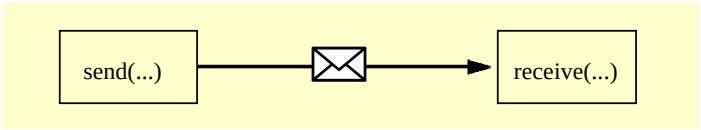
Mitteilungsorientierte Kommunikation



- Einfachste Form der Kommunikation
- Unidirektional
- Übermittelte Werte werden der Nachricht typw. als „Ausgabeparameter“ beim send (-API) übergeben

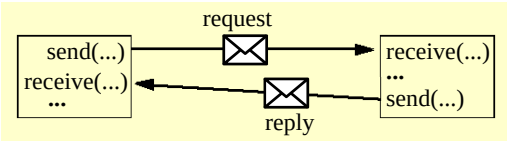
---

- Die Nachricht ist typischerweise eine Zeit lang unterwegs, bevor sie beim Empfängerprozess ankommt
- Sendprozess kann nach dem Absenden aber i. Allg. direkt weiterarbeiten

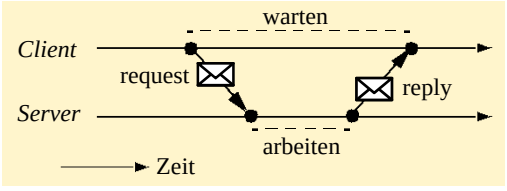
Asynchrone Kommunikation

Auftragsorientierte Kommunikation

Send und receive evtl. zu einem *einzigen* API zusammenfassen



- Bidirektional
- Ergebnis des Auftrags wird als „Antwortnachricht“ zurückgeschickt
- Auftraggeber („Client“) wartet, bis Antwort eintrifft

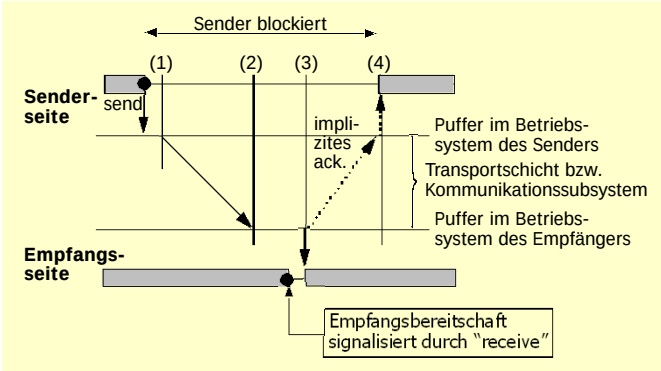


Blockierendes Senden

- Blocking send: Sender ist bis zum Abschluss der Nachrichtentransaktion blockiert

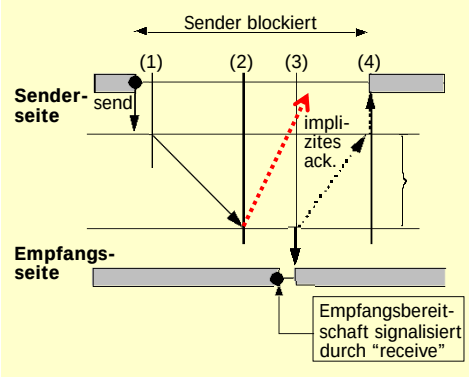
  was genau ist das?

  - Sender hat eine Garantie (Nachricht wurde zugestellt / empfangen)



Blockierendes Senden (2)

- Verschiedene Ansichten einer adäquaten Definition von „Abschluss der Transaktion“ aus Sendersicht:



Zeitpunkt 4 (automatische Bestätigung bei (3), dass der Empfänger receive ausgeführt hat) ist die höhere, anwendungsorientierte Sicht.

Falls eine Bestätigung bereits zum Zeitpunkt 2 geschickt wird, weiss der Sender nur, dass die Nachricht am Zielort zur Verfügung steht und der Sendepuffer wieder frei ist.

(Vorher sollte der Sendepuffer nicht überschrieben werden, weil die Nachricht bei fehlerhafter Übertragung evtl. wiederholt werden muss.)

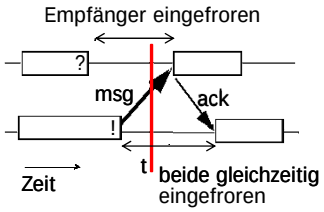
„**gleich**“ „**zeitig**“

Synchrone Kommunikation

- Idealisierung:
  Send und receive geschehen („im Prinzip“) gleichzeitig
- Wodurch ist diese Idealisierung gerechtfertigt?
  (kann man auch mit einer Marssonde synchron kommunizieren?)

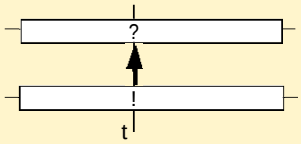
Synchrone Kommunikation mit „blocking send“ implementiert

a) „Receiver first“-Fall:



Bemerkung: „receive“ ist i.Allg. immer blockierend (d.h. Empfänger wartet so lange, bis eine Nachricht ankommt)

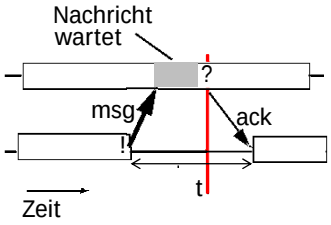
Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet („!“) und empfangen („?“) worden!

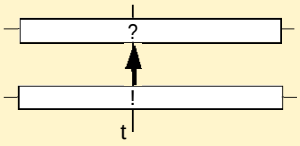
Synchrone Kommunikation mit „blocking send“ implementiert (2)

b) „Sender first“-Fall:



Der Sender ist eingefroren, seine Zeit steht quasi still → es gibt einen gemeinsamen Zeitpunkt t, wo die beiden Kommunikationspartner sich treffen → „Rendezvous“

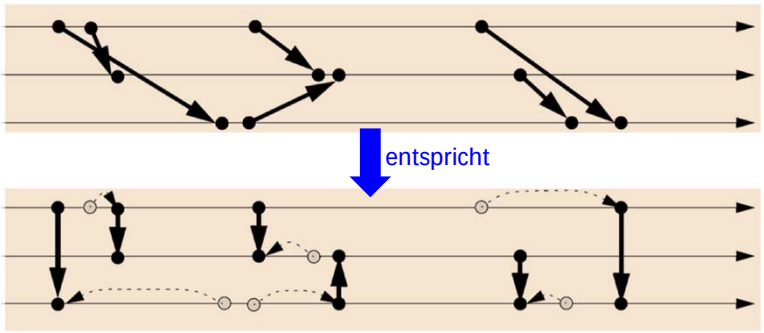
Idealisierung:
senkrechte Pfeile in den Zeitdiagrammen



Als wäre die Nachricht zum Zeitpunkt t gleichzeitig gesendet („!“) und empfangen („?“) worden!

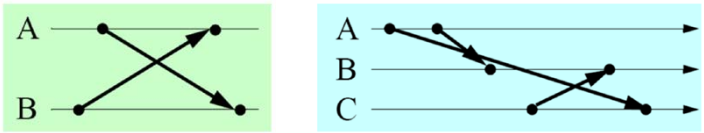
Virtuelle Gleichzeitigkeit

- Ein Ablauf, der synchrone Kommunikation benutzt, ist (bei Abstraktion von der Realzeit) durch ein äquivalentes Zeitdiagramm darstellbar, bei dem alle Nachrichtenpfeile senkrecht verlaufen
  - nur stetige Deformation erlaubt („Gummiband-Transformation“)



Virtuelle Gleichzeitigkeit?

- Folgendes geht nicht virtuell gleichzeitig (wieso?)



- Aber was geschieht eigentlich, wenn man mit synchronen Kommunikationskonstrukten so programmiert, dass dies provoziert wird?

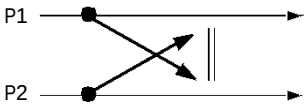
Mehr dazu (nur) für besonders Interessierte: B. Charron-Bost, F. Mattern, G. Tel: S*ynchronous, Asynchronous and Causally Ordered Communication.* Distributed Computing 9(4), pp. 173-191, www.vs.inf.ethz.ch/publ/

Deadlocks bei synchroner Kommunikation

```
P1:
    send (...) to P2;
    receive...
    ...
```

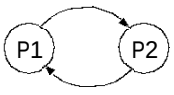
```
P2:
    send (...) to P1;
    receive...
    ...
```

In beiden Prozessen muss zunächst das send ganz ausgeführt werden, bevor es zu einem receive kommt



⇒ Kommunikationsdeadlock!

Zyklische Abhängigkeit der Prozesse voneinander: P1 wartet auf P2, und P2 wartet auf P1



„Wait-for-Graph“

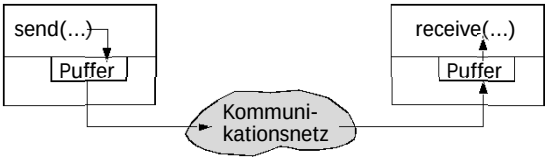
Genauso tödlich: 


```
P1:
    send (...) to P1;
    receive...
    ...
```

Gleichnishaft entspricht der *synchronen* Kommunikation das Telefonieren, der *asynchronen* Kommunikation der Briefwechsel

Asynchrone Kommunikation

- No-wait send: Sender ist nur (kurz) bis zur lokalen Ablieferung der Nachricht an das Transportsystem blockiert
  - diese kurzzeitige Blockade sollten für die Anwendung transparent sein
- Jedoch i.Allg. länger blockiert, falls das System momentan keinen Pufferplatz für die Nachricht frei hat
  - Alternative: Sendenden Prozess dann nicht länger blockieren, sondern mittels „return value“ über Misserfolg des send informieren



