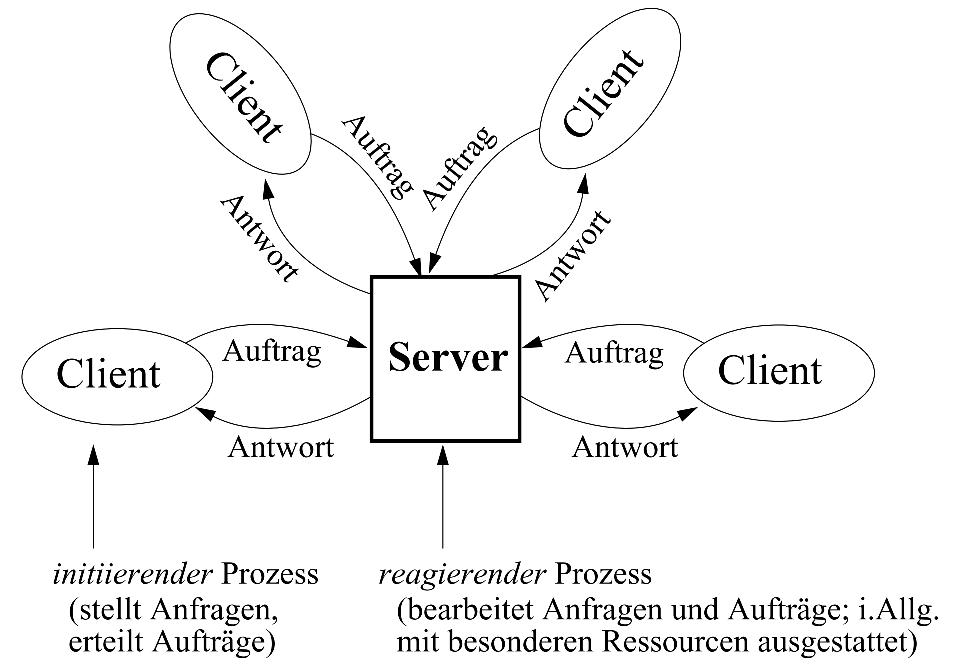
Client/Server-Modell

Das Client/Server-Modell



- Aufgabenteilung und asymmetrische Struktur
  - *Clients*: typischerweise Anwendungsprogramme und Benutzungsschnittstelle (“front end”) für einen Nutzer
  - *Server*: zuständig für Dienstleistungen für viele Clients
- Typisches Kommunikationsparadigma: RPC

Eignung des Client/Server-Paradigmas

- Populär wegen des eingängigen Modells
  - entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
  - gewohntes Muster → intuitive Struktur, gute Überschaubarkeit

- Dienstleistungsangebot im E-Commerce etc. durch Server
  - Clients in Form von ubiquitären Web-Browsern

- Effizienz durch spezialisierte Hard-/Software
  - grosszügige Ausstattung (CPU-Leistung, Speicherkapazität usw.)
  - bestückt mit spezieller Software (z.B. Datenbank)

- Kosteneffektivität durch bessere Auslastung wertvoller Ressourcen (z.B. bei "Compute Server")
  - Clients brauchen oft nur kurzfristig Spitzenleistung
  - einzelner Client kann Ressourcen aber nicht dauerhaft auslasten

- Passend für viele Kooperationsbeziehungen, z.B.
  - Client erbittet Auskunft von einem spezialisierten Service
  - gefährdete Clients geben wertvolle Daten in Obhut des (gegen Missbrauch, Verlust, Diebstahl usw.) hoch gesicherten Servers

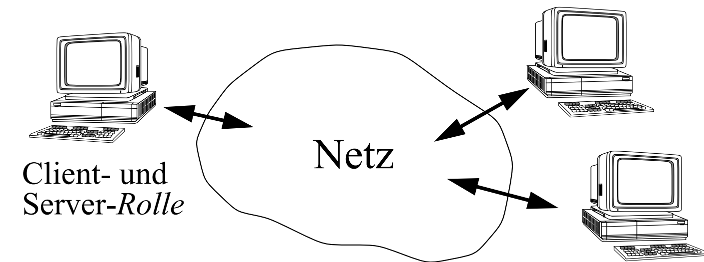
---

- Modell ist jedoch nicht für alle Zwecke geeignet, z.B.:
  - Pipelines
  - Asynchrone Mitteilung
  - Peer-to-peer

Peer-to-Peer-Strukturen

"Gleichrangiger"

- Im Gegensatz zum asymmetrischen Client/Server-Modell



- Ein Client fungiert zugleich als Server für seine Partner
  → keine (teuren) dedizierten Server notwendig

- In der Idealform keine zentralisierten Elemente
  - dies wird gelegentlich in "politischer" Weise artikuliert (vgl. Tauschbörsen)

- *Nachteile*:
  - "Anarchischer" als *maschinenbezogene* Client/Server-Architektur
  - Computer müssen leistungsfähig genug sein (cpu-Leistung, Speicher-ausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Besitzer kann seine Maschine ausschalten...)
  - Datensicherung i.Allg. problematischer als bei zentralen Servern
  - Sicherheit und Schutz kritisch: Lizenzen, Viren, Integrität,...

Zu vielen Aspekten von Peer-to-Peer-Systemen: R. Steinmetz, K. Wehrle (Eds): *Peer-to-Peer Systems and Applications*, Springer-Verlag, 2005

Zustandsändernde /-invariante Dienste

- Verändern Aufträge den Zustand des Servers wesentlich?

- Typische *zustandsinvariante* Dienste:
  - Auskunftsdienste (z.B. Name-Service)
  - Zeitservice

- Typische *zustandsändernde* Dienste:
  - Datei-Server

Idempotente Dienste / Aufträge

- Wiederholung eines Auftrags liefert gleiches Ergebnis

  - Beispiel: "Schreibe in Position 317 von Datei XYZ den Wert W" [nicht zustandsinvariant!]
  - Gegenbeispiel: "Schreibe ans Ende der Datei XYZ den Wert W"
  - Gegenbeispiel: "Wie spät ist es?" ← [aber zustandsinvariant!]

Wiederholbarkeit von Aufträgen

- Bei Idempotenz oder Zustandsinvarianz kann bei Verlust des Auftrags (timeout beim Client) dieser erneut abgesetzt werden (→ einfache Fehlertoleranz)
  - vgl. auch frühere Diskussion bzgl. RPC-Fehlersemantik!

Zustandslose / -behaftete Server

[Zustandslose: stateless] [behaftete: statefull]

- Hält der Server Zustandsinformation über Aufträge hinweg? ["session"]
  - z.B. (Protokoll)zustand des Clients
  - z.B. Information über frühere damit zusammenhängende (Teil)aufträge

- Aufträge an zustandslose Server müssen autonom sein

- *Beispiel*: Datei-Server

```
open("XYZ");
read;
read;
close;
```

In klassischen Systemen hält sich das Betriebssystem Zustandsinformation, z.B. über die Position des Dateizeigers geöffneter Dateien

  - bei zustandslosen Servern entfällt open/close; jeder Auftrag muss vollständig beschrieben sein (Position des Dateizeigers etc.) ← [notw. Zustandsinformation ist beim Client]
  - zustandsbehaftete Server daher i.Allg. effizienter
  - Dateisperren sind bei echten zustandslosen Servern nicht (einfach) möglich
  - zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. durch Speichern von Sequenznummern) → Idempotenz

- *Crash* eines Servers: Weniger Probleme im zustandslosen Fall (→ Fehlertoleranz)! ← [entscheidender Vorteil!]

- Datei-Server wurden sowohl schon zustandslos (z.B. NFS) als auch zustandsbehaftet (z.B. RFS) realisiert

Sind Webserver zustandslos?

- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - jeder link, den man anklickt, löst eine neue "Transaktion" aus

- Stellt ein Problem beim E-Commerce dar
  - gewünscht sind Transaktionen über mehrere Klicks hinweg und
  - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
  - erforderlich z.B. für Realisierung von "Warenkörben" von Kunden
  - gewünscht vom Marketing (Verhaltensanalyse von Kunden)

Lösungsmöglichkeiten (z.B. zur Realisierung von "Warenkörben" im WWW):

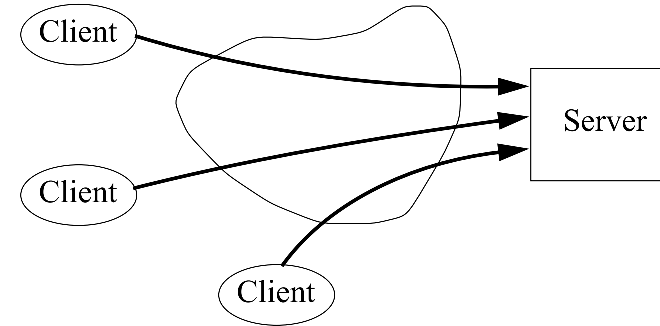
- IP-Adresse des Kunden an Auftrag anheften?
  - Problem: Proxy-Server → mehrere Kunden haben gleiche IP-Adresse
  - Problem: dynamische IP-Adressen → keine Langzeitwiedererkennung

- "URL rewriting" und dynamische Web-Seiten
  - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig aufruft
  - diese Nummer jedem link der Seite anheften und mit zurückübertragen

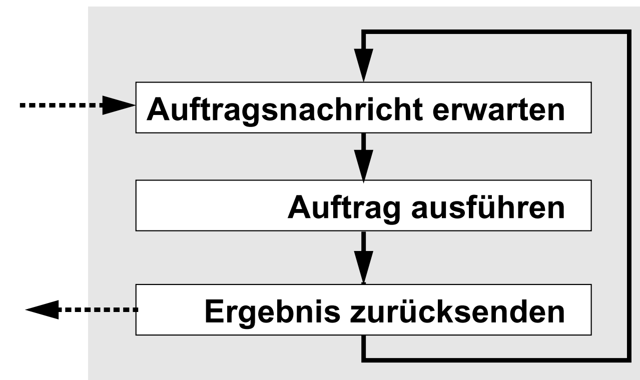
- Cookies
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - der Sender des Cookies kann dieses später wieder lesen und damit den Kunden wiedererkennen

Gleichzeitige Server-Aufträge?

- Problem: Oft viele "gleichzeitige" Aufträge



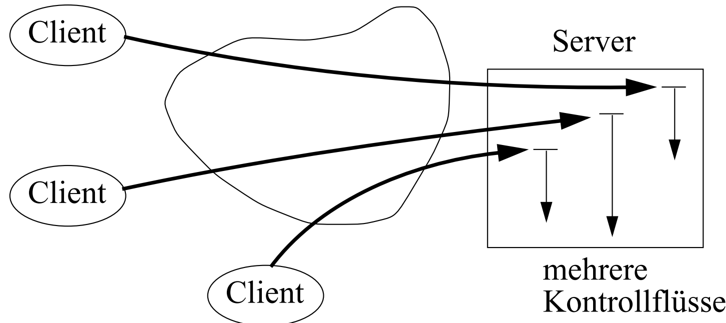
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- "single threaded" (nur ein einziger Kontrollfluss)
- eintreffende Anfragen während Auftragsbearbeitung: abweisen, puffern oder schlichtweg ignorieren
- einfach zu realisieren
- bei trivialen Diensten mit kurzer Bearbeitungszeit sinnvoll

Konkurrente (“nebenläufige”) Server

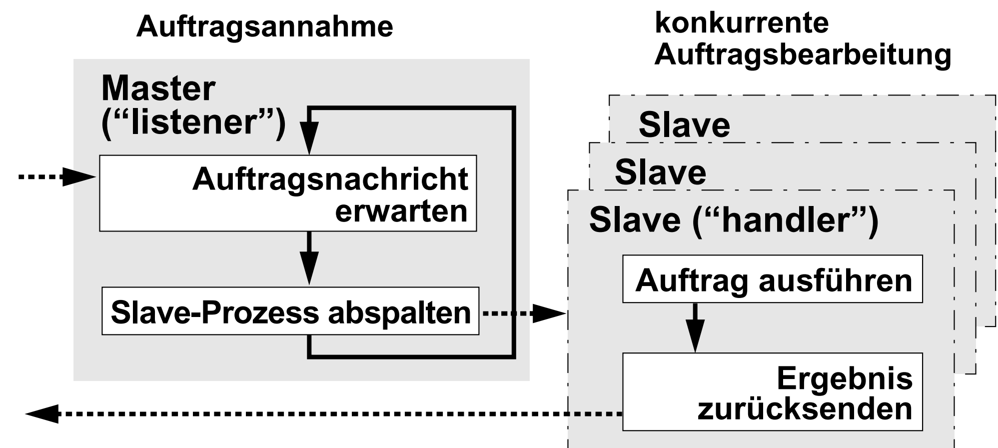
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei längeren Aufträgen



- Ideal bei physischer Parallelität (z.B. multicore)
  - aber auch bei Monoprozessor-Systemen (vgl. Argumente bei Timesharing-Systemen): Nutzung erzwungener Wartezeiten eines Auftrags für andere Jobs; kürzere mittlere Antwortzeiten bei Jobmix aus langen und kurzen Aufträgen
- Interne Synchronisation bei konkurrenten Aktivitäten sowie evtl. Lastbalancierung beachten
- Verschiedene denkbare Realisierungen, z.B.
  - mehrere Prozessoren bzw. Multicore-Prozessoren
  - Verbund verschiedener Server-Maschinen (Server-Farm, -Cluster)
  - feste Anzahl vorgegründeter Prozesse oder dynamische Prozesse

Konkurrente Server mit dynamischen Handler-Prozessen

- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave (“handler”) übernimmt den Auftrag
  - Client kommuniziert dann direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind oft Leichtgewichtsprozesse (“threads”)
  - Slaves terminieren i.Allg. nach Beendigung des Auftrags
  - die Anzahl gleichzeitiger Slaves sollte begrenzt werden



- Alternative: “Process preallocation”: Feste Anzahl statischer Slave-Prozesse
  - u.U. effizienter (u.a. Wegfall der Erzeugungskosten)
- Übungsaufgaben:
  - herausfinden, wie es bei Web-Servern konkret gemacht wird
  - wie sollte man bei Internet-Suchmaschinen vorgehen?

