Resümee (4)

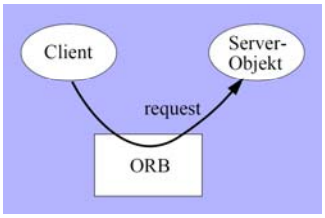
- RPC-Fehlersemantikklassen
  - maybe, at-least-once, at-most-once
- Weiteres zu RPCs
  - asynchroner RPC, call-back, context-handle, broadcast
  - Sicherheitsstufen (Authentifizierung, Verschlüsselung, Signatur)
- Lookup-Service
- Web Services
  - WSDL-Servicebeschreibung (als XML-Dokument)
  - SOAP-Serviceaufruf
- Ressourcen-orientierte Architektur
  - Ressource
  - Repräsentation einer Ressource
  - REST: Representational State Transfer

Middleware-Entwicklung, die Web Services historisch voranging

1. RPC-Bibliotheken (z.B. von SUN für UNIX)
   - Unterstützung des Client-Server-Paradigmas
   - einfache Schnittstellenbeschreibungssprache, Stubgeneratoren
   - Sicherheitskonzepte: Authentifizierung, Autorisierung, Verschlüsselung
2. Client-Server-Verteilungsplattformen (z.B. DCE)
   - Verzeichnisdienst, globaler Namensraum, globales Dateisystem
   - Programmierhilfen: u.a. Synchronisation, Multithreading
3. Objektbasierte Verteilungsplattformen (z.B. CORBA)
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache
   - „Object Request Broker" als Vermittlungsinstanz

CORBA (Version 2.0 ca. ab 1997)

- Common Object Request Broker Architecture
  - ORB (Object Request Broker) als Vermittlungsinfrastruktur (Weiterleitung von Methodenaufrufen etc.)
  - IDL (Interface Description Language) mit Stub-Generatoren
  - Systemfunktionen und Basisdienste in Form von Object Services



Methodenaufruf unterschiedlicher 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)

CORBA

- Ab ca. 2000 entstand der Wunsch nach einer wesentlichen Erweiterung der CORBA-Funktionalität aufgrund
  - neuer Anforderungen durch E-Commerce-Anwendungen
  - Ausbreitung des WWW
  - Aufkommen von Java
  - Aufkommen mobiler Geräte

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

- Allerdings geriet die CORBA-Weiterentwicklung ins Stocken
  - zu weitreichende Anforderungen → komplex / ineffizient
  - kommerzielle Implementierungen der Erweiterungen zögerlich
  - fehlende Unterstützung durch Microsoft (eigene Architektur: DCOM und .NET)
  - man versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse)
  - aufkommende Konkurrenzsysteme (z.B. Web Services), die z.T. direkter und besser an die neuen Anforderungen angepasst waren

Zustandsändernde / -invariante Dienste

- Verändern Aufträge den Zustand des Servers?
  - typische zustandsinvariante Dienste: Auskunftsdienste (z.B. Namensdienst oder Zeitservice)
  - typische zustandsändernde Dienste: Datei-Server
- Idempotente Dienste / Aufträge
  - Wiederholung eines Auftrags führt zum gleichen Effekt
  - Beispiel: „Schreibe in Position 317 von Datei XYZ den Wert W" (ist aber nicht zustandsinvariant!)
  - Gegenbeispiel: „Schreibe ans Ende der Datei XYZ den Wert W"
  - Gegenbeispiel: „Wie spät ist es?" (ist aber zustandsinvariant!)
- Bei Idempotenz oder Zustandsinvarianz kann bei fehlerhaftem Auftrag (timeout beim Client) dieser problemlos erneut abgesetzt werden (→ einfache Fehlertoleranz)

Zustandslose („stateless") / zustandsbehaftete („statefull") Server

- Hält der Server Zustandsinformation über Aufträge hinweg?
  - z.B. (Protokoll)zustand des Clients
  - z.B. Information über frühere damit zusammenhängende Aufträge
- 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.)
  - zustandsbehaftete Server daher i.Allg. effizienter
  - Dateisperren bei echten zustandslosen Servern nicht einfach möglich
- Zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. Speichern von Sequenznummern) → Idempotenz
- Crash eines Servers: Weniger Probleme im zustandslosen Fall

Sind Web Server zustandslos?

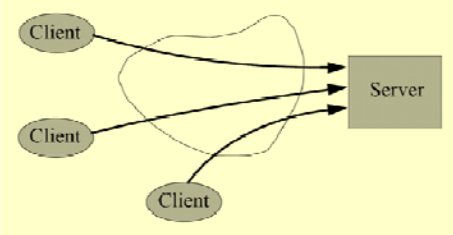
- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - - jeder link, den man anklickt, löst eine neue „Transaktion" aus
- Stellt z.B. ein Problem E-Commerce-Anwendungen dar
  - oft gewünscht: Transaktionen über mehrere Klicks hinweg und
  - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
  - erforderlich z.B. für Realisierung von „Warenkörben" von Kunden

Wiedererkennung von Kunden?

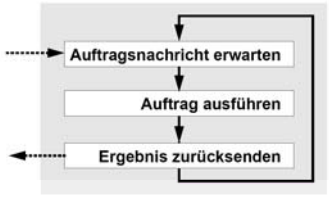
- „URL rewriting" und dynamische Web-Seiten
  - der Einstiegsseite eine eindeutige Identität anheften, wenn der Kunde diese erstmalig aufruft
  - diese Identität jedem link der Seite anheften und mit zurückübertragen
- „Cookie" als Context-Handle
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - der Server kann das Cookie später wieder lesen und damit den Kunden wiedererkennen
- Evtl. auch Wiedererkennung über IP-Adresse
  - aber oft Probleme bei dynamischen IP-Adressen, Proxies etc.

Gleichzeitige Server-Aufträge

- Problem: Oft viele „gleichzeitige" Aufträge



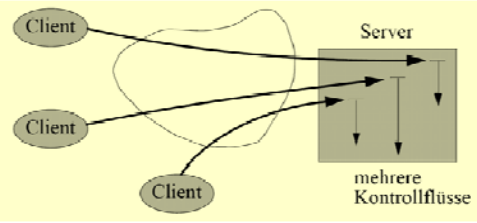
Iterativer Server („single threaded"):



- Iterative Server bearbeiten nur einen einzigen Auftrag pro Zeit
  - eintreffende Anfragen während der Auftragsbearbeitung: abweisen, puffern oder schlichtweg ignorieren
  - einfach zu realisieren
  - bei trivialen Diensten mit kurzer Bearbeitungszeit sinnvoll

Konkurrente („nebenläufige") Server

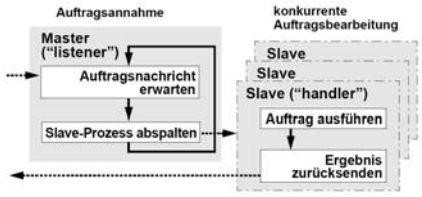
- (Quasi)gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei längeren Aufträgen