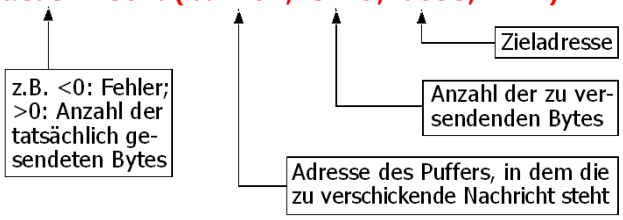
Asynchrone ↔ synchrone Kommunikation

- Vorteile asynchroner Kommunikation (im Vgl. zur syn. Komm.):
  - sendender Prozess kann weiterarbeiten, noch während die Nachricht übertragen wird
  - stärkere Entkoppelung von Sender / Empfänger
  - höherer Grad an Parallelität möglich
  - geringere Gefahr von Kommunikationsdeadlocks
- Nachteile
  - Sender weiss nicht, ob / wann Nachricht angekommen ist
  - Debugging der Anwendung oft schwierig (wieso?)
  - System muss Puffer verwalten

Sendeoperationen in der Praxis

- Es gibt Kommunikationsbibliotheken, deren Dienste von verschiedenen Programmiersprachen (z.B. C) aus aufgerufen werden können
  - z.B. MPI (Message Passing Interface) – Quasi-Standard; verfügbar auf diversen Plattformen
- Typischer Aufruf einer solchen Send-Operation:

```
status = send(buffer, size, dest, ...)
```



Sendeoperationen in der Praxis (2)

- Derartige Systeme bieten im Allgemeinen mehrere verschiedene Typen von Send-Operation an
  - Zweck: Hohe Effizienz durch möglichst spezifische Operationen
  - Achtung: Spezifische Operation kann in anderen Situationen u.U. eine falsche oder unbeabsichtigte Wirkung haben (z.B. wenn vorausgesetzt wird, dass der Empfänger schon im receive wartet)
  - Vorsicht: Semantik und Kontext der Anwendbarkeit ist oft nur informell beschrieben

Synchron $\stackrel{?}{=}$ blockierend

- Kommunikationsbibliotheken machen oft einen Unterschied zwischen synchronem und blockierendem Senden
  - bzw. analog zwischen asynchron und nicht-blockierend

leider etwas verwirrend!

- **Blockierung** ist dann ein rein senderseitiger Aspekt
  - blockierend: Sender wartet, bis die Nachricht lokal vom Kommunikationssystem abgenommen wurde (und der Puffer wieder frei ist)
  - nicht-blockierend: Sender informiert Kommunikationssystem lediglich, wo bzw. dass es eine zu versendende Nachricht gibt (Gefahr des Überschreibens des Puffers bevor dieser frei ist!)
- **Synchron / asynchron** nimmt Bezug auf den Empfänger
  - synchron: Nach Ende der Send-Operation wurde die Nachricht dem Empfänger zugestellt (asynchron: dies ist nicht garantiert)

Nicht-blockierend

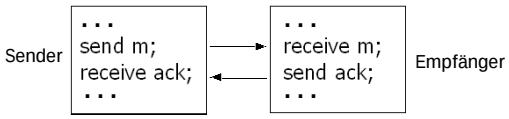
- Nicht-blockierende Operationen liefern oft einen „handle“

  ```
  handle = send(...)
  ```

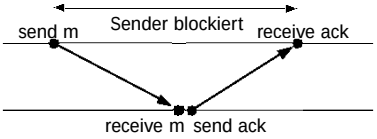
  - dieser kann in Test- bzw. Warteoperationen verwendet werden
  - z.B. Test, ob Send-Operation beendet:
    `if msgdone(handle)…`
  - oder z.B. warten auf Beendigung der Send-Operation:
    `msgwait(handle)`
- Nicht-blockierend ist oft effizienter, aber evtl. unsicherer und umständlicher (evtl. Test; warten) als blockierend

Dualität der Kommunikationsmodelle

Synchrone Kommunikation lässt sich mit asynchroner Kommunikation nachbilden:



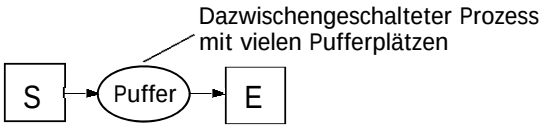
- Warten auf explizites Acknowledgment im Sender direkt nach dem send (receive wird als blockierend vorausgesetzt)
- Explizites Versenden des Acknowledgments durch den Empfänger direkt nach dem receive



Dualität der Kommunikationsmodelle (2)

Asynchrone Kommunikation mittels synchroner:

Idee: Zusätzlichen Prozess vorsehen, der für die Zwischenpufferung aller Nachrichten sorgt



→ Entkoppelung von Sender und Empfänger

Wie realisiert man einen Puffer-prozess bei syn. Kommunikation?

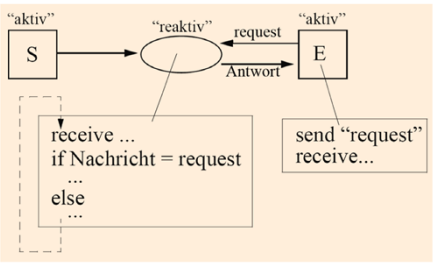
.....
receive - *warte,* bis Sender S etwas schickt; dann:
Nachricht im lokalen Speicher ablegen
?
send - *warte,* bis Empfänger E bereit ist
.....

- Dilemma: Was tut der Pufferprozess nach dem Ablegen der Nachricht im lokalen Speicher?
  - wieder im receive auf den Sender warten, *oder*
  - in einem (blocking) send auf den Empfänger warten?
- Entweder Sender S oder Empfänger E könnte unnötigerweise blockiert sein!

Bemerkung: Puffer der Gösse 1 lassen sich so realisieren → Kaskadier-ung im Prinzip möglich („Pufferpipeline“)

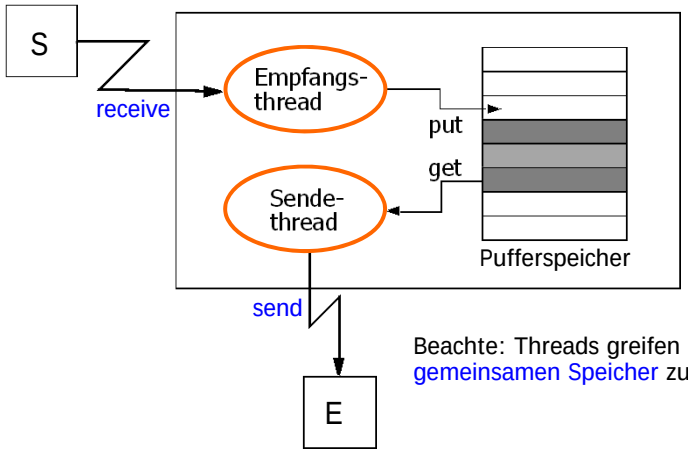
1. Lösung: Puffer als Server!

- E schickt „seinem“ Puffer einen „request“ (und muss dazu die Adresse des Puffers kennen)
- Puffer schickt E keine Antwort, wenn er leer ist
- Empfänger E wird nur dann verzögert, wenn Puffer leer
- Für Sender S ändert sich nichts



- Was tun bei vollem Puffer?
  → Dann sollte der Puffer keine Nachricht von S (wohl aber von E!) annehmen (Denkübung: wie programmiert man das?)

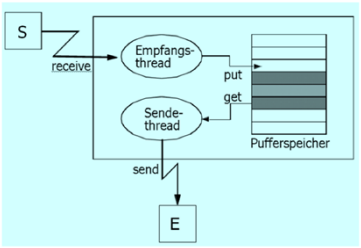
2. Lösung: Puffer als Multithread-Objekt



Beachte: Threads greifen auf gemeinsamen Speicher zu

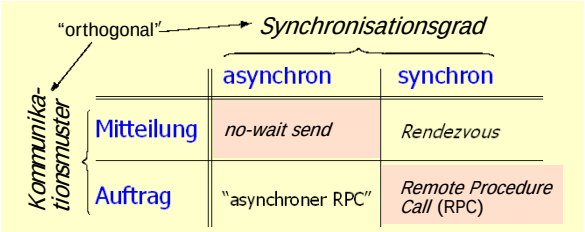
Puffer als Multithread-Objekt (2)

- Empfangsthread ist (fast) immer empfangsbereit
  - nur kurzzeitig anderweitig beschäftigt (put in lokalen Pufferspeicher)
  - evtl. nicht empfangsbereit, wenn lokaler Pufferspeicher voll
- Sendethread ist (fast) immer sendebereit
- Pufferspeicher ist i.Allg. zyklisch organisiert (→ FIFO)
- Pufferspeicher liegt im gemeinsamen Adressraum



- ⇒ Synchronisation der beiden Threads notwendig!
  - konkurrentes / paralleles Programmieren
  - klassische Themen der Betriebssystemtheorie

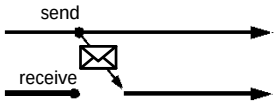
Hauptklassifikation von Kommunikationsmechanismen



- Häufigste Kombination:
  Mitteilung asynchron, Auftrag hingegen synchron
- Wir schauen uns nun der Reihe nach die 4 Kombinationen an

No-Wait Send

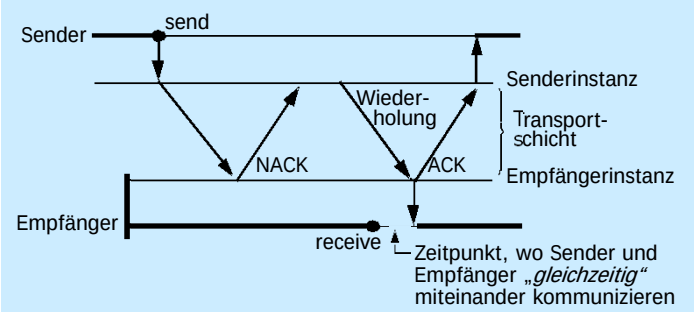
- Asynchron-mitteilungsorientierte Kommunikation



- Vorteile
  - weitgehende zeitliche Entkopplung von Sender und Empfänger
  - einfache, effiziente Implementierung (bei kurzen Nachrichten)
- Nachteile
  - keine Erfolgsgarantie für den Sender
  - Notwendigkeit der Zwischenpufferung (Kopieraufwand, Speicher-verwaltung,...) im Unterschied etwa zur synchronen Kommunikation
  - Gefahr des „Überrennens“ des Empfängers bei zu häufigen Nachrichten → Flusssteuerung („flow control“) notwendig