Master/Slave

Subject: Identification of equipment sold to LA County
Date: Tue, 18 Nov 2003 14:21:16 -0800
From: "Los Angeles County"

The County of Los Angeles actively promotes and is committed to ensure a work environment that is free from any discriminatory influence be it actual or perceived. As such, it is the County's expectation that our manufacturers, suppliers and contractors make a concentrated effort to ensure that any equipment, supplies or services that are provided to County departments do not possess or portray an image that may be construed as offensive or defamatory in nature.

One such recent example included the manufacturer's labeling of equipment where the words "Master/Slave" appeared to identify the primary and secondary sources. Based on the cultural diversity and sensitivity of Los Angeles County, this is not an acceptable identification label.

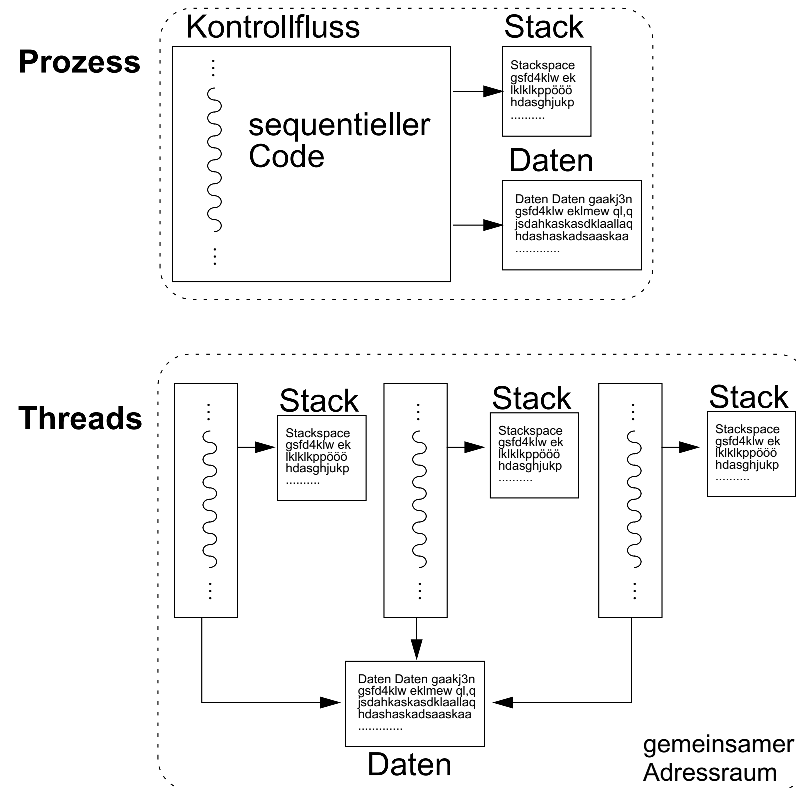
We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment or components thereof that could be interpreted as discriminatory or offensive in nature before such equipment is sold or otherwise provided to any County department.

Thank you in advance for your cooperation and assistance.

Joe Sandoval, Division Manager
Purchasing and Contract Services
Internal Services Department
County of Los Angeles

Threads

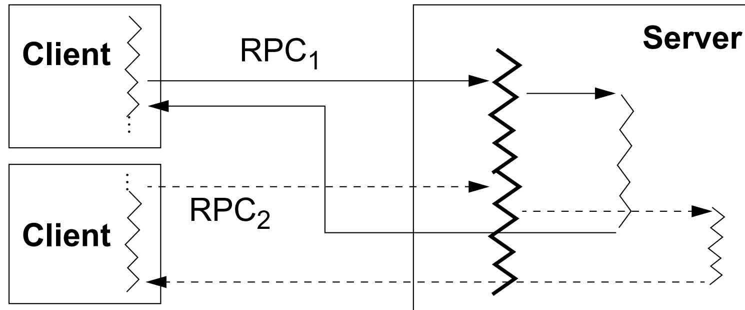
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz; Synchronisation bzgl. gem. Speicher
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel daher i.Allg. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look Aside Buffer (TLB) bleiben erhalten
  - kein aufwändiger Wechsel in / über privilegierten Modus

Wozu Multithreading bei Client-Server-Middleware?

- *Server*: quasiparallele Bearbeitung von Aufträgen
  - Server bleibt ständig empfangsbereit



- *Client*: Möglichkeit zum „asynchronen RPC“
  - Hauptkontrollfluss delegiert RPCs an nebenläufige Threads
  - keine Blockade durch Aufrufe im Hauptfluss
  - echte Parallelität von Client (Hauptkontrollfluss) und Server

Problematik von Threads

- Fehlender gegenseitiger Adressraumschutz
  → schwierig zu findende Fehler
- Stackgrösse muss bei Gründung i.Allg. statisch festgelegt werden
  → unkalkulierbares Verhalten bei Überschreitung
- Von asynchronen Meldungen (“Signale”, “Interrupts”) an den Prozess soll i.Allg. nur ein einziger (der “richtige”) Thread betroffen werden
- Schwierige Synchronisation → Deadlockgefahr

Aufrufe des Betriebssystem-Kerns können problematisch sein, wenn diese nicht dafür geeignet (“thread safe”) sind:

a) *nicht ablaufinvariante* (“non-reentrant”) Systemroutinen

- interne Statusinformation, die ausserhalb des Stacks der Routine gehalten wird, kann bei paralleler Verwendung überschrieben werden
- z.B. *printf*: ruft intern Speichergenerierungsroutine auf; diese benutzt prozesslokale Freispeicherliste, deren “gleichzeitige” nicht-atomare Manipulation zu Fehlverhalten führt
- “Lösung”: Verwendung von “Wrapper-Routinen”, die gefährdete Routinen kapseln und Aufrufe wechselseitig ausschliessen

b) *blockierende* (“synchrone”) Systemroutinen

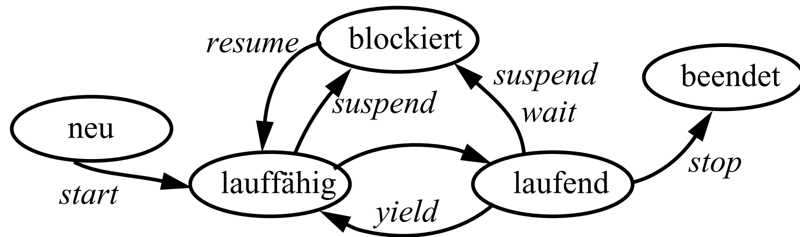
- z.B. synchrone E/A, die *alle* Threads eines Prozesses blockieren würde (statt nur den einen aufrufenden Thread)

Exkurs: Threads in Java

- Hier nur ein *Überblick*; zu weiteren Aspekten vgl. die Dokumentation (online bzw. in Büchern)

- *Konstruktor*:
  - **public** Thread()

- *Methoden*:
  - **void** start()
  - **void** suspend()
  - **void** stop()
  - **void** resume()
  - **void** wait()
  - **static void** yield()



- **static void** sleep(**long** millis) // blockiert für eine gewisse Zeit
- **void** join() // Synchronisation zweier Threads
- **void** setPriority(**int** prio)
- **int** getPriority()
- **void** setDaemon (**boolean** on)

"Hintergrundprozess": terminiert nicht mit dem Erzeuger

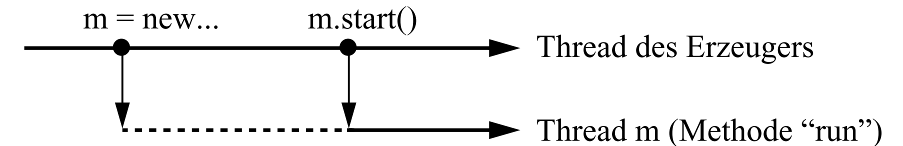
- Jeder Thread (genauer: jede von Thread abgeleitete Klasse) muss eine void-Methode *run()* enthalten
  - diese macht die eigentlichen Anweisungen des Threads aus!
  - "run" ist in Thread nur abstrakt definiert

Erzeugen von Threads

- Typisches Gerüst für einen Thread:

```
class Mythread extends Thread
{ int mynumber;
  public Mythread(int number)
  { mynumber = number; }

  public void run()
  { // hier die Anweisungen des Threads
     ...
  }

  // hier andere Methoden
}
```

Konstruktor



- Erzeugen aus einem anderen Thread heraus:

```
Mythread m = new Mythread(5);
m.start();
```

mit dieser Nummer identifizieren wir einen Thread individuell

damit kann man den Thread kontrollieren (z.B. m.suspend();)

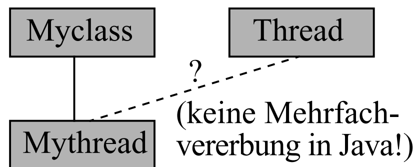
- Alternative: "Anonyme" Erzeugung

```
new Mythread(5).start();
```

Mythread "kann" start, da dies von Thread ererbt ist

- Zusammenfassen der beiden Anweisungen
- dann aber keine Kontrolle möglich, da kein Zugriff auf den Thread

Abgeleitete Klassen als Threads



- Myclass sei eine Klasse, die nicht von Thread abgeleitet ist
- Mythread soll Unterklasse von Myclass sein; gleichzeitig aber auch einen Thread darstellen

- Lösung über die Runnable-Schnittstelle:

```
class Mythread extends Myclass implements Runnable
{ ...
  public void run()
  { ... }
  ...
}
```

Definition ("Implementierung") von run muss hier erfolgen

- Erzeugung aus einem anderen Thread heraus:

```
Mythread m = new Mythread(...);

Thread t = new Thread (m);
t.start();
```

m "kann" kein start, da dies nicht in runnable enthalten; erst t als Thread kann start

zweite Form des Konstruktors!

- Es geht auch so:

```
Mythread m = new Mythread(...);
```

In der Klasse Mythread dann an geeigneter Stelle:

```
t = new Thread(this);
t.start();
```

Assoziation des Threads zur Methode run herstellen

Ein Thread-Beispiel

```
class T extends Thread
{ String wer; int delay = 0;

  T (String s, int zeit)
  { wer = s; delay = zeit;
    System.out.println(wer + " : " + delay);
  }

  public void run()
  { try
    { while (true)
      { System.out.println(wer);
        sleep(delay);
      }
    }
    catch (InterruptedException e) {return;}
  }

  public static void main(String args[])
         throws InterruptedException
  { int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    new T("She loves me", i).start();
    sleep(i/2);
    new T("  not", j).start();
    System.out.println("Jetzt sind beide
                        Threads gestartet");
  }
}
```

Konstruktor

Exception, falls während des sleep ein Interrupt ausgelöst wird

Methode run und damit den Thread beenden

statt try / catch

Gründen zweier Threads als Instanzen der eigenen Klasse

Die Threads in Aktion

```
> java T 500 500

She loves me: 500
She loves me
  not : 500
Jetzt sind beide Threads gestartet
  not
She loves me
  not
She loves me
  not
She loves me
  not
She loves me
  not
She loves me
  not
She loves me
....
```

Vielleicht hätte man für ein Happy-End doch die Parameter anders wählen sollen?

```
> java T 1000 300

She loves me: 1000
She loves me
  not : 300
Jetzt sind beide Threads gestartet
  not
  not
She loves me
  not
  not
  not
She loves me
  not
  not
  not
  not
She loves me
  not
  not
  not
She loves me
  not
  not
  not
She loves me
....
```


Ein weiteres Thread-Beispiel

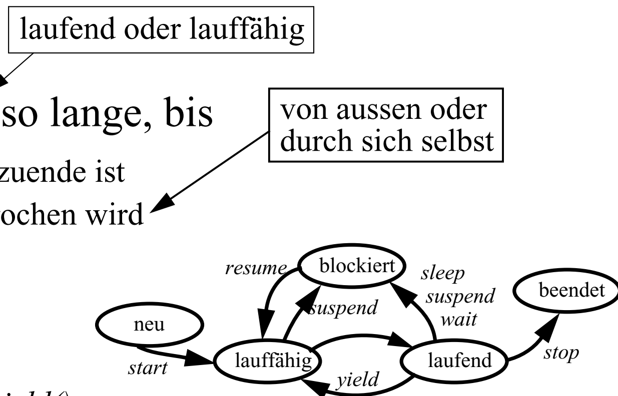
- Ein Thread “Hin” schreibt Sterne; ein Thread “Her” löscht Sterne - beide arbeiten (quasi)parallel
  - nach einer Idee von Ralf Kühnel (“Die Java-Fibel”, Addison-Wesley)

```
public class HinHer
{ public static void main (String args[])
  { System.out.print("********************");
    System.out.flush();
    new Hin().start();
    new Her().start();
  }
}

class Hin extends Thread
{ public void run()
  { try
    { while(true)
      { sleep((int)(Math.random() * 1000));
        repaint();
        System.out.flush();
      }
    }
    catch (InterruptedException e) {return;}
  }

  public void repaint()  // Sterne hinzufuegen
  { System.out.print("*****"); }
}

class Her extends Hin
{ public void repaint()  // Sterne loeschen
  { System.out.print("\b\b\b\b\b    \b\b\b\b\b"); }
}
```

Übung: Ausprobieren, wer gewinnt!