Verschiedene Realisierungen
- mehrere Prozessoren bzw. Multicore-Prozessoren
- Verbund verschiedener Server-Maschinen (Cluster)
- dynamische Prozesse oder feste Anzahl vorgegründeter

- Beachte: Auch bei Monoprozessor-Systemen ist Timesharing sinnvoll: Nutzung erzwungener Wartezeiten bei einer Auftrags-bearbeitung für andere Jobs; kürzere mittlere Antwortzeiten bei Jobmix aus langen und kurzen Aufträgen

Konkurrente Server mit dynamischen Handler-Prozessen



Für jeden Auftrag gründet der Master einen neuen Slave-Prozess und wartet dann auf einen neuen Auftrag

Alternative: „Process preallocation": Feste Anzahl statischer Slave-Prozesse (evtl. effizienter, da Wegfall der Erzeugungskosten)

- Neu gegründeter Slave („handler") übernimmt den Auftrag
- Client kommuniziert dann direkt mit dem Slave
- Slaves sind typischerweise Leichtgewichtsprozesse („threads")
- Slaves terminieren i.Allg. nach Beendigung des Auftrags
- Die Anzahl gleichzeitiger Slaves sollte begrenzt werden

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.

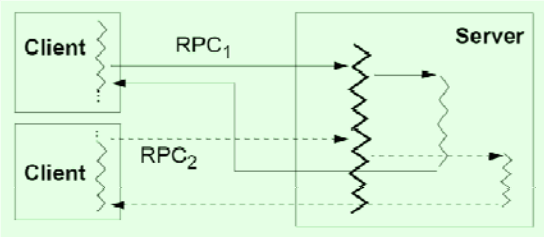
We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment or components thereof that could be interpreted as discriminatory or offensive in nature before such equipment is sold or otherwise provided to any County department.

Thank you in advance for your cooperation and assistance.

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

Multithreading beim Client-Server-Konzept

- 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

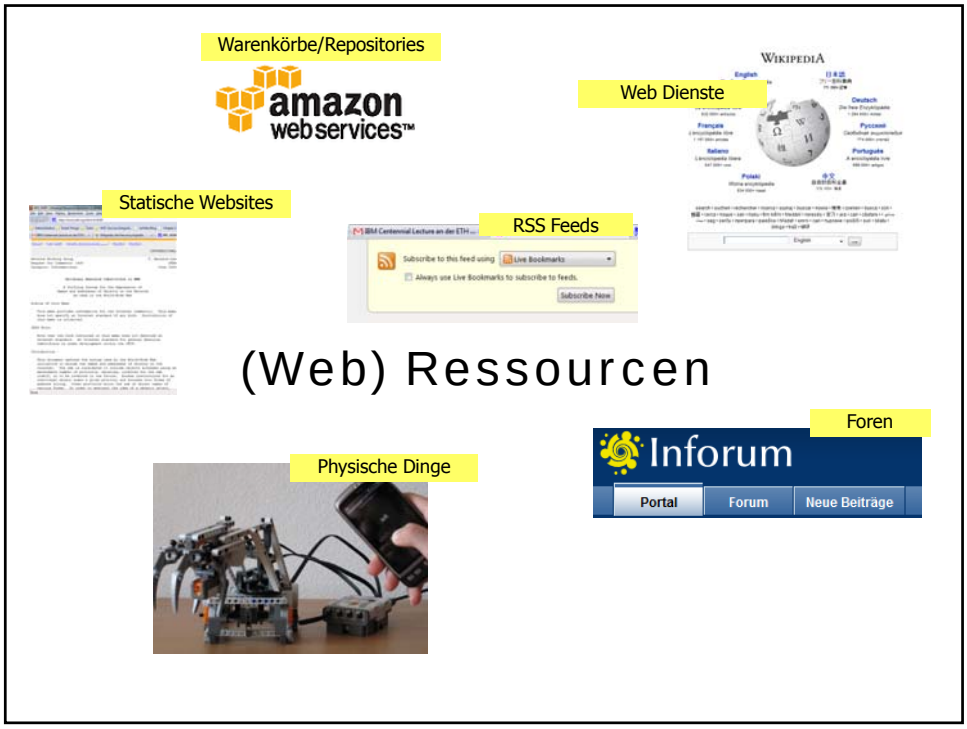
Ressourcen-orientierte Architektur (ROA)

- Funktionalität wird nicht durch Services („SOA"), sondern durch (Web) Ressourcen angeboten
- Ressource? Referent eines Uniform Resource Identifiers
  - RFC 1630 „URL" (1994) — Implizit: „Etwas, das adressiert werden kann"
  - RFC 2396 „URI" (1998)

> *A resource can be anything that has identity. Familiar examples include an electronic document, an image, a service (e.g., "today's weather report for Los Angeles"), and a collection of other resources. Not all resources are network "retrievable"; e.g., human beings, corporations, and bound books in a library can also be considered resources…*

  - RFC 3986 „URI" (2005)

> *…Likewise, abstract concepts can be resources, such as the operators and operands of a mathematical equation, the types of a relationship…*



Repräsentation – Beispiel „Buch“

- Das Buch als abstraktes Konzept
  - es gibt verschiedene Ausgaben, Exemplare etc.
  - identifiziert per ISBN: *URN:ISBN:978-3540002130*
- Was wir ausleihen / kaufen, ist eine Repräsentation des Buches
  - z.B. Hardcover, PDF, E-Book,…
  - auch ein Bild des Covers kann eine Repräsentation sein
  - oder ein maschinenlesbares XML-Dokument für das Bibliothekssystem

REST

- REST (als idealisierte Architektur des Web) steht für
  - Representational: Nicht Ressourcen, sondern deren Repräsentationen werden übertragen
  - State Transfer: Die Übertragung löst Zustandsübergänge aus und verändert damit die Ressourcen
- REST: Entwicklung
  - Zurückführen des Erfolgs des WWW (z.B. bzgl. Skalierbarkeit) auf Eigenschaften der verwendeten Protokolle und Mechanismen: Abstraktion von HTTP und Formulierung einer idealisierten Architektur: REST
  - Mit REST wird versucht, die Möglichkeiten, die das Web (bzw. HTTP) bietet, optimal auszunutzen (Caching; HTTP-Verben wie PUT und DELETE, die Web-Browser nicht kennen,…)

Eigenschaften von REST

- Zustandslosigkeit
  - entschärft Crash-Problematik und Orphans
  - erlaubt Caching und bessere Skalierbarkeit
- Einheitliche, a-priori bekannte, Schnittstelle für alle Ressourcen
  - einheitliche Aufrufe, z.B. GET, POST bei HTTP
  - Adressierung direkt durch URIs
  - selbstbeschreibende Nachrichten: alle benötigten Metadaten sind enthalten, z.B. im HTTP-Header
  - bekannte Repräsentationen, z.B. MIME-Typen

  auch andere Protokolle möglich!
- Bevorzugte Repräsentation wählbar
  - HTML, XML, JSON,...