Rendezvous-Protokolle

- Synchron-mitteilungsorientierte Kommunikation

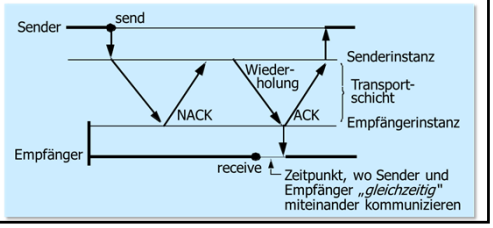


- Hier beispielhaft „Sender-first-Szenario“:
  Sender wartet als Erster („Receiver-first-Szenario“ analog)

Rendezvous-Protokolle (2)

- Rendezvous: Der erste wartet auf den anderen („Synchronisationspunkt“)

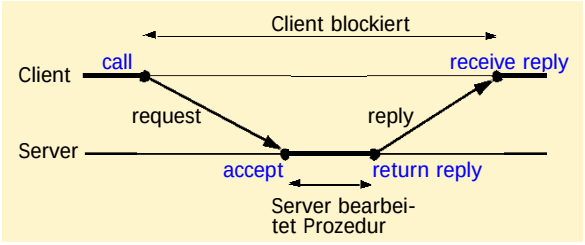
- Mit NACK / ACK (vgl. Bild) sind wenig Puffer nötig
  → aber aufwändiges Protokoll („busy waiting“)
  - Alternative 1: Statt NACK („negative ACK“): Nachricht auf Empfängerseite puffern (Nachteil: Platzbedarf für Puffer)
  - Alternative 2: Statt laufendem Wiederholungsversuch: Empfängerinstanz meldet sich bei Senderinstanz, sobald Empfänger bereit
- Insbesondere bei langen Nachrichten sinnvoll: Vorherige Anfrage, ob bei der Empfängerinstanz genügend Pufferplatz vorhanden ist, bzw. ob Empfänger bereits Synchronisationspunkt erreicht hat



Remote Procedure Call (RPC)

- Aufruf einer „entfernten Prozedur“
- Synchron-auftragsorientiertiertes Prinzip

Keine Parallelität zwischen Client und Server



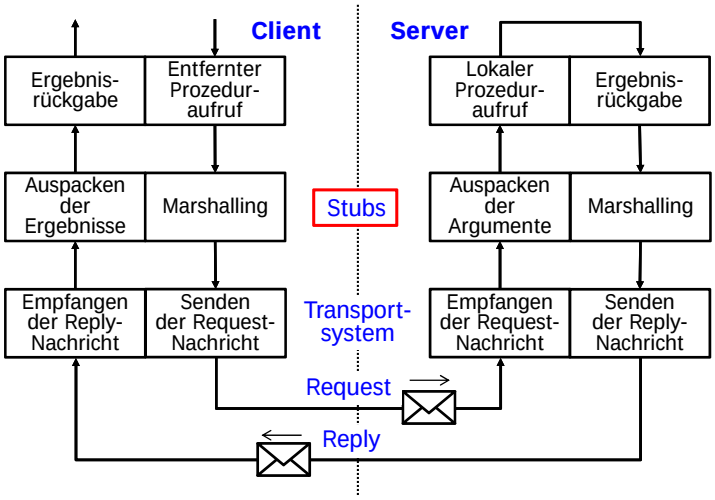
accept, return reply, receive reply etc. werden „unsichtbar“ durch Compiler bzw. Laufzeit-system erledigt

- Bei verteilten objektorientierten Systemen statt dessen: „Remote Method Invocation“ (RMI), z.B. Java RMI

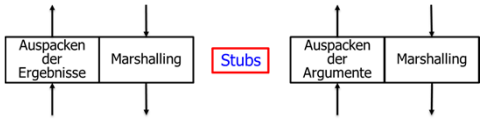
RPC: Pragmatik

- Soll dem klassischen Prozeduraufruf möglichst gleichen
  - klare Semantik für den Anwender (Auftrag als „Unterprogramm“)
- Einfaches Programmieren
  - kein Erstellen von Nachrichten, kein Quittieren etc. auf Anwendungsebene
  - Syntax analog zu traditionellem lokalen Prozeduraufruf
  - Analoge Verwendung lokaler / entfernter Prozeduren
  - Typsicherheit (Datentypprüfung auf Client- und Serverseite möglich)
- Implementierungsproblem: „Verteilungstransparenz“
  - Verteiltheit so gut wie möglich verbergen

RPC: Implementierung



RPC: Stubs



- Stub = Stummel, Stumpf

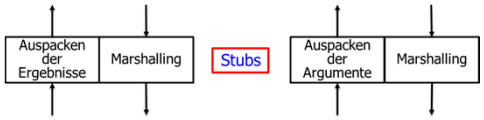
Client:

```
xxx ;
call S.X(out: a ; in: b);
xxx ;
```

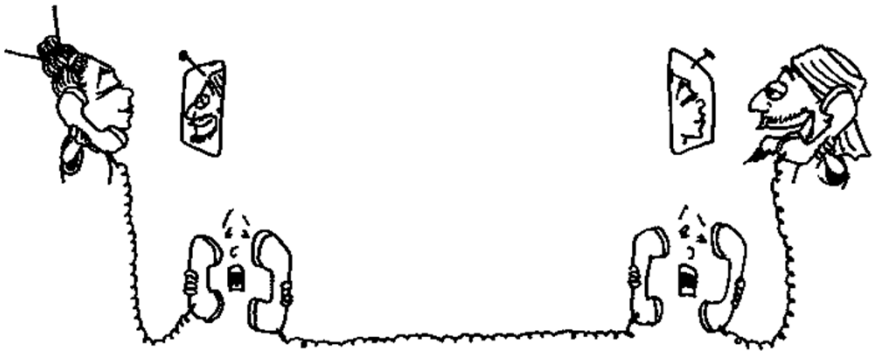
Ersetzt durch ein längeres Programmstück (*Client-Stub*), welches u.a.:

- Parameter a in eine Nachricht packt
- Nachricht an Server S versendet
- Timeout für die Antwort setzt
- Antwort entgegennimmt (oder evtl. exception bei timeout auslöst)
- Ergebnisparameter b mit den Werten der Antwortnachricht setzt

RPC: Stubs (2)



- Wirken als „proxy“
  - lokale Stellvertreter des entfernten Gegenübers
  - (Client-Stub bzw. Server-Stub)
- Simulieren einen lokalen Aufruf
- Sorgen für Zusammenbau und Entpacken von Nachrichten
- Konvertieren Datenrepräsentationen
  - bei heterogenen Umgebungen
- Können oft weitgehend automatisch generiert werden
  - z.B. aus dem Client- oder Server-Code sowie evtl. einer Schnittstellenbeschreibung
- Steuern das Übertragungsprotokoll
  - z.B. zur Behebung von Übertragungsfehlern



„Kommunikation mit Proxies“ (Bild aus dem Buch: „Java ist auch eine Insel“)

RPC: Kompatibilität von Datenformaten und -strukturen?

- Problem: Parameter komplexer Datentypen wie
  - Records, Strukturen
  - Objekte
  - Referenzen, Zeiger
  - Zeigergeflechte

  } sollen Adressen über Rechner- / Adressraumgrenzen erlaubt sein?
  } sollen Referenzen symbolisch, relativ,... interpretiert werden?
  } wie wird Typkompatibilität sichergestellt?
- Problem: RPCs werden oft in heterogenen Umgebungen eingesetzt mit unterschiedlicher Repräsentation, z.B. von
  - Strings (Längenfeld ↔ '\0' als Endekennung)
  - Character (ASCII ↔ Unicode)
  - Arrays (zeilen- ↔ spaltenweise)
  - Zahlen (niedrigstes Bit vorne oder hinten)

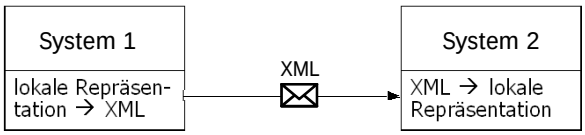
RPC: Marshalling

Marshalling: Zusammenstellen der Nachricht aus den aktuellen Prozedurparametern

- evtl. dabei geeignete „standardisierte" Codierung (komplexer) Datenstrukturen
- Glätten („flattening") komplexer (evtl. verzeigerter) Datenstrukturen zu einer Sequenz von Basistypen (evtl. mit Strukturinformation)
- umgekehrte Transformation wird oft als „unmarshalling" bezeichnet

RPC: Datenkonversion

1) Umwandlung in eine gemeinsame Standardrepräsentation
   - z.B. XML bei „XML-RPCs“ (in Web-Applikationen)



   - Beachte: Jeweils zwei Konvertierungen erforderlich; für jeden Datentyp jeweils Kodierungs- und Dekodierungsroutinen vorsehen

RPC: Datenkonversion (2)

2) Oder sendeseitig lokale Datenrepräsentation verwenden und dies in der Nachricht vermerken
   - „receiver makes it right“
   - Vorteil: bei gleichen Systemumgebungen/Computertypen ist keine (doppelte) Umwandlung nötig
   - Empfänger muss aber die Senderrepräsentation kennen und mit ihr umgehen können

Generell: Datenkonversion ist überflüssig, wenn sich alle Kommunikationspartner an einen gemeinsamen Standard halten

RPC: Transparenzproblematik

- RPCs sollten so weit wie möglich lokalen Prozedur-aufrufen (als bekanntes Programmierparadigma) gleichen, es gibt aber einige inhärente Unterschiede
  - z.B. bei Nichterreichbarkeit oder Absturz des Servers; RPCs dauern auch länger als lokale Prozeduraufrufe
- Beachte auch: Client-/Serverprozesse haben evtl. unterschiedliche Lebenszyklen
  - Server könnte z.B. noch nicht oder nicht mehr oder in einer „falschen“ (z.B. veralteten) Version existieren
  - Anwender oder Programmierer eines Clients hat typischerweise keine Kontrolle über den Server

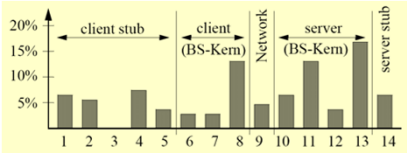
RPC: Leistungstransparenz?

