Verteilte Systeme

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2017/18

Inhalt

- Einführung
  - Motivation
  - Zielsetzung
  - Konzepte
- Programmiermodelle und Entwurfsmuster
  - Nachrichtenbasiert: Sockets
  - Auftragsorientiert: RPC
  - Objektorientiert: Java RMI, CORBA
  - Web-Services
- Architekturmodelle
  - Client/Server-Strukturen
  - Peer-To-Peer Netzwerke

Inhalt

- Verteilte Algorithmen
  - Logische Ordnung von Ereignissen
  - Wechselseitiger Ausschluss (Konkurenzdienst)
  - Wahlalgorithmen
  - Konsensalgorithmen
- Dienste
  - Namensdienst (DNS, DCE Directory Service)
  - Verzeichnisdienst (LDAP)
  - File-Dienst (NFS, DFS) und Replikation
  - Transaktionsdienst (2 Phase Commit Protocol)
  - Zeitdienst (NTP)
  - Sicherheit (SSL, Kerberos)

Literatur

Verteilte Systeme:

- Günter Bengel: Verteilte Systeme. Vieweg Verlag.
- G. Coulouris, J. Dollimore, T. Kindberg: Verteilte Systeme. Pearson Studium.
- Ulrike Hammerschall: Verteilte Systeme und Anwendungen. Pearson Studium.
- A. Schill, T. Springer: Verteilte Systeme. Springer Verlag.
- A.S. Tanenbaum, M. van Steen: Verteilte Systeme. Pearson Studium.

Entwurfsmuster:

- D. Schmidt, M. Stal, H. Rohnert, F. Buschmann: Pattern-Oriented Software Architecture. John Wiley & Sons, Ltd.
- Floyd Marinescu: EJB Design Patterns. John Wiley & Sons, Inc.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Entwurfsmuster. Addison-Wesley.
- E. Freeman, E. Freeman mit K. Sierra und B. Bates: Entwurfsmuster von Kopf bis Fuß. O'Reilly.

Literatur

Verschiedenes:

- Claudia Eckert: IT-Sicherheit. Oldenbourg.
- N.A. Lynch: Distributed Algorithms. Morgan Kaufmann.
- Andrew S. Tanenbaum: Moderne Betriebssysteme. Pearson Studium.
- Andrew S. Tanenbaum: Computernetzwerke. Prentice Hall.

Zur Vorlesung

Aktuelle Informationen, Sprechzeiten, Folien und Projektvorschläge unter

`http://lionel.kr.hsnr.de/~rethmann/index.html`

Anmerkungen, Tipps, Korrekturen oder Verbesserungsvorschläge sind immer willkommen! Sprechen Sie mich an oder schicken Sie mir eine E-Mail an `jochen.rethmann@hsnr.de`.

Stellen Sie Fragen! Nur so kann ich beurteilen, ob Sie etwas verstanden haben oder noch im Trüben fischen.

> Wer fragt, ist ein Narr für eine Minute.
> Wer nicht fragt, ist ein Narr sein Leben lang.
>
> (Konfuzius, chinesischer Philosoph, 551 - 479 v. Chr.)

Zur Übung

In der Übung arbeiten Sie aktiv und selbständig an einem konkreten Projekt, das Sie selbst auswählen.

> Erzähle es mir und ich vergesse es.
> Zeige es mir und ich erinnere mich.
> Lass es mich tun und ich verstehe es.
>
> (Konfuzius, chinesischer Philosoph, 551 - 479 v. Chr.)

Ablauf der Übung:

- baut stark auf vorbereitender und selbständiger Arbeit auf
- die Anwesenheitspflicht gilt für die gesamte Dauer der Übung
- der Teilnahmeschein wird nur bei erfolgreicher Bearbeitung aller Teilaufgaben ausgestellt
- Konzeptgespräche zu Beginn und Ergebnisbesprechung am Ende des Semesters (Vorlesungszeit)

Anforderungen

Sie sollten die folgenden Module erfolgreich absolviert haben:

- Einführung in die Programmierung (EPR) sowie Objektorientierte Anwendungsentwicklung (OOA), da der Schwerpunkt dieses Moduls auf der Programmierung von verteilten Systemen liegt.
- Betriebssysteme (BSY), da die entwickelten Programme der verteilten Systeme darauf ausgeführt werden.

Viele Module der zukünftigen Semester werden die angesprochenen Themen vertiefen, da das Modul Verteilte Systeme die Informatik vereint:

- Rechnernetze (DNM), da die physikalisch verteilten Rechner über Netzwerke verbunden sind.
- Datenbanksysteme (DBS), da oft riesige Datenmengen verteilt und parallel verarbeitet werden.

Was Sie beachten sollten

- Ich unterrichte meine Schüler nie; ich versuche nur, Bedingungen zu schaffen, unter denen sie lernen können.
  (Albert Einstein, deutscher Physiker, 1879 - 1955)
- Einen jungen Menschen unterrichten heißt nicht, einen Eimer füllen, sondern ein Feuer entzünden.
  (Aristoteles, griechischer Philosoph, 384 - 322 v. Chr.)
- Zu oft geben wir unseren Kindern Antworten, die sie behalten sollen, anstatt Aufgaben, die sie lösen sollen.
  (Roger Lewin, amerikanischer Anthropologe, geb. 1946)
- Der weise Mann gibt nicht die richtigen Antworten, er stellt die richtigen Fragen.
  (Claude Levi-Strauss, französischer Anthropologe, geb. 1908)

→ Wählen Sie ein Projekt, das Sie interessiert.

Was Sie beachten sollten

- Suche nicht andere, sondern dich selbst zu übertreffen.

  (Cicero, römischer Politiker und Philosoph, 106 - 43 v. Chr.)

- Es ist nicht wichtig, wie langsam du gehst, sofern du nicht stehen bleibst.

  (Konfuzius, chinesischer Philosoph, 551 - 479 v. Chr.)

→ Wählen Sie ein Projekt, das Ihrem Leistungsstand und Ihren Fähigkeiten angemessen ist.

Übersicht

Verteilte Systeme

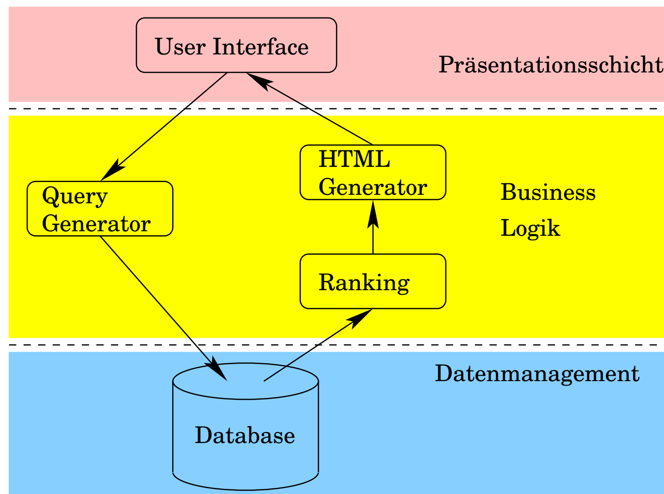
- *Einführung*
- Programmiermodelle und Entwurfsmuster
- Architekturmodelle
- Verteilte Algorithmen
- Dienste

Übersicht

Einführung

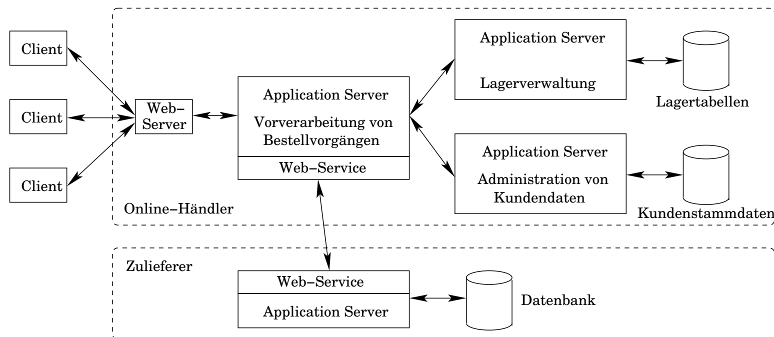
- *Motivation*
- Zielsetzung
- Konzepte

Beispiel: Suchmaschine



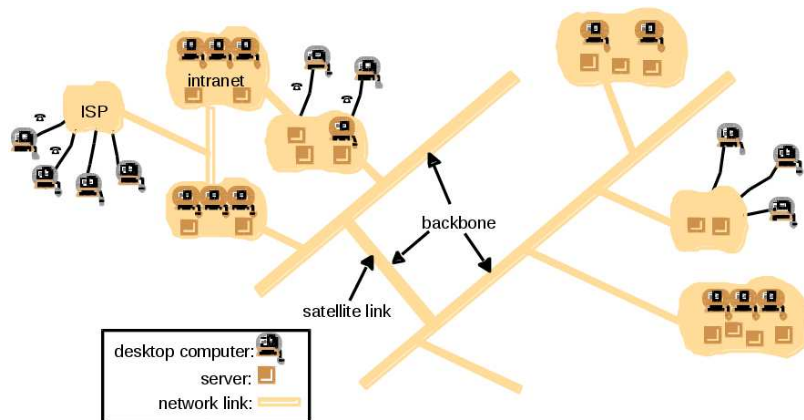
Hardware- und Software-Schichten

Beispiel: Auftragserfassung und -abwicklung



- Komplexe Geschäftsvorgänge werden durch mehrere, raümlich verteilte, kooperierende Server ausgeführt.
- Die Kommunikation in Rechnernetzen erfolgt mittels Nachrichtenaustausch, nicht über gemeinsamen Speicher.
- Die Datenbestände sind in Datenbanken abgelegt.

Internet



aus: Couloris, Dollimore, Kindberg: Verteilte Systeme. Pearson Studium.

Anwendungsbeispiel

Internet

- Zusammenschluss verschiedener Netzwerke und Rechner, sowie verschiedener Hard- und Software.
- Kommunikation durch Austausch von Nachrichten, kein gemeinsamer Speicher.
- Ortsunabhängige Nutzung bereitgestellter
  - Daten: Text, Bilder, Audio, Video, ...
  - Dienste: WWW, E-commerce, E-Mail, ...
  - Anwendungen (ASP = Application Service Provider)
- Ortsunabhängiger Zugang zu Rechenressourcen:
  - SETI@home (Search for ExtraTerrestrial Intelligence at home)
  - GRID-Computing
  - Cloud-Computing

Anwendungsbeispiel

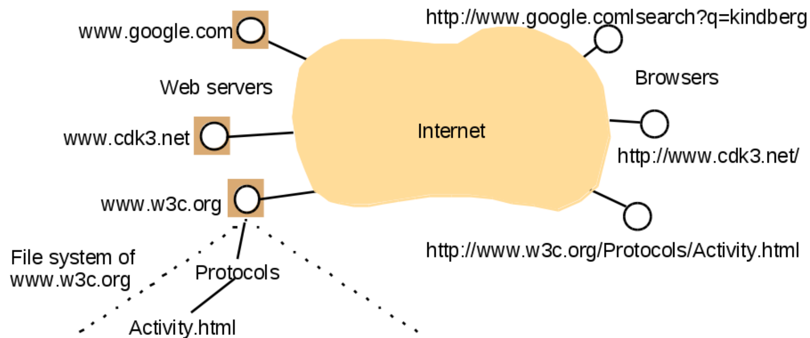
Intranet

- Teil des Internets, das separat verwaltet wird und dank einer Abgrenzung über lokale Sicherheitsstrategien verfügt.
- Die Konfiguration unterliegt der Verantwortlichkeit des Unternehmens.
- Ausdehnung ist nicht lokal beschränkt, mehrere LANs im Intranet sind möglich $\rightarrow$ VPN: Virtual Private Network
- Durch besondere Absicherung viele Möglichkeiten zur Ressourcenteilung.
- Firewalls: Filterung der ein-/ausgehenden Nachrichten.

Aber: Kenntnis über Ort einer Ressource/eines Dienstes notwendig und Zugriffsoperationen sind für lokale/entfernte Ressourcen unterschiedlich.

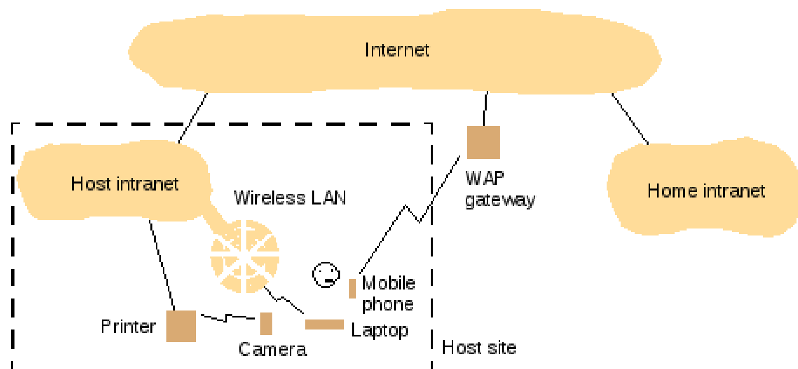
Grids versuchen diese Einschränkungen zu überwinden.

World Wide Web



aus: Coulouris, Dollimore, Kindberg: Verteilte Systeme. Pearson Studium.

Mobile Computing



aus: Coulouris, Dollimore, Kindberg: Verteilte Systeme. Pearson Studium.

Definitionsversuche

- G. Coulouris, J. Dollimore, T. Kindberg:

  Als Verteiltes System wird ein System bezeichnet, bei dem sich die Hardware- und Softwarekomponenten auf vernetzten Rechnern befinden und nur über den Austausch von Nachrichten kommunizieren und ihre Aktionen koordinieren.

- Lamport: (nicht ganz ernst gemeint)

  Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil ein Rechner abgestürzt ist, von dem ich nicht einmal weiß, dass es ihn überhaupt gibt. ☺

- A.S. Tanenbaum, M. van Steen:

  Ein verteiltes System ist eine Menge unabhängiger [durch ein Netzwerk verbundener] Computer, welche dem Benutzer [durch geeignete Software-Unterstützung] wie ein einzelnes, kohärentes System erscheinen. (kohärent: [lat.] zusammenhängend)

Grundeigenschaften

- Parallele/nebenläufige Aktivitäten (auch lokal mittels Threads)

  $\rightarrow$ *Koordination und Synchronisation erforderlich*

- Es gibt keine globale Uhr und aufgrund von Signallaufzeiten nur begrenzte Synchronisationsmöglichkeiten für die lokalen Uhren.

  $\rightarrow$ *Timing-Problem*

  Beispiele:
  - Make-Utility, wenn Compiler und Editor auf unterschiedlichen Rechnern ohne globale Uhr laufen.
  - auch: Welcher Kunde aus welchem Reisebüro wollte die Reise zuerst buchen?

- Interaktion durch Nachrichtenaustausch; hier hauptsächlich Sockets, RPC, Java RMI und CORBA.

Grundeigenschaften

- Die Systeme und Anwendungen können sehr groß sein.
  Beispiel LHC-Computing Grid: $\approx$ 70.000 PCs + große Cluster
- Fehler und Ausfälle sind wahrscheinlich. Fehlerquellen: lokaler Rechenknoten, Verbindungsstrukturen, Kommunikation, ...
  $\rightarrow$ *Fehlertoleranz notwendig*

  Fehlertoleranz heißt, dass ein System auch dann korrekt funktioniert, wenn Teilsysteme falsche Werte liefern. Wird im wesentlichen durch Redundanz erreicht.
- Heterogene Hardware- und Softwarekomponenten.
  $\rightarrow$ *Standardisierung von Schnittstellen erforderlich*
- Nachrichten werden über (öffentliche) Netzwerke ausgetauscht, Ressourcen werden gemeinsam genutzt.
  $\rightarrow$ *Sicherheitsprobleme* IT-Sicherheit bei Prof. Dr. Quade

Grundeigenschaften

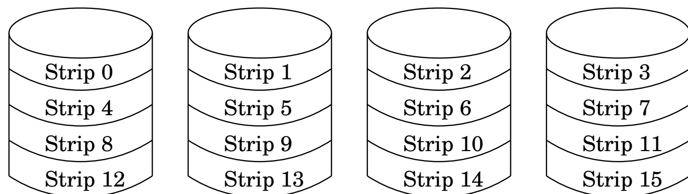
- Erhöhter Nutzwert durch Ressourcenteilung:
  - Ressource: Komponenten, die sich in einem vernetzten Rechnersystem sinnvoll gemeinsam nutzen lassen.
  - Beispiele: Drucker, Festplatten, Datenbanken, Dateien, Kalender, gemeinsame Dienste wie Web-Server und E-Mail.
  - Heute betrachtet man sogar die Rechner selbst als Ressource. „Unsere“ Rechner: Wenn bei uns Nacht ist, können die Rechner von Menschen genutzt werden, bei denen es Tag ist.
- Ausfallsicherheit: Das verteilte System funktioniert trotz Ausfall einer Komponente immer noch korrekt. Wird durch Redundanz erreicht.
  - physikalisch: Redundante Datenspeicherung auf räumlich verteilten Datenträgern. Oder: Redundante Ausführung von Diensten und Operationen fangen einzelne Server- oder Prozess-Ausfälle ab wie bei DNS.
  - zeitlich: mehrmalige Übertragung der Daten wie bei TCP
  - datentechnisch: Prüfsummen bei fehlerkorrigierenden Codes

Ausfallsicherheit durch Redundanz

Beispiel: Ein NAS (Network Attached Storage) speichert die Daten vieler Rechner und ist in der Regel mit einem RAID-System (Redundant Array of Independant Disks) ausgestattet.

RAID level 0 Fasse Festplatten zu logischem Laufwerk zusammen.

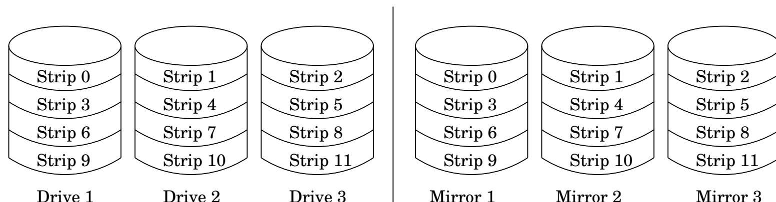
- Daten werden blockweise auf die Festplatten verteilt.
- Bei Ausfall einer Platte geht die gesamte Information verloren.
- Erhöhte Transferrate, da alle Festplatten gleichzeitig transferieren; aber Nachteile bei kurzen Requests durch Overhead beim Aufteilen.



Ausfallsicherheit durch Redundanz

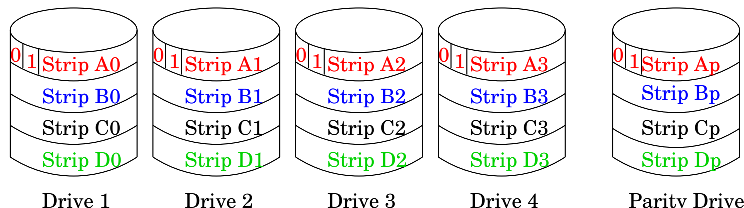
RAID level 1

- Spiegeln aller Daten einer Platte/eines RAID-0-Sets auf eine zweite Platte/ein zweites RAID-0-Set.
- Bei Ausfall einer Platte bleiben die Daten zugreifbar.
- Lesezugriff: Durch aufteilen auf zwei Festplatten/Sets doppelte Performanz. Schreibzugriff: Gleiche Performanz.
- Hohe Kosten: Nur die Hälfte des Plattenplatzes steht für die Originaldaten zur Verfügung.



RAID level 4

- Im Gegensatz zu RAID-0 werden auf einer zusätzlichen Festplatte Parity-Informationen gespeichert.
- Vorteil: Bei Ausfall einer Platte können Informationen Bit für Bit rekonstruiert werden.



RAID level 4 (Fortsetzung)

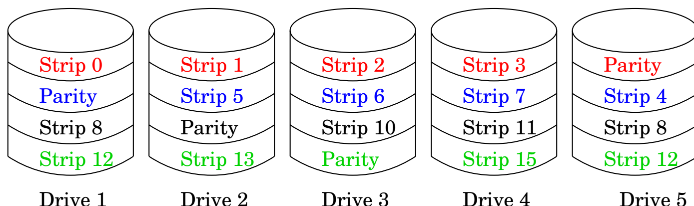
- Nachteil: Geringere Performanz als RAID-0/1 durch Schreiben zweier und Lesen aller Platten: Zur Berechnung der Parity-Informationen müssen alle beteiligten Strips gelesen und ausgewertet werden.
- Flaschenhals: Parity-Festplatte.
  Die Parity-Festplatte fällt oft aus, da diese Platte an jedem Schreibzugriff beteiligt ist und daher am stärksten genutzt wird.

Anmerkungen:

- Ein RAID-System ersetzt keine Datensicherung: Versehentliches oder fehlerhaftes Schreiben sowie Viren werden dauerhaft und redundant abgespeichert.
- RAID-Systeme sind nicht fehlertolerant: Beim Lesen werden die Parity-Informationen in der Regel nicht ausgewertet.

RAID level 5

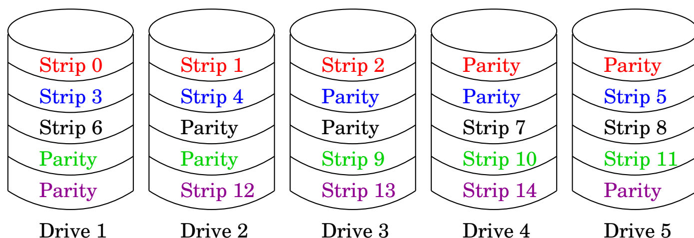
- Parity-Informationen und Daten werden blockbereichsweise auf die Platten verteilt: Jede Festplatte ist für einen bestimmten Blockbereich Parity-Platte.
- Performanz deutlich geringer als bei einzelnen Platten.



Ausfallsicherheit durch Redundanz

RAID level 6

- Wie RAID level 5, aber Parity-Informationen werden auf zwei unterschiedliche Platten verteilt.
- Es dürfen bis zu 2 Platten ausfallen.
- Performanz fällt nochmals ab gegenüber RAID level 5.



Ausfallsicherheit durch Redundanz

Anmerkungen:

- Es gibt weitere RAID level und Kombinationen von RAID level.
- Software-RAID: Der Controller ist nicht als Hardware realisiert.

GlusterFS

- verteiltes Dateisystem von Red Hat
- Replizierte Daten auf verschiedenen Bricks
  - im einfachsten Fall verschiedene Partitionen auf einem Server
  - oder auf verschiedenen Rechnern, die sich gegenseitig vertrauen müssen und gleichberechtigt sind: peers
- Geo-Replikation mittels *ssh* und *rsync*
- Metadaten verteilt mittels DHT: Distributed Hash Table

Parallelverarbeitung

Beschleunigung der Verarbeitung:

- Einsatz autonomer, vernetzter Verarbeitungseinheiten zur simultanen Verarbeitung (fast) unabhängiger Teilprobleme.
- Einfachste Vorgehensweise:
  - Aufteilen der Daten in (disjunkte) Teilmengen.
  - Verteilen der Teilmengen auf mehrere Prozessoren.
  - Simultane Verarbeitung und Zusammenfassen der Teilergebnisse zum Endergebnis.

→ *Parallel Computing* Prof. Dr. Ueberholz, Master Informatik

Lichtgeschwindigkeit ($c_{si} = 3 \cdot 10^{10} mm/s$) setzt prinzipielle Grenze für die Geschwindigkeit bei Einprozessorrechnern: Ein Chip mit 3cm Durchmesser kann Signal in etwas 1ns fortpflanzen $\rightarrow$ 1GFlops

Schnellere Rechner sind also nur möglich, wenn die Chips weiter verkleinert werden können. Leider gibt es dabei einige Probleme:

- Quantenmechanische Effekte bei Abmessungen kleiner 5nm.
- Bei der Lithografie werden Strukturen durch Belichtung auf den Wafer übertragen. Wie können kleinere Masken hergestellt werden? Belichtung muss dann mit sehr kurzwelligem „Licht“ erfolgen. Aber Fluor-Excimerlaser (157nm) sind extrem teuer, deshalb heute Argon-Fluorid-Excimerlaser (193nm).

Die sinnvolle Anwendung paralleler/verteilter Systeme hängt von der dem Problem innewohnenden Parallelität ab: Anzahl gleichzeitig verarbeitbarer Teilaufgaben, in die sich das Problem im Mittel zerlegen lässt.

Beispiel Matrixmultiplikation

Aufteilen der $n \times n$-Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen:

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

$$C = A \cdot B$$

mit

$$\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned} \Rightarrow$$

8 Multiplikationen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen
4 Additionen von $\frac{n}{2} \times \frac{n}{2}$-Matrizen

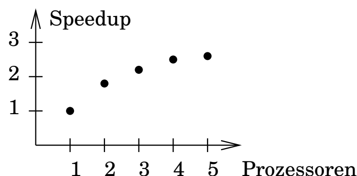
Die Teilmatrizen r, s, t und u können unabhängig voneinander berechnet werden, wenn A und B auf den Prozessoren zur Verfügung stehen.

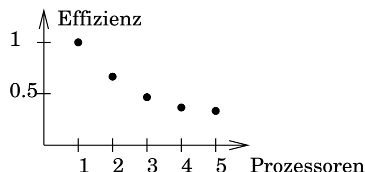
Frage: Wie werden Daten effizient verteilt? Broadcast?

Speedup/Effizienz

Leistungsmessung/-bewertung:

- $\text{Speedup(n)} = \frac{\text{Rechenzeit 1 Prozessor}}{\text{Rechenzeit n Prozessoren}}$
- $\text{Effizienz(n)} = \frac{\text{Speedup(n)}}{\text{n}}$





Sequentielle und parallele Algorithmen sind oft sehr verschieden. Daher vergleichen wir den besten sequentiellen mit dem besten parallelen Algorithmus.

Amdahls Gesetz:

> Ein paralleles Programm kann nicht schneller als sein sequentieller Anteil bearbeitet werden, unabhängig von der Anzahl Prozessoren.

Sei f der nur sequentiell ausführbare Programmteil:

$$T_1 = (1 - f) + f = 1$$

Parallelität kann nur den Teil $(1 - f)$ beschleunigen:

$$T_n = (1 - f)/p_n + f$$

Damit ergibt sich für den Speedup:

$$S_n = \frac{1}{(1 - f)/p_n + f} \leq \frac{1}{f}$$

Speedup/Effizienz

Warum betrachten wir Parallelität, Speedup und Effizienz? Sind unsere heutigen Rechner nicht schnell genug, um jede Aufgabe in einer vertretbaren Zeit zu erledigen?

Beispiel: Traveling Salesperson Problem TSP

- *Gegeben:* Eine Menge von Orten, die alle untereinander durch Wege verbunden sind. Die Wege sind unterschiedlich lang.
- *Gesucht:* Eine Rundreise, die durch alle Orte genau einmal führt und unter allen Rundreisen minimale Länge hat.

  Wird bei jeder Tourenplanung benötigt: Leerung von Briefkästen, Bestückung von Platinen durch Roboter, usw.
- *Komplexität:* TSP ist NP-vollständig.

Bester bekannter Algorithmus: Laufzeit $\approx 2^n$ bei n Orten.

Speedup/Effizienz

Aktuelle Intel Core i7 Prozessoren mit 2.66 GHz:

Desktop Prozessor i7-975: 42.56 GFlops

Mobile Prozessor i7-620M: 21.28 GFlops

Annahme: Der Prozessor führt $20 \cdot 10^9$ characteristische Operationen pro Sekunde aus.

Zeitdauer	Schritte
Sekunde	$20 \cdot 10^9$
Minute	$1.200 \cdot 10^9$
Stunde	$72.000 \cdot 10^9$

Zeitdauer	Schritte
Tag	$1.728.000 \cdot 10^9$
Woche	$12.096.000 \cdot 10^9$
Jahr	$615.168.000 \cdot 10^9$

Lösbare Problemgröße

in einer Sekunde:	34 Städte	an einem Tag:	50 Städte
in einer Minute:	40 Städte	in einem Jahr:	59 Städte
in einer Stunde:	46 Städte	in 100 Jahren:	65 Städte

In Deutschland gibt es ungefähr 5000 Städte!

Frage: Löst ein schnellerer Rechner das Problem?
Antwort: **Nein!**

Projektion:

- Bisher: Geschwindigkeit verdoppelte sich alle 1,5 Jahre.
- in 10 Jahren: Lösbare Problemgröße am Tag: 56 Orte
- in 100 Jahren: Lösbare Problemgröße am Tag: 116 Orte

Computer	Dauer	Anzahl Schritte
A	1 Tag	2^n
B	1 Tag	$2 \cdot 2^n = 2^{n+1}$

Selbst wenn ein schnellerer Rechner das Problem lösen würde, wäre dies nur eine theoretische Lösung, denn die Touren müssen jetzt geplant werden, nicht in 10 Jahren.

Speedup/Effizienz

Frage: Lösen parallele oder verteilte Systeme das Problem?
Antwort: Bei schweren Problemen **nein!**

Lösbare Problemgröße am Tag bei linearem Speedup:

#Rechner	TSP $\approx 2^n$
1	50
10	53
100	57
1.000	60
10.000	63

Speedup/Effizienz

Lösung: Wir benötigen bessere Algorithmen!

Laufzeit	Dauer für 5000 Städte		#Städte am Tag
$\approx n^2$	<1	Sekunde	41.569.219
$\approx n^3$	6	Sekunden	120.000
$\approx n^4$	9	Stunden	6.447
$\approx n^5$	5	Jahre	1.115
$\approx n^6$	24.774	Jahre	346

Speed is fun!

Frage: Gibt es bessere Algorithmen für das Problem?

→ *Effiziente Algorithmen*, Master Informatik.

Aber wir wollen nicht schwarz malen:

Denn es gibt ja auch andere Probleme, die in der Praxis eine bedeutende Rolle spielen, und die durch parallele Verarbeitung deutlich beschleunigt werden können.

Lösbare Problemgröße am Tag bei linearem Speedup:

#Rechner	TSP $\approx 2^n$	Matrixmultiplikation $\approx n^3$
1	50	120.000
10	53	258.532
100	57	556.990
1.000	60	1.200.000
10.000	63	2.585.321

Übersicht

Einführung

- Motivation
- *Zielsetzung*
- Konzepte

Aufgabenstellungen

Heterogenität: Verteilte Systeme basieren auf unterschiedlichen Netzwerken, Betriebssystemen, Programmiersprachen, Implementierungen, unterschiedlicher Hardware, ...

Wir benötigen daher

- gemeinsame bzw. standardisierte Netzwerkprotokolle,
- austauschbare, hardware-unabhängige Formate für Daten und Datenstrukturen
- und Standards für die Interprozesskommunikation.

Ansätze zur Lösung des Problems:

- Erzeuge Code für virtuelle Maschine, nicht für Hardware.
- Middleware: Eine Software-Schicht verbirgt die Heterogenität.

  Aber: Solche Programme sind zwar unabhängig von der Hardware und vom Betriebssystem, aber abhängig von der Middleware!

Aufgabenstellungen

Offenheit/Erweiterbarkeit des Systems:

- Grad, zu dem neue Dienste hinzugefügt und zur Verwendung von unterschiedlichen Clients bereitgestellt werden können.
- Veröffentlichung von Spezifikation/Dokumentation zu Schnittstellen und Internet-Protokollen:
  - RFC-Dokumente (Requests For Comment) enthalten Diskussionen und Spezifikationen.
  - Beispiel: Für CORBA existiert eine Reihe technischer Dokumente, siehe `http://www.omg.org`.
  - Oft wird so der langsame, offizielle Standardisierungsweg umgangen.
- herstellerunabhängig

Aufgabenstellungen

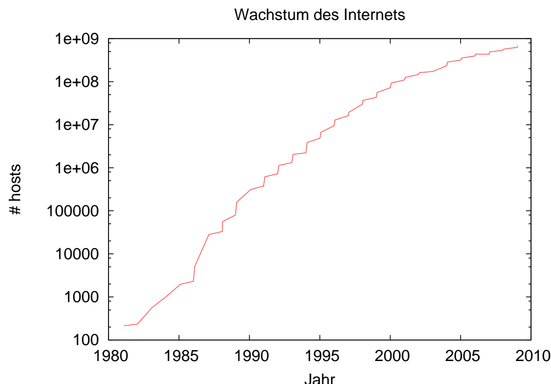
Sicherheit:

- Vertraulichkeit: Keine Offenlegung der Ressourcen gegenüber nicht-berechtigten Personen/Agenten.
- Integrität: Schutz der Ressourcen gegen Veränderung oder Beschädigung.
- Verfügbarkeit: Schutz gegen Störungen der Methoden für Ressourcenzugriff wie
  - *Denial-Of-Service-Angriffe:* Einen Server mit vielen sinnlosen Anfragen überfluten, so dass ernsthafte Anfragen nicht mehr beantwortet werden können.
  - *Computerviren:* Wie kann automatisch festgestellt werden, ob das mitgelieferte Skript einen Virus enthält? (Sicherheit mobilen Codes!)

Aufgabenstellungen

Skalierbarkeit:

- Verteilte Systeme müssen auch bei steigender Anzahl von Benutzern/Komponenten effizient/effektiv arbeiten.
- Beispiel: Internet Quelle: `https://www.isc.org`



Aufgabenstellungen

Kernprobleme beim Entwurf skalierbarer verteilter Systeme:

- Kostenkontrolle der physischen Ressourcen: Bei steigender Nachfrage Erweiterung zu vernünftigen Kosten.
- Vermeiden von Leistungsengpässen: Bevorzuge hierarchische Lösungen, so dass die Menge zu ladender/verarbeitender Daten nicht zu groß wird, z.B. DNS.
- Erschöpfung der SW-Ressourcen verhindern: Ressourcen so anlegen, dass sie für zukünftige Erweiterungen voraussichtlich ausreichen werden.

  Beispiel: Umstellung von 32-Bit Internetadressen auf 128-Bit. Wer hätte vor 20 Jahren gedacht, dass auch Waschmaschinen eine eigene IP-Adresse haben werden oder dass man für Mobiltelefone Internetverbindung benötigt?

Aufgabenstellungen

Nebenläufigkeit

- Zur Erhöhung des Durchsatzes.
- Macht Sicherungsmechanismen bei parallelen Zugriffen notwendig, z.B. Transaktionskonzept in verteilten Systemen.

Fehlerverarbeitung

- Ist in der Regel schwierig: Oft treten partielle Ausfälle auf.
- Fehler können in einigen Fällen problemlos erkannt werden:
  - Prüfsummen bei fehlerkorrigierenden Codes
  - Triple Modular Redundancy: Drei identische Prozesse teilen das Ergebnis ihrer Berechnung drei Auswertern (Voter) mit. Mehrheitsentscheidung kann einfachen Fehler ausgleichen.
- Fehler maskieren: Erkannte Fehler können durch Wiederholung der Aktion/Redundanz verborgen oder abgeschwächt werden.

Aufgabenstellungen

Fehlerverarbeitung (Fortsetzung)

- Fehler tolerieren: Fehler anzeigen und dem Benutzer die weitere Entscheidung überlassen, z.B. „Web-Seite nicht erreichbar“.
- Wiederherstellung nach Fehlern: Journaling File System, Database Recovery, ...
- Erhöhte Verfügbarkeit/Fehlertoleranz durch Redundanz.

  Beispiele:
  - Zwei mögliche Routen zwischen Routern im Internet.
  - Jede DNS-Tabelle (Domain Name Service) wird auf zwei Hosts repliziert.
  - Replikation der Datenbank-Partitionen auf unabhängigen Rechnern oder mittels RAID.

Transparenz

Verbergen der räumlichen Trennung der einzelnen Komponenten im verteilten System vor Benutzern/Anwendungen.

- *Zugriffstransparenz:* Identische Zugriffsoperationen für lokale und entfernte Ressourcen.
- *Ortstransparenz:* Es ist keine Kenntnis über den Ort einer Ressource bzw. eines Dienstes notwendig.
- *Replikationstransparenz:* Ressourcenreplikation zur Verbesserung der Leistung/Zuverlässigkeit sind für Benutzer/Anwendungen unsichtbar.

Seltsame Namensgebung, oder? Wenn etwas transparent ist, ist es durchsichtig. Aber bei Computersystemen bezeichnet Transparenz eine Hardware oder Software, deren Existenz für den Benutzer weder direkt erkennbar noch relevant ist. (Quelle: wikipedia)

Aufgabenstellungen

Transparenz (Fortsetzung)

- *Mobilitätstransparenz:* Verschiebung von Ressourcen/Clients innerhalb des Systems ist ohne Beeinträchtigung der Arbeit möglich. Ist eigentlich ein Aspekt der Ortstransparenz.
- *Leistungstransparenz:* Dynamische Rekonfiguration bei variierender Last ist möglich. Ist eigentlich ein Aspekt der Ortstransparenz.
  Idee: Rechenleistung wird vom Netz bereitgestellt; welcher Rechner die Leistung erbringt, ist unsichtbar.
- *Skalierungstransparenz:* Vergrößerung des Systems ist ohne Veränderung der Systemstruktur und Anwendungen möglich.

Verteilte Systeme

Vorteile:

- Kostenreduktion
- Lokale Kontrolle und Verfügbarkeit
- Leichte Erweiterbarkeit
- Ausfalltoleranz
- Hohe Leistung durch Parallelität
- Modulare Software
- Herstellerunabhängigkeit
- Übereinstimmung mit organisatorischen Strukturen

Nachteile:

- Hoher Bedienungs- und Wartungsaufwand
- Probleme durch Heterogenität
- Komplexer Design- und Implementierungsprozess
- Schwierige Verifikation der Korrektheit
- Komplexe Kommunikationssysteme
- Hoher Aufwand beim Übergang vom zentralen zum dezentralen System
- Sicherheitsprobleme
- Gesamtkosten schwer abschätzbar

Übersicht

Einführung

- Motivation
- Zielsetzung
- *Konzepte*

Schichtenmodelle

Verteilte Anwendung besteht aus Komponenten:

- graphische Präsentation / Benutzungsoberfläche
  Beispiel: der X-Client übermittelt Tastaturanschläge und Mausbewegungen an den X-Server
- Benutzungsinterface / Graphical User Interface (GUI)
  Beispiele: Eingabemaske, Suchdialog, ...
- Verarbeitung / Anwendung (Applikationslogik)
- Datenmanagement und
- persistente Datenspeicherung (Datenbank, XML-Datei)

→ Komponenten werden auf Rechenressourcen aufgeteilt.

Schichtenmodelle

Bei klassischen Client/Server-Architekturen ergeben sich zwei Schichten: *two tier model*.

Klassifikation von Clients: Unterscheidung danach, wo die Trennung zwischen Client und Server gelegt wird.

- *null client:* Die Trennlinie liegt zwischen graphischer Benutzungsoberfläche und -interface: Der Client übergibt nur Tastaturanschläge und Mausbewegungen an den Server.

→ host based computing

- *thin client:* Die Trennlinie liegt zwischen Benutzungsinterface und Anwendung. Der Client verwaltet die GUI sowie einige Zustandsinformationen, der Server implementiert die Applikationslogik und verwaltet die Daten.

→ remote presentation

Beispiel: NetPC

Schichtenmodelle

- *applet client:* Trennlinie liegt hinter Benutzungsinterface und schließt Teile der Applikationslogik ein.

→ cooperative processing

- *fat client:* Trennlinie liegt zwischen Applikationslogik und Datenmanagement.

Heute werden Anwendung zerlegt in einen Client und mehrere Server, wodurch mehrschichtige Anwendungen entstehen: *multi tier model*.

- Bessere Skalierbarkeit bei Servern.
- Einschluss der Intranet/Internet-Technologie.

Schichtenmodelle

Typisch ist ein Gliederung in drei Schichten: *three tier model*.

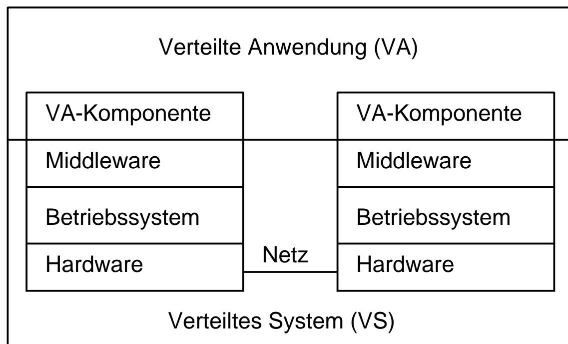
- Clients für Präsentationsschicht. (client tier, front-end)
- Applikations-Server für Anwendunglogik und Datenmanagement. (application-server tier, middle tier, enterprise tier, Fachkonzept-Schicht)
- Datenbank-Server zur persistenten Datenspeicherung. (data-server tier, back-end)

Heute wird trotz objektorientierter Programmierung oft noch ein relationales Datenbank-Management-System DBMS zur Speicherung der Daten eingesetzt. Um auch Vererbung und Polymorphie abbilden zu können, werden in der Datenzugriffsschicht objekt-relationale Mapping-Tools eingesetzt.

Middleware

Die Verteilung der Software-Komponenten soll für den Benutzer transparent sein. Die Anwendung soll genauso ablaufen, als ob sie auf einem einzigen Rechner abläuft. $\rightarrow$ Middleware Services verbergen die Heterogenität eines verteilten Systems.

Die Verteilung der Komponenten erfolgt dabei nicht durch das Betriebssystem, sondern durch Konfiguration.



Middleware

- Stellt Entwicklern ein praktisches Programmiermodell zur Verfügung:
  - Prozesse und Objekte, die durch Zusammenarbeit die Kommunikation und gemeinsame Ressourcennutzung unterstützen.
- Ein Middleware-Service ist verteilt:
  - Erlaubt entfernten Zugriff (z.B. Datenbankzugriff) oder
  - befähigt andere Services zu einem entfernten Zugriff.
- Abstrahiert von entfernten, über das Netz ablaufenden Interaktionen:
  - Client und Server benutzen API der Middleware. Die Middleware transportiert die Anfrage über das Netz zum Server und das Ergebnis des Servers zum Client.
  - Middleware läuft bei einer Interaktion auf beiden Seiten, dem Client und dem Server.
  - Middleware unterstützt Standard-Protokoll oder zumindest ein veröffentlichtes Protokoll, z.B. TCP/IP oder IPX von Novell.

Middleware

Eine Middleware stellt Bausteine/Dienste für den Aufbau von Software-Komponenten bereit, die in einem verteilten System zusammenarbeiten können.

Beispiele solcher Dienste:

- Abstraktion von Kommunikationsaktivitäten: Entfernte Prozedur-/Funktionsaufrufe, Gruppenkommunikation, Ereignisbenachrichtigung, Datenreplikation.
- Qualitätssicherung, z.B. Echtzeitübertragung multimedialer Daten.
- Dienste für Anwendungen: Namensdienste, Sicherheit, Transaktionen, persistenter Speicher, Zeitdienst, ...

Zusammenfassend: Middleware ist eine Software-Schicht, die auf Basis standardisierter Schnittstellen und Protokolle Dienste für eine transparente Kommunikation verteilter Anwendungen in einem heterogenen Umfeld bereitstellt.

Middleware

Kommunikationsorientierte Middleware konzentriert sich auf die Abstraktion der Netzwerkprogrammierung. Beispiele:

- Remote Procedure Call (RPC)
- Java Remote Method Invocation (RMI)
- Java Message Service (JMS)
- Web-Services

Anwendungsorientierte Middleware stellt neben Kommunikation die Unterstützung verteilter Anwendungen in den Mittelpunkt. Beispiele:

- Distributed Computing Environment (DCE): Dienste und Werkzeuge für verteilte Datenverarbeitung
- Common Object Request Broker Architecture (CORBA)
- Distributed Common Object Model (DCOM) und Dot-Net.

DCE

Distributed Computing Environment

- Wurde definiert durch die Open Software Foundation (OSF)
  - `http://www.opengroup.org/dce/`
  - Teilnehmer sind z.B. IBM, SAP, NEC und HP.
- Ist eines der ersten bekannten verteilten Systeme, mittlerweile als open source freigegeben.
- Etwas aus der Mode gekommen, wird aber z.B. in der Uni Karlsruhe eingesetzt, näheres dazu unter: `http://www.scc.kit.edu/produkte/4933.php`
- Ideen sind Grundlage vieler aktueller Systeme.