Entwicklung von REST-Komponenten mit Java-IDE

- JAX-RS: Java API for RESTful Web Services (Beispiel: Eclipse)

ShoppingCartResource.java

```
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.Response;

@Path("/cart/{cartID}")
public class ShoppingCartResource {

    @GET @Produces("text/html")
    public Response getCartContent(
            @PathParam("cartID") String cartID) {

        return Response.ok("Books in your cart: ...").build();
    }

    @POST @Consumes("application/json") @Produces("text/html")
    public Response addBookToCart(
            @PathParam("cartID") String cartID,
            @QueryParam("bookID") String bookID) {

        return Response.created(c.bookURIInCart).build();
    }
}
```

Erweiterung von einfachen Java-Objekten zu Ressourcen per Java Annotations

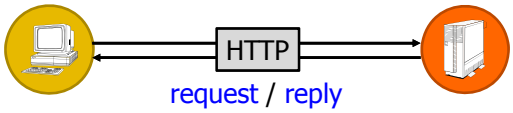
@Path: Pfad zur Ressource

Definition der verwendeten Media Types (@Consumes und @Produces)

Extraktion von Parametern aus dem Request mittels:
@PathParam: z.B. *{cartID}*
@QueryParam: z.B *?bookID=123*

REST-Anwendungsmodell

- „Hypermedia as the Engine of Application State"
  - Client kennt ausschliesslich die Basis-URI des Dienstes
  - Server leitet durch die Anwendungszustände durch Bekanntgabe von Wahlmöglichkeiten (hyperlinks)



  - Der Anwendungszustand kann beim Client oder beim Server liegen, oder auch über beide verteilt sein
  - HTTP-Kommunikationsprotokoll selbst bleibt aber zustandslos

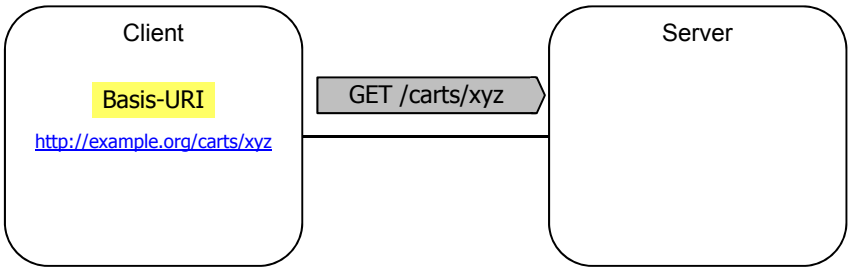
REST: Zustandsspeicherung

- Wenn der Server keinen Zustand hält, dann muss der Client alleine durch den Zustandsraum navigieren
  - Beispiel: man klickt sich durch eine Hypertext-Struktur (und merkt sich die früher besuchten Dokumente bzw. Zustände)
  - Aktueller Zustand = derzeitiges Hypertextdokument beim Client
  - Client und Server sind völlig entkoppelt
  - Dem Server ist es egal (und nicht bewusst), welches Hypertext-Dokument (d.h. Zustand) der Client vorher hatte
  - Der Client übergibt jeweils eine vollständige URI, die den Folgezustand bezeichnet
  - Bookmarks machen Sinn (sind vollständige URI losgelöst von jeglichem Kontext)
  - Back button im Browser macht Sinn: er führt zu einem früheren, im Client gecachten Zustand

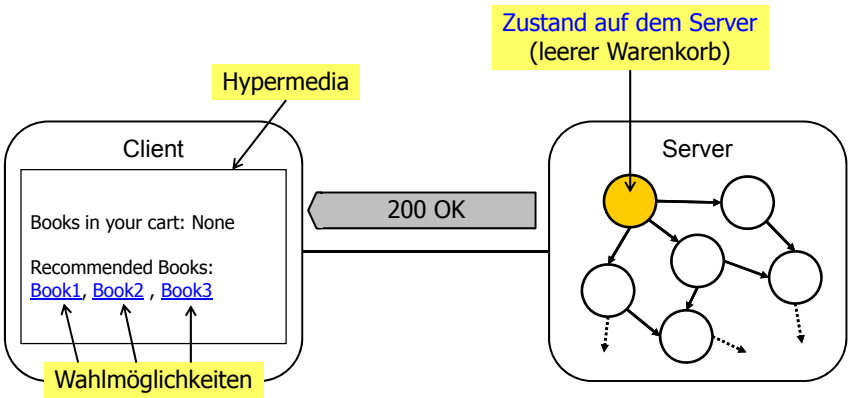
REST: Zustandsspeicherung

- Bei komplexeren Anwendungen residiert typischerweise der Anwendungszustand beim Server
  - z.B. Flugbuchung, wo der Server mit externen Diensten kommuniziert
  - e-shop, wo der Warenkorb beim Server geführt wird
- Dann funktioniert eine Kopie einer URI (bookmark) später meistens nicht, weil dem Server der Kontext dazu fehlt
  - auch back button im Browser ist problematisch: führt zu einer früheren Zustandskopie, ohne dass der Server dies mitbekommt – Client meint fälschlicherweise, in einem gewissen Zustand zu sein, der tatsächliche Zustand wird aber auf dem Server gehalten