Thread-Kontrolle

- Ein Thread läuft so lange, bis
  - seine run-Methode zuende ist
  - er mit stop() abgebrochen wird



- Ein Thread kann *sich selbst*
  - die cpu entziehen: *yield()* (Übergang in Zustand "lauffähig"; wird automatisch wieder "laufend", wenn keine wichtigeren Threads mehr laufen möchten)
  - schlafen legen: *sleep()* (automatisches resume nach gegebener Zeit)
  - anhalten: *suspend()* bzw. *wait()*
  - beenden: *stop()*
  - in der Priorität verändern: *setPriority()*

Prioritäten: normal 5, minimal 1, maximal 10 (anfangs: Priorität des erzeugenden Prozesses)

- Ein Thread kann einen *anderen* Thread t
  - *starten*: t.start()
  - *anhalten*: t.suspend()
  - *fortsetztbar machen*: t.resume();
  - *beenden*: t.stop()
  - in der *Priorität verändern*: t.setPriority()

hierzu ist eine Referenz auf den Thread notwendig

Beachte: *stop*, *suspend* und *resume* führen, unbedacht angewendet, zu unsicheren Programmen und sollte daher eigentlich nicht mehr verwendet werden

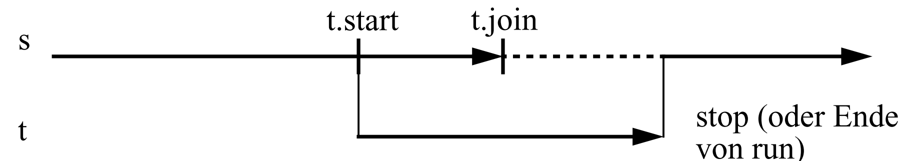
Warten auf Threads

- Methode *join* verwenden, wenn auf die Beendigung eines anderen Threads gewartet werden soll
  - z.B. weil auf die von ihm berechneten Daten zugegriffen werden soll

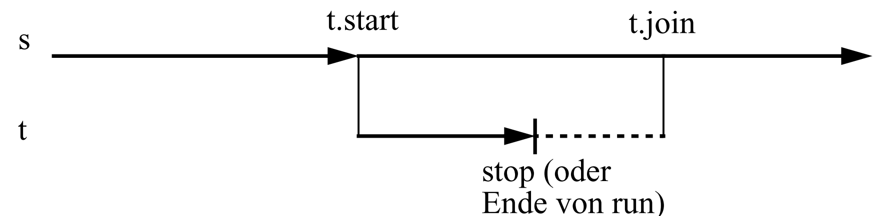
- Das Objekt eines beendeten Threads existiert weiter
  - auf dessen Zustand kann also noch zugegriffen werden

- Auf einen beendeten Thread kann *start* aufgerufen werden
  - run-Methode wird dann erneut ausgeführt

- Beispiel: Thread s wartet tatsächlich auf t:



- Thread t ist eher fertig:



- Nach t.join ist jedenfalls garantiert, dass t beendet ist

Thread-Scheduling

- Scheduling: Planvolle Zuordnung der cpu an die einzelnen Threads (jeweils für eine gewisse Zeit)
- Genaue Scheduling-Strategie ist *nicht* Bestandteil des Java-Sprachstandards
  - kann jede Implementierung für sich entscheiden (und damit Eigenheiten des zugrundeliegenden Betriebssystems effizient nutzen)
  - man darf sich daher nicht auf "Erfahrungen" verlassen
  - genauer: nicht auf die Wirkung von Zeitscheiben oder Prioritäten etc.

sonst nicht deterministisch und nicht portabel!

- Konsequenzen:
  - Test und Debugging ist sehr schwierig
  - alle denkbaren verzahnten Abläufe ("interleavings") berücksichtigen
  - Menge der verzahnten Abläufe durch geeignete *Synchronisation* einschränken (nur "korrekte" Abläufe zulassen)
  - ggf. mit "yield" Scheduling teilweise selbst realisieren (z.B. um andere Prozesse am Verhungern zu hindern)
- Vorgabe: Ein Thread-Scheduler *soll* Threads mit höherer Priorität bevorzugen
  - Priorität eines Threads entspricht zunächst der des Erzeugers
  - Priorität kann verändert werden (*setPriority*)
  - Thread mit der höchsten Priorität *sollte* immer laufen (ohne Garantie!)
  - wenn ein Thread mit höherer Priorität als der gegenwärtig ausgeführte lauffähig wird, wird der gegenwärtige i.a. unterbrochen

auch bei einem Rechner mit zwei cpus?

Thread-Scheduling (2)

im Sinne von "laufend"

(sofortiger) Threadwechsel ist aber nicht garantiert!

- Wie lange läuft ein Thread?
  - bis ein Thread mit höherer Priorität lauffähig wird
  - bis er sich beendet (mit "stop" oder dem Ende von "run")
  - bis er in den "blockiert"-Zustand übergeht (explizit mit "suspend", "wait", "sleep" etc. ; implizit durch E/A etc. - es ist aber umgekehrt nicht garantiert, dass ein auf E/A-wartender Thread die cpu freigibt!)
  - bis er mit "yield" die Kontrolle dem Scheduler übergibt
- Scheduling mit Zeitscheiben *kann* vom System realisiert sein, *muss* aber *nicht*
  - Thread läuft längstens bis zum Ablauf der Zeitscheibe (dann i.a. Round-Robin-Scheduling unter Threads gleicher maximaler Priorität)
- Konsequenzen:
  - Thread mit Endlosschleife kann gegebenenfalls das ganze System blockieren (andere Threads "verhungern")
  - Threads gleicher Priorität verhalten sich besonders willkürlich
  - "yield" ist insbesondere bei Systemen ohne Zeitscheiben wichtig
  - Prioritäten sollten besser nicht als Synchronisationsmittel (zum Erzwingen einer bestimmten Reihenfolge etc.) eingesetzt werden
  - Portabilität ist bei diletantischer Nutzung von Threads eingeschränkt

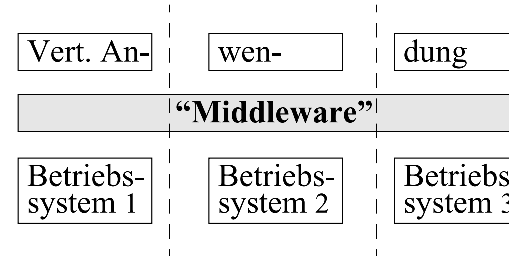
---

Exkurs-Ende (Threads in Java)

Middleware

Middleware

- Zweck: Durch eine geeignete Softwareinfrastruktur die Realisierung verteilter Anwendungen vereinfachen
  - kann man für viele Anwendungen gemeinsame Aspekte herausfaktorisieren?



- Aufgabe von Middleware:
  - Verteiltheit (für die Anwendung) möglichst transparent machen (z.B. globaler Namensraum, globale Zugreifbarkeit, Ortstransparenz)
  - zumindest aber die Verteiltheit einfach handhabbar machen

- Soll insbesondere Kommunikation und Kooperation zwischen Anwendungsprogrammen unterstützen
  - Verbergen von Heterogenität von Rechnern und Betriebsystemen (z.B. durch einheitliche Datenformate)
  - einheitliche „Umgangsformen“: Schnittstellen, Protokolle

- Bietet gewisse Basismechanismen und -dienste für verteiltes Programmieren an, z.B.
  - Verzeichnis- und Suchdienste (name service, lookup service,...)
  - Weiterleiten von events

Übersicht: “Historische” Entwicklung

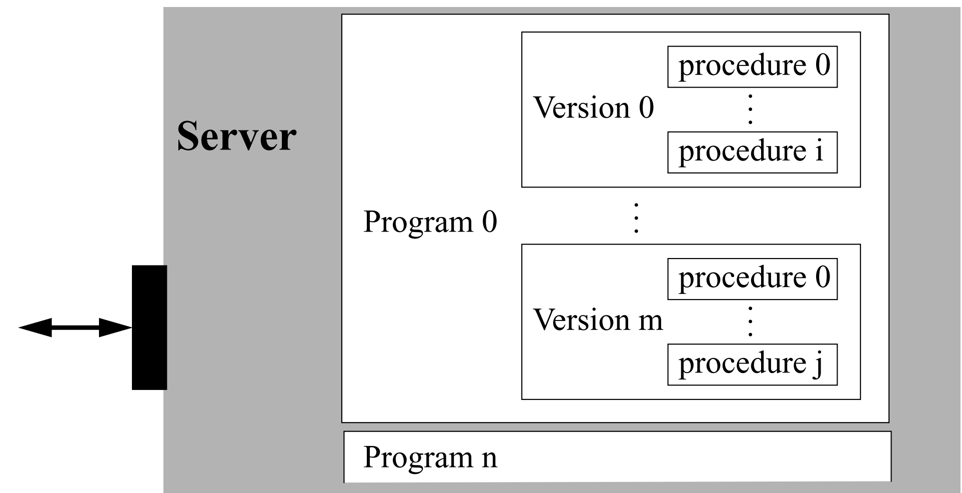
1. RPC-Bibliotheken: z.B. Sun-RPC
   - Unterstützung von Client-Server-Paradigma und RPC-Kommunikation
   - Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
   - Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)
2. Client-Server-Verteilungsplattformen: z.B. DCE
   - Zeitdienst, Verzeichnisdienst
   - globaler Namensraum, globales Dateisystem
   - Programmierhilfen: Synchronisation, Multithreading,…
3. Objektbasierte Verteilungsplattformen: z.B. CORBA
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache
   - “Object Request Broker”
4. Web-Services
   - Dienstorientierung aufbauend auf WWW als Plattform (SOAP, XML)
5. Infrastruktur für spontane Kooperation (z.B. Jini)
   - unterstützt Dienstorientierung, Mobilität, Dynamik

Beachte: Der Begriff “Middleware” ist im Laufe der Zeit zunehmend verwässert worden

- oft nicht nur gebraucht im technischen Sinne als Verteilungsplattform und Kommunikations- und Diensteinfrastruktur
- sondern auch für fast alles, was nicht direkt Anwendung oder Betriebssystem ist, also z.B. auch Datenbanken etc.

Sun-RPC: Service-Identifikation

- Sun-RPC: Entwickelt von der Firma Sun für UNIX, jedoch auch auf anderen Betriebssystemen vorhanden
- Eine entfernte Prozedur wird identifiziert durch das Tripel (prognum, versnum, procnum)



- Jede Prozedur eines Dienstes realisiert eine Teilfunktionalität (z.B. open, read, write,... bei einem Dateiserver)
- Prozedur Nummer 0 ist vereinbarungsgemäss für die “Nullprozedur” reserviert
  - keine Argumente, kein Resultat, sofortiger Rückkehr (“ping-Test”)
- Mit der Nullprozedur kann ein Client feststellen, ob ein Dienst in einer bestimmten Version existiert:
  - falls Aufruf von Version 4 des Dienstes XYZ nicht klappt, dann versuche, Version 3 aufzurufen...

Sun-RPC: Service-Registrierung

int *rpc_reg*(prognum, versnum, procnum, procname, inproc, outproc)

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameters; procname must be a procedure that returns a pointer to its static result; *inproc* is used to XDR-decode the parameters while *outproc* is used to XDR-encode the results.

- Welche Programmnummer bekommt ein Service?

→ Einige Programmnummern für *Standarddienste* sind vom System bereits fest konfiguriert:

portmapper	100000	portmap
rstatd	100001	rup
rusersd	100002	rusers
nfs	100003	nfsprog
ypserv	100004	ypprog
mountd	100005	mount
...	...	
keyserv	100029	keyserver

Linke Spalte: Servicename

Zuordnung mittels *getrpcbyname*() und *getrpcbynumber*() möglich

Rechte Spalte: Kommentar

→ Ansonsten freie Nummer wählen:

neu und “enhanced”: “rpcb_set”

TCP oder UDP

- Mit *pmap_set*(prognum, versnum, protocol, port) bekommt man den Returncode FALSE, falls prognum bereits (dynamisch) vergeben; ansonsten wird dem Service die Portnummer ‘port’ zugeordnet

Sun-RPC: Service-Aufruf

int *rpc_call*(host, prognum, versnum, procnum, inproc, in, outproc, out)

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine *host*. The parameter *in* is the address of the procedure’s argument, and *out* is the address of where to place the result; *inproc* is an XDR function used to encode the procedure’s parameters, and *outproc* is an XDR function used to decode the procedure’s results.

Warning: You do not have control of timeouts or authentication using this routine.

- Es gibt auch eine Broadcast-Variante:

rpc_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)

Like *rpc_call*(), except the call message is broadcast... Each time it receives a response, this routine calls *eachresult*(). If *eachresult*() returns 0, *rpc_broadcast*() waits for more replies.

Sun-RPC: Secure RPC

- Client und Server vereinbaren einen DES-Session-Key K nach dem Diffie-Hellman-Prinzip *("shared secret")*
  - wird zum Verschlüsseln der Nachrichten genutzt
- Mit jeder Request-Nachricht wird ein mit K kodierter Zeitstempel als Verifier mitgesandt
- Die erste Request-Nachricht enthält ausserdem verschlüsselt eine Zeitfenstergrösse W als zeitliches Toleranzintervall sowie (ebenfalls verschlüsselt) W-1
  - "zufälliges" Generieren einer ersten Nachricht ist nahezu unmöglich
  - replay (bei kleinem W) ist ebenfalls erfolglos
  - W ist verschlüsselt, um Angreifern keine Informationüber die Fenstergrösse und auch kein Klartext-Schlüsseltext-Paar zu geben
- Server überprüft jeweils, ob:
  - (a) Zeitstempel grösser als letzter Zeitstempel
  - (b) Zeitstempel innerhalb des Zeitfensters
- Die Antwort des Servers enthält (verschlüsselt) den letzten erhaltenen Zeitstempel-1 (→ Authentifizierung!)
  - gelegentliche Uhrenresynchronisation nötig (RPC-Aufruf kann hierzu optional die Adresse eines "remote time services" enthalten)

CORBA

- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - erste brauchbare (d.h. interoprable) Version: 1997 (CORBA 2.0)

eine *Architektur*, kein Produkt!