- RPC i.Allg. wesentlich langsamer als lokaler Prozeduraufruf
- Kommunikationsbandbreite ist bei umfangreichen Datenmengen relevant
- Oft ungewisse, variable Verzögerungen

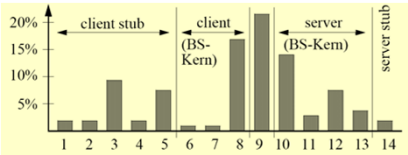
Effizienzzanalyse eines RPC-Protokolls

(zitiert nach A. Tanenbaum)

Null-RPC (Nutznachricht der Länge 0, keine Auftragsbearbeitung)



1440 Byte-Nutznachricht (ebenfalls keine Auftragsbearbeitung)



1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the bus
9. Network transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

- Eigentliche Übertragung (9) kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) ist nicht vernachlässigbar
- Bei kurzen Nachrichten ist Kontextwechsel zwischen Anwendung und Betriebssystem relevant (6, 13)
- Mehrfaches Kopieren kostet viel

RPC: Ortstransparenz?

- Evtl. muss Server („Zielort“) bei Adressierung explizit genannt werden
- Getrennte Adressräume von Client und Server
  - keine Kommunikation über globale Variablen möglich
  - typischerweise keine Pointer / Referenzparameter als Parameter möglich

RPC: Fehlertransparenz?

- Es gibt mehr Fehlerfälle
  - beim klassischen Prozeduraufruf gilt: Client = Server → „alles oder nichts“
  - hier: partielle („einseitige“) Systemausfälle typisch (Server-Absturz, Client-Absturz)

⇒ Fehlerproblematik ist also „kompliziert“!

- Nachrichtenverlust
  - ununterscheidbar von zu langsamer Nachricht!
- Client/Server haben zumindest zwischenzeitlich eine unterschiedliche Sicht des Zustands einer „RPC-Transaktion“
  - crash kann im „ungünstigsten Moment“ des RPC-Protokolls erfolgen

Typische RPC-Fehlerursachen

Wir besprechen nachfolgend 4 typische Fehlerursachen:

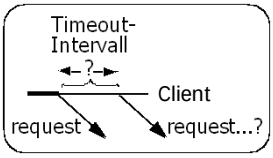
1. Verlorene Request-Nachricht
2. Verlorene Reply-Nachricht
3. Server-Crash
4. Client-Crash

Grundprobleme:

- Client / Server haben temporär eine inkonsistente Sicht
- Timeout beim Client kann verschiedene Ursachen haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server,...) → Fehlermaskierung schwierig

Verlorene Request-Nachricht

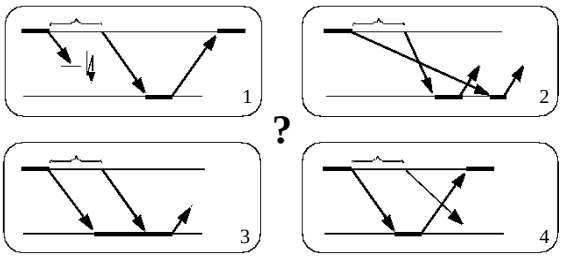
- Gegenmassnahme:
  Nach Timeout (kein Reply) die Request-Nachricht erneut senden



- Probleme:
  - Wie viele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Was, wenn die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam?

Verlorene Request-Nachricht (2)

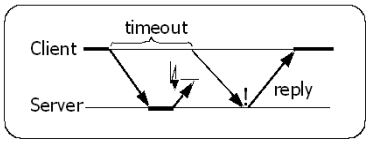
- Probleme, wenn Nachricht tatsächlich gar nicht verloren:
  - Doppelte Request-Nachricht!
    (Gefährlich bei nicht-idempotenten serverseitigen Operationen!)
  - Server sollte solche Duplikate erkennen (Denkübung: Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
  - Würde das Quittieren der Request-Nachricht etwas bringen?



Bei automatischer Wiederholung von Request-Nachrichten ist eine Vielzahl möglicher Fälle zu unterscheiden

Verlorene Reply-Nachricht

- Gegenmassnahme 1:
  analog zu verlorener Request-Nachricht
  - also: Anfrage bei Timeout wiederholen

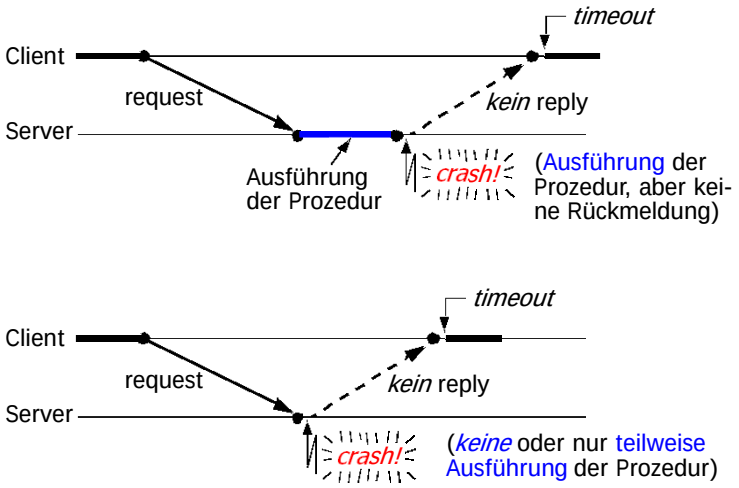


- Probleme:
  - vielleicht ging aber tatsächlich der Request verloren?
  - oder der Server war nur langsam und arbeitet noch?
  - ist aus Sicht des Clients nicht unterscheidbar!

Verlorene Reply-Nachricht (2)

- Gegenmassnahme 2:
  Server hält eine „Historie“ versendeter Replies
  - falls Server Request-Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne die Prozedur nochmal auszuführen
  - pro Client muss nur das neueste Reply gespeichert werden
- Bei vielen Clients u.U. dennoch Speicherplatzprobleme:
  → Historie nach „einiger“ Zeit löschen bzw. kürzen
  - und wenn man ein gelöschtes Reply später dennoch braucht?
  - ist in diesem Zusammenhang ein ack eines Reply sinnvoll?

Server-Crash

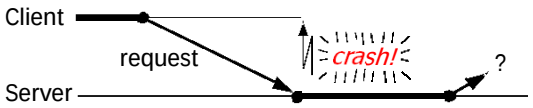


Server-Crash (2)

- Probleme: Wie soll der Client obige Fälle unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinn und Erfolg konkreter Gegenmassnahmen hängt u.U. davon ab!
  - Client meint evtl. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (→ falsche Sicht des Zustandes!)
- Evtl. Probleme nach einem Server-Restart
  - z.B. „Locks“, die noch bestehen (Gegenmassnahmen?) bzw. allgemein: „verschmutzter“ Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information, um in alte Transaktionszustände problemlos wieder einzusteigen

Client-Crash

- Oder auch:
  Client mittlerweile nicht mehr am reply interessiert



- Reply des Servers wird nicht abgenommen
  - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
  - blockiert i.Allg. Ressourcen beim Server!

Client-Crash (2)

- Problem: „Orphans“ (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert
- Nach Restart könnte ein Client versuchen, Orphans zu terminieren (z.B. durch Benachrichtigung der Server)
  - Orphans könnten aber bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben,...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)
- „Sehr“ alte Prozesse, die für einen Auftrag gegründet wurden, werden als Orphans angesehen und terminiert

RPC-Fehlersemantik-Klassen

- Operationale Sichtweise:
  - Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1. Maybe-Semantik:
   - Keine Wiederholung von Requests
   - Einfach und effizient
   - Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
   - Mögliche Anwendungsklasse: z.B. Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

RPC-Fehlersemantik-Klassen (2)

2. At-least-once-Semantik:
   - Automatische Wiederholung (evtl. mehrfach) von Requests
   - Keine Duplikatserkennung (zustandsloses Protokoll auf Serverseite)
   - Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

   1) und 2) werden etwas euphemistisch oft als „best effort“ bezeichnet

---

3. At-most-once-Semantik:
   - Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
   - Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
   - Geeignet auch für nicht-idempotente Operationen

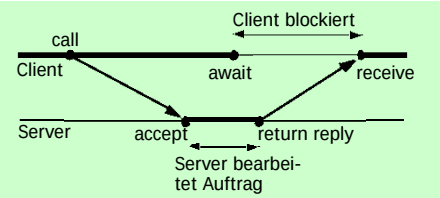
RPC-Fehlersemantik-Klassen (3)

- Maybe → at-least-once → at-most-once → ...
  ist zunehmend aufwändiger zu realisieren

  Ist „exactly-once“ machbar?
- Man begnügt sich daher, falls es die Anwendung erlaubt, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig

Asynchroner RPC

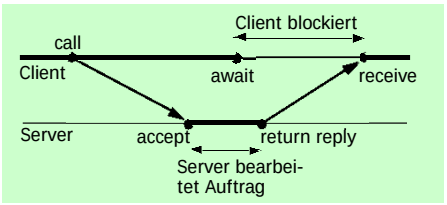
- Andere Bezeichnung: „Remote Service Invocation“
- Auftragsorientiert, d.h. also: Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Asynchroner RPC: Zuordnung Auftrag/Ergebnisempfang

- Unterschiedliche Ausprägung auf Sprachebene möglich
  - „await“ könnte z.B. einen bei „call“ zurückgelieferten „handle“ als Parameter erhalten, also z.B.: `Y = call X(…); … await(Y);`



  - evtl. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

Asynchroner RPC: Future-Variable

- Spracheinbettung evtl. auch durch „Future-Variablen“
- Future-Variable = „handle“ auf das in anderem Thread parallel berechnetes Funktionsergebnis
- Programm blockiert nur dann, wenn der Wert der Variable bei ihrer Nutzung noch nicht feststeht
- Beispiel (Scala):

```
...
val x = future(callRPC())
... // continue local computation
println(x()) // await x iff necessary
```

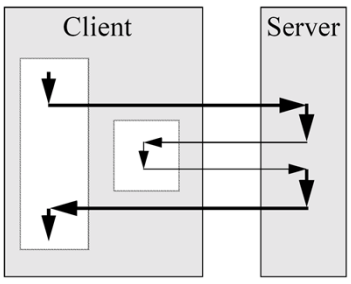
Einige Varianten / Ergänzungen zu RPCs in der Praxis