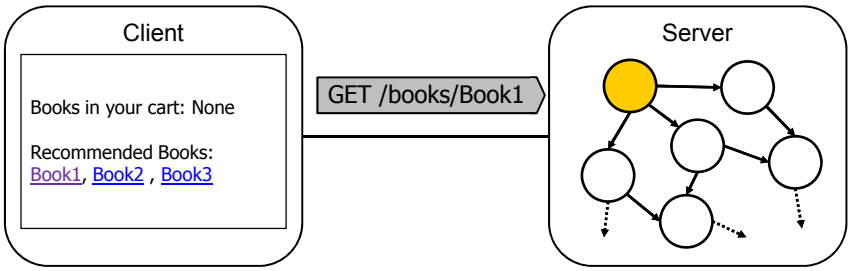
Zustandsspeicherung beim Server



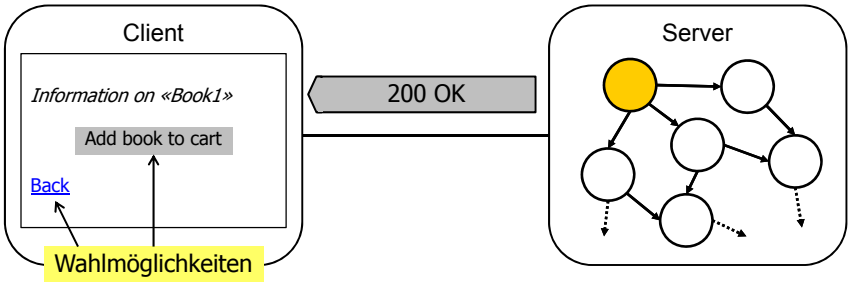
Zustandsspeicherung beim Server



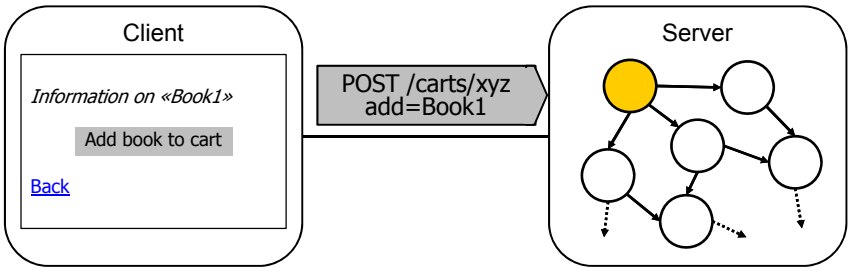
Zustandsspeicherung beim Server



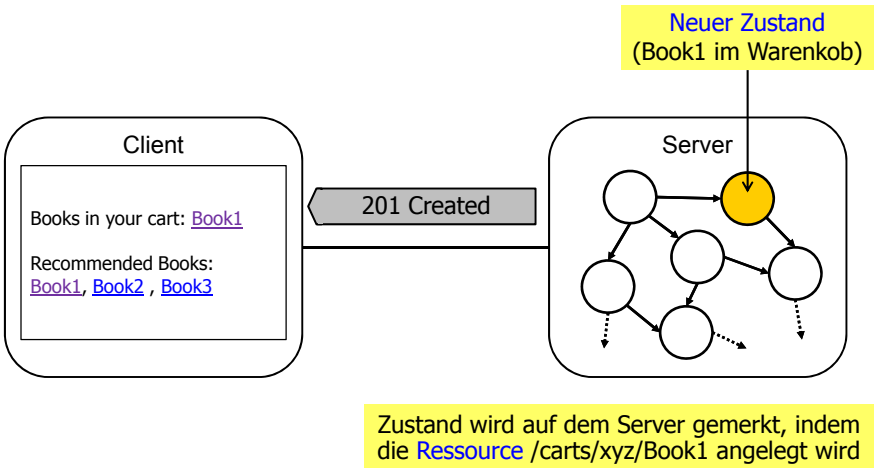
Zustandsspeicherung beim Server



Zustandsspeicherung beim Server



Zustandsspeicherung beim Server



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

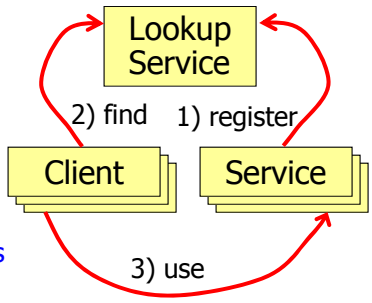
- Infrastructure ("middleware") for dynamic, cooperative, spontaneously networked systems
  - facilitates implementation of distributed applications
- Based on Java
  - uses RMI (Remote Method Invocation)
  - code shipping
  - requires JVM / bytecode everywhere
- Service-oriented
  - (almost) everything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

Service Paradigm

- (Almost) everything relevant is a service
- 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"

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
  - 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

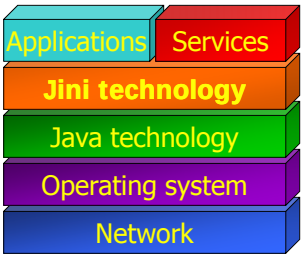
- Objects in an open, distributed, dynamic world find each other and form a transitory community
  - cooperation, service usage, …
- Typical scenario: client wakes up (device is switched on, plugged in, …) and asks for services in its vicinity
- Finding each other and establishing a connection should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

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

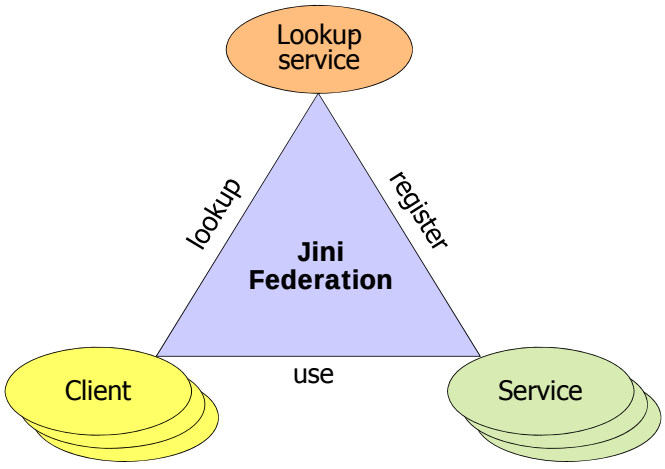
- Jini requires JVM (as bytecode interpreter)
  - homogeneity in a heterogeneous world
- Devices that are not "Jini-enabled" have to be managed by a Jini-enabled software proxy (somewhere in the net)

run protocols for discovery and join; have a JVM

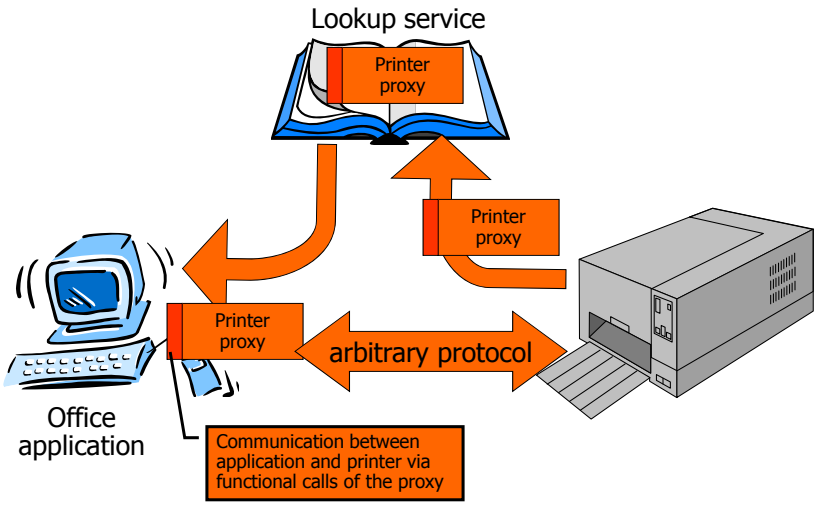
Main Components of the Jini Infrastructure

