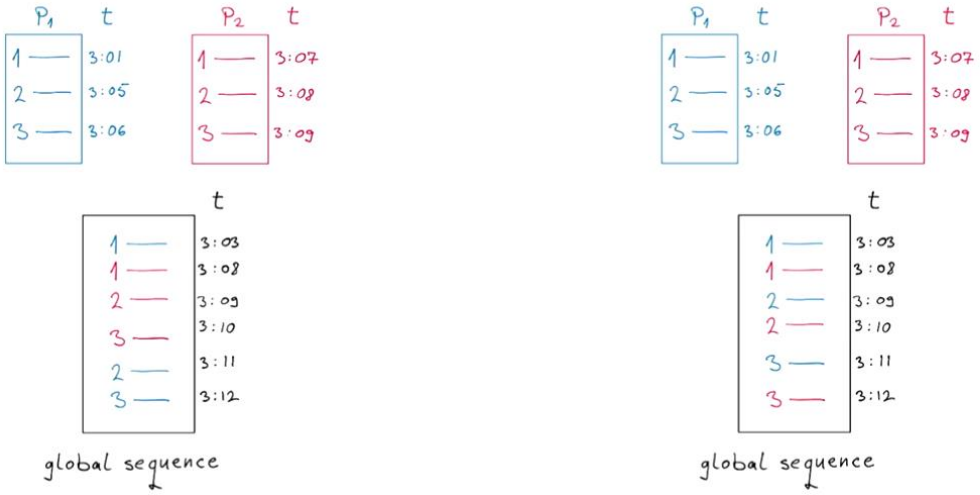
Konsistenzmodelle

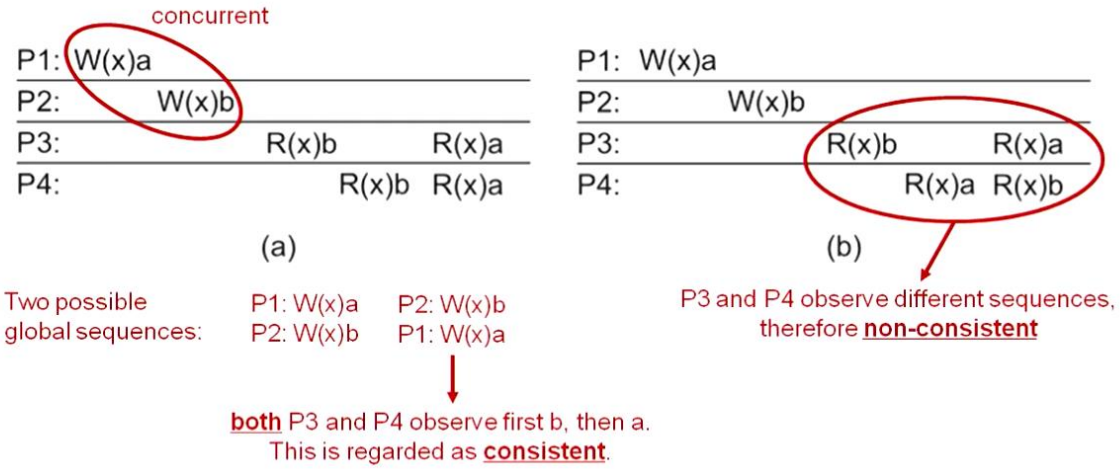
- **Ein Vertrag zwischen einem (verteilten) Datenspeicher und Prozessen, in welchem der Datenspeicher genau beschreibt, was die Ergebnisse von Lese- und Schreiboperationen sind bezüglich Nebenläufigkeit**
- Das Modell erlaubt es, diese fundamentale Erwartung zu erfüllen:
  Ein Prozess, der eine Leseoperation auf ein Element X von der lokalen Copy ausführt, erwartet den letzten Wert von X zu sehen, der von jedem anderen Prozess in dessen lokale Kopie geschrieben wurde
- In der Praxis können wir diese Erwartung lockern, solange die Funktionsweise des Gesamtsystems nicht in Gefahr ist
- **Zwei Ansätze**
  - **Datenzentriert**:
    garantiert die Konsistenz des gesamten Datenspeichers
  - **Clientzentriert**:
    garantiert die Konsistenz der Daten von einem einzigen Client, wenn der Client auf die Daten über verschiedene Knoten zugreift
- **Wir benutzen die folgende Notation**:

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

Sequenzielle Konsistenz

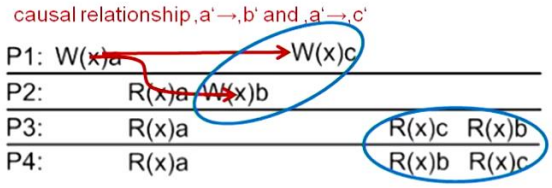
Das Ergebnis von jeder Ausführung ist das Gleiche wie wenn die Operationen von allen Prozessen in einer sequenziellen Reihenfolge ausgeführt wurden und die Operationen von jedem individuellen Prozess in dieser Sequenz in der vom Programm festgelegten Reihenfolge auftauchen.





Kausale Konsistenz

Schreiboperationen, die potenziell kausal verbunden sind, müssen von allen Prozessen in derselben Reihenfolge gesehen werden. Nebenläufige Schreiboperationen könnten von verschiedenen Prozessen in verschiedener Reihenfolge gesehen werden.

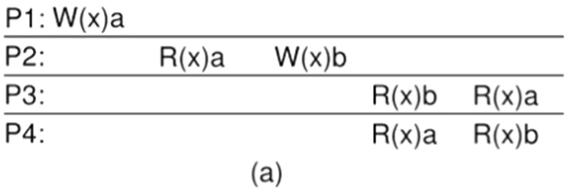


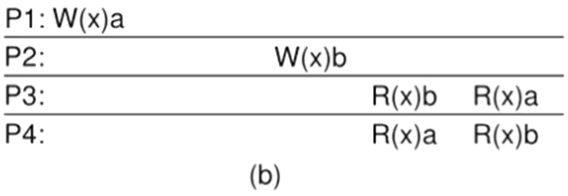
P3 und P4 beobachten beide ‚a' vor ‚b' und ‚a' vor ‚c'. Deshalb sind sie kausal konsistent.

Weil W1(x)c und W2(x)b nebenläufig sind, können P3 und P4 diese in irgendeiner Reihenfolge beobachten.

Dies ist aber nicht sequenziell konsistent, da P3 und P4 verschiedene Sequenzen von b und c beobachten.

Kausale Konsistenz – Mehr Beispiele

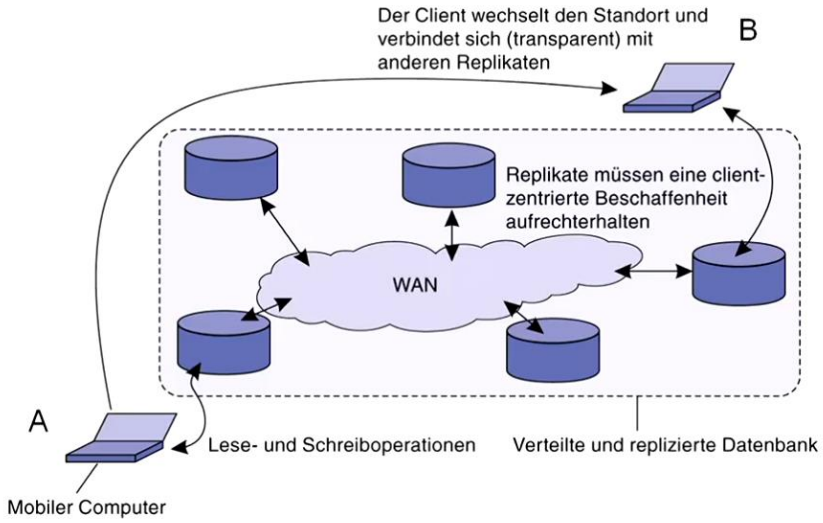




(a) kausal inkonsistent, (b) konsistent

Prinzip der eventuellen Konsistenz

- Bis jetzt haben wir (in datenzentrierter Konsistenz) angenommen, dass das System die Konsistenz während **nebenläufiger Aktualisierungen** verwalten muss
    - Aufrechterhaltung von Konsistenz ist sehr teuer, da praktische Implementationen Synchronisierungsmechanismen benötigen
- Wenn das reale System, das wir modellieren möchten, jedoch so ist, dass zu einem bestimmten Zeitpunkt nur eine einzelne Entität Schreibvorgänge ausführen kann, dann müssen keine nebenläufigen Aktualisierungen stattfinden und das Konsistenzmodell kann entspannt sein
- In diesem Fall reden wir von **eventueller Konsistenz**
- Haupteigenschaft von eventueller Konsistenz:
- *Nach einem Schreibvorgang werden die Replikate **allmählich** konsistent. (*Es kann eine Weile dauern, aber sie werden garantiert irgendwann aktualisiert)
- Viele Systeme zeichnen sich nicht durch nebenläufige Schreibvorgänge aus und können deshalb gut mit eventueller Konsistenz funktionieren
    - Beispiel: Nur der Webmaster aktualisiert statische Dateien, aber viele Clients können diese lesen
- Aber es gibt ein Problem:



Clientzentrierte Konsistenzmodelle

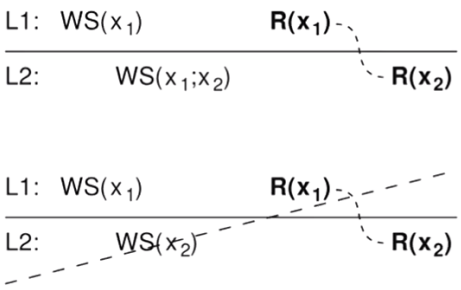
- Dieses Problem kann mit Hilfe von **clientzentrierten Konsistenzmodellen** gelindert werden
- Sie **garantieren für einen einzelnen Client**, der auf einen verteilten Dateispeicher zugreift, dass seine Zugriffe auf die Daten, die er besitzt, konsistent sein werden

Notation

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

Monotonische Lesekonsistenz

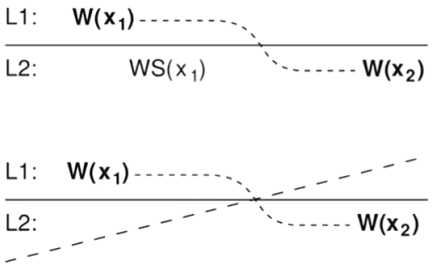
Wenn ein Prozess den Wert eines Datenelements *x* liest, wird jede aufeinanderfolgende Leseoperation auf *x* von diesem Prozess immer genau diesen oder einen neueren Wert zurückgeben.



Beispiel:

Automatisches Lesen Ihrer persönlichen Kalenderupdates von verschiedenen Servern: Monotonisches Lesen garantiert, dass der User alle Updates sehen wird, egal von welchem Server das automatische Lesen ausgeführt wird.

Eine Schreiboperation eines Prozesses auf ein Datenelement *x* ist fertig vor jeder aufeinanderfolgenden Schreiboperation auf *x* desselben Prozesses.

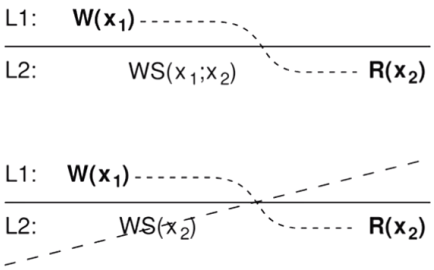


Beispiel:

Aufrechterhalten von Versionen replizierter Dateien in der richtigen Reihenfolge überall (die vorherige Version zu dem Server, wo die neuste Version installiert ist, verbreiten)

„Lies deine Schreiboperationen" Konsistenz

Der Wirkung einer Schreiboperation eines Prozesses auf ein Datenelement *x* wird immer von aufeinanderfolgenden Leseoperationen auf *x* desselben Prozesses gesehen.

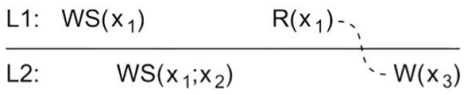


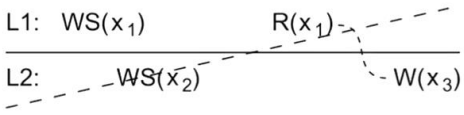
Beispiel:

Aktualisierung deiner Webseite und Garantie, dass der Webbrowser die neuste Version anzeigt statt der gecacheten Kopie

„Schreiben folgt dem Lesen“ Konsistenz

Eine Schreiboperation eines Prozesses auf ein Datenelement *x*, gefolgt von einer vorherigen Leseoperation auf *x* desselben Prozesses, wird garantiert an demselben oder neueren gelesenen Wert von *x* stattfinden.





Beispiel:

Posten einer Antwort zu einem Forumseintrag erst nach Holen des ursprünglichen Eintrags.

Von Konsistenzmodellen zu Replikatverwaltung

- Wir haben gelernt, dass es verschiedene Wege gibt, wodurch Objektrepliken konsistent gehalten werden können (Konsistenzmodelle)
- **Aber was ist die beste Strategie, die Replikate so zu platzieren, dass sie Replikation effizient nutzen und in der Lage sind, Konsistenz so günstig wie möglich aufrechtzuerhalten?**

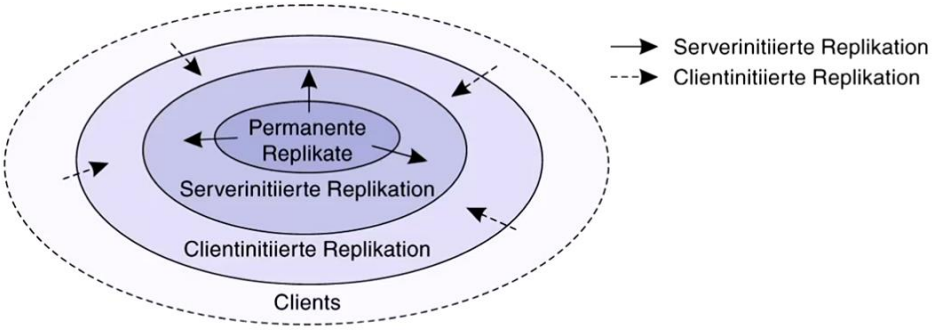
Grundlegende Überlegungen:

- Wenn keine Aktualisierungen für das Objekt → Kein Konsistenzproblem
- Wenn **Zugriff-zu-Aktualisierungsrate** hoch ist, zahlt sich Replikation aus
- Wenn **Aktualisierung-zu-Zugriffsrate** hoch ist, werden viele Aktualisierungen nie gelesen
- Idealerweise Replikate aktualisieren, auf die zugegriffen wird
- Als Standardregel versuchen wir ein Replikat nahe an seinen Clients zu halten

Replikatverwaltung – Herausforderungen

- Replikatserverplatzierung
  - Oft ein kommerzielles oder Verwaltungsproblem
- Inhaltsreplikation und Platzierung
- Inhaltverteilung
  - Status vs. Operation
  - Push vs. Pull vs. Lease
  - Blockierend vs. Nicht-Blockierend (eifrig vs. faul)
  - Unicast vs. Multicast (Gruppenkommunikation)