- Propagiert durch die OMG (Object Management Group)
  - herstellerübergreifendes Konsortium
  - Ziel: Bereitstellung von Konzepten für die Entwicklung verteilter Anwendungen
  - genauer: Definition und Entwicklung einer Architektur für kooperierende objektorientierte Softwarebausteine und Services in verteilten heterogenen Systemen (→ *"Middleware"*)

Beachte: *Objektorientierung* selbst ist eigentlich ein "altes" Konzept

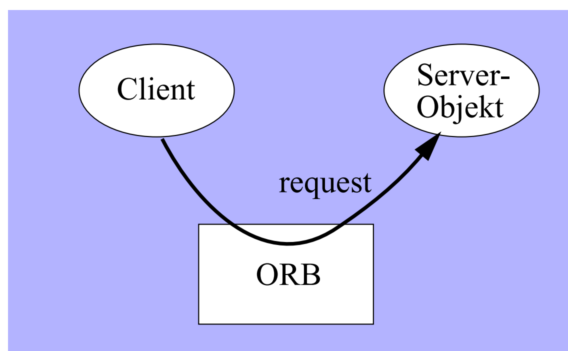
- Mitte der 1960er-Jahre (Programmiersprache "Simula")
- damals bereits fast alle Aspekte der Objektorientierung (Klassenhierarchien, virtuelle Klassen, Polymorphismus,...)

Man lese zu CORBA auch: Michi Henning: *The rise and fall of CORBA.* Commun. ACM, Vol. 51, No. 8 (August 2008), pp. 52-57

Mehr zur CORBA allgemein: Oliver Haase: *Kommunikation in verteilten Anwendungen (2. Auflage).* R. Oldenbourg Verlag, 2008, Kapitel 7

CORBA - Übersicht

- *Objektmodell*
- *IDL* (**I**nterface **D**escription **L**anguage) mit entsprechenden Generatoren und Compilern
- *ORB* (**O**bject **R**equest **B**roker) als Vermittlungsinfrastruktur

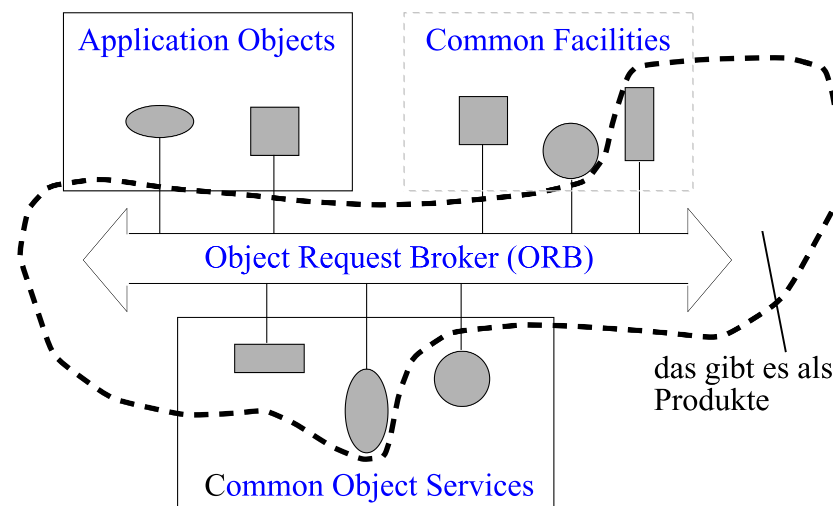


- *Systemfunktionen* in Form von Object Services
- Unterstützung von Anwendungen durch *Common Facilities*
- *Konventionen* bezüglich Schnittstellen, Protokollen etc.

- CORBA-Implementierungen sind also *Infrastrukturen* und unterstützen die *Ausführung* vert. objektorientierter Anwendungen in heterogenen Systemen
- *Entwurfs- und Spezifikationsaspekte* solcher Systeme werden dagegen mit anderen Konzepten unterstützt, z.B. *UML* ("Unified Modeling Language"), mit der u.a. Diagrammnotationen standardisiert werden

Object Management Architecture

- "OMA" ist eine *Referenzarchitektur*, welche die wesentlichen Bestandteile einer Plattform für verteilte objektorientierte Applikationen definiert



- *Application Objects*: Objekte der eigentlichen Anwendung
  - gehören damit nicht zur CORBA-Infrastruktur
- *ORB*: Vermittlung zwischen verschiedenen Objekten; Weiterleitung von Methodenaufrufen etc.
  → Ortstransparenz, Kommunikation,...
- *Object Services*: Schnittstelle zu standardisierten wichtigen Diensten
- *Common Facilities*: allgemein nützliche Funktionalität
  - nicht Teil aller CORBA-Implementierungen, oft sind nur wenige Funktionen davon realisiert

Object Services (1)

- COSS (Common Object Services Specification) als Basisdienste für eine systemweite Infrastruktur
  - mit objektorientierter Schnittstelle
  - nicht alle Dienste wurden aber vollständig spezifiziert (oder gar realisiert)

---

1) Ereignismeldung
   - Weiterleitung asynchroner Ereignisse an Ereigniskonsumenten
   - Einrichten von “event channels” mit Operationen wie push, pull,...

2) Persistenz
   - Dauerhaftes Speichern von Objekten auf externen Medien

3) Naming
   - Erzeugung von Namensräumen
   - Abbildung von Namen auf Objektreferenzen
   - Lokalisierung von Objekten

4) Trading
   - Matching von Services zu einer Service-Beschreibung eines Clients

5) Time
   - Uhrensynchronisation etc.

6) Security

Object Services (2)

7) Concurrency control
   - Semaphore, Locks,...

8) Transaktionen
   - 2-Phasen-Commit etc.

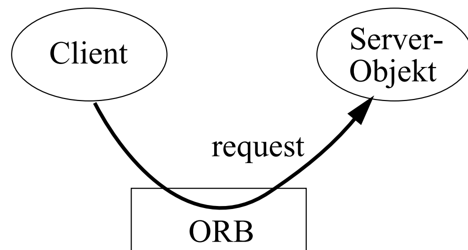
9) Replikation
   - Sicherstellung der Konsistenz replizierter Objekte in einer verteilten Umgebung

10) Externalization
   - Export von Objekten in sequentielle Dateien

- und noch einige weitere (hier nicht relevante) Services

Kommunikation zwischen Objekten

- Menge interagierender Objekte in zwei typischen Rollen
  - Client-Objekt (Aufrufer)
  - Server-Objekt

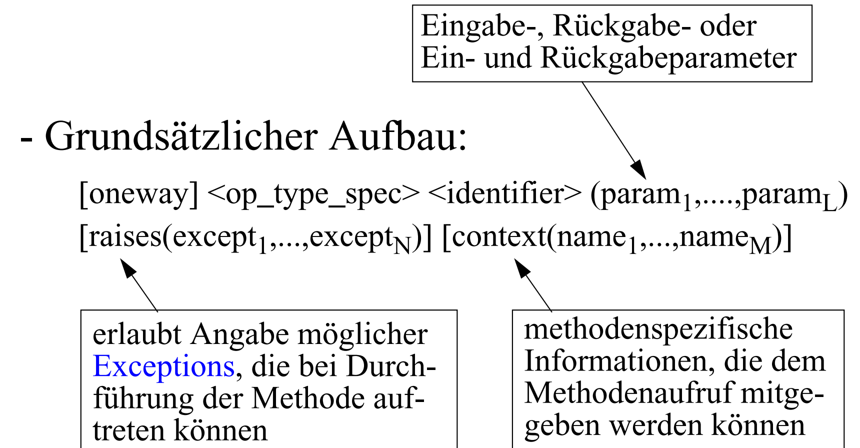


- Methodenaufruf durch requests unterschiedl. Semantik
  - *synchron* (mit Rückgabewerten; analog zu RPC)
  - *verzögert synchron* (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)
  - *one way* (asynchron: Aufrufer wartet nicht; keine Ergebnisrückgabe)

- Für den transparenten Transport eines Methodenaufrufs vom Client zum Server ist der ORB zuständig

Interface Description Language (IDL)

- Sprache zur Definition von Schnittstellen (Parameter, Attribute, Superklasse bzgl. Vererbung, Exceptions,...)
- Sprachneutral, aber lexikalisch an C++ angelehnt
- Bsp: `oneway void move (in long x, in long y)`

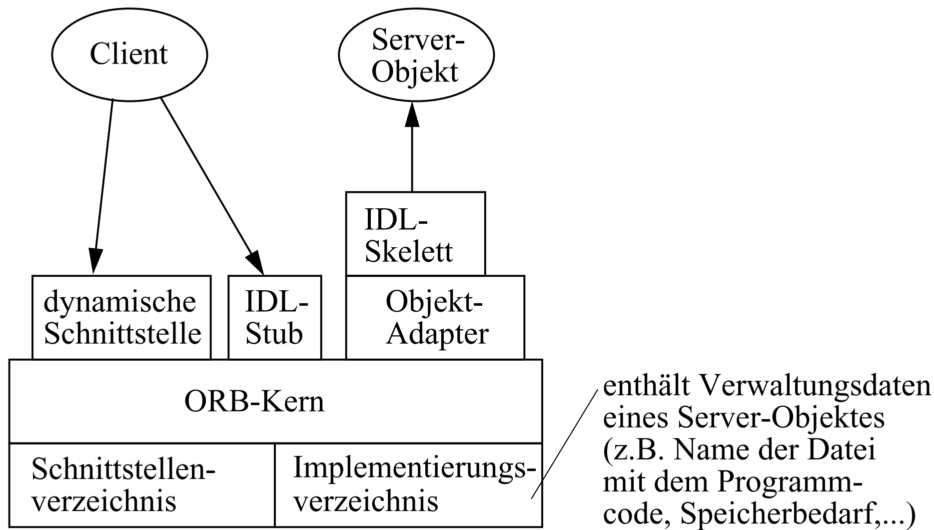


- Aus einer Schnittstellenspezifikation in IDL erzeugt ein Compiler einen IDL-Stub für Clients und ein IDL-Skelett für Server

---

- Mehr zur CORBA-IDL siehe z.B.: Oliver Haase: *Kommunikation in verteilten Anwendungen (2. Auflage).* R. Oldenbourg Verlag, 2008, Kapitel 7.2

ORB



- ORB bietet Clients zwei Arten von Schnittstellen für den Methodenaufruf an
  - *statische Schnittstelle* (Erzeugung von Stubs aus der IDL-Beschreibung mit Compiler analog zu RPCs)
  - *dynamische Schnittstelle* (Client kann zur Laufzeit das Schnittstellenverzeichnis abfragen und einen geeigneten Methodenaufruf zusammenstellen)

- Objektadapter: Steuert anwendungsunabhängige Funktionen des Server-Objekts
  - u.A. Aktivierung des Server-Objektes bei Eintreffen eines requests, Authentifizierung von requests, Zuordung von Objektreferenzen zu Objektinstanzen etc.
  - zuständig ausserdem für Registrierung von Services
  - es gibt einen standardisierten Basic Object Adapter (BOA), der für viele Anwendungen ausreichende Grundfunktionalität bereitstellt

Server-Objekte

- Bereitsstellung von Services für Clients
  - analog zu Prozeduren, die im Rahmen von RPCs verwendet werden

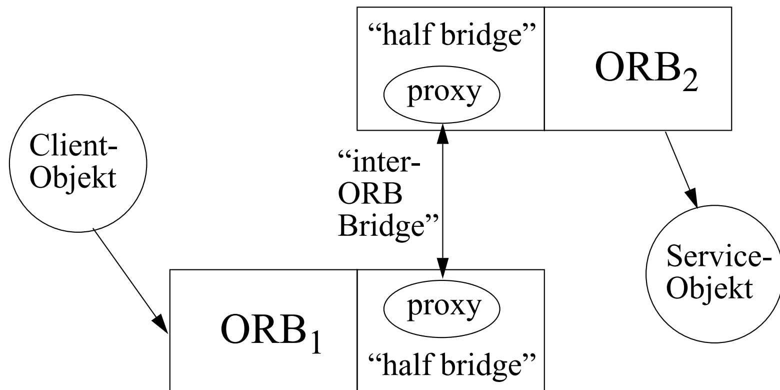
- Objekte können ein aus der IDL-Spezifikation generiertes Objekt-Skelett nutzen

- Objekt muss sich beim lokalen Objekt-Adapter anmelden und dabei eine “server policy” angeben:
  - *Shared Server:* kann zusammen mit mehreren anderen aktiven Server-Objekten von einem einzigen Prozess verwaltet werden
  - *Unshared Server*
  - *Server per Method:* Start eines eigenen Prozesses bei Methodenaufruf
  - *Persistent Server:* ein Shared Server, der von CORBA initial bereits gestartet wurde

- Objekt muss sich ferner beim Implementierungsverzeichnis anmelden
  - damit es bei einem Methodenaufruf durch Clients gefunden wird

ORB Bridges

- Interoperabilität von ORBs verschiedener Herstellerimplementierungen



- ORB Bridge: Formatkonvertierung und Weiterleitung eines requests etc. an einen anderen ORB
  - Schnittstellen und Konventionen für solche Bridges sind im CORBA-Standard festgelegt
  - Bridge besteht aus zwei Teilen mit einer CORBA-Schnittstelle, welche bei Bedarf Proxy-Objekte für die Aufrufkonvertierung erzeugen

- Inter-ORB-Kommunikation mittels GIOP (General Inter-ORB Protocol; z.B. TCP/IP-basiert)

CORBA - weitere Entwicklungen

Ab ca. 2000 entstand der Wunsch nach einer wesentlichen Erweiterung der CORBA-Funktionalität. Gründe:

- Anforderungen durch E-Commerce-Anwendungen
- Ausbreitung des WWW (und später: Web-Services, SOAP,...)
- Aufkommen von Java (und später: EJB, Jini,...)
- Aufkommen mobiler Geräte

Dem sollte durch Weiterentwicklung ("CORBA 3.0") der Spezifikation Rechnung getragen werden:

- Messaging Service und asynchroner Methodenaufruf
- Objects by Value
- Persistente Objekte
  - "Abspeichern" von Objekten
- Komponenten-Modell
- Java-Unterstützung
  - Generieren von IDL aus Java bzw. Java-RMI ("reverse mapping")
- Firewall-Unterstützung
  - klassische Firewalltechnik (z.B. Services identifiziert mit Portnummern) versagt teilweise; Callbacks erscheinen als Aufruf von aussen...
- Minimum CORBA
  - Unterstützung von embedded systems (i.w. Weglassen von Dynamik)
- Realzeit-Unterstützung
- Fault Tolerant CORBA
  - durch redundante Einheiten

Messaging Service

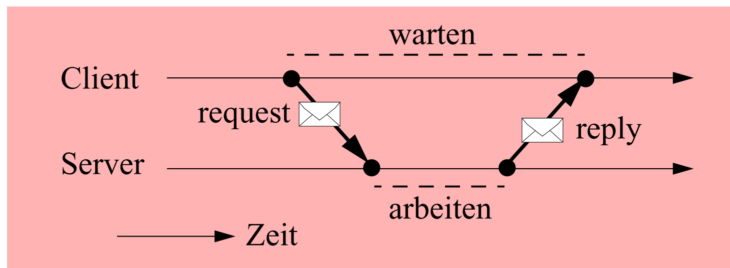
- Asynchrones Kommunikationsparadigma

- Motivation:

  - mobile Geräte (PDA, Laptop,...) sind oft nicht online

  - bei sehr grossen verteilten Systemen sind fast immer einige Geräte bzw. Services nicht erreichbar (Netzprobleme etc.)