- Lookup service (LUS)
  - 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

Lookup Service



Example



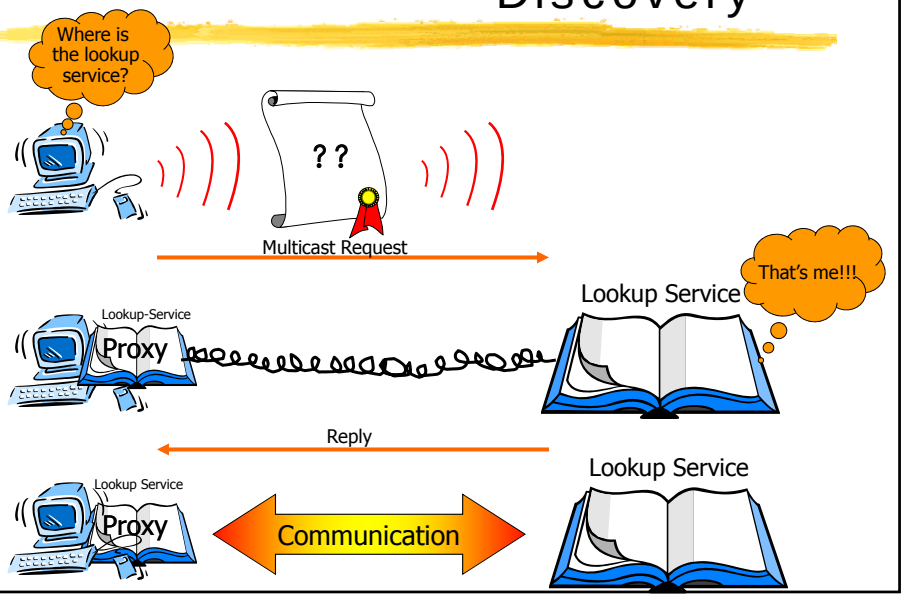
Lookup Service

- Uses Java RMI for communication
  - objects („proxies") can migrate over the network
- Besides the name/address of a service:
  - 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)
    - from then on communication with the LUS is via this proxy

Discovery



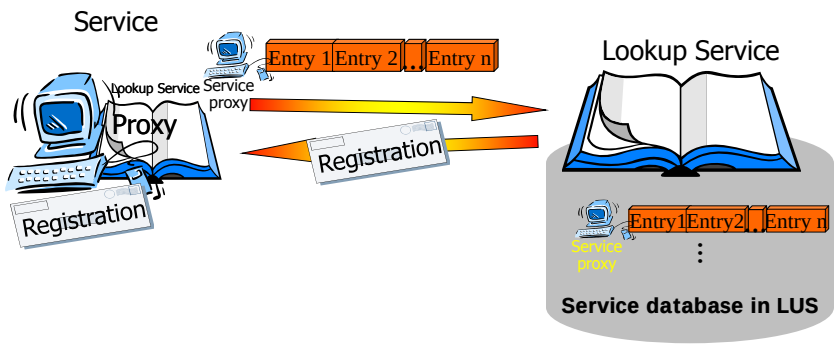
Multicast Discovery Protocol

- Search for lookup services
  - no information about the host network needed
- Discovery request uses multicast UDP packets
  - multicast address for discovery (224.0.1.85)
  - default port number of lookup services (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

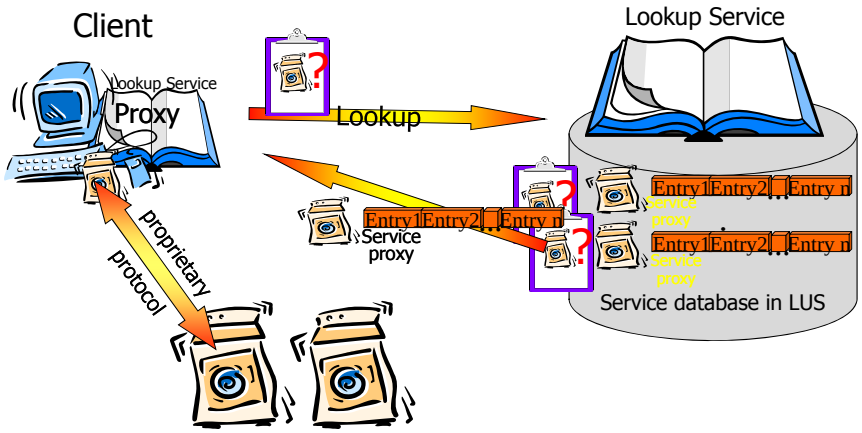
- To join, a service supplies:
  - its proxy
  - its ServiceID (if previously assigned; "universally unique identifier")
  - set of attributes
- Service waits a random amount of time after start-up
  - prevents packet storms after restarting a network segment
- Registration with a lookup service is bound 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
  - 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 the service by calling functions of the service proxy
- Any "private" protocol between service proxy and service provider is possible

Lookup

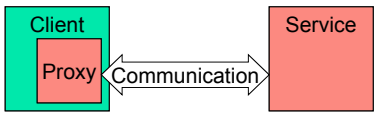


Proxies

- Proxy object is stored in the LUS 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 local 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
  - example: 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

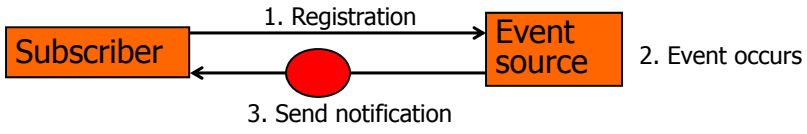


Leases

- 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

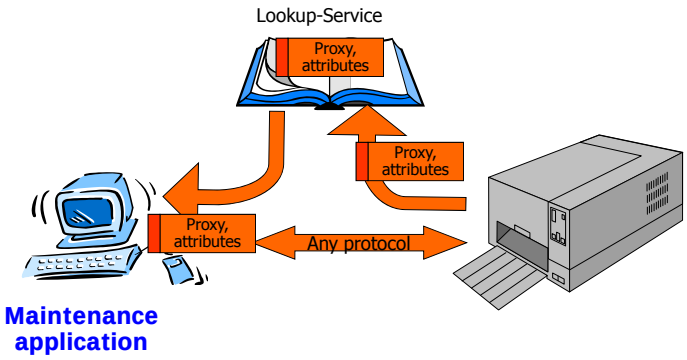
Distributed Events

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



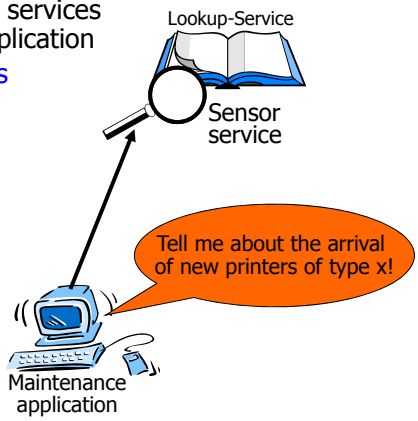
Distributed Events – Example

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



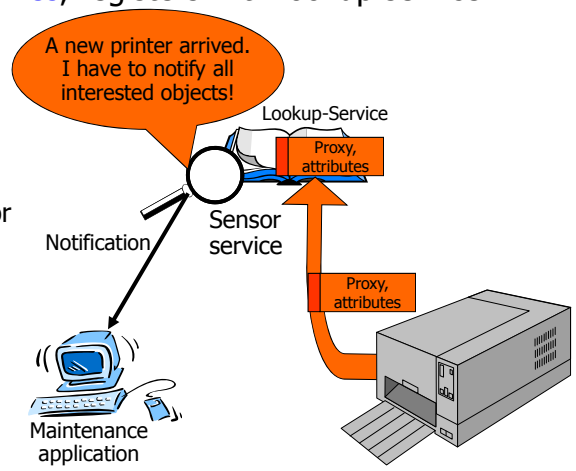
Distributed Events – Example

- Search for printers is "outsourced" to the lookup service
  - "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