DCE

Primäres Ziel: Herstellerunabhängigkeit

- Aufsatz für Betriebssysteme wie Windows, Unix, AIX, Solaris, OS/2, ...
- Unterstützung diverser Protokolle wie TCP/IP, X.25, ...
- Unterschiede der Maschinen werden durch automatische Konvertierungen vor dem Anwender versteckt $\rightarrow$ vereinfachte Anwendungsentwicklung

DCE: kein Betriebssystem oder Anwendung, sondern eine Sammlung von Services und Werkzeugen, eine Middleware.

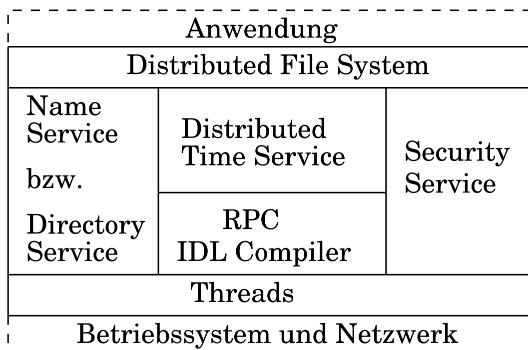
DCE: Wichtige Komponenten

- Thread-Package, ist evtl. im Betriebssystem integriert.
- RPC-Facility: Werkzeuge zur Gestaltung von Prozeduraufrufen auf entfernten Rechnern.
- Distributed Time Services (DTS): Die Uhren aller beteiligter Einheiten werden, so weit es geht, synchronisiert.
- Namensdienst/Cell Directory Service (CDS): Ortstransparenter Zugriff auf Server, Dateien, Geräte, ...
- Sicherheitsdienste: Authentifikation und Autorisierung.
- Distributed File Services (DFS): Verteiltes Dateisystem stellt einheitlichen, transparenten, systemweiten Zugriff auf Dateien bereit.

  Bietet Vorteile bzgl. Sicherheit, Performance, Verfügbarkeit, feinerer Rechtevergabe, ...

DCE: Grundsätzlicher Aufbau

- Jede Komponente nutzt die unter bzw. neben ihr liegenden Komponenten.
- Ein DCE-System kann tausende von Computern umfassen, die weltweit verteilt sind.



DCE

Strukturierung und Gliederung durch Konzept der Zellen:

- Eine Zelle ist eine administrative Einheit von Benutzern, Maschinen und Diensten, die einen gemeinsamen Zweck haben und sich DCE-Services teilen.
- Ausdehnung typischerweise wie eine NIS-Domäne.
- Jede Zelle hat einen weltweit eindeutigen Namen.
- Ein Rechner kann nur einer DCE-Zelle angehören.

Minimale Zellenkonfiguration: Cell Directory Service, Security Service, Distributed Time Service und Client-Maschinen.

Optional können für die Kommunikation zwischen den Zellen der Global Directory Service (GDS) oder das Domain Name System (DNS) herangezogen werden.

Übersicht

Verteilte Systeme

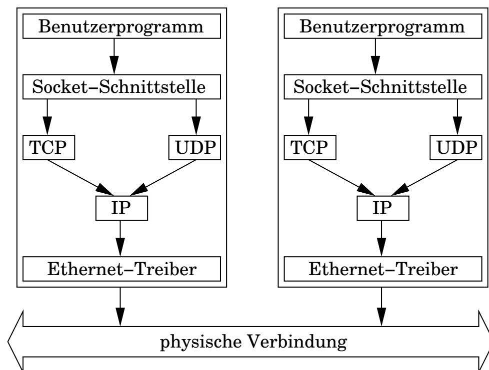
- Einführung
- *Programmiermodelle und Entwurfsmuster*
- Architekturmodelle
- Verteilte Algorithmen
- Dienste

Programmiermodelle und Entwurfsmuster

- *Nachrichtenbasiert: Sockets*
- Auftragsorientiert: RPC
- Objektorientiert: CORBA, Java RMI
- Web-Services

Sockets

- Prozesse verschiedener Rechner kommunizieren über Sockets.
- Socket-Schnittstellen (API) bietet jedes Betriebssystem an.
- Die Zugangspunkte für die Netzwerkverbindung heißen *Ports*.



- Verbindungsdetails übernehmen Kernel und Treiber.
- Austausch von kleinen Paketen auf unterer Ebene.

Anlegen von Sockets

Ein Socket ist eine interne Datenstruktur des Betriebssystems zur Abarbeitung der Kommunikation eines Prozesses.

`int socket(int domain, int type, int protocol)` aus der Bibliothek `sys/socket.h` erzeugt einen Socket-Deskriptor und liefert einen Handle für zukünftige Kommunikationsoperationen.

- `domain` definiert globale Einstellungen der Communication Domain. Wichtige Konstanten sind die Adressfamilien unter Unix oder des Internets: `AF_UNIX` und `AF_INET`
- `type` legt die Art der Verbindung fest: `SOCK_STREAM` (TCP), `SOCK_DGRAM` (UDP) und `SOCK_RAW` (Roh-Daten, Umgehen von Netzwerk-Protokollen)

  Verschiedene Kommunikationstypen unterscheiden sich insbesondere in der Zuverlässigkeit.
- Sind in einer Domäne mehrere Protokolle erlaubt, kann mittels `protocol` eins ausgewählt werden. Normalerweise 0.

Socket-Adressen

Sockets definieren einen Zugangspunkt für Netzwerkverbindungen. Heute werden im wesentlichen zwei Arten genutzt:

- Internet-Domain mit den Protokollen TCP, UDP und IP.
- Unix-Domain bei Kommunikation innerhalb eines Rechners.

Socket-Adressen werden in einem C-Programm mittels einer allgemeinen Struktur `sockaddr` dargestellt. Darauf aufbauend werden spezielle Strukturen definiert:

- `sockaddr_in` für Internet-Sockets
- `sockaddr_un` für Unix-Sockets
- `sockaddr_atalk` für Apple-Talk
- `sockaddr_ipx` für IPX von Novell

Socket-Adressen

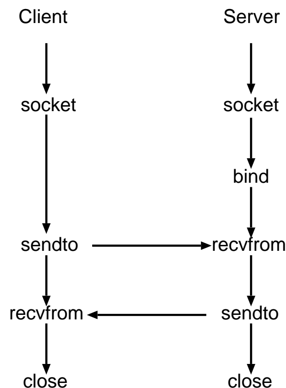
Internet-Domain:

```
struct sockaddr_in {
   short sin_family;              // Domain AF_INET
   unsigned short sin_port;       // Port-Nummer
   struct in_addr sin_addr;       // IP-Adresse
   unsigned char __pad[8];        // dummy
}
```

Unix-Domain:

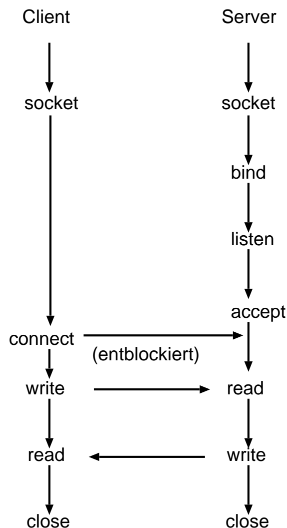
```
struct sockaddr_un {
   short sun_family;         // Domain AF_UNIX
   char sun_port[108];       // Path name
}
```

Verbindungslose Socket-Kommunikation



- Der Typ ist `SOCK_DGRAM`.
- `bind` bindet einen Socket an eine Adresse.
- Abschicken bzw. empfangen von Daten mittels `sendto` bzw. `recvfrom`.
- `close` schließt den Socket.
- Wird auf Grund der Unzuverlässigkeit nicht in Client/Server-Systemen verwendet.

Verbindungsorientierte Socket-Kommunikation



- Der Typ ist `SOCK_STREAM`.
- `bind` bindet einen Socket an eine Adresse.
- `listen` bereitet den Server-Socket für ankommende Anfragen vor und legt die Größe der Warteschlange fest.
- `accept` blockiert den Prozess und wartet auf ankommende Nachrichten.
- Client baut permanente Verbindung mittels `connect` auf.
- Abschicken bzw. empfangen von Daten: `write` / `read`

Datenaustausch über Sockets

`int send(int sfd, const void *msg, size_t len, int flags)` schickt Daten der Länge `len` Bytes ab der Adresse `msg` über den Socket `sfd`. Liefert Anzahl geschriebener Bytes.

`int recv(int sfd, void *buf, size_t len, int flags)` empfängt maximal `len` Bytes Daten über den Socket `sfd` und schreibt sie nach `buf`. Liefert Anzahl empfangener Bytes.

- Der `recv`-Aufruf blockiert, wenn keine Daten vorliegen. Aber mittels der Funktion `fcntl(...)` (steht für File Control) kann ein abweichendes Verhalten definiert werden.
- Liegen Daten von mehreren `send`-Aufrufen vor, können diese mit einem `recv`-Aufruf gelesen werden.
- Wird der Flag auf `MSG_PEEK` gesetzt, werden die Daten gelesen ohne diese aus dem Socket zu entfernen.
- Die Funktionen `read` und `write` entsprechen `recv` und `send`, aber ohne den Parameter `flags`.

Server-Sockets

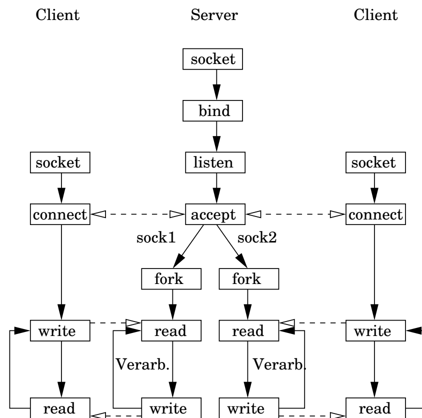
`int bind(int sockfd, struct sockaddr *addr, int len)`
bindet den Socket `fd` an die lokale Adresse `addr`. Die Größe der Struktur wird in `len` angegeben.

Mittels `bind` informieren wir das System über unseren Server, so dass alle an dem angegebenen Port eingehenden Nachrichten an das Programm weitergereicht werden sollen.

Auszug aus der Linux Manual Page:

> *When a socket is created with socket(2), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. [...] Traditionally, this operation is called „assigning a name to a socket“. [...] When INADDR_ANY is specified in the bind call, the socket will be bound to all local interfaces.*

Server-Sockets



- Ein Client nimmt über den Socket des Servers Verbindung zum Server auf.
- Damit der Server sofort für Anfragen weiterer Clients bereit ist, wird die Abarbeitung der Anfrage in einem Thread oder einem mittels `fork` erzeugtem Prozess durchgeführt.

Damit die Threads oder Prozesse nicht alle über denselben Socket kommunizieren, liefert `accept` einen neuen Socket, über den die weitere Kommunikation mit einem Client erfolgt.

Services

Jeder Dienst/Service läuft auf einem bestimmten Port. In der Datei `/etc/services` sind die Ports aufgelistet. Auszug:

```
ftp        21/tcp        pop3      110/tcp
ssh        22/tcp        sunrpc    111/tcp
telnet     23/tcp        ntp       123/tcp
smtp       25/tcp        imap3     220/tcp
http       80/tcp        talk      517/udp
```

Innerhalb eines Dienstes gibt es oft verschiedene Aktionen bzw. Methoden, die ausgeführt werden können. Bei HTTP sind dies unter anderem `GET`, `HEAD`, `POST`, `PUT`, `DELETE` und `TRACE`.

Services

Jede Programmiersprache stellt eine API zur Verfügung, um auf diese Daten zugreifen zu können. Auszug der Linux Manual Page:

- The `getservbyname(const char *name, const char *proto)` function returns a servent structure for the entry from the database that matches the service `name` using protocol `proto`. If `proto` is NULL, any protocol will be matched.
- The `getservbyport(int port, const char *proto)` function returns a servent structure for the entry from the database that matches the port `port` (given in network byte order) using protocol `proto`. If `proto` is NULL, any protocol will be matched.

- The `getservent(void)` function reads the next entry from the services database and returns a servent structure containing the broken-out fields from the entry.
- The `servent` structure is defined in `<netdb.h>` as follows:

```
struct servent {
    char  *s_name;    // official service name
    char **s_aliases; // alias list
    int    s_port;    // port number
    char  *s_proto;   // protocol to use
}
```

The members of the servent structure are:

  - `s_name` The official name of the service.
  - `s_aliases` A NULL-terminated list of alternative names for the service.
  - `s_port` The port number for the service given in network byte order.
  - `s_proto` The name of the protocol to use with this service.

Services

```
#include <netdb.h>
.....
int main(void) {
    struct servent *svc =
              getservbyname("ftp", "tcp");
    printf("ftp/tcp: %d\n",
             ntohs(svc->s_port));

    svc = getservbyport(htons(517), "udp");
    printf("517/udp: %s\n", svc->s_name);

    for (int i = 0; i < 20; i++) {
        svc = getservent();
        printf("%3d: %10s %s\n",
               ntohs(svc->s_port),
               svc->s_name, svc->s_proto);
    }
}
```

Resolver

Resolver: Einfach aufgebaute Software-Module des DNS (domain name service), die Informationen vom Nameserver abrufen können. Sie bilden die Schnittstelle zwischen Anwendung und Nameserver[1].

`struct hostent *gethostbyname(const char *name)` liefert eine Struktur `hostent` mit Informationen über den Host mit dem Namen `name`.

```
struct hostent {
    char  *h_name;          // official name of host
    char **h_aliases;       // alias list
    int    h_addrtype;      // host address type
    int    h_length;        // length of address
    char **h_addr_list;     // list of addresses
}
```

[1] http://de.wikipedia.org/wiki/Domain_Name_System#Resolver

Resolver

```
#include <stdio.h>        // printf()
#include <netdb.h>        // gethostbyname()
#include <arpa/inet.h>    // inet_ntoa()

int main(void) {
    struct hostent *host;
    char **alias;

    host = gethostbyname("www.hsnr.de");

    printf("Official name: %s\n", host->h_name);
    for (alias = host->h_aliases;
                        *alias != 0; alias++) {
        printf("alternative name: %s\n",
               *alias);
    }
```

```
printf("Address type: ");
switch (host->h_addrtype) {
case AF_INET:
    printf("AF_INET\n");
    break;
case AF_INET6:
    printf("AF_INET6\n");
    break;
case AF_IPX:
    printf("AF_IPX\n");
    break;
default:
    printf(" %d\n", host->h_addrtype);
    break;
}
printf("Address length: %d\n",
       host->h_length);
```

```
    if (host->h_addrtype == AF_INET) {
        int i = 0;
        while (host->h_addr_list[i] != 0) {
            struct in_addr addr;

            addr.s_addr = *(u_long *)
                    host->h_addr_list[i++];
            printf("IPv4 Address: %s\n",
                    inet_ntoa(addr));
        }
    }

    return 0;
}
```

Die Adressen werden in der Struktur `in_addr` in einer binären Form abgelegt. Die Funktion `inet_ntoa()` wandelt die binäre Form in die bekannte Numbers-And-Dots-Notation um.

Network Byte Order

`int inet_aton(const char *cp, struct in_addr *inp)`
converts the Internet host address cp from the standard numbers-and-dots notation into binary data and stores it in the structure that `inp` points to.

`char *inet_ntoa(struct in_addr in)`
converts the Internet host address given in network byte order into a string in standard numbers-and-dots notation. The string is returned in a statically allocated buffer, which subsequent calls will overwrite.

`in_addr_t inet_addr(const char *cp)`
converts the Internet host address cp from numbers-and-dots notation into binary data in network byte order. If the input is invalid, `INADDR_NONE` is returned. This is an obsolete interface to `inet_aton`.

Network Byte Order

Aus dem Grundstudium wissen wir, dass Zahlen auf den Rechnern in binärer Form gespeichert werden. Leider nicht einheitlich:

- Little Endian (least significant byte first): Einsatz bei dem legendären Prozessor 6502 des C64 Home-Computers, der NEC-V800-Reihe oder den Intel-x86-Prozessoren.
- Big-Endian (most significant byte first): Einsatz bei der Motorola-68000-Familie, den Prozessoren von IBM der System-z-Reihe, den SPARC-CPUs von SUN und dem PowerPC.

Computer verschiedener Plattformen können nur dann fehlerfrei kommunizieren, wenn bei den Netzwerkprotokollen die Byte-Reihenfolge festgeschrieben ist. Diese *Network Byte Order* ist bei TCP-IP festgelegt auf das Big-Endian-Format.

Network Byte Order

Die Byte-Reihenfolge des Systems wird als *Host Byte Order* bezeichnet. Wir haben in der Aufzählung oben gesehen, dass die x86-Prozessoren von Intel mit dem Little-Endian-Format arbeiten.

Daher muss bei den Rechnern in unseren Arbeitsräumen diese im Anwendungsprogramm ggf. umgewandelt werden. Die Bibliothek `arpa/inet.h` definiert dazu einige Funktionen:

- `uint32_t htonl(uint32_t hlong)` converts the unsigned integer `hlong` from host to network byte order.
- `uint16_t htons(uint16_t hshort)` converts the unsigned short integer `hshort` from host to network byte order.
- `uint32_t ntohl(uint32_t nlong)` converts the unsigned integer `nlong` from network to host byte order.
- `uint16_t ntohs(uint16_t nshort)` converts the unsigned short integer `nshort` from network to host byte order.

Auf dem Client:

`int connect(int fd, struct sockaddr *adr, int l)` öffnet eine Verbindung vom Socket `fd` zu einem passenden Socket auf dem Server mit der Adresse `adr`. Da unterschiedliche Typen von Strukturen bei `adr` angegeben werden können, muss die Größe der Struktur in `l` übergeben werden.

Auf dem Server:

`int listen(int sfd, int backlog)` definiert für den Socket `sfd` die Länge der Warteschlange für eingehende Verbindungen.

`int accept(int s, struct sockaddr *addr, int len)` akzeptiert Verbindung über Server-Socket `s`, erzeugt passenden Accept-Socket und gibt dessen Deskriptor zurück. `len` muss Länge der Adressstruktur enthalten. Füllt `addr` mit Informationen über anfragenden Client.

HTTP-Listener

Das folgende Programm protokolliert alle Anfragen, die auf Port 8080 (HTTP) eintreffen und schickt als Antwort `I am alive` an den Client.

Zum Testen des Programms sollte der Browser mit der URL `http://localhost:8080` aufgerufen werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define LEN     1024
#define PORT    8080
```

HTTP-Listener

```
int main(int argc, char *argv[]) {
    int cnt = 0;

    // create a server socket
    int aSocket = socket(AF_INET,
                         SOCK_STREAM, 0);
    if (aSocket == -1) {
        perror("cannot open socket ");
        exit(1);
    }
```

HTTP-Listener

```
// bind server port
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(PORT);

int r = bind(aSocket,
             (struct sockaddr *)&addr,
             sizeof(addr));
if (r == -1) {
    perror("cannot bind socket ");
    exit(2);
}
```

HTTP-Listener

```
// listen for incoming requests
listen(aSocket, 3);
int addrlen = sizeof(struct sockaddr_in);

// server loops forever
while (1) {

    // waiting for incoming requests
    int conn = accept(aSocket,
            (struct sockaddr *)&addr,
            &addrlen);

    if (conn == 0)
        continue;

    cnt += 1;
    printf("client %s is connected...\n",
          inet_ntoa(addr.sin_addr));
```

```c
// read the request
char buf[LEN];
do {
    int len = recv(conn, buf, LEN, 0);
    buf[len] = '\0';
    printf("%d: %s\n", len, buf);
} while (!strstr(buf, "\r\n\r\n") &&
         !strstr(buf, "\n\n"));
```

Der Header der Anfrage ist vom Body durch eine Leerzeile getrennt.

```
        // handle the request
        if (strstr(buf, "GET / HTTP"))
            sprintf(buf, "HTTP/1.1 200 OK\r\n"
                    "\r\nI am alive: %d", cnt);
        else sprintf(buf, "HTTP/1.1"
                     " 404 Not Found\r\n");

        send(conn, buf, strlen(buf), 0);
        close(conn);
    }
    close(aSocket);

    return 0;
}
```

HyperText Transfer Protocol

HTTP ist die gemeinsame Sprache von Web-Browser und -Server: Immer dann, wenn zwei Programme miteinander kommunizieren wollen, müssen sie sich auf eine gemeinsame Sprache einigen.

- `GET` weist den Web-Server an, den Inhalt der spezifizierten Datei an den anfordernden Web-Browser zu schicken. Als Parameter wird zum einen die gewünschte Datei, zum anderen das verwendete Protokoll angegeben.
  Beispiel: `GET /index.html HTTP/1.0`
- `HEAD` ist analog zu dem Befehl GET. Der Web-Server liefert aber nicht den Inhalt der Datei aus, sondern schickt nur den Header zurück.
- Mittels `POST` kann der Browser dem Server Informationen übermitteln, die bspw. mittels eines HTML-Formulars vom Benutzer eingegeben wurden.

HyperText Transfer Protocol

Weitere Befehle:

- `PUT` legt Dateien auf dem Web-Server ab.
- `DELETE` löscht Dateien vom Server.
- `TRACE` liefert die genaue Anfrage des Clients an den Web-Server zurück an den Client.

Dem Dateiinhalt, den der Web-Server ausliefert, geht immer ein Header voraus, wo sich Angaben wie

- der Typ des Inhalts (mime type: text/html, text/plain, image/gif, image/jpeg, usw.),
- die Größe der Datei,
- Status-Codes und ähnliches finden lassen.

Der Header ist vom Body durch eine Leerzeile getrennt.

HyperText Transfer Protocol

Lassen wir auf dem Client einen Packet-Sniffer wie `ngrep -d lo` auf dem loopback-Device mitlaufen, so können wir uns die gesamte Anfrage des Web-Browsers ansehen:

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.78 [de]
            (X11; U; Linux 2.4.10-4GB i686)
Host: localhost
Accept: image/gif, image/x-xbitmap, image/jpeg,
        image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

Auf die gleiche Art erhalten wir die Antwort des Web-Servers:

```
HTTP/1.1 200 OK
Date: Sat, 17 Nov 2001 09:15:06 GMT
Server: Apache/1.3.20 (Linux/SuSE) mod_perl/1.26
Last-Modified: Sat, 17 Nov 2001 09:10:28 GMT
ETag: "15b52b-7b-3bf62984"
Accept-Ranges: bytes
Content-Length: 123                 <--------- !!!!!
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html
                                    <--------- !!!!!
<html>
<head>
.....
```

Probleme beim Software-Entwurf

Erweiterbarkeit

- Modularer Aufbau, Kapselung und die Definition sauberer Schnittstellen unterstützen Erweiterbarkeit.
- Neue Anforderungen und Änderungen können vorausschauend beim Entwurf der Software berücksichtigt werden: Soll der Server zukünftig auch Protokolle wie ftp oder https unterstützen?

Wiederverwendbarkeit

- Einmal geschriebener Code soll möglichst wiederverwendet werden können. Das ermöglicht den Aufbau von Bibliotheken, wodurch die Kosten-Nutzen-Relation verbessert wird.
- Häufig wird Code nur durch Copy & Paste wiederverwendet.

Probleme beim Software-Entwurf

Unabhängigkeit von Hardware- und Software-Plattformen

- Bei modernen Programmiersprachen wird die Entkopplung der Programme von Hardware und Betriebssystem durch eine Laufzeitumgebung gewährleistet:
  - Java Virtual Machine von SUN, IBM, Blackdawn und anderen
  - Common Language Runtime von Microsoft für C#
- Bei Sprachen wie C++ sollten wir die API-Aufrufe kapseln, damit eine Portierung auf andere Systeme vereinfacht wird.

Unabhängigkeit von Algorithmen

- Die von einem Algorithmus abhängigen Objekte sollten nicht geändert werden müssen, wenn sich der Algorithmus ändert.

Ursachen für Entwurfsrevisionen

Erzeugen eines Objekts unter expliziter Nennung seiner Klasse.

- Dadurch legt man sich auf eine bestimmte Implementierung statt einer bestimmten Schnittstelle fest.
- Lösung: Objekte indirekt erzeugen. (Abstrakte Fabrik, Fabrikmethode, Prototyp, ...)

Abhängigkeiten von spezifischen Operationen.

- Durch Angabe einer bestimmten Operation legt man sich auf genau einen Weg fest, eine Anfrage zu befriedigen.
- Lösung: Operationen werden in Klassen gekapselt und nicht fest codiert. (Zuständigkeitskette, Befehl, ...)

Ursachen für Entwurfsrevisionen

Enge Kopplung

- Eng miteinander gekoppelte Klassen können nur schwer alleinstehend wiederverwendet werden, weil sie voneinander abhängig sind.
- Lösung: Abstrakte Kopplung und Schichtenbildung (Brücke, Fassade, Vermittler, Beobachter, ...)

Erweiterung der Funktionalität durch Unterklassenbildung

- Das Überschreiben einer Operation erzwingt oft das Überschreiben einer weiteren Operation, selbst einfache Erweiterungen führen so evtl. zu vielen neuen Unterklassen.
- Lösung: Objektkomposition und Delegation (Kompositum, Dekorierer, Strategie, ...)

Entwurf eines Servers

Design-Alternativen:

- Unterschiedliche Benutzeranforderungen bzgl.
  - traffic workload,
  - Hardware-/Software-Plattformen,
  - Ausfallsicherheit,
  - Konfigurierbarkeit usw.
- erfordern ein flexibles Design:
  - Concurrency Model: single threaded, multi threaded, thread-per-request, thread pool, ...
  - File Caching Strategy: least recently used, least frequently used, ...
  - Content Delivery Protocol: HTTP 1.0, HTTP 1.1, HTTPS, ...

Entwurfsmuster Wrapper Facade

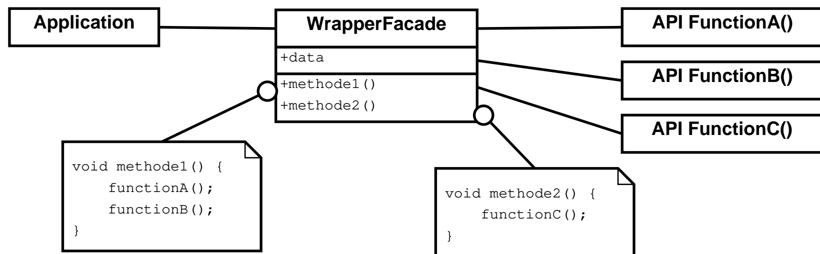
Zweck: Encapsulating Low-level Operating System APIs

Motivation:

- Software, die direkt Funktionen des Betriebssystems aufruft, ist nicht plattformunabhängig und nur schwer portierbar.
- Wir führen daher eine Zwischenschicht ein, die der Applikation eine definierte Schnittstelle bereitstellt.
- Die Implementierung der Schnittstelle kann dann für jedes Betriebssystem, unabhängig von der Anwendung, erstellt werden.

Entwurfsmuster aus dem Bereich Service Access & Configuration

Entwurfsmuster Wrapper Facade



Teilnehmer:

- API Funktionen stellen in der Regel einen Dienst mittels einer wohldefinierten Schnittstelle zur Verfügung, wie bspw. die Socket-API in C.
- Die WrapperFacade ist eine Menge von Klassen, die bestehende Funktionen und Daten kapselt, um einen direkten Aufruf von Betriebssystem-APIs oder Bibliotheken zu vermeiden.

Entwurfsmuster Wrapper Facade

- Sowohl die Java Virtual Machine als auch viele Java class libraries kapseln Betriebssystemaufrufe und GUI-APIs.
- Die Microsoft Foundation Classes kapseln Win32-APIs, der Schwerpunkt liegt auf GUI-Elementen, die die Microsoft Document-View Architektur implementieren.
- Das Paket `java.net` enthält Klassen, die den Zugriff auf Netzwerke kapseln. Auszug aus `PlainSocketImpl`:

```
private native void socketConnect(InetAddress ia,
    int port, int timeout) throws IOException;
private native void socketBind(InetAddress addr,
    int port) throws IOException;
private native void socketListen(int count)
    throws IOException;
private native void socketAccept(SocketImpl s)
        throws IOException;
```

Entwurfsmuster Wrapper Facade

Anmerkungen:

- Native Methoden werden mittels shared libraries zur Verfügung gestellt. Diese befinden sich unter Linux im Verzeichnis `<javahome>/jre/lib/i386`.
- Die Library `libnet.so` enthält die Implementierung obiger Klassen.
- Mit dem Befehl `nm -D libnet.so | less` können die enthaltenen Symbole angezeigt werden.
- Die Klasse `java.net.ServerSocket` greift auf obige Implementierungen zurück.

Socket-Wrapper in C++

socket.hpp

```
#include <string>
#include <cerrno>
#include <cstring>
#include <netinet/in.h>

class Socket {
private:
    sockaddr_in _address;
    int _socket;

public:
    Socket(std::string ip, int port);
    Socket(int socket);

    void send(std::string msg);
    std::string recv(void);
    void close(void);
};
```

socket.cpp

```
#include ....

Socket::Socket(string ip, int port) {
    _socket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (_socket == -1)
        throw SocketException(strerror(errno));

    _address.sin_family = AF_INET;
    _address.sin_addr.s_addr =
        ::inet_addr(ip.c_str());
    _address.sin_port = ::htons(port);
    int len = sizeof(_address);

    int rc = ::connect(_socket,
            (struct sockaddr *) &_address, len);
    if (rc == -1)
        throw SocketException(strerror(errno));
}
```

Socket-Wrapper in C++

```
Socket::Socket(int socket) {
    _socket = socket;
}

void Socket::send(string msg) {
    msg += "]]]";          // append end-of-message
    ::write(_socket, msg.c_str(), msg.length());
    cout << "sent: " << msg << endl;
}

void Socket::close(void) {
    ::close(_socket);
}
```

```
string Socket::recv() {
    char block[256];
    string res;
    int len;

    do {  // while not end-of-message found do
        block[0] = '\0';
        len = ::read(_socket, block, 256);
        if (len != 0)
            res += block;
    } while (res.find("]]]") == string::npos);

    // extract end-of-message
    string::size_type pos = res.find("]]]");
    res = res.substr(0, pos);

    return res;
}
```

Server-Socket-Wrapper in C++

servSocket.hpp

```
#include <string>
#include <cerrno>
#include <cstring>
#include <netinet/in.h>

class ServerSocket {
private:
    sockaddr_in _address;
    int _socket;
    socklen_t _addrlen;
public:
    ServerSocket(int port, int queueSize);
    ~ServerSocket(void);

    int accept(void);
    void send(std::string msg);
    std::string recv(void);
};
```

Server-Socket-Wrapper in C++

servSocket.cpp

```
#include ....

ServerSocket::ServerSocket(int port, int qSize){
    // create a server-socket
    _socket = ::socket(AF_INET, SOCK_STREAM, 0);
    if (_socket == -1)
        throw SocketException(strerror(errno));

    // Fehler "cannot bind socket: Address
    // already in use" abfangen
    int i = 1;
    ::setsockopt(_socket, SOL_SOCKET,
                 SO_REUSEADDR, &i, sizeof(i));

    // bind server port
    _address.sin_family = AF_INET;
    _address.sin_addr.s_addr = INADDR_ANY;
    _address.sin_port = ::htons(port);
```

Server-Socket-Wrapper in C++

```
    int r = ::bind(_socket,
            (struct sockaddr *) &_address,
            sizeof(_address));
    if (r == -1)
        throw SocketException(strerror(errno));

    // listen for incoming requests
    ::listen(_socket, qSize);
    _addrlen = sizeof(struct sockaddr_in);
}

ServerSocket::~ServerSocket() {
    ::close(_socket);
}
```

Server-Socket-Wrapper in C++

```
int ServerSocket::accept(void) {
    // waiting for incoming requests
    cout << "waiting for incoming requests ..."
         << endl;
    int conn = ::accept(_socket,
                (struct sockaddr *) &_address,
                &_addrlen);

    if (conn == -1)
        throw SocketException(strerror(errno));

    cout << "----> accept socket: " << conn
         << endl;
    return conn;
}
```

ECHO-Server in C++

main.cpp

```
#include "socket.hpp"
#include "servSocket.hpp"

int main(int argc, char **argv) {
    ServerSocket server(6200, 10);

    while (1) {
        int conn = server.accept();
        Socket acceptSocket(conn);
        std::string req = acceptSocket.recv();
        acceptSocket.send("ECHO REPLY: " + req);
        acceptSocket.close();
    }
    return 0;
}
```

Hinweis: Obige Implementierung kann zu Problemen führen, wenn Client und Server unterschiedliche Codierungen für Strings nutzen.

Paralleler Server

Schauen wir uns ein weiteres Beispiel an. Wir wollen einen Dienst auf einem Server bereit stellen, wobei der Dienst verschiedene Methoden anbietet:

- `echo` schickt genau die Zeichenketten zurück, die gesendet wurde.
- `date` schickt das Systemdatum an den Aufrufer zurück.
- `time` schickt die aktuelle Systemzeit an den Aufrufer zurück.
- `random` schickt eine zufällige Zahl an den Aufrufer zurück.
- `prime` testet, ob die gesendete Zahl eine Primzahl ist.

Damit Anfragen mehrerer Clients gleichzeitig bearbeitet werden können, wird die Bearbeitung einer Anfrage in einen Thread ausgelagert. $\rightarrow$ Thread-per-Request

```c
#include <stdio.h>
#include .....

// ******************************************
typedef struct {
    int conn;
    char request[1024];
} arg_t;

// ******************************************
void handleEcho(arg_t *arg) {
    char reply[1024];

    strncpy(reply, arg->request, 1024);
    strncat(reply, "]]]", 1024);
    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

```
// ******************************************
void handleDate(arg_t *arg) {
    time_t now = time(NULL);
    struct tm *today = localtime(&now);
    char reply[1024];

    sprintf(reply, "%02d.%02d.%04d]]]",
            today->tm_mday,
            today->tm_mon + 1,
            today->tm_year + 1900);
    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

```
// ****************************************
void handleTime(arg_t *arg) {
    time_t now = time(NULL);
    struct tm *today = localtime(&now);
    char reply[1024];

    sprintf(reply, "%02d:%02d:%02d]]]",
            today->tm_hour,
            today->tm_min,
            today->tm_sec);
    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

Paralleler Server

```
// ************************************
void handleRandom(arg_t *arg) {
    char reply[1024];

    sprintf(reply, "%d]]]", rand());
    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

```c
// ***************************************
void handlePrime(arg_t *arg) {
    char reply[1024];
    char prime = 1;
    int val;

    sscanf(arg->request, "PRIME %d", &val);
    for (int i=2; prime && i <= sqrt(val); i++)
        if (val % i == 0)
            prime = 0;

    if (prime)
        sprintf(reply, "true]]]");
    else sprintf(reply, "false]]]");

    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

```
// ***************************************
void handleError(arg_t *arg) {
    char reply[1024];

    strncpy(reply, "unknown service: ", 1024);
    strncat(reply, arg->request, 1024);
    strncat(reply, "]]]", 1024);
    write(arg->conn, reply, strlen(reply));
    free(arg);
}
```

Paralleler Server

```
// ***************************************
void handleRequests(void) {
    int len;
    unsigned int addrlen;
    int port = 6200;
    int aSocket, conn;
    char *p, buf[256], msg[1024];
    struct sockaddr_in serverAddr;
    pthread_t thread;

    // create a socket
    aSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (aSocket == -1) {
        perror("cannot open socket ");
        exit(1);
    }
```

Paralleler Server

```c
// Fehler "Address already in use" abfangen
int i = 1;
setsockopt(aSocket, SOL_SOCKET,
           SO_REUSEADDR, &i, sizeof(i));

// bind server port
serverAddr.sin_family = AF_INET;
serverAddr.sin_addr.s_addr = INADDR_ANY;
serverAddr.sin_port = htons(port);

int r = bind(aSocket,
             (struct sockaddr *)&serverAddr,
             sizeof(serverAddr));
if (r == -1) {
    perror("cannot bind socket ");
    exit(2);
}
```

Paralleler Server

```
// listen for incoming requests
listen(aSocket, 8);
addrlen = sizeof(struct sockaddr_in);

// server loops forever
while (1) {

    // waiting for incoming requests
    conn = accept(aSocket,
           (struct sockaddr *)&serverAddr,
           &addrlen);

    if (conn == -1) {
        perror("Server: ");
        continue;
    }
    printf("\n%s is connected...\n",
           inet_ntoa(serverAddr.sin_addr));
```

Paralleler Server

```
// read the request
msg[0] = '\0';
do {
    buf[0] = '\0';
    len = recv(conn, buf, 256, 0);
    strncat(msg, buf, len);
} while (!strstr(msg, "]]]"));

// remove "]]]" from message
p = strstr(msg, "]]]");
while (*p == ']') {
    *p = '\0';
    p++;
}
```

```c
// split request in service and args
printf("handle %s request ...\n", msg);
arg_t *arg = (arg_t *)
        malloc(sizeof(arg_t));
arg->conn = conn;
strcpy(arg->request, msg);

int err;

if (strstr(buf, "ECHO"))
    err = pthread_create(&thread, NULL,
          (void *(*)(void *))handleEcho,
          arg);
else if (strstr(buf, "DATE"))
    err = pthread_create(&thread, NULL,
          (void *(*)(void *))handleDate,
          arg);
// .....
```

```
            else err = pthread_create(&thread, NULL,
                    (void *(*)(void *))handleError,
                    arg);

            if (err) {
                fprintf(stderr,
                        "couldn't create thread\n");
                exit(2);
            }
        }
        close(aSocket);
}

// ***************************************
int main(int argc, char **argv) {
        handleRequests();
        return 0;
}
```

Anmerkung: Die Funktion `pthread_create` erwartet als dritten Parameter einen Zeiger auf eine Funktion vom (Rückgabe-) Typ `void *`, die nur einen Parameter vom Typ `void *` hat.

Da die Funktionen wie `handleEcho` aber einen Parameter vom Typ `arg_t *` haben, nutzen wir eine explizite Typumwandlung um Warnungen des Compilers zu vermeiden.

```
(  void *   (*)  (void *)  )
```

Parameter der Funktion

Zeiger auf Funktion

(Rückgabe–) Typ der Funktion

explizite Typumwandlung

Test-Client

Da der Server in einer Endlosschleife läuft, gibt es im Umgang mit den Threads kein Problem. Anders ist das beim Client zum Testen des Servers:

- Der Client startet für jede Anfrage einen neuen Thread mittels der Funktion `pthread_create()`.
- Endet das Hauptprogramm, bevor ein Thread abgearbeitet ist, wird der Thread automatisch beendet. Das hätte in unserem Programm zur Folge, dass die Antwort vom Server nicht mehr empfangen wird und der Server die Antwort nicht verschicken kann.
- Daher muss das Hauptprogramm warten, bis alle Threads beendet wurden. Dafür steht die Funktion `pthread_join()` zur Verfügung.

Test-Client

```
#include <stdio.h>
#include .....

#define N         12
#define MSG_LEN  256

// ***********************************
void doRequest(char *request) {
    int aSocket;
    int len, res;
    struct sockaddr_in address;
    char reply[MSG_LEN], block[MSG_LEN];

    aSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (aSocket == -1) {
        perror("socket(): ");
        exit(1);
    }
```

Test-Client

```
// connect to server
address.sin_family = AF_INET;
address.sin_addr.s_addr =
        inet_addr("127.0.0.1");
address.sin_port = htons(6200);
len = sizeof(address);

res = connect(aSocket,
        (struct sockaddr *)&address, len);
if (res == -1) {
    perror("server 127.0.0.1: ");
    exit(1);
}
```

```
    // send request to server
    printf("request: %s\n", request);
    write(aSocket, request, strlen(request));

    // get answer from server
    reply[0] = '\0';
    do {
        block[0] = '\0';
        res = read(aSocket, block, MSG_LEN);
        strncat(reply, block, res);
    } while (!strstr(reply, "]]]"));
    printf("reply from server for message"
           " %s: %s\n", request, reply);

    close(aSocket);
}
```

Test-Client

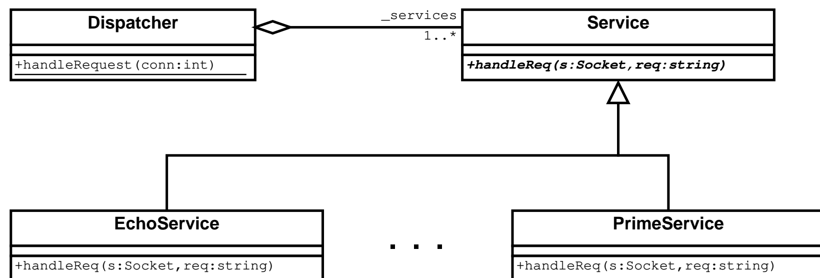
```
int main(int argc, char **argv) {
    char args[N][32];
    pthread_t thread[N];

    for (int i = 0; i < N; i++) {
        if (i % 6 == 0)
            sprintf(args[i], "ECHO]]]");
        else if (i % 6 == 1)
            sprintf(args[i], "DATE]]]");
        else if (i % 6 == 2)
            sprintf(args[i], "TIME]]]");
        else if (i % 6 == 3)
            sprintf(args[i], "RANDOM]]]");
        else if (i % 6 == 4)
            sprintf(args[i], "PRIME %d]]]",
                    rand());
        else sprintf(args[i], "blubb]]]");
```

Test-Client

```
        int err = pthread_create(&thread[i],
                NULL,
                (void *(*)(void *))doRequest,
                args[i]);

        if (err) {
            fprintf(stderr, "couldn't create"
                    " thread %d\n", i);
            exit(2);
        }
    }

    // wait for the threads to finish
    for (int i = 0; i < N; i++)
        pthread_join(thread[i], &res);

    return 0;
}
```

Paralleler Server in C++

Wir wollen nun diesen zuletzt betrachteten Server in C++ schreiben und Polymorphie nutzen, um die einzelnen Dienste bereitzustellen.



Da wir die Methode `handleRequest` der Klasse `Dispatcher` in einem Thread nutzen wollen, muss die Methode als `static` deklariert sein.

Paralleler Server in C++

dispatcher.hpp

```
#include "socket.hpp"
#include <string>
#include <map>

// *****************************************
class Service {
public:
    virtual void handleReq(Socket s,
            std::string req) = 0;
};

class EchoService : public Service {
    void handleReq(Socket s, std::string req);
};
.....
```

```
.....  // weitere Service-Klassen

class PrimeService : public Service {
    void handleReq(Socket s, std::string req);
};

// ************************************
class Dispatcher {
private:
    static std::map<std::string, Service *>
            _services;

public:
    static void handleRequest(int sock);
};
```

dispatcher.cpp

```
#include "dispatcher.hpp"
.....
void EchoService::handleReq(Socket s,
                            string req) {
    s.send(req);
    s.close();
}

void RandomService::handleReq(Socket s,
                              string req) {
    int val = stoi(req);

    string answ = to_string(rand() % val);
    s.send(answ);
    s.close();
}
```

```
void DateService::handleReq(Socket s,
                             string req) {
    time_t now = time(NULL);
    tm *today = localtime(&now);

    ostringstream os;
    os << setw(2) << setfill('0')
       << today->tm_mday << ".";
    os << setw(2) << setfill('0')
       << today->tm_mon + 1 << ".";
    os << setw(4) << setfill('0')
       << today->tm_year + 1900;

    s.send(os.str());
    s.close();
}

.....  // weitere Implementierungen der Dienste
```

Paralleler Server in C++

```
// initialize static map
std::map<std::string, Service *>
Dispatcher::_services = {
    {"ECHO", new EchoService()},
    {"TIME", new TimeService()},
    {"DATE", new DateService()},
    {"RANDOM", new RandomService()},
    {"PRIME", new PrimeService()}
};

// -------------------------------------------
void Dispatcher::handleRequest(int conn) {
    Socket sock(conn);

    string req = sock.recv();
```

Paralleler Server in C++

```
// split request into service and args
Tokenizer tok(req, ":");  // see OOA
if (tok.countTokens() <= 1) {
    string msg = "unknown protocol!";
    sock.send(msg);
    sock.close();
    return;
}

string service = tok.nextToken();
string param = tok.nextToken();
```

```
        try {
            Service *svc = _services.at(service);
            svc->handleReq(sock, param);
        } catch (const out_of_range &e) {
            // cout << e.what() << endl;
            string msg = "unknown service!";
            sock.send(msg);
            sock.close();
        }
    }
```

Weitere Dienste können einfach hinzugefügt werden. Es muss nur eine weitere Unterklasse definiert und zusammen mit einem Schlüsselwort in der `_services`-Map eingetragen werden.