- CORBA basierte bisher auf einer engen (“synchronen”) Kopplung von Client und Server



- Asynchron:

  - Entkopplung von Sender / Empfänger
  - Sender blockiert nicht solange, bis Nachricht angekommen ist
  - Nachricht kann von Hilfsinstanzen zwischengespeichert werden, bis Kommunikationspartner erreichbar ist (“store & forward”)
  - Antwort kann evtl. von anderem Client als ursprünglichem Sender entgegengenommen werde

Asynchronous Method Invocation

- Bisherige Möglichkeiten eines Methodenaufrufs in CORBA:

  - *synchron* (insbes. bei Rückgabewerten; analog zu RPC)

  - *verzögert synchron* (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)

  - *one way* (Aufrufer wartet nicht) mit “best effort”-Semantik (“fire and forget”)

gedacht war an UDP-Implementierung; Semantik (z.B. Fehlermeldung bei Misslingen?) implementierungsabhängig

---

- Neu: Asynchronous Method Invocation (AMI)

  - bisher eher umständlich und fehleranfällig durchThreads zu simulieren

- Zwei Aufruftechniken bei AMI:

  - (1) Callback

    - Client gibt dem Aufruf eine Objektreferenz für die Antwort mit
    - Callback-Objekt kann sich im Prinzip irgendwo befinden
    - Kommunikations-Exceptions werden im Callback-Objekt ausgelöst

  - (2) Polling

    - Client erhält sofort ein Objekt zurück, das er für Polling oder zum Warten auf Antwort nutzen kann

Time-independent Invocation

- Aufruf von Objekten, die nicht aktiv sind oder zeitweise nicht erreichbar sind

- Aufruf-Nachrichten werden von zwischengeschalteten "Router Agents" verwaltet
  - Store and Forward-Prinzip
  - Router Agent beim Client ermöglicht disconnected operations
  - Router Agent beim Server kann dessen Eingangsqueue verwalten

- "Interoperable Routing Protocol" sorgt dafür, dass Router Agents verschiedener Hersteller interagieren

- Sogen. Quality of Service steuerbar (als "Policy")
  - z.B. max. Round Trip-Zeit: dadurch brauchen Router Agents die Nachrichten nicht beliebig lange aufbewahren
  - oder z.B. Aufrufreihenfolge: Soll Router seine gespeicherten Aufträge zeitlich geordnet oder nach Prioritäten oder... ausliefern?

CORBA-Probleme

- Die Weiterentwicklung von CORBA geriet ins Stocken
  - zu weitreichende Anforderungen → komplex / ineffizient
  - kommerzielle Implementierungen zögerlich
  - fehlende Unterstützung durch Microsoft (eigene Architektur)
  - OMG versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse,...)
  - aufkommende Konkurrenzsysteme, die z.T. besser an die neuen Anforderungen angepasst sind

---

Konkurrenzsysteme bzw. alternative Ansätze sind z.B.:

- Microsoft: DCOM und .Net
- Java-Technologie
  - z.B. Enterprise Java Beans (EJB), RMI
- Web-Services (und verwandte Systeme)
  - XML, WSDL, SOAP,...
  - Integrationsplattformen (z.B. WebSphere von IBM)

DCOM

Component Object Model

- DCOM: verteilte Version von COM (Microsoft)

- Teilweise analoge Zielsetzung zu CORBA
    - Integration, Interoperabilität, Komponenten-Modell
    - transparenter Zugriff auf entfernte Objekte (RPC-Idee)
    - statische und dynamische Aufrufe
    - IDL heisst hier MIDL (‘M’ für Microsoft)
    - Aufgaben von ORB und Object Adapter werden von einem “Service Control Manager” (SCM) wahrgenommen

- DCOM aber proprietär auf MS-Technologie ausgerichtet
    - Kapselung binärer Bibliotheken

- Keine Vererbung, stattdessen *Delegation* und *Aggregation*
    - Einbettung von Objekten in andere mit Weiterreichung von Schnittstellen nach aussen
    - *Aggregation:* automatisches Durchreichen der kompletten inneren Schnittstelle nach aussen
    - *Delegation:* Durchreichen eines Teils der inneren Schnittstelle nach aussen; äussere Schnittstelle ruft innere Methoden explizit auf

- Abgelöst durch .NET

.NET-Framework

- Microsoft-Softwareplattform; integriert in Windows
    - Laufzeitumgebung, Klassenbibliotheken (API), Services
    - für gemischtsprachige Programmierung in sogenannten .NET-Sprachen (u.a. C#, C++, J#, Visual Basic.NET (VB.NET))

- Quellcode in .NET-Sprache wird kompiliert in Common Intermediate Language (CIL)
    - entspricht etwa Java Bytecode

- Virtuelle .NET-Maschine (Common Language Runtime, CLR) führt CIL-Code aus
    - entspricht Java Virtual Machine
    - CLR enthält Just-in-Time (JIT) Compiler

- .NET-Remoting: entfernter Methodenaufruf
    - Clients verwenden lokale Proxies, die dieselbe Schnittstelle wie das entfernte Serverobjekt anbieten und Methodenaufrufe weiterleiten

- Verschiedene *Server-Aktivierungsmodi* möglich:
    - *Singleton:* ein einziges Serverobjekt für alle Methodenaufrufe
    - *SingleCall:* ein eigenes Serverobjekt für jeden Methodenaufruf
    - *klientenaktiviertes Objekt:* erzeugt neues Serverobjekt, das für alle Aufrufe desselben Klienten genutzt wird (dadurch kann klientenspezifische Zustandsinformation über Aufrufe hinweg gehalten werden)

- Alternativ zu .NET-Remoting bietet .NET auch *Socket-Kommunikation* und XML-basierte *Web-Services* an

Objektserialisierung

- Konvertierung von Objekten in Byteströme (und Rückkonvertierung in identische Kopie des Ausgangsobjekts)
  - Abspeichern von Objekten auf externen Medien
  - Kommunikation
- Java RMI (**R**emote **M**ethode **I**nvocation)
  - Schnittstelle "Serializable" implementieren
  - Serialisierung durch Schreiben auf "ObjectOutputStream"
  - Instanzvariablen, die nicht serialisiert werden sollen, werden mit "transient" gekennzeichnet
  - Deserialisierung durch Lesen von "ObjectInputStream"
- .NET: zwei mögliche Serialisierungsformate
  - *Binärformat:* platzsparend, effizient, nicht menschlich lesbar, gut für homogene .NET-Anwendungen
  - *SOAP-Format:* XML-Kodierung, interoperable mit anderen SOAP-Komponenten (auch Nicht-.NET)
  - zu serialisierende Klasse wird mit Attribut "Serializable" versehen → entspricht bei Java "Serializable"
  - nicht zu serialisierende Komponente wird mit Attribut "NonSerialized" markiert → entspricht bei Java "transient"

Web Services

- Definition des W3C (World Wide Web Consortium):

  A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

- Also: Ein entfernter Dienst, der über *SOAP-Nachrichten* aufgerufen werden kann; in der Web Service Description Language *(WSDL) beschrieben* ist; und im Universal Description Discovery and Integration *(UDDI)-Registry* verzeichnet ist.
- *WDSL:* XML-basierte Sprache zur Spezikation der Schnittstellen von Web Services
  - Rolle vergleichbar mit CORBA IDL
- *SOAP* (Simple Object Access Protocol): Austausch von XML-Nachrichten über HTTP (oder HTTPS) zum Zweck des Remote Procedure Calls
- *UDDI:* XML-basierter Verzeichnisdienst für Web Services
  - UDDI ist selbst ein Web Service → Anfragen über SOAP-Nachrichten
  - Ergebnisse sind WSDL-Dokumente

Web Services (2)

- Im Wesentlichen also RPC über Internet / WWW
  - das Web aufgefasst und genutzt als Software-Layer
- Eigenschaften
  - unabhängig von existierenden Plattformen (Sprachen, Middleware)
  - sehr lockere Koppelung von Client und Server
  - ubiquitär nutzbar, (im Prinzip) weltweit zugreifbar
- Web-Browser als kanonische Clients nutzbar
  - fungiert als Interface für Nutzer bei Web-Service-Applikationen
  - Browser werden mit Java-VM etc. auch zunehmend leistungsfähiger
- Problembereiche
  - Overhead für einen Aufruf ist relativ gross
  - http war als reines Dokumentenaustauschprotokoll für menschl. Nutzer konzipiert worden - nicht zur Kommunikation zwischen Computern
  - die Beschreibung von global anwendbaren Diensten für E-Commerce etc. stellt andere Anforderungen als die (rein syntaktische) Interface-Beschreibung klassischer Prozeduren in Programmiersprachen
  - UDDI-Service global ("universell") zu etablieren, ist eine technische und kommerzielle Herausforderung (teilweise Suchmaschinen-funktionalität!)
- Erweiterung der Web-Service-Idee:
  Service-orientierten Architekturen (SOA)

Mehr zu Web-Services in anderen Vorlesungen (→ Prof. G. Alonso)

Jini

Teil der Vorlesung „Verteilte Systeme"

Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications

Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications

- framework of APIs with useful functions / services
- helper services (discovery, lookup,...)
- suite of standard protocols and conventions

Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications

> - services, devices, … find each other automatically ("plug and play")
> - dynamically added / removed components
> - changing communication relationships
> - mobility

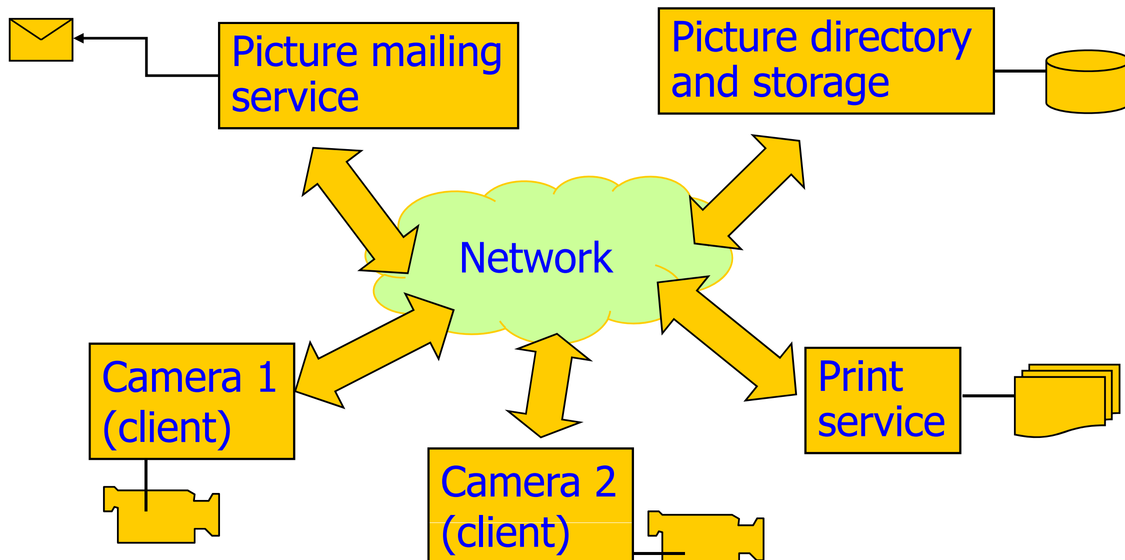
Jini

- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications
- Based on Java
  - uses RMI (Remote Method Invocation)
  - code shipping
  - requires JVM / bytecode everywhere
- Service-oriented
  - (almost) everything is considered a service
  - Jini system is a federation of services
  - service access through mobile proxy objects

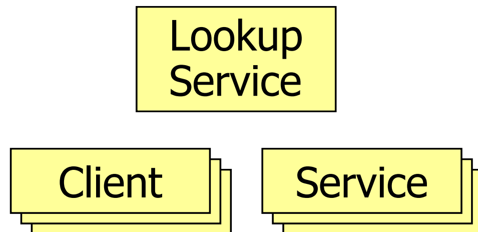
Service Paradigm

- (Almost) everything is a service
  - e.g. persistent storage, software filter, …
- Jini's run-time infrastructure offers mechanisms for adding, removing, finding, and using services
- Services are defined by interfaces and provide their functionality via their interfaces
  - services are characterized by their type and their attributes (e.g. "600 dpi", "version 21.1")
- Services (and service users) may "spontaneously" form a so-called "federation"