Replikatplatzierung

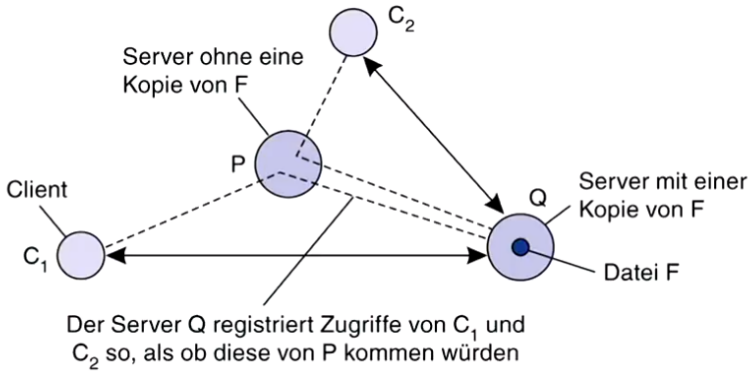


Permanente Replikate

- Grundlegende Replikation
  - Oft erste Verteilung der Daten
  - Unterstützt von (Speicher-)System selbst
- Beispiele:
  - Replizierte Webseitenserver für Lastverteilung (Client sieht keinen Unterschied)
  - Spiegelung (Mirroring) (Client weiß Bescheid und schaut nach bestimmtem Server)
  - Normalerweise nicht so viele Kopien

Serverinitiierte Replikate

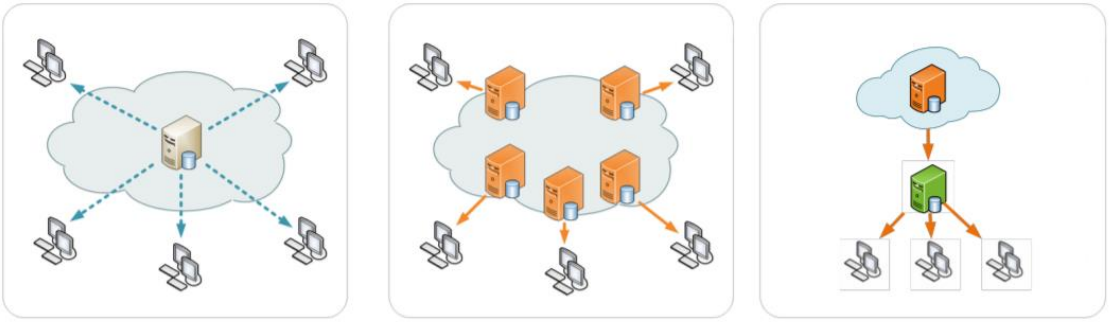
- Der Server entscheidet allein, wann und wo mehr Replikate erstellt werden sollen
- Viele verschiedene und komplexe Algorithmen
  - Verwenden von Monitoring und Statistiken der Zugriffe, um optimale Platzierung herauszufinden
- Basis für Web Hosting und Cloud-Dienste



- Oft von Web Hosting Firmen genutzt
- Jeder Server weiß, welcher der nächste Server für jede Clientanfrage wäre
- Jeder Server behält die Zugriffsanzahl pro Datei im Auge, aggregiert unter Berücksichtigung des Servers, der dem anfragenden Client $cnt_Q(P,F)$ am nächsten liegt
- Jeder Server hat eine **Replikationsschwelle R** und eine **Löschungsschwelle D**

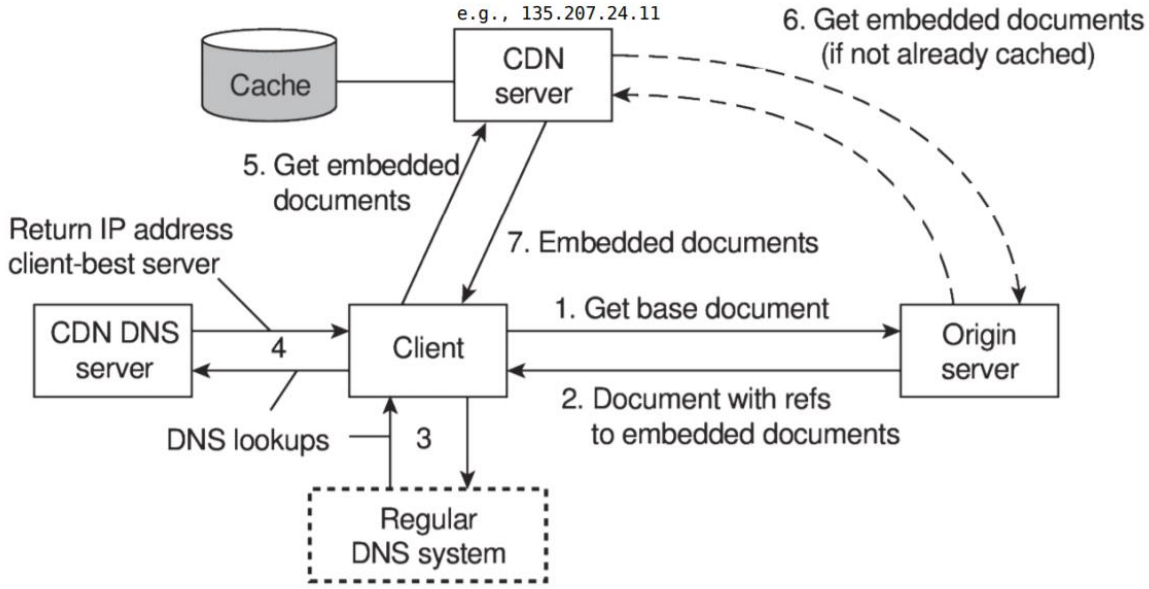
- Zugriffsanzahl fällt unter Schwelle D → Datei wird fallengelassen
- Zugriffsanzahl überschreitet Schwelle R → Datei wird repliziert
- Zugriffsanzahl fällt zwischen D und R → Datei wird migriert

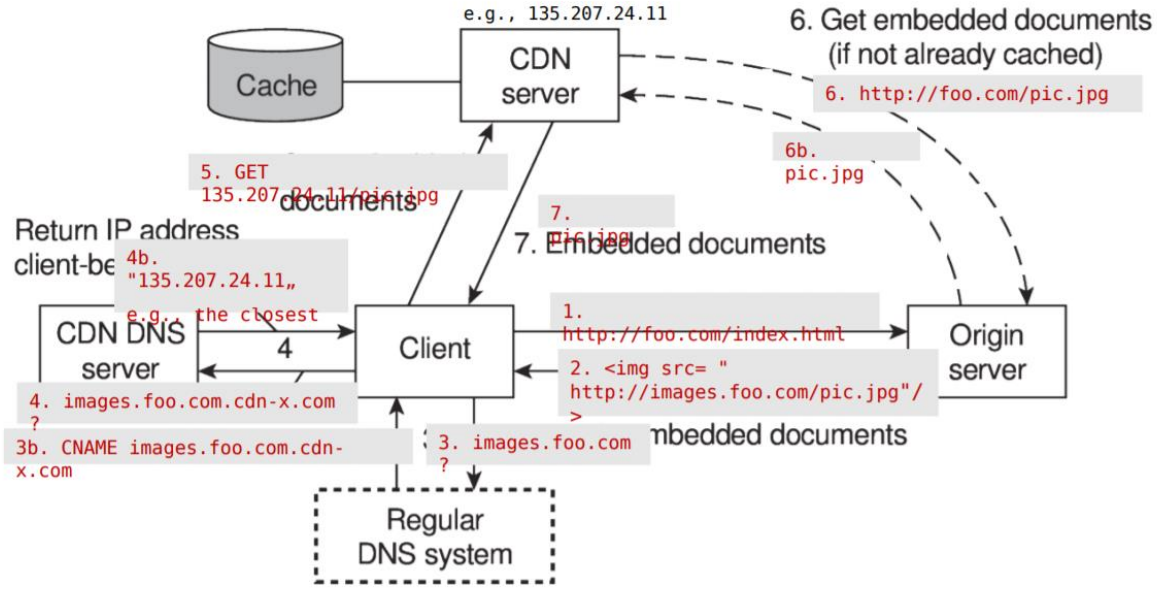
- Ob Migration/Replikation stattfindet kommt auf die Kosten der Operation an
  - Z.B.: Ist die Gesamtlast des Zielservers zu hoch? Hat er genug Speicherplatz?



Serverinitiierte Replikate: CND

- Beispiel: Content Delivery Network (CDN)
- Fungiert als verteilter Webhosting-Dienst
    - Webseiten betten (statische) Dokumente ein, die groß sind und sich selten ändern (Bilder, Audio-/Videodateien)
    - Es macht sinn diese über viele geographisch verteilte Server zu verteilen, um schnelleren Zugriff für viele Clients zu ermöglichen (parallel)
        - Maßstab: ~100k Server in über 100 Ländern
- Replikation wird dynamisch ausgelöst, basierend auf den Laufzeitmetriken:
    - (z.B.: Latenz, Bandweite, finanzielle Aspekte)





Inhaltverteilung zwischen Replikaten

Wir haben schon konkrete Beispiele gesehen, wie Daten über mehrere Server verteilt werden

Bevor wir mit der clientinitiierten Replikation fortfahren, lassen Sie uns kurz über **allgemeine** Strategien zur Inhaltsverteilung diskutieren (beide für server- und clientinitiierte Replikation)

- Annahmen:
  - Wir gehen von einem generischen System auf N Replikatmaschinen aus
  - Daten werden von einer der Replikate aktualisiert
  - **Wie bekommen andere Replikate diese Aktualisierung?**

1. Invalidation:

- Replikate werden nicht darüber informiert, dass es eine Aktualisierung gab.
  Es werden keine wirklichen Daten gesendet → Geringe Bandweite
- Andere Replikate markieren die Daten als ungültig
  - Sie entscheiden, wann aktualisiert wird (hängt von dem Konsistenzmodell ab)
- Gut, wenn es viele Aktualisierungen gibt, die selten gelesen werden

2. Data Transfer:

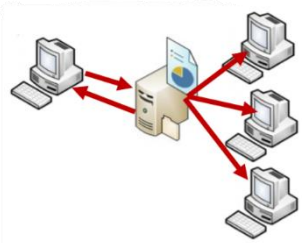
- Die aktualisierten Daten werden zu anderen Replikaten über eine Aktualisierung kopiert
  - Kann server- oder clientinitiiert sein
- Es ist sinnvoll, wenn Lese-Schreib-Rate hoch und strikt konsistent ist
  - Versichert, dass Aktualisierungen gesehen werden sobald sie passieren
  - Verbraucht aber Bandweite
    - Mehrere Änderungen können zusammengepackt und als einzelne Aktualisierung gesendet werden, falls sehr häufig

3. „Aktive“ Replikation:

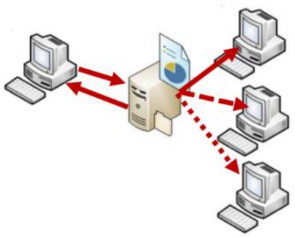
- Nicht die wirklichen Daten senden, sondern die Instruktionen, die zu den neuen Werten führen:
  - Z.B.: Senden von Operationen und Parametern wie bspw. *invert_matrix(A)*
- Das ist für eine begrenze Anzahl an Spezialfällen möglich

Inhaltsverteilung – Blockierend vs. Nicht-Blockierend

Ein Replikat möchte anderen Replikaten eine Aktualisierung senden:



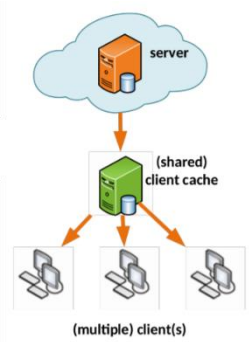
- Synchron (**blockierend**, **eifrig**):
  Alle Replikate werden sofort aktualisiert; erst dann antworten sie dem ursprünglichen Replikat



- Asynchron **(nicht-blockierend**, **faul**)
  Sobald die Aktualisierung bei einem anderen Replikat ankommt, fährt der Absender fort. Die Vermehrung zu anderen Replikaten passiert danach.

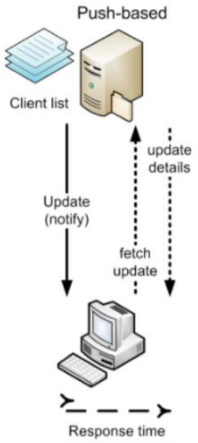
Clientinitiierte Replikation

- **(Client) Cache** – lokale Speicherstätte, die von Client für das temporäre Speichern von Daten und der Verbesserung der Zugriffszeit verwaltet und benutzt wird
    - Besonders brauchbar, wenn Lese-Schreib-Rate hoch ist
    - **Cache-Treffer**, wenn Daten im Cache gefunden werden
- Daten werden auf Anfrage des Clients im Cache abgerufen
- Daten werden nur für begrenzte Zeit gespeichert
- Normalerweise befindlich auf der Clientmaschine selbst oder in der Nähe (falls mit mehreren Clients geteilt)



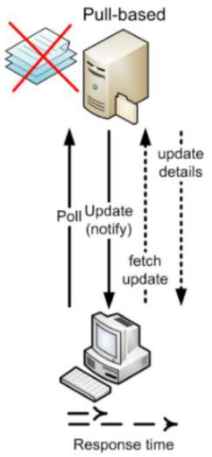
Inhaltsverteilung – Push vs. Pull

Push-basierte (serverbasierte) Protokolle



- Aktualisierungen werden zu anderen Clients weitergegeben, ohne danach fragen zu müssen
    - Schnelleres Erreichen von Konsistenz! (Hauptanwendungsgrund)
- Wenn der Server alle Clients auf einmal aktualisieren möchte, ist das nicht effizient (skalierbar): dauert lange und jeder Client kann fehlschlagen
    - Außer effiziente **Multicast**-Implementation ist verfügbar (z.B. in LAN)
- Alternativ nur Clients aktualisieren, die Informationen benötigen
    - Der Server muss wissen, was jeder Client hat → Statusbehafteter Server

Pull-basierte (clientbasierte) Protokolle



- Client fragt Server ab, um zu überprüfen, ob Aktualisierungen verfügbar sind und fragt anschließend nach einer Aktualisierung
- Antwortzeit des Clients erhöht sich im Falle eines Cache-Fehlers
- Am meisten von Client Caches benutzt

Vergleichszusammenfassung

Problem	**Push-basiert**	**Pull-basiert**
Serverstatus	Liste der Clientreplikate und Caches	Nichts
Nachricht gesendet	Aktualisierung (und möglicherweise Aktualisierung später abrufen	Abfrage und Aktualisierung
Antwortzeit beim Client	Unmittelbar (oder Abruf-Aktualisierungszeit)	Abruf-Aktualisierungszeit

- Können wir die Vorteile beider Ansätze gleichzeitig nutzen?
    - → **Leasing**: Eine Hybrid-Lösung, die dynamisch zwischen Pull und Push wechselt

Inhaltsverteilung – Leasings

Leasing – Ein Vertrag, in dem der Server verspricht, dem Client so lange Aktualisierungen zu pushen, bis das Leasing abläuft.

