

Verteilte Betriebssysteme

4. Kapitel
Architekturen

Matthias Werner
Professur Betriebssysteme

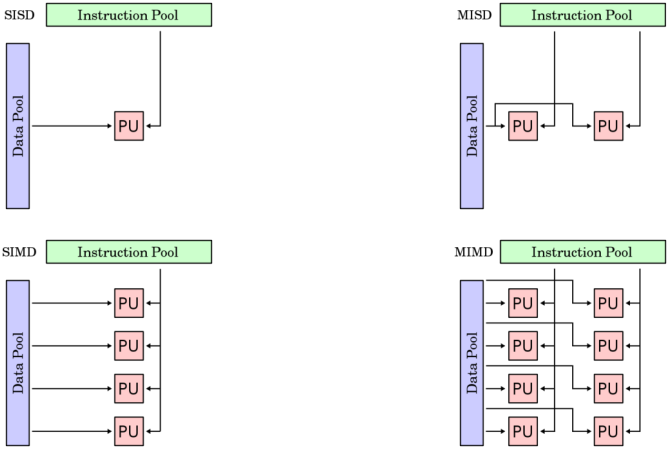
4.1 Hardware

Wiederholung: Klassifikation nach Flynn

- Das bekannteste und einfachste Klassifikationsschema unterscheidet nach Einfachheit oder Vielfachheit von Befehls- und Datenströmen
- Durch Kombination gelangt man zu vier Klassen:

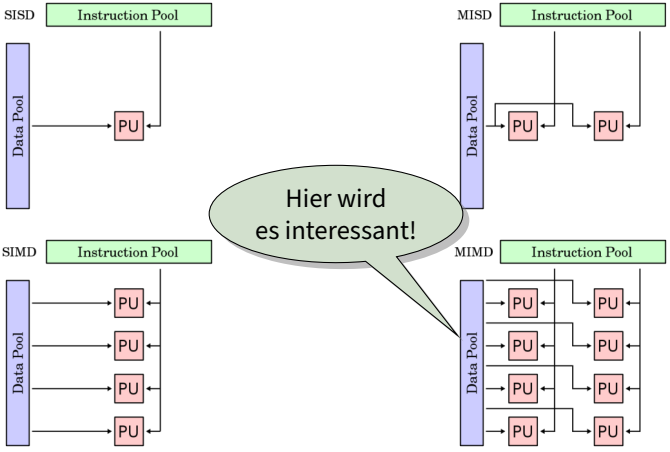
	SI (single instruction)	MI (multiple instruction)
SD (single data)	**SISD** klassischer von-Neumann-Rechner, Harvard-Rechner	**MISD** Pipeline, Datenflussarchitektur (?)
MD (multiple data)	**SIMD** Vektorrechner, Feldrechner (Cray-1, CM2, MasPar)	**MIMD** Multiprozessorsysteme, Verteilte Systeme (Cray T3E, Cluster, ...)

Wiederholung: Klassifikation nach Flynn (Forts.)



Bildquelle: wikipedia

Wiederholung: Klassifikation nach Flynn (Forts.)

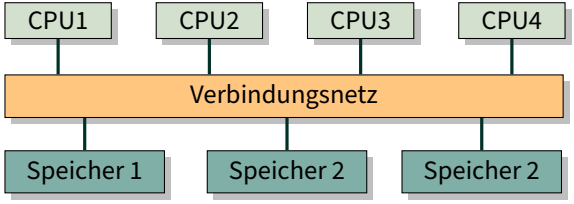


Bildquelle: wikipedia

MIMD-Maschinen

a) **Multiprozessorsysteme** (shared memory systems, tightly coupled systems)

- Alle Prozessoren greifen über ein gemeinsames Kommunikationsnetz auf gemeinsame Speichermodule zu

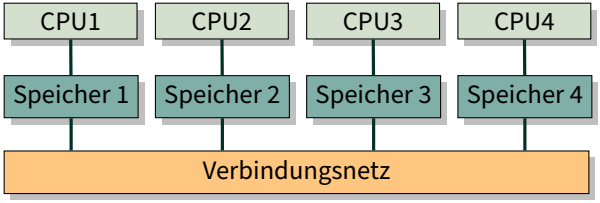


- Nach der Art der Erreichbarkeit der Speichermodule kann weiter unterschieden werden:
  - einheitlicher Speicherzugriff (uniform memory access, **UMA**)
    ➡ Zugriffszeit zum Speichermodul ist unabhängig von der Adresse des Prozessors oder des Speichermoduls
  - uneinheitlicher Speicherzugriff (nonuniform memory access, **NUMA**)
    ➡ Zugriffszeit hängt von Adresse ab

