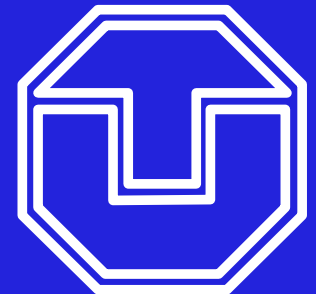


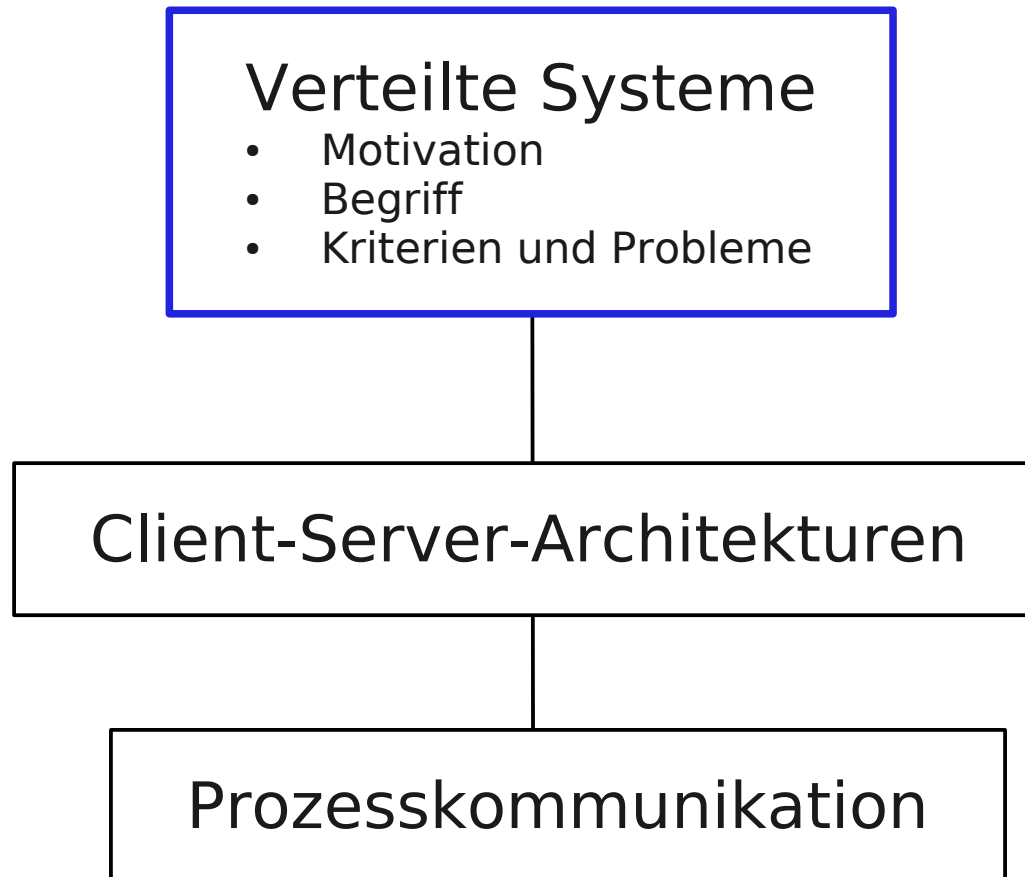
Client-Server-Architekturen und Prozess-Kommunikation in verteilten Systemen