- Wann soll ein Leasing ablaufen?
  - Kommt auf das Verhalten des Servers an (anpassungsfähige Leasings)
  - Durch die Wahl unterschiedlicher Leasingdauern wollen wir die Belastung des Servers und Serverstatus minimieren und das Aktualisieren von Clients beschleunigen (Höheres Konsistenzniveau)

Inhaltsverteilung – Leasingablauf

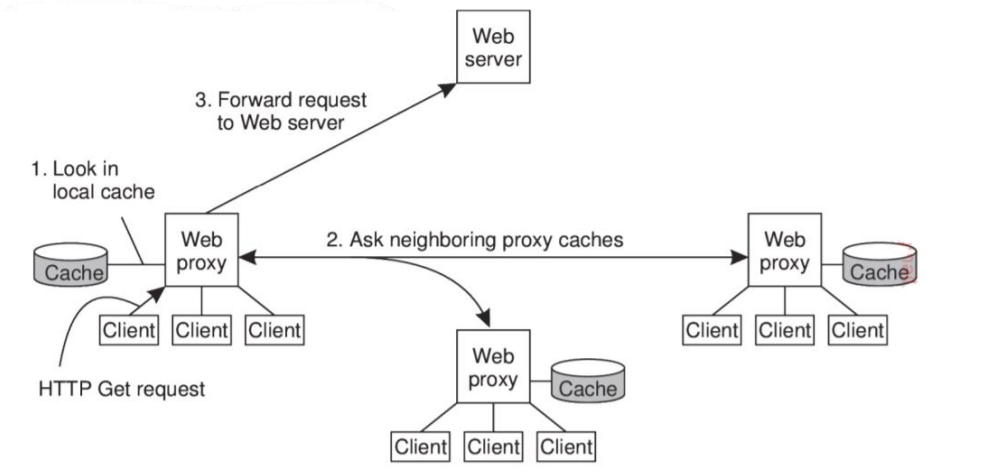
- **Altersbasierende Leasings**: Ein Objekt, das sich eine lange Zeit nicht verändert hat, wird sich in naher Zukunft nicht verändern, also wird ein langlebiges Leasing zur Verfügung gestellt
- **Erneuerungsfrequenzbasierte Leasings**: Je öfter ein Client ein bestimmtes Objekt anfragt, desto länger wird die Ablaufzeit für diesen Client (für dieses Objekt) sein
- **Statusbasierte Leasings**: Je belasteter ein Server ist, desto kürzer wird die Ablaufzeit

Was erreichen wir damit?

- Serverstatus ist kleiner
  - Begrenzt für geleaste Clients und Daten
- Nur die Clients, die wirklich ein höheres Konsistenzniveau brauchen (erreicht durch Pushes), beantragen ein Leasing, damit der Server sich in dieser Zeit seinen Ressourcen widmen kann
  - Bessere Nutzung von Server und Netzwerk. Weniger unnötige Kommunikation und Datenübertragungen

Applikation: Caching im Web

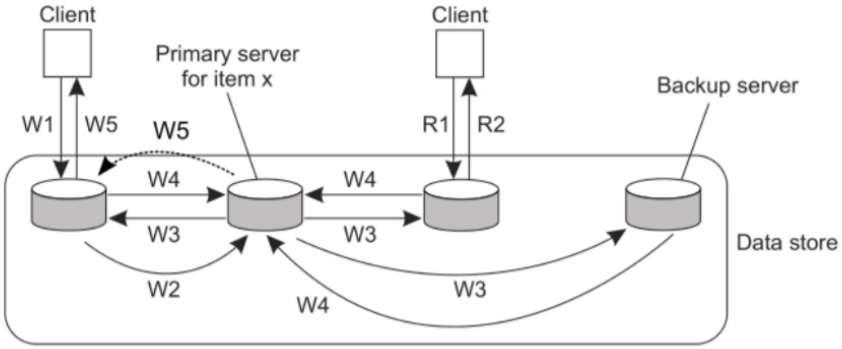
- Browsercache (privat)
- Web Proxy (geteilt)
- Webserver kann angeben, ob die Daten via HTTTP-Headern gecached werden können
- Web Proxys implementieren ein pull-basiertes Protokoll:
  - Wenn Daten älter als eine bestimmte Ablaufschwelle sind:
    - Server mit *If-Modified-Since*-Header abfragen
    - Wenn seit letzter Aktualisierung verändert, Server sendet Aktualisierung
  - Ansonsten: Auf Clientanfrage gecachete Kopie bereitstellen
- Normalerweise richten ISPs eine hierarchische Proxy-/Caching-Struktur ein
- Alternativ: Kooperatives Caching (Gut für höchst dezentralisierte Systeme



Konsistenzprotokolle

- Beschreiben die **Implementation eines bestimmten Konsistenzmodells**
- Datenzentrierte Konsistenzprotokolle:
    - Primärbasierte Protokolle
        - Primäres Backupprotokoll mit Remote-Schreibvorgängen
        - Primäres Backupprotokoll mit lokalen Schreibvorgängen
    - Replizierte Schreibprotokolle
        - Quorumbasierte Protokolle
    - Clientzentrierte Konsistenzprotokolle

Primäres Backupprotokoll

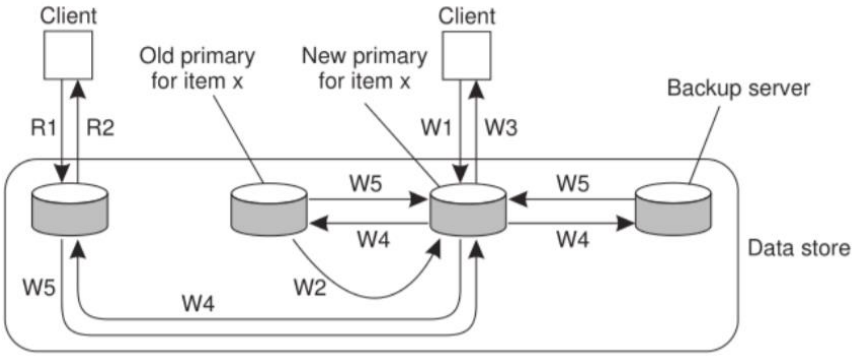


W1: Schreibanfrage
W2: Anfrage weiterleiten zu Primärserver
W3: Backups anweisen zu aktualisieren
W4: Aktualisierung bestätigen
W5: Abschließen des Schreibvorgangs bestätigen

R1: Leseanfrage
R2: Antwort auf Leseoperation

- Implementiert das **sequenzielle Konsistenzmodell**
    - Alle Schreiboperationen werden vom Primärserver geordnet und zu den übrigen Servern geschickt
    - Die lokale Kopie zu lesen gibt den aktuellsten Wert zurück. Änderungen sind atomar. Keine Inkonsistenzen
    - Lesen geht schnell, Schreiben ist langsam (Blockieroperation)
    - Wenn kein Knoten verfügbar ist, ist es nicht möglich, eine Schreiboperation durchzuführen → nicht widerstandsfähig gegenüber Netzwerk- oder Knotenfehler
- Ein nicht-blockierendes (asynchrones) Schema ist auch möglich
    - ACK sobald Primärserver die Aktualisierung erhalten hat
    - Beschleunigt das Schreiben
    - Widerstandsfähig gegen Knoten- und Verbindungsfehler
    - Aber Daten**inkonsistenzen** können auftreten
        - Ein lokaler Lesevorgang gibt nicht immer den aktuellsten Wert zurück

Primäres Backupprotokoll mit lokalen Schreibvorgängen



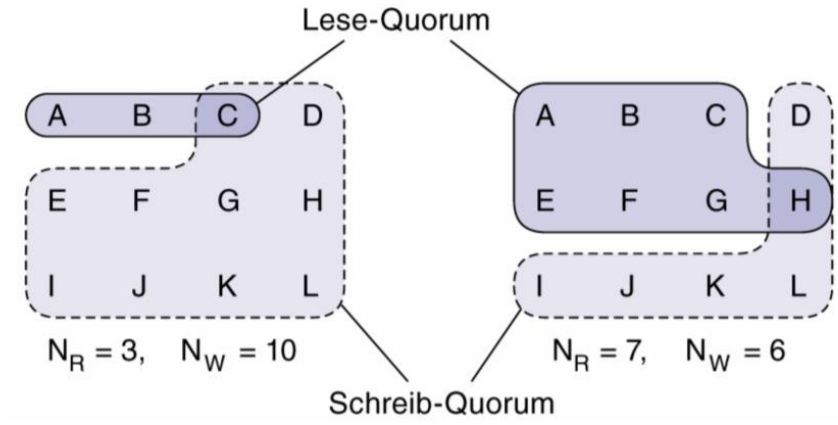
W1: Schreibanfrage
W2: Element x zu neuem Primärserver verschieben
W3: Abschließen des Schreibvorgangs bestätigen
W4: Backup anweisen zu aktualisieren
W5: Update bestätigen

R1: Leseanfrage
R2: Antwort auf Leseoperation

- Wenn es nicht zu viele nebenläufige Schreibvorgänge gibt, geht das Schreiben schnell
- Der Primärserver ist eine Fehlerstelle. Wenn er versagt, kann kein anderer Knoten Primärserver werden, oder es muss eine teure Rekonfiguration vorgenommen werden um einen anderen Knoten zum Primärserver zu machen

Quorumbasierte Protokolle

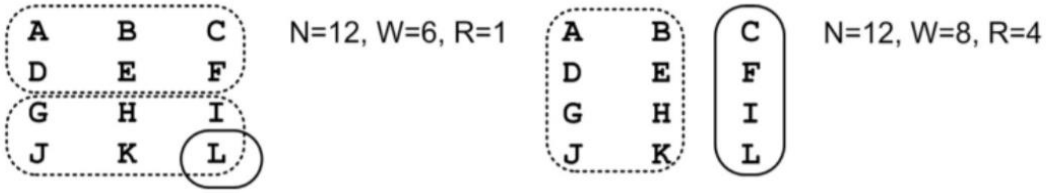
- Versichert, dass jede Operation so durchgeführt wird, dass eine Mehrheitsentscheidung getroffen wird:
  Wir unterscheiden **Lese-Quorum** und **Schreib-Quorum:**



- Grundidee: Wir haben 5 Knoten. Um auf das System schreiben zu können, muss der Client synchron bei mehr als der Hälfte der Knoten schreiben/lesen
- Daten haben monotonisch ansteigende Versionsnummern (z.B. Zeitstempel), sodass wir wissen können, welche Version aktueller ist
- Vorteil: Weniger Knoten müssen kontaktiert werden.
  Das ist wichtig, wenn es viele Knoten gibt (verringert die Operationszeit) und/oder manche Knoten (oft) unzugänglich für den Client sind

Quorumbasiertes Protokoll Konfigurationsberücksichtigungen

- Lesen optimieren: R=1, W=N
- Schreiben optimieren: W=1, R=N (Keine Haltbarkeitsgarantien)
- Schreibkonflikte vermeiden: W >= (N+1)/2
- Starke Konsistenz: W+R > N

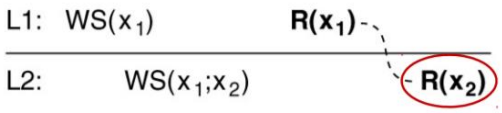


Clientzentrierte Konsistenzprotokolle

Anforderungen für (eine naive) Implementation:

- Jede Schreiboperation W ist einer eindeutigen ID zugewiesen
- Von der ID aus ist es möglich, den Server (Ursprung) zu ermitteln, wo die Schreiboperation stattgefunden hat
- Für jeden Client behalten wir zwei Gruppen:
  - Lesegruppe – Gruppe von **Schreib**operation-IDs, von welchen die Leseoperationen des Clients abhängen
  - Schreibgruppe – Gruppe der eigenen Schreiboperationen des Clients

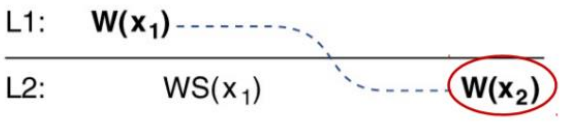
Monotonische Lesevorgänge:



Wenn ein Client eine Leseoperation auf einem Server durchführt:

1. Der Server bekommt die Lesegruppe vom Client, um zu überprüfen, ob alle leserelevanten Schreibvorgänge lokal stattgefunden haben
2. Wenn nicht, kontaktiert er die anderen Server, um sicherzugehen, dass er auf den neuesten Stand gebracht wird bevor die Leseoperation durchgeführt wird
3. Lesegruppe wird mit den relevanten lokalen Schreiboperationen aktualisiert
   [$\subseteq WS(x_2)$]

Monotonische Schreibvorgänge:



Wenn ein Client eine Schreiboperation auf einem Server durchführt:

1. Der Server bekommt die Schreibgruppe vom Client, um sicherzugehen, dass alle vorherigen Schreibvorgänge lokal empfangen wurden
2. Wenn nicht, kontaktiert er die anderen Server, um diese abzufragen.
   Kann einige Zeit in Anspruch nehmen!
3. Lesegruppe wird mit ID(x_2) aktualisiert

Zusammenfassung

- Replikation ist ein Mechanismus, um die Performance (Verfügbarkeit, Skalierbarkeit) und Fehlertoleranz zu verbessern
- Das große Problem: Konsistenz
- Für Systeme mit verschiedenen Anforderungen haben wir verschiedene Konsistenzmodelle definiert
    - Um verschiedene Konsistenzmodelle zu implementieren, brauchen wir verschiedene Inhaltsverteilungs- und Konsistenzprotokolle
- **Vor allem müssen wir die Auswirkungen der Anwendung verschiedener Protokolle und Techniken verstehen, korrekte technische Trade-Offs machen und Entscheidungen zum Aufbau eines effizienten Systems entwerfen**

Sicherheit

Einführung

- Der Begriff der **Sicherheit** überschneidet sich mit anderen gewünschten Eigenschaften verteilter Systeme
- Die wichtigsten Aspekte von Sicherheit sind **Integrität** und **Vertraulichkeit**

Eigenschaft	**Beschreibung**
Vertraulichkeit	Keine unautorisierte Offenlegung von Informationen
Integrität	Keine versehentlichen oder bösartigen Änderungen der Informationen wurden durchgeführt (auch nicht von autorisierten Entitäten)

Modelle und Akteure

Subjekt:

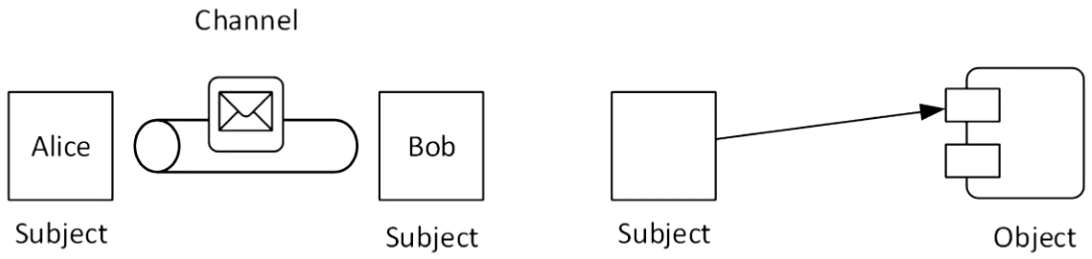
Entität, die dazu fähig ist, mit anderen Subjekten zu kommunizieren ODER Operationen auf ein Objekt aufzurufen

Kanal:

Kommunikationsmedium – Befördert Anfragen und Antworten (Nachrichten oder Aufrufe) zwischen Subjekten oder zwischen einem Subjekt und einem Objekt

Objekt (Server):