MIMD-Maschinen (Forts.)

b) **Multirechnersysteme** (message passing systems, loosely coupled systems)

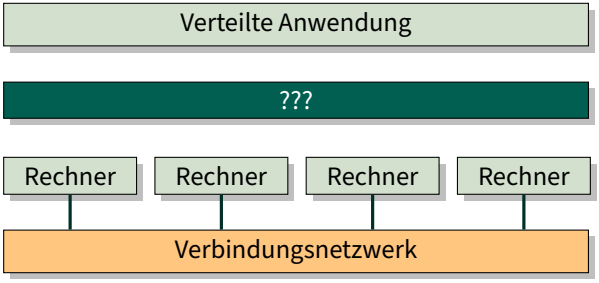
- Jeder Prozessor verfügt über einen eigenen Speicher, auf den nur er Zugriff hat
- Datenaustausch geschieht durch Versenden von **Nachrichten** über ein Verbindungsnetzwerk



Wenn nichts anderes gesagt wird, gehen wir in dieser Lehrveranstaltung stets von Multirechnersystemen aus.

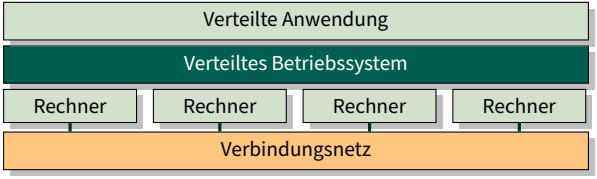
4.2 Architektur verteilter Betriebssysteme

- Anwendungen sollen in möglichst transparenter Weise unterstützt werden, d.h. es wird eine SW-Infrastruktur benötigt, welche die Verteiltheit weitgehend **verdeckt**

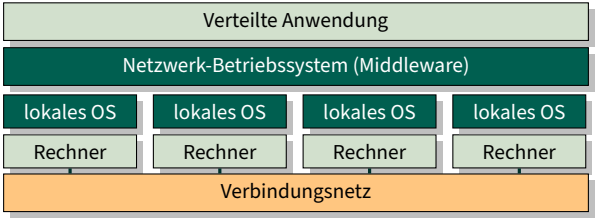


Architekturansätze für verteilte Systeme

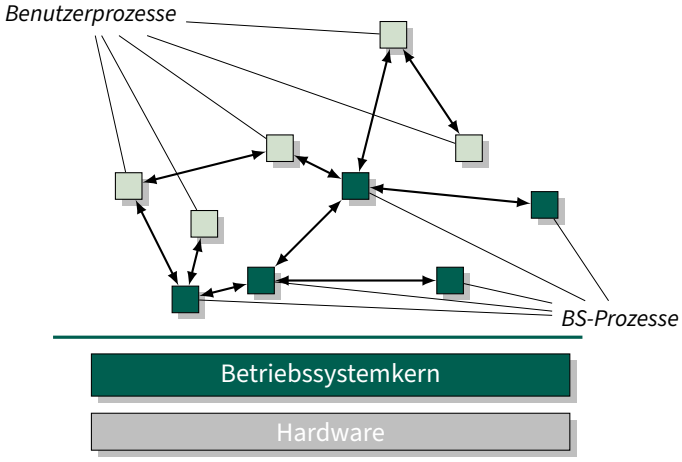
- **Verteiltes Betriebssystem** ➡ Das Betriebssystem selbst stellt bereits die verteilungsrelevanten Funktionen zur Verfügung