Betriebssysteme

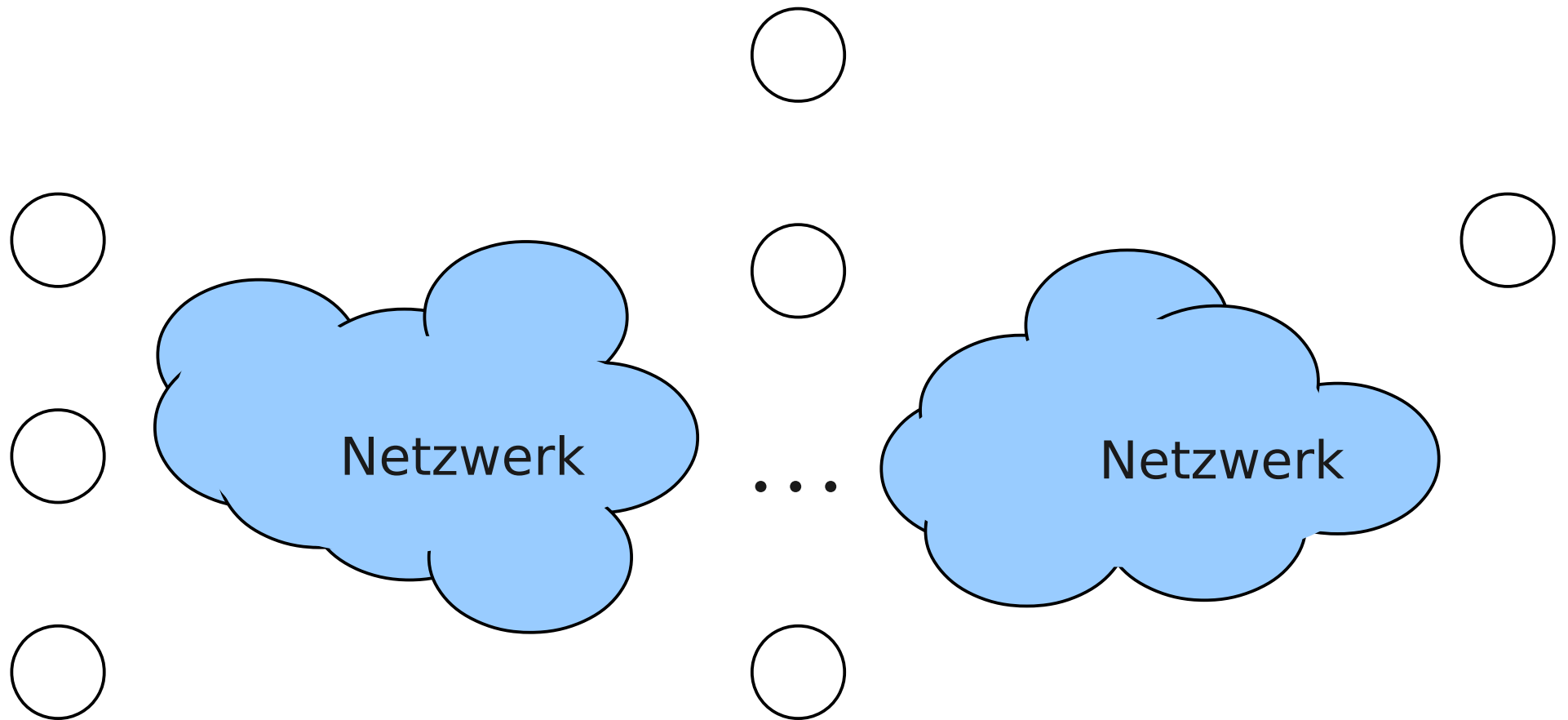
Hermann Härtig
TU Dresden



Wegweiser

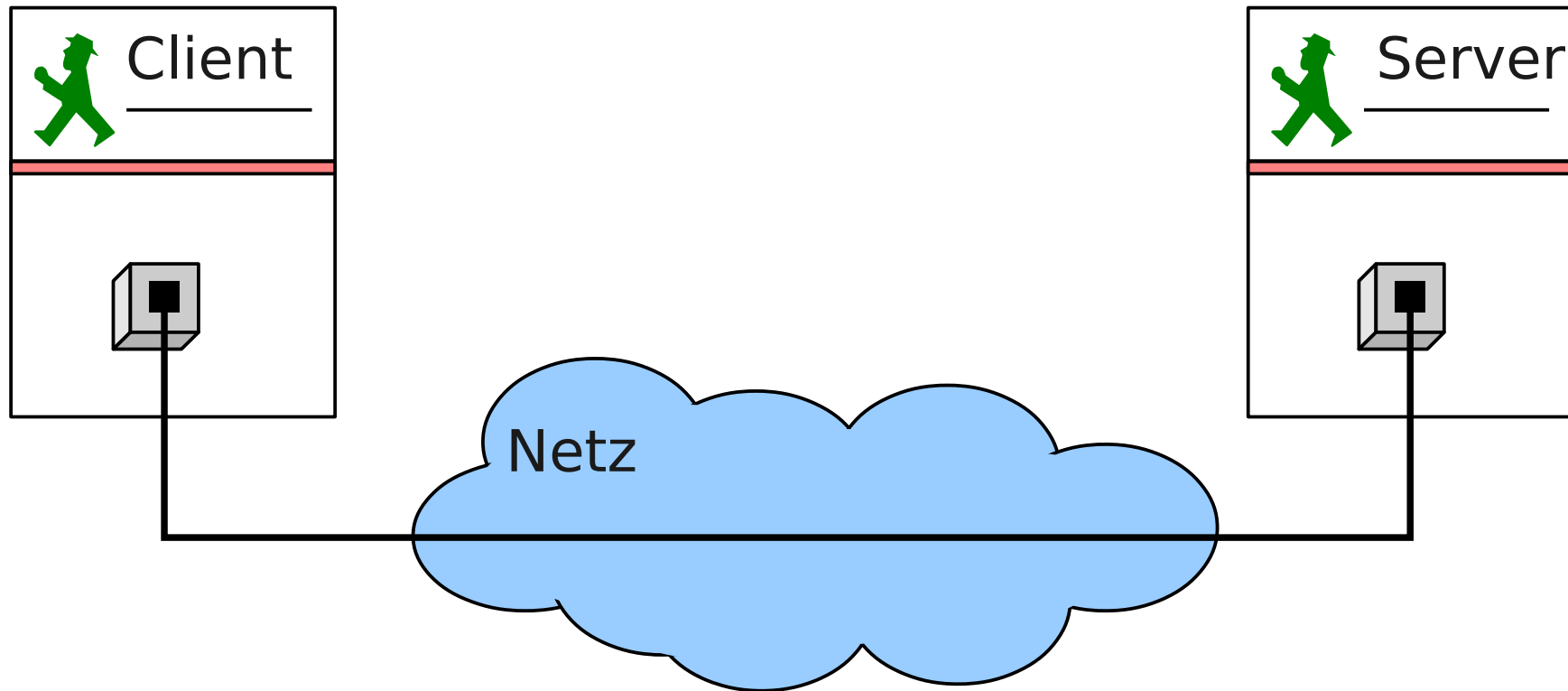