Entität, die Dienste für Subjekte offenlegt/bereitstellt



Sicherheitsbedrohungen

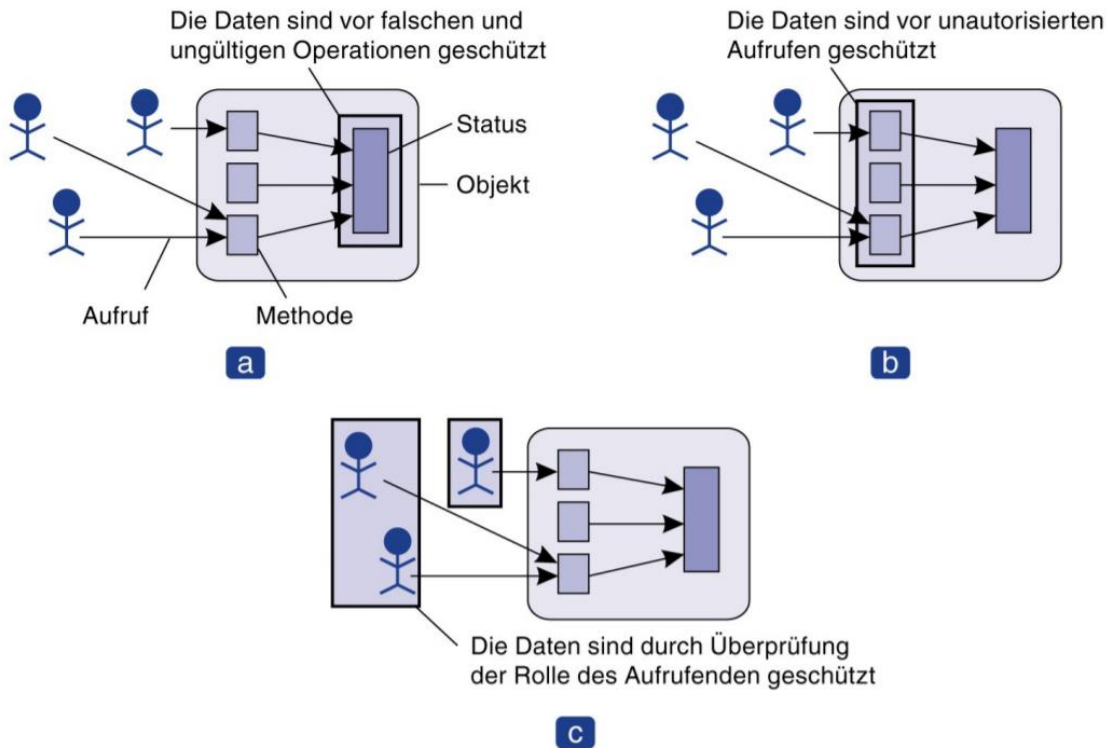
Bedrohung	Kanal	Objekt/Server
Abfangen	Lesen des Inhalts von übertragenen Nachrichten	Lesen der Daten, die in einem Objekt/Server enthalten sind
Unterbrechung	Verhindern der Nachrichtenübertragung	Denial of Service
Änderung	Ändern des Nachrichteninhalts	Ändern der gekapselten Daten von einem Objekt/Server
Herstellung	Einfügen von Nachrichten	Spoofing eines Objektes/Servers

Sicherheitsmechanismen

Um sich vor Sicherheitsbedrohungen zu schützen, haben wir eine Mehrzahl von Sicherheitsmechanismen zur Verfügung:

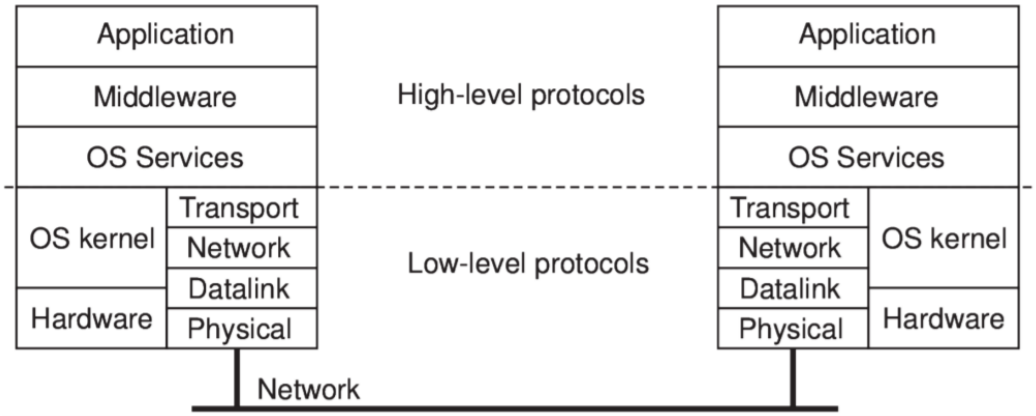
- **Verschlüsselung**: Umändern der Daten in etwas, das der Angreifer nicht versteht (Vertraulichkeit). Dies wird auch benutzt, um zu überprüfen, ob etwas verändert wurde (Integrität)
- **Authentifizierung**: Überprüfung der Behauptung eines Subjekts, dass es ist, was es sagt, das es ist. Überprüfung der **Identität** eines Subjekts (**Wer greift zu**/fragt an?)
- **Autorisierung**: Herausfinden, ob ein Subjekt die Erlaubnis hat, einen gewissen Dienst zu benutzen (**Wer ist erlaubt**, auf einen Dienst zuzugreifen oder ihn anzufragen)
- **Auditierung**: Ermitteln, welche Subjekte auf was zugreifen und wie sie dies tun. Nur dann nützlich, wenn es beim Fangen des Angreifers helfen kann (Angreifer wird versuchen das Hinterlassen von Spuren zu vermeiden)

Design-Problem: Fokus der Kontrolle



Design-Problem: Schichtung der Mechanismen

Auf welcher logischen Ebene werden wir die Sicherheitsmechanismen implementieren?

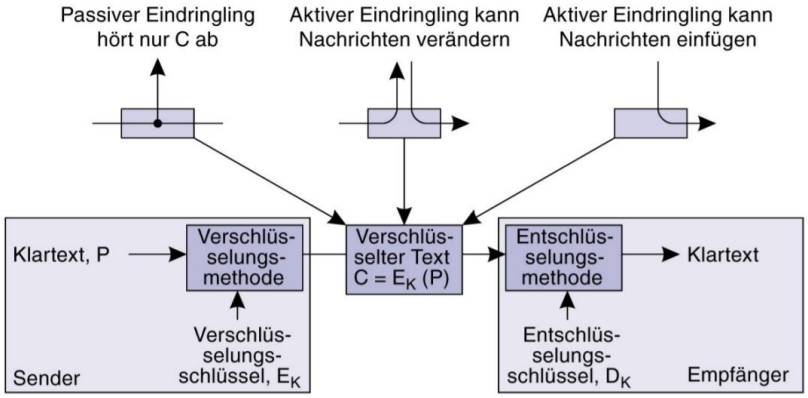


Fundamentale Sicherheitsgesetze

Wichtig: Die Sicherheit jedes verteilten Systems ist genau so gut wie seine schwächste Komponente/Schnittstelle.

Die Sicherheit deines Systems musss von technischen und mathematischen Fakten abhängen und niemals von versteckten Informationen.

Kryptographie



Methoden:

- **Symmetrisches System**: Ein einzelne Schlüssel wird benutzt, um zu verschlüsseln und zu entschlüsseln. Dazu müssen der Sender und der Empfänger sich einen Geheimschlüssel (Secret Key) **teilen** (z.B. DES, AES) $P = D_K(E_K(P))$
- **Asymmetrisches System**: Verschiedene Schlüssel werden benutzt, um zu verschlüsseln und zu entschlüsseln, wobei einer **privat** (K^-_A) ist und der andere **öffentlich** (K^+_A) (z.B. RSA) $P = D_{K+}(E_{K-}(P))$
- **Hashing-System**: Nur die Daten verschlüsseln und einen Digest mit fixer Länge erzeigen. Es gibt keine Entschlüsselung; nur Vergleich ist möglich (z.B. MD5)

Anwendungsfälle:

- **Symmetrisches System**: Verschlüsselung (Abfangprävention)
- **Asymmetrisches System**: Authentifizierung (Herstellungsprävention)
- **Hashing-System**: Integrität (Änderungsprävention)

Kryptgraphische Funktionen

Die kryptographische Funktion: $E_K(m_{in}) = m_{out}$

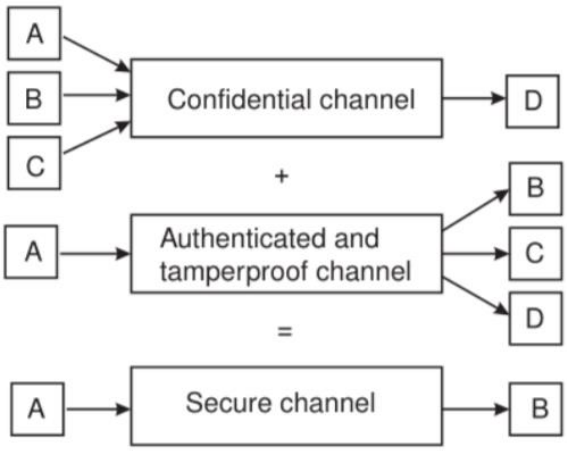
Essenzielle Eigenschaften:

- Die Verschlüsselungsmethode *E* wird öffentlich gemacht, aber die komplette Verschlüsselung wird parametrisiert mittels einem **Schlüssel** *S* (Dasselbe für Entschlüsselung)
- **Einwegfunktion**: Gegeben eine Ausgabe m_{out} von E_k, ist es (analytisch oder) rechnerisch nicht realisierbar m_{in} zu finden
- **Schwache Kollisionsresistenz**: Gegeben ein Paar $\{m, E_k(m)\}$, ist es rechnerisch nicht realisierbar ein $m^* =/= m$ zu finden, sodass $E_K(m^*) = E_K(m)$
- **Schwache Kollisionsresistenz**: Es ist rechnerisch nicht realisierbar zwei verschiedene Eingaben m^* und m zu finden, sodass $E_K(m^*) = E_K(m)$
- **Einwegschlüssel**: Gegeben eine verschlüsselte Nachricht m_{out}, Nachricht m_{in} und eine verschlüsselte Funktion *E*, ist es analytisch und rechnerisch nicht realisierbar einen Schlüssel K zu finden, sodass $m_{out} = E_k(m_{in})$
- **Schwache Schlüsselkollisionsresistenz**: Gegeben eine Dreiergruppe $\{m, K, E\}$, ist es rechnerisch nicht realisierbar ein $K^* =/= K$ zu finden, sodass $E_{K^*}(m) = E_K(m)$
- **Starke Schlüsselkollisionsresistenz**: Es ist rechnerisch nicht realisierbar zwei verschiedene Schlüssel *K* und *K** zu finden, sodass für alle m: $E_k(m^*) = E_k(m)$

Sichere Kanäle

Was ist ein sicherer Kanal?

- Beide Parteien wissen, wer auf der anderen Seite ist (authentifiziert)
- Beide Parteien wissen, dass die Nachricht nicht verändert werden kann (Integrität)
- Beide Parteien wissen, dass Nachrichten nicht entweichen können (Vertraulichkeit)



Authentifizierung vs. Integrität

Wichtig:

Authentifizierung und Datenintegrität bauen aufeinander auf: Betrachten wir einen aktiven Angriff von Trudy auf die Kommunikation von Alice zu Bob.

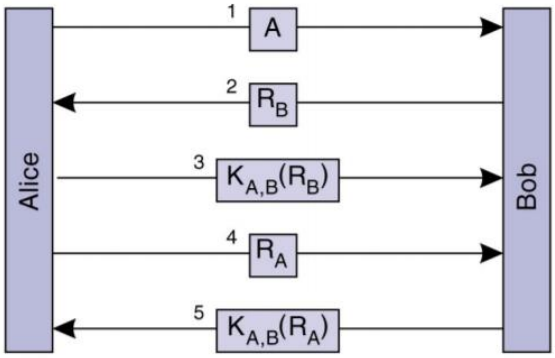
Authentifizierung ohne Integrität:

Alices Nachricht wird authentifiziert und abgefangen von Trudy, die den Inhalt der Nachricht verändert, aber den Authentifizierungsteil unverändert. Authentifizierung ist nun bedeutungslos.

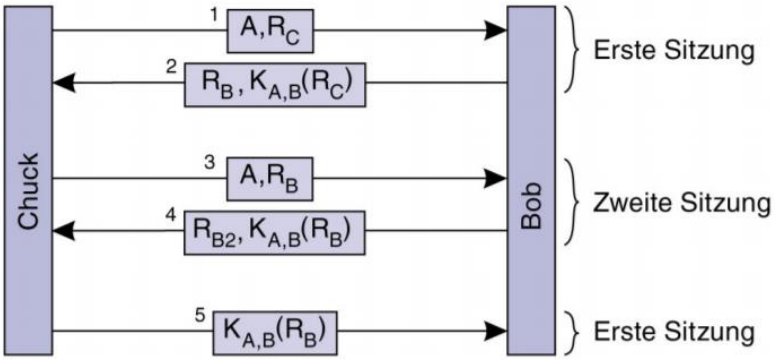
Integrität ohne Authentifizierung:

Trudy fängt eine Nachricht von Alice ab und lässt Bob in dem glauben, dass der Inhalt wirklich von Alice gesendet wurde. Integrität ist nun bedeutungslos.