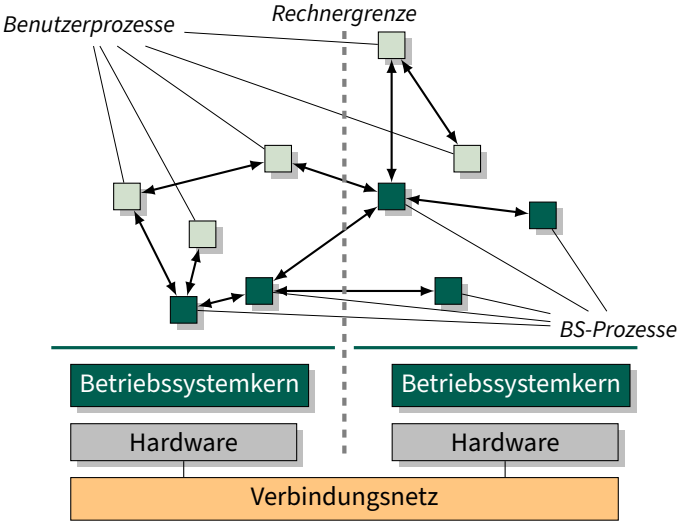
- **Netzwerk-Betriebssystem** ➡ Die lokalen Betriebssysteme werden ergänzt um eine weitere Schicht (Netzwerkbetriebssystem, Middleware), welche die verteilten Funktionen anbietet



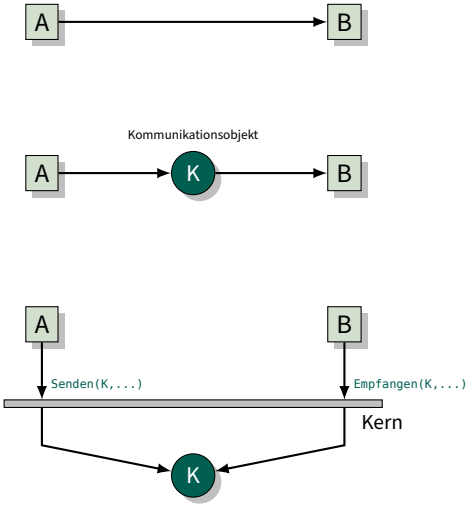
Architektur lokaler Betriebssysteme



Architektur verteilter Betriebssysteme



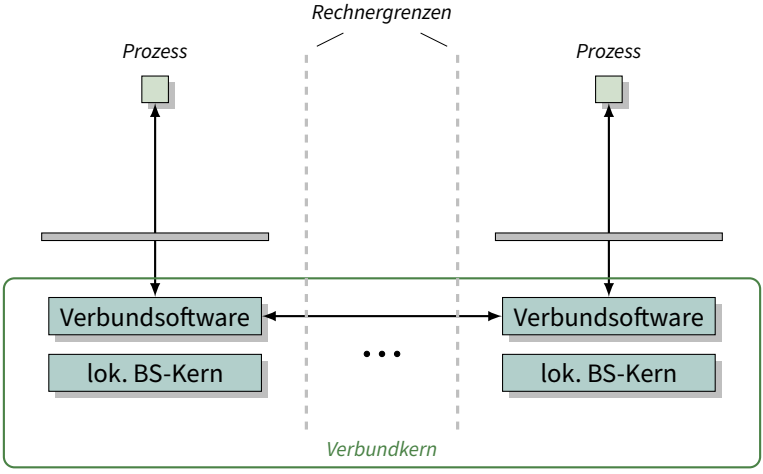
Prozesskommunikation



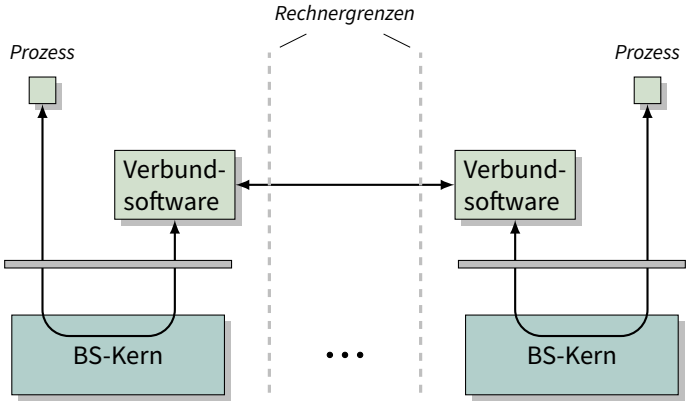
Architekturvarianten

- In verteilten Systemen müssen zur Prozessinteraktion Rechnergrenzen überwunden werden
- Erforderliche Kommunikationssoftware kann nun
  - im Betriebssystemkern integriert sein ➡ **Kernverbund** (*kernel federation*)
    oder
  - als spezielle Komponente außerhalb des Kerns existieren ➡ **Prozessverbund**
- Beim **Kernverbund** ist die Verteiltheit unter der Kernschnittstelle verborgen
  - Kernaufrufe können sich auf beliebige Objekte im Netz beziehen
  - Die Vereinigung aller Kerne bildet den **Verbundkern**
- Beim **Prozessverbund** (*process federation* oder *server federation*) bleibt die lokale Kernschnittstelle unangetastet
  - Der lokale Kern ist sich nicht „bewusst“, dass er Teil eines verteilten Systems ist
  - Auf diese Weise können bestehende Betriebssysteme zu verteilten Betriebssystemen erweitert werden