Wir besprechen nachfolgend 4 ergänzende Aspekte:

1. Rückrufe
2. Context-Handles
3. Broadcast bzw. Multicast
4. Sicherheit

Rückrufe („call back RPC“)

- Temporärer Rollentausch von Client und Server
  - um eventuell bei langen Aktionen Zwischenresultate zurückzumelden
  - um eventuell weitere Daten vom Client anzufordern
- Client muss Rückrufadresse beim call übergeben



Context-Handles

- Struktur mit Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg
  - vgl. „cookies“: kleine Textdatei, die ein Web-Server einem Browser (= Client) schickt und die im Browser gespeichert wird
- Context-Handles werden vom Server dynamisch erzeugt und an den Client (bei „reply“) zurückgegeben
- Client kann diese beim nächsten Aufruf (unverändert) wieder mitsenden
  - Server „erinnert“ sich so an den Kontext
- Vorteil: Server selbst arbeitet „zustandslos“

Weitere Varianten / Ergänzungen

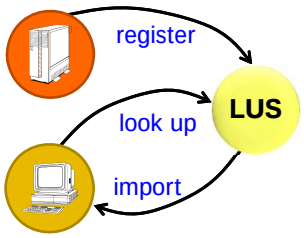
- Broadcast bzw. Multicast
  - Request wird „gleichzeitig“ an mehrere Server geschickt
  - RPC ist beendet mit der ersten empfangenen Antwort oder Client hat die Möglichkeit, nach einer Antwort auf eine weitere Antwort zu warten
- Oft wählbare Sicherheitsstufen, z.B.:
  - Authentifizierung bei Aufbau der Verbindung (“binding”)
  - Authentifizierung pro RPC-Aufruf oder pro Nachricht
  - Verschlüsselung der zugrundeliegenden Nachrichten
  - Schutz gegen Verfälschung
    (digitale Signatur, verschlüsselte Prüfsumme, MAC)

Beispiel „Secure RPC“
(Teil des „SUN RPC“)

- Client und Server vereinbaren (geheim) einen Session-Key K
  - wird zum Verschlüsseln der Nachrichten genutzt
- Jeder Request enthält einen mit K kodierten Zeitstempel
- Erster Request enthält zusätzlich verschlüsselt eine Zeitfenstergrösse W als zeitliches Toleranzintervall sowie (ebenfalls verschlüsselt) W-1
  - „zufälliges“ Generieren einer ersten Nachricht ist so nahezu unmöglich
  - replay (bei kleinem W) ist ebenfalls erfolglos
  - W ist verschlüsselt, auch um Angreifern keine Information über die Fenstergrösse zu geben
- Server akzeptiert einen Request nur, wenn:
  - Zeitstempel grösser als letzter Zeitstempel und
  - Zeitstempel innerhalb des Zeitfensters
- Zur Authentifizierung enthält die Antwort des Servers (verschlüsselt) den letzten erhaltenen Zeitstempel-1

Lookup-Service

- Problem: Wie finden sich Client und Server?
  - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gemeinsame Umgebung)
  - → Lookup-Service (LUS) oder „Registry“ fungiert als „Service-Broker“
- Server gibt seinen Service (d.h. RPC-Routine) dem LUS bekannt
  - register: RPC-Schnittstelle „exportieren“ (Name, Parameter, Typen,...)
  - evtl. später auch wieder abmelden



- Client erfragt beim LUS die Adresse eines geeigneten Servers
  - Angabe des gewünschten Typs von Service beim look up oder discovery
  - „importieren“ der RPC-Schnittstelle

Lookup-Service (2)

- Vorteile („Mehrwert“): im Prinzip kann LUS
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Verfügbarkeit eines Services testen
  - verschiedene Versionen eines Dienstes verwalten
- Probleme:
  - look up kostet Ausführungszeit (gegenüber statischem Binden)
  - zentraler LUS ist ein potentieller Engpass / „single point of failure“ (evtl. LUS geeignet replizieren / verteilen)
  - wie lernen Client oder Server die Adresse des „richtigen“ oder „zuständigen“ LUS kennen?
    - z.B. feste („well-know“) Adresse oder Broadcast in lokaler Umgebung

Web Services als Beispiel für das Client-Server-Modell

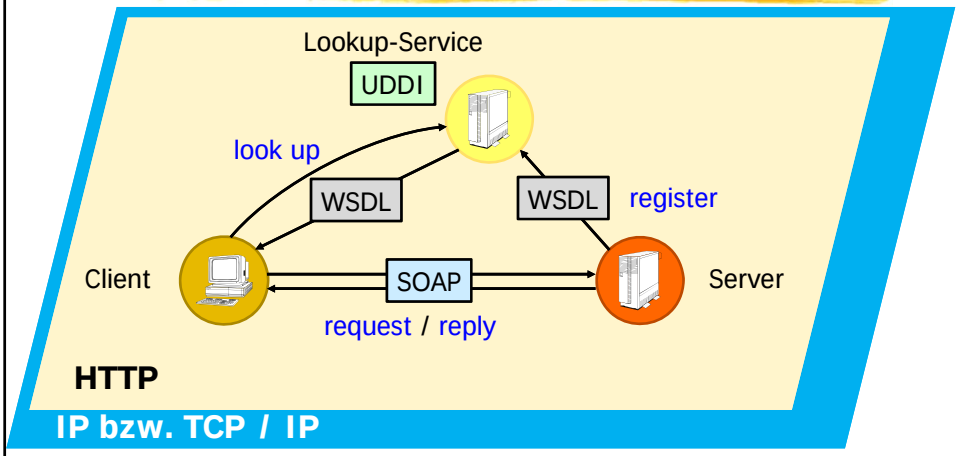
- Problem: Internet ist zu heterogen für eine einheitliche Programmiersprache oder RPC-Laufzeitumgebung
  - „Lösung“: Web Services als offener, plattform- bzw. sprach-unabhängiger Standard, dessen Schnittstellen von diversen Plattformen implementiert werden können

---

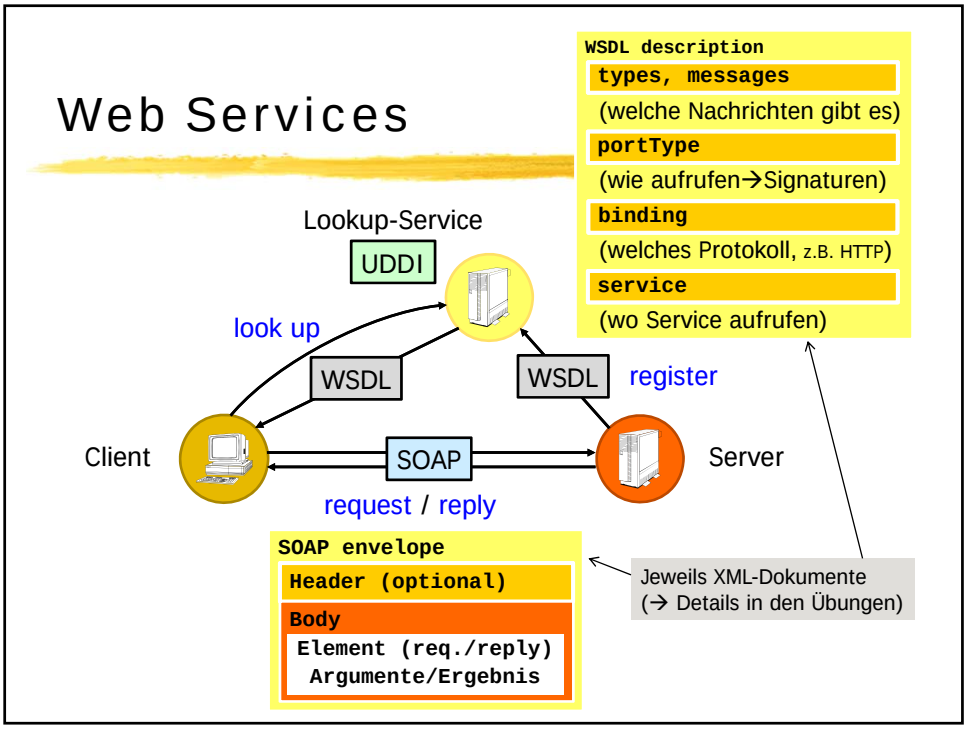
- HTTP fungiert als Transportschicht ←
- SOAP als plattformunabhängige Protokollspezifikation (ursprünglich: „Simple Object Access Protocol“)
- UDDI als Lookup-Service („Universal Description, Discovery and Integration“)
- WSDL als standardisierte Service-Beschreibung („Web Services Description Language“)

Alternativ zu HTTP: UDP oder SMTP, z.B. für Mitteilungen ohne Antwort oder wenn Resultatberechnung länger als typischer HTTP-Timeout dauert

Web Services



Web Services



Web Services

WSDL description
- **types, messages**
  (welche Nachrichten gibt es)
- **portType**
  (wie aufrufen→Signaturen)
- **binding**
  (welches Protokoll, z.B. HTTP)
- **service**
  (wo Service aufrufen)

XML-Dokument, das (bzgl. UDDI bzw. für einen Client) den Service beschreibt

XML (Extensible Markup Language):

- Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten (z.B. ASCII)
- Unterstrukturen mit Start- (<Tag-Name>) und End-Tag (</Tag-Name>) oder einem Empty-Element-Tag (< *Tag-Name* />)
- Attribute bei einem Tag (Attribut-Name= "Attribut-Wert") für Zusatzinformationen

```
<books>
  <book>
    <author>Karl May</author>
    <title>Winnetou</title>
    <ISBN>3-7802-0170-4</ISBN>
    <price format="EUR"/>
  </book>
  <pubinfo>
    <publisher>
      KM-Verlag
    </publisher>
    <town>Bamberg</town>
  </pubinfo>
</books>
```

WSDL: types

WSDL description
- **types, messages**
  (welche Nachrichten gibt es)
- **portType**
  (wie aufrufen→Signaturen)
- **binding**
  (welches Protokoll, z.B. HTTP)
- **service**
  (wo Service aufrufen)

types

XML-Schema zur Typenbeschreibung

- Definition von Typen (bzw. deren Namen) als XML-„element“
- Meistens als Verweis auf einen „complexType“, der aus „elements“ der Basistypen (int, float,...) besteht
- Schema definiert auch einen Namens-raum, der als URI angegeben wird
- Schema oft in externe .xsd-Datei ausgelagert