Ausgangspunkt 1: Netzwerke



Zustellen von Botschaften

Sockets



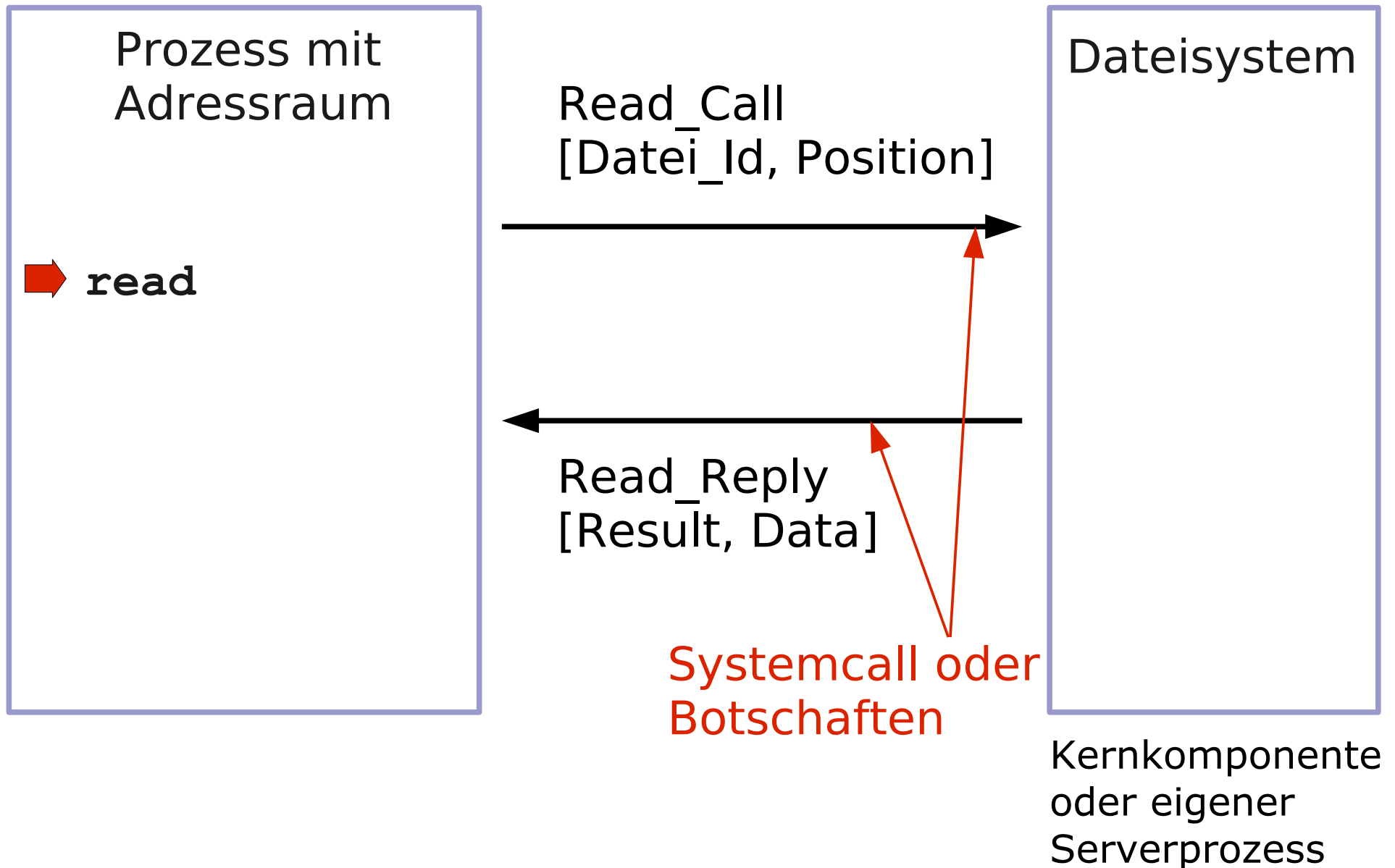
Client

Server

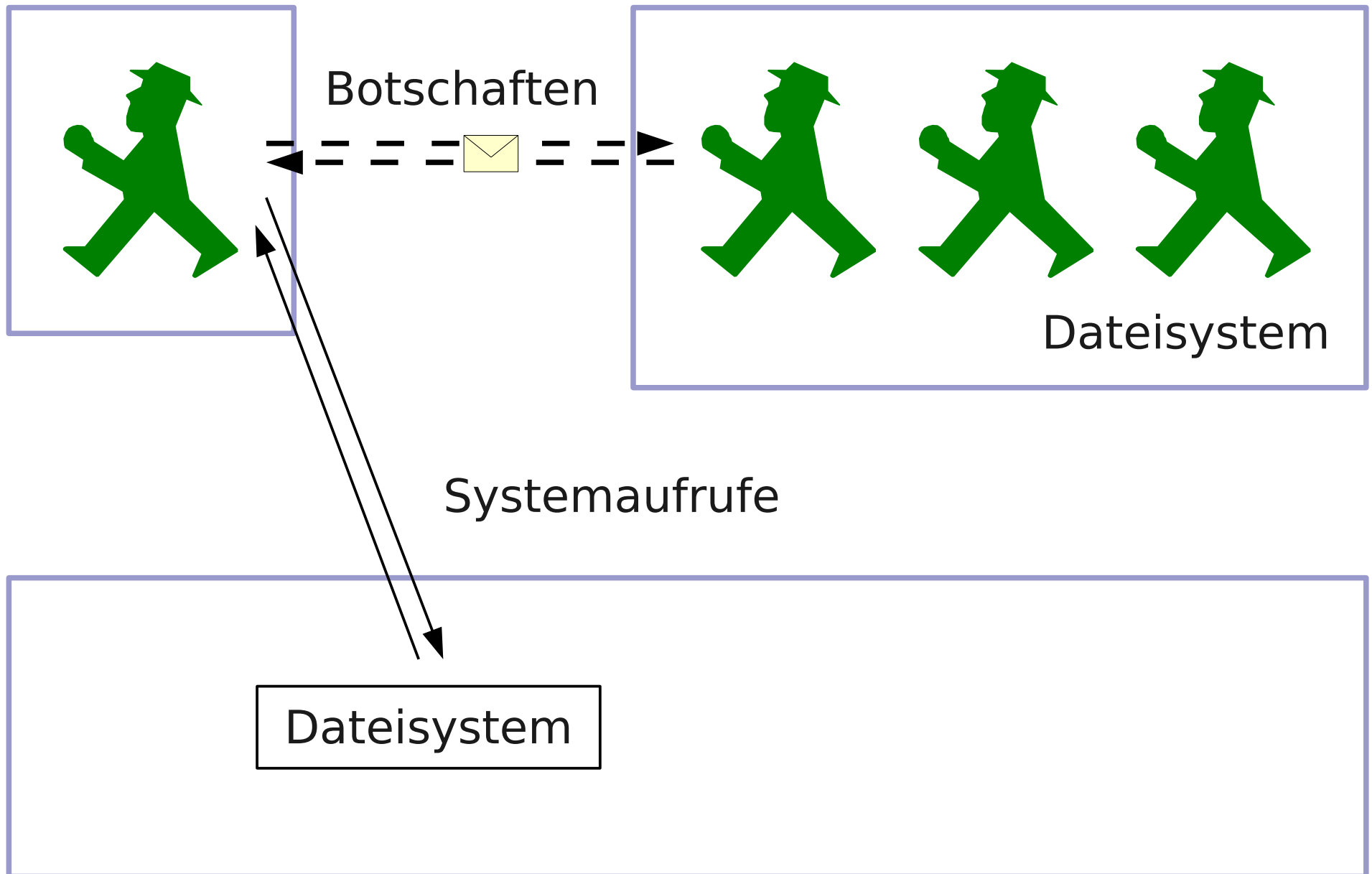
```
create sock(protocol type)
connect(ExampleAddress)
```

```
create sock(protocol type)
bind(ExampleAddress)
listen
accept
```