Kernverbund



Prozessverbund



Anmerkung

Die Abgrenzung zwischen verteilten Betriebssystemen mit Prozessverbund und Netzwerk-Betriebssystemen sind fließend.

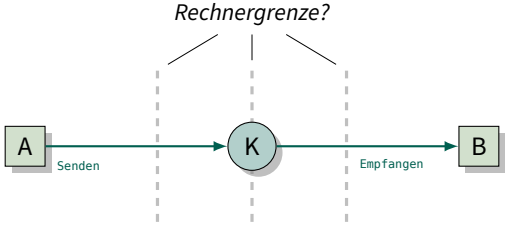
Abgrenzung verteilter Betriebssysteme

	Multiprozessor-BS	**Verteiltes BS**	**Netzwerk-BS**
Alle Knoten mit gleichem BS?	ja	ja	nein
# lokaler BS-Instanzen	1	n	n
Gemeinsame Prozesswarteschlange?	ja	nein	nein
Kommunikation	gemeinsamer Speicher	Nachrichtenaustausch	Nachrichtenaustausch / gemeinsame Daten

4.3 Kommunikation

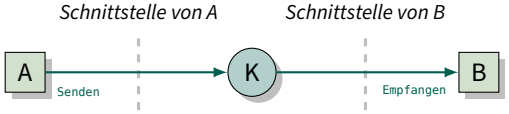
Brückenprinzip

- Komplexere Programmsysteme können nach dem **Client-Server-** oder **Peer-to-Peer-**Prinzip als Geflecht kommunizierender Prozesse aufgebaut werden
- Kanalansatz kann problemlos auf verteilte Systeme übertragen werden
- Zu klären: **Wie** wird Kanal-Prinzip über Rechnergrenzen hinweg realisiert?

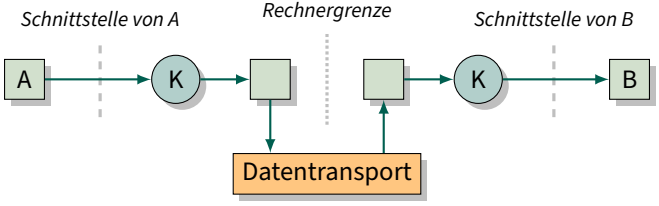


Brückenprinzip (Forts.)

- Wegen Verteilungstransparenz sollten Schnittstellen der beteiligten Prozesse unverändert bleiben



- Umsetzung: Jedem Prozess wird die lokale Existenz des Kommunikationspartners „vorgegaukelt“
- Dies führt zur folgenden Konstruktion:

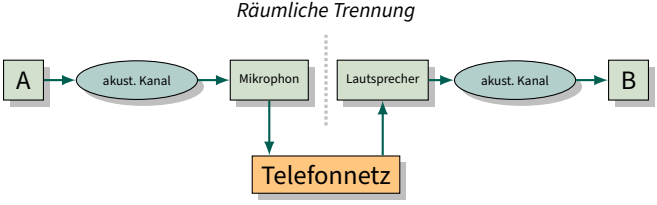


Analogie: Gespräch zwischen Menschen

- Lokale Kommunikation



- Entfernte Kommunikation via Telefon

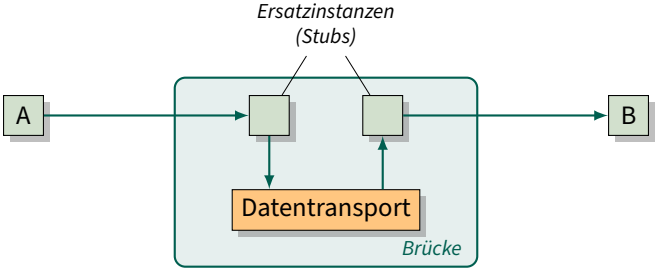


Grobstruktur einer Brücke

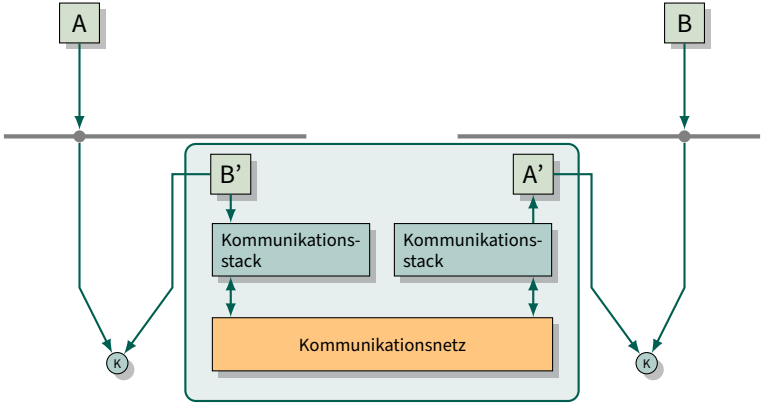
- Lokale Beziehung zwischen Prozessen / Prozeduren



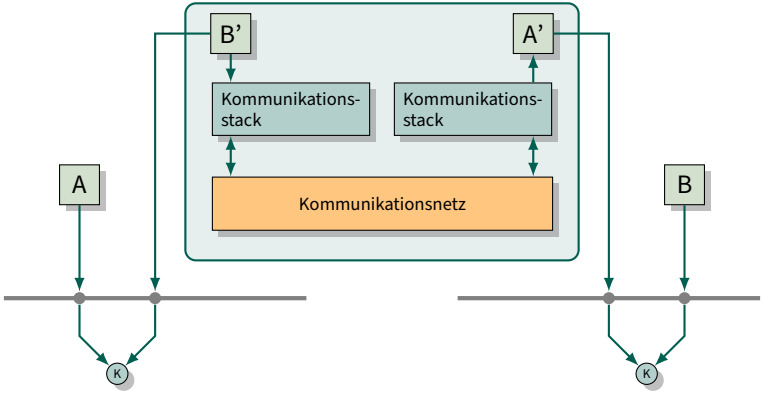
- Beziehung im verteilten Fall



Brücke im Kernverbund



Brücke im Prozessverbund

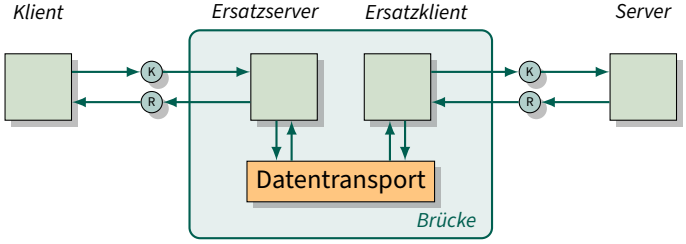


Realisierung der Brücken

- **Kernverbund**
  - Brücken im Kern prozedural realisiert
  - Meist einfach, auf das Nötigste beschränkt
  - Meist nicht oder kaum fehlertolerant
  - Häufig in Parallelrechnern
- **Prozessverbund**
  - Brücken durch Prozesse realisiert
  - Meist flexibel, mächtig, aufwendig
  - Meist fehlertolerant
  - Häufig in heterogenen verteilten Systemen

Dienstleistungsbeziehung im verteilten Fall

- Bei Client-Server-Strukturen ist in der Regel eine Kommunikation in beiden Richtungen erforderlich
- Eine Brücke muss in diesem Fall Ersatzinstanzen bereitstellen, die in beide Richtungen kommunizieren



Aufgaben der Ersatzinstanzen

- Exakte Nachbildung der Schnittstelle (Syntax und Semantik)
- Ein- und Auspacken der Parameter (Marshaling/Demarshaling)
- Behandlung von Wert- bzw. Referenzübergabe
- Umgang mit Heterogenität (z.B. Wandlung von Formaten)