Resümee (5)

- Middleware: historischer Kurzüberblick
  - CORBA: Architektur (ORB, IDL), Schicksal der Weiterentwicklung
- Konkurrente Server
  - z.B. dynamische / statische Handler-Prozesse („slaves")
- Zustandsändernde / -invariante Dienste und Server
- Idempotente und wiederholbare Aufträge
- Stateless / statefull Server
- Zustandshaltung bei Webservern
  - URL rewriting, cookies
- Zustandshaltung bei REST
  - Zustand als Hypermedia-Dokument
  - Zustandsspeicherung beim Server vs. zustandsloser Server

Resümee (5b)

- Jini
  - Dienstparadigma, Netzzentrierung
  - Lookup-Service
  - Discovery
  - Join
  - Proxies und smart Proxies
  - Leases
  - verteilte Ereignisse

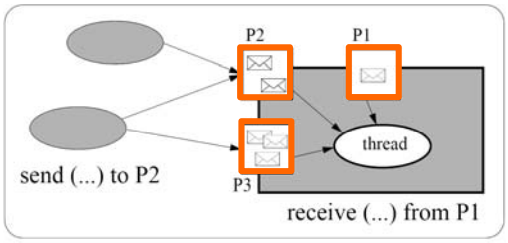
Mehr zu allgemeinen Kommunikationsprinzipien

Im Folgenden:

- Port-Konzept
- Kommunikationskanäle
- Ereigniskanäle
- Timeouts bei der Kommunikation
- Broadcast / Multicast

Das Port-Konzept

- Port = adressierbarer Kommunikationsendpunkt, der die interne Struktur eines Nachrichtenempfänger abkapselt
- Ein Prozess kann mehrere (evtl. typisierte) Ports haben

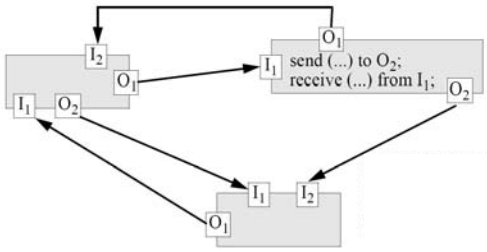


- Manchmal bilden Ports Stauräume („message queues") für Nachrichten
- Manchmal können Ports dynamisch gegründet oder auch geschlossen / geöffnet werden

- Neben Eingangsports („In-Port") sind manchmal auch Ausgangsports („Out-Port") möglich

Kommunikationskanäle

- Kanäle, z.B. eingerichtet mit Ports als Endpunkten
  - dazu je einen In- und Out-Port miteinander verbinden

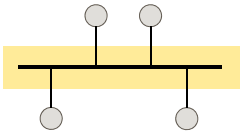


- Alternativ: Kanäle benennen und etwas *auf den Kanal senden* bzw. von ihm lesen
- Evtl. Broadcast-Kanäle: Nachricht geht an alle angeschlossenen Empfänger

- Flexibilität durch Umkonfiguration der Verbindungsstruktur
  - eigentlicher Adressat wird den Prozessen verborgen („virtualisiert")

Ereigniskanäle für autonome Software-Komponenten

- Kooperierende autonome Software-Komponenten
  - nicht notwendigerweise geographisch verteilt
  - mit i.Allg. getrennten Lebenszyklen
  - anonym: kennen nicht die Identität der Anderen
  - Auslösen von „Ereignissen" durch Sender
  - Reagieren auf Ereignisse beim Empfänger
- Ereigniskanal als „Softwarebus"
  - agiert als Zwischeninstanz und verknüpft die Komponenten
  - registriert Interessenten (vgl. LUS)
  - Dispatching eingehender Ereignisse
  - evtl. Puffern, Filtern, Umlenken von Ereignissen

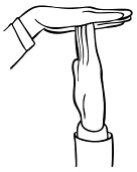


Ereigniskanäle (2)

- Probleme
  - Ereignisse können „jederzeit" ausgelöst werden, werden von Empfängern aber i.Allg. nicht jederzeit entgegengenommen (→ Pufferung?)
  - falls Komponenten nicht lokal, sondern geographisch verteilt sind → „übliche" Probleme nachrichtenbasierter Kommunikation: Verzögerungen, evtl. verlorene Ereignisse, falsche Reihenfolge,…
- Beispiele
  - Microsoft-Komponentenarchitektur (.NET etc.)
  - „Distributed Events" bei JavaBeans und Jini (event generator bzw. remote event listener)

Zeitüberwachter Nachrichtenempfang

- Idee: Receive soll max. eine gewisse Zeit lang blockieren
  - z.B. über Rückgabewert abfragen, ob Kommunikation geklappt hat oder der Timeout zugeschlagen hat
  - Timeout-Wert adäquat setzen (oft schwierig)



- Im Timeout-Fall geeignete Recovery-Massnahmen treffen oder Exception auslösen
- Verwendung bei:
  - Echtzeitprogrammierung
  - Aufheben von Blockaden im Fehlerfall (z.B. bei abgestürztem Kommunikationspartner)
- Timeout evtl. auch beim synchronen (!) Senden sinnvoll

Zeitüberwachter Nachrichtenempfang (2)

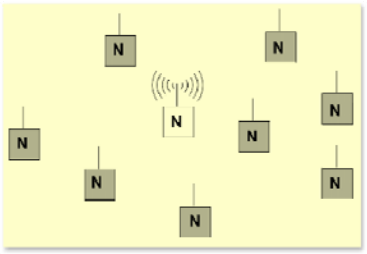
- Sprachliche Einbindung z.B. so:

```
receive ... delay t
   {...}
 else
   {...}
end
```

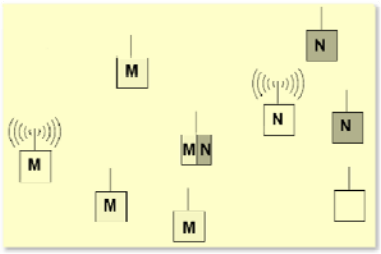
Wird nach mindestens t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen

- Beachte: Es wird mindestens so lange auf Kommunikation gewartet – danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Ablaufs verstreichen
- Was könnte „delay 0" bedeuten? Ist das sinnvoll?