Zugriff mittels Kopieren (read/write)



Systemarchitekturen: Prozeduraufrufe oder Botschaften



Verteilte Systeme

Ansammlung autonomer Rechner, die dem Benutzer als ein System präsentiert werden.

(Tanenbaum)

Ansammlung autonomer Rechner, die über ein Netzwerk verbunden sind und mit verteilter Systemsoftware ausgestattet sind.

(Colouris)

You know you have one when the crash of computer you have never heard of stops you from getting any work done

(Leslie Lamport)

Verteilte Systeme – Begriff

Ein verteiltes System liegt vor, wenn mehrere Rechner eine gemeinsame Aufgabe übernehmen.

Vier Problembereiche:

- voneinander unabhängige Ausfälle
- unzuverlässige Kommunikation
- unsichere (insecure) Kommunikation
(Mithören, Manipulieren)
- teure Kommunikation
(Bandbreite, Latenz, Kosten)

(Michael D. Schroeder)

Verteilte Systeme – Kriterien

Gründe für den Einsatz verteilter Systeme

- inhärent verteilter Charakter der Anwendungen
- Rechenleistung (Wirtschaftlichkeit)
- gemeinsame Nutzung von Betriebsmitteln (resource sharing)
- Ausfallsicherheit

Verteilte Systeme – Kriterien

Beurteilungskriterien

- **Transparenz:** Orts-, Zugriffs-, Namens-Transparenz
Replikations-, Migrations-Transparenz
- Skalierbarkeit
- Leistung
- Administrierbarkeit (Betriebsmittelnutzung)
- Zuverlässigkeit (Umgang mit Fehlern, Ausfällen)
- Sicherheit (gegen Angriffe)
- Offenheit

Einige Begriffe und Abkürzungen

IPC	Inter Process Communication
RPC	Remote Procedure Call
lokal	auf einem Rechner
entfernt (remote)	über Rechnergrenzen

Offenheit durch Prozesskommunikation

- Funktionalität eines Systems wird nicht durch statischen Kern implementiert, sondern durch eine Menge von Benutzerprozessen
 - z. B. Dateisystemaufrufe nicht durch Kernaufruf, sondern durch Botschaft an Prozess
- wenn Prozesskommunikation netzwerktransparent, dann fällt Zugriffstransparenz dabei ab
 - (auch) daraus resultiert die große Bedeutung der IPC
- offene **verteilte** Systeme sind charakterisiert durch
 - einen einheitlichen Prozesskommunikations-Mechanismus
 - Veröffentlichung der wesentlichen Schnittstellen
 - Heterogenität von Hardware und Software