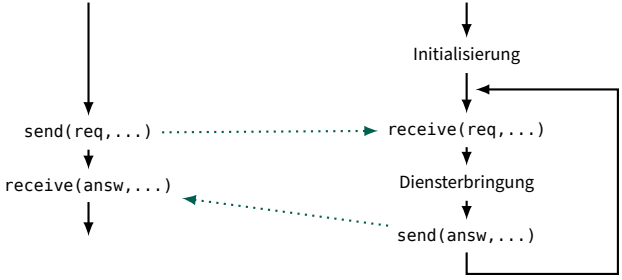
- Für bestimmte Fälle können Ersatzinstanzen automatisch generiert werden (stub generator)
- In komplexeren Fällen sind Ersatzinstanzen umfangreichere Gebilde (z.B. mehrere Prozesse)

4.4 Beispiel RPC

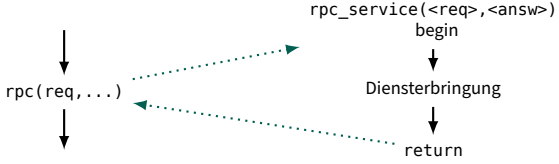
- Realisierung von Dienstleistungsbeziehungen durch Prozesskommunikation ist für Anwendungsprogrammierer etwas umständlich
- Im lokalen Fall sind Aufträge an das Betriebssystem i.d.R. hinter Bibliotheksprozeduren versteckt
- **Idee**: Übertragung auf verteilten Fall ➡ Dienstleistung, die auf einem anderen Rechner erbracht wird, wird durch Prozeduraufruf zur Verfügung gestellt
- Mechanismus ist als **Remote Procedure Call** (RPC) bekannt
  - Meistverwendete Realisierung von Dienstleistungsbeziehungen in verteilten Systemen
  - Eigentliche Interprozesskommunikation zwischen Rechnern bleibt dann unter RPC-Schnittstelle verborgen

Client-Server-Beziehung mit Prozesskommunikation

- Explizite Kommunikation



- RPC



Parameterübergabe

- Wertübergabe (call by value)
  - Standardfall, einfach zu realisieren
  - Werte werden eingepackt und im Nachrichtenpaket verschickt
- Referenzübergabe (call by reference)
  - Referenzübergabe schon im lokalen Fall schwierig ➡ verschiedene Adressräume möglich
  - im verteilten Fall vermeiden
- Ersatz durch call by copy/restore (auch: call by value/result)
  - Bei Aufruf: Übertragung zum Server wie bei Wertaufruf
  - Nach Beendigung: Zurückkopieren zum Client, Überschreiben der alten Werte
  - **Achtung:** andere Semantik

Beispiel Parameterübergabe

- Entfernte Prozedur:

```
procedure p(var integer x, var integer y)
   begin
     x := x+1;
     y := y*y;
   end;
```

- Aufruf:

```
i := 4;
p(i,i);
print(i);
```

- Ergebnis:
  - Call-by-reference: **25**
  - Call-by-copy/restore: **???**

RPC in heterogener Umgebung

- Rechner verwenden unterschiedliche Zahl- und Zeichendarstellungen („little endian“/„big endian“, EBCDIC/ASCII, ...)
- Konvertierung erforderlich
- Übertragung entweder...
  - direkt in das Format des Kommunikationspartners
    oder
  - in kanonisches Format
- Wird in der Regel automatisch durchgeführt

RPC in heterogener Umgebung (Forts.)

- Darstellung von `0x1A2B3C4D` ab Adresse `10000`
- Kleinste Adresseinheit ist ein Byte

	Big Endian		Little Endian	
Adresse	**Hex**	**Binär**	**Hex**	**Binär**
10000	1A	00011010	4D	01001101
10001	2B	00101011	3C	00111100
10002	3C	00111100	2B	00101011
10003	4D	01001101	1A	00011010

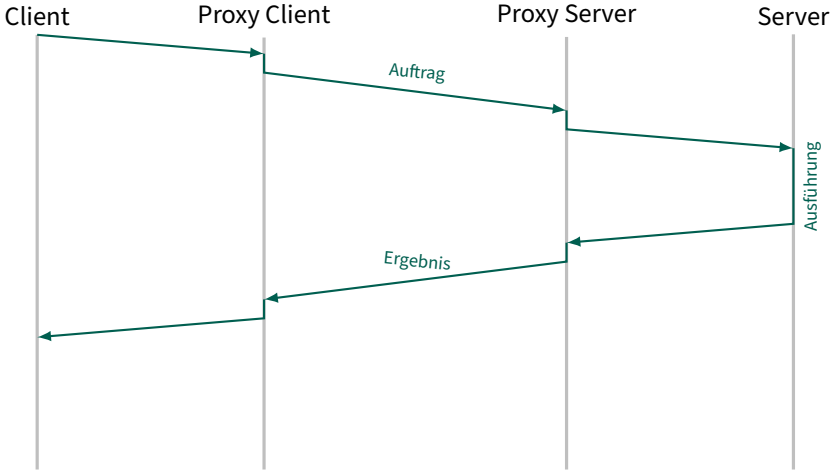
- **Big-Endian-Architekturen**: IBM Großrechner, Motorola 680X0, SPARC
- **Little-Endian-Architekturen**: Intel x86, VAX, Transputer