Gruppenkommunikation
(Broadcast / Multicast)



Broadcast: Senden an die Gesamtheit aller Teilnehmer

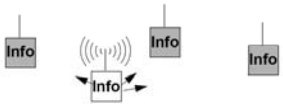


Multicast: Senden an eine Untergruppe aller Teilnehmer

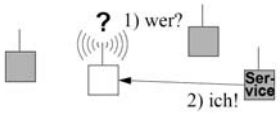
- Multicast entspricht Broadcast bezogen auf die Gruppe
  - verschiedene Gruppen können sich evtl. überlappen
  - jede Gruppe hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

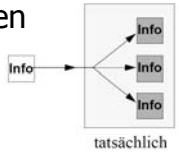
- Informieren
  - z.B. Newsdienste



- Suchen
  - z.B. Finden von Objekten und Diensten



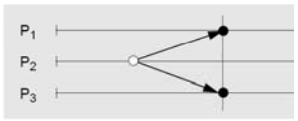
- „Logischer Unicast"
  - an replizierte Komponenten



Typische Anwendungsklasse von Replikation: Fehlertoleranz

Gruppenkommunikation – idealisierte Semantik

1. Modellhaftes Vorbild: Speicherbasierte Kommunikation
   - augenblicklicher „Empfang" in zentralistischen Systemen
   - vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



2. Idealisierte Sicht: Nachrichtenbasierte Kommunikation:
   - gleichzeitiger Empfang
   - vollständige Zuverlässigkeit

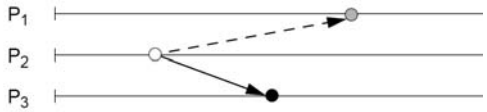
- Beide Ansichten sind aber in vert. Sys. nicht realistisch
  (2. kann in der Realität bzw. bei Vorhandensein einer globalen Zeit durch eine „Sperrfrist" als Zeitpunkt in der Zukunft erreicht werden)

Gruppenkommunikation – tatsächliche Situation

- Zugrundeliegendes Medium (Netz) oft nicht multicastfähig
  - LANs höchstens innerhalb von Teilstrukturen; WLAN als Funknetz a priori anfällig für Übertragungsstörungen
  - multicastfähige Netze sind typischerweise nicht verlässlich (keine Empfangsgarantie)
- Daher typischerweise „Simulation" von Multicast durch ein Protokoll aus vielen Punkt-zu-Punkt-Einzelnachrichten
  - z.B. Multicast-Server, der die Information an alle Empfänger einzeln weiterverteilt

Gruppenkommunikation – tatsächliche Situation (2)

- Nachrichtenkommunikation ist nicht „ideal"
  - nicht-deterministische Zeitverzögerung
    → Empfang zu unterschiedlichen Zeiten
  - keine garantierte Zuverlässigkeit der Nachrichtenübermittlung



- Ziel von Broadcast- / Multicast-Protokollen
  - möglichst gute Approximation der Idealsituation
- Hauptproblem bei der Realisierung von Broadcast:
  1. Zuverlässigkeit
  2. garantierte Empfangsreihenfolge ←

Das soll dem Problem nicht-deterministischer Nachrich-tenverzögerungen begegnen

Typische Fehlerfälle bei der Gruppenkommunikation

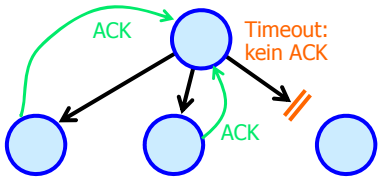
- Während des Protokolls: Verlust von Einzelnachrichten oder Ausfall des Senders
  - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu, Hilfsprozess in der Kommunikationsinfrastruktur abgestürzt,...)
- Problem dann: Empfänger sind sich nicht einig, da manche, aber nicht alle, informiert werden
  - Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein (da wäre es manchmal besser, gar kein Prozess hätte die Nachricht empfangen!)
  - partielle Abhilfe durch aufwändigere Protokolle und mehr Redundanz

„Best Effort“ Broadcast

- Euphemistische Bezeichnung, da keine extra Anstrengung
  - typischerweise einfache Realisierung ohne Ack etc.
- Keinerlei Garantien
  - unbestimmt, wie viele / welche Empfänger erreicht wurden
  - unbestimmte Empfangsreihenfolge
- Allerdings effizient (im Erfolgsfall)
- Geeignet für die Verbreitung unkritischer Informationen
  - z.B. Informationen, die Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
- Evtl. als Grundlage zur Realisierung höherer Protokolle
  - wenn Fehlerfall selten → aufwändiges Recovery auf höherer Ebene tolerierbar

„Reliable“ Broadcast

- Ziel: Unter gewissen (welchen?) Fehlermodellen einen „möglichst zuverlässigen“ Broadcast-Dienst realisieren



- 1. Idee: Quittung („positives Acknowledgement“: ACK) für jede Einzelnachricht
  - im Verlustfall z.B. einzeln nachliefern oder (bei broadcastfähigem Medium) einen zweiten Broadcast-Versuch (→ Duplikaterkennung!)
  - viele ACKs → teuer (d.h., Prinzip skaliert schlecht)

„Reliable“ Broadcast mit negativen Acknowledements („NACK“)

- Alle broadcasts werden vom Sender nummeriert
- Empfänger erkennt evtl. Lücke bei nächstem Empfang
- Bei fehlender Nachrichten wird ein NACK gesendet
- Fehlende Nachricht wird nachgeliefert
  - Sender muss daher Kopien von Nachrichten aufbewahren
  - aber wie lange?
- Broadcast einer „Nullnachricht“ ist evtl. sinnvoll
  - → schnelles Erkennen von Lücken
- Kombination von ACK / NACK mag sinnvoll sein

- Verfahren hilft gegen Verlust von Nachrichten, aber nicht gegen den Crash des Senders „mittendrin“

Ein weiterer Reliable-Broadcast-Algorithmus

- Zweck: Jeder nicht gecrashte und zumindest indirekt erreichbare Prozess soll die Broadcast-Nachricht erhalten
- Voraussetzung: zusammenhängendes „gut vermaschtes“ Punkt-zu-Punkt-Netz

Denkübungen:
- Bidirektionale Kommunikationskanäle notwendig?
- Wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
- Wie fehlertolerant? (Wie viel darf kaputt sein / verloren gehen...?)

Sender s: Realisierung von broadcast(N)

– *send(N, s, sequ_num)* an alle Nachbarn (inklusive an s selber);

Denkübung: Wieso? Ist das notwendig?