Geheime (geteilte) Schlüssel



Geheime Schlüssel Reflexionsangriff

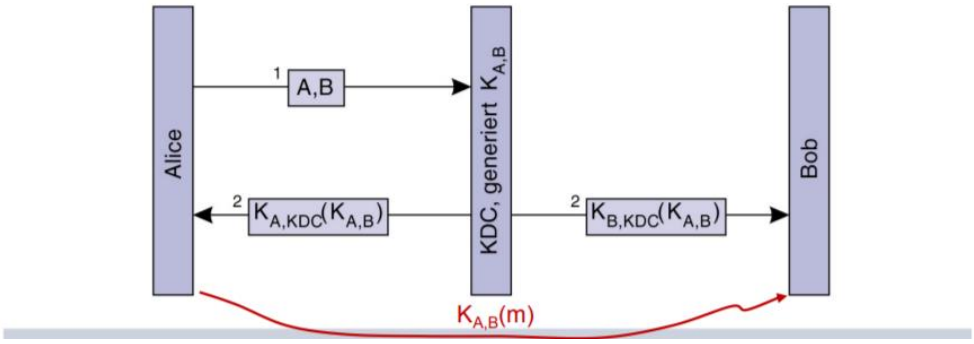


Erkenntnisse:

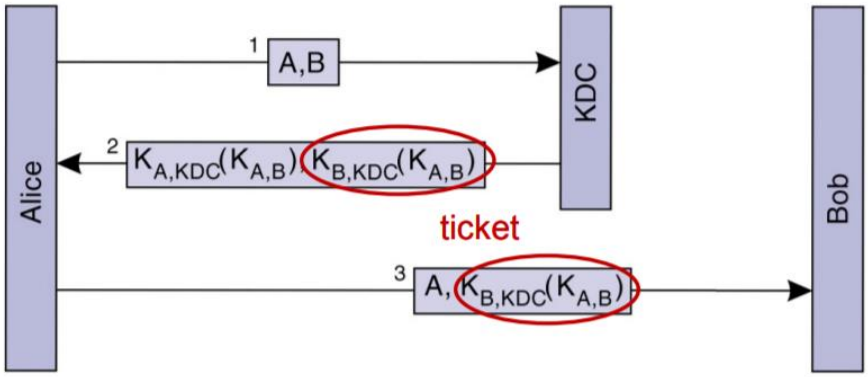
- Niemals wertvolle Informationen an unbekannte Parteien geben
- Auch wenn Alice gerad eund Bob ungerade Challenges benutzt, das Protokoll ist immer noch anfällig für Man-In-The-Middle Attacken

Key Distribution Center

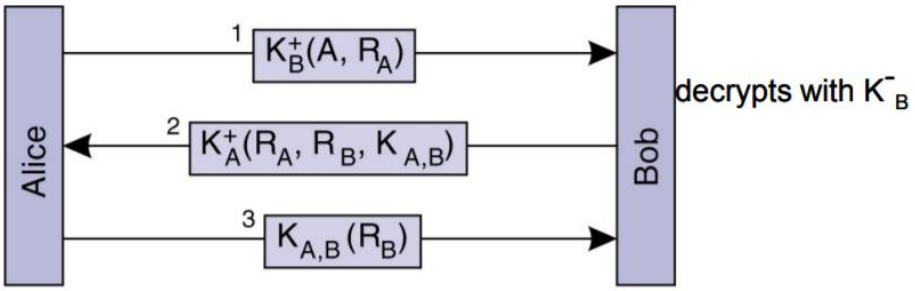
Mit N Subjekten müssen wir N(N – 1)/2 Schlüssel verwalten, wobei jedes Subjekt N – 1 Schlüssel kennt → Benutzen eines vertrauenswürdigen **Key Distribution Centers**, welches Schlüssel generiert falls nötig



Key Distribution Center mit Tickets



Öffentlich-Private Schlüsselauthentifizierung



Vertraulichkeit

Lösungen:

- **Geheimer Schlüssel**: Einen geheimen Schlüssel, um alle Nachrichten, die zwischen Alice und Bob gesendet werden, zu verschlüsseln und zu entschlüsseln
- **Öffentlicher Schlüssel**: Wenn Alice eine Nachricht m zu Bob sendet, verschlüsselt sie diese mit Bobs Öffentlichem Schlüssel: $K^+_B(m)$

Probleme mit Schlüsseln:

- **Schlüssel nutzen sich ab:** Je mehr Daten von einem einzelnen Schlüssel verschlüsselt, desto einfacher wird es, diesen Schlüssel zu finden → **Schlüssel nicht zu oft verwenden**
- **Wiederholungsgefahr**: Denselben Schlüssel für verschiedene Kommunikationssitzungen zu benutzen, erlaubt es alte Nachrichten in die momentane Sitzung einzufügen → **Schlüssel nicht für verschiedene Sitzungen verwenden**
- **Gefährdete Schlüssel**: Wenn ein Schlüssel gefährdet ist, kann man diesen niemals wieder benutzen. Das ist sehr schlecht, wenn alle Kommunikationen zwischen Alice und Bob immer auf demselben Schlüssel basieren → **Nicht denselben Schlüssel für verschiedene Sachen verwenden**
- **Temporäre Schlüssel**: Nicht vertrauenswürdige Komponenten können möglicherweise nur einmal mitspielen, aber man würde niemals wollen, dass diese zu jeder Zeit Wissen über deinen wirklich guten Schlüssel haben → **Schlüssel wegwerfbar machen**

Essenz:

Wertvolle und teure Schlüssel nicht für die ganze Kommunikation verwenden, sondern nur zu Authentifizierungszwecken

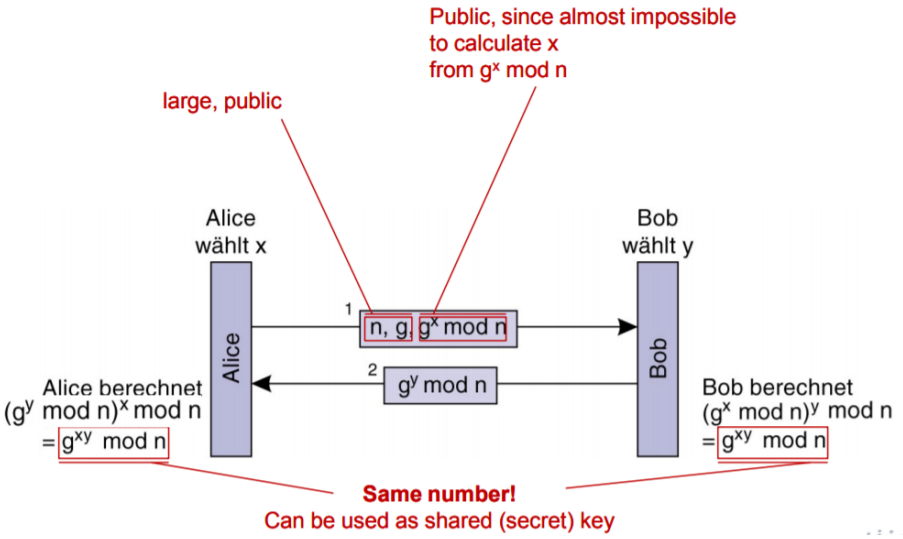
Konsequenz:

Einen „günstigen“ **Sitzungsschlüssel** einführen, der nur während einer einzigen Konversation oder Verbindung benutzt wird („günstig“ heißt auch effizient in Verschlüsselung und Entschlüsselung: in RSA 100x-1000x langsamer als DES).

Schlüsselaustausch: Diffie-Hellman

Wir können geheime Schlüssel auf sichere Weise herstellen, ohne einem Dritten vertrauen zu müssen (d.h. ein KDC):

- Alice und Bob müssen sich auf zwei große Zahlen einigen, *n* (primär) und *g*. Beide Zahlen können öffentlich sein.
- Alice wählt eine große Zahl *x* und behält sie für sich. Bob macht das gleiche, sagen wir *y*.



- Wie sicher ist das?
  - Die Schwere des „Diskreten Logarithmusproblems“ garantiert Vertraulichkeit…
  - …**wenn** wir wissen mit wem wir reden
- Wie kann ein MITM-Angriff stattfinden?
  - Eindringling sitzt ind er mitte:
    - Antwortet Alice mit g^k mod n
    - Sendet Bob: n, g, g^z mod n
    - Erhält 2 Sitzungsschlüssel: g^{xk} mod n, g^{yz} mod n

Schlüsselverteilung

Authentifizierung benötigt kryptographische Protokolle, diese wiederrum benötigen Sitzungsschlüssel, um sichere Kanäle zu schaffen. Wer ist verantwortlich dafür, Schlüssel auszuhändigen?

- **Geheime Schlüssel**
  - Eigenen erstellen und außerhalb der Bandbreite austauschen
  - Einem Key Distribution Center (KDC) vertrauen und dieses nach einem Schlüssel fragen
- **Öffentliche Schlüssel**: Wie garantiert man, dass As öffentlicher Schlüssel auch wirklich von A ist?
  - Persönlich ausgetauscht außerhalb der Bandbreite
  - Eine **Zertifikationsstelle** (CA) verwenden, um öffentliche Schlüssel auszuhändigen. Ein öffentlicher Schlüssel wird in ein **Zertifikat** gepackt, signiert von der CA.

Digitale Signaturen

Anforderungen:

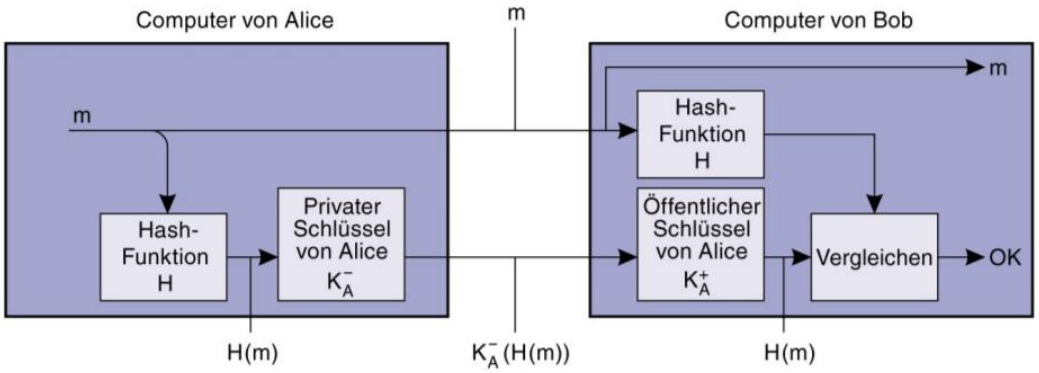
- **Authentifizierung**: Empfänger kann die behauptete Identität des Senders überprüfen
- **Nicht-Ablehnung**: Der Sender kann später nicht ablehnen, dass er/sie die Nachricht gesendet hat
- **Integrität**: Die Nachricht kann während oder nach dem Empfang nicht böswillig verändert werden

Lösung:

Ein Sender signiert alle übermittelten Nachrichten, sodass die Signatur überprüft werden kann UND die Nachricht und Signatur eindeutig verbunden sind

Grundidee:

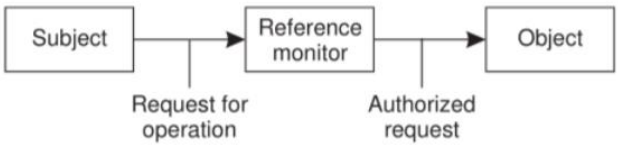
Authentifizierung und Geheimhaltung nicht vermischen. Stattdessen sollte es auch möglich sein, eine Nachricht im Klartext zu schicken, aber diese ebenfalls signieren zu lassen → Einen Message-Digest nehmen und diesen signieren



Autorisierung vs. Authentifizierung

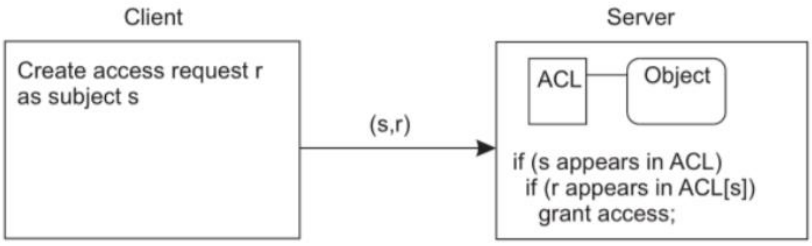
- **Authentifizierung**: Überprüfen, dass ein Subjekt ist, was es behauptet zu sein: Überprüfen der **Identität** eines Subjekts
- **Autorisierung**: Herausfinden, ob ein Subjekt die Erlaubnis für gewisse Dienste eines Objektes hat

Autorisierung macht nur dann Sinn, wenn das angefragte Subjekt schon authentifiziert wurde.



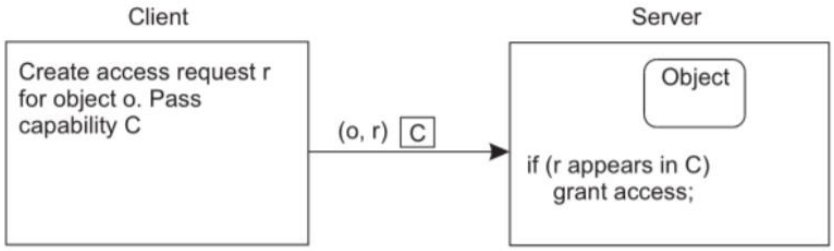
Zugriffskontrolle

Pflegen einer **Zugriffskontrollmatrix** ACM, in der der Eintrag *ACM[S,O]* die erlaubten Operationen enthält, die Subjekt *S* auf Objekt *O* durchführen kann (oft R/W/+/- → lesen/schreiben/administrieren, nichts)



Zugriffskontrollliste (ACL):

Jedes Objekt O erhält eine **Zugriffskontrollliste** (ACL): *ACM[*,P]*, welche die erlaubten Operationen pro Subjekt (oder Gruppe von Subjekten) beschreibt.
z.B.:
[Objekt X] Alice rw+; Bob rw; Chuck -
[Objekt Z] Alice rw; Bob rw+; Chuck -



Fähigkeiten:

Jedes Subjekt hat eine Fähigkeit: ACM[S,*], welche die erlaubten Operationen pro Objekt (oder Kategorie des Objekts) beschreibt

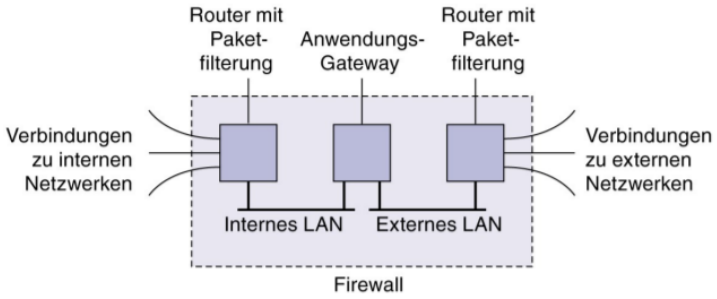
z.B.: [Alice] Object X rw+; Object Y rw+; Object Z rw

Projektdomänen

- **Gruppen**: Benutzer gehören zu einer bestimmten Gruppe, jede Gruppe hat damit verbundene Zugriffsrechte
- **Rollen**: Unterscheiden nicht zwischen Benutzern, sondern nur zwischen den Rollen, die sie spielen können. Deine Rolle wird beim Login festgelegt. Rollen zu wechseln ist erlaubt
- **Zertifikate**: Benutzer stellen Zertifikate zur Verfügung, zu welchen Gruppen/Rollen sie gehören

Firewalls

Manchmal ist es besser, Serviceanfragen auf der niedrigsten Ebene auszuwählen: Netzwerkpakete. Pakete, die nicht gewissen Anforderungen entsprechen, werden einfach vom Kanal entfernt → Schutz durch eine Firewall: Sie implementiert Zugriffskontrolle



Router filtern:

- **Regeln**: Maßnahme bestimmen (erlauben, verweigern), Quellenmuster Adresse/Port, Zielmuster Addresse/Port, +/- Flags
- **Abstimmung**: Regeln in geordneten Sequenzen anwenden und nach passender oder Standardaktion ausführen

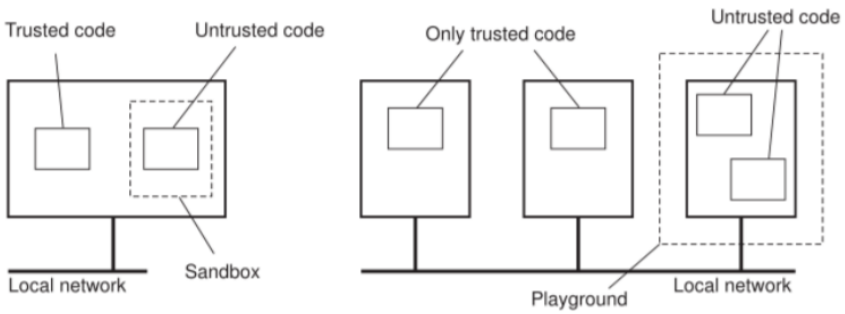
Durchgang auf Anwendungsebene:

- **Paketinspektion**: Inhalt basierend auf Anwendungssemantiken interpretieren
- **Beispiel Mail**: Anhänge mit exe-Dateien nicht zulassen
- **Beispiel Web**: Skripts oder Applets rausfiltern

Sicherer mobiler Code: Schützender Host

Eine einzelne (sehr strenge) Richtlinie erzwingen und diese mittels ein paar einfacher Mechanismen implementieren:

- **Sandkastenmodell**: Richtlinie: Remote-Code darf nur auf eine vordefinierte Sammlung an Ressourcen und Diensten zugreifen. Mechanismus: Instruktionen auf illegalen Speicher- und Dienstzugriff überprüfen
- **Spielplatzmodell**: Gleiche Richtlinie, aber Mechanismus: Code auf verschiedenen „ungeschützten“ Maschinen laufen lassen.