Neue, durch Verteilung verursachte Probleme

Neue Fehlertypen

- Botschaften können verloren gehen oder verfälscht werden
- Ausfall von Rechnern oder Netzwerk

Neue Angriffsmöglichkeiten

- Botschaften werden unterdrückt, mitgehört, manipuliert, wiederholt
- Absenderangaben werden gefälscht
- Rechner werden mit Botschaften bombardiert

Neues Betriebsmittel Netzwerke

- Netz als Flaschenhals; Bandbreite, Latenz, Kosten
- Adressierung: Finden des richtigen Rechners bzw. Ports

Heterogenität

- Hardware, Betriebssystem

Beispiel für neue Fehlersituation

- Beispiel: Nutzerprogramm nutzt entferntes Dateisystem
 - lokal : Programm allein fällt aus oder Programm **und** Dateisystem
 - verteilt: das entfernte Dateisystem fällt (allein) aus

Denkbare Varianten

- Programm fällt auch aus
- Programmierer muss neue Fehlersituation berücksichtigen

Eine grundsätzliche Überlegung (1)

Gegeben folgendes Problem

- p, q Prozesse,
fallen nicht aus,
kommunizieren via send/receive
Botschaften können verloren gehen
keine Annahme über Laufzeit der Botschaften möglich
- a, b mögliche Operationen von p, q
- p, q sollen entweder:
(i) dieselbe (nichtriviale) Operation einmal ausführen
(ii) keine Op. ausführen
- Gibt es eine Botschaftenfolge, die das leistet ?

Eine grundsätzliche Überlegung (2)

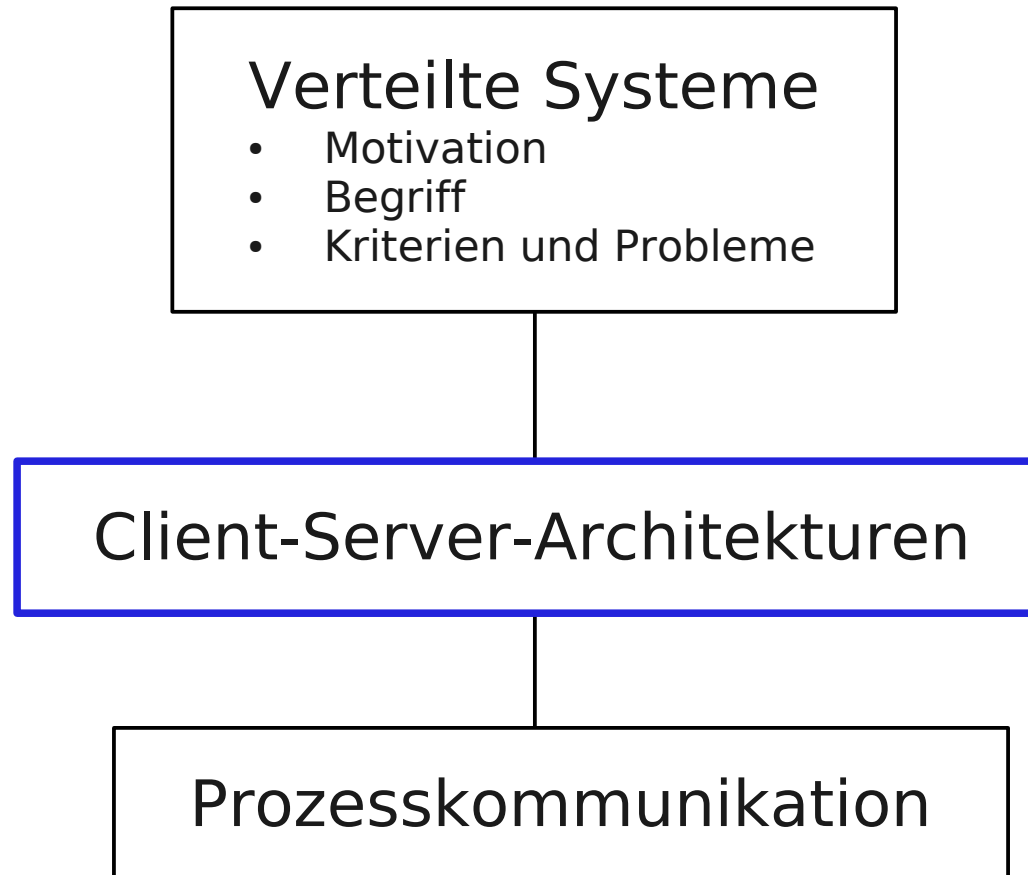
Behauptung

Es gibt kein solches Protokoll, das mit endlich vielen Botschaften auskommt!