→ **nächste 2 slides**

WSDL-Namensräume

- ExampleWebServices.wsdl

Namensraum der beschriebenen Web Services

Weitere Namensräume, aus denen Tags benötigt werden

Importiert die Typendefinitionen; können auch eigenen Namensraum haben

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
  name="ExampleWebServices"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap">

  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://example.org/VS/WebServices/"
        schemaLocation="ExampleSchema.xsd"/>
    </xsd:schema>
  </types>
...
```

nächste slide

WSDL-Namensräume

- ExampleSchema.xsd

Hier gleicher Namensraum wie Services

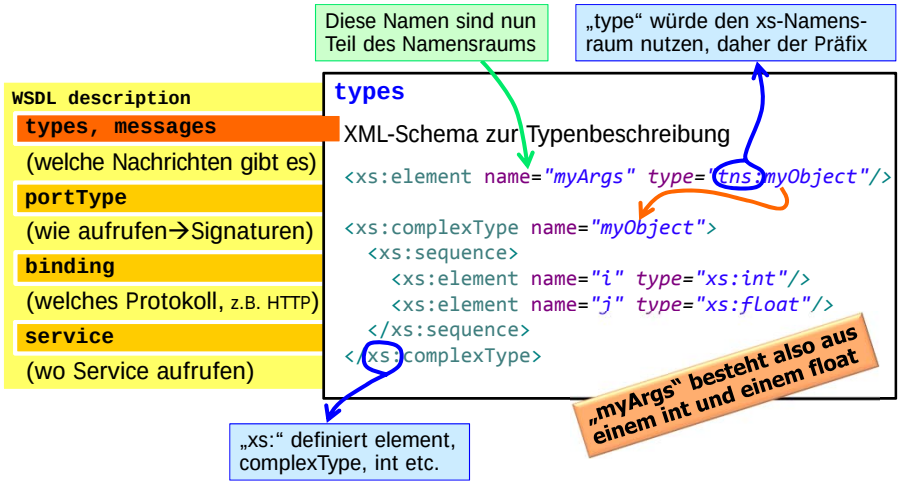
Definiert das Präfix „tns:“, um den targetNamespace anzusprechen

Lässt „xs:“ auf den allgemeinen XML-Schema-Namensraum zeigen

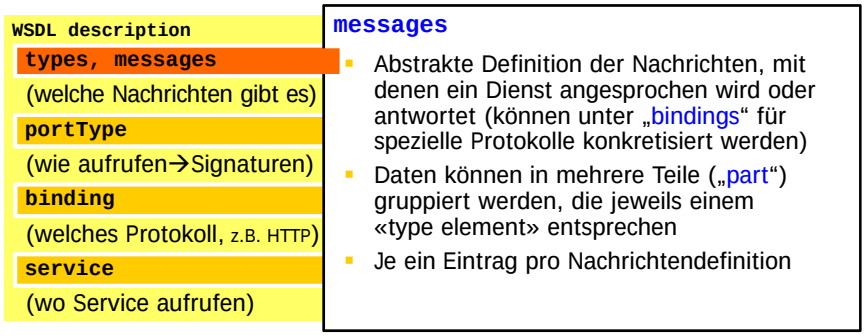
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
  version="1.0"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Elementdefinitionen -->

</xs:schema>
```

nächste slide

WSDL-Beispiel: types



WSDL: messages



WSDL-Beispiel: messages

„myArgs“ wurde oben definiert

WSDL description
- **types, messages**
- (welche Nachrichten gibt es)
- **portType**
- (wie aufrufen→Signaturen)
- **binding**
- (welches Protokoll, z.B. HTTP)
- **service**
- (wo Service aufrufen)

messages

Die Gliederung in „parts“ ist optional

```
<message name="myRequest">
  <part name="parameters"
        element="tns:myArgs"/>
  <part name="optionalParameters"
        element="tns:myOpt"/>
</message>

<message name="myResponse">
  <part name="result" element="tns:myRet"/>
</message>
```

„myRequest“ hat also einen int und einen float als Parameter

WSDL: portType

WSDL description
- **types, messages**
- (welche Nachrichten gibt es)
- **portType**
- (wie aufrufen→Signaturen)
- **binding**
- (welches Protokoll, z.B. HTTP)
- **service**
- (wo Service aufrufen)

portType

- Definition der einzelnen Web Services
- Jeder Service entspricht einer „operation“
  - hat typischerweise eine „input“-Nachricht und eine „output“-Nachricht
  - zusätzlich können Fehlerbenachrich-tigungen angegeben werden („fault“)
- Services können auch unidirektional sein, (z.B. nur „output“ für Benachrichtigungen)

WSDL-Beispiel: portType

WSDL description

- **types, messages**
- (welche Nachrichten gibt es)
- **portType**
- (wie aufrufen→Signaturen)
- **binding**
- (welches Protokoll, z.B. HTTP)
- **service**
- (wo Service aufrufen)

portType

```
<portType name="groupOfServices">
  <operation name="myService">
    <input message="tns:myRequest"/>
    <output message="tns:myResponse"/>
    <fault message="tns:someFault"/>
  </operation>
</portType>
```

Hier könnten weitere „operation" stehen

„myService" wird also mit einem int und einem float aufgerufen

WSDL: binding

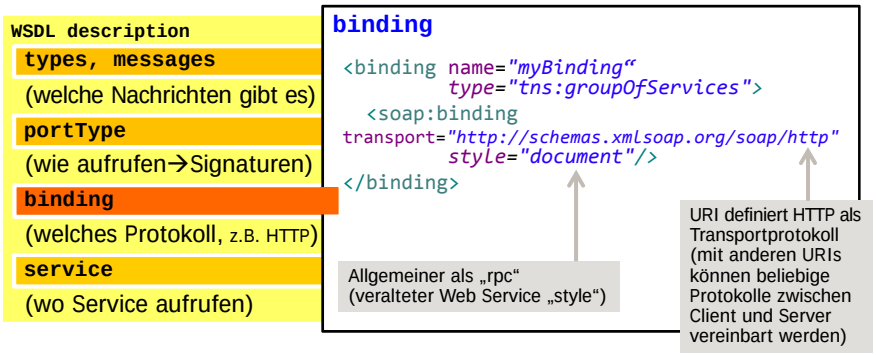
WSDL description

- **types, messages**
- (welche Nachrichten gibt es)
- **portType**
- (wie aufrufen→Signaturen)
- **binding**
- (welches Protokoll, z.B. HTTP)
- **service**
- (wo Service aufrufen)

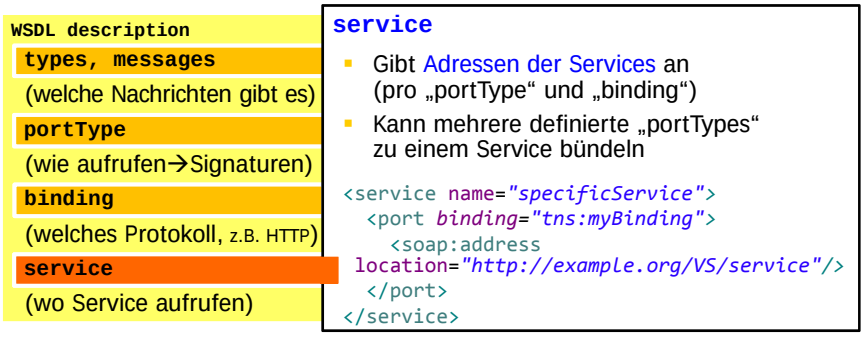
binding

- Bindet „portType" an ein Protokoll (z.B. HTTP)
- Es kann mehrere „binding"-Einträge für verschiedene Protokolle geben
- Im Normalfall genügen die Informationen aus „message" und „portType" für die Abbildung der Nachrichten auf ein konkretes Format (d.h. „binding" enthält kaum Information)

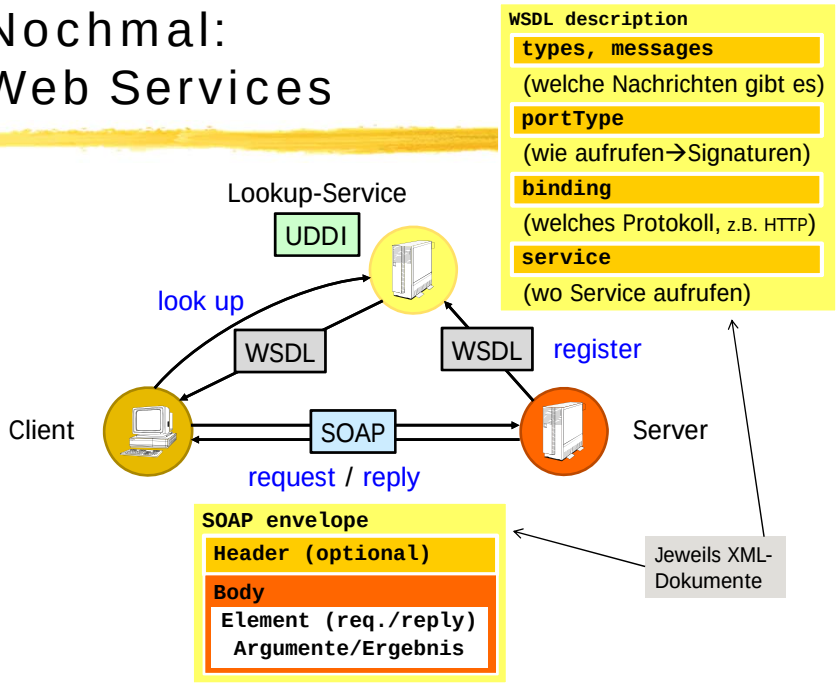
WSDL-Beispiel: binding



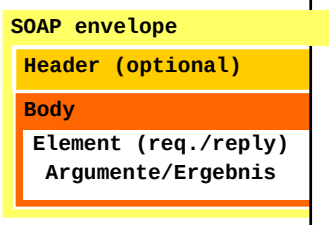
WSDL: service



Nochmal: Web Services



SOAP: envelope



Envelope

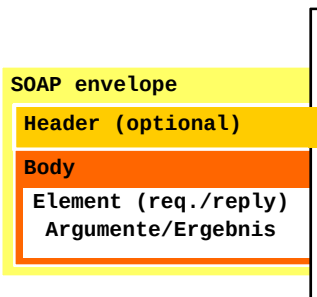
Erforderliche Attribute:

- „namespace“ definiert das Dokument als SOAP-Nachricht
- „encodingStyle“ gibt Kodierungsregeln für die SOAP-Serialisierung an

Enthaltene Teilstrukturen:

- Header (optional)
- Body (erforderlich)

SOAP: header

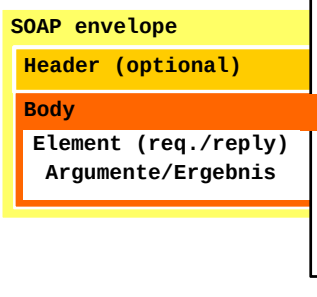


Header

Der SOAP-Header muss nicht enthalten sein

Durch ihn könnten zusätzliche Informationen über den Transaktionskontext, z.B. bezüglich Authentifizierung oder Bezahlung, angegeben werden

SOAP: body

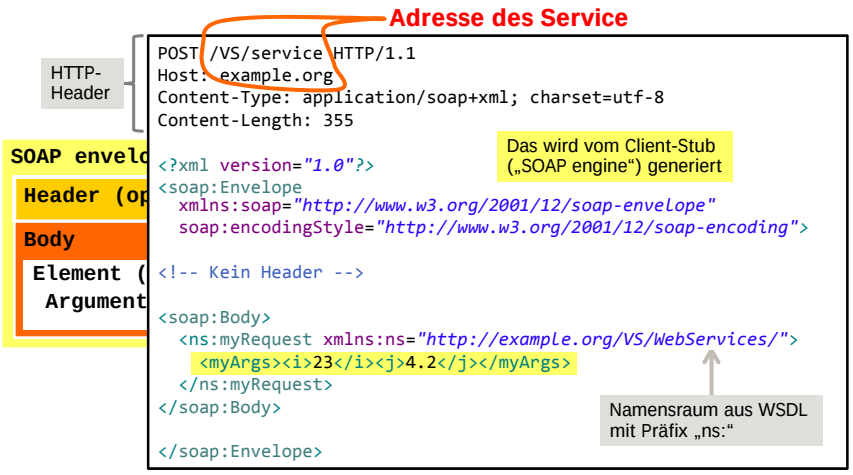


Body

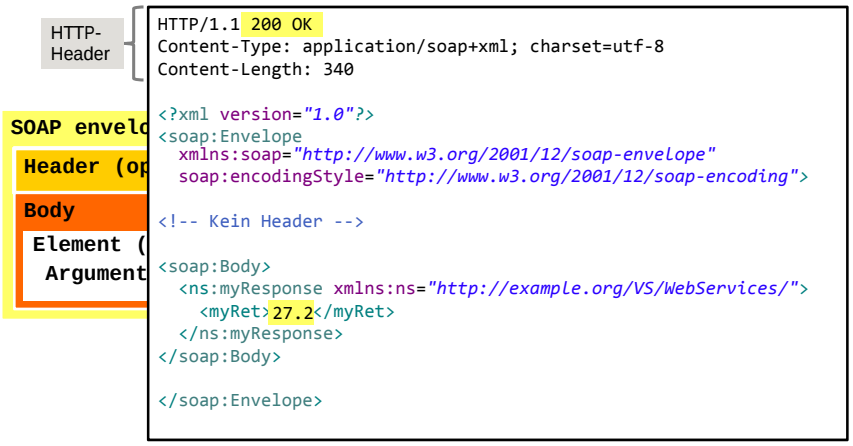
SOAP-Nutzlast:

- Enthält die im WSDL definierte „message“ mit ihren Elementen
- Es wird der Namensraum aus dem WSDL angegeben, womit die dort definierten Tags verwendet werden können

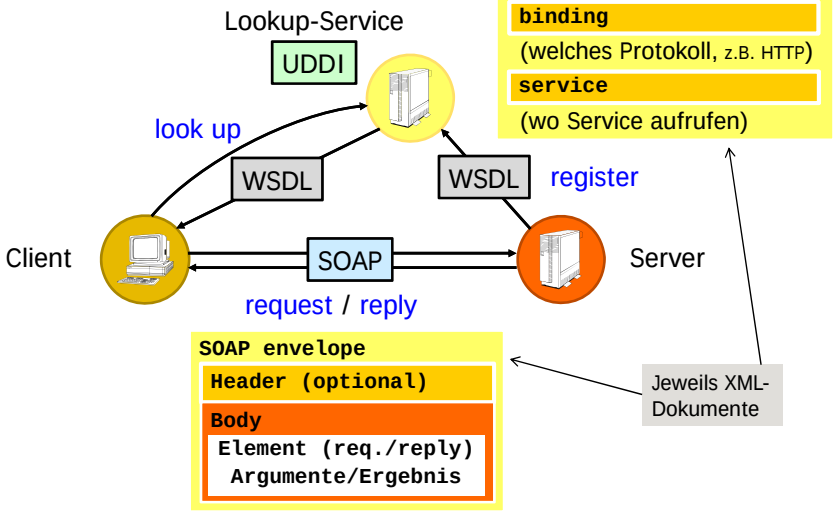
SOAP-Beispiel: Request



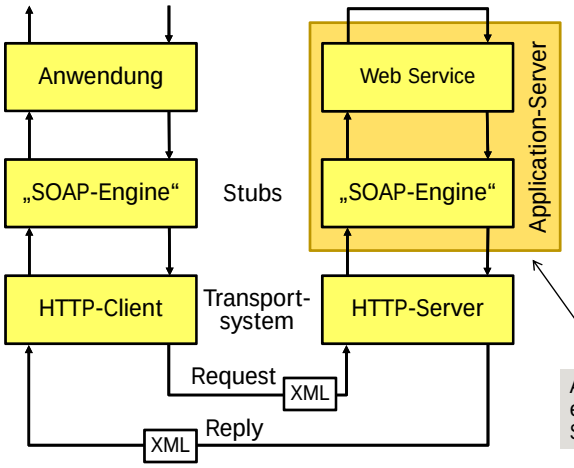
SOAP-Beispiel: Response



Nochmal: Web Services



Web Service Stubs



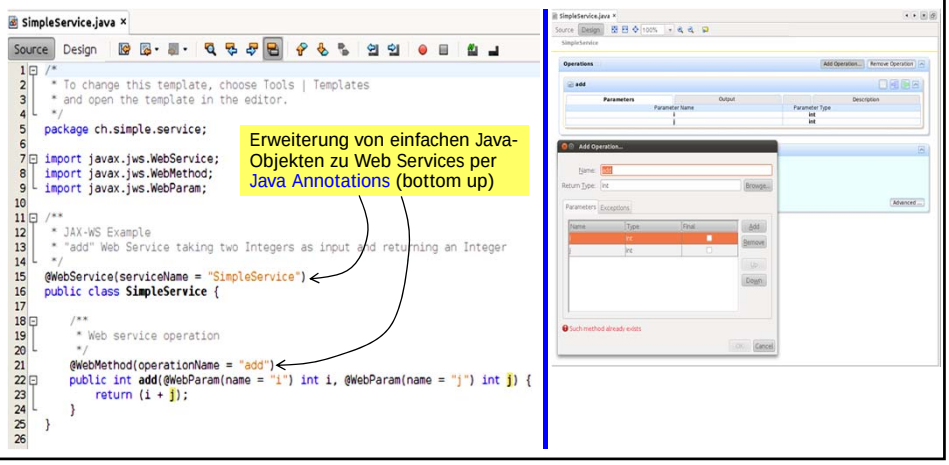
- Server-Stubs werden oft aus einer Web Service-Implementierung generiert
  (bottom up / „code first“)
- ...oder automatisch aus einer WSDL-Beschreibung des Interfaces
  (top down / „contract first“)
- Client-Stubs können ebenfalls aus WSDL generiert werden

Application-Server haben oft einen integrierten HTTP-Server, z.B. „Jakarta Tomcat“

vgl. frühere Folie „RPC: Implementierung“

Entwicklung von Web Service-Komponenten mit Java-IDE

- JAX-WS: Java API for XML Web Services (Beispiel: NetBeans)

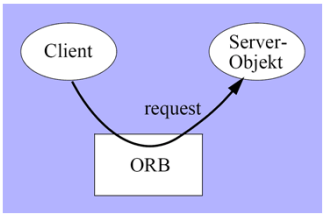


Middleware-Entwicklung, die Web Services historisch voranging

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 Aufträge

- Verändern Aufträge den Zustand des Servers?
  - typische zustandsinvariante Aufträge: Anfrage bei Auskunftsdienst (z.B. Namensdienst oder Zeitservice)
  - typische zustandsändernde Aufträge: Schreiben bei Datei-Server
- Idempotente 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 fehlgeschlagenem 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 auf 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.