Häufige Angriffsszenarien

Sicherheit von verteilten Systemen kann auf jeder Schicht gefährdet sein, aber vergiss nicht: Jede Sicherheitslücke macht möglicherweise das gesamte System unsicher.

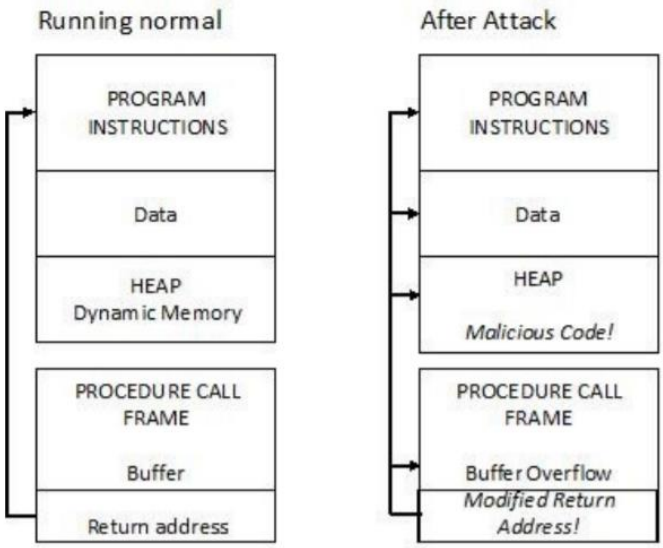
Folgende Angriffe passieren andauernd in der Praxis:

- Buffer Overflows
- SQL-Injection
- Cross-Side-Scripting (XSS)
- Distributed Denial of Service (DDoS)
- Sidechannel-Angriffe
- Social Engineering

Stack Buffer Overflow

Häufiges Sicherheitsproblem in nicht verwalteten Programmiersprachen (z.B. C / C++)

- Eingabedaten größer als reservierter Platz im Stack
- Erlaubt einem Angreifer, den vorhandenen Rückgabeadresszeiger des Prozeduraufrufs mit einer benutzerdefinierten Adresse zu überschreiben, die auf den Schadcode des Angreifers zeigt, welche dieser vorher haufenweise gespeichert hat
- Erlaubt dem Angreifer, arbitären Code auszuführen



SQL-Injection

Manche Webanwendungen überprüfen nicht ausreichend die Daten, die von Benutzern empfangen werden, bevor sie eine SQL-Abfrage durchführen.

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

Jetzt nimm folgende Eingabe an:

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

Und dann erhältst du:

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

Cross-Side Scripting Angriff (XSS)

Manche Webanwendungen überprüfen nicht ausreichend die Daten, die von Benutzern empfangen werden.

- Selbes Prinzip wie bei SQL-Injection
- Erlaubt es dem Angreifer, arbiträre Skripts in eine legitime (vertrauenswürdige) Webseite zu injizieren
- Bleispiel: Blog mit Kommentarfunktion die arbiträren HTML-Code akzeptiert

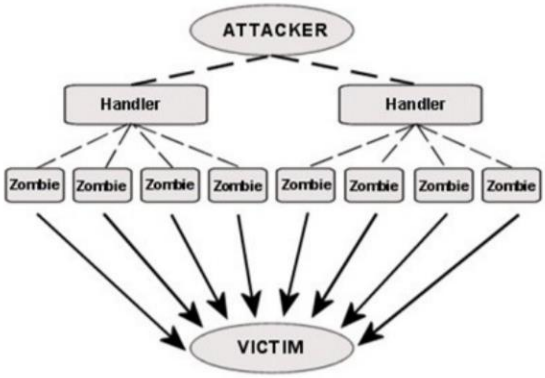
```
Very interesting article!

<script type = "text/javascript">
<!-- window.location="http://62.178.71.105";
-->
</script>
```

Distributed Denial-of-Service (DDos)

Angreifer benutzt ein Netzwerk von gehackten Maschinen

- Bots/Zombies überladen die Ressourcen vom Ziel mit Anfragen
- Schwer sich dagegen zu schützen (muss auf ISP-Ebene gemacht werden)
- Schwer den Angreifer zu identifizieren (Alle Anfragen kommen von nichts ahnenden Zombies)



Sidechannel Angriffe und Social Engineering

Es werden die technischen Sicherheitsmechanismen ignoriert und das Geheimnis herausgefunden, auf dem der Mechanismus basiert:

- Phishing nach Passwörtern und Schlüsseln
- NSA
- Reverse Engineering von Schlüsseln, die in Geräten eingebettet sind, indem der Energieverbrauch gemessen wird

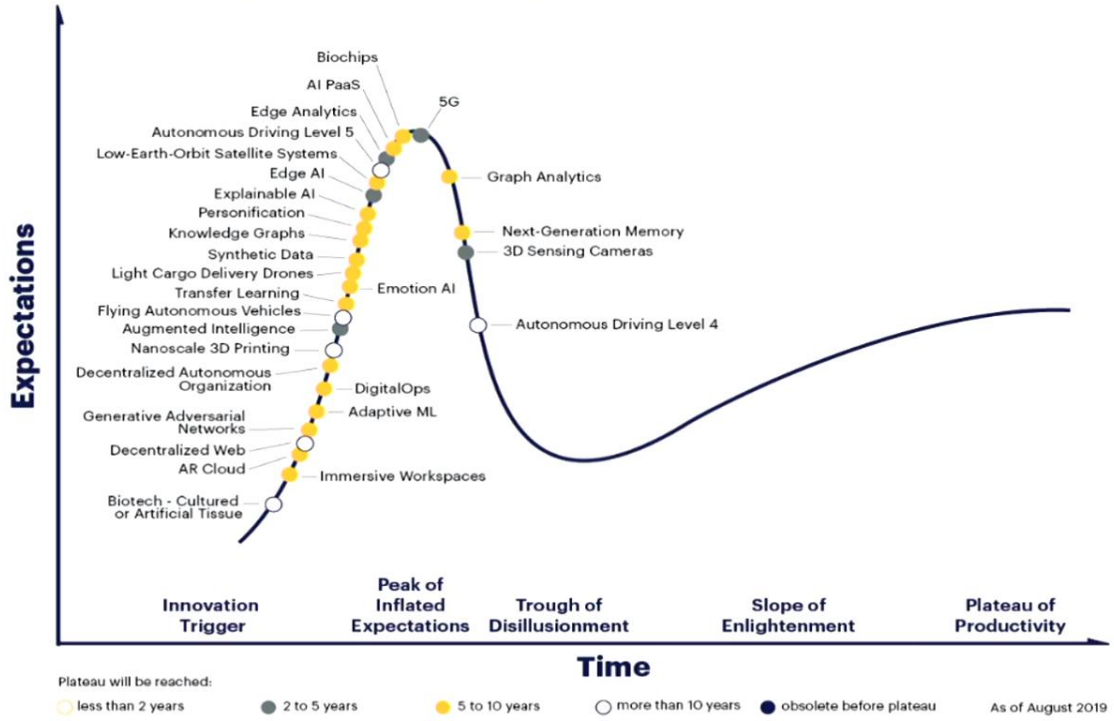
Social Engineering:

Sidechannel Angriffe auf Menschen hinter dem „sicheren“ technischen System

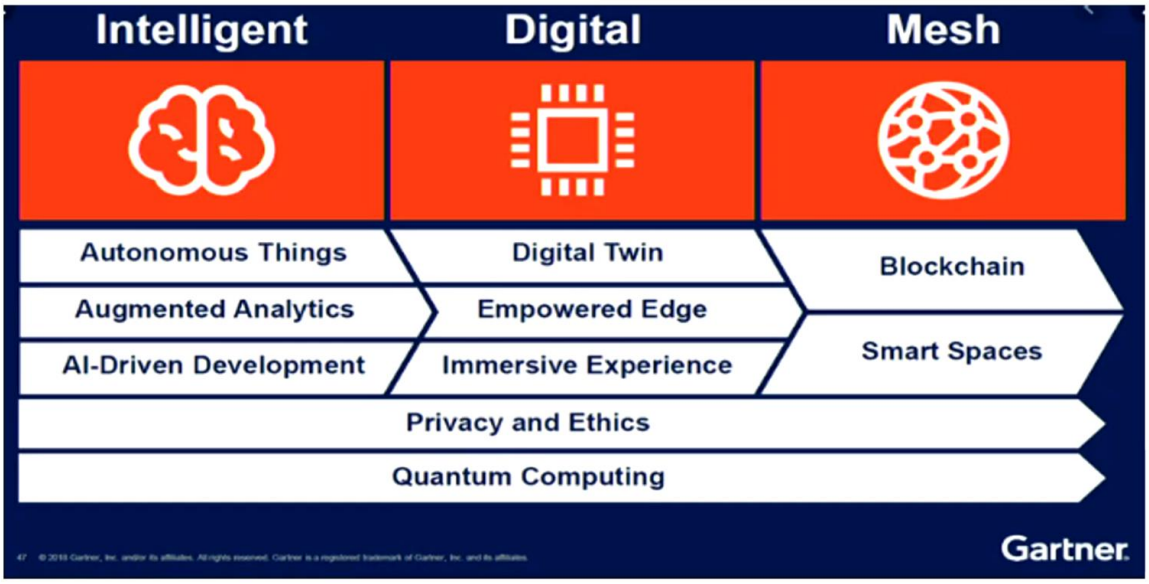
- Übliche Annahme: Menschen sind leicht zu manipulieren
- Z.B.: eingehender Anruf „Hallo, ich bin von der IT-Abteilung, wir haben ein Problem mit ihrem Account“

Momentane Trends in verteilten Systemen

Gartner Hype-Zyklus für entstehende Technologien, 2019



Top 10 Strategische Technologietrends

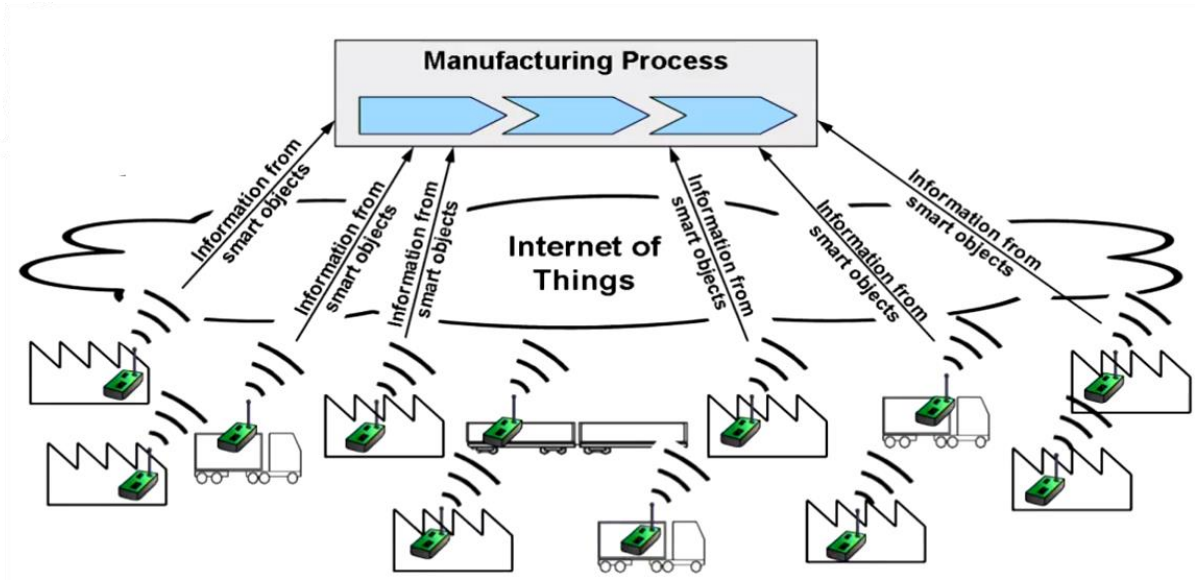


Haupttrends in verteilen Systemen I

- **Internet of Things (IoT)**
  - Physische Gegenstände werden nahtlos in das Informationsnetzwerk integriert
  - Physische Gegenstände werden aktive Teilnehmer in Geschäftsprozessen
  - Physische Gegenstände werden zu „Smart Objects“
  - Technoligen: RFID, Sensornetzwerke, Internet Protokoll Version 6 (IPv6)

IoT – Beispiel: Fabriken der Zukunft

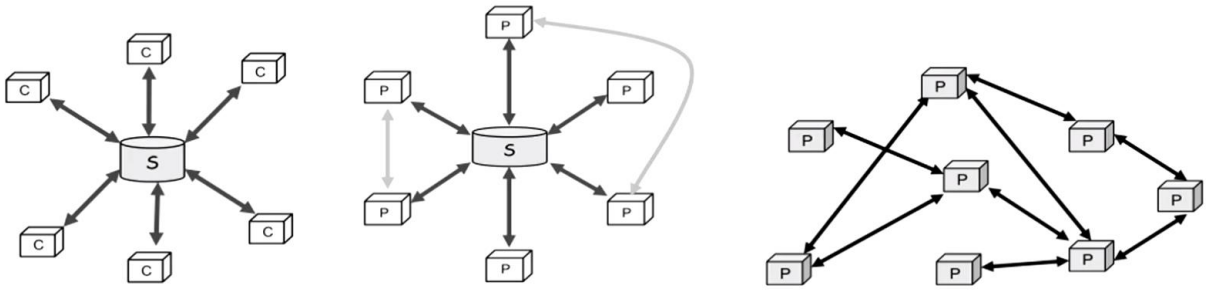
- Verknüpfen der Leistung unabhängiger Fabriken
- Erreichen komplexer Herstellungsprozesse
- Bereitstellen konkreter Werkzeuge für
    - Prozesserstellung
    - Prozessoptimierung
    - Informationsaustausch
- Echtzeitüberwachung



Haupttrends in verteilen Systemen II

- Internet of Services (IoS)
    - Softwaredienste werden über das Internet angeboten
    - Technologien: REST, WSDL, SOAP, WS-„stack“, Microservices
    - Grundlage für Cloud Computing und Edge Computing
- Serviceorientierte Architekturen vs. IoS:
    - IoS = Globale SOA?
    - SOA: Ursprünglich hauptsächlich ein Konzept, IT-Softwarearchitekturen in einem Unternehmen zu organisieren

Peer-to-Peer: Übersicht



(a) Client/Server (b) Hybrid (c) Peer-to-Peer

Peer-to-Peer

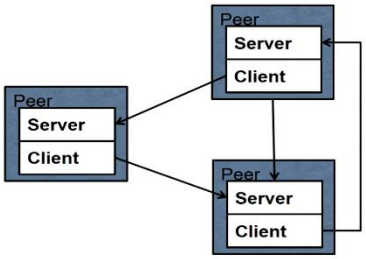
- Komponenten interagieren direkt miteinander als Peers, indem sie Dienste austauschen
- Anfrage-/Antwortinteraktion ohne Asymmetrie, welche im Client-Server-Muster zu finden ist – Alle Peers sind gleich
- Jede Peer-Komponente liefert und konsumiert die gleichen Dienste

Was ist P2P?