Paralleler Server in C++

server.cpp

```
#include "servSocket.hpp"
#include "dispatcher.hpp"
#include <thread>

int main(int argc, char **argv) {
   ServerSocket server(6200, 10);

   while (1) {
      std::thread t(Dispatcher::handleRequest,
                    server.accept());
      t.detach(); // detach from calling thread
   }
}
```

`detach()` erlaubt beiden Threads eine voneinander unabhängige Ausführung. Belegte Ressourcen werden nach Beendigung des Threads automatisch freigegeben. Die Threads sind nicht mehr joinable.

Parallele Server

Thread-per-Request ist nicht immer sinnvoll:

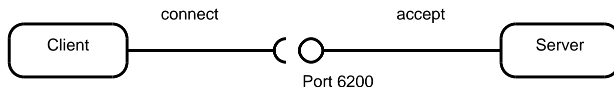
- Ineffizient und nicht skalierbar aufgrund von Kontextwechseln, Synchronisation und Austausch von Daten zwischen CPUs.
- Komplizierte Kontrolle der Nebenläufigkeit um Zugriffe auf gemeinsam genutzte Daten zu organisieren.
- Aufteilung der Threads auf Ressourcen wie CPUs oder CPU-Kerne erscheint sinnvoller als eine Zuordnung auf Requests. $\Rightarrow$ Thread-Pool nutzen!

Thread-Pool:

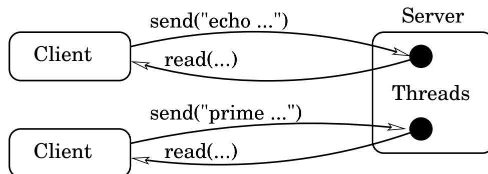
- Beim Start des Servers wird eine bestimmte Anzahl von Threads bereits erzeugt, aber direkt schlafen gelegt.
- Erst wenn ein Thread aus dem Pool entnommen wird, wird der Thread geweckt und verrichtet seine Arbeit.
- Wurde die Anfrage bearbeitet, wird der Thread wieder in den Pool eingefügt und schlafen gelegt.

Die Funktion `accept` blockiert den aufrufenden Prozess, daher

- kann nur auf einem Socket auf Anfragen gewartet werden

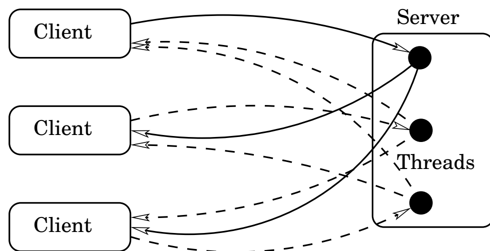


- und mehrere Clients werden mittels weiterer Threads versorgt.



Ist diese Struktur geeignet für ein Programm, bei dem sich beliebig viele Clients bei einem Server melden können und jede Nachricht eines Clients sofort an alle andere Clients verteilt wird?

Die Kommunikation beim Chat besteht für einen langen Zeitraum, und jeder Thread muss mit jedem Client kommunizieren können. Dazu muss jeder Thread die Sockets kennen, mit denen die Clients erreicht werden können.



Die Sockets müssen also in einer gemeinsam genutzten Liste verwaltet werden. Schicken zwei Clients gleichzeitig eine Nachricht, muss der Zugriff der zwei zugeordneten Threads auf diese Liste synchronisiert werden.

select

Wesentlich einfacher zu realisieren ist ein sequentieller Server, der trotzdem gleichzeitg auf verschiedenen Sockets auf Nachrichten wartet.

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout)
```

- Wartet auf Status-Änderungen bei beliebig vielen File-Deskriptoren bzw. Anfragen bei Sockets. (Bsp: `inetd`)
- Funktion blockiert, bis über einen Deskriptor `readfds`, `writefds` oder `exceptfds` Daten gelesen oder geschrieben werden können bzw. eine Datei ihren Status ändert.

Die Nachrichten können nacheinander bearbeitet werden, so dass der Aufwand zur Synchronisation entfällt.

select

```
int select(int nfds, fd_set *readfds, ...,
           struct timeval *timeout)
```

- Liefert die Anzahl der Deskriptoren, die das Ende von `select` ausgelöst haben.
- Handelt es sich um Client-Anfragen, können diese mit `accept` angenommen werden, ohne dass `accept` blockiert.
- Der Parameter `timeout` begrenzt die Wartezeit. Wird der Parameter auf 0 gesetzt, ist die Wartezeit unbegrenzt.

Manipulation der Deskriptor-Mengen mittels Makros:

- `FD_ZERO(fd_set *set)` leeren,
- `FD_SET(int fd, fd_set *set)` Deskriptor eintragen,
- `FD_CLR(int fd, fd_set *set)` Deskriptor austragen,
- `FD_ISSET(int fd, fd_set *set)` testet, ob der angegebene Deskriptor das Ende von `select` ausgelöst hat.

Single Threaded Server

server.cpp

```
#include <iostream>
.....
class Server {
private:
    int _minFD, _maxFD;
    int _servSockfd;
    fd_set _actorFDset;
    set<int> _clients;

    void addClient(int sockfd);
    void delClient(int sockfd);
    int getPostingClient(fd_set *fdSet);
    void sendToAllClients(string msg, int em);
public:
    Server(int servSockfd);
    void run(void);
};
```

Single Threaded Server

```
Server::Server(int servSockfd) {
    _servSockfd = servSockfd;
    FD_ZERO(&_actorFDset);
    FD_SET(_serverSockfd, &_actorFDset);
    _minFD = _maxFD = _serverSockfd;
}

void Server::addClient(int sockfd) {
    cout << "new Client on Socket " << sockfd
         << " connected!" << endl;

    if (sockfd > _maxFD)
        _maxFD = sockfd;
    if (sockfd < _minFD)
        _minFD = sockfd;
    FD_SET(sockfd, &_actorFDset);
    _clients.insert(sockfd);
}
```

Single Threaded Server

```
void Server::delClient(int sockfd) {
    cout << "Client on Socket " << sockfd
         << " has closed connection!\n";

    if (sockfd == _maxFD)
        _maxFD -= 1;
    if (sockfd == _minFD)
        _minFD += 1;
    FD_CLR(sockfd, &_actorFDset);

    _clients.erase(sockfd);
}
```

Single Threaded Server

```
int Server::getPostingClient(fd_set *fdSet) {
    int i = _minFD;
    while (!FD_ISSET(i, fdSet))
        i++;
    return i;
}

void Server::sendToAllClients(string msg,
                              int emitter) {
    set<int>::iterator iter = _clients.begin();
    for ( ; iter != _clients.end(); iter++) {
        int sockfd = *iter;
        if (sockfd == emitter)
            continue;
        Socket sock(sockfd);
        sock.send(msg);
    }
}
```

```
void Server::run(void) {
    while (true) {
        // reset the FDset, because select will
        // modify it to indicate which file
        // descriptors actually changed status
        fd_set fdSet = _actorFDset;

        int err = select(_maxFD + 1, &fdSet,
                         NULL, NULL, NULL);

        // if a signal has interrupted the
        // select, do it once again
        if (err == EINTR)
            continue;
```

```
    if (FD_ISSET(_servSockfd, &fdSet)) {
    // if a client connected on server socket
       int c = ::accept(_servSockfd, NULL, 0);
       addClient(c);
    } else {
    // otherwise send message to all clients
       int clientfd = getPostingClient(fdSet);
       Socket sock(clientfd);
       string msg = sock.recv();
       if (msg == "")
           // client has disconnected
           delClient(clientfd);
       else sendToAllClients(msg, clientfd);
    }
 } // end of while
} // end of Server::run()
```

Single Threaded Server

```
int main(void) {
    ServerSocket servSock(6200, 10);
    Server serv(servSock.getSocketFD());
    serv.run();
}
```

Anmerkungen:

- Zu Beginn wird mittels `select` nur der Server-Socket überwacht.
- Sobald ein Client den Server kontaktet,
  - stellt `select` den Versuch des Verbindungsaufbaus fest,
  - wird mittels `accept` die Verbindung angenommen, ohne zu blockieren,
  - der accept-Socket des neuen Clients wird in die Menge `_clients` und das `_actorFDset` aufgenommen.
- Im nächsten Durchlauf der Schleife wird dann ein weiterer Socket mittels `select` überwacht.

Receiver als Client

receiver.cpp

```
#include "socket.hpp"
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char **argv) {
    string host = "127.0.0.1";
    Socket sock(host, 6200);

    while (1) {
        try {
            string msg = sock.recv();
            cout << msg << endl;
        } catch (SocketException e) {
            cout << e.getError() << endl;
        }
    }
}
```

Emitter als Client

emitter.cpp

```
#include "socket.hpp"
#include <iostream>
.....

int main(int argc, char **argv) {
    string host = "127.0.0.1";
    string req[6] = { "Hallo!", .... };

    Socket sock(host, 6200);
    for (int j = 0; j < 6; j++) {
        try {
            sock.send(req[j]);
        } catch (SocketException e) {
            cout << e.getError() << endl;
        }
    }
    sock.close();
}
```

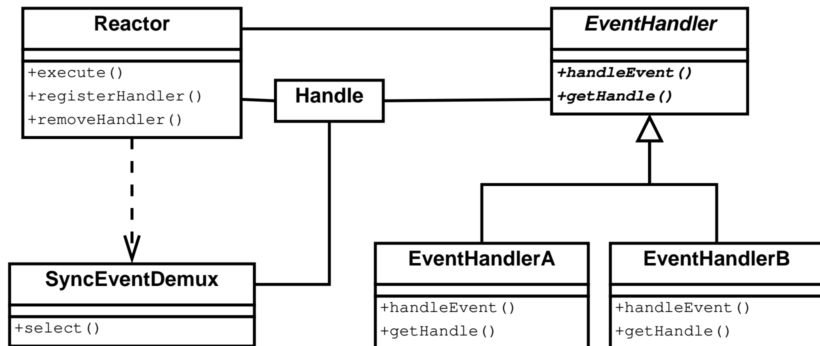
Entwurfsmuster Reactor

Bisher sind Event-Demultiplexing und Connection-Management zu eng mit dem Event-Handling verknüpft. Diese Code-Teile können daher nur schwer wiederverwendet werden, falls der Server weitere Aktionen ausführen soll.

- Event-Demultiplexing und Connection-Management von der Protokollverarbeitung entkoppeln. Das erhöht die Wiederverwendbarkeit und vereinfacht die Erweiterbarkeit.
- Der Reactor (auch Dispatcher oder Notifier genannt) nimmt die Anfragen an, wertet sie aus und leitet die Anfrage an den zuständigen Protocol-Handler weiter.
- Damit der Server verschiedene Methoden unterstützen kann, definiert der Reactor eine Schnittstelle, mit der beliebige Handler registriert bzw. wieder entfernt werden können.
- Der Protocol-Handler bearbeitet die Anfragen: parsen, loggen, verschicken der Antwort und freigeben der nicht mehr benötigten Ressourcen.

Betrachten wir noch einmal unseren Dienst, der die Methoden `echo`, `date`, `time`, `random` und `prime` bereit stellt.

Wir wollen den Parallelen Server von Thread-per-Request auf Single-Threaded mittels `select` so ändern, dass jeder Dienst auf einem eigenen Port läuft.



Teilnehmer:

- Der SynchronousEventDemultiplexer kapselt eine Funktion der Socket-API. Die Funktion `select` wartet auf Events auf einer Menge von Handles (hier Ports) und wird in einer Wrapper-Klasse gekapselt.
- Handles werden vom Betriebssystem bereitgestellt und identifizieren Ressourcen wie Netzwerkverbindungen oder Dateien. Auch die Handles werden durch Wrapper-Klassen gekapselt. Tritt ein Ereignis bei einer Ressource auf, so wird das Event gepuffert und das Handle als bereit markiert.

Entwurfsmuster Reactor

Teilnehmer:

- Ein Event-Handler spezifiziert eine Schnittstelle aus einer oder mehreren Methoden.
- Ein konkreter Event-Handler ist eine Spezialisierung des Event-Handlers und implementiert einen speziellen Dienst, den die Anwendung bereitstellt.

  Der konkrete Event-Handler ist mit einem Handle (hier Port) assoziiert, das den Dienst identifiziert.
- Die Applikation kann beim Reactor beliebige Event-Handler und deren zugeordneten Handles registrieren bzw. entfernen.

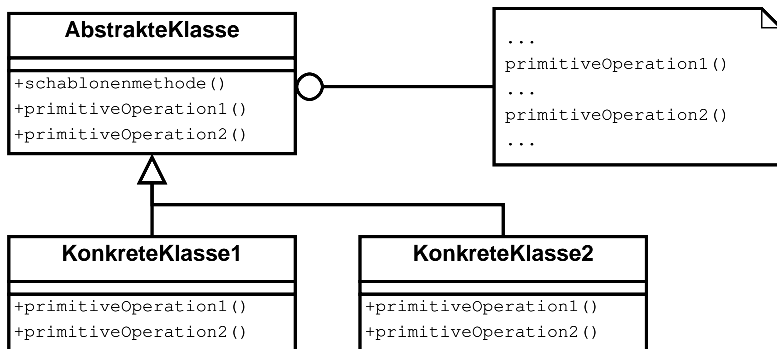
  Der Reactor wartet auf Events (hier TCP-Connections) und reicht das Event an den entsprechenden Handler weiter, der dieses Event bearbeitet.

Entwurfsentscheidung: Welche Methoden soll die Schnittstelle Handler enthalten?

- *Single-method dispatch strategy:* Die Schnittstelle definiert nur eine einzige Methode, bspw. `handleEvent` oder `execute`.
  - Pro: Weitere Handler können einfach hinzugefügt werden, da die Schnittstelle sehr einfach ist.
  - Con: Komplizierte `if/else-` bzw. `switch/case`-Konstrukte (unterscheide GET, POST, PUT, usw.)
  - Con: Für jeden Handler muss die Methode neu geschrieben werden. Lösung: Schablonenmethode (nächste Folie)
- *Multi-method dispatch strategy:* Die Schnittstelle definiert mehrere Methoden, bspw. `read`, `write` und `delete`.
  - Pro: Einzelne Methoden können in einem Generic-Handler bereitgestellt und ggf. von Unterklassen der Handler überschrieben werden.
  - Con: Können gemeinsame Funktionalitäten gefunden werden?

Schablonenmethode

- Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen.
- Die Verwendung von Schablonenmethoden ermöglicht es den Unterklassen, bestimmte Schritte des Algorithmus zu überschreiben, ohne seine Struktur zu ändern.



dispatcher.hpp

```
#include "socket.hpp"
#include <string>
#include <map>

// ******************************************
class Service {
public:
    virtual void handleRequest(int fd) = 0;
    virtual int getPort(void) = 0;
};

// ------------------------------------------
class EchoService : public Service {
    void handleRequest(int fd);
    int getPort(void);
};

..... // weitere Service-Unterklassen
```

Entwurfsmuster Reactor in C++

```
// ***********************************************
class Dispatcher {
private:
    fd_set _rfds;
    int _minFD, _maxFD;
    std::map<int, Service *> _services;

public:
    Dispatcher(void);
    void execute(void);
    void addHandler(Service *service);
};
```

dispatcher.cpp

```
#include "dispatcher.hpp"
#include <iostream>
.....

// ****************************************
void EchoService::handleRequest(int fd) {
    Socket sock(::accept(fd, NULL, 0));
    string req = sock.recv();

    sock.send(req);
    sock.close();
}

int EchoService::getPort(void) {
    // should come from /etc/services
    return 6200;
}
```

```
// ------------------------------------------
void RandomService::handleRequest(int fd) {
    Socket sock(::accept(fd, NULL, 0));
    string req = sock.recv();

    int val = stoi(req);

    string msg = to_string(rand() % val);
    sock.send(msg);
    sock.close();
}

int RandomService::getPort(void) {
    // should come from /etc/services
    return 6201;
}

..... // weitere Service-Unterklassen
```

Entwurfsmuster Reactor in C++

```
Dispatcher::Dispatcher(void) {
    _minFD = INT_MAX;
    _maxFD = INT_MIN;
    FD_ZERO(&_rfds);
}

void Dispatcher::addHandler(Service *service) {
    ServerSocket sock(service->getPort(), 10);
    int sockfd = sock.getSocketFD();

    _services.insert(pair<int, Service *>
                     (sockfd, service));
    FD_SET(sockfd, &_rfds);
    if (sockfd < _minFD)
        _minFD = sockfd;
    if (sockfd > _maxFD)
        _maxFD = sockfd;
};
```

```
void Dispatcher::execute(void) {
    cout << "waiting for connections ...\n";
    while (1) {
        fd_set rfds = _rfds;

        int err = select(_maxFD + 1, &rfds,
                         NULL, NULL, NULL);
        if (err == EINTR) continue;

        map<int, Service *>::iterator iter;
        for (int i = _minFD; i <= _maxFD; i++) {
            if (FD_ISSET(i, &rfds)) {
                iter = _services.find(i);
                iter->second->handleRequest(i);
            }
        }
    }
}
```

Entwurfsmuster Reactor in C++

server.cpp

```
#include "dispatcher.hpp"

int main(int argc, char **argv) {
    Dispatcher reactor;

    reactor.addHandler(new EchoService());
    reactor.addHandler(new DateService());
    reactor.addHandler(new TimeService());
    reactor.addHandler(new RandomService());
    reactor.addHandler(new PrimeService());

    reactor.execute();
    return 0;
}
```

Auf die Klassen `SynchEventDemux` und `Handle` wurde verzichtet. Wenn wir wirklich plattformunabhängig entwickeln wollen, wären entsprechende Wrapper-Klassen zu definieren.

Übersicht

Programmiermodelle und Entwurfsmuster

- Nachrichtenbasiert: Sockets
- *Auftragsorientiert: RPC*
- Objektorientiert: CORBA, Java RMI
- Web-Services

Remote Procedure Call – RPC

Nachteile einer auf Sockets basierenden Kommunikation:

- Bei UDP muss die Unzuverlässigkeit des Protokolls in der Anwendungsschicht ausgeglichen werden. Bei TCP sind Verbindungsaufbau und -abbau explizit zu programmieren.
- Beim Server müssen Aufgaben in Threads ausgelagert werden und Parallelität und Mechanismen zur Synchronisation müssen explizit programmiert werden.
- Die zu übertragenen Daten müssen ggf. formatiert werden.
- Paradigmenbruch zur prozeduralen und zur objektorientierten Programmierung:
  - Multi-Threading-Systeme kommunizieren über gemeinsamen Speicher, nicht mittels Signalen oder Nachrichten.
  - Innerhalb eines Threads werden Aufgaben in Teilaufgaben zerlegt, die in Funktionen bereitgestellt oder von Klassen übernommen werden. Nachrichten zu verschicken ist unüblich.

Remote Procedure Call – RPC

Falls das System bereits in Client, Server oder verteilte Prozesse gegliedert ist, die verteilte Struktur also schon vorliegt, dann mag das Senden und Empfangen von Nachrichten bei kleinen Projekten trotz der Nachteile noch adäquat sein.

Aber wie kann eine bestehende, nicht-verteilte Anwendung auf mehrere Rechner verteilt werden?

- 1990 wurde Quellcode auf 120 Milliarden Zeilen geschätzt.[2] Die Mehrheit dieser Systeme ist in COBOL oder FORTRAN geschrieben und läuft auf Großrechnern.
- Monolithische Anwendungen: Lokale Funktionen können auf verschiedene Rechner verteilt werden, um eine Aufteilung in Schichten zu ermöglichen. Dabei unterscheiden wir nach Prozeduraufrufer (Client) und die Prozedur selbst (Server).

[2] W.M. Ulrich. The evolutionary growth of software reengineering and the decade ahead. American Programmer, 3(10), 14-20.

Remote Procedure Call – RPC

Problem: Monolithische Anwendungen

- Keine Trennung in Schichten: Datenhaltung, Verarbeitung und Präsentation sind eng gekoppelt und zentralisiert.
- Die Systemteile sind nicht getrennt, die Anwendung ist nicht modular aufgebaut oder in Komponenten gegliedert.
- Dadurch ist Skalierbarkeit, Wartbarkeit, Erweiterbarkeit oder Wiederverwendbarkeit kaum möglich.
- Im günstigsten Fall ist die Anwendung bereits mittels Prozeduren strukturiert, bei alten Cobol-Programmen ist selbst das nicht gegeben.

  Dann wird im ersten Schritt eine Zerlegung in funktionaler Sicht vorgenommen und erst in einem zweiten Schritt können zusammengehörende Funktionen zu Modulen oder Klassen zusammengefasst werden.

RPC

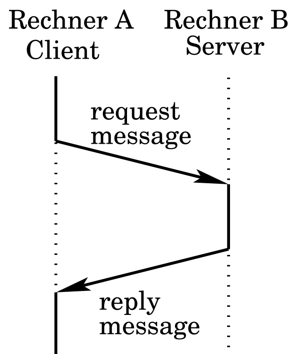
Entstehung

- RPC ist das älteste für Interprozesskommunikation eingesetzte verteilte Programmiermodell.
- Wurde schon 1984 von SUN für das Network File System NFS entwickelt, Unix-Systeme dominierten damals den Markt.
  - Das Network File System soll die Verteilung von Dateisystemen für den Anwender transparent halten.
  - Eine lokale Datei und eine „entfernte“ Datei soll für den Benutzer nicht zu unterscheiden sein.

Heute:

- RMI ist ein RPC-Mechanismus für Java.
- XML-RPC ist ein RPC-Ableger, der auf XML-Dokumenten und HTTP basiert, um RPC durch Firewalls zu realisieren. Ist für viele Sprachen wie C/C++, Java und Python verfügbar.
- Web-Services können wie RPC genutzt werden.

Aufruf einer entfernten Prozedur:



- Der aufrufende Prozess wird blockiert.
- Abarbeitung der Prozedur findet auf anderer Maschine statt.
- Parameter und Ergebnisse werden zwischen den Maschinen transportiert.
- Remote Procedure Call zeigt (fast) das vertraute Verhalten von lokalen Prozeduraufrufen.

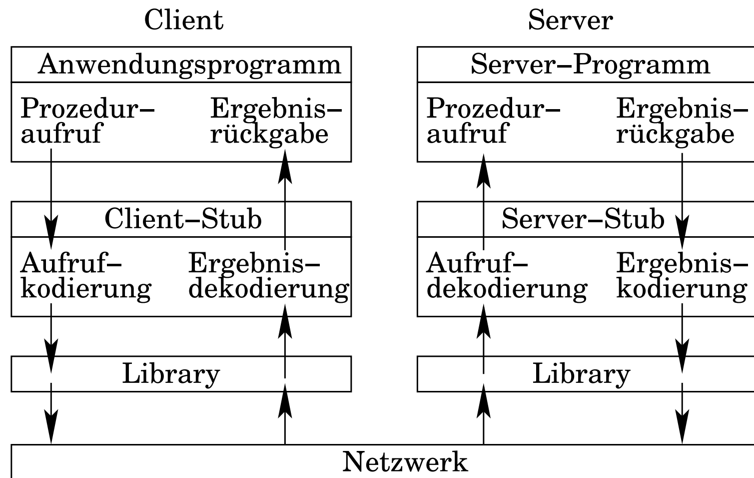
Die im Hintergrund stattfindende Nachrichtenübertragung wird vor dem Anwender versteckt. Der RPC-Compiler erzeugt automatisch alle erforderlichen Funktionen zur Formatierung der Daten.

RPC

Um die im Hintergrund stattfindende Nachrichtenübertragung vor dem Anwender zu verstecken, werden Stubs verwendet.

- Der *Client-Stub*
  - bestimmt die Server-Adresse,
  - stellt die Nachricht zusammen,
  - gleicht unterschiedliche Codierungen und Datenformate an,
  - verschickt die Nachricht,
  - überwacht die korrekte Übertragung, ...

  Ein Client kann einen Server durch Broadcast in einem lokalen Netz suchen oder einen Auskunftsdienst fragen.
- Der *Server-Stub*
  - führt eine Endlosschleife aus und wartet auf Nachrichten,
  - entpackt eine Nachricht und ruft die gewünschte Prozedur auf
  - und verpackt das Ergebnis und schickt es an den Client.



Stub: Stummel, Stumpf

Parameterübertragung

- *call by value:* Für die entfernte Prozedur ist ein Wert-Parameter eine initialisierte lokale Variable, die beliebig modifizierbar ist.

  $\rightarrow$ keine Probleme

- *call by reference:* Die Adresse einer Variablen kann nicht an die entfernte Prozedur übergeben werden, da Aufrufer und Prozedur in verschiedenen Adressräumen laufen.

  Lösung: Verbieten von Pointern und Referenzübergaben oder Nachbilden der call by reference-Semantik, s. nächster Punkt.

- *call by copy/restore:*
  - Kopiere die Variablen auf den Stack des Aufgerufenen $\rightarrow$ wie call by value.
  - Kopiere die Parameter am Ende der Prozedur zurück in die Variablen, überschreibe also die Werte des Aufrufers.

Parameterübertragung

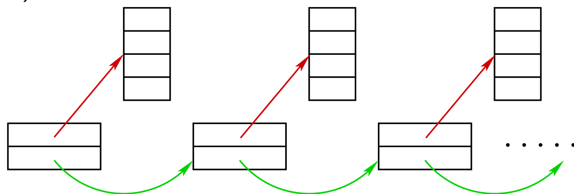
Ein Problem tritt auf, wenn der gleiche Parameter mehrfach in der Parameterliste vorkommt:

```
void proc(int *x, int *y) {
    *x = *x + 1;
    *y = *y + 1;
}

int main(void) {
    int a = 0;

    proc(&a, &a);
    printf("%d\n", a);
}
```

Parameterübertragung

Wie sollen Zeiger auf komplexe Datenstrukturen wie Listen, Mengen oder Hash-Tabellen übertragen werden?

Wir müssen zwei Arten von Adressen unterscheiden: Adressen zur Verwaltung der Struktur (grün) und Adressen auf gespeicherte Inhalte (rot).



Komplette Datenstruktur auf den Server kopieren und

- rote Adressen durch Werte ersetzen oder
- der Server erfragt beim Client den Inhalt jeder roten Adresse.

⇒ sehr ineffizient!

Parameterübertragung

weitere Anmerkungen:

- Ein Zugriff auf globale Variablen ist für entfernte Prozeduren nicht möglich, da Aufrufer und Prozedur in verschiedenen Adressräumen laufen.
- Unterschiedliche Datenrepräsentation:
  - Strings: EBCDIC-, ASCII-, latin1- oder utf16-Code
  - Integer: Einer- oder Zweierkomplement, little-/big-endian.
  - Float: Größe von Mantisse und Exponent.

  $\rightarrow$ automatische Konvertierung notwendig!

automatische Konvertierung

direkte Konvertierung:

- Der Client-Stub hängt vor die Nachricht eine Indikation des verwendeten Formats.
- Der Server-Stub wandelt ggf. vom fremden Datenformat ins eigene Format um.

Problem: Bei n verschiedenen Datenformaten im Netz sind $n \cdot (n - 1)$ Konvertierungsfunktionen notwendig.

Lösung: Verwende maschinenunabhängiges Netzwerkdatenformat: Reduktion auf $2n$ Konvertierungsfunktionen.

---

alte Klausurfrage: Erläutern Sie kurz, wofür ein Netzwerkdatenformat verwendet wird.

- Konvertierung der Daten, um Datenformate verschiedener Plattformen anzugleichen.
- Reduktion auf $2n$ Konvertierungsfunktionen.

automatische Konvertierung

maschinenunabhängiges Netzwerkdatenformat:

- Der Client-Stub wandelt das eigene Datenformat in die Netzwerkdatendarstellung.
- Der Server-Stub wandelt die Netzwerkdatendarstellung ins eigene Format.

Nachteile:

- Zwei unnötige Konvertierungen, falls Client und Server dieselbe Datendarstellung verwenden.
- Direkte Konvertierung erfordert nur eine Umwandlung im Gegensatz zu zwei Umwandlungen bei Verwendung von maschinenunabhängigen Formaten.

Identifikation und Binden der Parameter

Lokalisierung des Servers (binden):

- *statisch:* direkte Angabe der Server-Adresse
- *dynamisch:* mittels Broadcast oder über einen Broker

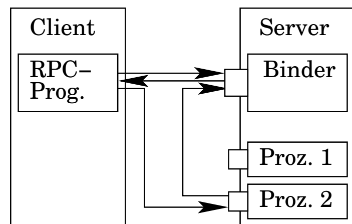
Für die Server muss ein Mechanismus zum Exportieren der angebotenen Prozedurschnittstellen existieren:

- Server schickt Name, Versionsnummer, Identifikation und Adresse zu dem Broker, der bei RPC Binder genannt wird.
- Der Binder trägt den Service und die Adresse in eine Tabelle ein (Registrierung) oder löscht die Einträge (Deregistrierung).
- Der Binder hat eine feste Port-Nummer. Die Datei `/etc/services` enthält folgende Einträge:

```
sunrpc      111/tcp     portmapper
sunrpc      111/udp     portmapper
```

Identifikation und Binden der Parameter

In Unix-Systemen heißt der Binder `portmap` oder `rpcbind`.



1. Server registriert die Prozedur beim Binder
2. Client erfragt Serveradresse beim Binder
3. Binder liefert Serveradresse
4. Client ruft RPC-Prozedur auf

In `/etc/rpc` sind Standard RPC-Dienste aufgelistet:

```
portmapper    100000    portmap sunrpc rpcbind
nfs           100003    nfsprog
ypserv        100004    ypprog
mountd        100005    mount showmount
```

---

alte Klausurfrage: Ist der Binder bei SUN-RPC als handle-driven Broker oder als forwarding Broker implementiert? Begründen Sie Ihre Aussage?

Identifikation und Binden der Parameter

`rpcinfo -p localhost` liefert die Zuordnung zwischen RPC und Port auf dem lokalen Rechner:

```
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
 536870920    1   udp  58233
 536870920    1   tcp  49220
```

Auszug aus `rpcinfo(8)`*:*

rpcinfo makes an RPC call to an RPC server and reports what it finds.

OPTION -p: Probe the portmapper on host, and print a list of all registered RPC programs. If host is not specified, it defaults to the value returned by `hostname(1)`.

RPC: Programmierung

Die Include-Dateien befinden sich unter /usr/include/rpc und /usr/include/rpcsvc.

Ein Entwickler muss nicht alle RPC-Funktionen kennen, da ein Großteil der Arbeit vom RPC-Compiler rpcgen übernommen wird:

- Auf der Server-Seite müssen nur die Funktionen implementiert werden, die aufgerufen werden sollen.
- Funktion, die von der Client-Seite die Kommunikation mit dem Server in Gang setzt:

  ```
  CLIENT *clnt_create(char *srv, PROG, VERS, "tcp")
  ```

  srv ist der Name des entfernten Rechners, der den Service bereit stellt. PROG ist eine Programm-, VERS eine Versionsnummer, die einheitlich von rpcgen festgelegt werden. Neben tcp wird zur Zeit noch udp unterstützt.
- Liste aller low-level Funktionen: man -S3 rpc

RPC: Programmierung

`clnt_pcreateerror(char *s)`
gibt eine Fehlermeldung auf `stderr` aus, falls `clnt_create` fehlgeschlagen ist. Meldung wird an `s` angehängt.

`clnt_perror(CLIENT *clnt, char *s)`
gibt eine Fehlermeldung auf `stderr` aus, falls eine entfernte Prozedur einen Fehler zurückgeliefert hat. `clnt` ist das Handle, mit dem die Prozedur aufgerufen wurde.

`clnt_freeres(CLIENT *clnt, xdrproc_t proc, char *out)`
gibt den Speicher frei, der vom RPC/XDR-System für das Dekodieren des Prozedur-Ergebnisses belegt wurde.

`clnt_destroy(CLIENT *clnt)`
gibt den Speicher wieder frei, der bei `clnt_create` allokiert wurde.

RPC: Programmierung

Die Interface-Routinen benutzen XDR-Routinen, um die Daten von maschinenabhängiger Form in XDR-Standard zu konvertieren und umgekehrt.

Möglich: low-level RPC-Routinen zum Gestalten der Netzwerkfunktionalitäten, XDR-Routinen zum Konvertieren der Daten von und zum gemeinsamen Format direkt aufrufen.

Einfacher: `rpcgen` generiert aus einer Spezifikationsdatei in RPC Language mehrere C-Files und Header-Files.

- C-Files: Quellcode, enthält Aufrufe der low-level RPC- und XDR-Routinen.
- Header-Files enthalten Strukturdefinitionen, die von den entfernten Prozeduren benötigt werden.

RPC: Programmierung

Spezifikationsdatei:

- wird abgelegt in einer Datei mit Endung `.x`
- üblich: Programm-, Versions- und Prozedurnamen werden in Großbuchstaben geschrieben.
- Programmdefinition:

```
program identifier {
    version_list
} = value
```

`identifier`: der Name des Programms
`value`: positive, ganze Zahl von 2000 0000 bis 3fff ffff, kleinere Nummern sind reserviert.

RPC: Programmierung

Spezifikationsdatei: (Fortsetzung)

- Versionsliste:

```
version identifier {
    procedure_list
} = value
```

- Jede Version enthält eine Liste von Prozeduren der Form

```
data_type procedure_name(data_type) = value
```

  `data_type` kann irgendein einfacher oder ein komplexer C-Datentyp sein. Komplexe Datentypen werden in der Spezifikationsdatei definiert.

Beispiel

Der Server soll eine Funktion ggT bereit stellen, die zu zwei ganzen Zahlen den größten gemeinsamen Teiler berechnet.

ggT.x

```
struct args {
    long p;
    long q;
};

program GGT_PROG {
    version GGT_VERS {
        long GGT(args) = 1;
    } = 1;
} = 0x20000008;
```

Beispiel

```
#include <rpc/rpc.h>

struct args {
    long p;
    long q;
};
typedef struct args args;

#define GGT_PROG 0x20000008
#define GGT_VERS 1

extern long *
    ggt_1(args *, CLIENT *);
extern long *
    ggt_1_svc(args *, struct svc_req *);
.....
```

rpcgen ggT.x → ggT.h

Beispiel

rpcgen ggT.x → ggT_clnt.c

```
#include <memory.h>
#include "ggT.h"

// default can be changed using clnt_control()
static struct timeval TIMEOUT = {25, 0};

long *ggt_1(args *argp, CLIENT *clnt) {
    static long res;

    memset((char *) &res, 0, sizeof(res));
    if (clnt_call(clnt, GGT, (xdrproc_t)xdr_args,
            (caddr_t) argp, (xdrproc_t) xdr_long,
            (caddr_t) &res,
            TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&res);
}
```

Beispiel

```
#include "ggT.h"

bool_t xdr_args(XDR *xdrs, args *objp) {
    register int32_t *buf;

    if (!xdr_long(xdrs, &objp->p))
        return FALSE;
    if (!xdr_long(xdrs, &objp->q))
        return FALSE;
    return TRUE;
}
```

rpcgen ggT.x → ggT_xdr.c

konvertieren der Daten in heterogener Umgebung

Beispiel

Außerdem erzeugt `rpcgen ggT.x` den kompletten Server inklusive der `main`-Funktion, der auf den Prozeduraufruf reagiert: `ggT_svc.c`

Nun ist noch die Funktion auf Server-Seite zu schreiben:

- Der Name besteht aus dem in der Spezifikationsdatei angegebenen Namen, jedoch in Kleinbuchstaben. An den Funktionsnamen wird bei Solaris ein `_1`, bei Linux ein `_1_svc` für die Versionssnummer 1 angehängt.
- Rückgabewerte und Argumente sind Zeiger auf die in der Spezifikationsdatei angegebenen Typen.

Beispiel

server.c

```
#include "ggT.h"    // generated by rpcgen
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

// den Funktionen auf der Serverseite wird noch
// ein Requesthandler "*req" mitgegeben
long *ggt_1_svc(args *arg, struct svc_req *req){
    long r, p;

    // return value is address type
    static long q;
    .....
```

Da die Rückgabe von `rpcgen` als Zeiger festgelegt wurde, darf der zurückgegebene Wert nach Ablauf der Funktion nicht automatisch zerstört werden. Daher muss die Variable q als `static` definiert werden.

Beispiel

```
    p = arg->p;
    q = arg->q;
    do {
        r = p % q;
        if (r != 0) {
            p = q;
            q = r;
        }
    } while (r != 0);

    return &q;
}
```

Problem: Statische Variablen in Multi-Threading-Umgebungen!

Beispiel

Wird der RPC-Compiler mit der Option `-M` aufgerufen, wird Code erzeugt, der für Multi-Threading-Umgebungen geeignet ist.

rpcgen -M ggT.x $\rightarrow$ ggT.h

```
#include <rpc/rpc.h>
#include <pthread.h>

struct args {
    long p, q;
};
typedef struct args args;

#define GGT_PROG 0x20000008
#define GGT_VERS 1

extern enum clnt_stat
    ggt_1(args *, long *, CLIENT *);
extern bool_t
    ggt_1_svc(args *, long *, struct svc_req *);
```

Beispiel

Übersetzen und Linken des Servers:

```
gcc -Wall -pedantic -c server.c
gcc -Wall -pedantic -c ggT_svc.c
gcc -Wall -pedantic -c ggT_xdr.c
gcc -o ggTServer server.o ggT_svc.o ggT_xdr.o
```

Schließlich ist noch der Client zu schreiben ...

Beispiel

client.c

```
#include <stdio.h>
#include "ggT.h"

int main(int argc, char *argv[]) {
    CLIENT *cl;
    long *r;
    args arg;

    if (argc != 4) {
        printf("usage: %s server num1 num2\n",
               argv[0]);
        return 1;
    }
```

Beispiel

```c
    cl = clnt_create(argv[1], GGT_PROG,
            GGT_VERS, "tcp");
    if (cl == NULL) {
        clnt_pcreateerror(argv[1]);
        return 2;
    }

    arg.p = atol(argv[2]);
    arg.q = atol(argv[3]);
    r = ggt_1(&arg, cl);  // remote proc. call
    if (r == NULL) {
        clnt_perror(cl, "ggt_1");
        return 3;
    }
    printf("ggT(%ld,%ld) = %ld\n",
           arg.p, arg.q, *r);
    return 0;
}
```

Beispiel

Übersetzen und Linken des Clients:

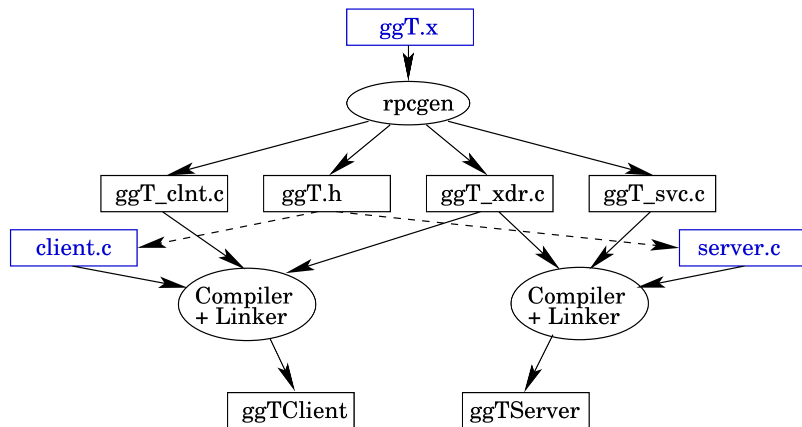
```
gcc -Wall -pedantic -c client.c
gcc -Wall -pedantic -c ggT_clnt.c
gcc -Wall -pedantic -c ggT_xdr.c
gcc -o ggTClient client.o ggT_clnt.o ggT_xdr.o
```

Testen des Programms:

- ggf. starten des Portmappers
- starten des Servers
- starten des Clients

Beispiel

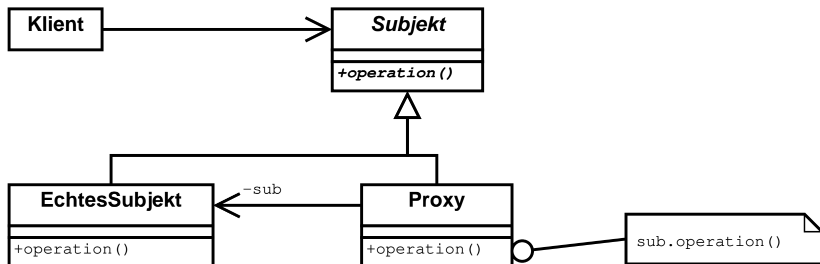
Das Ganze nochmal in der Übersicht:



Entwurfsmuster Proxy

Der Client-Stub implementiert das Entwurfsmuster Proxy.

- Der Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.
- Befindet sich das echte Objekt in einem anderen Adressraum, dann spricht man von einem *Remote-Proxy*.
- Überprüft der Proxy die Zugriffsrechte beim Zugriff auf das echte Objekt, so spricht man von einem *Schutz-Proxy*.



Übersicht

Programmiermodelle und Entwurfsmuster

- Nachrichtenbasiert: sockets
- Auftragsorientiert: RPC
- *Objektorientiert: CORBA*
- Web-Services

Common Object Request Broker Architecture – CORBA:

- Von der Object Management Group (OMG) definiert und mittlerweile als open source freigegeben.
- Im Gegensatz zur Kommunikation über Sockets muss sich der Softwareentwickler nicht um Details wie Datenformate oder Kommunikationsprotokolle kümmern.
- Bietet dem C++-Programmierer gewohnte Sicht auf Objekte, die über Methodenaufrufe miteinander kommunizieren.
- Kümmert sich um das Konvertieren und Verpacken der zu verschickenden Daten (Parameter und Rückgabewert) und um den entfernten Methodenaufruf über das Netzwerk.

→ Ermöglicht Kommunikation zwischen und Koordination von Objekten auf verschiedenen Rechnern.

Jedes entfernte Objekt spezifiziert ein Interface, das festlegt, welche Methoden von Clients aufgerufen werden können.

Zur Zeit wichtige objektorientierte Systeme:

- Auf Java basierendes Remote Methode Invocation (RMI).
  - Ist plattformunabhängig durch Java Virtual Maschine.
  - Es lassen sich nur Java-Objekte ansprechen.
- Microsofts Distributed Component Object Model (DCOM):
  - Ist an Microsoft-Betriebssysteme gebunden.
  - Wurde entwickelt aus dem Component Object Model (COM) und dem Object Linking and Embedding (OLE).
  - Spezifikation der Schnittstellen mittels Object Definition Language (ODL) oder Interface Definition Language (IDL).
  - Wird in .NET durch Windows Communication Foundation (WCF) abgelöst.
  - Es gibt Adapter wie JCOM, um ohne Microsofts DCOM-Implementierung per DCOM-Protokoll zu kommunizieren. Erlaubt Java-Klassen auf COM-Objekte zuzugreifen, um bspw. eine Excel-Tabelle aus Java heraus zu erstellen.

Object Management Group (OMG) und CORBA

- Internationales Konsortium: 1989 von 8 Mitgliedern gegründet, hat mittlerweile über 700 Vertreter aus allen wesentlichen Computerfirmen.
- Implementiert keine Produkte, sondern legt Spezifikationen für Schnittstellen und Protokolle fest.
- Mitglieder reichen Spezifikationen ein, die veröffentlicht und diskutiert werden, und schließlich einer Abstimmung unterliegen.
- Hat ein abstraktes Objektmodell und eine objektorientierte Referenzarchitektur definiert: Object Management Architecture (OMA), besser bekannt unter CORBA.
- Unter Linux existiert eine freie CORBA-Implementierung namens MICO, das als Abkürzung für „Mico Is COrba“ steht.

  Informationen unter `http://www.mico.org`.

Interface-Definition

Erster Schritt:

- Definition aller Interfaces: Welche Methoden können von einem Client auf den Server-Objekten aufgerufen werden?
- Interface-Definitionen erfolgen in der Interface Definition Language IDL: Die Syntax ähnelt einer C++-Klassendefinition.

Wir wollen uns CORBA anhand eines einfachen Beispiels[3] erarbeiten: In einem Adressbuch können Einträge bestehend aus Name und Telefonnummer gespeichert werden.

Das Interface `AddrBook` enthält zwei Methoden:

- `addAddr(name, no)`: Hinzufügen eines neuen Eintrags.
- `getAddr(name)`: Suchen eines Eintrags anhand des Namens.

[3] `http://cplus.kompf.de/corbatut.html`

Interface-Definition in Datei `AddrBook.idl`:

```
exception AddrNotFoundException {
    string msg;
};

interface AddrBook {
    void addAddr(in string name, in string no);
    string getAddr(in string name)
        raises (AddrNotFoundException);
};
```

Bei Parametern die Richtung der Datenübergabe angeben:

- `in:` Daten werden vom Aufrufer zum Objekt transportiert.
- `out:` Datentransport vom Objekt zurück zum Aufrufer.
- `inout:` Bidirektionaler Datentransfer.
- Bei einem Return-Value werden immer Daten zum Aufrufer zurück übertragen, daher ist keine Angabe von out nötig.

Interface-Definition

Folgende Datentypen stehen in der IDL zur Verfügung:

```
void
boolean
char / wchar
float / double / long double
short / long / long long
octet
string
enum Tage {Mo, Di, Mi, Do, Fr, Sa, So};
struct Datum {short t, short m, short j};
```

Interface-Definition

Beachte: Jeder Methodenaufruf ist eine Netzwerkübertragung!

- Wenn nur wenige Bytes an Nutzdaten übertragen werden, entsteht ein beträchtlicher Overhead.

→ Evtl. abweichen von Theorie des objektorientierten Entwurfs!

Arbeite nicht mit entfernten Objekten wie `Entry` mit Methoden wie `getName()`, `setName()`, `getNumber()` usw.

Beim Entwurf von CORBA-Interfaces gilt:

- Minimiere die Anzahl von Interfaces!
- Statt n Methodenaufrufen mit einem Parameter besser ein Methodenaufruf mit n Parametern!
- Um bei langen Parameterlisten den Überblick zu behalten, sollten zusammengehörige Daten besser in Strukturen oder Felder als in Objekte (= Interfaces) zusammengefasst werden!

Interface-Definition

Zur Erinnerung: Refactoring-Katalog

- Problem: Lange Parameterlisten sind schwierig zu benutzen und schwer zu verstehen.
- Lösung: *Ganzes Objekt übergeben*, falls Parameter zu einem Objekt gehören, oder *Parameterobjekt einführen*, falls Daten aus verschiedenen Objekten stammen.

Entwurfsmuster wie „Transfer Object“ für verteilte Anwendungen finden Sie bspw. im Buch Core J2EE Patterns von D. Alur, D. Malks und J. Crupi, Prentice Hall.

Interface-Definition

Interface Definition Language:

- Definiert die Schnittstelle zwischen dem Client-Code und der server-seitigen Objektimplementierung.
- Ist sprachunabhängig und wurde von der OMG bei der ISO als Standard eingereicht.
- Spezifiziert die
  - benutzerdefinierten Datentypen,
  - die öffentlichen Attribute einer Komponente,
  - die Basisklasse, von der sie erbt,
  - die Ausnahmen, die ausgelöst werden und
  - die Methoden inkl. Parameter und Rückgabewert.

Erzeugung Stub-/Skeleton-Klasse

Die Remote-Objekte auf Server-Seite sind aufgeteilt in

- Servants, die die Applikationslogik implementieren – also in unserem Beispiel die Methoden `addAddr()` und `getAddr()`.
- Objekte, die die Verbindung eines Servants zur CORBA Implementierung herstellen, also zum Object Request Broker (ORB), der für die Vermittlung der verteilten Funktionsaufrufe verantwortlich ist.

Zweiter Schritt: Die Implementierung der Servants. Das Übersetzen der Datei `AddrBook.idl` mittels

```
idl --c++-suffix cpp AddrBook.idl
```

liefert die Dateien `AddrBook.h` und `AddrBook.cpp`, die die Stub- und Skeleton-Klassen enthalten. Ohne die `suffix`-Option würde die Datei `AddrBook.cc` erzeugt werden.

Erzeugung Stub-/Skeleton-Klasse

Die Datei `AddrBook.h` enthält unter anderem:

```
// exception class
struct AddrNotFoundException :
      public CORBA::UserException {...}

// Base class for interface AddrBook
class AddrBook : virtual public CORBA::Object {
public:
   ....
   virtual void addAddr(const char* name,
                        const char* no) = 0;
   virtual char* getAddr(const char* name) = 0;
}
```

Erzeugung Stub-/Skeleton-Klasse

```
// Stub for the interface AddrBook
class AddrBook_stub: virtual public AddrBook {
    ...
    void addAddr(const char* name,
                 const char* no);
    char* getAddr(const char* name);
}

// Skeleton for the Server
class POA_AddrBook : virtual public
        PortableServer::StaticImplementation {
   ...
   virtual void addAddr(const char* name,
                        const char* no) = 0;
   virtual char* getAddr(const char* name) = 0;
}
```

Implementieren des Server-Objekts

Anmerkungen:

- Aus dem IDL Datentyp `string` ist der C++-Datentyp `char*` geworden.
- Aus dem IDL Attribut `in` ist eine `const`-Spezifikation des Funktionsparameters geworden.

Die Datei `AddrBook.cpp` beinhaltet die Implementierung des Server-Objektes:

- Die Klasse `POA_AddrBook` dient als *Basisklasse* für den Servant (Skeleton-Klasse).
- Wir müssen nur noch eine neue Klasse schreiben, die von `POA_AddrBook` abgeleitet ist und die die beiden rein virtuellen Funktionen `addAddr()` und `getAddr()` implementiert.

  üblich: Benenne die Klasse mit Nachsatz `_impl`.

Implementieren des Servants

Deklaration des Servants in Datei `AddrBook_impl.h`:

```
#include "AddrBook.h"
#include <map>
#include <string>

using namespace std;

class AddrBook_impl :
        virtual public POA_AddrBook {
private:
    map<string, string> _items;

public:
    virtual void addAddr(const char* name,
                         const char* no);
    virtual char* getAddr(const char* name);
};
```

Implementieren des Servants

Definition der Servants in Datei `AddrBook_impl.cpp`:

```
#include <CORBA.h>
#include "AddrBook_impl.h"

void AddrBook_impl::addAddr(const char* name,
                            const char* no) {
    _items[name] = no;
}
...
```

Beachte: Übergabeparameter vom Typ `char *` belegen Speicherplatz auf dem Heap.

- Dieser Speicherplatz wird vom ORB allokiert und nach Beendigung der Funktion wieder freigegeben.
- Danach dürfen im Server-Objekt keinerlei Referenzen mehr auf diesen Speicher existieren!

Implementieren des Server-Objekts

Hier: Beim Einfügen in die Map vom Typ `map<string, string>` werden `name` und `no` kopiert und alles ist gut.

Fortsetzung:

```
char* AddrBook_impl::getAddr(const char* name) {
    map<string, string>::iterator r;

    r = _items.find(name);
    if (r == _items.end())
        throw AddrNotFoundException(name);

    string no = (*r).second;
    return CORBA::string_dup(no.c_str());
}
```

`getAddr()` liefert einen Zeiger auf `char` an den ORB zurück.

Implementieren des Server-Objekts

Damit keine Speicherlecks entstehen können, wurde im CORBA-Standard festgelegt: Der ORB ist für die Freigabe dieses Speicherbereiches verantwortlich.

Wir müssen also auf dem Heap liegenden Speicher zurückliefern. Dafür steht die Funktion `CORBA::string_dup()` zur Verfügung.

Kompilieren der bisher erstellten Module:

```
mico-c++ -c AddrBook.cpp
mico-c++ -c AddrBook_impl.cpp
```

Implementieren der Server-Applikation

Dritter Schritt: Erstellen des Server-Programms `server.cpp` durch das Implementieren der `main()`-Funktion.

Header-Files inkludieren: Absolut notwendig sind `CORBA.h` sowie die Deklaration des Servers in `AddrBook_impl.h`:

```
#include <CORBA.h>
#include "AddrBook_impl.h"
#include <fstream>

using namespace std;

int main(int argc, char *argv[]) {
    int rc = 0;
```

Implementieren der Server-Applikation

ORB erzeugen und initialisieren: Dies erfolgt, bevor die Kommandozeilenargumente ausgewertet werden.

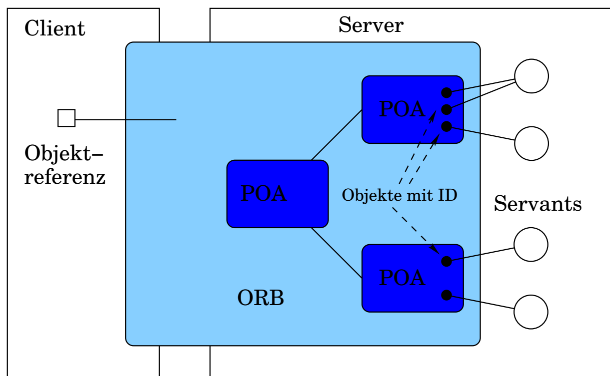
- Zusätzliche Parameter für den ORB können mit auf der Kommandozeile übergeben werden.
- Die Funktion `CORBA::ORB_init(argc, argv)` wertet die ORB-spezifischen Parameter aus und passt `argc` und `argv` entsprechend an.

  Die Infrastruktur für die Kommunikation wird angelegt.

```
try {
    // Get a reference to an ORB object
    CORBA::ORB_var orb =
            CORBA::ORB_init(argc, argv);
```

Alle CORBA-Systemaufrufe werden in einem `try-catch`-Block gekapselt. Anschließend wird der Portable Object Adapter POA erzeugt und initialisiert.

Der *Portable Object Adapter* ist Teil des ORB und Bindeglied zwischen CORBA-Objekten und Servants. Er verwaltet die Ressourcen auf der Server-Seite, um Skalierung zu ermöglichen.



Marshalling und Unmarshalling erfolgt im ORB anhand der IDL.

Implementieren der Server-Applikation

Aus Sicht des Clients sieht es so aus, als sei ein CORBA-Objekt immer zugreifbar/aktiv. Aber auf Server-Seite kann der POA einen Servant, also die Objekt-Implementierung, deaktivieren, um Ressourcen zu sparen.

Begriffe:

- *Client:* Umgebung, die einen Request für ein Objekt auslöst.
- *Server:* Umgebung, in der die Objekte und Servants existieren.

  Client und Server definieren sich immer bzgl. eines Objekts. Ein Programm kann Client bzgl. des Objekts A und Server bzgl. des Objekts B sein.
- *Objekt:* Abstrakte Einheit, an die die Clients ihre Requests senden und die über eine Objektreferenz identifiziert wird.
- *Servant:* Konkretes Objekt, das die Request-Bearbeitung für ein oder mehrere Objekte übernimmt.

Implementieren der Server-Applikation

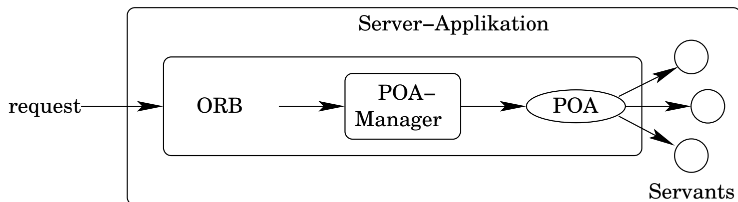
Einem POA und seinen Objekten können Policy-Informationen zugewiesen werden. Eine *Policy* beschreibt, wie die in diesem POA implementierten Objekte zu verwalten sind.

Beispiele:

- `IdUniquenessPolicy:` Kann ein Servant mehrere Object-IDs (sind in der Objektreferenz enthalten) unterstützen?
- `ServantRetentionPolicy:` Bleiben Servants nach der Abarbeitung eines Requests aktiv und in der Active Object Map vermerkt?
- `ThreadPolicy:` Wie werden die Requests den Threads zugeordnet?

Implementieren der Server-Applikation

Abarbeitung eines Requests:



Der POA-Manager ist mit einem Wasserhahn vergleichbar:

- Mit dem Wasserhahn kann der Wasserfluss kontrolliert werden,
- der POA-Manager kontrolliert den Datenfluss in einen POA.

Er erlaubt dem Entwickler, POAs zu deaktivieren, Requests für POAs in Warteschlangen zu stellen oder zu verwerfen.

Jeder POA ist mit einem POA-Manager assoziiert.

Implementieren der Server-Applikation

POA-Manager erzeugen und initialisieren: In diesem einfachen Beispiel verwenden wir den vom System zur Verfügung gestellten Root-POA-Manager.

```
// Get a reference to a POA object
CORBA::Object_var poaobj = orb->
        resolve_initial_references("RootPOA");
```

Weil `resolve_initial_references` ein `CORBA::Object` liefert, müssen wir eine explizite Typkonvertierung (cast) in den korrekten Typ `PortableServer::POA` durchführen.

Für jedes IDL-Interface generiert der IDL-Compiler eine statische Funktion `_narrow()`, die genau das tut. Diese Funktion verhält sich im wesentlichen wie ein C++ dynamic cast:

- Schlägt die Operation fehl, wir ein nil-Pointer geliefert.
- Ist die Operation erfolgreich, wird eine Kopie der Referenz des neuen Typs geliefert.

Implementieren der Server-Applikation

Es ist wichtig, dass beide Variablen, poaobj und poa vom Typ _var sind, damit der Speicherbereich automatisch freigegeben wird, wenn der Scope verlassen wird.

```
// down-cast CORBA::Object_var into
// the type PortableServer::POA_var
PortableServer::POA_var poa =
        PortableServer::POA::_narrow(poaobj);

if (CORBA::is_nil(poa)) {
    cerr << "konnte POA nicht erzeugen!\n";
    return 1;
}

// POA-Manager anlegen
PortableServer::POAManager_var mgr =
        poa->the_POAManager();
```

Implementieren der Server-Applikation

Server-Objekt erzeugen und aktivieren: CORBA stellt mehrere Aktivierungsmethoden zur Verfügung. Hier: implizite Aktivierung

```
// Servant anlegen
AddrBook_impl *servant = new AddrBook_impl();

// Servant aktivieren
AddrBook_var f = servant->_this();
```

`_this` ist eine Methode der Skeleton-Klasse `POA_AddrBook`:

- `servant->_this()` erzeugt ein Objekt und assoziert es mit dem Servant.
- Aufruf von `_this()` innerhalb einer Request-Bearbeitung liefert die Objektreferenz des Objekts, mit dem der Servant assoziiert ist.
- Aufruf von `_this()` außerhalb einer Request-Bearbeitung aktiviert den Servant implizit.

Implementieren der Server-Applikation

Interoperable Object Reference erzeugen und bekannt geben:

- Jedes CORBA-Server-Objekt ist eindeutig identifiziert durch seine Interoperable Object Reference IOR.
- Damit ein Client das Server-Objekt finden kann, muss die IOR dem Client irgendwie bekannt gemacht werden.

Einfachste Möglichkeit: IOR in Zeichenkette umwandeln und in eine Datei speichern, auf die Server und Client beide Zugriff haben:

```
CORBA::String_var s = orb->object_to_string(f);

ofstream out("AddrBook.ref");  // open file
out << s << endl;              // write IOR
out.close();                   // close file
```

Implementieren der Server-Applikation

POA-Manager aktivieren und ORB starten:

```
mgr->activate();
orb->run();
```

Das Programm kehrt nur bei einem expliziten Shutdown des ORBs zurück. In einem solchen Fall räumen wir noch auf: POA-Manager und Server-Objekt freigeben.

```
poa->destroy(TRUE, TRUE);
delete servant;
rc = 0;
```

Implementieren der Server-Applikation

Zum Schluss wird noch die Fehlerbehandlung definiert:

```
    } catch(CORBA::SystemException_catch& ex) {
        ex->_print(cerr);
        cerr << endl;
        rc = 1;
    }
    return rc;
}
```

Übersetzen und linken des Servers:

```
mico-c++ -c server.cpp
mico-ld -o server server.o AddrBook.o \
        AddrBook_impl.o -lmico2.3.13
```

Implementieren des Client

Nachdem der Server nun implementiert ist, kümmern wir uns jetzt um den Client.

Inkludieren der Header-Files: Notwendig sind `CORBA.h` und die vom IDL-Prozessor erzeugte Datei `AddrBook.h`, die den Client-Stub enthält.

```
#include <CORBA.h>
#include "AddrBook.h"
#include <fstream>

#define APPEND 1
#define FIND   2

int main(int argc, char *argv[]) {
    CORBA::ORB_var orb =
            CORBA::ORB_init(argc, argv);
    int rc = 0;
```

Überprüfen der Programm-Parameter:

```
int mode;
if ((argc < 3) || (argc > 4)) {
    cout << "usage: " << argv[0]
         << " -a name no | -f name";
    exit(1);
}
if (strcmp(argv[1], "-a") == 0)
    mode = APPEND;
else if (strcmp(argv[1], "-f") == 0)
    mode = FIND;
else {
    cout << "usage: " << argv[0]
         << " -a name no | -f name";
    exit(1);
}
```

Implementieren des Client

IOR des Servers lesen: Der Client benötigt die Information, welches CORBA-Objekt er verwenden soll und liest dazu die vom Server erzeugte Datei `AddrBook.ref`.

```
char s[1000];
ifstream in("AddrBook.ref");  // open file
in >> s;                      // read IOR
in.close();                   // close file
```

Mittels IOR ein Objekt vom Typ `AddrBook_var` erstellen:

```
try {
    CORBA::Object_var obj =
            orb->string_to_object(s);
    AddrBook_var f = AddrBook::_narrow(obj);
```

Auch hier ist wieder ein down-cast auf den korrekten Typ notwendig.

Implementieren des Client

Anfrage starten, Antwort ausgeben, ggf. Fehlerbehandlung:

```
        if (mode == APPEND)
            f->addAddr(argv[2], argv[3]);
        else {
            string no = f->getAddr(argv[2]);
            cout << "number of " << argv[2]
                 << ": " << no << endl;
        }
    } catch (AddrNotFoundException& ex) {
        cout << "number of " << ex.msg
             << " not found\n";
        rc = 0;
    }
```

Implementieren des Client

Auffangen von CORBA-Ausnahmen und Ausgabe ihrer Ursache:

```
    } catch(CORBA::SystemException_catch& ex) {
        ex->_print(cerr);
        cerr << endl;
        rc = 1;
    }

    return rc;
}
```

Übersetzen und linken des Clients:

```
mico-c++ -c client.cpp
mico-ld -o client client.o AddrBook.o \
        -lmico2.3.13
```

Implementieren des Client

Problem dieser Implementierung: Die Objektreferenz (IOR) des Server-Objekts wird in einer Datei gespeichert.

Sowohl das Client- als auch das Server-Programm benötigt Zugriff auf diese Datei. (Bei einem einzelnen Rechner oder gemeinsamen Netzlaufwerken wie SAMBA- oder NFS-Shares stellt dies kein Problem dar.)

Lösung: Verwende den properitären MICO-Binder oder den CORBA Naming Service.

Der MICO-Binder

Idee:

- Eindeutige Identifizierung des Server-Objekts nicht durch eine IOR sondern durch seine Netzwerkadresse: IP-Adresse des Rechners und Portnummer.
- Stellt das CORBA-Server-Programm mehrere Objekte zur Verfügung: Unterscheide Objekte anhand des Typs des Servers. (Repository-Id der Interface-Definition)
- Existieren mehrere Server gleichen Typs: Programmierer muss jedem Server-Programm einen eindeutigen Namen geben.

Der MICO-Binder

Unser Adressbuch-Server stellt nur ein CORBA-Objekt nach außen bereit $\rightarrow$ der Server kann ohne Veränderung für die Benutzung des MICO-Binders verwendet werden.

Voraussetzung: Stelle sicher, dass der Server beim Starten immer die gleiche Portnummer benutzt. Dazu nutzen wir die folgende Kommandozeilenoption:

```
-ORBIIOPAddr inet:<hostname>:<port>
```

Der MICO-Binder

Einige Parameter von `CORBA::ORB_init(argc, argv)`:

- `ORBIIOPAddr <adress>` legt die Adresse fest, unter der der Internet Inter ORB Protocol IIOP Server läuft.
- `ORBImplRepoAddr <implementation repository address>`
- `ORBIfaceRepoAddr <interface repository address>`
- `OBRNamingAddr <naming service address>`
- `OBGDebugLevel <level>`
- `ORBConfFile <rcfile>`

Änderungen am Client:

- Der Aufruf von `string_to_object(stringified_ior)` muss durch `bind(repository_id, address)` ersetzt werden.
- Adresse als Kommandozeilenargument übergeben.

```
CORBA::Object_var obj =
        orb->bind("IDL:AddrBook:1.0", argv[1]);

if (CORBA::is_nil(obj)) {
    cerr << "no object at " << argv[1]
         << " found.\n";
    return 1;
}

AddrBook_var f = AddrBook::_narrow(obj);
```

Mittels des MICO-Tools `iordump` kann der Inhalt des IOR analysiert werden. `cat AddrBook.ref | iordump` liefert:

```
    Repo Id:  IDL:AddrBook:1.0

IIOP Profile
    Version:  1.0
    Address:  inet:192.168.2.2:48674
   Location:  corbaloc::192.168.2.2:48674//...
  ...
```

Der MICO-Binder

Nachteile:

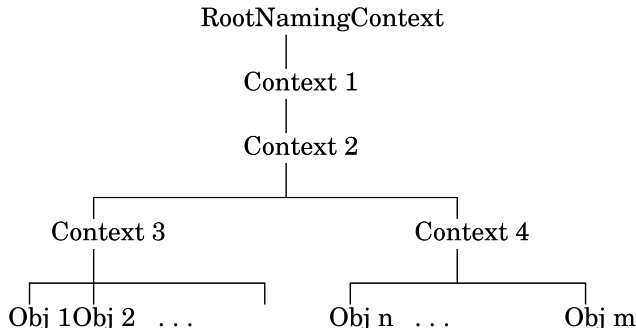
- Properitäre Erweiterung des CORBA-Standards: Falls es in der verteilten Anwendung Objekte gibt, die eine andere CORBA-Implementierung benutzen, funktioniert diese Methode nicht.
- Falls die Applikation auf viele CORBA-Objekte zugreift, wird die Verwaltung der vielen Portnummern kompliziert.

$\Rightarrow$ CORBA Naming Service

CORBA Naming Service

Hierachisches Verzeichnis (ähnlich: LDAP)

- Kontext: eine Art Unterverzeichnis
- Eintrag: enthält IOR eines konkreten CORBA-Objektes



Änderungen am Server:

- Header-Datei `coss/CosNaming.h` inkludieren
- mittels `resolve_initial_references()` die per Kommandozeilenoption `-ORBNamingAddr` übergebene Objektreferenz auf den Naming Service besorgen.

```
CORBA::Object_var nsobj = orb->
    resolve_initial_references("NameService");

if (CORBA::is_nil(nsobj)) {
    cerr << "can't resolve NameService\n";
    return 1;
}

CosNaming::NamingContext_var nc =
      CosNaming::NamingContext::_narrow (nsobj);
```

CORBA Naming Service

- Vom NamingContext (`RootNamingContext`) aus kann die Hierarchie des Naming Service durchlaufen werden.

  hier: Beschränken auf einfachen Namenseintrag `AddressBook`

```
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("AddressBook");
name[0].kind = CORBA::string_dup("");
```

CORBA Naming Service

- anlegen des Namenseintrags mittels bind():

```
try {
    nc->bind(name, f);
} catch (CosNaming::NamingContext
        ::AlreadyBound_catch &ex) {
    nc->rebind(name, f);
}
```

- Fehler bei Adressangabe des Naming Service abfangen

```
catch(CORBA::ORB::InvalidName_catch& ex) {
    ex->_print(cerr);
    cerr << "\ncan't locate Naming Service\n";
    rc = 1;
}
```

Änderungen am Client:

- Header-Datei `coss/CosNaming.h` inkludieren
- Verwenden des Naming Service: Root-Kontext auflösen

```
CORBA::Object_var nsobj = orb->
    resolve_initial_references("NameService");

if (CORBA::is_nil(nsobj)) {
    cerr << "can't resolve NameService\n";
    return 1;
}

CosNaming::NamingContext_var nc =
    CosNaming::NamingContext::_narrow(nsobj);
```

CORBA Naming Service

- `resolve()` aufrufen, um den vom Server angelegten Namenseintrag zu lesen.

```
CosNaming::Name name;
name.length(1);
name[0].id = CORBA::string_dup("AddressBook");
name[0].kind = CORBA::string_dup("");

CORBA::Object_var obj = nc->resolve(name);
```

- Fehlerbehandlung erweitern:
  - `CORBA::ORB::InvalidName_catch`
  - `CosNaming::NamingContext::NotFound_catch`

Übersetzen und linken:

```
mico-c++ -c server.cpp
mico-ld -o server server.o AddrBook.o \
           AddrBook_impl.o \
        -lmico2.3.13 -lmicocoss2.3.13

mico-c++ -c client.cpp
mico-ld -o client client.o AddrBook.o \
        -lmico2.3.13 -lmicocoss2.3.13
```

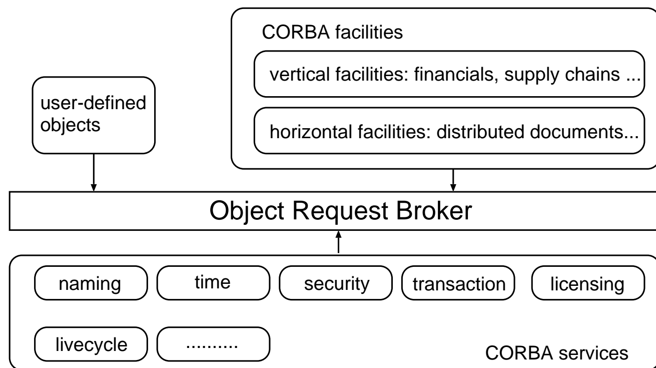
CORBA Naming Service:

- starten: `nsd -ORBIIOPAddr inet:<hostname>:<port>`
- Administration: `nsadmin -ORBNamingAddr inet:...`

Server starten: `server -ORBNamingAddr inet:<host>:<port>`
Client starten: `client -ORBNamingAddr inet:<host>:<port>`

Die *IANA: Internet Assigned Numbers Authority* hat den Port für CORBA auf 2809 festgelegt!

Object Management Architecture



Object Management Architecture

CORBAservices: Fundament der Architektur. Services auf Systemebene bieten Basisoperationen auf Objekten:

- Lifecycle Services: create, delete, copy und move
- Concurrency Control: Lock Manager führt im Auftrag von Transaktionen oder Threads Lock/Unlock-Funktionen für nebenläufige Verarbeitung aus
- Transaction Service: stellt ein Zwei-Phasen-Commit-Protokoll bereit
- Event Service: Dynamische Registrierung für Ereignisse $\rightarrow$ permanentes Objekt `EventChannel` sammelt Ereignisse und verteilt diese an Komponenten, die nichts voneinander wissen
- Naming Service, Time Service, Security Service, Persistence Service, Trader Service, ...

Object Management Architecture

CORBAfacility: Direkt von Applikationsobjekten nutzbare Services. CORBAfacilities können CORBAservices benutzen, von ihnen erben oder sie erweitern.

Sie stellen allgemeine Funktionalität analog zu großen Klassenbibliotheken bereit.

Zwei Gruppen von CORBAfacility:

- *horizontal CORBAfacilities:* anwendungsunabhängige High-Level-Dienste: Benutzeroberfläche, Informations-, System-, Task-Verwaltung usw.
- *domain oder vertical CORBAfacilities:* High-Level-Dienste für spezifische Applikationsdomäne wie Gesundheitswesen, Kunden, Produkte, Zahlungsvorgänge, Bestellungen, Geld, ...

ORB-Interoperabilität

CORBA ist ein offener Standard → Unterstützung unterschiedlicher Produkte erforderlich:

- Verantwortung für die Interoperabilität liegt beim ORB
  - Festlegung der Kommunikation/Koordination durch das Protokoll *GIOP* (General Inter-ORB Protocol)
  - Definition der Transfersyntax *CDR* (Common Data Representation) und 7 Nachrichtentypen
- Beispiel: *IIOP* (Internet Inter-ORB Protocol)
  - Beschreibung der Abbildung GIOP auf TCP/IP
  - Interoperable Objektreferenz enthält die ORB-interne Objektreferenz, Internetadresse, Portnummer → Verwaltung durch ORB, unsichtbar für SW-Entwickler
- analog: DCE Environment Specific Inter-ORB Protocol

Dynamisches CORBA

Bindung zur Laufzeit ohne Stubs: Das Dynamic Invocation Interface DII ermöglicht einem Client, irgendein Objekt zur Laufzeit auszuwählen und dynamisch seine Methoden aufzurufen.

Wie lokalisieren die Clients die entfernten Objekte?

- Der Client bekommt eine Zeichenkette zugeschickt, die in eine Objektreferenz umgewandelt wird. Anschließend wird die Verbindung aufgebaut.
- Der Client durchsucht den Namensdienst von CORBA nach dem Namen des Objekts.
- Der Client fragt die gelben Seiten von CORBA (ein Trader-Dienst) ab und sucht nach einer Funktionalität unter Angabe von Metainformationen.

Dynamisches CORBA

- `get_interface()` liefert eine Referenz auf ein Objekt im Schnittstellenverzeichnis, das diese Schnittstelle beschreibt.
  $\rightarrow$ Einstieg für Navigation im Schnittstellenverzeichnis
- Methodenbeschreibung verschaffen:
  - `lookup_name()` findet die gewünschte Methode.
  - `describe()` liefert IDL-Definition der Methode.
- Argumentliste erzeugen: `create_list()` und `add_item()`
- Anfrage erzeugen:
  `create_request(ObjectRef, Method, Arguments)`
- und vieles mehr, wäre eine eigene Vorlesungsreihe.

Weiteres zu CORBA

- Dokumentation und viele Beispiele werden mit MICO mitgeliefert.
- Zahlreiche gute und ausführliche Tutorials sind im Netz. Zwei Beispiele sind:

  `http://www.codeproject.com/Articles/24863/`
  `A-Simple-C-Client-Server-in-CORBA`
  `http://www.yolinux.com/TUTORIALS/CORBA.html`

CORBA im Überblick

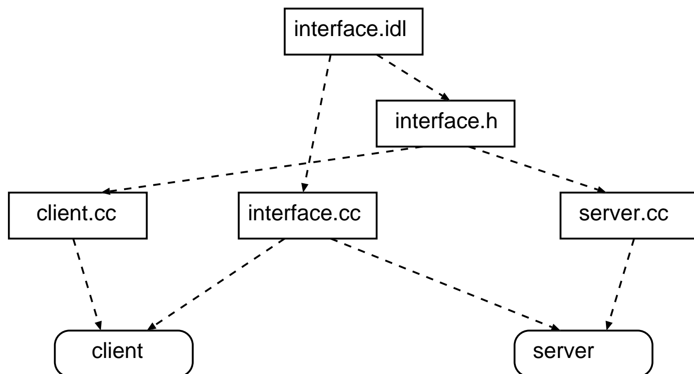
Aufbau ähnlich zu RPC:

- Interface definieren: IDL-Datei für IDL-Prozessor
- Server-Klasse erstellen: ableiten von Skeleton-Klasse
- Server-Programm schreiben: Objekt erzeugen und in Endlosschleife gehen.
- Client-Programm schreiben: entferntes Objekt (Stub-Klasse) „erzeugen“.
- Einfachste Objektidentifikation über Interoperable Object Reference (IOR) in einer Datei.

  Üblich: Keine gemeinsame Datei, sondern Client findet Objekt über CORBA Naming Service oder MICO-Binder.

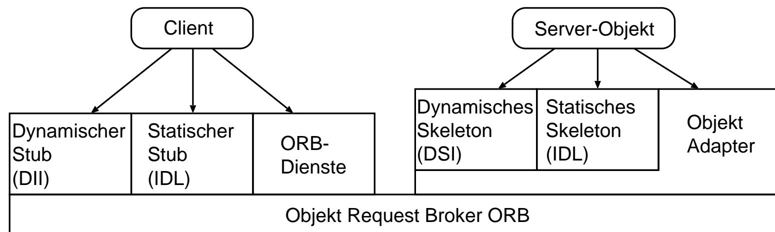
CORBA im Überblick

Implementationsüberblick:



CORBA im Überblick

Interface Architektur:



Object Request Broker (ORB)

- Der Object Request Broker ist ein forwarding Broker und stellt den Kern der Architektur dar.
- Er schafft die Kommunikationsinfrastruktur zum Weiterleiten von Anfragen an andere Architekturkomponenten.
- Kommunikationsservice zwischen Client und Server, allgemeine Services wie Namens- oder Sicherheitsdienst.
- Ortstranzparenz: Der Client muss nicht wissen, welcher Server den Objektdienst bereitstellt.
- Unterstützt Heterogenität bezüglich Betriebssystem, Netzwerkprotokoll, Implementierungssprache usw.:

  Ein Java-Client unter Linux kann einen C++-Server unter Windows aufrufen.

Interfaces auf Client-Seite

- *IDL Stub:* Enthält die vom IDL-Prozessor erzeugten Funktionen für das Client-Programm.
- *Dynamic Invocation Interface (DII):* Wird verwendet, wenn zur Compile-Zeit das Interface des Objektes noch nicht bekannt war und nicht zum Client-Code hinzugebunden werden kann.

  Schnittstelleninformationen über das Objekt werden im *Interface Repository* abgelegt, sodass der ORB die Objektimplementierung lokalisieren und aktivieren kann.
- *ORB-Interface:* Direkter Zugriff auf Funktionen des ORB, z.B. die Funktion zur Umwandlung einer Objektreferenz in einen String.

Interfaces auf Server-Seite

- *IDL Skeleton:* Gegenstück zum IDL Stub, wird aus der Interface-Definition generiert.
- *Dynamic Skeleton Interface (DSI):* Gegenstück zum Dynamic Invocation Interface, bindet zur Laufzeit Anfragen vom ORB an eine Objektimplementierung.

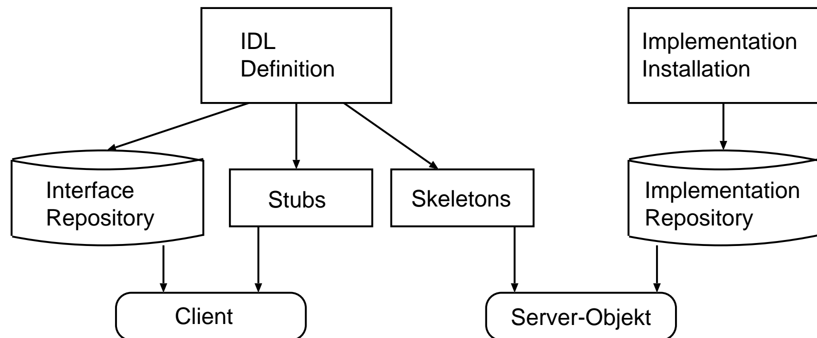
  DSI inspiziert Parameter einer vom ORB eingehenden Anfrage, bestimmt Zielobjekt und Methode mit Hilfe des *Implementation Repositories*, nimmt Antwort entgegen.
- *Object Adapter:* Übernimmt die Kommunikationsanbindung an die Objektimplementierung.
  - Laufzeitumgebung: Instanziieren von Server-Objekten.
  - Weiterleiten der Anforderungen an die Server-Objekte.
  - Abbilden der Objektreferenzen auf die Server-Objekte.

Objekt-Adapter

Portable Object Adapter POA:

- Implementierungsverzeichnis zur Installation und Registrierung von Objektimplementierungen
- Methoden zur Generierung und Interpretation von Objektreferenzen
- aktivieren/deaktivieren von Objektimplementierungen bei Anforderung
- unterstützt verscheidene Aktivierungsverfahren des Servers
- Eine Hierarchie von POAs ist möglich (default "RootPOA").

Interface-Struktur



Übersicht

Programmiermodelle und Entwurfsmuster

- Nachrichtenbasiert: sockets
- Auftragsorientiert: RPC
- *Objektorientiert: Java RMI*
- Web-Services

Server-Interface

In unserem Beispiel wollen wir wieder ein Telefonbuch mit den Methoden `add`, `find` und `erase` implementieren. Wir beschreiben diese Funktionalität in einem Interface.

Server-Interface:

- Alle Methoden, die serverseitig ausgeführt werden sollen, werden in einem Java-Interface definiert, das von `java.rmi.Remote` abgeleitet ist.
- Jede in diesem Interface definierte Methode, kann eine `RemoteException` auslösen, da die Methoden über das Netzwerk ausgeführt werden.

Server-Interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Phonebook extends Remote {

    void add(String name, String no)
            throws RemoteException;

    void erase(String name)
            throws RemoteException;

    String find(String name)
            throws RemoteException;
}
```

Server-Implementierung

Wir speichern die Einträge in einer `HashMap` namens `book`. Diese `HashMap` wird im Konstruktor des Servers initialisiert.

Server-Implementierung:

- Ist von `UnicastRemoteObject` abgeleitet und implementiert die Server-Schnittstelle `Phonebook`.
- Beachte: Der Konstruktor der RMI-Objekt-Implementierung kann eine `RemoteException` auslösen.

  Die Methoden, die die Server-Schnittstelle implementieren, lösen keine `RemoteException` aus. Eine solche Exception wird nur von den Methoden im Server-Stub ausgelöst.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
```

Server-Implementierung

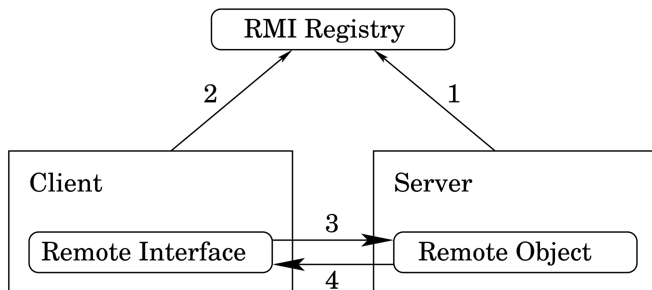
```java
import java.util.HashMap;

public class Server extends UnicastRemoteObject
                    implements Phonebook {

  private HashMap<String, String> book;

  public Server() throws RemoteException {
    book = new HashMap<String, String>();
  }

  public void add(String name, String no) {
    book.put(name, no);
  }

  public String find(String name) {
    return book.get(name);
  }
```

Server-Implementierung

```
    public void erase(String name) {
        book.remove(name);
    }

    public static void main(String[] args) {
        System.setSecurityManager(
                new RMISecurityManager());
        try {
            Server server = new Server();
            Naming.bind(
                    "rmi://127.0.0.1:9090/server",
                    server);
        } catch(Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

Server-Implementierung

Die Klasse `java.rmi.Naming` stellt eine statische Methode `bind` bereit, mit der sich der Server bei der `RMI Registry` anmelden kann.



Der Client kann mittels der statischen Methode `lookup` einen Dienst bei der Registry erfragen und bekommt einen Stub auf das gewünschte Objekt geliefert.

Client

Hier nun ein Client, um die Server-Implementierung zu testen.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class Client {
    public static void main(String[] args) {
        System.setSecurityManager(
                new RMISecurityManager());
        try {
            Phonebook book = (Phonebook)
                Naming.lookup(
                    "rmi://127.0.0.1:9090/server");
```

Auszug aus der Java-API: RMI's class loader will not download any classes from remote locations if no security manager has been set. `RMISecurityManager` does not apply to applets, which run under the protection of their browser's security manager.

```
            if (args[0].equals("find")) {
                System.out.println("reply: "
                        + book.find(args[1]));
            } else if (args[0].equals("add")) {
                book.add(args[1], args[2]);
            } else if (args[0].equals("erase")) {
                book.erase(args[1]);
            } else System.out.println("usage: "
                    + "[find|add|erase] name [no]");
        } catch(Exception ex) {
            System.out.println("Exception: "
                    + ex.getMessage());
        }
    }
}
```

- Zuerst werden die Java-Klassen und -Interfaces kompiliert:

```
javac Phonebook.java
javac Client.java
javac Server.java
```

- Der Server-Stub wird mit dem RMI-Compiler `rmic` generiert. Dazu muss die Server-Implementierung `Server` bereits kompiliert sein.

```
rmic Server
```

- Auf dem Server starten wir die Registry auf dem Port 9090:

```
rmiregistry 9090 &
```

  Ohne Angabe des Ports läuft die Registry auf Port 1099.
- Datei `java.policy.test` mit folgendem Inhalt anlegen:

```
grant {
    permission java.security.AllPermission;
};
```

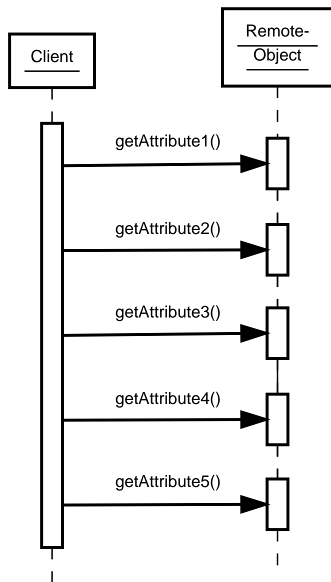
- Server starten:

```
java -Djava.security.policy=./java.policy.test
    Server
```

- Client starten:

```
java -Djava.security.policy=./java.policy.test
    Client add Anton 1234
java -Djava.security.policy=./java.policy.test
    Client add Berta 2345
java -Djava.security.policy=./java.policy.test
    Client find Berta
java -Djava.security.policy=./java.policy.test
    Client find Carla
java -Djava.security.policy=./java.policy.test
    Client erase Berta
```

Entwurfsmuster Transfer-Object

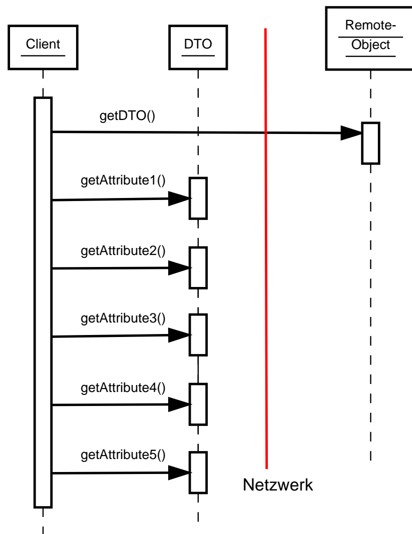


Wollen wir in einer GUI alle Attribute eines Objekts anzeigen und editieren, müssen alle Attribute über das Netzwerk bezogen werden.

Da jeder Methodenaufruf über das Netzwerk um ein Vielfaches teurer ist als ein lokaler Methodenaufruf, fassen wir Daten in einem Objekt zusammen und übertragen das Objekt mit einem einzigen Netzwerkzugriff.

Dadurch wird das Netzwerk entlastet, und die Zeit für das Lesen der Attribute wird deutlich verkürzt.

Entwurfsmuster Transfer-Object



So wird nur einmal über das Netz kommuniziert, das Netz entlastet und die Zeit für das Lesen der Attribute deutlich verkürzt.

Übersicht

Programmiermodelle und Entwurfsmuster

- Nachrichtenbasiert: sockets
- Auftragsorientiert: RPC
- Objektorientiert: CORBA, Java RMI
- *Web-Services*

Dynamische Web-Dokumente

1. CGI (Common Gateway Interface)

- CGI wird verwendet, um Dokumente je nach Bedarf auf dem Web-Server zu erstellen.
- Dokumente werden basierend aus den Daten des Servers als auch aus übergebenen Parametern des Clients erstellt.
- Ein neuer Prozess bearbeitet die Anfrage.
- CGI-Skripte können in vielen Programmiersprachen erstellt werden, sogar als Unix-Skripte.

FastCGI wurde entwickelt, um die Performance-Probleme von CGI zu umgehen:

- Das auszuführende Programm inklusive Interpreter (falls nötig) wird nur einmal geladen und steht dann für mehrere Requests zur Verfügung.
- Die Kommunikation mit dem Web-Server erfolgt über Sockets.

2. Aktive Seiten

Server Site Includes (SSI) sind bestimmte Anweisungen in HTML-Dokumenten, die den Web-Server veranlassen, bestimmte Aktionen auszuführen.