Example of a Jini Federation



Jini: Global Architecture



- Lookup Service (LUS)
  - main registry entity and brokerage service for services and clients
  - maintains information about available services
- Services
  - specified by Java interfaces
  - register together with proxy objects and attributes at the LUS
- Clients
  - know the Java interfaces of the services, but not their implementation
  - find services via the LUS
  - use services via proxy objects

Network Centric

- Jini is based on the network paradigm
  - "the network is the computer"
- Network = hardware and software infrastructure
- View is "network to which devices are connected to", not "devices that get networked"
  - network always exists, devices and services are transient
- Jini supports dynamic networks and adaptive systems
  - adding and removing components or communication relations should only minimally affect other components

Spontaneous Networking

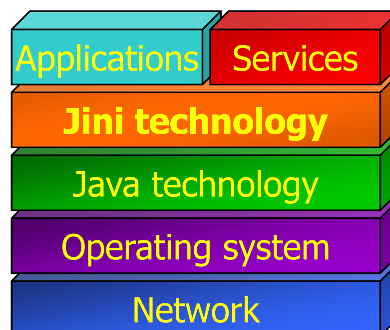
- Objects in an open, distributed, dynamic world find each other and form a transitory community
  - cooperation, service usage, ...
- Typical scenario: client wakes up (device is switched on, plugged in, ...) and asks for services in its vicinity
- Finding each other and establishing a connection should be fast, easy, and automatic

Some Fallacies of Common Distributed Computing Systems

- The "classical" idealistic view...
  - the network is reliable
  - latency is zero
  - bandwidth is infinite
  - the network is secure
  - the topology is stable
  - there is a single administrator
- ...isn't true in fact
  - Jini addresses some of these issues
  - (or at least it does not hide or ignore them)

Bird's-Eye View on Jini as a Middleware Infrastructure

- Jini consists of a number of APIs
- Is an extension to the Java platform dealing with distributed computing
- Is an abstraction layer between the application and the underlying infra-structure (network, OS)



Jini's Use of Java

- Jini requires JVM (as bytecode interpreter)
  - homogeneity in a heterogeneous world
  - but is this a realistic assumption?
- Devices that are not "Jini-enabled" or that do not have a JVM can be managed by a software proxy (somewhere in the net)

run protocols for discovery and join; have a JVM

Main Components of the Jini Infrastructure

- Lookup service
  - as repository / naming service / trader
- Protocols
  - discovery & join, lookup of services
  - based on TCP/UDP/IP
- Proxy objects
  - transferred from service to clients (via LUS)
  - represent the service locally at the client

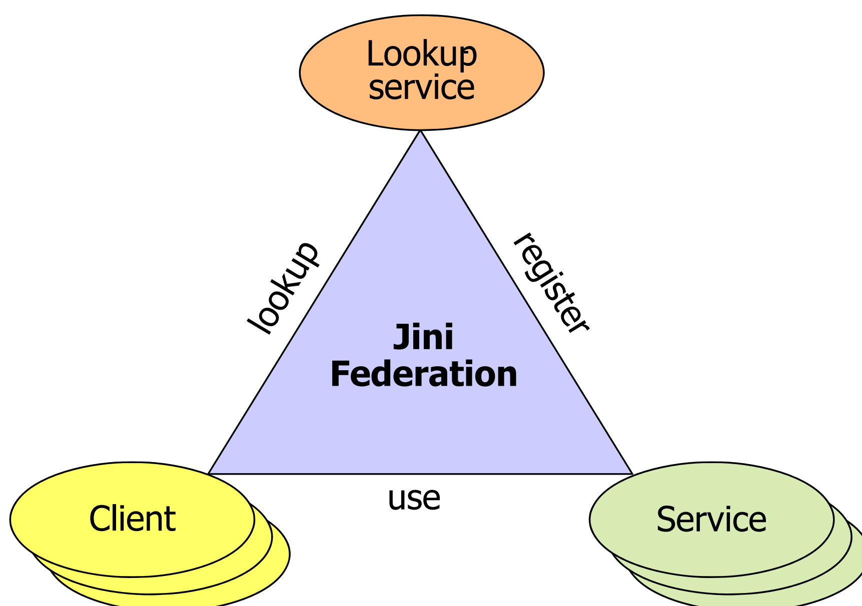
Context-Knowledge?

- Jini advocates spontaneous networking and formation of federations without prior knowledge of local environment
- Problem: How do service providers and clients learn about their local environments?
  - → lookup service!

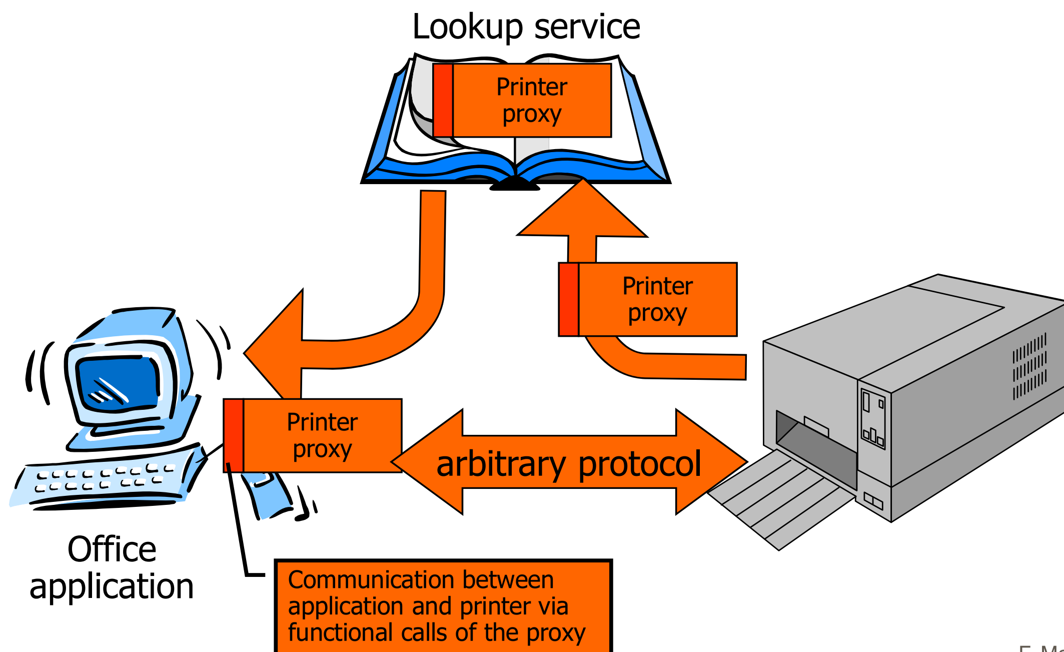
Lookup Service (LUS)

- Central component of every Jini federation
- Repository of services
- Similar to naming services (e.g., RMI registry) of other middleware architectures
- Functions as a "help-desk" for services and clients
  - registration of services (services advertise themselves)
  - distribution of services (clients lookup and find services)
- Has mechanisms to bring together services and clients

Lookup Service



Example



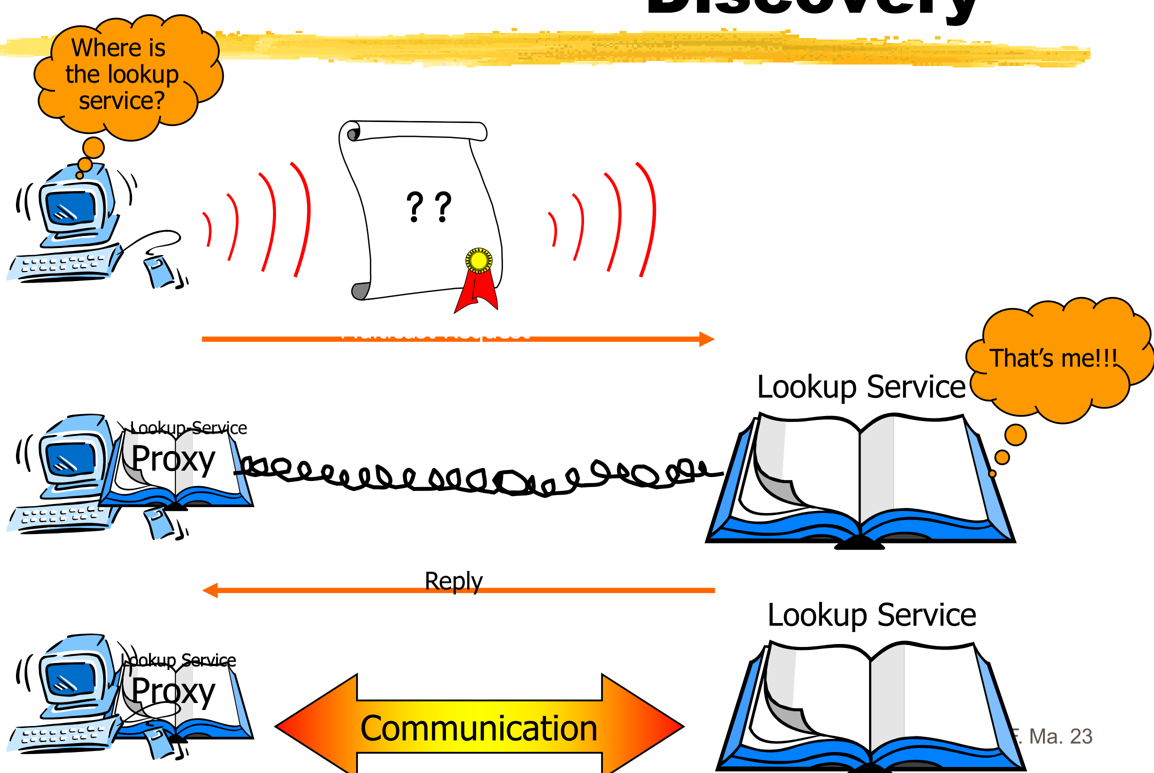
Lookup Service

- Uses Java RMI for communication
  - objects („proxies“) can migrate over the network
- Not only name/address of a service is stored (as in traditional naming services), but also:
  - set of attributes
    - e.g.: printer(color: true, dpi: 600, ...)
  - proxies, which may be complex classes
    - e.g. user interfaces
- Further possibilities:
  - responsibility can be distributed to a number of (logically separated) lookup services
  - increase robustness by running redundant lookup services

Discovery: Finding a LUS

- Goal: Find a lookup service (without knowing anything about the network) to
  - advertise (register) a service, or
  - find (look up) an existing service
- Discovery protocol:
  - multicast to well-known address/port
  - lookup service replies with a serialized object (its proxy)
    - communication with LUS then via this proxy

Discovery



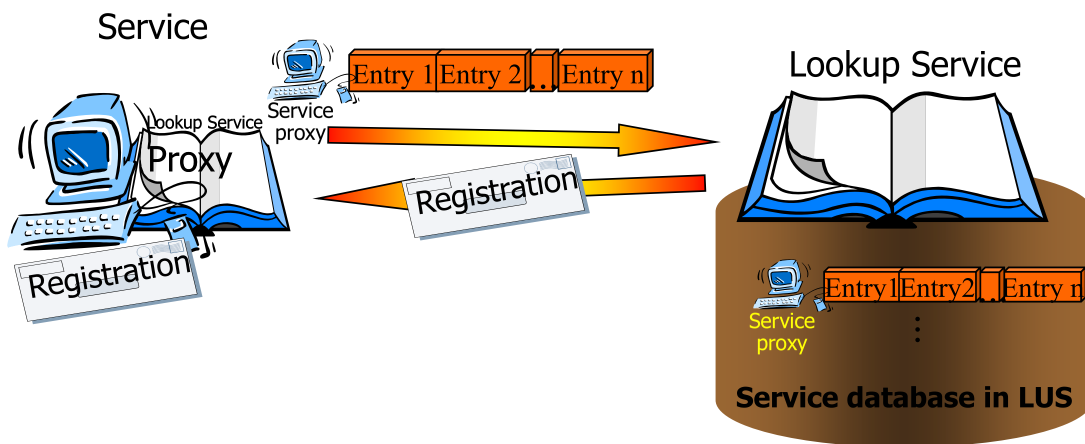
Multicast Discovery Protocol

- Search for lookup services
  - no information about the host network needed
- Discovery request uses multicast UDP packets
  - multicast address for discovery is 224.0.1.85
  - default port number of lookup services is 4160
  - recommended time-to-live is 15
  - usually does not cross subnet boundaries
- Discovery reply is establishment of a TCP connection
  - port for reply is included in multicast request packet

Join: Registering a Service

- Assumption: Service provider already has a proxy of the lookup service (→ discovery)
- It uses this proxy to register its service
- Gives to the lookup service
  - its service proxy
  - attributes that further describe the service
- Service provider can now be found and used in this Jini federation