- Definition nach Oram et al.:
  - Ein Peer-to-Peer (P2P) System ist ein „selbstorganisierendes System von gleichen, autonomen Entitäten (Peers), welches auf die gemeinsame Nutzung verteilter Ressourcen in einer Netzwerkumgebung zielt, ohne zentrale Dienste nutzen zu müssen."
  - „Ein System mit komplett dezentralisierter Selbstorganisation und Ressourcennutzung"
- Schlüsseleigenschaften eines P2P Systems:
  - Gleichheit – Alle Peers sind gleicht (peer = gleichgestellt)
  - Autonomie – Keine zentrale Kontrolle
  - Dezentralisierung – Keine zentralisierten Dienste
  - Selbstorganisation – Keine Koordination von außen
  - Geteilte Ressourcen – Peers können Ressourcen benutzen, die von anderen Peers bereitgestellt werden

Peers

- Peers
  - Sind Knoten, die in einem P2P-Overlay laufen
  - Haben alle die gleichen Fähigkeiten (in jeder Rolle zu agieren)
  - Können als „Clients" und „Servers" gleichzeitig agieren



Overlay-Netzwerk

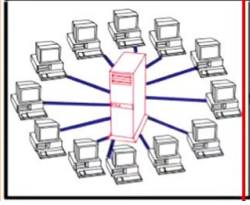
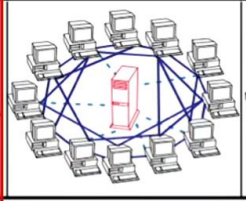
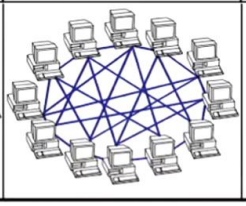
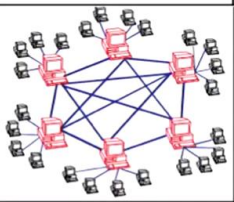
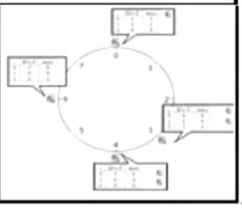
- Zusammengesetzt aus Direktverbindungen zwischen Peers
- Typischerweise ein „darauflegendes" Netzwerk auf einem anderen Netzwerk (z.B. das Internet)
- Aber komplett unabhängig vom physischen Netzwerk aufgrund der Abstraktion der TCP/IP Schicht
- Separates Adressierungsschema

P2P: Anwendungsbereiche

- Einige Anwendungsgebiete
  - VoIP (Skype/FastTrack)
  - Medien-Streaming (Joost)
- Im Jahr 2006 hat P2P 70% des Online-Datenverkehrs ausgemacht (CacheLogic Research):
  - P2P hält für ~19% des Festzugriffsverkehrs in Nordamerika laut Sandvine (2010/11)
  - Bittorent war die größte Anwendung hinsichtlich Upstream-Verkehres in Nordamerika in 2010/11 (52%)
- Offensicht ist File Sharing ein Bereich, wo P2P stark benutzt wird:
  - Napster, Gnutella, Kademlia, etc.

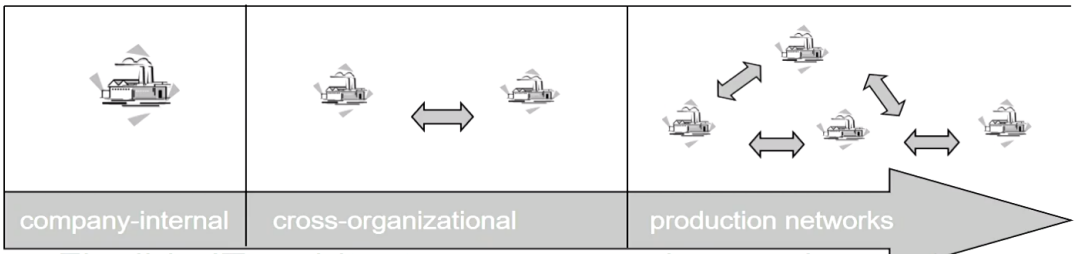
Gründe für die Anwendung von P2P

- Kosten: Rechnen/Speicher kann ausgelagert werden
- Hohe Erweiterbarkeit (Einfaches Hinzufügen weiterer Ressourcen)
- Hohe Skalierbarkeit (System kann zu einer hohen Anzahl an Peers wachsen)
- Fehlertoleranz: Wenn ein Peer ausfällt, wird das Gesamtsystem trotzdem funktionieren
- Widerstand gegen Klagen

Client-Server	**Peer-to-Peer**			
	1. Resources are shared between the peers 2. Resources can be accessed directly from other peers 3. Peer is provider and requestor (Servent concept)			
	Unstructured P2P			**Structured P2P**
	1st Generation		**2nd Generation**	
1. Server is the central entity and only provider of service and content. → Network managed by the Server 2. Server as the higher performance system. 3. Clients as the lower performance system Example: WWW	*Centralized P2P* 1. All features of Peer-to-Peer included 2. Central entity is necessary to provide the service 3. Central entity is some kind of index/group database Example: Napster	*Pure P2P* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities Examples: Gnutella 0.4, Freenet	*Hybrid P2P* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → dynamic central entities Example: Gnutella 0.6, JXTA	*DHT-Based* 1. All features of Peer-to-Peer included 2. Any terminal entity can be removed without loss of functionality 3. → No central entities 4. Connections in the overlay are "fixed" Examples: Chord, CAN
				

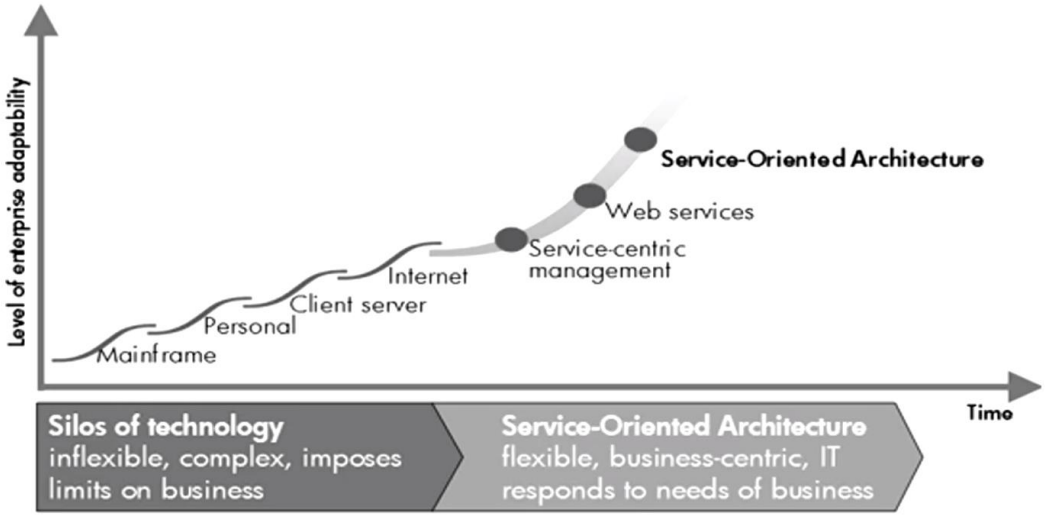
Serviceorientierung: Motivation

- Maßgeblicher Trend seit den 1990ern:
  - Globalisierung, Deregulierung der Märkte
  - Organisationsübergreifende Workflows und Geschäftsprozesse sind von hoher Wichtigkeit
  - Business Process Outsourcing (BPO)
  - Flexibilität von Geschäftsprozessen ist ein wesentlicher Erfolgsfaktor



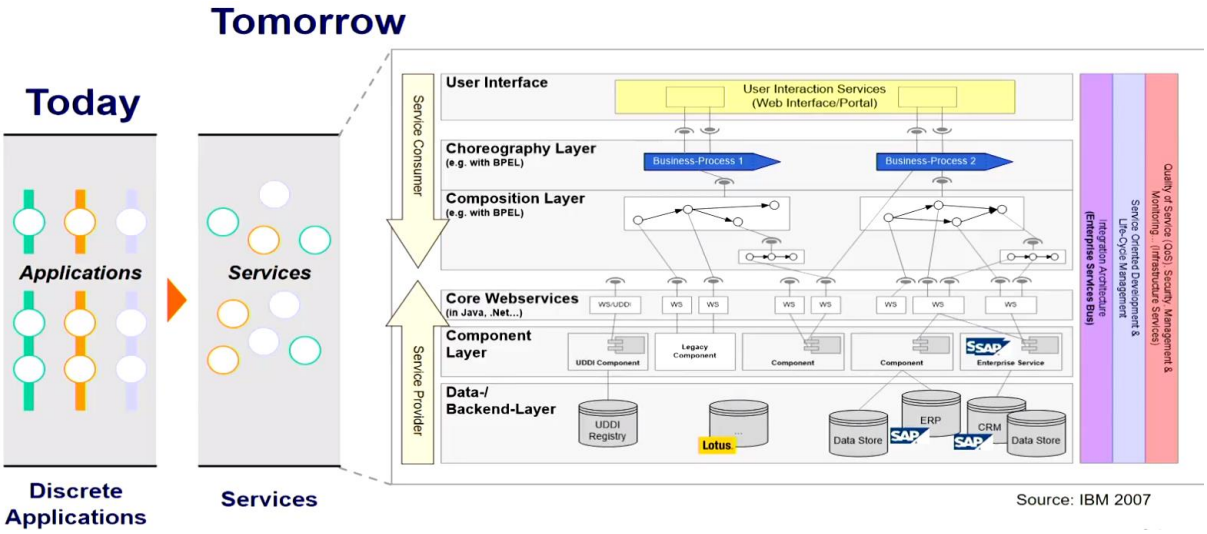
- Flexible IT-Architekturen sind eine wesentliche Anforderung:
  - Integration von Legacy-Systems
  - Kopplung zu IT-Systemen von Geschäftsprozessen

Motivation – Ein Paradigmenwechsel



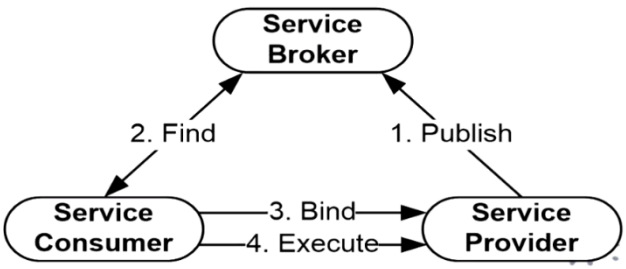
Vorstellung einer serviceorientierten Architektur

„Lose gekoppelt, prozessgesteuerte Dienste und Komponenten“



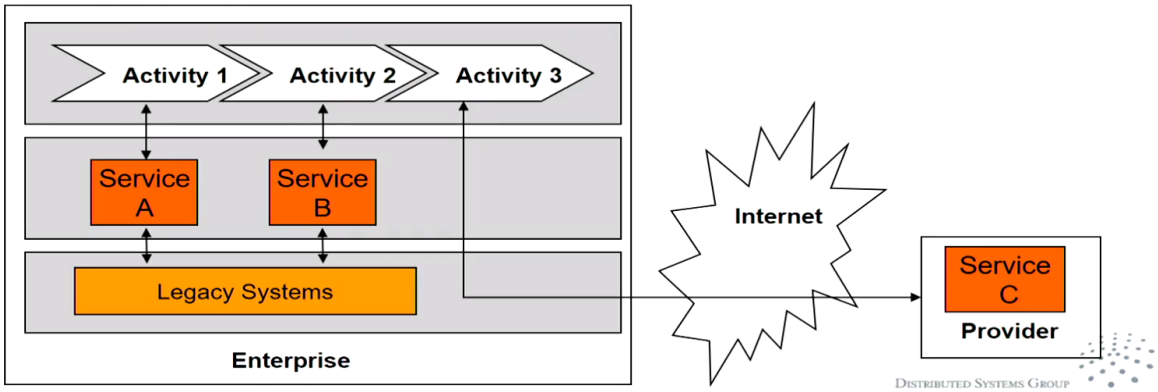
SOA – Übersicht und Rollen

- Serviceorientierte Architekturen:
  - IT-Architektur aus einzelnen Diensten gemacht, d.h. in sich geschlossene Softwarekomponenten mit einer deutlichen Funktionalität
  - Komplexe Applikationen entstehen von der Kopplung von einzelnen Diensten z.B.:
    - Servicebasierte Workflows
    - Mashups
  - Jedoch ist es auch möglich, einzelne Dienste aufzurufen
- Rollen in einer serviceorientierten Architektur
  - Service Provider
  - Service Consumer
  - Vermittler (optional), z.B. Service Broker



Workflows und Dienste

- Workflows und Dienste:
  - Workflows sind IT-fähige Geschäftsprozesse
  - Dienste können zu Workflows zusammengesetzt werden (2-Ebenen-Programmierung)
  - Dienste verpacken Funktionalität von Legacy-Systemen (z.B. Service A/B)
  - Integration externer Dienste (z.B. Service C)
- Dienste unterstützen schnelle Zusammensetzung von verteilten Workflows



Motivation – Willst du Milch?