RPC – Fehlersituationen

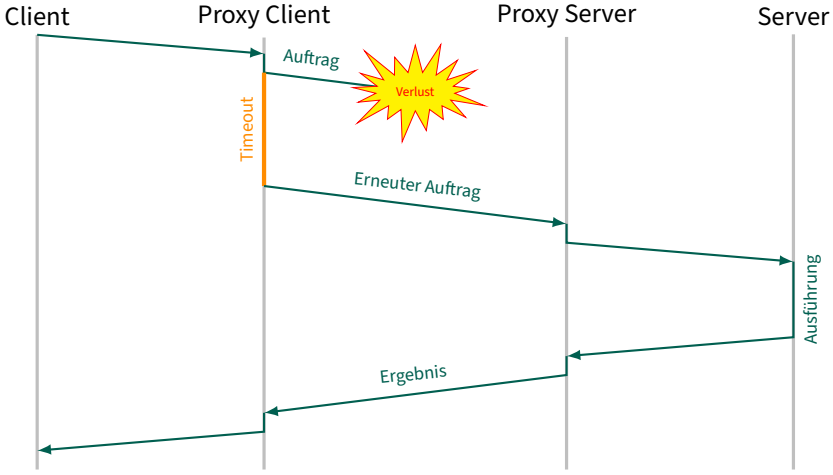
- Verlust von Auftragsnachrichten
- Verlust von Ergebnisnachrichten
- Server-Ausfall
  - Vor Auftragsausführung
  - Nach Auftragsausführung

- **Allgemeine Lösung:** Client Proxy verwendet Frist (time out) und wiederholt Auftrag gegebenenfalls
- **Probleme:**
  - Idempotenz der Server-Operation: Ist ein evtl. mehrmaliges Ausführen der Operation tolerierbar?
- Unterscheidung:
  - **at-most-once-Semantik**: Auftrag wird **höchstens** einmal ausgeführt
  - **at-least-once-Semantik**: Auftrag wird **mindestens** einmal ausgeführt
  - **maybe-Semantik**: keine Aussage möglich