```
<html>
  <head><title>Demo</title></head>
  <body>
    <h2>Server Side Includes</h2>
      aktuelles Datum:
      <!-- #exec cmd="echo `/bin/date`" -->
      <br>
      Und hier die Ausgabe eines Perl-Scripts:
      <br>
      <!-- #exec cgi="/cgi-bin/ssi.pl" -->
  </body>
</html>
```

Dynamische Web-Dokumente

Server-seitige Skripte können auf dem Web-Server eingebunden werden. Beliebt sind:

- *PHP (Personal Home Page* oder *PHP Hypertext Preprocessor)*
- *Perl (Practical Extraction and Report Language)*
- *ASP (Active Server Pages)*
- *JSP (Java Server Pages)*

3. Servlets

Servlets trennen die Anwendungslogik von der Darstellung:

- JSP zur Darstellung (heute JSF)
- Servlets für die Anwendungslogik

Damit Servlets keine HTML-Ausgabe wie hier generieren müssen

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Demo extends HttpServlet {
   public void doGet(HttpServletRequest req,
                     HttpServletResponse res)
         throws ServletException, IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Demo</title>");
      ...
   }
}
```

kann die Ausgabe mittels Java Server Page erfolgen:

```
import javax.servlet.*;
...
public class Demo extends HttpServlet {
   public void doGet(HttpServletRequest req,
                     HttpServletResponse res)
          throws ServletException, IOException {
      // do some computation here
      MyBean bean = new MyBean();
      // add bean to the request
      request.setAttribute("mybean",bean);
      // forward to JSP
      RequestDispatcher rd = req.
          getRequestDispatcher("/welcome.jsp");
      rd.forward(req, res);
   }
}
```

Und hier die JSP zur Darstellung:

```
<% MyBean aBean = (MyBean)
        request.getAttribute("mybean"); %>
<html>
  <head><title>Demo</title></head>
  <body>
    <h2>Java Server Page</h2>
      Zugriff auf die Java-Bean:
      <%= aBean.getMessage() %>
      <%= aBean.getAuthor() %>
  </body>
</html>
```

Servlets können auch als Server Side Include aufgerufen werden und mittels des `Servlet`-Tags in eine HTML-Seite eingebettet werden:

```
<SERVLET CODE=ServletName
         CODEBASE=http://server:port/directory
         initParam1=initValue1
         initParam2=initValue2>
    <PARAM NAME=param1 VALUE=value1>
    <PARAM NAME=param2 VALUE=value2>
    Wenn Sie diesen Text lesen, dann ist
    irgendetwas mit Ihrem Web-Server nicht
    in Ordnung.
</SERVLET>
```

Extensible Markup Language (XML)

- XML ist wie HTML eine Untermenge von *SGML* (Standard Generalized Markup Language)
- 1998 wurde der Standard XML 1.0 vom World Wide Web Konsortium verabschiedet.
- XML ermöglicht inhaltliche Charakterisierung von Texten.
- XML als Datenaustausch-Format für RPC-Mechanismen
  - XML-RPC
  - Simple Object Access Protocol (SOAP)

XML-RPC for Java

Datenaustausch mittels XML:

Java Type	XML Tag Name
Integer	i4
Boolean	boolean
String	string
Double	double
java.util.Date	dateTime.iso8601
byte[ ]	base64
java.util.Map	struct
Object[ ] und java.util.List	array

Näheres zu XML-RPC for Java finden Sie unter
`http://ws.apache.org/xmlrpc/`

Kommunikation mittels HTTP:

Wir lassen den Paket-Sniffer `ngrep -d lo` auf dem Loopback-Device laufen und schneiden die Kommunikation zwischen XmlRpcClient und XmlRpcServer mit.

Anfrage des Clients:

```
POST / HTTP/1.1
Content-Type: text/xml
User-Agent: Apache XML RPC 3.1.3 (Sun HTTP Transport)
Cache-Control: no-cache
Pragma: no-cache
Host: 127.0.0.1:8080
Accept: text/html, image/gif, image/jpeg, ...
Connection: keep-alive
Content-Length: 206
```

XML-RPC for Java

Anfrage des Clients: (Fortsetzung)

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
  <methodName>demo.sumAndDiff</methodName>
  <params>
    <param><value><i4>5</i4></value></param>
    <param><value><i4>3</i4></value></param>
  </params>
</methodCall>
```

Der Name der aufzurufenden Funktion sowie alle benötigten Parameter werden mittels XML verpackt und mittels HTTP zum Server geschickt.

XML-RPC for Java

Antwort des Servers:

```
HTTP/1.1 200 OK
Server: Apache XML-RPC 1.0
Connection: close
Content-Type: text/xml
Content-Length: 258

<?xml version="1.0" encoding="UTF-8"?>
```

Antwort des Servers: (Fortsetzung)

```
<methodResponse>
  <params><param><value>
    <struct>
      <member>
        <name>diff</name>
        <value><i4>2</i4></value>
      </member>
      <member>
        <name>sum</name>
        <value><i4>8</i4></value>
      </member>
    </struct>
  </value></param></params>
</methodResponse>
```

XML-RPC for Java

Client:

```
import java.util.Vector;
import java.util.HashMap;
import java.net.URL;
import org.apache.xmlrpc.client.*;
import org.apache.xmlrpc.XmlRpcException;

public class Client {
   public static void main (String [] args) {
      try {
         XmlRpcClientConfigImpl config =
             new XmlRpcClientConfigImpl();
         config.setServerURL(
             new URL("http://127.0.0.1:8080/"));
         XmlRpcClient clnt = new XmlRpcClient();
         clnt.setConfig(config);
```

```java
// Build our parameter list.
Vector<Integer> params =
    new Vector<Integer>();
params.addElement(new Integer(5));
params.addElement(new Integer(3));

// Call the server, and get our result.
HashMap result = (HashMap) clnt.
    execute("demo.sumAndDiff", params);
int sum = ((Integer) result.
    get("sum")).intValue();
int difference = ((Integer) result.
    get("diff")).intValue();

// Print out our result.
System.out.println("Sum: " + sum +
    ", Difference: " + difference);
```

```
        } catch (XmlRpcException ex) {
          System.err.println("XML-RPC Fault #" +
              ex.code + ": " + ex.toString());
        } catch (Exception ex) {
          System.err.println(ex.toString());
        }
    }
}
```

XML-RPC for Java

Server:

```
import java.util.HashMap;
import org.apache.xmlrpc.server.*;
import org.apache.xmlrpc.webserver.WebServer;

public class Server {
    public Server() {
        // Regular Java object with constructor
        // and member variables. Public methods
        // will be exposed to XML-RPC clients.
    }
    public HashMap sumAndDiff(int x, int y) {
        HashMap<String, Integer> result =
            new HashMap<String, Integer>();
        result.put("sum", new Integer(x + y));
        result.put("diff", new Integer(x - y));
        return result;
    }
```

```java
    public static void main(String [] args) {
        try {
            // invoke as http://localhost:8080/
            WebServer webServer =
                new WebServer(8080);
            XmlRpcServer xmlRpcServer =
                webServer.getXmlRpcServer();
            PropertyHandlerMapping phm =
                new PropertyHandlerMapping();
            phm.addHandler("demo", Server.class);
            xmlRpcServer.setHandlerMapping(phm);

            webServer.start();
        } catch (Exception ex) {
            System.err.println("Server: " + ex);
        }
    }
}
```

Auch für C++ gibt es XML-RPC-Implementierungen.

Hier ein einfacher Client:

```
#include <cstdlib>
#include <string>
#include <iostream>
#include <xmlrpc-c/girerr.hpp>
#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/client_simple.hpp>

using namespace std;

int main(int argc, char **) {
    if (argc > 1) {
        cerr << "program has no arguments!\n";
        exit(1);
    }
```

```
    try {
        xmlrpc_c::clientSimple myClient;
        xmlrpc_c::value result;

        myClient.call(
                "http://localhost:8080/RPC2",
                "sample.add",
                "ii", &result, 5, 7);

        int sum = xmlrpc_c::value_int(result);
        cout << "Result: " << sum << endl;
    } catch (exception const& e) {
        cerr << "Client threw error: "
             << e.what() << endl;
    } catch (...) {
        cerr << "Unexpected client error.\n";
    }
}
```

Und hier der Server:

```
#include <stdexcept>
#include <iostream>
#include <unistd.h>

#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/registry.hpp>
#include <xmlrpc-c/server_abyss.hpp>

using namespace std;
```

```
class sampleAddMethod :
        public xmlrpc_c::method {

public:
    sampleAddMethod() {
        // signature and help strings are
        // documentation -- the client
        // can query this information with
        // a system.methodSignature and
        // system.methodHelp RPC.
        this->_signature = "i:ii";
            // method's result and two arguments
            // are integers
        this->_help = "This method adds two"
                " integers together";
    }
```

```
    void
    execute(xmlrpc_c::paramList const& paramList,
            xmlrpc_c::value *   const  retvalP) {

        int const addend(paramList.getInt(0));
        int const adder(paramList.getInt(1));

        paramList.verifyEnd(2);

        cout << "param1: " << addend << endl;
        cout << "param2: " << adder << endl;

        *retvalP = xmlrpc_c::value_int(addend +
                                       adder);
    }
};
```

```
int main(void) {
    try {
        xmlrpc_c::registry myRegistry;
        xmlrpc_c::methodPtr sampleAddMethodP(
                new sampleAddMethod);
        myRegistry.addMethod("sample.add",
                sampleAddMethodP);

        xmlrpc_c::serverAbyss myAbyssServer(
            xmlrpc_c::serverAbyss::constrOpt()
            .registryP(&myRegistry)
            .portNumber(8080));
        myAbyssServer.run();
    } catch (exception const& e) {
        cerr << "Something failed. " << e.what()
             << endl;
    }
}
```

XML-RPC for C/C++

Übersetzen der Programme:

```
g++ cl.cpp ‘xmlrpc-c-config c++2 client --libs‘
g++ sv.cpp ‘xmlrpc-c-config c++2 abyss-server --libs‘
```

Näheres zu XML-RPC for C++ finden Sie unter
`http://xmlrpc-c.sourceforge.net/`

Web Services

- Web-Services sollen bisherige Möglichkeiten für komplexe Dienste erweitern.
- Es wird allgemein von Web-Diensten oder Web-Services gesprochen, wenn der Client (Browser) auf hinter dem Web-Server liegende Dienste zugreift.
- In diesem Sinne ist CGI ein Web-Service.
- Es wird im engeren Sinne nur von Web-Services gesprochen, wenn die XML-Technologie eingesetzt wird und zur Übertragung SOAP (Simple Object Access Protocol) verwendet wird.

Web Services

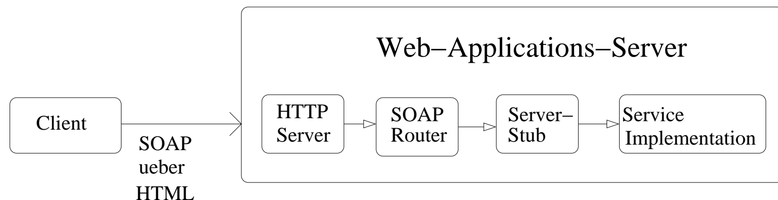
Bestandteile von Web-Services:

- *SOAP* (Simple Object Access Protocol) für die Kommunikation.
- *WSDL* (Web Service Description Language), um die Services zu beschreiben.
- *UDDI* (Universal Description, Discovery and Integration), um die Services zu verwalten.

Überblick

- RPC-Call mit Parametern oder Text-Nachricht in XML schreiben und in SOAP-Nachricht verpacken.
- Nachricht mit HTTP (synchrones Transportprotokoll) oder SMTP (asynchron) übermitteln.
- HTTP-Server leitet Nachricht ans SOAP-System weiter.
- Der XML-Inhalt wird analysiert und bearbeitet.
- Das Ergebnis in SOAP verpacken und zurückschicken.

Aufbau einer Anfrage



Entwicklungsumgebungen

- .NET (Microsoft)
- SUN ONE (Sun)
- WebSphere (IBM)
- gSOAP (`http://gsoap2.sourceforge.net/`)

Simple Object Access Protokoll

SOAP gibt Struktur einer Nachricht vor, bestehend aus Envelope, Header (optional) und Body.

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope xmlns:s="http://www.w3.org/..."> ...
    <s:Header>
    </s:Header>
    <s:Body>
    </s:Body>
</s:Envelope>
```

- SOAP wird mit HTTP-POST an den HTTP-Server geschickt.
- Es entspricht z.B. dem Internet Inter-ORB Protocol (IIOP) bei CORBA.
- Es ist zustandslos.

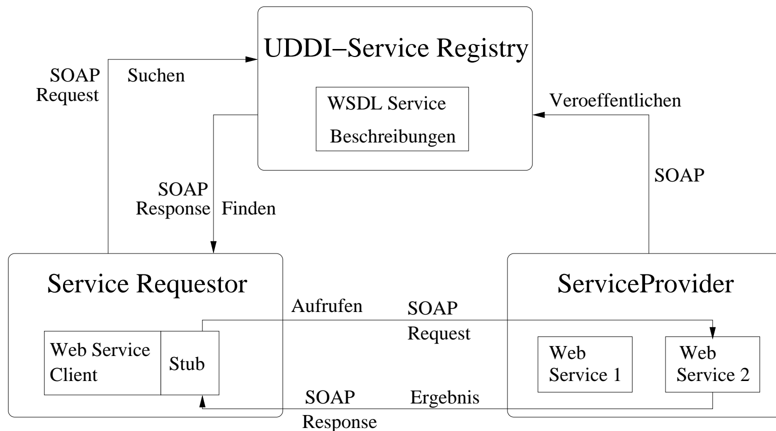
Web Service Description Language

- WSDL ist eine XML-basierte Beschreibungssprache, die die gleiche Funktion hat wie IDL bei RPC oder CORBA.
- Es beinhaltet die Elemente
  - `<types>`: Datentypen
  - `<message>`: Anfrage oder Antwort
  - `<portType>`: Funktion
  - `<binding>`: Transportprotokoll
  - `<service>`: URL

Universal Description Discovery and Integration

- UDDI ist eine Spezifikation für weltweit web-basierte Registrierung von Web Services.
- UDDI-Registrierungsstellen wurden von IBM, Microsoft, SAP und NTT-Com betrieben.
- Ein Eintrag enthält
  - Business Entity (Unternehmen)
  - Business Service (Web-Service)
  - Binding Template (technische Beschreibung)
  - tModel (exakte Signatur)
- Zugriff über Browser oder programmgesteuert über SOAP.

Web Service Architektur



Übersicht

Verteilte Systeme

- Einführung
- Programmiermodelle und Entwurfsmuster
- *Architekturmodelle*
- Verteilte Algorithmen
- Dienste

Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - *Interaktionssemantik*
  - Server-Klassifikation
  - Caching
  - Broker, Trader, Proxy, Balancer, Agent
- Peer-to-Peer Netzwerke
  - unstrukturiert: zentralisiert, pur, hybrid
  - strukturiert: DHT basiert

Client/Server

Gliederung einer Anwendung in zwei logische Teile:

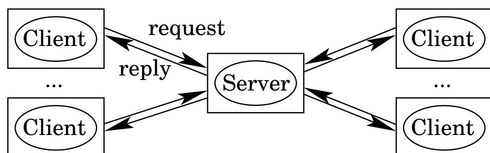
- Clients nehmen Dienste oder Daten eines Servers in Anspruch.
  $\rightarrow$ auslösendes Element
- Server stellen Dienste oder Daten zur Verfügung.
  $\rightarrow$ reagierendes Element
- Zuständigkeiten/Rollen sind fest zugeordnet: Entweder ist ein Prozess ein Client oder ein Server.
- Clients haben keinerlei Kenntnis voneinander und stehen in keinem Bezug zueinander.

$\rightarrow$ Client und Server sind zwei Ausführungspfade mit Interaktion nach dem Erzeuger/Verbraucher-Muster.

Client/Server

Interaktionen zwischen Client und Server verlaufen nach einem fest vorgegebenen Protokoll:

- Client sendet Anforderung (request) an den Server.
- Server erledigt Anforderung oder Anfrage und schickt Rückantwort (reply) an den Client.



Wie koordinieren sich Client und Server?

- Wartet der Client auf die Antwort des Servers?
- Oder ruft der Server am Ende eine callback-Funktion auf?

Koordination der Interaktion

blockierend/synchron: Der Client wartet nach Absenden der Anforderung auf die Antwort des Servers.

- Leicht zu implementieren.
- Ineffizient, da die Arbeit des Clients ruht, während der Server die Anfrage bearbeitet.

nicht blockierend/asynchron: Der Client verschickt die Anforderung, arbeitet sofort weiter und nimmt irgendwann später die Antwort entgegen.

- Die Antwort kann in einer Warteschlange abgelegt werden
- oder der Server ruft am Ende eine callback-Funktion auf
- oder beschränken auf Einweg-Kommunikation, wenn die Antwort nicht benötigt wird.

Interaktionssemantik

Anforderung: Lokale und entfernte Interaktionen sollen gleiche Syntax und Semantik besitzen.

- Reagieren auf Übertragungsfehler und Ausfälle durch Ausnahmebehandlung (exception handling)

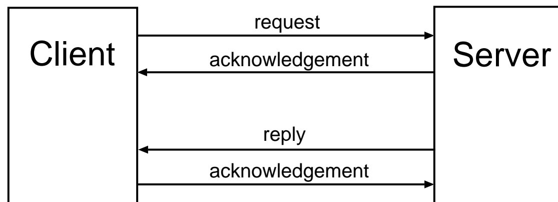
Interaktionsfehler:

- Anforderung (request) geht verloren oder wird verzögert.
- Antwort (reply) geht verloren oder wird verzögert.
- Server oder Client sind zwischenzeitlich abgestürzt.

Unzuverlässige Kommunikation: Nachricht wird ans Netz übergeben, aber es gibt keine Garantie, dass die Nachricht beim Empfänger ankommt. → *may-be-Semantik*

Zuverlässige Kommunikation (1):

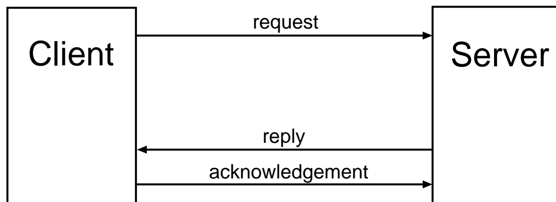
- Jede Nachrichtenübertragung wird durch Senden einer Rückantwort quittiert: Ein Request mit anschließendem Reply benötigt daher vier Nachrichtenübertragungen.



Interaktionssemantik

Zuverlässige Kommunikation (2):

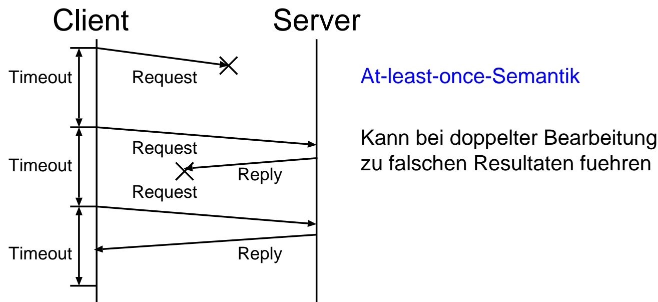
- Request und Reply werden zusammen durch eine Antwort quittiert: Der Client blockiert, bis die Antwort eintrifft; nur die Antwort wird quittiert.



At-least-once-Semantik:

- Der Sendeprozess wartet, bis die Rückantwort innerhalb einer gewissen Zeit eintrifft. Ein Überschreiten der Zeitschranke führt zu einem erneuten Verschicken der Nachricht und Setzen der Zeitschranke.
- Schlagen Versuche mehrmals fehl, ist im Moment kein Senden möglich. Vielleicht ist die Leitung gestört, der Adressat nicht empfangsbereit, ...
- Erhält der Empfänger durch mehrfaches Senden dieselbe Nachricht mehrmals, kann durch erneute Bearbeitung sichergestellt werden, dass die Anforderung mindestens einmal bearbeitet wird.

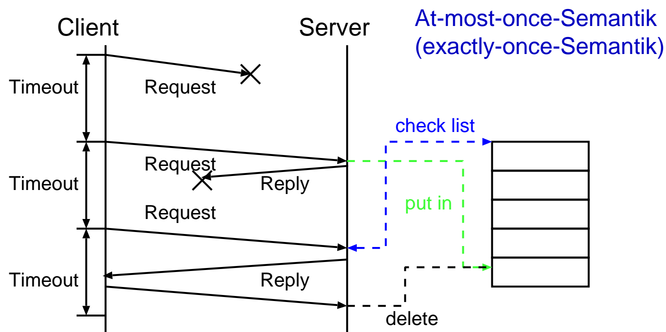
Der Client kann nicht unterscheiden, ob die Anfrage fehlschlug, also der Request verloren ging, oder ob der Server ausgelastet ist und noch arbeitet, oder ob die Antwort verloren ging.



Nachteil: Durch mehrfaches Bearbeiten derselben Anfrage ergeben sich evtl. inkonsistente Daten! Beispiel: Ein File-Server hängt den gesendeten Datensatz mehrfach an die bestehende Datei an.

At-most-once-Semantik:

- Der Empfänger verwaltet eine Anforderungsliste, die die bisher gesendeten Anfragen enthält.
- Anfragen werden mit Hilfe der Liste überprüft:
  - Steht die Anfrage bereits in der Liste, so ging die Rückantwort verloren und die Rückantwort wird nochmal gesendet.
  - Andernfalls wird die Anfrage in der Liste vermerkt, die Anfrage bearbeitet und die Rückantwort gesendet.
- Wurde die Rückantwort bestätigt, wird die zugehörige Anfrage aus der Liste gelöscht.
- Bei vielen Clients und Anfragen muss die Historie nach einiger Zeit gelöscht werden, um den Speicherbedarf zu beschränken.



Exactly-once-Semantik: Um Systemfehler wie einen Plattenausfall auszuschalten, wird die Anforderungsliste in einem stabilen Speicher gehalten. So wird aus der at-most-once-Semantik die exactly-once-Semantik.

Interaktionssemantik

Was passiert bei Ausfall eines Rechners?

- *Server-Ausfall*, nachdem die Anfrage den Server erreicht hat: Der Client kann nicht über den Fehler benachrichtigt werden. Da der Client den Fehler nicht feststellen kann, schickt er die Anfrage erneut.

  Ist exactly-once-Semantik nicht realisierbar? Kann entfernte Interaktion nicht die Semantik von lokaler Interaktion erreichen?

- *Client-Ausfall*, nachdem Anfrage gesendet wurde: Für den Server ist kein Partner mehr vorhanden, die Ergebnisse werden nicht abgenommen.

  Wie oft soll der Server die Antwort verschicken? Was passiert, wenn der Client erneut gestartet wird? Kann der „Neue“ die Antworten entgegen nehmen?

Parallele Server

Ein Server muss mehrere Clients bedienen können:

- Ein *iterativer Server* bearbeitet zunächst die Anfrage, sendet die Rückantwort und wartet erst dann auf die nächste Anfrage.
- Ein *paralleler Server* startet nach dem Eintreffen einer Anfrage einen neuen Prozess/Thread zur Bearbeitung der Anfrage. Der Thread sendet die Rückantwort, der Hauptprozess steht sofort zur Annahme weiterer Anfragen zur Verfügung.

→ Ein paralleler Server hat i.Allg. kürzere Antwortzeiten als ein iterativer Server.

Die Prozesserzeugung und -umschaltung muss mit minimalem Aufwand erfolgen, daher wird in der Regel mit Threads und nicht mit Prozessen gearbeitet.

Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - Interaktionssemantik
  - *Server-Klassifikation*
  - Caching
  - Broker, Trader, Proxy, Balancer, Agent
- Peer-to-Peer Netzwerke
  - unstrukturiert: zentralisiert, pur, hybrid
  - strukturiert: DHT basiert

Server-Klassifikation

Unterscheidung nach Anzahl der Dienste pro Server:

- Ein *shared Server* stellt mehrere Dienste bereit, für jeden Service/Dienst steht ein eigener Prozess zur Verfügung.

  Bei einer Anfrage wird zunächst getestet, ob der Prozess aktiv ist und ggf. wird ein neuer Prozess gestartet.

  Ein solcher Prozess kann zur Erledigung der Anforderung einen Thread starten.
- Ein *unshared Server* stellt nur einen Dienst bereit, verschiedene Dienste liegen auf unterschiedlichen Servern, nicht notwendigerweise auf verschiedenen Rechnern.

In beiden Fällen bleibt der Server nach der ersten Anfrage aktiv und kann weitere Anfragen entgegennehmen.

Server-Klassifikation

Ein Server muss aktiv sein, um Client-Anfragen beantworten zu können. Server können auch anhand der Aktivierung unterschieden werden:

- Ein *per request Server* startet jedesmal neu, wenn eine Anfrage eintrifft. Mehrere Prozesse für den gleichen Dienst können konkurrent aktiv sein.
- Ein *persistent Server* wird beim Starten (Boot-Vorgang) des Systems oder beim Start der Server-Applikation aktiv.

  Client-Anfragen werden direkt an den Server-Prozess weitergeleitet oder bei nicht gestartetem Server vom BS mit einer Fehlermeldung beantwortet.

Server-Klassifikation

Client-Anfragen können Änderungen an den vom Server verwalteten Daten und Objekten bewirken. Dies kann sich wiederum auf nachfolgende Client-Anfragen auswirken:

- *Zustandsinvariante* Server liefern Informationen, die sich zwar ändern können, die aber unabhängig von Client-Anfragen sind.

  Beispiele: Web-, Name- und Time-Server.

  Die Reihenfolge der Anfragen spielt keine Rolle.
- *Zustandsändernde* Server wechseln zwischen 2 Client-Anfragen ihren Zustand, wodurch evtl. Anforderungen von Clients nicht mehr erfüllt werden können (z.B. File-Server: löschen einer Datei).

  Die Reihenfolge der Anfragen ist zur Erledigung der Aufgaben entscheidend.

Server-Klassifikation

Unterscheide zustandsändernde Server danach, ob neuer Zustand gespeichert wird oder nicht.

Zustandsspeichernde (stateful) Server speichern Zustände in internen Zustandstabellen (Gedächtnis).

- Vorteile:
  - Clients müssen den Zustand nicht in jeder Anfrage mitteilen, die Netzlast wird also reduziert.
  - Der Server kann auf zukünftige Anfragen schließen und entsprechende Operationen im Voraus durchführen.
  - Informationen sind gegen Ausspähen schützbar.
- Nachteil: Hoher Speicherbedarf beim Server.

Server-Klassifikation

Zustandslose (stateless) Server müssen vom Client bei jeder Anfrage die komplette Zustandsinformation erhalten, um die Anforderung korrekt erfüllen zu können.

$\rightarrow$ Stateless Server = Stateful Client

- Vorteil: Nach einem Absturz des Servers muss der alte Zustand nicht wiederhergestellt werden, siehe at-most-once-Semantik. Der Client schickt seine Anfrage einfach noch einmal.
- Nachteil: Es müssen evtl. sensitive Daten übertragen werden.

Server-Klassifikation

Beispiel: zustandsspeichernder File-Server

- Öffnet ein Client eine Datei, liefert der Server einen Verbindungsidentifier zurück, der eindeutig ist für die Datei und den zugehörigen Client: Wer hat die Datei geöffnet, gegenwärtige Dateiposition, letzter geschriebener/gelesener Satz, ...
- Nachfolgende Zugriffe nutzen den Identifier zum Zugriff auf die Datei. Dies reduziert die Nachrichtenlänge, da der Zustand der Datei nicht übertragen werden muss.
- Der Zustand enthält Informationen darüber, ob die Datei für sequentiellen, direkten oder index-sequentiellen Zugriff geöffnet wurde. $\rightarrow$ Durch vorausschauendes Lesen wird evtl. der nächste Block bereitgestellt.

Network File System (NFS) von SUN ist zustandslos!

Server-Klassifikation

Beispiel: zustandsloser Web-Server

- HTTP ist ein zustandsloses Protokoll. Es gibt keinen eingebauten Mechanismus, um einen Zusammenhang zwischen verschiedenen Anfragen auf der Server-Seite herzustellen: Bei zwei aufeinanderfolgenden Anfragen kann der Server nicht entscheiden, ob die Anfragen vom gleichen Benutzer oder von verschiedenen Benutzern stammen.
- Stellt ein Problem beim E-Commerce dar:
  - Gewünscht sind Transaktionen über mehrere Klicks hinweg, ist z.B. für die Realisierung von „Warenkörben“ erforderlich.
  - Zur Verhaltensanalyse von Kunden wird von den Marketing-Abteilungen das Wiedererkennen von Kunden beim nächsten Klick oder auch Tage später gewünscht.
- Identifizierung anhand IP-Adresse ist nicht möglich: Proxy, Firewall mit NAT, Multi-User-Systeme.

Server-Klassifikation

Möglichkeiten zur Lösung:

- Cookies erlauben es, Informationen im Web-Browser zu speichern. Der Web-Browser schickt diese Informationen bei jedem Aufruf einer Seite mit an den Web-Server.
  Problem: Cookies können vom Benutzer deaktiviert werden.
- Hidden Fields sind unsichtbare Felder in HTML-Formularen. Informationen werden also nicht im Web-Browser, sondern direkt in der HTML-Seite gespeichert.
  Problem: Hidden Field Tampering: Verändern der in hidden fields gespeicherten Informationen. Werte müssen daher auf Server-Seite zwingend geprüft werden.
- Bei URL-rewriting werden alle URLs, die ein Benutzer anklicken kann, modifiziert, indem zusätzliche Informationen angehängt werden.
  Beispiel: `http://my.server.de/site.html?id=12345`.

Server-Klassifikation

Eine andere Möglichkeit zur Identifizierung von Anfragen ist die Authentifizierung durch den Web-Server.

- Notwendig, wenn ein Benutzer auf vertrauliche Informationen zugreifen möchte. Authentifizierter Benutzername ist in der Umgebungsvariablen `REMOTE_USER` gespeichert.
- Der Web-Browser speichert die Zugangsdaten und schickt diese bei jedem Zugriff auf die geschützte URL mit der Anfrage mit.
- Um die Risiken bezüglich zwischengespeicherter Zugangsdaten zu minimieren, sollte immer folgendes beachtet werden:
  - Passwort nicht permanent im Browser speichern.
  - Web-Browser nach Abmeldung schließen, um temporären Speicher zu löschen.
  - Web-Server Authentifizierung immer mittels HTTPS absichern, damit Zugangsdaten nicht im Klartext übermittelt werden.

Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - Interaktionssemantik
  - Server-Klassifikation
  - *Caching*
  - Broker, Trader, Proxy, Balancer, Agent
- Peer-to-Peer Netzwerke
  - unstrukturiert: zentralisiert, pur, hybrid
  - strukturiert: DHT basiert

Client-Caching

Wir wollen Zugriffe auf Daten/Inhalte, die bereits einmal vorlagen, beim nächsten Zugriff schnell zur Verfügung zu stellen. Dazu speichern wir eine lokale Kopie beim Client oder bei einem Proxy-Server.

- Beispiel: Bei File- und Web-Servern werden komplette Dateien im Cache abgelegt (Dokumente, Bilder, Audio-Dateien, ...)
- Caching im Hauptspeicher oder auf Platte? Frage des Platzes gegenüber der Performance. Stehen überhaupt Platten zur Verfügung (diskless clients)?
- Größenbeschränkter Cache: Lagere die am längsten nicht benutzen Daten aus (least recently used).

  Dirty-Bit zeigt an, ob Daten überschrieben werden können, oder modifizierte Daten zunächst abzuspeichern sind.

Client-Caching

Zustandsinvariante Server: Enthält der Cache gültige Daten?

- Die vom Client angestoßenen Operationen ändern die Daten nicht, aber die Server-Daten können sich unabhängig vom Client ändern, z.B. durch Ersetzen einer Web-Seite oder bei einem Zeit-Server.
- Lösung:
  - Verfallsdatum für Cache-Eintrag einführen, oder
  - vor dem Verwenden der Daten (einmal pro Sitzung, bei jedem Zugriff auf die Seite, ...) deren Gültigkeit vom Server bestätigen lassen: Datum der letzten Modifikation, Versionsnummer der Datei, Checksumme, ...

    Auf diese Weise sind viel weniger Daten zu emfangen als wenn die gesamte Web-Seite inklusive Bild- und Audio-Dateien übertragen werden müsste.

Client-Caching

Zustandsändernde Server: Enthält der Cache gültige Daten?

- Betrachte einen File-Server und 2 Clients, die je eine Kopie derselben Datei besitzen und modifiziert haben.
  - Ein dritter Client liest die Originaldatei, erhält aber nicht die Änderungen von Client 1 und 2.
  - Was passiert, wenn die Datei von einem der Clients zurückgeschrieben wird? Das Ergebnis hängt davon ab, welcher Client die Datei zuletzt schließt.

⇒ Algorithmen wie *write through*, *delayed write* oder *write on close* lösen nur (bedingt) das erste Problem.

Client-Caching

write through: Ändert ein Client einen Cache-Eintrag, so wird der neue Wert auch an den Server geschickt, damit der Server die Änderung an der Originaldatei nachvollziehen kann.

Nachteile:

- Erhöhte Netzlast, kein Vorteil durch Caching.
- Client sieht Änderungen anderer Clients nicht, wenn ein alter Cache-Eintrag existiert: Vor einem Zugriff Gültigkeit prüfen!

  Beispiel:

  - Prozess P_1 auf Rechner A liest f ohne Modifikation.
  - Prozess P_2 auf Rechner B ändert f und schreibt es zurück.
  - Prozess P_3 auf Rechner A liest alten Cache-Eintrag.

Client-Caching

delayed write: Zur Reduktion des Netzverkehrs werden mehrere Änderungen gesammelt und erst nach Ablauf eines Zeitintervals (bspw. 30s bei SUN NFS) an den Server geschickt.

write on close: Weitere Reduktion, wenn Änderungen erst beim Schließen der Datei an Server geschickt werden.

Probleme:

- Nicht fehlertolerant bei Client-Crash: Alle noch nicht geschriebenen Daten gehen verloren.
- Bei *delayed write* und *write on close* bekommt ein dritter Client die Änderungen von Client 1 und 2 ggf. nicht mit.
- Die Konsistenz zwischen Client- und Server-Daten ist nicht sichergestellt. Beispiel: Client 1 und 2 löschen aus derselben Datei jeden zweiten Datensatz. Ergebnis?

Lösung: zentrale Kontrollinstanz

Client-Caching

Zwei Möglichkeiten der zentralen Kontrollinstanz:

- Der Server tabelliert, welcher Client welche Datei zum Lesen oder Schreiben geöffnet hat:
  - Eine zum Lesen geöffnete Datei kann anderen Clients zum Lesen bereitgestellt werden.
  - Der Zugriff auf eine zum Schreiben geöffnete Datei wird anderen Clients untersagt. Die Anfrage wird entweder verweigert oder in eine Warteschlange gestellt.
- Sobald ein Client eine Datei zum Schreiben öffnet, schickt der Server allen lesenden Clients eine Nachricht zum Löschen der Cache-Einträge: Schreib-/Lesezugriffe gehen ab dann direkt zum Server, der Cache wird quasi ausgeschaltet.
  - $\Rightarrow$ Leser und Schreiber können parallel weiterarbeiten, aber die Netzlast steigt aufgrund des ausgeschalteten Caches.

Nachteile bei zentraler Kontrollinstanz?

Verteilte Prozesse

Nachteile bei einfachen Client/Server-Systemen:

- Der Ausfall des zentralen Servers führt zum Ausfall aller Dienste. → *Single Point of Failure:* Teil eines Systems, dessen Ausfall den Ausfall des gesamten Systems nach sich zieht.
- Keine Skalierung: Der Server ist in seiner Leistung begrenzt.

Ausweg: Wir müssen den Server für die Anwendungen transparent replizieren.

- Die Server müssen untereinander die Konsistenz der Daten sicherstellen
- und sich untereinander koordinieren: Ein Server wird zum Client eines anderen Servers.

→ *Verteilte Prozesse:* Prozesse, die Client und Server sein können.

Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - Interaktionssemantik
  - Server-Klassifikation
  - Caching
  - *Broker, Trader, Proxy, Balancer, Agent*
- Peer-to-Peer Netzwerke
  - unstrukturiert: zentralisiert, pur, hybrid
  - strukturiert: DHT basiert

Client/Server-Server

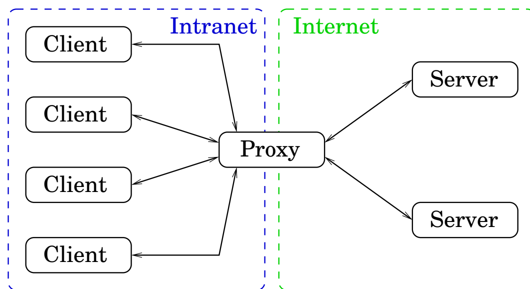
Zwischen den Clients und mehreren Servern kann ein weiterer Server plaziert sein. Man unterscheidet dabei folgende Aufgaben:

- *Proxy:* Stellvertreter für mehrere Server. Oft wird Clients der Zugriff auf externe Server untersagt, nur über den Proxy kann mit der Außenwelt kommuniziert werden.
- *Broker:* Vermittelt zwischen Client und Server: Welcher Server stellt den gesuchten Dienst zur Verfügung?
- *Trader:* Sucht den am besten geeigneten Server heraus: Welcher Zeit-Server hat die geforderte Genauigkeit?
- *Balancer:* Spezieller Trader, der die Arbeitslast auf verschiedene Server verteilt.
- *Agent:* Koordiniert mehrere Server-Anfragen für einen Client: Durch eine Suche über verschiedene Web-Seiten findet der Agent z.B. den günstigsten Anbieter eines gesuchten Produkts.

Proxy

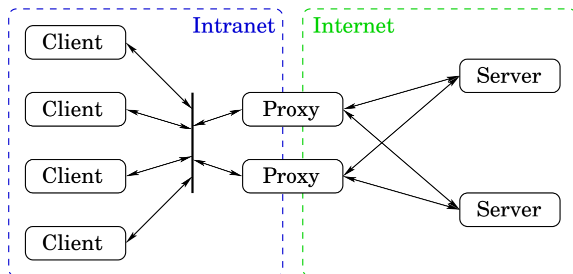
Einsatz als Cache für zustandsinvariante Server:

- Ordne nicht jedem Client einen eigenen Cache zu. Stattdessen wird für mehrere Clients ein gemeinsamer Cache genutzt, den des zwischen Client und Server liegenden Proxy-Servers.
- Der dazwischenliegende Server ist ein Server für die Clients und ein Client für die Server.
- Beispiel: HTTP-Proxy `squid`



Proxy

Problem: Fällt der Proxy aus, ist kein Zugriff auf externe Server möglich. Wir müssen also den Proxy und den Internet-Zugang möglichst redundant auslegen!



Können im Web-Browser mehrere Proxies eingetragen werden?

Proxy-Verbund

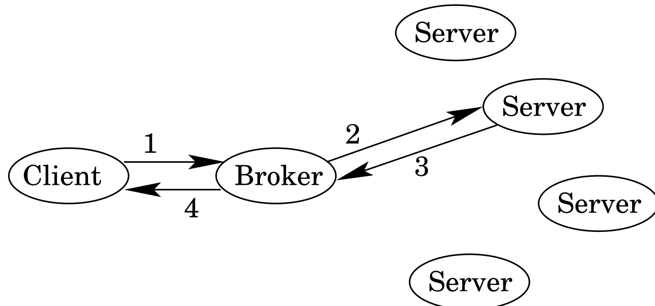
- Bei Squid können übergeordnete Server (Eltern, parents) und gleichberechtigte (Geschwister, siblings) eingetragen werden.
- Ausfälle und Wiederverfügbarkeit von Eltern und Geschwistern werden automatisch mittels Internet Cache Protocol erkannt.
- Beispiel 1: Ausfallsicherheit
  - Proxies im gleichen lokalen Subnetz sind gegenseitig als Geschwister eingetragen.
  - Bevor sie eine Anfrage ans Internet stellen, fragen sie sich gegenseitig, ob sie die angeforderte Seite schon haben.
- Beispiel 2: Hierarchisch
  - Subnetze haben jeweils einen eigenen Proxy, die Zugriffsregeln sind auf einem zentralen Proxy (parent) definiert.
  - Anfragen vom lokalen Proxy ans Internet sind nur bei Ausfall des zentralen Proxys erlaubt: `prefer_direct=off`
- Kombiniere 1 und 2: Ausfallsicherheit und Skalierung
  - Den zentralen Proxy durch mehrere Geschwister ersetzen.
  - Den lokalen Proxy durch mehrere Geschwister ersetzen.

Broker

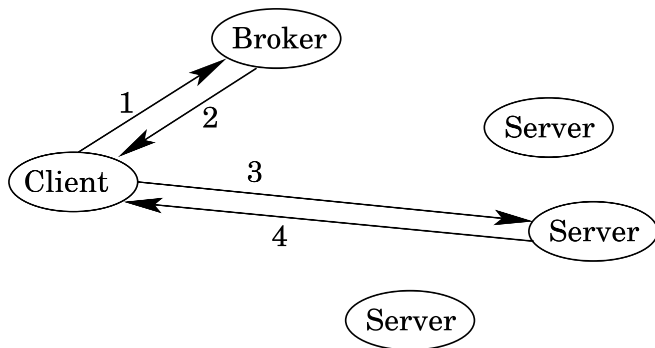
- Ein Broker besitzt Informationen über die Dienste/Services von verschiedenen Servern.
- Alle Server des verteilten Systems registrieren ihre angebotenen Dienste und Eigenschaften beim Broker.
- Durch den Einsatz von Brokern kann Ortstranzparenz hergestellt werden: Die Clients wissen nichts über die Server.
- Ein Broker wird oft zur Adressverwaltung erweitert.
- Wir unterscheiden 3 Typen: Intermediate oder forwarding broker, separate oder handle-driven broker, hybrider broker.

Broker

Intermediate oder *forwarding broker:* Die Client-Anfrage wird vom Broker angenommen und an den betroffenen Server übergeben. Der Broker nimmt die Rückantwort des Servers an und leitet die Antwort an den Client weiter.



Separate oder *handle-driven broker:* Der Broker gibt die ermittelten Server-Informationen an den Client zurück. Anschließend kommuniziert der Client direkt mit dem entsprechenden Server.



Ein *hybrider Broker* unterstützt beide genannten Versionen. Der Client gibt an, welches Modell er wünscht.

Broker

Problem: Der Broker ist ein Single Point of Failure. Wir müssen also Broker mehrfach/redundant auslegen!

Im Extremfall ist jedem Client ein Broker zugeordnet.

- Vorteil: Ausfall eines Brokers betrifft nur einen Client.
- Nachteil: Aufwendige Konsistenzhaltung der Datenbasis.