– sequ_num ++

Beachte: receive ≠ deliver (unterscheide Anwendungsebene und Transportebene)

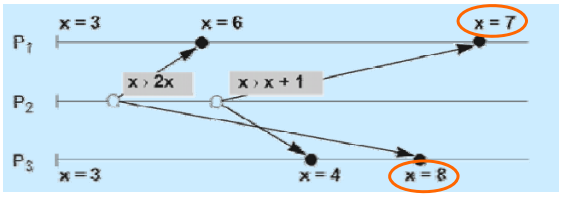
Empfänger r: Realisierung des Nachrichtenempfangs

– *receive(N, s, sequ_num);*
wenn r noch kein *deliver(N)* für sequ_num ausgeführt hat, *dann*:
wenn r ≠ s *dann* *send(N, s, sequ_num)* an alle Nachbarn von r ;
Nachricht an die Anwendungsebene ausliefern („*deliver(N)*“) ;

Prinzip: „Fluten“ des Netzes (→ Vorlesung „Verteilte Algorithmen“)

Broadcast: Empfangsreihenfolge

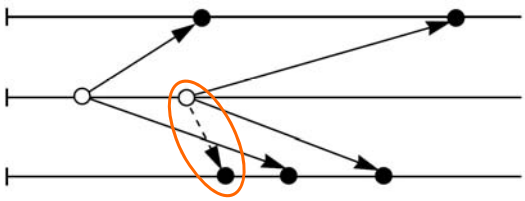
- Wie ist die Empfangsreihenfolge von Gruppennachrichten?
  - problematisch wegen der i.Allg. ungleichen Übermittlungszeiten
  - Bsp.: Update einer replizierten Variablen mittels „function shipping“:



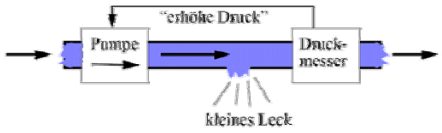
- Es sind verschiedene Grade des Ordnungserhalts denkbar
  - z.B. keine (ungeordnet), FIFO, kausal geordnet, total geordnet

FIFO-Ordnung bei Multicast

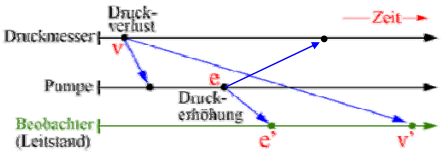
- Alle Broadcast-Nachrichten eines (d.h.: ein und des selben) Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an
  - Denkübung: wie dies in einem Protokoll garantieren?



Probleme mit FIFO-Broadcasts



Annahme: Steuerelemente kommunizieren über FIFO-Broadcasts:



Falsche Schlussfolgerung des Beobachters:

„Aufgrund einer unbegründeten Pumpenaktivität wurde ein Leck erzeugt, wodurch schliesslich der Druck absank."

„Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

Man sieht also:

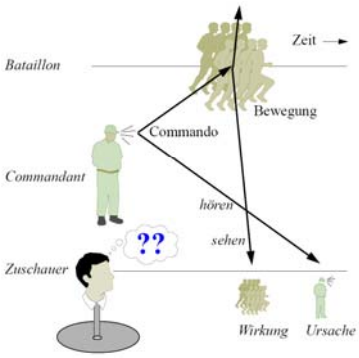
- FIFO-Reihenfolge nicht immer ausreichend, um Semantik zu wahren
- eine Nachricht verursacht oft das Senden einer anderen (→ Kausalität!)

Das „Broadcastproblem" ist nicht neu

wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, ehe er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der Causalzusammenhänge weiss er, dass die Bewegungen die Wirkung des gehörten Commandos sind, dieses also jenen objectiv vorangehen muss, und er wird sich sofort der Täuschung bewusst, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.

Das „Broadcastproblem“ ist nicht neu

- Natürlicher Broadcast bei Licht- und Schallwellen
  - wann handelt es sich dabei um FIFO-Broadcasts?
  - wie ist es mit dem Kausalitätserhalt?

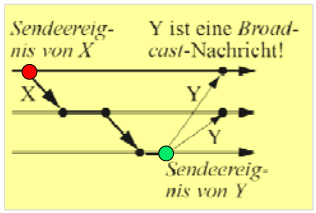


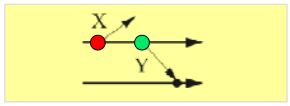
„Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, ehe er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der Causalzusammenhänge weiss er, dass die Bewegungen die Wirkung des gehörten Commandos sind, dieses also jenen objectiv vorangehen muss, und er wird sich sofort der Täuschung bewusst, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.“

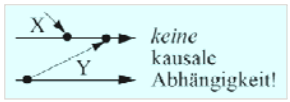
Christoph von Sigwart (1830-1904): *Logik. Zweiter Band. Die Methodenlehre* (1878)

Kausale Nachrichtenabhängigkeit

- *Definition:* Nachricht Y hängt kausal von Nachricht X ab, wenn es im Raum-Zeit-Diagramm einen von links nach rechts verlaufenden Pfad gibt, der vom Sendeereignis von X zum Sendeereignis von Y führt.





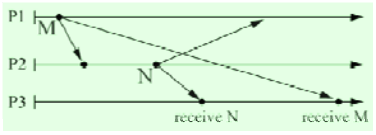


- *Beachte:* Dies lässt sich bei geeigneter Modellierung auch abstrakter fassen (→ „logische Zeit" später in der Vorlesung und auch Vorlesung „Verteilte Algorithmen")

Kausaler Broadcast

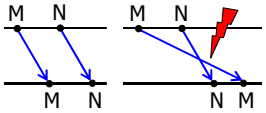
Zweck: Wahrung von Kausalität bei der Kommunikation

- *Definition:* Kausale Reihenfolge („causal order"): Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben.
- „Kausale Reihenfolge" (bzw. „kausale Abhängigkeit") lassen sich insbesondere auch auf Broadcasts anwenden



Gegenbeispiel: *Keine* kausalen Broadcasts!

- Kausale Reihenfolge impliziert FIFO-Reihenfolge: kausale Reihenfolge ist eine Art „globales FIFO"



Das Erzwingen der kausalen Reihenfolge ist mittels geeigneter Algorithmen möglich (→ Vorlesung „Verteilte Algorithmen", z.B. Verallgemeinerung der Sequenzzählermethode für FIFO)

Resümee (6a)

- Kommunikationskonzepte
  - Ports; Kanäle; Ereigniskanäle als „Softwarebus"
  - Timeouts beim Empfangen von Nachrichten
- Übersicht Gruppenkommunikation (Broadcast / Multicast)
  - Anwendungen
  - idealisierte Sicht
  - Fehlerproblematik
  - Vorbeugung gegen Fehler mit ACK, NACK
- Algorithmus für „reliable Broadcast"
  - Redundanz („Fluten" des Netzes) zur Erhöhung der Fehlertoleranz
  - Effizienz / Kosten (Zahl von Einzelnachrichten)

Resümee (6b)

- FIFO-Broadcasts
  - zwei nacheinander ausgeführte Broadcasts ein und desselben Senders erreichen alle Empfänger in dieser Reihenfolge
  - nicht stark genug, um „akausale" Beobachtungen zu verhindern
- Kausale Broadcasts
  - kausale Abhängigkeit zweier Nachrichten
  - „causal order": Nachrichtenempfang „respektiert" kausale Abhängigkeit von Nachrichten (ist also „kausaltreu")
  - „globalisiertes" FIFO