Join



Join: More Features

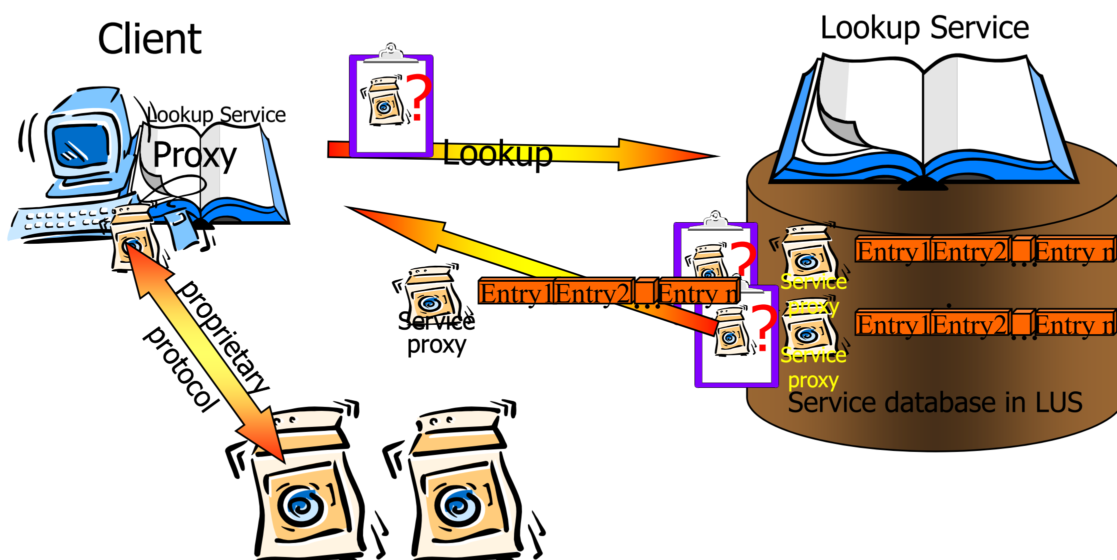
- To join, a service supplies:
  - its proxy
  - its ServiceID (if previously assigned; “universally unique identifier”)
  - set of attributes
  - (possibly empty) set of specific lookup services to join
- Service waits a random amount of time after start-up
  - prevents packet storms after restarting a network segment
- Registration with a lookup service is bound to a lease
  - service has to renew its lease periodically

Lookup: Searching Services

- Client creates query for lookup service
  - matching by registration number of service and/or service type and/or attributes possible
  - attributes: only exact matching possible (no “larger-than”, ...)
  - wildcards are possible („null“)
- Via its proxy at the client, the lookup service returns zero, one or more matches (i.e., server proxies)
- Selection among several matches is done by client

- Client uses service by calling functions of the service proxy
- Any “private” protocol between service proxy and service provider is possible

Lookup



Proxies

- Proxy object is stored in the lookup service upon registration
  - serialized object
  - implements the service interfaces
- Upon request, service proxy is sent to the client
  - client communicates with service implementation via its proxy: client invokes methods of the proxy object
  - proxy implementation hidden from client

Smart Proxies

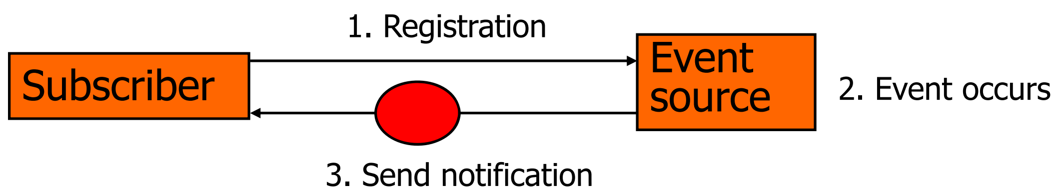
- Parts of (or the whole) service functionality may be executed by the proxy at the client
- When dealing with large volumes of data, it usually makes sense to preprocess parts of the data
  - e.g.: compressing video data before transfer
- Partition of service functionality depends on service implementer's choice
  - client needs appropriate resources

Leases

- Leases are contracts between two parties
- Leases introduce a notion of time
  - resource usage is restricted to a certain time frame
- Repeatedly express interest in some resource:
  - I'm still interested in X
    - renew lease periodically
    - lease renewal can be denied
  - I don't need X anymore
    - cancel lease or let it expire
    - lease grantor can use X for something else

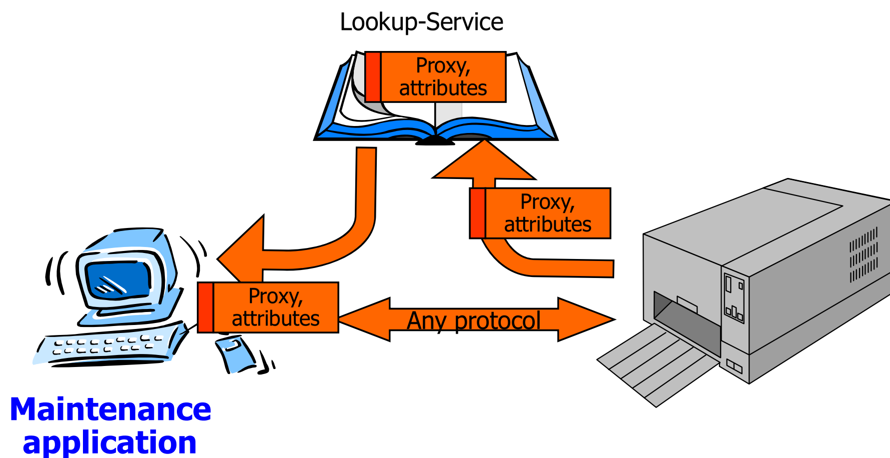
Distributed Events

- Objects on one JVM can register interest in certain events of another object on a different JVM
- "Publisher/subscriber" model



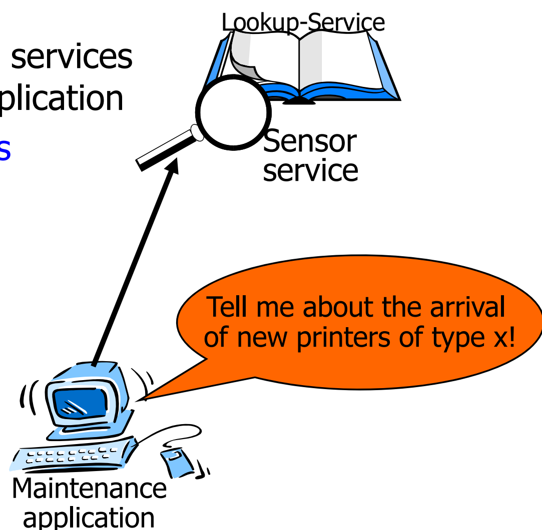
Distributed Events – Example

- Example: printer is plugged in
  - printer registers itself with local lookup service
- Maintenance application wants to update software



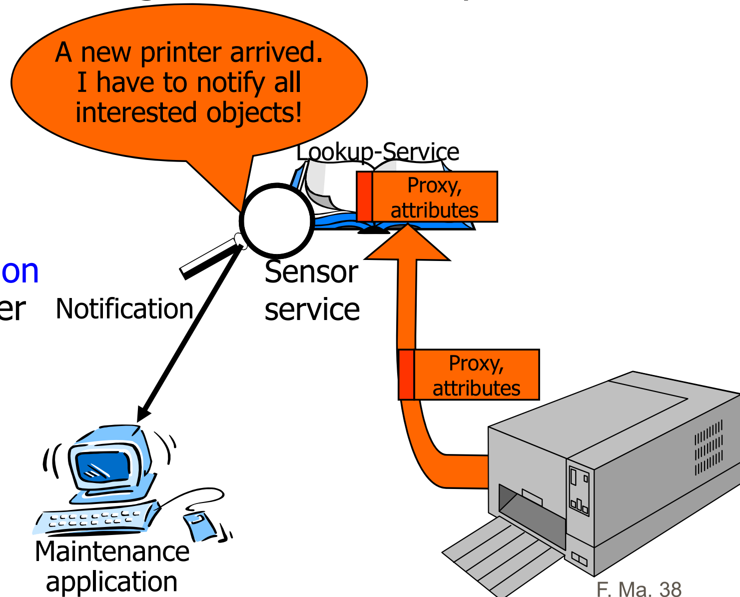
Distributed Events – Example

- Maintenance application is run on demand, search for printers is "outsourced"
  - "sensor service" looks for certain services on behalf of the maintenance application
  - maintenance application registers for events showing the arrival of certain types of printers
  - sensor observes the lookup service
  - notifies application as soon as matching printer arrives via distributed events



Distributed Events – Example

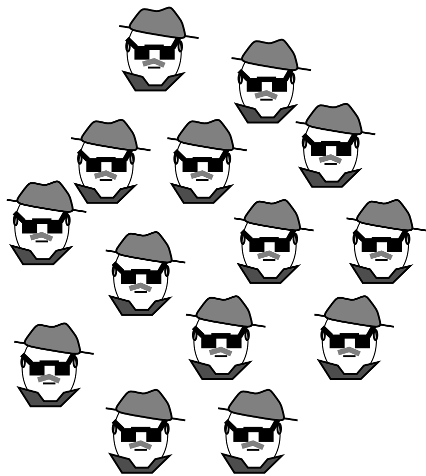
- Example: printer arrives, registers with lookup service
  - printer performs discovery and join
  - sensor finds new printer in lookup service
  - checks if there is an event registration for this type of printer
  - notifies all interested objects
  - maintenance application retrieves printer proxy and updates software



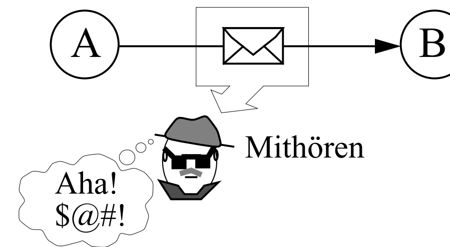
Jini Issues and Problem Areas

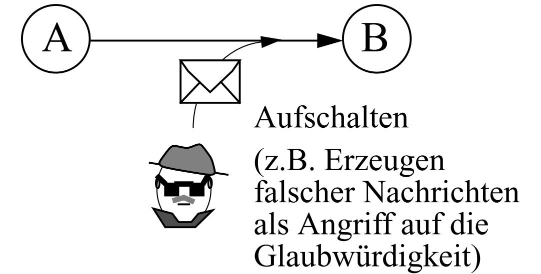
- Security
  - important especially in dynamic environments
  - services use other services on behalf of the user
    - principals, delegation
- Scalability
  - how well does Jini scale to a global level?
- Java centric
- Similar, non-Java-based systems
  - UPnP, Bluetooth SDP, SLP, HAVi, Salutation, e-speak, HP Chai,...
  - open, Internet-scale infrastructures (e.g., Web services)

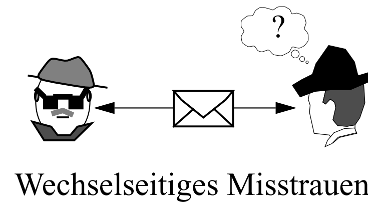
Sicherheit



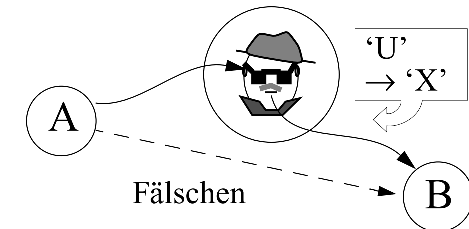
Sicherheit in verteilten Systemen

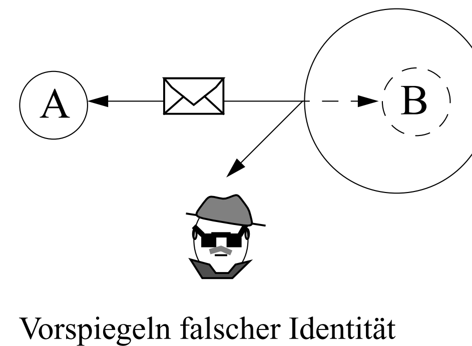




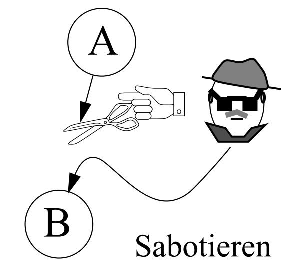


Wechselseitiges Misstrauen





Vorspiegeln falscher Identität



Sabotieren

Sicherheit: Anforderungen

- Autorisierung / Zugriffsschutz
  - Einschränkung der Nutzung auf den Kreis der Berechtigten
- Vertraulichkeit
  - Daten / Nachrichteninhalt gegen Lesen Unberechtigter schützten
  - Kommunikationsverhalten (wer mit wem etc.) geheim halten
- Authentizität
  - Absender "stimmt" (z.B. Server ist der, für den er sich ausgibt)
  - Daten sind "echt" und aktuell (→ Integrität)
- Integrität
  - Wahrung der Unversehrtheit von Nachrichten, Programmen und Daten
- Verfügbarkeit der wichtigsten Dienste
  - keine Zugangsbehinderung ("denial of service") durch andere
  - kein provozierter Absturz ("Sabotage")

- Weitergehende Anforderungen, z.B.:
  - Nichtabstreitbarkeit, accountability
  - strafrechtliche Verfolgbarkeit (z.B. Protokollierung; „Key Escrow")
  - Konformität zu rechtlich / politischen Vorgaben
  - ...

Sicherheit: Verteilungsaspekte

- *Offenheit* in verteilten Systemen "fördert" Angriffe
  - grosse Systeme → vielfältige Angriffspunkte
  - standardisierte Kommunikationsprotokolle → Angriff *einfach*
  - räumliche Distanz → Ortung des Angreifers schwierig, Angriff *sicher*
  - breiter Einsatz, allgemeine Verwendung → Angriff *reizvoller*
  - physische Abschottung nicht durchsetzbar
  - technologische Gegebenheiten: z.B. Wireless LAN ("broadcast")
- *Heterogenität*
  - sorgt für zusätzliche Schwachstellen
  - erschwert Durchsetzung einer einheitlichen Schutzphilosophie
- *Dezentralität*
  - fehlende netzweite Sicherheitsautorität

→ Gewährleistung der Sicherheit ist in verteilten Systemen *wichtiger* und *schwieriger* als in alleinstehenden Systemen!