Trader, Balancer und Agenten

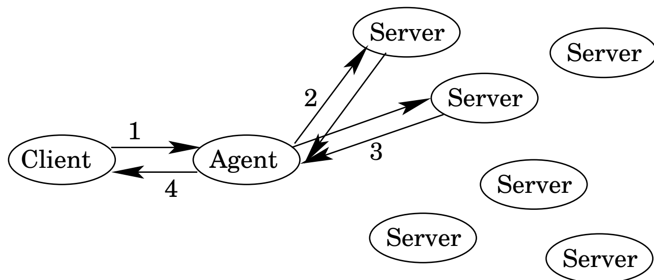
- Falls mehrere Server für einen Dienst zur Verfügung stehen, die Dienste jedoch unterschiedliche Qualität haben, verteilt ein *Trader* die Client-Anfragen anhand der geforderten Qualitätsanforderung.

  Unterscheide drei Typen wie beim Broker: forwarding, handle-driven und hybrid.

- Ein *Balancer* ist ein spezieller Trader, dessen einziges Entscheidungskriterium die Last der Server ist. Dies kann bspw. die Speicherauslastung, die CPU-Nutzung, die Anzahl offener Verbindungen oder eine Kombination daraus sein.
- Clients, die mehrere Dienste abwickeln wollen oder komplexere Dienste in Anspruch nehmen, benötigen einen Ablaufplaner, einen *Agenten*: Anfrage in Teilanfragen zerlegen, Antworten sammeln und eine einzige Rückantwort an den Client schicken.

Ziele: Einfachheit und Performance.

- Mehrere Services hinter einem Service verstecken.
- Teilanfragen parallel durch mehrere Server bearbeiten lassen. Divide-and-Conquer oder Master/Slave-Verfahren.



Oft kennt der Agent die Server nicht, deren Dienste gebraucht werden: Schalte Broker zwischen Agent und Server.

Agenten

Ein Agent kann Teilanfragen iterativ oder parallel ausführen:

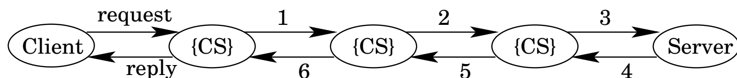
- iterativ: Die angeforderten Dienste bauen aufeinander auf und müssen in einer bestimmten Reihenfolge bearbeitet werden. Ein neu anzufordernder Dienst kann von dem Ergebnis eines vorhergehenden Dienstes abhängen.
- Parallel: Falls Dienste unabhängig voneinander sind oder alle Repliken eines replizierten Servers benachrichtigt werden müssen.

Nachteil: Auch ein Agent ist ein Single Point of Failure.

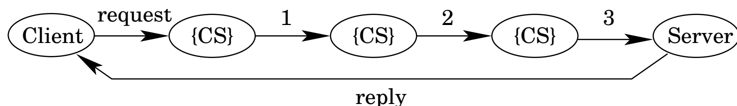
Um dies zu vermeiden, verteilen wir die Agenten-Funktionalität auf die Server: Jeder Server bestimmt seinen Nachfolge-Server.

Client/Server-Ketten

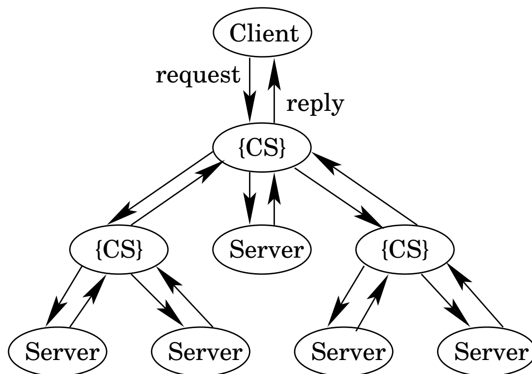
rekursiv: Anforderungen werden an den nächsten verteilten Prozess {CS} weitergereicht, die Rückantworten werden an den verteilten Prozess der vorhergehenden Rekursionsstufe zurückgereicht.



transitiv: Anforderungen werden an den nächsten Prozess weitergereicht, die Rückantwort des rekursionsbeendenden Prozesses wird an den Client zurückgeschickt.



Ist die Anzahl der Nachfolgeprozesse größer als eins, so ergeben sich Baumstrukturen:



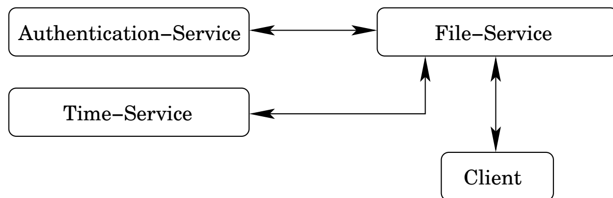
Die einzelnen Anforderungsketten innerhalb des Baumes können wieder rekursiv oder transitiv sein.

Client/Server-Ketten und -Bäume

Bei mehreren Nachfolgeknoten enthält jeder verteilte Prozess {CS} einen internen Agenten, der die nachfolgenden (iterativen oder parallelen) Server-Anfragen koordiniert.

- rekursive Kette: parallele Prozesse, die durch einen Kommunikationskanal oder eine Pipe verbunden sind
- transitive Kette: Ring- oder token-basierte Algorithmen.

Beispiel: File-Server



Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - Interaktionssemantik
  - Server-Klassifikation
  - Caching
  - Broker, Trader, Proxy, Balancer, Agent
- Peer-to-Peer Netzwerke
  - *unstrukturiert: zentralisiert, pur, hybrid*
  - strukturiert: DHT-basiert

Peer-To-Peer-Netzwerke allgemein

Wie wir bereits gesehen haben, sind Client-Server-Systeme geplant und werden administriert:

- Netzwerkverkehr konzentriert sich am und vor den Servern.
- Skalierbarkeit und Robustheit sind problematisch.

Im Gegensatz dazu bestehen Peer-to-Peer-Netzwerke aus autonomen und gleichberechtigten Rechnern, die sich selbst organisieren.[4]

Beispiele für bekannte Peer-to-Peer Applikationen sind:

- Filesharing: eDonkey, Gnutella
- VoIP: Skype
- E-Mail: SMTP

---

[4] Dies ist nur ein Definitionsversuch, denn nicht alle P2P-Netzwerke bestehen tatsächlich aus gleichberechtigten Knoten.

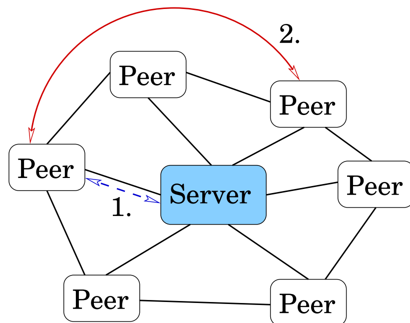
Peer-To-Peer-Netzwerke allgemein

- P2P-Netzwerke sind auf der Anwendungsschicht implementiert und bilden somit ein logisches, virtuelles Netzwerk auf ein physikalisches Netzwerk ab: Overlay-Netzwerk
- Die einzelnen Knoten (Peers) haben eindeutige IDs und bieten Inhalte an, konsumieren Inhalte und sind gleichzeitig Router im Overlay-Netzwerk.
- Peer bezeichnet im Englischen einen Ebenbürtigen oder Gleichgestellten. Ein P2P-Netzwerk ist also ein Netzwerk aus Gleichrangigen: keine Server mit speziellen Aufgaben.
- P2P-Netzwerke werden nicht nur zum illegalen Download von Musik- oder Videodateien genutzt: Skype[5] verwendet ein P2P-Netzwerk für sein Internet-Telefonsystem.

---

[5] Wie die Netzwerkstruktur genau funktioniert, lässt sich nur mutmaßen. Erste Erkenntnisse durch Nachrichtenanalyse wurden veröffentlicht in: An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Salman A. Baset and Henning Schulzrinne.

Die Dienstsuche erfolgt mittels eines zentralen Verzeichnisses, der eigentliche Datei-Download erfolgt direkt zwischen zwei Clients.



Nachteil: Der zentrale Server stellt einen Engpass dar und die Architektur ist weder fehlertolerant noch ausfallsicher.

zentralisiert

Beispiel: Napster (Mai 1999)

- Anwender laden nicht nur Daten vom Server, sondern stellen selber ihre eigenen Daten zur Verfügung.
- Die Teilnehmer etablieren ein virtuelles Netzwerk, das vollständig unabhängig vom physikalischen Netzwerk und administrativen Autoritäten oder Beschränkungen ist.
- Genutzt werden UDP- und TCP-Kanäle zwischen den Peers.
- Napster stellt einen zentralisierten Index bereit:
  - Die Clients laden die Liste der Dateien, die sie zur Verfügung stellen wollen, auf den zentralen Napster-Server.
  - Für einen Datei-Download befragen die Clients den Index-Server und erhalten eine vollständige Anbieterliste.
- Der Datenaustausch erfolgt unmittelbar zwischen den Peers, Napster ist am eigentlichen Datenaustausch nicht beteiligt.

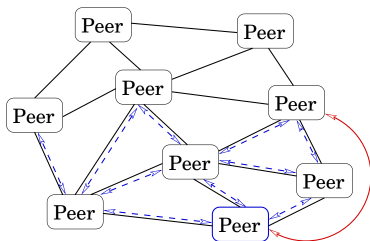
Wie es mit Napster weiterging:

- Ende des Jahres 1999 klagt die Recording Industry Association of America (RIAA) gegen Napster Inc., da im wesentlichen Musikdateien über Napster getauscht werden.
- Anfang 2001 wurden mehr als 2,5 Milliarden Dateien pro Monat via Napster getauscht, und im gleichen Jahr wird Napster Inc. verurteilt.
- Napster muss den Betrieb des zentralen Index-Servers einstellen, woraufhin das Netzwerk zusammen bricht.
- Sowohl rechtlich als auch technisch hat Napster an seinem Single Point of Failure versagt.

Da die Menschen trotzdem weiter illegal Musikdateien tauschen wollten, gab es sehr bald neue Peer-to-Peer-Ansätze. ☺

Ein reines P2P-Netzwerk besteht aus gleichartigen, gleichrangigen Knoten, es gibt keine spezialisierten Knoten.

Die Suche erfolgt mittels beschränktem Broadcast: Das TTL-Attribut (Time To Live) wird bei jedem Knoten um eins erniedrigt, bevor das Paket an alle Nachbarn weitergereicht wird.



Nachteile: Die Suche ist nicht zielgerichtet. Das Netzwerk ist um den Suchenden herum sehr stark belastet. Der Wert des TTL-Attributs muss nach erfolgloser Suche angepasst werden.

Beispiel: Gnutella 0.4 (März 2000)

- Reines P2P-Netzwerk: Es gibt keinen zentralen Index-Server. Der gemeinsame Dateizugriff (file sharing) ist dezentralisiert.
- Um den Anbieter einer Datei zu lokalisieren, wird ein Broadcast ins Netz geschickt, das Netzwerk wird geflutet.
- Skalierungsprobleme durch Fluten des Netzwerks.
  $\rightarrow$ Der Zusammenbruch des Netzwerks erfolgte relativ früh.
- Eine Suche kann irrtümlich erfolglos sein, wenn das TTL-Attribut zu klein gewählt wird.

Wie wird bei Gnutella 0.4 eine Datei gefunden?

- Ein Peer flutet eine Anfrage (Query), in der unter anderem die minimale Download-Rate und eine Liste von Schlüsselwörtern angegeben ist.
- Peers, die eine Datei anbieten, die der Anfrage entspricht, schicken eine Antwort (QueryHit), die die IP-Adresse, den Port, die GUID, die Verbindungsgeschwindigkeit und für jede passende Datei einen MD5-Hash, die Größe und den Namen der Datei enthält.
- Aus den eintreffenden Antworten wird die „beste“ ausgewählt.
- Schließlich wird die Verbindung zum anbietenden Peer aufgebaut und der Download der Datei beginnt.

Diese Art der Suche entspricht einer parallelen Breitensuche und führt zu einem hohen Nachrichtenaufkommen.

Alternativ:

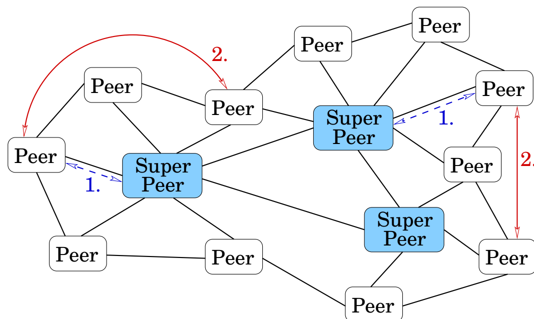
- Eine als Random-Walk bezeichnete Suche.
    - Entspricht einer sequentiellen Tiefensuche.
    - Sehr hohe Latenz.
- Spezielle Knoten: Super-Peers
    - Jeder Peer ist mit einem Super-Peer verbunden.
    - Gesucht wird ausschließlich über Super-Peers.
    - Super-Peers sind leistungsfähiger und enger vermascht.

Wie kann ein neuer Knoten dem Netzwerk beitreten, wenn die Knoten des Netzwerks nicht bekannt sind?

Bootstrap:

- Versuche einen der Peers zu erreichen, der in der der Software mitgelieferten Liste aufgeführt ist.
- Von diesem aus wird die k-Nachbarschaft mittels Ping/Pong erkundet:
  - Ein neu hinzugekommener Peer flutet eine Ping-Nachricht.
  - Alle Peers, die diese Nachricht erhalten, antworten mit einem Pong: IP-Adresse, Port, Anzahl und Größe der bereitgestellten Dateien und eine GUID.
- Die Rückmeldungen werden in einer Liste gespeichert. Diese Liste wird anstelle oder zusätzlich zu der in der Software gespeicherten Liste genutzt, um sich beim nächsten Start ins Netzwerk einzubinden.

Diese Mischform soll die Nachteile der anderen Formen vermeiden. Schnelle Rechner bieten spezielle Dienste und Daten an.



Beispiel: Gnutella 0.6 (Frühling 2001)

- Peers (leaf nodes) geben ihr Inhaltsangebot ihrem Super-Peer bekannt.
- Die Super-Peers speichern lokale Routingtabellen.

hybrid

Wie wird ein Super-Peer gewählt?

- Ein Peer tritt dem Netzwerk bei, indem es sich mit einem Super-Peer verbindet.
- Ein neuer Super-Peer muss dann gewählt werden, wenn
  - ein Super-Peer das Netzwerk verlässt, oder
  - ein Super-Peer zu viele Kindknoten besitzt, oder
  - ein Super-Peer zu wenige Kindknoten besitzt.
- Die Wahl wird beeinflusst durch die CPU-Leistung und Speichergröße des Peers, der Bandbreite des Netzwerkzugangs sowie durch die Verfügbarkeit (uptime) des Peers.

Übersicht

Architekturmodelle

- Client/Server-Strukturen
  - Interaktionssemantik
  - Server-Klassifikation
  - Caching
  - Broker, Trader, Proxy, Balancer, Agent
- Peer-to-Peer Netzwerke
  - unstrukturiert: zentralisiert, pur, hybrid
  - *strukturiert: DHT-basiert*

Verteilte Indexierung:

- Die bekannten Ideen zur Realisierung eines verteilten gemeinsamen Speichers (shared memory) werden für P2P-Netzwerke übernommen.
- Die Rechner-Knoten und die Daten werden in den gleichen Adressraum abgebildet.
- Die Knoten erhalten Routing-Informationen.

→ Effizientes und zuverlässiges Auffinden der Ziele.

Basiert auf speziellen Hash-Funktionen: consistent hashing[6]

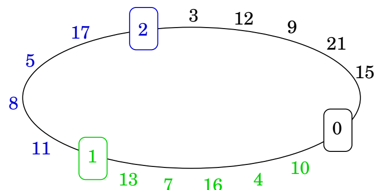
[6] Heute auch üblich in NoSQL-Datenbanken und in Cloud-Systemen. NoSQL steht übrigens nicht für *no SQL*, sondern für *not only SQL* und soll ausdrücken, das auch nicht strukturierte Daten abgespeichert werden können.

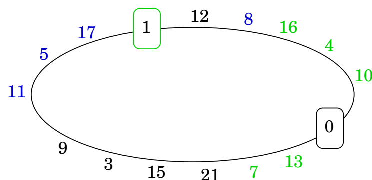
Distributed Hash Table

Grundidee:

- Es liegt eine feste Netz-Struktur zugrunde, bspw. ein Ring.
- Abbildung der Knoten und Daten in den selben Adressraum.
- Der Speicherplatz ist auf einem Netzwerk mit mehreren hundert oder mehreren tausend Peers verteilt.
- Peers und Inhalte nutzen flache Identifizierer, z.B. die IP-Adresse kombiniert mit der Port-Nummer.
- Die Knoten sind verantwortlich für fest definierte Teile des Adressraums.
- Das System ist in der Lage, Zu- und Abgänge von Peers zu kompensieren, ohne dass der laufende Betrieb gestört wird.
- Die Assoziation von Daten zu Knoten ändert sich, wenn Knoten hinzukommen oder verschwinden.
- Starke Knoten verwalten mehr Daten als schwächere Knoten.

Versuch: Betrachte Ring mit 3 Knoten, und $hash(x) := x \bmod 3$.





- Fällt Knoten 2 aus, gibt es nur noch zwei Speicherbereiche:
  - Wenn mit einer neuen Hash-Funktion $hash'(x) := x \bmod 2$ gearbeitet wird, müssen viele Datensätze verschoben werden, obwohl sie überhaupt nichts mit dem ausgefallenen Knoten zu tun haben. Außerdem: kommunizieren der neuen Funktion.
- Kommt dann wieder ein Knoten neu hinzu, muss die ursprüngliche Datenverteilung wieder hergestellt werden.
- Außerdem werden die Daten nicht entsprechend der Rechenleistung auf die Knoten verteilt.

Distributed Hash Table

Dieser Ansatz ist also ungeeignet!

Stattdessen nutzen alle Knoten zu jeder Zeit dieselbe Hash-Funktion, bspw. SHA-1.

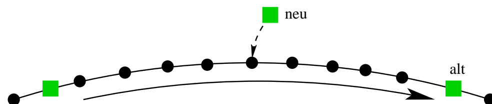
- Die Knoten werden gemäß des Hash-Wertes ihrer GUIDs auf dem Adressring angeordnet.
- Die Datensätze werden ebenfalls gemäß ihres Hash-Wertes auf dem Adressring angeordnet und auf dem im Ring im Uhrzeigersinn nach ihnen liegenden Knoten gespeichert.
- SHA-1 liefert einen Output der Länge 160 bit. Der Hash-Wert liegt also im Bereich zwischen 0 und 2^{160}. Es gilt:
  - Die Anzahl der Knoten im Netzwerk ist verschwindend gering im Vergleich zum Hash-Wert.
  - Die Hash-Werte sind gleichverteilt.

Also sind die Peers gleichmäßig ausgelastet.

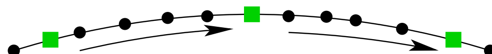
Distributed Hash Table

Wie werden Zu- oder Abgänge von Peers behandelt?

- Ein neuer Knoten wird gemäß seines Hash-Wertes auf dem Ring platziert:



  - Der neue Peer kontaktiert den alten Peer, um die Zuständigkeiten neu zu regeln.



  - Nur wenige Datensätze werden vom ursprünglichen Knoten auf den neuen verschoben. Alle anderen Datensätze sind nicht betroffen von dieser Erweiterung.

- Der Austritt eines Knotens wird entsprechend behandelt.

Distributed Hash Table

Wie werden unterschiedlich starke Peers behandelt?

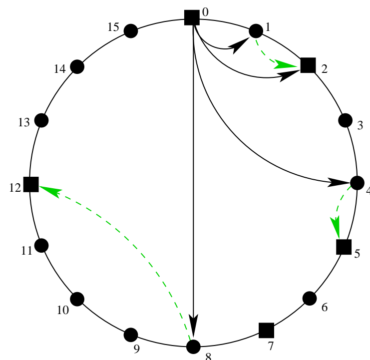
- Für leistungsfähige Knoten werden entsprechend viele virtuelle Knoten mit eigener GUID erzeugt: anderer Hash-Wert, andere Platzierung auf dem Ring.
- Auf den realen Knoten werden dann alle Daten kopiert, deren Adressen vor den zugehörigen virtuellen Knoten auf dem Ring angeordnet sind.

Wie die Daten auf die einzelnen Peers verteilt werden, haben wir gerade gesehen. Aber wie erfolgt der Zugriff auf die Daten? Jeder Knoten kennt nur den nachfolgenden Knoten, denn die zugrunde liegende Topologie ist ein Ring:

- Beim Zugriff auf einen Wert muss im schlimmsten Fall eine Nachricht einmal komplett durch das Netzwerk, also über n Peers, geschickt werden. $\rightarrow$ ineffizient!
- Bei Ausfall eines einzigen Peers zerfällt der Ring in zwei Teile. $\rightarrow$ nicht ausfallsicher!

Um beide Probleme zu lösen, werden Finger-Zeiger in der Routingtabelle jedes Peers gespeichert:

- pro Knoten werden zusätzliche Zeiger gespeichert
- die Distanzen, die die Zeiger überdecken, wachsen exponentiell



schwarze Pfeile: Der i-te Finger-Zeiger eines Knotens zeigt auf den 2^{i-1}-ten Nachfolger.

grüne Pfeile: Falls dieser Nachfolger nicht mit einem Peer besetzt ist, wird der Finger-Zeiger so geändert, dass er auf den für diesen Hash-Wert zuständigen Peer zeigt.

Haben wir durch unsere Hash-Funktion einen Wertebereich der Größe 2^m, dann werden m Zeiger in jedem Knoten gespeichert.

Die Anzahl der Zeiger wächst also nur logarithmisch mit der Größe des Netzwerks. Also bleibt die Größe der Routing-Tabelle auch für sehr große Netze überschaubar.

Man kann sogar zeigen, dass mit hoher Wahrscheinlichkeit nur $O(\log(n))$ Einträge der Routingtabelle tatsächlich auf verschiedene Rechner verweisen.

Das liegt daran, dass gerade die ersten Finger-Zeiger nur eine geringe Distanz überbrücken und oft auf freie Plätze zeigen. Daher werden diese Zeiger auf den selben Rechner *umgebogen*.

Eigenschaften der Einträge der Finger-Tabelle:

- Der letzte Eintrag der Tabelle zeigt auf einen Knoten, der mindestens eine halbe Umrundung des Ringes entfernt liegt.
- Der vorletzte Eintrag der Tabelle zeigt auf einen Knoten, der mindestens eine viertel Umrundung entfernt ist usw.

→ Das Routing im Chord benötigt höchstens $O(m)$ Hops, wenn der Chord die Größe 2^m hat.

Routing im Chord

Knoten p sucht den Wert x an der Position $hash(x) = k$:

- Knoten p prüft, ob sein Nachfolger für den gesuchten Wert verantwortlich ist. Falls ja, wird die Anfrage an diesen Nachfolger geschickt und das Verfahren endet.
- Sonst routet Knoten p seine Anfrage nach dem Wert x an den entferntesten Knoten $finger(i) = p'$, für den $hash(p') \leq k$ gilt.
- Anschließend routet Knoten p' nach dem gleichen Verfahren eine Anfrage nach Wert x an einen Knoten p'' weiter.

Mit hoher Wahrscheinlichkeit werden sogar nur $O(\log(n))$ Hops benötigt, wenn $n \ll 2^m$ viele Peers im Netzwerk vorhanden sind.

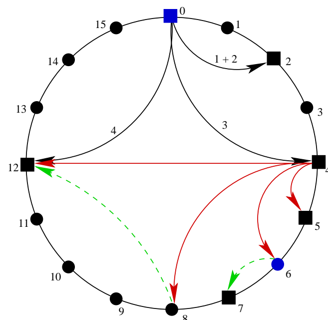
Also auch bei sehr großen Netzwerken ist das Routing effizient!

Routing im Chord

Beispiel: Anfrage nach Wert 3510 auf Position $hash(3510) = 6$.

Der Einfachheit halber nutzen wir die Hash-Funktion $hash(x) := x \bmod 16$.

Der Peer auf einem Knoten p habe die Einträge $finger_p(1)$ bis $finger_p(4)$, da unser Netzwerk aus $2^4 = 16$ Knoten besteht.



1. $hash(finger_0(3)) = 4 \leq 6 < hash(finger_0(4)) = 12$
   $\rightarrow$ Knoten 0 routet Anfrage zum Knoten 4.
2. $hash(finger_4(1)) = 5 \leq 6 < hash(finger_4(2)) = 7$
   $\rightarrow$ Knoten 4 routet Anfrage zum Knoten 5.
3. Der Nachfolger von Knoten 5 ist der Knoten 7, und dieser ist zuständig für den Wert auf Position 6. Das Verfahren endet damit, dass Knoten 5 die Anfrage zum Knoten 7 routet.

Wenn ein neuer Knoten p_{neu} dem Netzwerk beitritt, kann dessen Routing-Tabelle berechnet werden.

- Beim Bootstrap findet p_{neu} einen Knoten p des Netzwerks.
- Knoten p bestimmt den Knoten p_{alt}, der bisher für die Verwaltung der Daten zuständig war, für die ab jetzt Knoten p_{neu} zuständig sein wird. Diese Suche kostet $O(\log(n))$ Hops.
- Der Knoten p_{neu} benachrichtigt den Knoten p_{alt} und bekommt einen Teil dessen Daten.
- Damit der neue Knoten im Ring problemlos eingefügt werden kann, bekommt jeder Knoten nicht nur einen Zeiger auf den Nachfolger, sondern auch einen Zeiger auf den Vorgänger.
- Die Finger-Tabelle des neuen Knoten muss erstellt werden:
  - Es müssen $O(\log(n))$ Einträge erstellt werden.
  - Für jeden Eintrag muss ein Knoten gesucht werden.
  - Die Suche eines Knoten kostet $O(\log(n))$ Hops.
  - $\rightarrow$ Die Finger-Tabelle kann also in Zeit $O(\log^2(n))$ erstellt werden.

Chord

Die Finger-Tabellen der anderen Knoten müssen ggf. auch angepasst werden. Man kann zeigen:

> Die Anzahl der Peers, die einen Finger-Zeiger auf einen Peer p besitzen, ist im Erwartungswert $O(\log(n))$ und mit hoher Wahrscheinlichkeit durch $O(\log^2(n))$ beschränkt.

Es müssen also mit hoher Wahrscheinlichkeit $O(\log^2(n))$ viele Peers über p_{neu} unterrichtet werden:

- Da die $O(\log^2(n))$ Peers in $O(\log(n))$ zusammenhängenden Bereichen liegen, gibt es $O(\log(n))$ Peers pro Bereich.
- Der erste Peer eines Bereiches wird benachrichtigt. Dazu kann die Suchfunktion verwendet werden.
- Die Nachricht wird entlang des Chord-Rings weitergeleitet, dazu sind weitere $O(\log(n))$ Nachrichten nötig.
- $O(\log(n))$ Bereiche mal $O(\log(n))$ Nachrichten ergibt $O(\log^2(n))$ Nachrichten, um einen neuen Peer einzufügen.

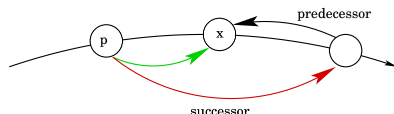
Chord

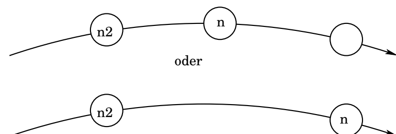
Da in einem Peer-To-Peer-Netzwerk mehrere Peers gleichzeitig dem Netzwerk beitreten können, ist obiges Verfahren nicht praxistauglich. Das Netzwerk würde zu stark belastet durch die Nachrichten zum Anpassen der Routing-Tabellen.

Daher wird in der Praxis ein Stabilisierungsprotokoll eingesetzt, das in regelmäßigen Zeitabständen aufgerufen wird, um die Vorgänger- und Nachfolgerzeiger sowie die Finger-Tabelle zu kontrollieren und ggf. zu korrigieren.

Betrachten wir zunächst das Prüfen und Korrigieren der Vorgänger- und Nachfolgerzeiger: Der Vorgänger vom Nachfolger eines Knotens sollte eigentlich der Knoten selbst sein. Wenn das nicht so ist, muss die Ring-Struktur korrigiert werden.

```
// Periodischer Aufruf: successor korrekt?
p.stabilize() {
    x = successor->predecessor;
    if (x in (p, successor))
        successor = x;
    successor.notify(p);
}
// n2 predecessor von n? Kein periodischer
// Aufruf, sondern aus stabilize() heraus!
n.notify(n2) {
    if ((predecessor is nil) or
            (n2 in (predecessor, n)))
        predecessor = n2;
}
```





Chord

Um die Finger-Zeiger zu korrigieren, muss für jeden bestehenden Eintrag der Routing-Tabelle eine Suchanfrage abgesetzt werden und der Routing-Eintrag ggf. korrigiert werden.

Damit diese Suchanfragen das Netzwerk nicht zu stark belasten, werden in einem Durchlauf nicht alle Einträge geprüft und korrigiert, sondern immer nur ein zufällig ausgewählter Eintrag:

- Eine korrekte Finger-Tabelle hat nur Einfluss auf die Dauer einer Suchanfrage, aber nicht auf deren korrekte Ausführung.
- Die Korrektheit einer Suchanfrage ist immer dann gegeben, wenn die Vorgänger- und Nachfolgerzeiger korrekt sind.
- Fehlerhafte Finger-Zeiger sind daher eine Zeit lang tollerierbar.

Ein weiteres Problem sind Knoten, die sich nicht ordentlich vom Netzwerk abmelden, sondern spontan ausfallen. Die auf diesen Knoten gespeicherten Daten gehen verloren. Damit das nicht passiert, müssen Daten repliziert werden:

- Eine zweite Hash-Funktion verteilt die Daten anders auf die Knoten, und zwar mit hoher Wahrscheinlichkeit so, dass ein Datenblock auf verschiedenen Peers landet.
- Falls die erste Hash-Funktion keinen Knoten liefert, der die Daten gespeichert hat, wird eine zweite Anfrage mit der anderen Hash-Funktion gestartet.

Außerdem wird nicht nur ein Nachfolger und ein Vorgänger zu einem Knoten abgespeichert, sondern k viele, damit der Ring wieder geschlossen werden kann.

Content Addressable Network

Bei CAN werden die Daten nicht auf einem Ring verteilt, sondern man weist den Daten Orte in einem zweidimensionalen Feld zu:

- Jeder Knoten übernimmt ein Rechteck aus dem Schlüsselraum.
- Beim Hinzufügen eines Knotens wird ein Rechteck aufgeteilt.
- Gleiche Last für alle Knoten bei gleichverteilter Nutzung des Schlüsselraums.

weiterführende Literatur:

- Peter Mahlmann, Christian Schindelhauer: Peer-To-Peer-Netzwerke. Springer Verlag, 2007.
- `http://sarwiki.informatik.hu-berlin.de/Chord`

Übersicht

Verteilte Systeme

- Einführung
- Programmiermodelle und Entwurfsmuster
- Architekturmodelle
- *Verteilte Algorithmen*
- Dienste

Übersicht

Verteilte Algorithmen

- *Logische Ordnung von Ereignissen*
- Wechselseitiger Ausschluss (Konkurenzdienst)
- Wahlalgorithmen
- Konsensalgorithmen

Unterschiede zu zentralisiertem System

Es gibt keinen gemeinsamen Zustand.

Der globale Zustand eines verteilten Systems besteht aus den lokalen Zuständen aller Prozesse und allen noch im Umlauf befindlichen Nachrichten.

- Die Knoten in einem verteilten System kennen nur ihren eigenen Zustand und können keine Entscheidungen anhand des globalen Zustands fällen.
- Es hilft nicht, zunächst alle Informationen aller Knoten zu sammeln: Ein Knoten kann seinen Zustand bereits geändert haben, bis die Information beim Anfrager eintrifft. Die gefällte Entscheidung beruht daher auf alten und somit ungültigen Daten.[7]

[7] Während des Ermittelns des globalen Zustands könnte man Variablen, die für den globalen Zustand wichtig sind, mittels „globaler“ Semaphore vor Veränderung schützen!

Unterschiede zu zentralisiertem System

Die Abläufe sind unbestimmt.

In einem zentralisiertem System ist bei einem gegebenen Programm und einer gegebenen Eingabe nur eine Berechnung möglich.

Dahingegen ist in einem verteilten System die globale Reihenfolge der Ereignisse nicht deterministisch.

Beispiel: Ein Server erhält viele Anfragen von einer unbekannten Anzahl Clients.

- Der Server kann nicht warten, bis alle Anfragen eingetroffen sind, um sie dann in einer bestimmten Reihenfolge zu bearbeiten, denn die Anzahl ist für ihn unbekannt.
- Der Server bearbeitet die Anfragen also sofort, aber die Reihenfolge, in der die Anfragen eintreffen, kann bei jedem Programmlauf unterschiedlich sein.

Unterschiede zu zentralisiertem System

Es gibt keine gemeinsame Zeit.

Die Ereignisse sind nicht total geordnet:

- Für Ereignisse auf einer Maschine kann entschieden werden, ob ein Ereignis vor einem anderen liegt.
- Bei Ereignissen auf zwei Maschinen, wobei die Ereignisse nicht in einer Ursache-Wirkung-Relation stehen, kann dies im Allg. nicht entschieden werden.[8]

$\Rightarrow$ insgesamt: fehlerhafte verteilte Anwendungen

- *Wichtig:* Synchronisationspunkte setzen
- fehlerhafte Synchronisationspunkte mittels Monitorsystem entdecken: spiegelt die zeitliche Reihenfolge der Aktionen und Ereignisse wieder.

[8] UTC-Empfänger gehören nicht zur Standardausstattung eines Rechners und selbst dann sind noch geringe Zeitdifferenzen möglich.

Synchrones Modell:

- Die Ausführungszeit jedes Prozessschrittes hat bekannte untere und obere Grenzen.
- Jede übertragene Nachricht wird innerhalb einer bekannten, begrenzten Zeit empfangen.
- Der Abweichung der lokalen Uhren von der Echtzeit ist eine Grenze gesetzt.
- Aber: Alle Werte sind wahrscheinliche Grenzen.
  - Schwierige Festlegung von realistischen Werten.
  - Probleme mit Zuverlässigkeit.

Asynchrones Modell:

- Keine Beschränkung bzgl. Ausführungsgeschwindigkeit, Übertragungsverzögerung, Uhrabweichraten usw.
- Folgerungen: keine Annahme über Zeitintervalle erlaubt
    - Herstellen einer Abfolge von Ereignissen erfordert zusätzliche Konzepte: logische Zeit
    - Eine exakte Abstimmung zwischen Ereignissen ist in diesem Modell nicht möglich.
- Verteilte Systeme sind häufig asynchron:
    - Prozesse müssen Prozessoren und Netzwerke gemeinsam nutzen.
    - Angabe von Grenzen nicht möglich: Unvorhersehbare Auslastung der Prozessoren und Netzwerke.

Zeit in Verteilten Systemen

Zeitstempel werden benötigt, um

- Leistungsmessungen durchzuführen.
- die Aktualität von Daten zu bewerten.
- eine Totalordnung von Objekten zu bewirken: Wer zuerst kommt, malt zuerst, z.B. beim wechselseitigen Ausschluss.

Probleme bei Verteilten Systemen:

- Uhren sind mit Ungenauigkeiten behaftet: Eine eindeutige, globale Zeit ist nicht verfügbar.
- Selbst bei zentralen Zeitgeber: Ungenauigkeiten durch unterschiedliche Nachrichtenlaufzeiten.

Abfolge von Ereignissen

Beispiel: Nachrichtenaustausch zwischen Benutzern X, Y und Z.

- X sendet eine Nachricht mit dem Betreff `Meeting`.
- Y antwortet mit der Betreffzeile `Re: Meeting`.
- Z antwortet ebenfalls mit `Re: Meeting`, bezieht sich aber auf die Nachricht von X und von Y.

Aufgrund voneinander unabhängiger Verzögerungen bei der Auslieferung der Nachrichten und unterschiedlicher lokaler Zeiten kann ein vierter Benutzer A folgende Reihenfolge bekommen:

Eintrag	Von	Betreff
23	Z	`Re: Meeting`
24	X	`Meeting`
25	Y	`Re: Meeting`

Problem: A muss anhand der Texte die richtige Reihenfolge der Nachrichten ermitteln, um den Inhalt entsprechend zu verstehen.

Logische Ordnung von Ereignissen

Idee: Rechner einigen sich auf gemeinsame Zeit. Diese muss nicht mit der realen Zeit übereinstimmen. $\rightarrow$ logische Uhr

- Ordnung der Ereignisse in relativer anstatt absoluter Lage.
- Ist bis auf Instruktionsgranularität möglich.

Anforderung: Zeit auf allen Rechnern ist intern konsistent.

- Kausale Abhängigkeiten werden korrekt berücksichtigt.
- Ereignisse aus unabhängigen Abläufen müssen nicht relativ zueinander geordnet werden.

Relative Lage der Ereignisse:

- Wenn zwei Ereignisse im selben Prozess stattfinden, dann fanden sie in der Reihenfolge ihrer Beobachtung statt.
- Wenn eine Nachricht zwischen zwei Prozessen ausgetauscht wird: Sendeereignis liegt immer vor dem Empfangsereignis.

Logische Ordnung von Ereignissen

Ereignistypen:

- Lokale Ereignisse
- Sendeereignisse: Versenden einer Nachricht
- Empfangsereignisse: Empfangen einer Nachricht

Lamport entwickelte 1978 die Relation *liegt_vor*:

- Sind a und b Ereignisse des gleichen Prozesses, dann bedeutet $a \rightarrow b$: Ereignis a tritt vor Ereignis b auf, d.h. a *liegt_vor* b.
- Ist a ein Ereignis des Sendens einer Nachricht und b das Ereignis des Empfangens in einem anderen Prozess, so gilt $a \rightarrow b$.
- *liegt_vor* ist transitiv: Aus $a \rightarrow b$ und $b \rightarrow c$ folgt $a \rightarrow c$.
- Außerdem ist *liegt_vor* irreflexiv und asymmetrisch und daher eine (partielle) strenge Ordnungsrelation.

Logische Ordnung von Ereignissen

Falls zwei Ereignisse a und b in verschiedenen Prozessen liegen und kein Nachrichtenaustausch vorliegt, auch nicht indirekt über einen dritten Prozess, dann stehen a und b nicht in *liegt_vor*-Relation.

- Solche Ereignisse werden als nebenläufig (engl. concurrent) bezeichnet.

  Sollte besser als „steht in keinem kausalen Zusammenhang“ bezeichnet werden, da Nebenläufigkeit vermuten lässt, die Ereignisse würden zur selben Zeit auftreten.
- Keine Aussage über die relative Lage der Ereignisse möglich.

Lamport-Zeit: Algorithmus

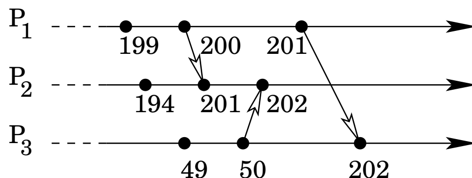
- Jeder Prozess P_i hat eine logische Uhr, die beim Auftreten eines Ereignisses a ausgelesen wird und den lokalen Zeitstempel $L_i(a)$ liefert.

  logische Uhr: monoton steigender Software-Zähler
- Dieser Wert $L_i(a)$ muss so angepasst werden, dass er als $L(a)$ eindeutig im ganzen verteilten System ist.
  - $L(a)$ ist der globale *Zeitstempel* des Ereignisses a.
  - Der Algorithmus zum Stellen der logischen Uhren muss folgendes umsetzen: Falls $a \rightarrow b$ gilt, dann muss auch $L(a) < L(b)$ gelten.
  - Zeitstempel dürfen also nur anwachsen, d.h. Zeitkorrekturen sind ausschließlich durch Addition von positiven Werten möglich.

Zeitstempel

Vorgehensweise zur Modellierung von *liegt_vor* im Prozess P_i:

- lokales Ereignis: der Software-Zähler L_i wird erhöht, also $L_i = L_i + 1$
- Sendeereignis: der Software-Zähler wird erhöht $L_i = L_i + 1$ und der neue Wert L_i wird zusammen mit der Nachricht m verschickt.
- Empfangsereignis: Die Nachricht m und der Wert L_k werden entnommen und $L_i = \max(L_i, L_k) + 1$ gesetzt.



Also: Die Uhr eines Prozesses wird vorgestellt, wenn er eine Nachricht erhält mit einem Zeitstempel größer als sein eigener Zeitstempel.

Problem: Identische Zeitstempel für zwei Ereignisse a und b in unterschiedlichen Prozessen P_i und P_j möglich.

totale Ordnung: Beziehe Prozess-Identifikationsnummer mit ein: Ereignis a im Prozess P_i erhält als globalen Zeitstempel $L(a) = (L_i(a), P_i)$. Dann gilt

$$(L_i(a), P_i) < (L_j(b), P_j) \iff \begin{cases} L_i(a) < L_j(b) \\ \quad \text{oder} \\ L_i(a) = L_j(b) \land P_i < P_j \end{cases}$$

Zeitstempel

Anwendung:

- bildet Grundlage für Monitor- und Debugging-System für verteilte Systeme
- verwendet in Algorithmus zur Lösung des Konkurrenzproblems für Transaktionen
- verwendet in verteiltem Algorithmus für wechselseitigen Ausschluss

Übersicht

Verteilte Algorithmen

- Logische Ordnung von Ereignissen
- *Wechselseitiger Ausschluss (Konkurenzdienst)*
- Wahlalgorithmen
- Konsensalgorithmen

Wechselseitiger Ausschluss

Ziel: Koordination von Aktivitäten, z.B. beim Zugriff auf gemeinsam genutzte Ressourcen wie einen Drucker. Zwei Druckaufträge sollen nicht durcheinander, sondern hintereinander gedruckt werden.

Grundsätzliche Realisierungsmöglichkeiten:

- Ein Server sammelt alle notwendigen Informationen und legt die Entscheidung fest.
  - Vorteil: Einfache Realisierung
  - Nachteil: Single Point of Failure
- Dynamische Festlegung in einer Gruppe von verteilten Komponenten.
  - Vorteil: kein Single Point of Failure
  - Nachteil: komplexe Umsetzung

Wechselseitiger Ausschluss

In Systemsoftware wird der wechselseitige Ausschluss realisiert mittels binärer Semaphore.

In verteilten Systemen existiert kein gemeinsamer Speicher. Lösungen basieren auf dem Verschicken von Nachrichten.

Einfache Lösung mittels zentralem Server:

- Nur der Server gewährt Zutritt in kritische Bereiche.
- Ein Prozess sendet eine Nachricht: request
- Eintrittswünsche werden in einer Warteschlange verwaltet.
- Nur einem Prozess wird der Eintritt erlaubt: reply
- Prozess muss beim Verlassen des kritischen Abschnitts den Server benachrichtigen: release
- Problem, falls Prozess im kritischen Abschnitt hängen bleibt oder die Release-Nachricht verloren geht.

Wechselseitiger Ausschluss

Bewertung verteilter Algorithmen:

- *Bandbreite:* Anzahl/Größe der gesendeten Nachrichten
  oder: Anzahl Nachrichten pro Ein-/Austritt
- *Clientverzögerungen:* Werden verursacht durch Warten auf Ein-/Austritt aus dem kritischen Bereich.
- *Systemdurchsatz:* Geschwindigkeit, mit der die Gesamtheit der Prozesse auf den kritischen Bereich zugreifen kann.

Token-basierter Algorithmus im Ring:

- Wechselseitiger Ausschluss zwischen n Prozessen ohne zusätzlichen Server-Prozess.
- Prozesse sind als logischer Ring angeordnet: Der Prozess P_i ist mittels eines Kommunikationskanals mit dem Prozess $P_{(i+1) \bmod n}$ verbunden.
- Vorgehensweise:
  - Eintritts-Token wird als Nachricht weitergereicht.
  - Prozesse, die nicht in den kritischen Abschnitt eintreten wollen, leiten das Token einfach weiter.
  - sonst: Prozess behält Token, nach Verlassen des kritischen Abschnitts Token weitergeben an Nachbarn.