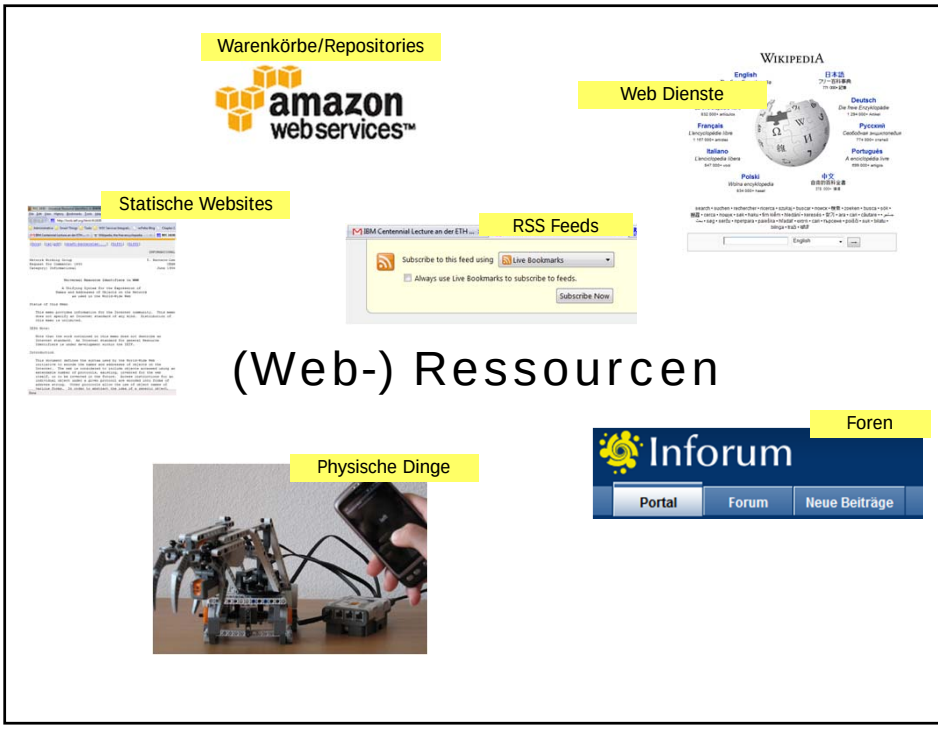
Ressourcen-orientierte Architektur (ROA)

- Funktionalität wird nicht durch Services („SOA“), sondern durch (Web-) Ressourcen angeboten
- Ressource? Bezugsobjekt 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: *ISBN-13: 978-3780200075*
- Was wir kaufen oder ausleihen 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
- Motivation und Entwicklung
  - Erfolg des WWW in technischer Hinsicht (z.B. bzgl. Skalierbarkeit) beruht auf Eigenschaften der zugrundeliegenden Protokolle und Mechanismen
  - Idealisierung dieser Architektur durch Abstraktion von HTTP
  - Mit REST wird des weiteren versucht, die Möglichkeiten, die das Web (bzw. HTTP) bietet, optimal auszunutzen

Eigenschaften von REST

- Zustandslosigkeit
  - entschärft Crash-Problematik und Orphans
  - erlaubt Caching und bessere Skalierbarkeit
- Einheitliche und 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
- Bevorzugte Repräsentation wählbar
  - HTML, XML, JSON,...

auch andere Protokolle möglich!

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

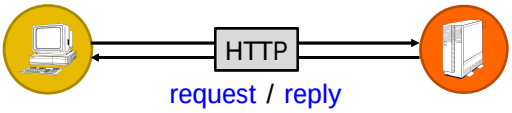
@Path: Pfad zur Ressource

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

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

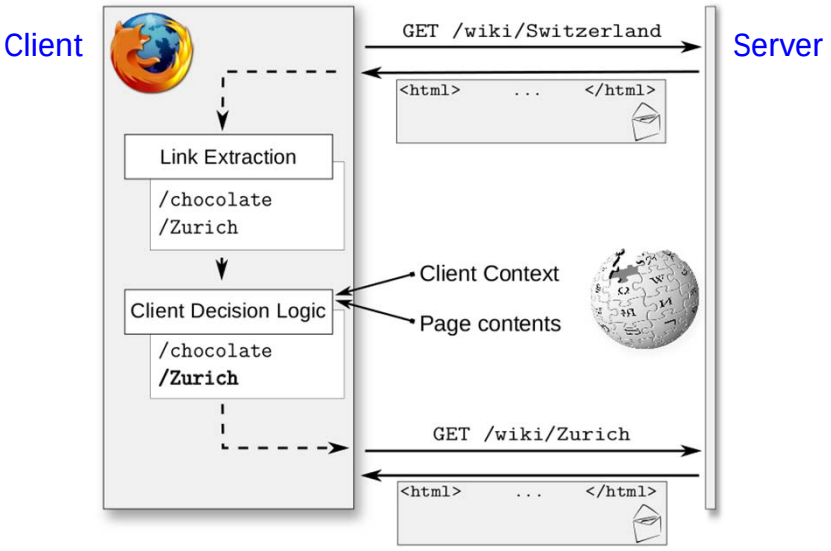
REST-Anwendungsmodell

- „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

Beispiel:



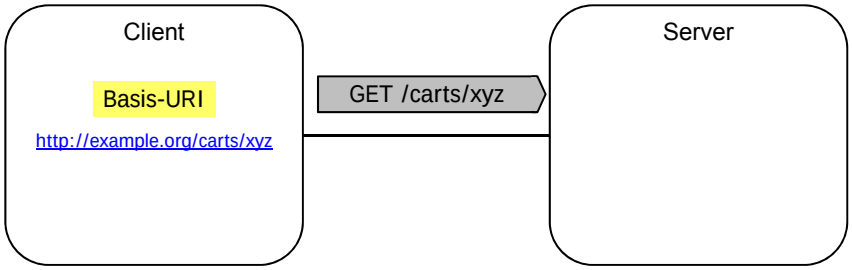
REST: Zustandsspeicherung

- 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 ergeben Sinn (sind vollständige URI, losgelöst von jeglichem Kontext)
  - Back button im Browser ergibt Sinn: er führt zu einem früheren (im Client gecachten) Zustand

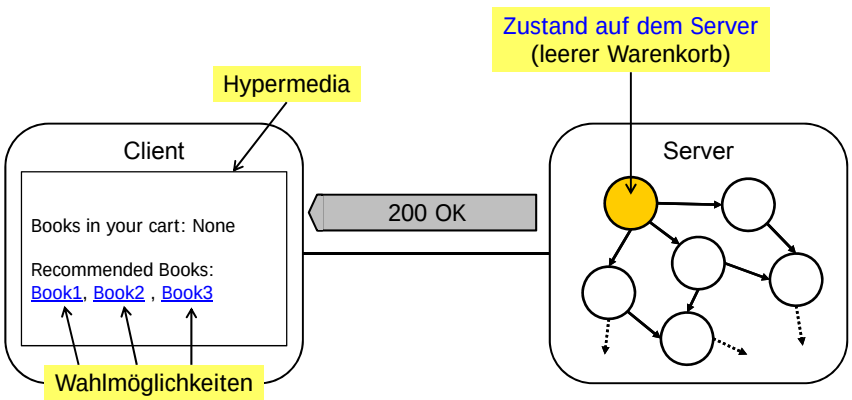
REST: Zustandsspeicherung

- Bei komplexeren Anwendungen residiert typischerweise der Anwendungszustand beim Server
  - z.B. Flugbuchung, wo 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

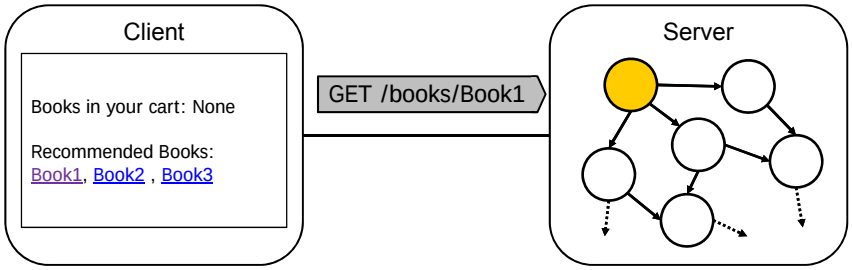
Zustandsspeicherung beim Server



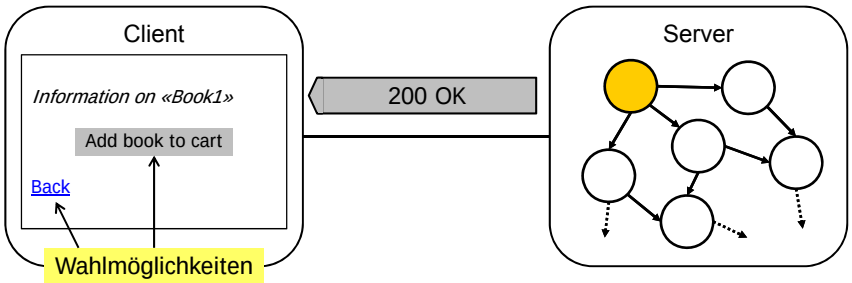
Zustandsspeicherung beim Server



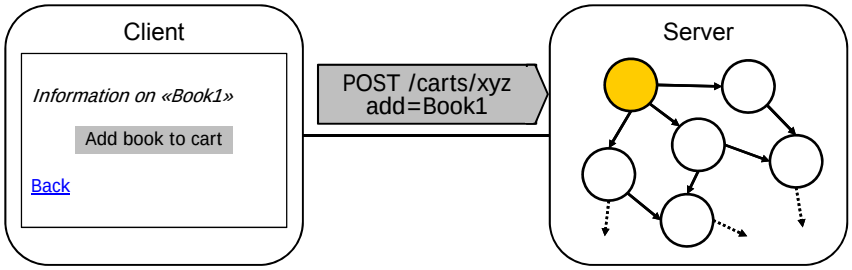
Zustandsspeicherung beim Server



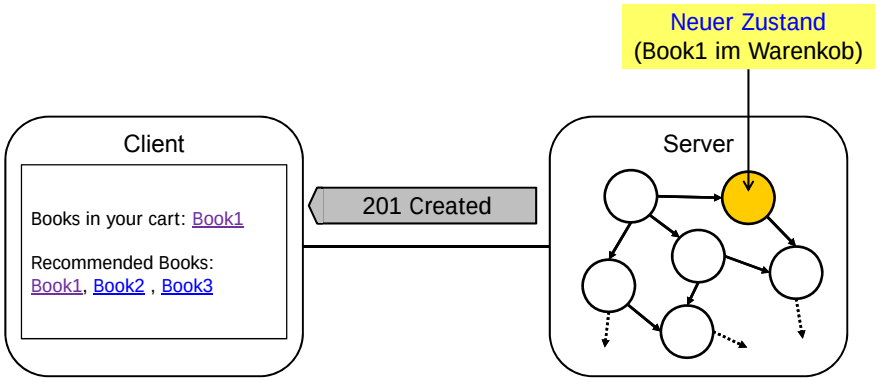
Zustandsspeicherung beim Server



Zustandsspeicherung beim Server



Zustandsspeicherung beim Server



Zustand wird auf dem Server gemerkt, indem die Ressource /carts/xyz/Book1 angelegt wird