Kaufe eine Kuh:

- Hohe Vorabinvestition
- Hohe Wartungskosten
- Produziert mehr oder weniger eine fixe Menge an Milch
- Schrittweise (diskrete) Skalierung

Kaufe eine Flasche Milch:

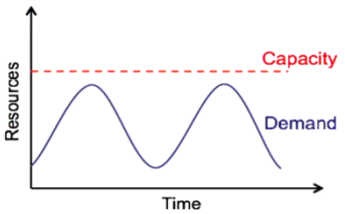
- Bezahlen pro Benutzung
- Geringere Wartungskosten
- Lineare (kontinuierliche) Skalierung
- Fehlertolerant

Anwendungsfälle für Cloud Computing

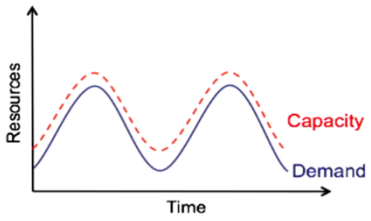
- Nachfrage nach einem sich im Laufe der Zeit variierenden Service
  - Z.B. Spitzenbelastung
- Nachfrage ist im Vorhinein unbekannt
  - Z.B. neues Startup-Unternehmen
- Batch-Analyse
  - Z.B. 1000 EC2 Instanzen für eine Stunde kosten genauso viel wie eine Instanz für 1000 Stunden

Traditionelle Datenzentren vs. Cloud

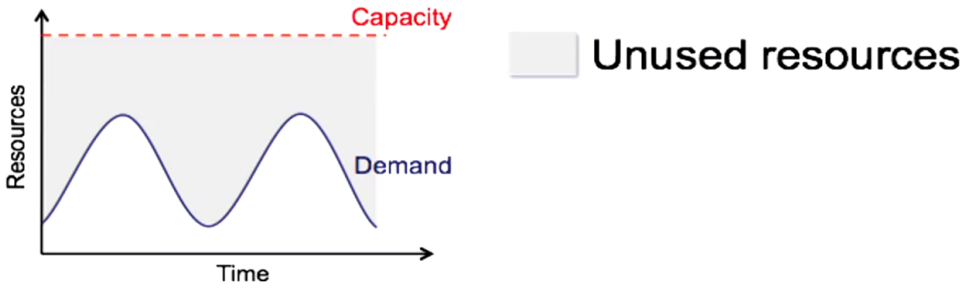
- Traditionelles Datenzentrum



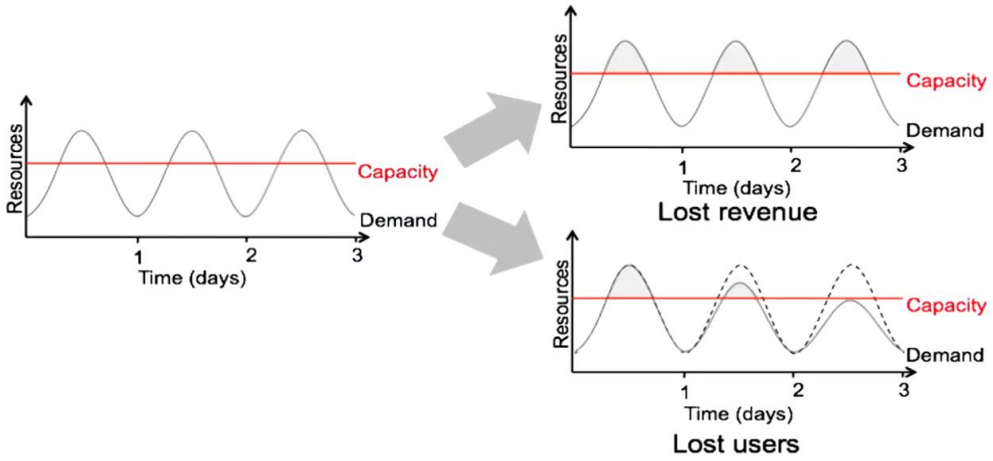
- Virtuelles Datenzentrum in der Cloud



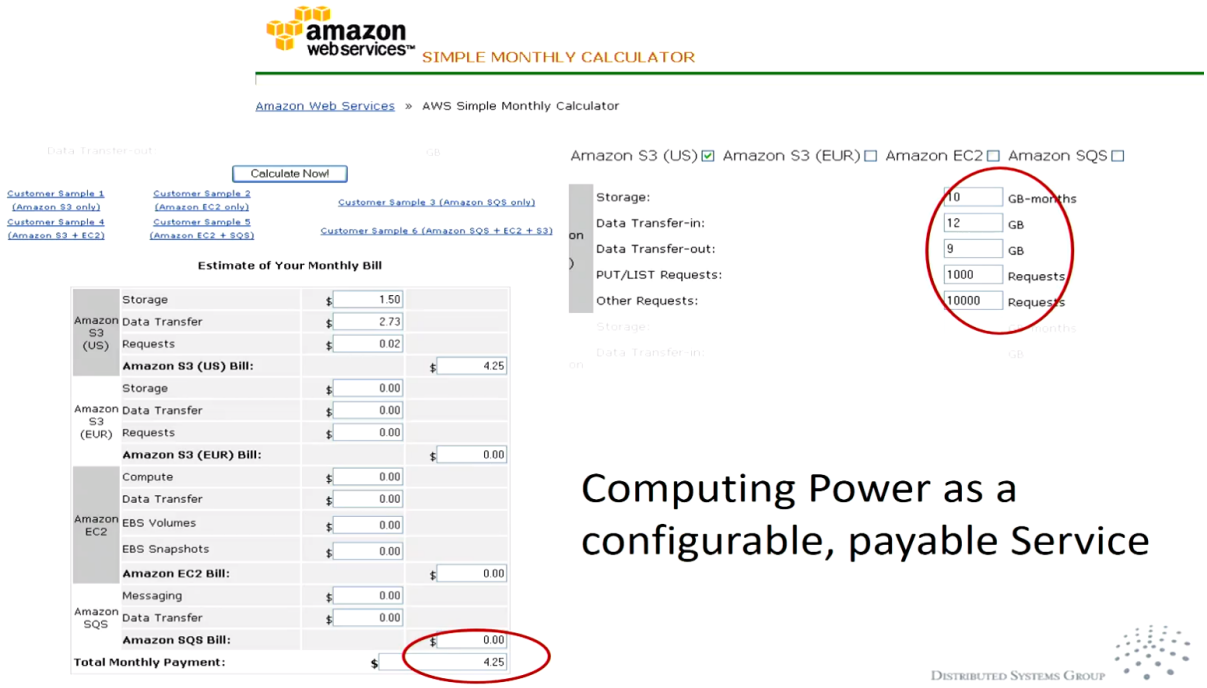
Risiko der Überprovisionierung



Risiko der Unterprovisionierung



Cloud Computing



Definition

- Gemäß des National Institute of Standard and Technology (NIST):
  - Selbstdienste auf Nachfrage: Schnell automatisierte Vermietung von Kapazität mittels Webschnittstellen
  - Breiter Netzwerkzugriff
  - Ressourcenzusammenlegung: Benutzen von Virtualisierungstechniken
  - Schnelle Elastizität: Virtuell unbegrenzte Kapazität und Skalierbarkeit
  - Gemessener Dienst: pay-as-you-go

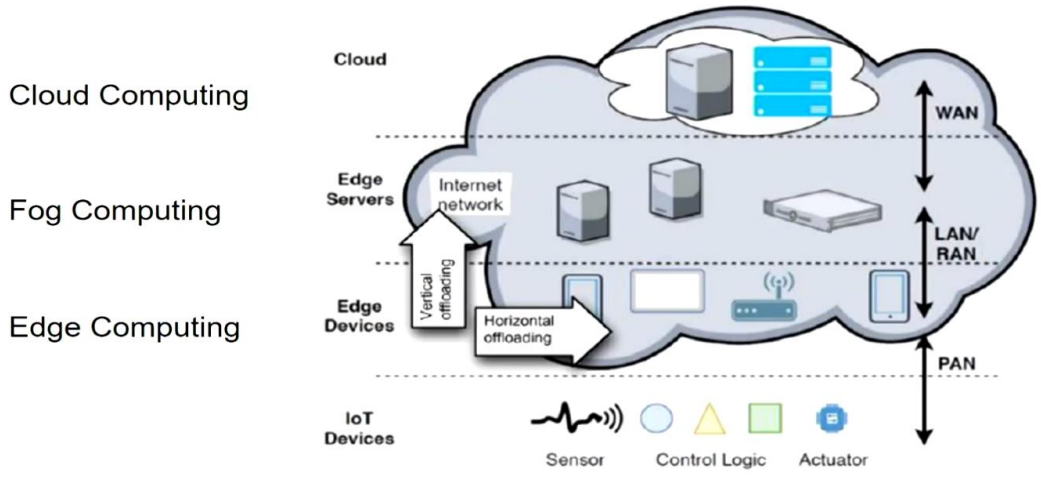
NIST – 3 Servicemodelle

- Cloud-Infrastruktur als ein Service (**IaaS**)
  - Liefern von Computerinfrastruktur als ein Service (Virtuelle Maschinen, Speicher, etc.)
  - Beispiel: Amazon EC2, Amazon S3
- Cloud Plattform als ein Service (**PaaS**)
  - Liefern einer Rechnungsplattform und eines Lösungsstapels als ein Service (Ausführungsumgebung/-framework)
  - Beispiel: Google App Engine
- Cloud Software als ein Service (**SaaS**)
  - Beispiel: ERP Software als ein Service, salesforce.com

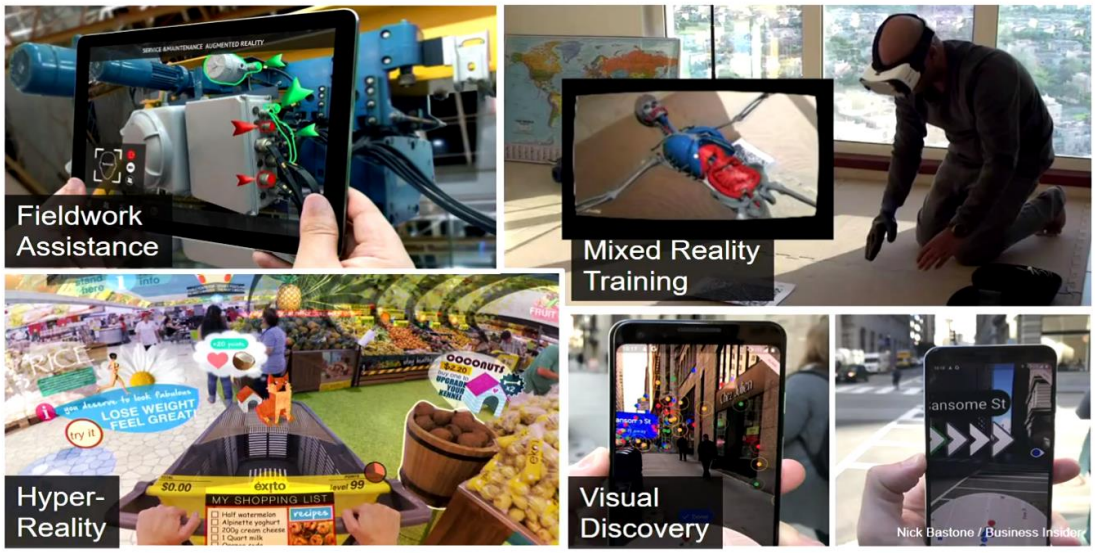
NIST: 4 Bereitstellungsmodelle

- **Private Cloud**: Nur betrieben für ein einziges Unternehmen
- **Gemeinschafts-Cloud**: Geteilt von vielen Unternehmen
- **Öffentliche Cloud**: Für die breite Öffentlichkeit zugänglich, in Besitz von einem Unternehmen, welches Cloud-Dienste verkauft
- **Hybrid-Cloud**: Zusammensetzung von zwei oder mehr Cloud-Bereitstellungsmodellen (Privat, Gemeinschaft, oder Öffentlich)

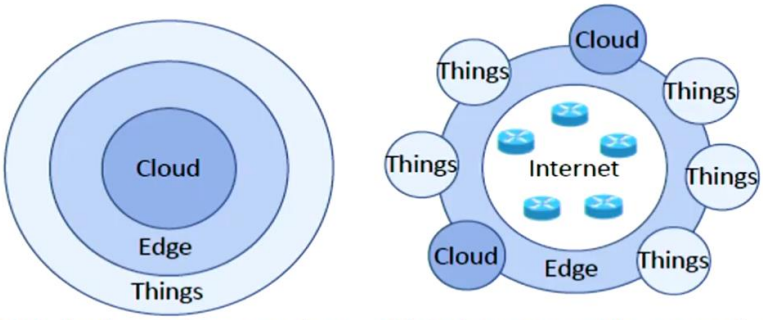
Vertikale vs. Horizontale Randarchitektur



Edge Computing für intelligente Augmentation



Perspektiven auf IoT: Edge, Cloud, Internet



(a) A cloud-centric perspective: Edge as "edge of the cloud"

(b) An Internet-centric perspective: Edge as "edge of the Internet"

Cloud-zentrierte Perspektive

Annahmen

- Cloud bietet Kerndienste; Edge bietet lokale Proxys für die Cloud (Teile der Arbeitsbelastung der Cloud entlasten)

Edge Computer

- Spielen eine unterstützende Rolle für die IoT Dienste und Anwendungen
- IoT Lösungen, die auf Cloud Computing basieren, benutzen Cloud Server für verschiedene Zwecke wie massive Berechnung, Datenspeicherung, Kommunikation zwischen IoT Systemen und Sicherheit/Privatsphäre

Verfehlung

- In der Netzwerkarchitektur liegt die Cloud auch am Netzwerkrand, nicht umgeben von Edge Computern am Rand, welche nicht immer auf die Cloud angewiesen sind; sie können autonom operieren und kollaborieren direkt miteinander ohne die Hilfe der Cloud

Internet-zentrierte Perspektive

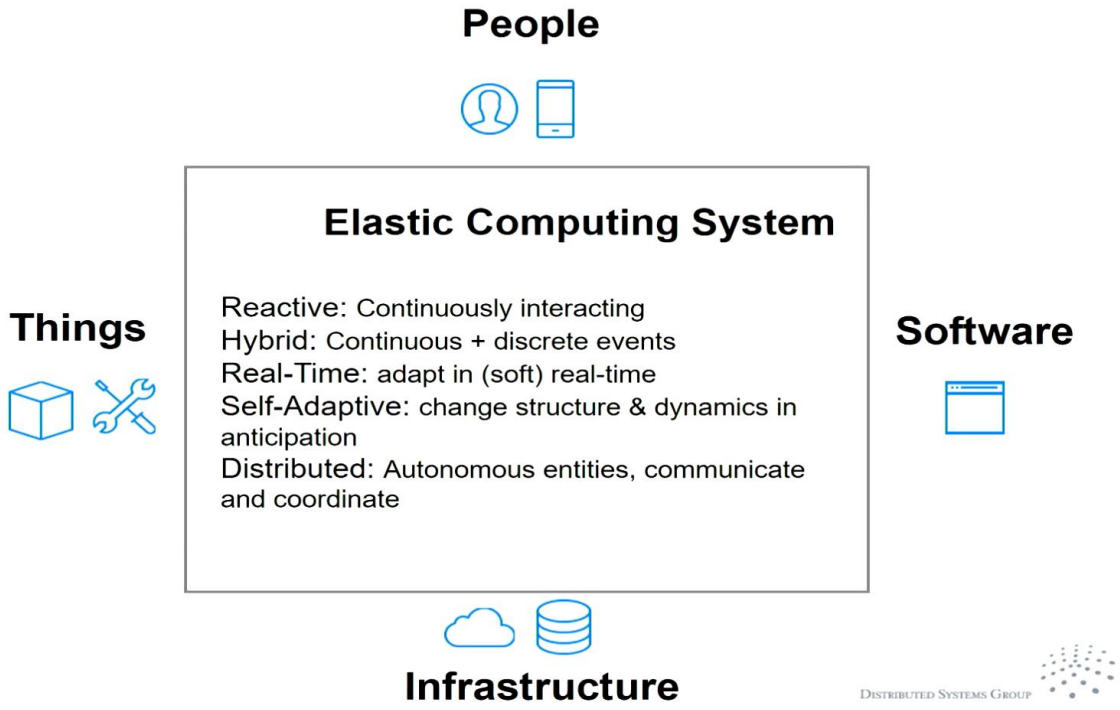
Annahmen

- Internet ist das Zentrum der IoT Architektur; Randgeräte sind Gateways zum Internet (nicht die Cloud)
- Jedes LAN kann autonom um Randgeräte organisiert werden
- Lokale Geräte hängen nicht von der Cloud ab

Deshalb

- Dinge gehören eher zu partitionierten Subsystemen und LANs, als direkt zu einem zentralisierten System
- Die Cloud ist über den Rand des Netzwerks mit dem Internet verbunden
- Entfernte IoT Systeme können direkt über das Internet verbunden werden. Kommunikationen müssen nicht über die Cloud verlaufen
- Der Rand kann Dinge zum Internet verbinden und den Verkehr außerhalb des LANs trennen, um Dinge zu schützen → IoT System muss in der Lange sein autonom zu handeln

Ökosystem – Bausteine



Elastizität =/= Skalierbarkeit