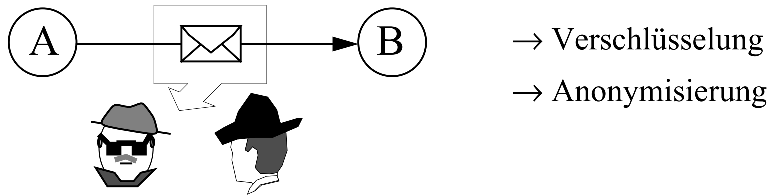
Typische Techniken und "Sicherheitsdienste":

- *Verschlüsselung*
- *Autorisierung* ("der darf das!")
- *Authentisierung* ("X ist wirklich X!")

} Hierfür Kryptosysteme und Protokolle als "Security Service", z.B. *Kerberos*

Angriffsformen

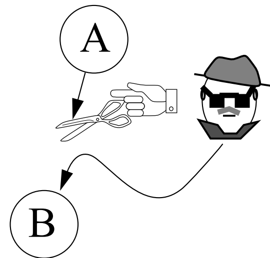
- *Passive Angriffe*: Beobachten der Kommunikation
    - Inhalt von Nachrichten in Erfahrung bringen
    - Kommunikationsverhalten analysieren ("wer mit wem wie oft?")



→ Verschlüsselung

→ Anonymisierung

- *Aktive Angriffe*: vorsätzliche Täuschung; Eindringen
    - Durchbrechen von Zugangsschranken
    - Verändern des Nachrichtenstroms (Verändern, Vernichten, Erzeugen, Vertauschen, Verzögern, Wiederholen ("replay") von Nachrichten)
    - Vorspiegelung falscher Identitäten (Maskerade: Nachahmen anderer Prozesse oder Nutzung eines fremden Passwortes)
    - Missbräuchliche Nutzung von Diensten
    - Denial of Service durch Sabotage oder Verhindern des Dienstzugangs, z.B. durch Überfluten mit Nachrichten



Authentifizierung

> *...Seid auf eurer Hut vor dem Wolf; wenn er hereinkommt, so frisst er euch alle mit Haut und Haar. Der Bösewicht verstellt sich oft, aber an seiner rauen Stimme und seinen schwarzen Füssen werdet ihr ihn gleich erkennen. ...*
>
> („Der Wolf und die sieben Geisslein" aus den Märchen der Gebrüder Grimm)

- *Authentizität* ist essentiell für die Sicherheit eines verteilten Systems
    - zu authentischen Nachrichten / Daten vgl. auch den Begriff "Integrität"
- Authentizität eines *Subjekts (Client)*
    - ist er wirklich der, der er vorgibt zu sein?
    - darf ich als Server daher ihm (?) den Zugriff gewähren?
- Authentizität eines *Dienstes (Server)*
    - Bsp.: Handelt es sich wirklich um den Druckdienst oder um einen böswilligen Dienst, der die Datei ausserdem noch heimlich kopiert?
- Authentizität einer *Nachricht*
    - hat mein Kommunikationspartner dies wirklich so gesagt?
    - soll ich als Geldautomat wirklich so viel Geld ausspucken?
- Authentizität *gespeicherter Daten*
    - ist dies wirklich der Vertragstext, den wir gemeinsam elektronisch hinterlegt haben?
    - hat der Autor Casimir von Hinkelstein wirklich *das* geschrieben?
    - ist das Foto nicht eine Fälschung?
    - ist dieser elektronische Schlüssel wirklich echt?

Hilfsmittel zur Authentifizierung

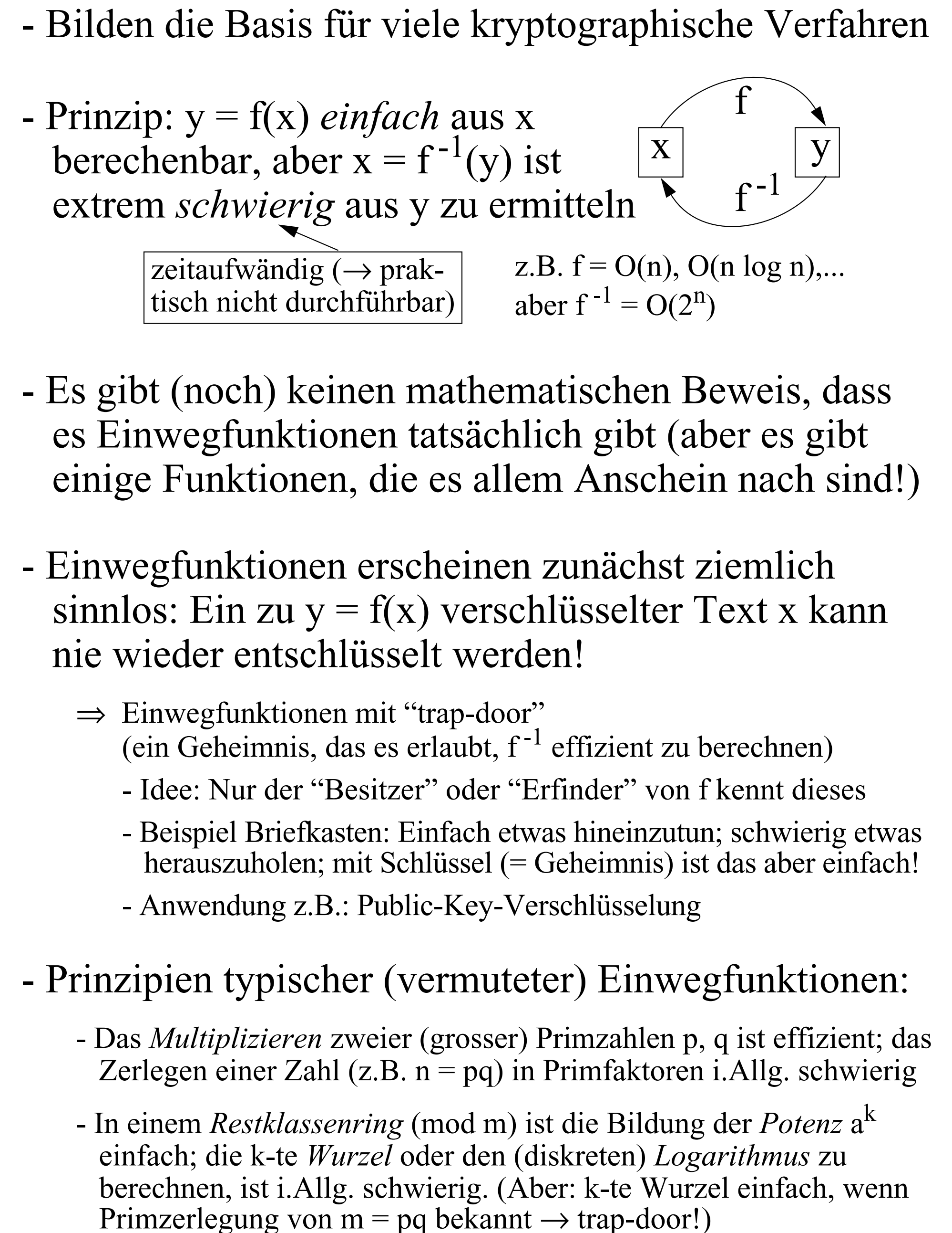
- Wahrung der Nachrichten-Authentizität
  - Verschlüsselung, so dass inhaltliche Änderungen auffallen (Signatur)
  - Fälschung dann nur bei Kenntnis der Verschlüsselungsfunktion möglich
  - Beachte: Authentizität des Nachrichteninhalts garantiert nicht Authentizität der Nachricht als solche! (z.B. Replay-Attacke: Neuversenden einer früher abgehörten Nachricht)
  - Massnahmen gegen Replays: z.B. mitcodierte Sequenznummer

- Peer-Authentifizierung mit *Frage-Antwort-Spiel*
  - "challenge / response": Antworten sollte nur der echte Kommunikationspartner kennen
  - idealerweise stets neue Fragen verwenden (Replay-Attacken!)

- Peer-Authentifizierung mit *Passwort*
  - typischerweise zur Authentifizierung eines Benutzers ("Client") zum Schutz des Dienstes vor unbefugter Benutzung (Autorisierung)
  - Kenntnis des Passworts gilt als Identitätsbeweis (ist das gerechtfertigt?)

- Potentielle *Schwächen von Passwörtern*
  - Geheimhaltung (Benutzer kann Passwörter "verleihen" etc.)
  - Raten oder systematische Suche ("dictionary attack")
    - Zurückweisung zu "simpler" Passwörter
    - Zeitverzögerung nach jedem Fehlversuch
    - security logs
  - Abhörgefahr (kein Passwortaustausch im Klartext; Speicherung des Passworts nur in codierter Form, so dass Invertierung prakt. unmöglich)
  - Replay-Attacke (Gegenmassnahme: Einmalpasswörter)

hierfür geeignet: Einwegfunktionen



Einwegfunktionen

- Bilden die Basis für viele kryptographische Verfahren

- Prinzip: $y = f(x)$ *einfach* aus x berechenbar, aber $x = f^{-1}(y)$ ist extrem *schwierig* aus y zu ermitteln

zeitaufwändig (→ praktisch nicht durchführbar)

z.B. $f = O(n)$, $O(n \log n)$,...
aber $f^{-1} = O(2^n)$

- Es gibt (noch) keinen mathematischen Beweis, dass es Einwegfunktionen tatsächlich gibt (aber es gibt einige Funktionen, die es allem Anschein nach sind!)

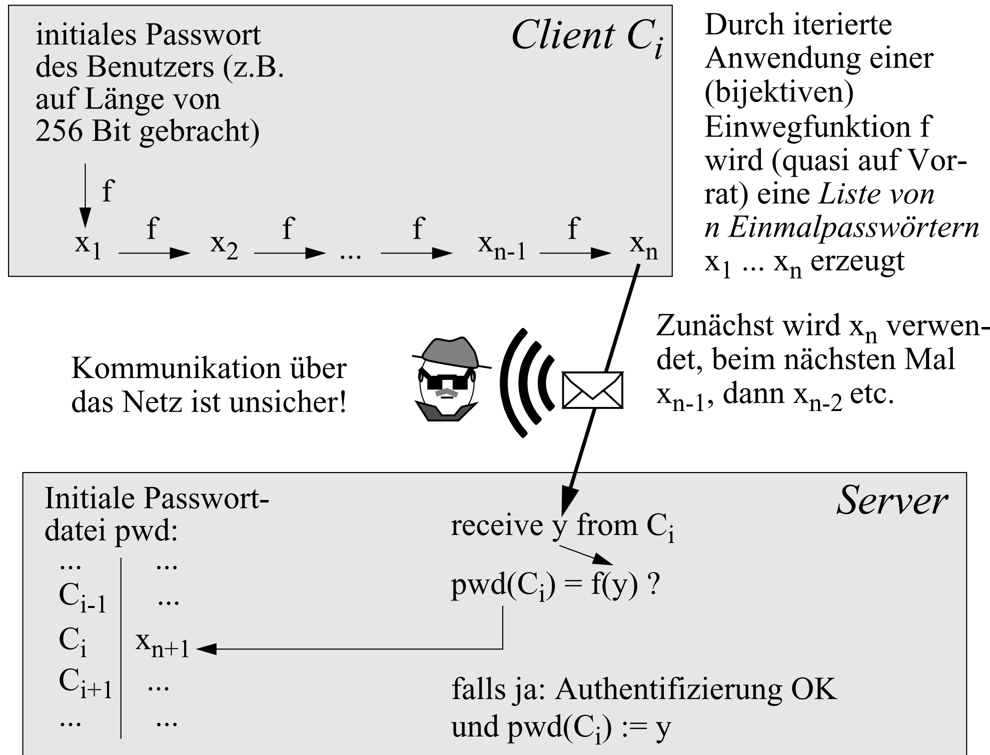
- Einwegfunktionen erscheinen zunächst ziemlich sinnlos: Ein zu $y = f(x)$ verschlüsselter Text x kann nie wieder entschlüsselt werden!

  ⇒ Einwegfunktionen mit "trap-door" (ein Geheimnis, das es erlaubt, f^{-1} effizient zu berechnen)
  - Idee: Nur der "Besitzer" oder "Erfinder" von f kennt dieses
  - Beispiel Briefkasten: Einfach etwas hineinzutun; schwierig etwas herauszuholen; mit Schlüssel (= Geheimnis) ist das aber einfach!
  - Anwendung z.B.: Public-Key-Verschlüsselung

- Prinzipien typischer (vermuteter) Einwegfunktionen:
  - Das *Multiplizieren* zweier (grosser) Primzahlen p, q ist effizient; das Zerlegen einer Zahl (z.B. $n = pq$) in Primfaktoren i.Allg. schwierig
  - In einem *Restklassenring* (mod m) ist die Bildung der *Potenz* a^k einfach; die k-te *Wurzel* oder den (diskreten) *Logarithmus* zu berechnen, ist i.Allg. schwierig. (Aber: k-te Wurzel einfach, wenn Primzerlegung von $m = pq$ bekannt → trap-door!)

Einmalpasswörter mit Einwegfunktionen

- Szenario: Client gehört dem Benutzer (Notebook, Chipkarte...); Passwörter sind dort sicher aufgehoben



Durch iterierte Anwendung einer (bijektiven) Einwegfunktion f wird (quasi auf Vorrat) eine *Liste von n Einmalpasswörtern* $x_1 \dots x_n$ erzeugt

Zunächst wird x_n verwendet, beim nächsten Mal x_{n-1}, dann x_{n-2} etc.

- Ein abgehörtes Passwort x_i nützt nicht viel
  - Berechnung von x_{i-1} aus x_i ist (praktisch) nicht möglich
- Ein Lesen der Passwortdatei des Servers ist nutzlos
  - dort ist nur das *vergangene* Passwort vermerkt
- Einwegfunktion f muss nicht geheimgehalten werden
- Realisiert z.B. im S/KEY-Verfahren (RFC 1760)