Bewertung:

- Unnötiger Token-Wechsel, falls kein Prozess in den kritischen Abschnitt will. Resultat: hohe Bandbreite.
- Fehleranfällig:
  - Ring muss immer geschlossen sein: Rekonfiguration nach Ausfall eines Prozesses.
  - Token-Fehler (Duplikat oder Verlust) müssen erkannt und behoben werden. Erfordert zentrale Überwachung.
- Eintrittsverzögerung: 0 bis $n - 1$
- Nachrichten pro Ein-/Austritt: 1 bis ∞

Abfragebasierter Algorithmus:

- Der Prozess, der in den kritischen Abschnitt eintreten will, sendet eine Anforderung als Multicast und wartet auf die Erlaubnis aller anderen Prozesse.
- Unterscheide zwei Fälle: Der angefragte Prozess ist
  - selber nicht am Eintritt in den kritischen Abschnitt interessiert: Erlaubnis direkt verschicken.
  - selber interessiert: Prozess, dessen Anfrage früher war, erhält die Erlaubnis. Dies erfordert einen Einsatz von Zeitstempeln.

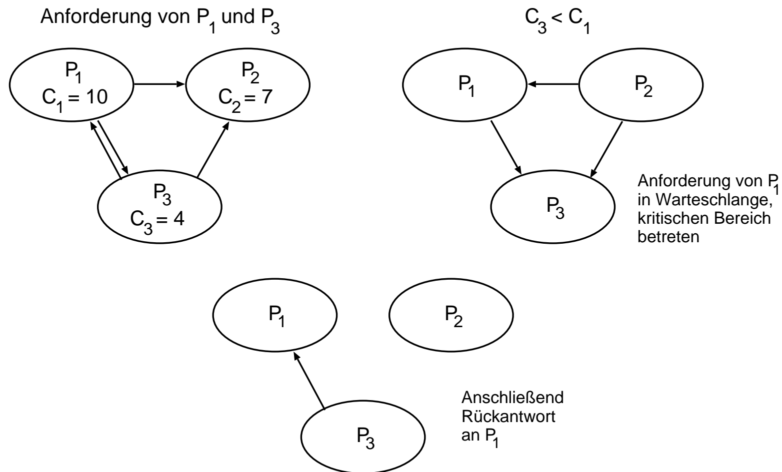
Voraussetzungen des abfragebasierten Algorithmus:

- Jeder Prozess hat eine logische Uhr.
- Anfragen bestehen aus Zeitstempel und Prozess-ID.
- Jeder Prozess kennt seinen Zustand.
  - `RELEASED`: Außerhalb des kritischen Bereichs
  - `WANTED`: Zugang wird angefordert
  - `HELD`: Prozess ist im kritischen Bereich
- Zuverlässige Kommunikation.

Wechselseitiger Ausschluss

Protokoll des abfragebasierten Algorithmus:

- Anfrage verschicken
- Antwort von Prozess P_i hängt von seinem Zustand ab:
  - `RELEASED`: Erlaubnis gewährt
  - `HELD`: keine Antwort, Anforderung in Warteschlange stellen
  - `WANTED`: falls Zeitstempel der Anfrage kleiner als eigener Zeitstempel, dann Antwort verschicken, sonst Anfrage in Warteschlange stellen
- Warte auf alle Antworten, dann Eintreten in kritischen Abschnitt.
- Verlassen des kritischen Abschnitts: In der Warteschlange befindlichen Prozessen eine Antwort schicken und aus der Warteschlange entfernen.


Anforderung von P1 und P3
P1
C1 = 10
P2
C2 = 7
P3
C3 = 4
C3 < C1
P1
P2
P3
Anforderung von P1 in Warteschlange, kritischen Bereich betreten
P1
P2
P3
Anschließend Rückantwort an P1

Bewertung:

- Hohe Bandbreite: $\Theta(n^2)$, falls alle Prozesse ungefähr gleichzeitig in den kritischen Abschnitt wollen.
- PID muss global eindeutig sein:
    - zentraler Manager
    - bei Prozesserzeugung Namen aller Prozesse erfragen und neuen Namen allen mitteilen
    - Struktur `IP:PID`
- fehleranfällig: Ausfall des Systems bei Ausfall eines Prozesses.

  Daher: Überwachung durch zentralen Manager erforderlich. Der Manager informiert die Prozesse über den Ausfall eines Prozesses P_k: Rückantwort von P_k muss nicht abgewartet werden, P_k muss nicht mehr um Erlaubnis gefragt werden.
- Prozesse, die nicht in k.A. wollen, werden mit Arbeit belastet: Anfrage annehmen und Erlaubnis verschicken.

Übersicht

Verteilte Algorithmen

- Logische Ordnung von Ereignissen
- Wechselseitiger Ausschluss (Konkurenzdienst)
- *Wahlalgorithmen*
- Konsensalgorithmen

Wahlalgorithmen

Algorithmen zur Wahl eines Prozesses, der dann eine bestimmte Aufgabe übernimmt bzw. eine besondere Rolle spielt: Koordinator, Monitor usw.

- typisch: Welcher Knoten wird zum Server deklariert?
- wichtig: Alle Prozesse müssen sich auf die Wahl einigen.

Erfordert ein Replizieren der Server: Bei Ausfall eines Servers kann ein anderer Server dessen Funktion übernehmen.

Nach einem Ausfall kann sich ein Prozess nach seinem Neustart wieder in die Menge der aktiven Prozesse eingliedern. Dazu startet er einfach eine neue Wahl.

Wahlalgorithmen

Eine solche Wahl wird oft durch die Bestimmung eines Extremwertes auf einer Ordnung der beteiligten Prozesse durchgeführt:

- Jeder Prozess hat eine global eindeutige Nummer, die allen anderen Prozessen bekannt ist.[9]
- Kein Prozess weiß, welcher andere Prozess zur Zeit aktiv ist, also funktioniert.
- Alle Algorithmen einigen sich o.B.d.A. auf den Prozess mit der höchsten Nummer.
- Jeder Prozess kennt nach der Wahl den gewählten Prozess. Erfordert ein spezielles Symbol zur Darstellung undefinierter Werte.

[9] Wie wird das realisiert? Mittels eines zentralen Managers?

Wahlalgorithmen

Zu beachten:

- Eine Kopie des Servers befindet sich auf n Rechnern: Nur ein Prozess ist tätig (Master), alle anderen dienen der Reserve.
- Ausfall des Masters wird z.B. durch Timeout-Kontrolle von den untergeordneten Prozessen bemerkt.
- Ein Prozess veranlasst eine Wahl, ein einzelner Prozess kann nur eine Wahl veranlassen.
- n Prozesse können n nebenläufige Wahlen veranlassen, wobei jede Wahl auch bei Nebenläufigkeit eindeutig sein muss.
- Ein Prozess ist zu jedem Zeitpunkt entweder an der Wahl beteiligt oder nicht.

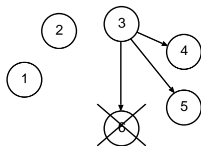
Bully-Algorithmus: Der Prozess P_i bemerkt Ausfall des Masters.

1. P_i schickt eine ELECTION-Nachricht an alle P_k mit $k > i$ und an den bisherigen Master und P_i wartet ein Zeitintervall T auf Antwort.
2. Erhält ein Prozess P_j eine ELECTION-Nachricht, schickt er eine ANTWORT-Nachricht und startet seinen eigene Auswahl.

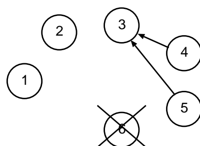
3a P_i erhält innerhalb von T keine ANTWORT-Nachricht: P_i bestimmt sich selbst zum neuen Master und schickt diese Info an alle Prozesse P_j mit $j < i$.

3b P_i erhält mindestens eine ANTWORT-Nachricht innerhalb des Zeitintervalls T: Warte ein Zeitintervall T' auf Bestätigung, dass ein Prozess P_k mit $k > i$ als neuer Master bestimmt wurde.

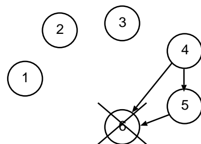
Falls innerhalb von T' keine Bestätigung eintrifft, startet P_i den Wahlalgorithmus erneut.



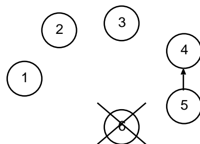
3 bemerkt, dass 6 ausgefallen ist, startet Wahl



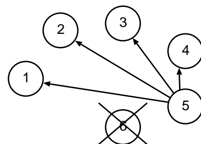
4 und 5 antworten,
3 beendet Wahl



4 und 5 starten Wahl



5 antwortet,
4 beendet Wahl



5 informiert alle, dass
er neuer Koordinator ist

Anmerkungen zum Bully-Algorithmus:

- Timeout T: schätze maximale Übertragungszeit T_{trans} und maximale Verarbeitungszeit $T_{process}$

$$T = 2 \cdot T_{trans} + T_{process}$$

- T' muss viel größer als T gewählt werden
- Anzahl verschickter Nachrichten:
  - $\Theta(n^2)$, wenn P_n ausfällt und P_1 den Ausfall bemerkt.
  - $\Theta(n^3)$, falls zusätzlich die Prozesse $P_{n-1}, P_{n-2}, \ldots$ ausfallen, bevor die Bestätigungsnachrichten geschickt werden können.

Ring-Algorithmus:

Prozesse bilden einen logischen Ring: Jeder Prozess kennt seinen Nachbarn, Nachrichten werden nur in eine Richtung gesendet.

Ist der Nachfolgeprozess ausgefallen, kann der Sender übergehen zu dessen Nachfolger, und falls der ausgefallen ist, auf dessen Nachfolger usw. Dies ist möglich, da aufgrund unserer Annahmen jede Prozessnummer allen anderen Prozessen bekannt ist.[10]

- Initialisierung:
  - Prozess P_i veranlasst die Wahl, markiert sich als Teilnehmer und sendet die Wahl-Nachricht inklusive seiner PID.
  - Jeder empfangende Prozess vergleicht die in der Nachricht enthaltene PID mit seiner eigenen PID.

[10] Wie realistisch ist diese Annahme?

- Vergleichsergebnisse:
  - Empfangene PID größer und Empfänger ist noch kein Teilnehmer: Nachricht unverändert weiterreichen, Prozess markiert sich als Teilnehmer.
  - Empfangene PID kleiner und Empfänger ist noch kein Teilnehmer: Eintragen der eigenen PID anstelle der ursprünglichen PID, weitergeben der Nachricht an Nachbarn, markieren als Teilnehmer.
  - Empfangene PID ist gleich der eigenen PID: Der aktuelle Prozess hat die größte PID, die Wahl ist entschieden. Markierung löschen, Ende-Nachricht mit PID an Nachbarn weitergeben.
  - Empfangende Prozesse löschen die Markierung und reichen PID weiter.

Frage: Warum muss eine Ende-Nachricht verschickt werden? Wenn ein bereits markierter Prozess eine Nachricht bekommt, definiert die darin enthaltene PID den Wahlsieger.

Anmerkungen:

- Anzahl verschickter Nachrichten: $3n - 3$ im worst-case; ohne Ende-Nachricht würde sich die Anzahl Nachrichten auf $2n - 2$ reduzieren.
- Kommunikationsfehler oder Abstürze einzelner Komponenten führen zum Ausfall des gesamten Algorithmus.
- Geringerer Aufwand als Bully-Algorithmus.
- Es existieren verschiedene Variationen mit weniger Kommunikationen, dafür aber mehr verschickten Daten.

Frage: Warum bestimmt nicht jeder Prozess anhand der eigenen Informationen den neuen Master? Jeder Prozess kennt doch jeden anderen.

Antwort: Prozesse könnten den Ausfall des Masters zu unterschiedlichen Zeiten bemerken. Dies führt zu Inkonsistenzen nach Neustart eines früheren, ausgefallenen Masters.

Übersicht

Verteilte Algorithmen

- Logische Ordnung von Ereignissen
- Wechselseitiger Ausschluss (Konkurenzdienst)
- Wahlalgorithmen
- *Konsensalgorithmen*

Konsensalgorithmen

Konsens: Übereinstimmung bzgl. eines gemeinsamen Wertes.

Motivation: Bei fehlertoleranten Systemen gibt es mehrere Informationsquellen, z.B. Sensoren, und steuernde Einheiten. Die Steuereinheiten müssen sich auf einen Konsens einigen, um zu entscheiden, ob eine Aktion durchgeführt oder abgebrochen wird.

Beispiele:

- Abbruch eines Startvorgangs z.B. bei Flugzeugen.
- Einleiten einer automatischen Bremsung z.B. bei Zügen.
- Überweisung: Beide Rechner buchen gleichen Betrag ab.
- Eintritt in k.A.: Einigen auf gewählten Prozess.

Kein Problem bei fehlerfreien Kommunikationsmedien, Prozessen und Prozessumgebungen (Hardware).

Fehlerquellen:

- fehlerhafte Prozesse: unvorhersehbares Verhalten
  - Fail-stop failure: Prozess stoppt nach Fehler
    - $\rightarrow$ Entdeckung des Fehlers einfach
  - Byzantine failure: Prozess arbeitet fehlerhaft und erzeugt falsche Ergebnisse
    - $\rightarrow$ inkorrekte Nachrichten und Gefahr für Integrität des Verteilten Systems
- fehlerhafte Kommunikationskanäle: Verfälschung oder Verlust einer Nachricht

Konsensproblem: Jeder Vertreter aus einer Menge von Prozessen muss sich nach endlicher Zeit für einen Wert aus einer Menge von möglichen Werten entscheiden. Außerdem müssen sich am Ende alle Prozesse auf den selben Wert geeinigt haben.

Ablauf: Jeder Prozess P_i beginnt im nicht dedizierten Zustand und schlägt einen Wert v_i vor.

- Prozesse kommunizieren und tauschen Werte aus.
- Jeder Prozess P_i setzt seine Entscheidungsvariable d_i auf den Wert, auf den sich die Prozesse geeinigt haben.
- Übergang in dedizierten Zustand: d_i darf nicht mehr geändert werden.

Konsensalgorithmen

Für jeden Konsensalgorithmus muss gelten:

- *Terminierung:* Jeder korrekte Prozess setzt irgendwann die Entscheidungsvariable.
- *Einigung:* Entscheidungswert aller korrekten Prozesse ist gleich.
- *Integrität:* Haben alle korrekten Prozesse denselben Wert vorgeschlagen, dann hat jeder korrekte Prozess im dedizierten Zustand diesen Wert gewählt.

Konsensalgorithmen

Kein Algorithmus kann Konsens in einem asynchronen System garantieren,

- wenn Prozesse abstürzen können.
  - Prozesse können zu beliebigen Zeitpunkten auf Nachrichten antworten, daher ist keine Unterscheidung zwischen langsamen und abgestürzten Prozessen möglich.
  - Beweis der Nicht-Existenz von Fischer et al.
- wenn die Kommunikation unzuverlässig ist.
  - Kommunikation mittels Quittierungen: Ein Prozess P_i schickt P_j seine Information, P_j bestätigt den Erhalt der Information, P_i bestätigt den Erhalt der Quittung.
  - Problem: Quittung kann verloren gehen (unsicher)
  - Beweis durch Widerspruch: keine minimale Sequenz

Ansätze zur Umgehung des Unmöglichkeitsergebnisses:

- Verwende partiell synchrone Systeme: abgeschwächte synchrone Verteilte Systeme, die aber strengeren Voraussetzungen als asynchrone Systeme unterliegen.
- z.B. Fehlerdetektoren: Prozesse einigen sich darauf, dass ein Prozess, der eine bestimmte Zeit nicht geantwortet hat, als ausgefallen betrachtet wird und von Entscheidungen ausgeschlossen wird.

Problem der „byzantinischen Generäle“:

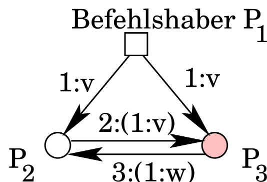
- spezielle Variante des Konsensproblems
- Situation:
  - Mehrere Divisionen osmanischer Generäle belagern im Jahr 1453 n.Chr. Konstantinopel (Byzantion).
  - Jede der geografisch verstreuten Divisionen wird von einem General kommandiert.
  - Generäle müssen übereinstimmen, ob im Morgengrauen ein Angriff gestartet werden soll.
    Ein Angriff mit nicht maximaler Stärke würde in einer Niederlage enden.

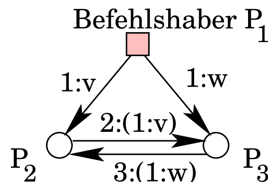
Problem der byzantinischen Generäle: (Fortsetzung)

- Die Kommunikation zwischen den Divisionen erfolgt mit Botschaftern.
- Übereinstimmung der Generäle kann verhindert werden:
  - Einige Botschafter werden gefangen genommen.
    $\rightarrow$ unzuverlässige Kommunikation
  - Einige Generäle sind Verräter, die die Übereinstimmung verhindern wollen.
    $\rightarrow$ fehlerhafte Prozesse
- Voraussetzungen:
  - Bis zu f der N Prozesse weisen beliebige Fehler auf: beliebige Nachricht mit beliebigem Wert senden
  - Ausbleibende Nachrichten können durch Timeouts erkannt werden.
  - Übertragungskanäle sind privat: kein Abhören möglich

zunächst: Befehl wird von einem Befehlshaber erteilt.

- Fällt einer von 3 Prozessen aus, so gibt es keine Lösung
- allgemein: Erweiterung für $N \leq 3 \cdot f$

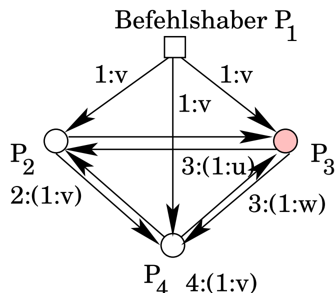


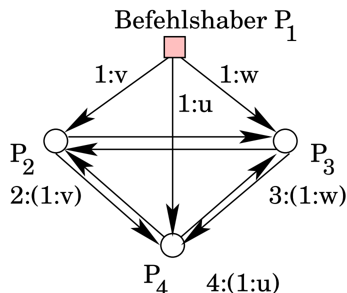


Problem: P_2 kann nicht erkennen, ob der Befehl des Befehlshabers oder des Untergebenen P_3 falsch ist

Lösung für $N = 4$ und einem fehlerhaften Prozess:

- Die korrekten Generäle erzielen Einigung in zwei Nachrichtenrunden:
  - Befehlshaber sendet Wert an jeden Untergebenen.
  - Jeder Untergebene sendet empfangenen Wert an die gleichgestellten Generäle.
- Jeder General empfängt einen Wert vom Befehlshaber und $N - 2$ Werte von gleichgestellten Generälen:
  - Wenn alle untergeordneten Generäle die gleiche Wertmenge aufzeichnen, dann ist der Befehlshaber fehlerhaft.
  - Bei abweichenden Wertmengen ist ein untergebener General fehlerhaft. Bestimmung durch einfache Mehrheitsfunktion.





P_2: `majority(v,-,u,v) = v`
P_4: `majority(v,v,w,-) = v`

P_2: `majority(v,-,w,u) = %`
P_3: `majority(w,v,-,u) = %`
P_4: `majority(u,v,w,-) = %`

Verallgemeinerung: Alle Generäle sind gleichberechtigt.

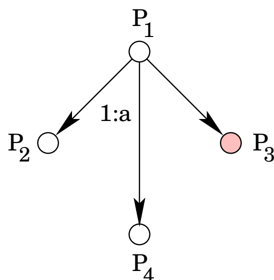
Komplizierter rekursiver Algorithmus, der aber für $f = 1$ nur 2 Runden benötigt:

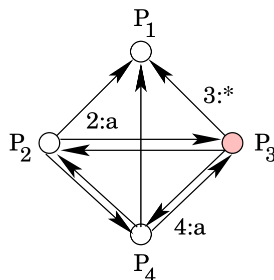
- Ein Prozess P_k beginnt die Wahl: Senden seines Wertes d_k an alle anderen.

  Anmerkung: Wert d_i vom Prozess P_i ist gleich d_k, falls Sender, Empfänger und Kommunikation o.k. sind.
- Jeder Prozess P_i sendet seinen Wert d_i allen anderen Prozessen.
- Jeder Prozess sendet die in der ersten Runde erhaltenen Informationen zu allen anderen Prozessen.

$\rightarrow$ Prozess speichert $n \times n$-Matrix anstelle eines Vektors.

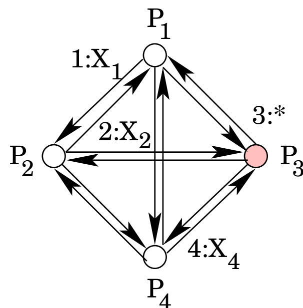
Runde 1:





	init	*nach 1*	*nach 2*
P_1	$d_1 = a$	$X_1 = (a, \cdot, \cdot, \cdot)$	$X_1 = (a, a, *, a)$
P_2	$d_2 = a$	$X_2 = (a, a, \cdot, \cdot)$	$X_2 = (a, a, *, a)$
P_3	$d_3 = *$	$X_3 = (*, \cdot, *, \cdot)$	$X_3 = (*, *, *, *)$
P_4	$d_4 = a$	$X_4 = (a, \cdot, \cdot, a)$	$X_4 = (a, a, *, a)$

Runde 2:



	vorher	*nachher*
P_1	$X_1 = (a, a, *, a)$	$X_1, X_2, *, X_4$
P_2	$X_2 = (a, a, *, a)$	$X_1, X_2, *, X_4$
P_3	$X_3 = (*, a, *, a)$	$*, *, *, *$
P_4	$X_4 = (a, a, *, a)$	$X_1, X_2, *, X_4$

Damit weiß jeder Prozess welches Resultat die anderen Prozesse haben und kann daraufhin eine Entscheidung fällen.

Übersicht

Verteilte Systeme

- Einführung
- Programmiermodelle und Entwurfsmuster
- Architekturmodelle
- Verteilte Algorithmen
- *Dienste*

Übersicht

Dienste

- *Dienste unter Linux*
- Namensdienst (DNS, DCE Directory Service)
- Verzeichnisdienst (LDAP)
- File-Dienst (NFS, DFS)
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- Zeitdienst (NTP)
- Sicherheit (SSL, Kerberos)

Linux

Linux bietet verschiedene Netzwerkdienste:

- Eine Liste möglicher Dienste finden Sie in `/etc/services`.
- Dienste werden von Hintergrundprozessen, genannt Dämonen (daemon), verwaltet.
- Programme werden bei Anforderung aktiv, führen die Kommunikation mit dem anfordernden System durch und die Dienstleistung aus.
- Gestartet werden die Dienste entweder direkt oder mittels `inetd`.
- Beispiele: ftp, telnet, ssh, daytime, http.

inetd: It listens for connections on certain internet sockets. When a connection is found on one of its sockets, it decides what service the socket corresponds to, and invokes a program to service the request. After the program is finished, it continues to listen on the socket.

- Netzwerkdämon, verwaltet verschiedene Protokolle.
- Konfiguration mittels `/etc/inetd.conf`. (Versions-abhängig)
- Um das Netzwerk-Handling nur einmal implementieren zu müssen, sind einfache Dienste[11] direkt implementiert: echo, time, daytime
- Dienstprogramme werden nur bei Bedarf geladen.
  $\rightarrow$ Ressourcen schonen! Effizienz?

[11] siehe `http://www.rhyshaden.com/ip_small.htm` für Details zu den Diensten

Bedeutung der Spalten in inetd.conf:

1. Service-Name (zugehöriger Port in `/etc/services`)
2. Socket-Typ: `stream` oder `dgram`
3. Protokoll: `tcp` oder `udp`
4. Flags: `wait` oder `nowait` (nach Anforderung wieder frei)
5. Privilegien für das Programm: `root`, `nobody`, ...
6. Pfad zum ausführenden Programm plus Argumente

Beispiele:

```
time    stream  tcp  nowait  root  internal
telnet  stream  tcp  nowait  root  /usr/sbin/tcpd  telnetd
```

tcpd: TCP-Wrapper

- Sicherheitsprogramm: Prüfe, ob angeforderte Verbindung zulässig ist. → Entweder Verbindung beenden oder angegebenes Programm aufrufen.
- Feststellen der IP-Adresse des anfragenden Systems und vergleichen mit einem Regelsatz in `/etc/hosts.allow` und `/etc/hosts.deny`.
- Nähere Informationen in den man-pages über `tcpd(8)` und `hosts_acces(5)`.

Einfache Richtlinien:

- Alles abschalten, was nicht unbedingt notwendig ist: `echo`, `chargen`, `discard` usw. können durch gefälschte Pakete unnötig Last erzeugen.
- Alle r-Dienste abschalten: `rlogin`, `rsh` usw. gelten als hochgradig unsicher.

Mit einem Portscanner wie `nmap` können alle, auf einem Rechner aktivierten Dienste aufgelistet werden.

Starten von Diensten beim Systemstart: (Versions-abhängig)

- The scripts for controlling the system are placed in `/etc/init.d/`. These scripts are executed by `/sbin/init`.
- The configuration of `/sbin/init` is given by the file `/etc/inittab`.
- `/sbin/init` calls the run level master script `/etc/init.d/rc` to start or stop services provided by the other scripts under `/etc/init.d/`.
- To control the services of a run level, the corresponding scripts are linked into directories `/etc/init.d/rc<X>.d/`.

Übersicht

Dienste

- Dienste unter Linux
- *Namensdienst (DNS, DCE Directory Service)*
- Verzeichnisdienst (LDAP)
- File-Dienst (NFS, DFS)
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- Zeitdienst (NTP)
- Sicherheit (SSL, Kerberos)

Namensdienst

Ein Namensdienst erlaubt Zugriff auf Ressourcen über systemweite Namen anstatt über physikalische Adressen.

→ Auflistung mit Zuordnung zwischen Namen und Attributen von Benutzern, Computern, Diensten, entfernten Objekten, usw.

Unterscheidung:

- Ein *Namensdienst* entspricht einem Telefonbuch: Der Name des Dienstes muss für den Zugriff bekannt sein.
- Ein *Verzeichnisdienst* entspricht den Gelben Seiten: Der Name ist nicht bekannt, aber der gesuchte Benutzer, Computer, Dienst, usw. kann durch Metainformationen beschrieben werden.

Beispiele für Namensdienste:

- DNS: Auflösung von Internet-Adressen
- RMI registry bei Java RMI, CORBA Namensdienst

Beispiele für Verzeichnisdienste:

- X.500 Directory Service: Standardisierter Dienst für Zugriff auf Einheiten aus realer Welt, häufig auch für Informationen bzgl. Hardware/Software-Dienste.
- CORBA Traderdienst: Attributbasierte Auffindung von Diensten.
- LDAP (Lightweight Directory Access Protocol): Sicherer (durch Authentifizierung) Zugang für unterschiedliche Internet-Verzeichnisdienste.

Namensdienst

Vorgänge:

- *Adressierung:* Bestimmen einer Position/Zugangsorts
- *Bindung:* Zuordnung Adresse zu Name
- *Auflösung:* Namen in zugehörige Adresse überführen

Abbildung von Namen auf Attribute des entsprechenden Objekts, nicht auf Implementierung des Objekts. Beispiele:

- DNS: Rechnername auf Attribut IP-Adresse abbilden
- CORBA Namensdienst: Abbildung Name auf entfernte Objektreferenz
- X.500: Personenname wird z.B. auf Wohnort, Telefonnummer abgebildet

Namensdienst

Anforderungen an Namensdienste:

- *Skalierbarkeit:* Keine a-priori Einschränkung der maximalen Anzahl aufzunehmender Einträge.
- *Flexibilität:* Möglichkeiten für spätere Veränderungen berücksichtigen, sorge also für lange Lebensdauer.
- *Verfügbarkeit:* Ohne Namensdienste fallen viele verteilte Systeme aus. Stelle daher eine Mindestverfügbarkeit sicher.
- *Fehlerisolation:* Fehler einer Komponente darf andere Komponenten gleicher Funktionalität nicht beeinflussen.
- *Misstrauenstoleranz:* Vermeide Komponenten, denen alle Benutzer des verteilten Systems vertrauen müssen.

Namensräume

Namensraum: Menge gültiger Adressen, die ein Dienst erkennt.

- Syntaktische Definition der internen Namensstruktur.
- Ein Name ist gültig, auch wenn er nicht gebunden ist.
- Namen sollen
  - absolut (ausgehend von der Wurzel)
  - oder relativ (abhängig vom aktuellen Kontext)

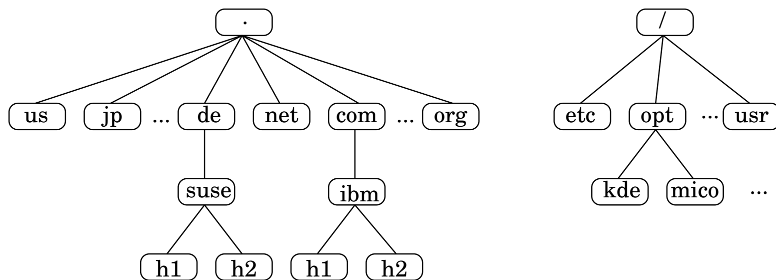
  aufgelöst werden können.

  Beispiele:
  - Datei: `../src/prog.c` (relativ)
  - DNS: `ssh pc03111`, die Endung `hsnr.de` wird, bei entsprechender Konfiguration, automatisch hinzugefügt
- Aliase: Abbilden mehrerer Namen auf gleiche Adresse.

Namensräume

Unterscheidung

- *flacher Namensraum:* Die Länge des Namens ist durch eine maximale Anzahl möglicher Zeichen festgelegt.
- *hierarchischer Namensraum:* Anhängen von Suffixen
  - Dateisystem: `vollständiger Name = Pfad + Name`
  - DNS: `www.hsnr.de`



Namensauflösung

Namensauflösung: einen Namen wiederholt Namenskontexten präsentieren $\rightarrow$ Navigation in Namensdiensten

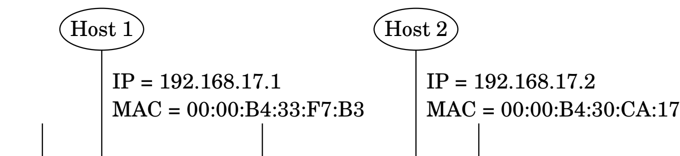
Unterscheidung abhängig von Aufteilung der Verantwortung zwischen verschiedenen Namensdiensten:

- *iterative Navigation:* Der Client fragt eine Reihe von Servern ab.
- *iterative, server-gesteuerte Navigation:* Ein lokaler Server führt eine iterative Navigation durch, koordiniert die Anfragen und liefert das Ergebnis an den Client zurück.
- *rekursive, server-gesteuerte Navigation:* Falls der erste Server den Namen nicht auflösen kann, wird ein übergeordneter Server mit der Auflösung beauftragt. Der Server wird also selbst zum Client.

Namensauflösung: ARP

Ethernet basiert auf ARP (Address Resolution Protocol):

- Jede Netzwerkkarte hat eine weltweit eindeutige, fest eingebrannte Adresse: Media Access Control, MAC
- Zugriff auf Netzwerk wird über MAC-Adresse kontrolliert: Die Knoten werden grundsätzlich mit ihrer MAC-Adresse angesprochen, nicht mit der IP-Adresse.



Beispiele nicht auf IP basierender Netze: Novell NetBIOS, Microsoft NetBEUI, Apple Appletalk, ...

Namensauflösung: ARP

ARP arbeitet in drei Schritten:

- Host 1 schickt eine Nachricht an alle Rechner:
  - eigene IP- und MAC-Adresse
  - IP-Adresse des gesuchten Hosts
- Alle Rechner des Netzwerks speichern die IP- und die MAC-Adresse des Absenders für mögliche, spätere Verbindungen.
- Host 2 sendet die MAC-Adresse seiner Netzwerkkarte.

⇒ Host 1 kennt nun die MAC-Adresse von Host 2 und baut die Verbindung auf.

Die Zuordnungen der MAC- zu den IP-Adressen ist in einer Tabelle gespeichert, die unter Linux mittels `arp` einsehbar ist.

Falls Host 2 nur über einen Router von Host 1 aus erreichbar ist:

- Host 1 erfragt die MAC-Adresse des Routers
- Host 1 übergibt dem Router die Daten
- der Router kümmert sich um die weitere Zustellung

DARPA Internet Domain Name Service

Haupteinsatzzweck von DNS:

- Auflösung von Hostnamen, also Umwandlung eines Rechnernamens in die zugehörige IP-Adresse.
- Mailhost-Position: Welcher Mailserver ist für Empfang bzw. Auslieferung von E-Mails zuständig.

Spezielle Implementierungen verfügen über:

- Umgekehrte Auflösung: Ermitteln des Hostnamens anhand IP-Adresse, z.B. `nslookup`.
- Host-Informationen: Angaben über aktuelle Architektur, Maschinentyp, Betriebssystem, ...

  Vorsicht: Infos können für Angriffe missbraucht werden!

DNS-Abfragen

- Skalierbarkeit von DNS durch Partitionierung, sowie Replikation und Caching der einzelnen Partitionen.
  - → Aufteilung über ein logisches Netzwerk von Servern.
- Aufteilung der DNS-Datenbank in Zonen definiert durch
  - Attributdaten für Namen in einer Domain minus aller Subdomains, die von tiefer liegenden Autoritäten verwaltet werden.
  - Verweise auf mindestens zwei übergeordnete Name-Server.
  - Verweise auf DNS-Server für untergeordnete Domains.
- Replikation: primärer und mindestens ein sekundärer Server
  - Der primäre Server liest die Daten direkt aus einer Datei.
  - Slaves laden Daten regelmäßig vom Master herunter oder werden vom Master bei Änderungen mit Daten versorgt: Zonentransfer

DNS-Auflösung

Die Verteilung und Lokation der Name-Server ist für Clients transparent durch Verwendung von Agenten:

- Agenten werden Auflöser oder Resolver genannt.
- Agenten sind üblicherweise Bestandteil der Standard-bibliotheken.
- Anfragen werden angenommen, entsprechend dem DNS-Protokoll verpackt und versendet.
- Einfaches Request/Reply Protokoll: Mehrere Anfragen können in einem Paket versendet und entsprechend viele Antworten empfangen werden.
- Spezifikation der Vorgehensweise: iterativ, iterativ Server-basiert, rekursiv

DNS: Implementierung

bekannt: BIND (Berkeley Internet Name Domain)

- Server führen `named`-Dämon aus und hören festgelegte Portnummer ab.

  Konfigurationsdatei: `/etc/bind/named.conf`
- Clients binden Resolver aus Standardbibliothek ein:

  `gethostbyname`, `gethostbyaddr`
- Unterstützung von Primär-Server, Sekundär-Server und reine Caching-Server.
- Veraltete Einträge werden nicht erkannt: Ändern sich Namensdaten, liefern Server eventuell noch mehrere Tage später alte Daten.
- Caching unter Linux mittels des Name Service Caching Daemon `nscd`.

  Konfigurationsdatei: `/etc/nscd.conf`

DNS: Implementierung

Jeder Datensatz einer DNS-Datenbank besteht aus folgenden Elementen:

- Domain-name: Name der Domain, auf den sich der Datensatz bezieht.
- Time-to-live: Wie lange soll ein Eintrag gültig sein?

  Dieser Eintrag ist optional und wird oft weggelassen.
- Class: für Internet-Infos steht hier immer `IN`
- Value: Wert des Datensatzes, abhängig vom Typ.
- Type: Um welchen Typ Datensatz handelt es sich?

DNS: Implementierung

Unterstützte Typen:

- `SOA` (Start Of Authority): Verschiedene Parameter der Zone, die der Name-Server verwalten soll.
- `A` (Address): Adresse eines Internet Hosts.
- `MX` (Mail Exchange): Priorität und Name des Mail-Servers der Domain.
- `NS` (Name Server): Name-Server der Domain.
- `CNAME` (Canonical Name): Domain-Name eines Rechners (Aliasfunktion)
- `PTR` (Pointer): Alias für eine numerische IP-Adresse.
- `HINFO` (Host Information): ASCII Beschreibung des Hosts (CPU, OS, ...)
- `TXT` (Text): nicht verwertbarer Text - Kommentar

DNS: Implementierung

DNS-Server: Konfigurationsdatei `/etc/named.conf`

- Angabe des Verzeichnisses, wo weitere Info-Dateien des Name-Servers liegen.
- Definition der Zonen, die der Name-Server bedient.
- Eine Zugriffskontroll-Liste (Access Control List - ACL) kann mit dem `acl`-Befehl oder ähnlichem erstellt werden:

```
acl my_net {
    10.230.1.0/24;
    10.230.4.0/24;
};
```

  Ein so konfigurierter Name-Server ist nur für ein lokales Netz zugreifbar!

  Server, die fremden Rechnern Informationen über interne Adressen anbieten, dürfen keine Einschränkungen haben.

DNS: Implementierung

Zone, die Verbindung zu Root-Servern herstellen kann:

```
zone "." in {
        type hint;
        file "root.hint";
};
```

Zwei Zonen, die grundsätzlich vorhanden sein müssen:

```
zone "localhost" in {
        type master;
        file "localhost.zone";
};
zone "0.0.127.in-addr.arpa" in {
        type master;
        file "127.0.0.zone";
};
```

DNS: Implementierung

Beispiel: DNS-Server für ein Netz `10.230.1.0/24` mit dem Domain-Namen `mydomain.de`

```
zone "mydomain.de" IN {
    type master;
    file "mydomain.de.zone";
};
zone "1.230.10.in-addr.arpa" in {
    type master;
    file "10.230.1.zone";
};
```

Die Zonendatei `mydomain.de.zone`

```
$TTL 86400        ; seconds (1D)
```

Legt die Gültigkeitsdauer dieser Datei fest. Änderungen an dieser Datei werden spätestens nach einem Tag vom Server aktualisiert, wenn er nicht vorher neu gestartet wird.

```
mydomain.de.  IN  SOA  majestix
              hostmaster.majestix.mydomain.de. (
      2010110802 ; Serial: date (yyyymmdd) + serial #
      10800      ; Refresh, seconds (3H)
      1800       ; Retry, seconds (30M)
      604800     ; Expire, seconds (1W)
      259200 )   ; Minimum, seconds (3D)
```

Bei der E-Mail-Adresse des Verwalters ist `@`-Symbol durch einen Punkt ersetzt. `majestix` ist der Name des Name-Servers.

Bedeutung der Einträge:

- `refresh` Zeitangabe in Sekunden, nach der der Slave (Secondary) die Zonendatei vom Master aktualisieren soll. RFC 1912 empfiehlt 1200 bis 43200 Sekunden: Es sollte eine kurze Zeit (1200) gewählt werden, falls die Daten unbeständig (volatile) sind, andernfalls hoch 43200 (12 hours).
- `retry` Zeitangabe in Sekunden, nach der der Slave erneut versuchen soll, die Zonendatei vom Master zu aktualisieren, falls die vorherige Aktualisierung fehlschlug. Typische Werte liegen zwischen 180 (3 min) und 900 (15 min) oder höher.
- `expiry` Zeitangabe in Sekunden, die nur von Slaves genutzt wird. Gibt an, nach welcher Zeit die Zoneninformationen nicht mehr gültig sind, falls alle Aktualisierungsversuche fehlschlugen.

Die Zonendatei `mydomain.de.zone`

Der nächste Eintrag gehört noch zur Domain-Angabe und hat daher keine erneute Nennung eines Namens:

```
IN NS       majestix
IN NS       miraculix
IN MX   20  asterix
IN MX   10  obelix
```

Die beiden Name-Server sind `majestix` und `miraculix`. Da die Namen nicht mit einem Punkt enden, wird der Domänenname automatisch angehängt.

Der Mail-Server (MX) höherer Priorität (20) ist `asterix`, der niedriger Priorität (10) ist `obelix`.

Zum Schluss listen wir noch die einzelnen Rechner auf:

```
localhost   IN  A  127.0.0.1
majestix    IN  A  10.230.1.1
miraculix   IN  A  10.230.1.2
asterix     IN  A  10.230.1.3
obelix      IN  A  10.230.1.4
verleihnix  IN  A  10.230.1.5
automatix   IN  A  10.230.1.6
```

Wir können auch Nick-Names (Aliase) einrichten:

```
loopback    IN  CNAME  localhost
www         IN  CNAME  majestix.mydomain.de.
```

Die Zonendatei `10.230.1.zone`

Dient zur Auflösung von Nummern in Namen und enthält die Informationen aus `mydomain.de.zone` in umgekehrter Reihenfolge.

Die Domäne wird auch in umgekehrter Nummernform angegeben und mit dem Begriff `in-addr.arpa.` abgeschlossen.

Zunächst wieder der TTL und SOA Eintrag:

```
$TTL 1D
1.230.10.in-addr.arpa.  IN  SOA  mydomain.de.
              root.mydomain.de. (
      2010110802 ; Serial: date (yyyymmdd) + serial #
      10800      ; Refresh, seconds (3H)
      1800       ; Retry, seconds (30M)
      604800     ; Expire, seconds (1W)
      259200 )   ; Minimum, seconds (3D)
      IN  NS   majestix.mydomain.de.
```

Die Zonendatei 10.230.1.zone

Es folgen einfach die Nummern und Verweise auf die Namen. Die Nummern sind nur die, die in der obigen Adressangabe fehlen.

```
1    IN  PTR  majestix.mydomain.de.
2    IN  PTR  miraculix.mydomain.de.
3    IN  PTR  asterix.mydomain.de.
4    IN  PTR  obelix.mydomain.de.
5    IN  PTR  verleihnix.mydomain.de.
6    IN  PTR  automatix.mydomain.de.
```

`/etc/nsswitch.conf` Auzug aus der man-page

System Databases and Name Service Switch configuration file.

Various functions in the C Library need to be configured to work correctly in the local environment. Traditionally, this was done by using files (e.g., `/etc/passwd`), but other nameservices (like the Network Information Service (NIS) and the Domain Name Service (DNS)) became popular, and were hacked into the C library, usually with a fixed search order.

The Linux libc5 with NYS support and the GNU C Library 2.x (libc.so.6) contain a cleaner solution of this problem. It is designed after a method used by Sun Microsystems in the C library of Solaris 2. We follow their name and call this scheme "Name Service Switch" (NSS). The sources for the "databases" and their lookup order are specified in the `/etc/nsswitch.conf` file.

aliases	Mail aliases, used by sendmail(8). Presently ignored.
ethers	Ethernet numbers.
group	Groups of users, used by getgrent(3) functions.
hosts	Host names and numbers, used by gethostbyname(3).
netgroup	Network wide list of hosts and users, used for access rules. C libraries before glibc 2.1 only support netgroups over NIS.
networks	Network names and numbers, used by getnetent(3).
passwd	User passwords, used by getpwent(3) functions.
protocols	Network protocols, used by getprotoent(3) functions.
publickey	Public and secret keys for Secure_RPC used by NFS and NIS+.
rpc	Remote procedure call names and numbers, used by getrpcbyname(3) and similar functions.
services	Network services, used by getservent(3) functions.
shadow	Shadow user passwords, used by getspnam(3).

DNS-Client

An example /etc/nsswitch.conf:

```
passwd:         compat
group:          compat
shadow:         compat
hosts:          dns [!UNAVAIL=return] files
networks:       nis [NOTFOUND=return] files
ethers:         nis [NOTFOUND=return] files
protocols:      nis [NOTFOUND=return] files
rpc:            nis [NOTFOUND=return] files
services:       nis [NOTFOUND=return] files
```

- `compat` support '+/−' in the "passwd" and "group" databases. If this is present, it must be the only source for that entry.
- `NOTFOUND=return` sets a policy of "if the user is not-found in nis, don't try files." This treats nis as the authoritative source of information, except when the server is down.

DNS-Client

`resolv.conf` Auzug aus der man-page

The resolver is a set of routines in the C library that provide access to the Internet Domain Name System (DNS). The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of keywords with values that provide various types of resolver information.

```
search kr.hs-niederrhein.de hs-niederrhein.de
nameserver 194.94.120.1
nameserver 194.94.120.2
```

DNS-Client

/etc/host.conf steuert den Resolver für Programme, die gegen die veralteten Bibliotheken libc4 oder libc5 gelinkt sind. Aktuelle Programme nutzen die glibc2 und /etc/nsswitch.conf.