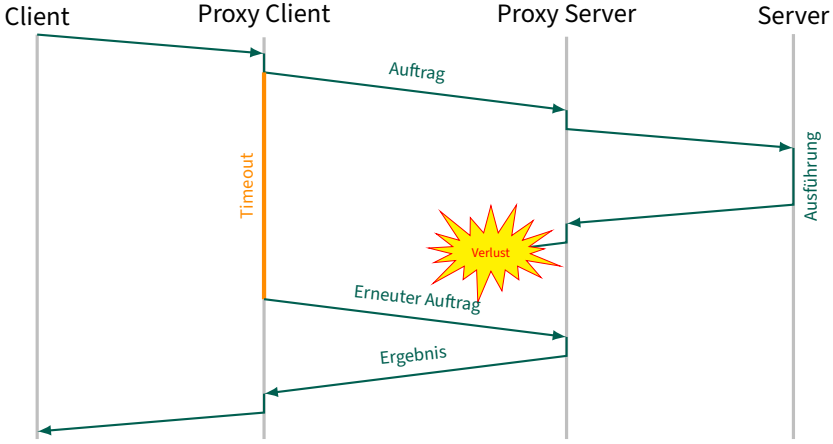
RPC-Ablauf: Fehlerfreier Fall



RPC-Ablauf: Verlust der Auftragsnachricht

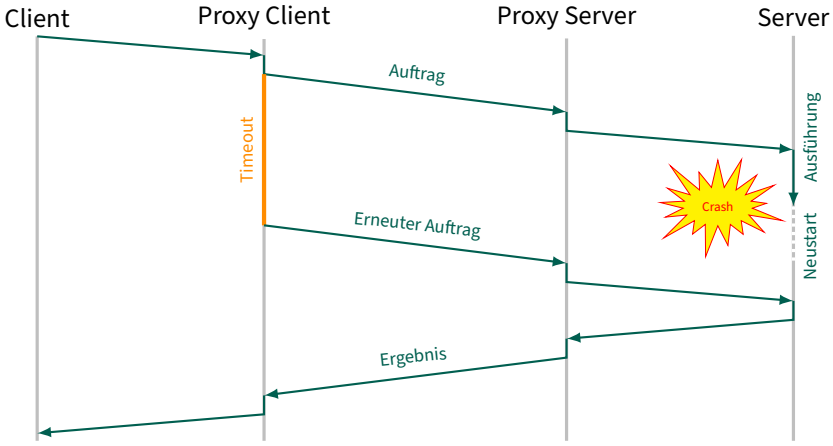


RPC-Ablauf: Verlust der Ergebnisnachricht



- Erfordert Nummerierung der Auftragsnachrichten und „Gedächtnis“ auf Serverseite

RPC-Ablauf: Server-Ausfall nach Ausführung



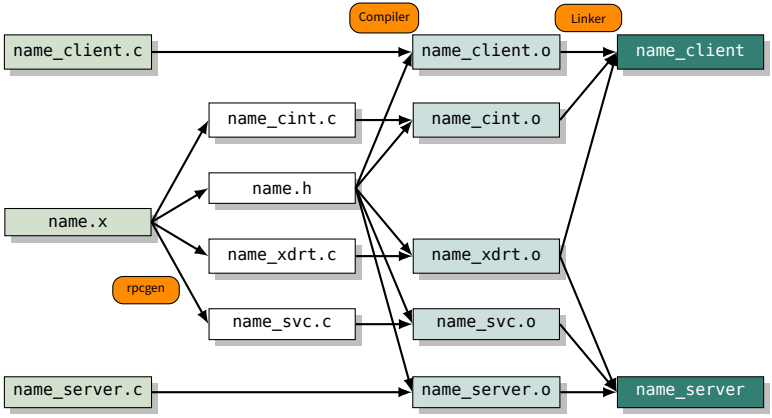
- Erfordert Nummerierung der Nachrichten und persistente Protokollierung auf Festplatte (Serverseite)

Implementationsvariante: Sun RPC

OSI-Modell	Sun RPC
Anwendung	RPC
Darstellung	XDR
Sitzung	–
Transport	UDP oder TCP
Vermittlung	IP
Sicherung	IEEE 802 (z.B. Ethernet)
Bitübertragung	–

- Durch `rpcgen` automatische Generierung von
  - Client Proxy
  - Server Proxy
- Automatische Hin- und Rücktransformation der Daten in XDR-Format
- Wahl des Timeout-Werts
- Wahl der Zahl der Wiederholungen
- Semantik abhängig von gewähltem Transportprotokoll:
  - UDP: maybe-Semantik
  - TCP: at-most-once-Semantik

Automatische Generierung mit rpcgen



- Eine solche Generierung mit Hilfe einer **Interface Description Language (IDL)** wird häufig genutzt (Corba, DCE, XCB, …)

Diskussion

- RPC bietet eine einfache und komfortable Programmierschnittstelle, jedoch keinerlei Parallelität (Nebenläufigkeit) zwischen Client und Server
- Leistungsfähigere Dienstleistungsstrukturen durch allgemeines Kanalkonzept
- Im verteilten Fall ist die Kanal-Kommunikation durch entsprechende Brücken zu ergänzen
  - Im Prozessverbund ist die Brücke durch Prozesse und lokale Kanäle realisiert
  - Im Kernverbund ist die Brücke im Kern angesiedelt: Eine der beiden Kommunikationsoperationen greift über die Brücke auf einen Kanal auf einem anderen Rechner zu

Literatur

[ADG90] Autenrieth, K., H. Dappa und M. Grevel: *Technik verteilter Betriebssysteme*. Hüthig, 1990 , Anhang A

[TaSt02] Tanenbaum, A. S. und M. van Steen: *Distributed Systems*. Prentice Hall, 2002 , Chapter 1

[Sin97] Sinha, P. K.: *Distributed Operating Systems*. IEEE Press, 1997 , Chapter 4