Auflösung des symbolischen Rechnernamens:

- Nachschauen in der Datei /etc/hosts, oder
- Befragen eines Domain Name Servers, oder
- Befragen eines NIS oder NIS+ Servers.

```
order hosts nis bind
multi on
```

- Die order-Zeile bestimmt die Namensauflösung: Zunächst in /etc/hosts nachsehen, anschließend NIS-Server befragen, falls kein Ergebnis erzielt wurde, DNS befragen.
- multi on besagt, dass ein Rechner mehrere IP-Adressen haben kann, wie bspw. Gateways oder Router.

DCE Directory Service

Der DCE Directory Service besteht aus zwei Komponenten:

- *Cell Directory Service (CDS):* Verwaltet Namensraum einer Zelle, jede Zelle besitzt mindestens einen CDS-Server.
- *Global Directory Service (GDS):* Verwaltet globalen Namensraum außerhalb und zwischen den Zellen, verbindet Zellen untereinander.

Namensnotation:

- Weltweite Struktur mit globaler Wurzel /...
- Name einer Zelle wird in X.500-Notation oder in DNS-Notation angegeben.
- Jede Zelle besitzt eigenen, internen Namensraum.

DCE Directory Service

Ein DCE-Name besteht aus drei Teilen:

- *Präfix:* `/...` steht für global, `/.:` steht für lokal.
- *Zellname:* Muss nur bei globalen Namen angegeben werden.
- *lokaler Name:* Identifiziert ein Objekt innerhalb einer Zelle, Notation nach Unix File-Benennungsschema.
- Die einzelnen Teile werden mittels Schrägstrich getrennt.

X.500-Notation:

- hierarchisches, attribut-basiertes Namensschema
- ISO/OSI-Standard
- `Country=DE/OrgType=EDU/OrgName=HSNR/Dept=FB03/...`

Cell Directory Server

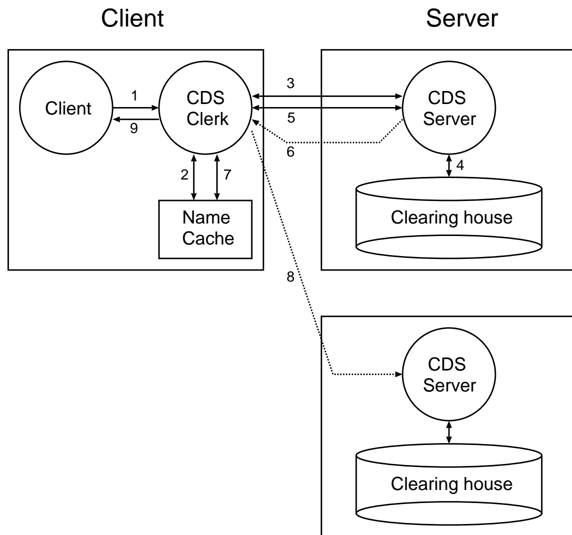
Ein Cell Directory Server verwaltet Verzeichnisse einer Zelle:

- Verzeichniseintrag besteht aus Name und Attributen: Schutzinformationen/Zugriffsrechte, Lokation, Charakteristiken (z.B. bei Drucker: Typ und Geschwindigkeit)
- *Clearing house:*
  - Datenbank, wird vom CDS-Server verwaltet, enthält Verzeichnisse.
  - Ein Server kann mehrere Datenbanken verwalten.
  - Wurzelverzeichnis ist in allen DBs repliziert.
    $\rightarrow$ Suche kann bei irgendeinem CDS-Server begonnen werden.

Cell Directory Server

- *Clerks:*
  - Agent (Transparenz bzgl. replizierter Daten)
  - Läuft auf jeder Maschine, die einen CDS-Server benutzt.
  - Erhält Anfragen eines Clients, befragt einen/iterativ mehrere CDS-Server, stellt Antwort zusammen und gibt sie an Client zurück.
  - Verwendet Cache, der regelmäßig auf Platte geschrieben wird. Der Cache überlebt einen Neustart des Systems/Anwendung.
- *Lookup:* Namensresolution/-auflösung

Cell Directory Server



Global Directory Server

Auffinden eines CDS-Servers in einer anderen Zelle mittels Agententechnologie: *Global Directory Agent* GDA

- GDA kann auf beliebiger Maschine liegen (unabhängig von CDS)
- Erhöhung der Verfügbarkeit und Zuverlässigkeit: jede Zelle kann mehrere GDAs besitzen
- jeder CDS-Server einer Zelle besitzt Information über Lokation des lokalen GDA

Global Directory Server

- CDS-Clerk fragt CDS-Server nach GDA-Adresse
- CDS-Clerk sendet Zellnamen an GDA
- GDA überprüft Notation (DNS oder X.500) und sendet Anfrage an entsprechenden Server weiter
- Diese geben Servernamen zu der gesuchten Zelle an GDA zurück
- GDA leitet die Adresse des CDS-Servers von der Zelle an CDS-Clerk

Übersicht

Dienste

- Dienste unter Linux
- Namensdienst (DNS, DCE Directory Service)
- *Verzeichnisdienst (LDAP)*
- File-Dienst (NFS, DFS)
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- Zeitdienst (NTP)
- Sicherheit (SSL, Kerberos)

LDAP: Lightweight Directory Access Protocol

- Im Prinzip eine hierarchisch strukturierte Datenbank, in der beliebige Informationen abgelegt werden können.
- Die oberste Ebene enthält immer das Element `root`.

  `root` ist die Wurzel des Verzeichnisdienstes, analog zur Wurzel / des Verzeichnisbaums in Unix-File-Systemen.
- Ein fachlicher Zusammenhang zwischen Dateisystem und dem Verzeichnisdienst besteht nicht.
- Strukturierte Speicherung der Informationen. → Schneller Zugriff auf Daten, da nur in einem Teil des Baums gesucht werden muss.
- Organisation des Verzeichnisdienstes und Bearbeiten der enthaltenen Informationen erfolgt über LDAP.

LDAP

Begriffe:

- *Objekt:* Eintrag im Verzeichnisdienst.

  Das Objekt `root` ist grundsätzlich vorhanden und muss nicht explizit angelegt werden. Es beschreibt den Anfang der Struktur.
- *Container:* Objekte, die nur strukturelle Funktion haben. Enthalten keine Informationen, sondern dienen der Gliederung.

  analog: Verzeichnisse in Dateisystemen
- *Blattobjekte:* enthalten die eigentlichen Informationen

  analog: Dateien in Dateisystemen

LDAP

Vordefinierte Container-Objekte:

- root: Wurzel des Baums
- country: beschreibt ein Land, Kürzel `c`, kann nur unterhalb von root erstellt werden
- organization: Ist ein country-Objekt vorhanden, so muss ein Organization-Objekt direkt unterhalb eines Landes stehen, sonst steht organization direkt unter root. Kürzel `o`.
- organizational unit: Organisation kann in verschiedene Units gegliedert sein, Kennzeichnung durch `ou`, kann auch unterhalb von `ou` stehen
- common name: eigentliche Informationen ist in Blattobjekten enthalten, Kürzel `cn`

Beispiel: German Grid

```
O=GermanGrid,OU=FHNiederrhein,CN=host,pc03108.kr.hsnr.de
O=GermanGrid,OU=FHNiederrhein,CN=Peer Ueberholz
```

Benutzer und sogenannte Computing Elemente, also Rechner, die die Rechenleistung dem Grid zur Verfügung stellen, werden auf der Blatt-Ebene `common name` angelegt.

Prinzip der Replizierung:

- Auf beiden Servern (Master und Replica): LDAP-Server installieren und Daten einmalig synchronisieren.
- Konfigurationsdatei `slapd.conf` des Masters: Parameter `replogfile` und `replica` eintragen.
  - Im `replogfile` werden alle Änderungen am Datenbestand protokolliert.
  - Der Master schickt diese Datei selbständig dem Replica zu.
  - Replica führt diese Änderungen auf eigenem Datenbestand nach.
- Parameter `updatedn` in Konfigurationsdatei des Replica-Rechners eintragen.
  - → definiert das Objekt, das Update-Informationen einpflegen kann.

Prinzip der Replizierung: (Fortsetzung)

- Datenkonsistenz: Durch Vergabe von access-Regeln nur lesenden Zugriff erlauben.
- Auf Master den `slurpd`-Dämon starten
  → `replogfile` periodisch zur Replica übertragen

sonstiges:

- LDAP Data Interchange Format (LDIF)
- Kommandos: `ldapadd`, `ldapsearch`, `ldapmodify`
- Client-Konfigurationsdatei: `ldap.conf`

Übersicht

Dienste

- Dienste unter Linux
- Namensdienst (DNS, DCE Directory Service)
- Verzeichnisdienst (LDAP)
- *File-Dienst (NFS, DFS)*
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- Zeitdienst (NTP)
- Sicherheit (SSL, Kerberos)

File-Dienst

Verteilte Dateisysteme unterstützen gemeinsame Nutzung von Dateien im Intranet:

- Zugriff auf Dateien unabhängig von deren tatsächlicher Position
- Grundlage für effiziente Umsetzung zahlreicher Dienste wie Namens-, Druck-, Authentifizierungsdienste, Web-Server, ...
- Beispiele:
  - NFS (Network File System)
  - AFS (Andrew File System)
  - DFS (DCE Distributed File System)

File-Dienst

Anforderungen an verteilte Dateisysteme:

- Der File-Dienst ist oft der am stärksten ausgelastete Dienst im Intranet. Die Funktionalität und Leistung sind daher kritisch für das Gesamtsystem.
- *Zugriffstransparenz:* Keine Unterscheidung zwischen lokalen und entfernten Daten.
- *Ortstransparenz:* Clients sehen einheitlichen Namensraum.
  - → Dateien und Dateigruppen können an andere Position verschoben werden, ohne dass sich ihre Pfadnamen ändern.
- *Mobilitätstransparenz:* Weder Clients noch Administrationstabellen in Clients werden bei einer Datenverschiebung modifiziert.

File-Dienst

Anforderungen an verteilte Dateisysteme: (Fortsetzung)

- *Leistungstransparenz:* Eine Mindestverfügbarkeit auch bei sehr hoher Auslastung sicherstellen.
- *Skalierungstransparenz:* Einfache inkrementelle Erweiterung des Dateidienstes möglich.
- *Offenheit:* Heterogene Hardware und Betriebssysteme werden unterstützt.
- *Nebenläufige Dateiaktualisierung:* Dateiänderungen sollen keine Operationen der anderen Clients stören → Kontrolle für schreibende Zugriffe (z.B. UNIX-Standard `lockd`)

Anforderungen an verteilte Dateisysteme: (Fortsetzung)

- *Fehlertoleranz, Dateireplikation:* Datei wird durch mehrere Kopien an unterschiedlichen Positionen dargestellt. $\rightarrow$ Erhöht die Fehlertoleranz und ermöglicht eine Lastbalancierung.
- *Effizienz:* Verteilter Dateidienst soll bzgl. Leistung, Umfang, Funktionalität und Zuverlässigkeit mit einem lokalen Dateisystem vergleichbar oder besser sein.
- *Sicherheit:* Zugriffskontrollmechanismen wie in lokalen Dateisystemen mit zusätzlichen Forderungen nach
  - Verschlüsselung und Signierung von Anforderungs- und Antwortnachrichten.
  - zuverlässiges Abbilden von Benutzer-ID auf lokale ID.
- *Konsistenz:* Modell für nebenläufigen Dateizugriff so, dass alle Prozesse den gleichen Inhalt sehen, als gäbe es nur eine einzige Datei.

Network File System - NFS

- 1985 von Sun entwickelt.
- NFS-Protokoll ist ein Internet-Standard und definiert in RFC 1813 (v3) bzw. 3010, 3530 (v4).
- Es wird kein Dateisystem im Netz verteilt und es ist daher kein verteiltes Dateisystem im engeren Sinn.
- Zugriffe auf entfernte Dateien werden in Aufträge an Server umgesetzt, d.h. Dateien werden im Netzwerk zur Verfügung gestellt.
- Symmetrische Client/Server-Beziehung: Ein Rechner kann gleichzeitig Client (Dateien importieren) und Server (Dateien exportieren) sein.
- Implementiert mit RPCs.

NFS

- Keine Einschränkung bezüglich der Anzahl importierter oder exportierter Dateisysteme.
- NFSv3: zustandsloser Server
  - Übertragung der Zugangsdaten und Sicherheitsüberprüfung bei jedem Zugriff erforderlich (GID/UID)
  - Verifikation durch Zusatzdienste wie NIS (Network Information Service)

  ⇒ Bei Absturz des Servers gehen keine Informationen verloren, aber Server besitzt keine Informationen über gesetzte Sperren (locks)
- mount: Einbinden eines (Teil-) Dateisystems ins lokale Dateisystem.
  analog: Einbinden eines CDROM-Dateisystems

NFS: mount

- Zuständigkeit bei Mount-Prozess im Benutzerraum:
  - Datei `/etc/exports` bei NFS-Server gibt an, welche lokalen Dateisysteme exportiert werden dürfen.
  - Liste berechtigter Clients für jeden Namen angeben.
- Integrationsmöglichkeiten:
  - einbinden beim Booten (`/etc/fstab`)
  - manuell einbinden mittels `mount`
  - Automounter: Verzeichnisse beim ersten Zugriff (für die Nutzungsdauer) einbinden.

NFS: Protokoll

- NFS-Protokoll verwendet XDR und UDP
- NFS-Server: 15 Operationen (Prozeduren)
  - `lookup(dir, name)` liefert File-Handle und -Attribute für die Datei `name` aus dem durch `dir` spezifizierten Verzeichnis
  - `read(file, offs, cnt)` liest bis zu `cnt` Zeichen aus der Datei `file` beginnend bei `offs`
  - `write(file, offs, cnt, data)` schreibt `cnt` Zeichen in die Datei `file` beginnend bei `offs`
- keine Dateioperationen `open` und `close` (widersprechen Zustandslosigkeit des NFS Dienstes)

NFS: Protokoll

- Sperren trotz Zustandslosigkeit durch zusätzliches Protokoll mit `lockd` (Lock-Dämon)
  - verwaltet Tabellen zur Speicherung von Sperren auf lokalen Dateien
  - auf jeder Maschine läuft ein `lockd`: behandelt Locks auf lokalen Dateien oder sendet die Lock-Operation an den `lockd` eines anderen Rechners
- NFS unterscheidet Lese- und Schreibsperren
- im Fehlerfall kompliziertes Freigeben/Weiterverwenden der Sperren mittels Statusmonitor `statd (stat)`
- Blockgröße bei Transfer: 8K Byte
- NFSv4 ist zustandsspeichernd und wird (hat) NFSv3 ersetzen (ersetzt)

DCE Distributed File System - DFS

- echtes verteiltes Dateisystem
- Blockgröße bei Transfer: 64K Byte
- DFS-Server ist multithreaded und zustandsspeichernd
- DFS-Client verwendet Cache
- DFS bietet Unix-File-Semantik: jede Leseoperation sieht Effekte von vorher durchgeführten Schreiboperationen auf der Datei (realisiert mittels Token-Manager)
- DFS unterstützt File-Replikation

DFS: Token-Manager

Performancesteigerung durch

- verwenden *operationsspezifischer Token* für Open-, Read-, Write-, Lock- und Status-Operationen (Check-File und Update-File)
- *fein granulierte Token:* beschränken Read-, Write- und Lock-Operationen auf kleinen Teil einer Datei

umgehen von Client-Ausfällen:

- Client reagiert nicht auf Token-Entzugsaufforderung: Server wartet 2 Minuten
- nach Ablauf der Frist: Server nimmt an, dass Client ausgefallen ist und verwendet neues Token

File-Replikation

Zur Erhöhung der Zuverlässigkeit, der Verfügbarkeit und des Systemdurchsatzes werden Dateien repliziert. Unterscheide drei Vorgehensweisen:

- *explizite Replikation:* Programmierer legt Kopien an und verwaltet sie.
- *Master/Slave-Server:*
  - Eine Datei wird zunächst auf dem Master angelegt.
  - Der Master kopiert diese Datei von sich aus auf die Slaves.
  - Änderungen von Clients an der Datei erfolgen zunächst auf dem Master, der dann die Änderungen den Slaves mitteilt.

  → Bei Ausfall des Masters muss ein neuer Master gewählt werden.
- *File-Suite:* Um Dateizugriffe an irgendeinen Server schicken zu können, ist ein verteilter Mechanismus erforderlich:
  - Änderungskonflikte müssen aufgelöst werden.
  - Die Konsistenz der Kopien muss sichergestellt sein.

  → Verwende Zeitstempel pro Kopie, und nutze bei Zugriffen nur eine aktuelle Kopie.

File-Replikation

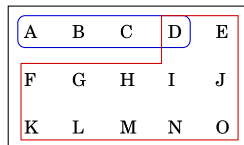
Eine File-Suite erlaubt auch dann noch Dateizugriffe und -änderungen, wenn ein oder mehrere Server ausgefallen sind.

- perfekte Konsistenz der Replikate: Jede Änderung eines Replikaten müsste sofort bei allen anderen Replikaten nachvollzogen werden, was nicht möglich und auch nicht wünschenswert ist.
  - Der Server, der ein Replikat verwaltet, kann abgestürzt sein.
  - Die Signallaufzeit verhindert ein „sofortiges“ Update.
- Daher wird im Allgemeinen eine abgeschwächte Form der Konsistenz erlaubt, bei der zunächst nur einige Kopien aktualisiert werden, und erst im Laufe der Zeit auch die restlichen Kopien aktualisiert werden.

abgeschwächte Konsistenz:

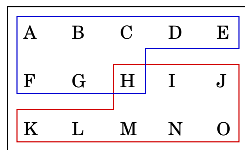
- Das Ergebnis einer Leseoperation muss auf einer aktuellen Kopie basieren, was anhand eines Zeitstempels oder einer Versionsnummer erkannt wird:
  - Ein Server, der eine Dateianfrage erhält, fordert bei anderen Servern die Versionsnummer an.
  - Falls dabei eine höhere Versionsnummer geliefert wird, aktualisiert der Server zunächst seine Kopie und führt dann die entsprechende Dateioperation aus.
  - Wird stets die Mehrheit der Kopien bei einer Schreiboperation aktualisiert, und die Mehrheit der Server bei einem Dateizugriff angefragt, so ist immer mindestens eine Kopie aktuell.
- Die Schreib- und Änderungsoperationen werden nur auf einem Teil der Replikate sofort ausgeführt, die restlichen Kopien werden aktualisiert, wenn der Server verfügbar wird oder wenn es zu wenige aktuelle Replikate gibt.

File-Replikation



Falls häufiger lesend als schreibend auf die Dateien zugegriffen wird, sollte die Anzahl der Versionsnummernanfragen bei Lesezugriffen klein sein.

Je weniger Kopien direkt aktualisiert werden müssen, umso schneller kann eine schreibende Operation durchgeführt werden.



Bevor eine Datei geschrieben werden kann, muss ein Schreib-Quorum stattfinden:

- Werden weniger als N_{write} aktuelle Kopien gefunden, müssen zunächst veraltete Kopien durch aktuelle ersetzt werden.
- Die Schreiboperation wird dann auf N_q viele Replikate angewandt. Die verbleibenden Replikate können danach mit einer Hintergrund-Task geändert werden.

Übersicht

Dienste

- Dienste unter Linux
- Namensdienst (DNS, DCE Directory Service)
- Verzeichnisdienst (LDAP)
- File-Dienst (NFS, DFS)
- *Transaktionsdienst (2-Phasen-Commit Protokoll)*
- Zeitdienst (NTP)
- Sicherheit (SSL, Kerberos)

Transaktionsdienst

zunächst: es gibt nur einen einzigen Server

Greifen mehrere Clients gleichzeitig schreibend auf die gleiche Datei zu, muss wechselseitiger Ausschluss garantiert werden, damit das Ergebnis eindeutig ist.

Fehlermodell für Transaktionen nach B.W. Lampson (1981)

- Ein Algorithmus muss folgende, vorhersehbare Fehler behandeln können:
  - Server-Absturz
  - Beliebige Verzögerungen bei der Nachrichtenübertragung
- Im Katastrophenfall keine Aussage über die Folgen.
  - Schreiben in falschen Block ist Katastrophe
  - Gefälschte Nachrichten sowie nicht erkannte fehlerhafte Nachrichten sind Katastrophe

Anforderungen an Transaktionen:

- *Atomicity:* Alles-oder-Nichts Eigenschaft: Die Transaktion wird entweder vollständig ausgeführt oder hinterlässt keine Wirkung.
  - Eine Überweisung kommt nur zustande, wenn der Betrag bei dem einen Konto abgebucht und auf dem anderen Konto gutgeschrieben wird.
  - Der Flug von Düsseldorf nach Amsterdam und der Flug von Amsterdam nach Seoul und das Hotel in Seoul müssen zusammen gebucht werden.
- *Consistency:* Eine Transaktion überführt das System von einem konsistenten Zustand in einen konsistenten Zustand.
  - Nach einer Überweisung ist die Summe der beiden Kontostände die gleiche wie vor der Überweisung.
  - Bei Datenbanken müssen die referentiellen Integritätsregeln erfüllt sein.

Anforderungen an Transaktionen: (Fortsetzung)

- *Isolation:* Jede Transaktion muss von der Ausführung anderer Transaktionen unabhängig bleiben, also isoliert von anderen ablaufen.
  - Verschiedene Transaktionen können sich nicht überlappen und sind seriell in irgendeiner Reihenfolge auszuführen.
  - Das Ergebnis parallel laufender Transaktionen ist das gleiche, als würden die Transaktionen zeitlich nacheinander ablaufen.

  $\rightarrow$ Serialisierbarkeit
- *Durability:* Die Änderungen einer beendeten und bestätigten (committed) Transaktion können weder verloren gehen noch rückgängig gemacht werden.

Transaktionsdienst

Serielle Ausführung der Transaktionen ist nicht gewünscht, da die Performance des Servers schlecht wäre weil mögliche nebenläufige Ausführungen von Transaktionen nicht genutzt würden.

- Beim Locking werden parallele Transaktionen zugelassen, wenn verschiedene Datenelemente betroffen sind.
  - Zu Beginn wird eine Sperre auf ein Datenelement gesetzt, am Ende wird die Sperre wieder aufgehoben.
  - Ist ein Element gesperrt, darf nur die Transaktion darauf zugreifen, die die Sperre gesetzt hat, alle anderen müssen warten.
- Ohne Locking kann nur die erste Transaktion verbindlich gemacht werden, alle anderen müssen zurückgesetzt werden.
  - Alle Transaktionen greifen auf Datenelemente zu.
  - Am Ende der ersten Phase wird geprüft, ob das gleiche Datenelement von einer anderen Transaktion benutzt wurde.
  - Falls ja: Transaktion abbrechen und erneut starten.

  $\rightarrow$ Optimistische Konkurrenz-Kontrolle

Sperren können zu Verklemmungen (Deadlocks) führen:

- Zwei Transaktionen fordern das gleiche Paar von Sperren an, jedoch in jeder Transaktion in umgekehrter Reihenfolge.
- Daher müssen alle Transaktionen die Sperren zu Beginn der Transaktion anfordern.
  - Gleichzeitiges Halten einer Sperre und Warten auf eine andere Sperre wird verhindert.
  - Die Parallelität der Transaktionen wird eingeschränkt.

Sperren im Vergleich zu optimistischer Konkurrenz-Kontrolle:

- Bei der optimistischen Konkurrenz-Kontrolle können keine Verklemmungen auftreten und die Parallelität ist maximal.
- Der Nachteil ist, dass eine Transaktion oft neu gestartet werden muss und so verhungert.

Transaktionsdienst

Regenerierbarkeit:

- Atomicity und Durability sind gefährdet durch eine fehlerhafte Umgebung und werden erreicht durch die Verwendung wiederherstellbarer Objekte.
- Wenn ein Server während der Abarbeitung einer Transaktion abstürzt und anschließend ein neuer Server gestartet wird, dann muss dieser
  - den alten Zustand der Objekte wieder laden können
  - und seine Operationen erneut ausführen können.

  $\rightarrow$ Log-Datei ist notwendig, die alle Operationen enthält.
- Wenn die Transaktion abgeschlossen ist, muss das Objekt den neuen Zustand repräsentieren und abgespeichert werden.

Transaktionsdienst

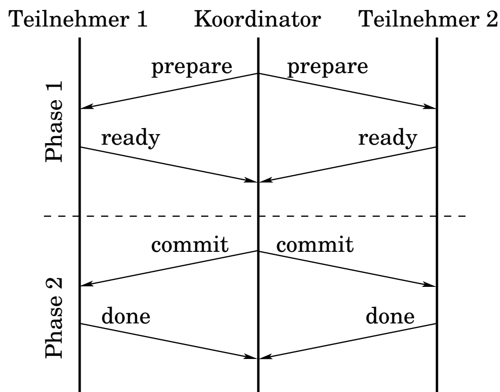
jetzt: Verteilte Transaktionen bei Zugriff auf Objekte, die auf verschiedenen Servern liegen.

Zwei-Phasen-Commit Protokoll von J. Gray (1978)

- Aufgabe: Atomicity der verteilten Transaktion herstellen
- Anforderungen:
  - Alle Knoten, die eine Entscheidung treffen, treffen dieselbe Entscheidung.
  - Ein Knoten kann seine Entscheidung nicht nachträglich ändern.
  - Die Entscheidung, eine Transaktion zu bestätigen, kann nur von allen Knoten einstimmig getroffen werden.
  - Falls keine Ausfälle vorliegen und alle Knoten zugestimmt haben, wird die Transaktion festgeschrieben.
  - Nach Behebung eventueller Ausfälle muss eine Entscheidung getroffen werden.

Zwei-Phasen-Commit Protokoll

- Der die Transaktion auslösende Knoten übernimmt die Rolle des Koordinators, alle anderen sind Teilnehmer.
- Jeder Knoten unterhält eine spezielle Log-Datei, in die alle relevanten Ereignisse geschrieben werden.
- Das Protokoll besteht aus zwei Phasen mit jeweils zwei Schritten:
  - Phase 1: Abstimmungsphase (prepare to commit)
    - Aufforderung zur Stimmabgabe
    - Stimmabgabe
  - Phase 2: Abschluss (commit/rollback)
    - Mitteilung über die Entscheidung
    - Bestätigung der Entscheidung (acknowledge)



Wie wird auf den Ausfall eines Teilnehmers reagiert, wenn dieser bereits erfolgreich eine Ready-Nachricht versendet hat und alle weiteren Teilnehmer sowie der Koordinator ebenfalls mit Ready geantwortet hat?

Übersicht

Dienste

- Dienste unter Linux
- Namensdienst (DNS, DCE Directory Service)
- Verzeichnisdienst (LDAP)
- File-Dienst (NFS, DFS)
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- *Zeitdienst (NTP)*
- Sicherheit (SSL, Kerberos)

Physikalische Uhren

Die Sonne: Es gab einmal die Idee, die Zeit anhand des Sonnendurchlaufs zu bestimmen:

- Sonnentag: Intervall zwischen zwei aufeinanderfolgenden Zeitpunkten, an denen die Sonne jeweils ihren höchsten Stand erreicht hatte.
- Ein Sonnentag wurde in 24 Stunden, eine Stunde in 60 Minuten und eine Minute in 60 Sekunden unterteilt.

$\rightarrow$ Sonnensekunde: Sonnentag / 86.400

Die Idee ist gut, aber in der Praxis gibt es, wie immer, ein Problem: Die Rotationsgeschwindigkeit der Erde ist nicht konstant.

- langfristig: Rotation wird durch Reibung gebremst.
- kurzfristig: Turbulenzen im flüssigen Erdkern.

Punkt zwei wird durch Mittelwertbildung über eine große Anzahl von Tagen abgeschwächt. Dies führt zur mittleren Sonnensekunde.

Physikalische Uhren

Atomzeit: Heute reicht eine solch unpräzise Zeitmessung nicht mehr aus und man legt eine Sekunde fest als die Zeit, die ein Cäsium-133-Atom für 9.192.631.770 Zustandsübergänge benötigt.

- Definiert durch Bureau International de l'Heure (BIH) in Paris.
- Im Jahr 1958 entsprach die so festgelegte Atomsekunde genau der mittleren Sonnensekunde.
- Der Mittelwert über eine größere Anzahl von Atomuhren führt zur Time Atomic International TAI.

Da die Rotationsgeschwindigkeit der Erde durch Reibungsverluste stetig geringer wird, driften Atomzeit und Sonnenzeit auseinander.

- Zur Zeit ist ein TAI-Tag um 3 ms kürzer als ein Sonnentag.
- Es wird eine Schaltsekunde eingefügt, wenn die Differenz zwischen TAI und Sonnenzeit auf 800 ms angewachsen ist.

$\rightarrow$ Universal Time Coordinated UTC

Physikalische Uhren

UTC:

- Seit 1967 werden Schaltsekunden berücksichtigt.
- Internationale Basis für die Zeitmessung.
- UTC-Signale werden von Satelliten und Rundfunksendern ausgestrahlt.
- Genauigkeit:
  - Rundfunk: ± 10 ms
  - Geostationary Environmental Operational Satellite: $\pm 0,1$ ms
  - Global Positioning System: ± 1 ms
- Durch Mittelwertbildung bei Verwendung mehrerer UTC-Quellen kann die Genauigkeit erhöht werden.

Zeitdienst

Ziel: Maximale Abweichung der Uhren aller Rechner minimieren.

oft: Ein Rechner R besitzt einen Funkempfänger für das UTC-Signal. $\rightarrow$ Synchronisiere alle anderen Rechner mit R.

erster Versuch: (Synchronisation am UTC-Rechner)

- Sei p die maximale Abweichung der Uhren. Diese wird vom Hersteller angegeben.
- Zwei Uhren können nach Zeit t um $2p \cdot t$ Sekunden abweichen.

$\rightarrow$ Uhren alle $d/2p$ Sekunden synchronisieren

- Jeder Client erfragt regelmäßig beim Zeit-Server die aktuelle Zeit und setzt seine Uhr entsprechend.

Zeitdienst

Probleme:

- Zeit darf niemals rückwärts laufen: Eindeutigkeit geht verloren!
- Nachrichtenlaufzeit der Antwort verfälscht Synchronisation.

Lösung:

- zum ersten Fehler: Zu schnelle Uhren verlangsamen, indem die vom Timer ausgelöste Unterbrechungsroutine weniger Millisekunden addiert als üblich, bis Angleichung vollzogen ist.
- zum zweiten Fehler: Messen der Verzögerungszeit und schätzen der Dauer der Unterbrechungsbehandlung $\rightarrow$ einfache Übertragungszeit ist

$$T = (T_1 - T_0 - T_{interrupt})/2$$

anderer Ansatz (BSD-Unix):

- Zeit-Server fragt periodisch die Rechner nach ihrer Zeit
- aus den Antworten berechnet er eine mittlere Zeit
- Server teilt Rechnern neue Zeit mit: Rechner setzen ihre Uhr auf die neue Zeit oder verlangsamen ihre Uhr

verteilter Algorithmus:

- Zeit einteilen in Synchronisationsintervalle fester Länge
- zu Beginn eines Intervalls: lokale Zeit an alle anderen Rechner schicken
- neue Zeit berechnen aus allen, innerhalb eines gegebenen Zeitintervalls eintreffenden Nachrichten

NTP: Network Time Protocol

Eigenschaften:

- präzise Synchronisation eines Clients mit UTC trotz großer/variabler Nachrichtenverzögerungen bei Kommunikation in großen Netzen (Internet)
- redundante Server und Pfade ermöglichen zuverlässigen Dienst (übersteht Verbindungsunterbrechungen), RFC 1305/2030.
- Rekonfiguration der Server bei Ausfall eines Servers
- Auslegung auf große Anzahl Clients und Server: häufige Synchronisation $\rightarrow$ ausreichende Anpassung der Abweichgeschwindigkeit
- Authentifizierungsschutz vor Manipulation

NTP

Architektur:

- Zeit-Server im hierarchischen baumartigen Subnetz
- primäre Server in Wurzelschicht besitzen UTC-Empfänger
- sekundäre Server in Schicht 2 werden direkt vom primären Server synchronisiert
- Genauigkeit nimmt zu den Blättern hin ab
- NTP berücksichtigt Gesamtverzögerungen: bewerte Qualität der Zeitdaten auf den Servern
- Infos unter `http://www.ntp.org`

NTP

Synchronisationsmöglichkeiten:

- Multicast: Verwendung bei schnellen lokalen Netzen
  - Server schickt periodisch aktuelle Zeit per Multicast an LAN-Rechner
  - geringe Genauigkeit
- Procedure-Call-Modus:
  - Server beantwortet Zeitanfragen mit Zeitstempel
  - mittlere Genauigkeit
- symmetrisch: Synchronisation zwischen Zeitservern
  - wechselseitiger Austausch von Zeitstempeln
  - hohe Genauigkeit

Übersicht

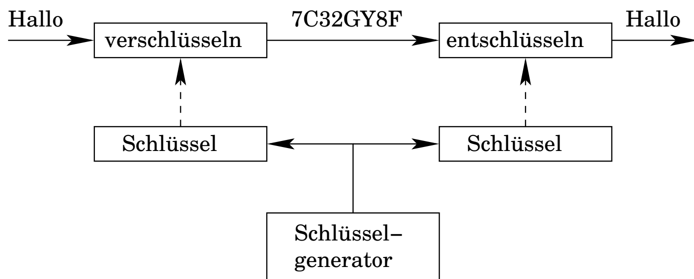
Dienste

- Dienste unter Linux
- Namensdienst (DNS, DCE Directory Service)
- Verzeichnisdienst (LDAP)
- File-Dienst (NFS, DFS)
- Transaktionsdienst (2-Phasen-Commit Protokoll)
- Zeitdienst (NTP)
- *Sicherheit (SSL, Kerberos)*

Schutzziele

- *Vertraulichkeit:* Unberechtigte Teilnehmer (Personen oder andere Systeme) dürfen keinen Zugriff auf Daten oder Nachrichteninhalte haben. → Verschlüsselung
- *Integrität:* Übertragene oder gespeicherte Informationen dürfen von Unberechtigten nicht verändert werden können. Zumindest muss eine solche Änderung zuverlässig erkannt und ggf. korrigiert werden. → digitale Signatur
- *Verfügbarkeit:* Dienste, Daten und Kommunikation stehen zu dem Zeitpunkt, zu dem sie benötigt werden, auch zur Verfügung. Denial-of-Service Attacken müssen verhindert werden. → Firewalls
- *Authentizität:* Die Echtheit einer Nachricht und die Identität eines Nachrichtenabsenders oder Dienstnutzers muss überprüfbar sein. → Authentisierung
- *Zugriffsschutz:* Unberechtigte Zugriffe auf Ressourcen müssen verhindert werden. → Autorisierung

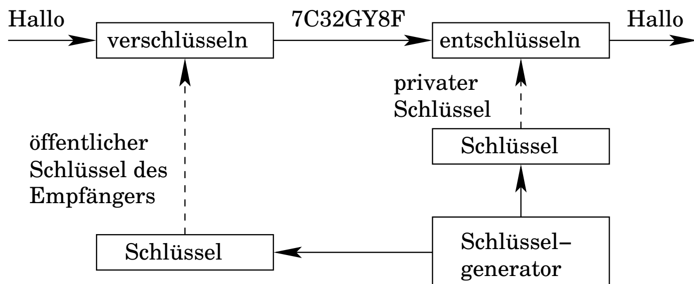
Zum Ver- und Entschlüsseln wird der gleiche Schlüssel verwendet:



- Pro: Sehr schnelles Verfahren (Beispiele: 3DES, RC5, IDEA)
- Contra: Der Schlüssel muss auf einem gesonderten, sicheren Weg ausgetauscht werden.
- Contra: Keine digitalen Signaturen möglich, da immer mindestens zwei Personen den Schlüssel kennen und somit keine eindeutige Zuordnung möglich ist.

Asymmetrische Kryptoverfahren

Das Ver- und Entschlüsseln nutzt unterschiedliche Schlüssel:



- Damit nur der Empfänger die Nachricht lesen kann, wird mit dem öffentlichen Schlüssel des Empfängers chiffriert. Nur der echte Empfänger kennt den privaten Schlüssel.
- Contra: Sehr langsames Verfahren
- Contra: Die öffentlichen Schlüssel müssen so verwaltet werden, dass Angreifer diese nicht manipulieren können. → Zertifikate

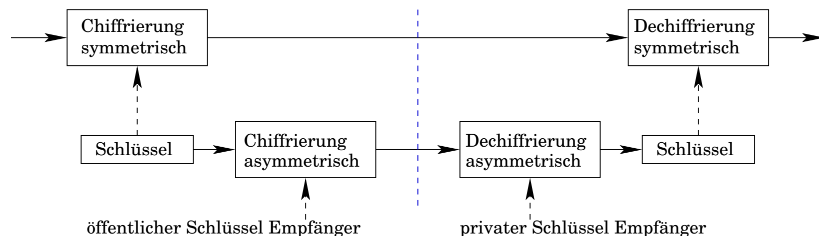
Zertifikate

Ein Angreifer könnte einen öffentlichen Schlüssel unter falschen Namen verteilen und so an Informationen kommen, die nicht für ihn bestimmt sind:

- Man-in-the-middle: Gutgläubige Teilnehmer nutzen den falschen Schlüssel, um geheime Informationen wie eine TAN an den vermeintlichen Partner zu schicken.
- Lösung: Trust Center zertifizieren den öffentlichen Schlüssel eines Benutzers durch eine digitale Unterschrift unter dem Paar bestehend aus öffentlichem Schlüssel und Benutzername.
- Die Zertifizierungsstelle muss selbst vertrauenswürdig sein oder durch eine andere, übergeordnete Zertifizierungsstelle als vertrauenswürdig eingestuft sein.
- Web-Browser verwalten eine Reihe von Zertifikaten solcher Trust Center. Solange die Browser vor Modifikationen und falschen Updates geschützt sind, ist das Verfahren sicher.

Kombinierte Kryptoverfahren

Um die Vorteile der symmetrischen und asymmetrischen Verfahren zu verknüpfen, bildet man kombinierte Verfahren.



- Zunächst erstellt der Absender einen Meldungsschlüssel, der nur für diese eine Nachricht verwendet wird.
- Der Meldungsschlüssel wird mit dem öffentlichen Schlüssel des Empfängers chiffriert, so dass nur der gewünschte Empfänger die Nachricht (den Meldungsschlüssel) entschlüsseln kann.

Kombinierte Kryptoverfahren

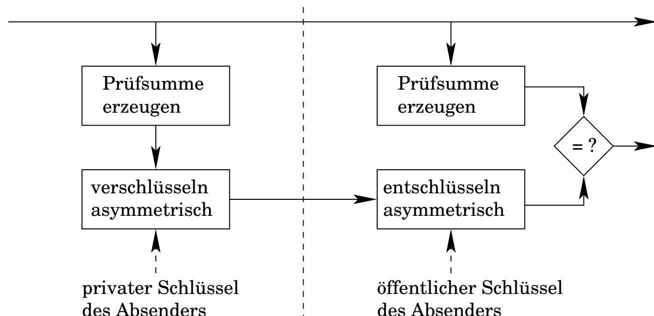
- Anschließend wird die eigentliche Nachricht mit dem, nun beiden Teilnehmern bekannten, Meldungsschlüssel chiffriert.
- Das Verfahren ist schnell, da nur eine sehr kurze Nachricht (der Meldungsschlüssel) mit dem aufwändigen asymmetrischen Verfahren verschlüsselt wird. Die eigentliche Nachricht wird mit dem effizienten symmetrischen Verfahren verschlüsselt.

Diese Verfahren sind in der Praxis weit verbreitet:

- PGP (Pretty Good Privacy)
- SSL (Secure Socket Layer)
- TLS (Transport Layer Security)

Digitale Signatur

Um Änderungen an der Nachricht zu erkennen, wird mittels einer Hash-Funktion zunächst eine Prüfsumme berechnet:



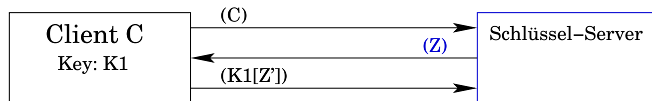
- Die Hash-Funktion ist sehr einfach zu berechnen, aber eine Nachricht zu generieren, die den gleichen Hash-Wert erzeugen würde, ist nur mit extrem hohen Rechenaufwand möglich.
- Die Prüfsumme ist relativ kurz: SHA-256, SHA-384, SHA-512

Digitale Signatur

- Die Nachricht und die chiffrierte Prüfsumme werden an den Empfänger übertragen.
- Wird die Prüfsumme mit dem privaten Schlüssel des Absenders chiffriert, dann kann der Empfänger prüfen, ob die Nachricht tatsächlich vom echten Absender versendet wurde, denn nur der kennt den privaten Schlüssel.
- Um zu prüfen, ob die Nachricht verändert wurde, wird die Prüfsumme zur empfangenen Nachricht gebildet und mit der dechiffrierten Prüfsumme, die mitgeschickt wurde, verglichen.
- Ein Angreifer kann zwar beide Teile, also Nachricht und chiffrierte Prüfsumme, abfangen und durch neue ersetzen. Aber er kann die digitale Signatur nicht erstellen, da nur der echte Absender den privaten Schlüssel kennt.

Authentifizierung

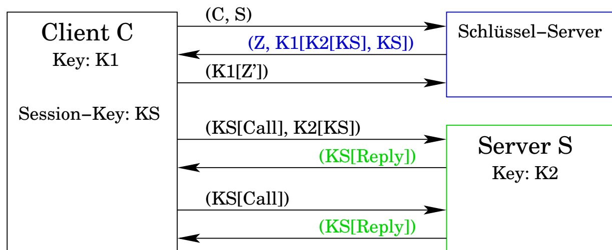
Wie können wir die Identität eines Partners feststellen? Wie können wir sicher sein, dass er der ist, für den er sich ausgibt?



- Der Client C besitzt einen geheimen Schlüssel K1, der auch dem Schlüssel-Server bekannt ist. Der Schlüssel-Server muss vertrauenswürdig sein.
- Der Client startet die Authentifizierung und schickt C.
- Der Schlüssel-Server generiert eine zufällige Zahl Z.
- Client C modifiziert Z nach bekanntem Verfahren und schickt die modifizierte Zahl Z' chiffriert mit Schlüssel K1 zurück.
- Der Schlüssel-Server dechiffriert die Nachricht und prüft, ob Z korrekt modifiziert wurde. Falls ja, ist Client C als authentisch eingestuft, denn nur Client C kennt den Schlüssel K1.

Authentifizierung

Zusätzlich kann der Schlüssel-Server einen Session-Key für den Zugriff auf einen Server generieren und mitschicken:



- Client C fragt nach Session-Key für Zugriff auf Server S.
- Die Authentifizierung läuft wie oben beschrieben, aber der Client erhält gleichzeitig einen chiffrierten Session-Key.
- Anfragen an Server S chiffriert Client C mit dem Session-Key.
- Bei der ersten Anfrage muss der Session-Key auch dem Server mitgeteilt werden.

Authentifizierung

Damit der Session-Key nicht verfälscht werden kann, wird er direkt vom Schlüssel-Server mit K2 chiffriert. K2 ist nur dem Server S und dem Schlüssel-Server bekannt.

- Ein Angreifer kann den Session-Key nicht selber erzeugen und sich für C ausgeben, da er den Schlüssel K2 nicht kennt, mit dem Server S den chiffrierten Session-Key entschlüsselt.
- Der Schlüssel-Server könnte den Session-Key direkt an Server S schicken. Aber aufgrund unterschiedlicher Laufzeiten der Nachrichten könnte dann der Client bereits eine Anfrage beim Server S stellen, bevor der Session-Key bei S eingetroffen ist.

Dieses Authentifizierungsprotokoll wurde von Needham und Schroeder entwickelt und wurde im System Kerberos für Unix-, Linux- und Windows-Betriebssysteme implementiert.