Beweis: Angenommen, es gebe solche Protokolle

- bestehend aus Folgen von Botschaften: $(m_{p \rightarrow q}, m_{q \rightarrow p})^*$.
 - Wähle: Protokoll mit geringster Zahl von Botschaften.
 - o. B. d. A.: $\underline{m}_{p \rightarrow q}$ sei letzte Botschaft dieses Protokolls, dann kann p eine solche Entscheidung nicht auf der Annahme aufbauen, dass $\underline{m}_{p \rightarrow q}$ ankommt.
 - dito für q.
 - Also kann $\underline{m}_{p \rightarrow q}$ auch wegfallen.
- Widerspruch, da offensichtlich nicht kleinstes Protokoll!

Wegweiser

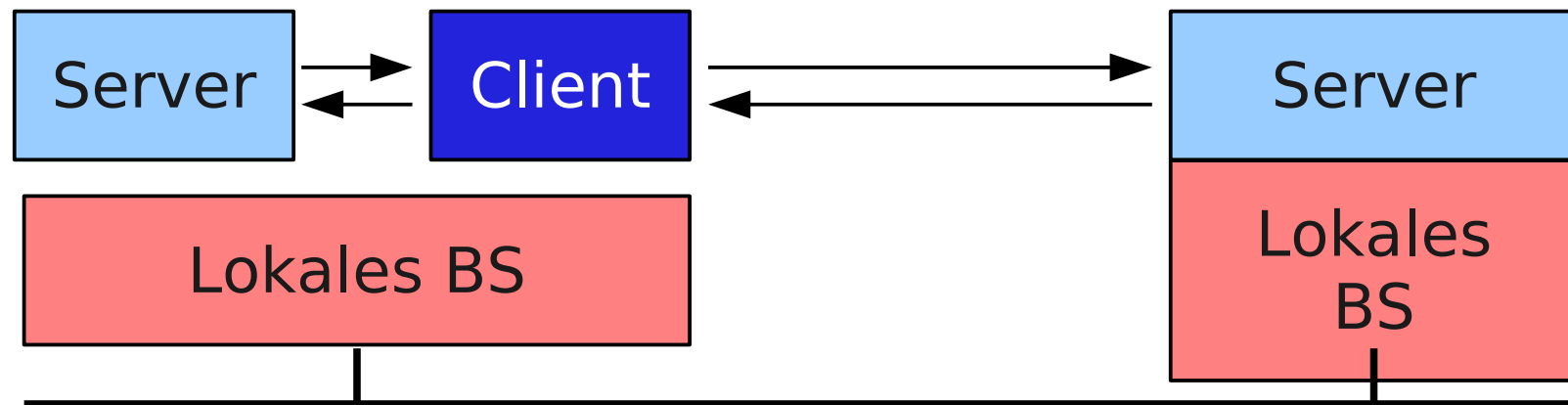


Konstruktionsprinzipien für verteilte Systeme

- Client-Server-Architekturen
 - ◊ Dienste werden durch Serverprozesse erbracht
 - ◊ Klienten nehmen diese durch IPC in Anspruch
- Kommunikation basierend auf Botschaften:
Zustellung der Botschaften über Netzwerk
(z.B. via Sockets)
- Kommunikation basierend auf („Prozedur“-) Aufrufen:
Simulation durch Botschaften
 - ◊ Verpackung der *In*-Parameter in Botschaft
 - ◊ Versenden, Durchführen der Operation auf der anderen Seite
 - ◊ Verpackung der *Out*-Parameter und Zurücksenden

Client-Server-Architekturen

- Strukturierung von (Betriebs-) Systemen als eine Menge kooperierender Prozesse (Server), die Dienste anbieten
- Benutzer-Prozesse (Clients) nehmen Dienste durch IPC in Anspruch



- mit oder ohne Netzwerk
- Mikrokern oder monolithisches System

Mikrokernbasierte Betriebssysteme

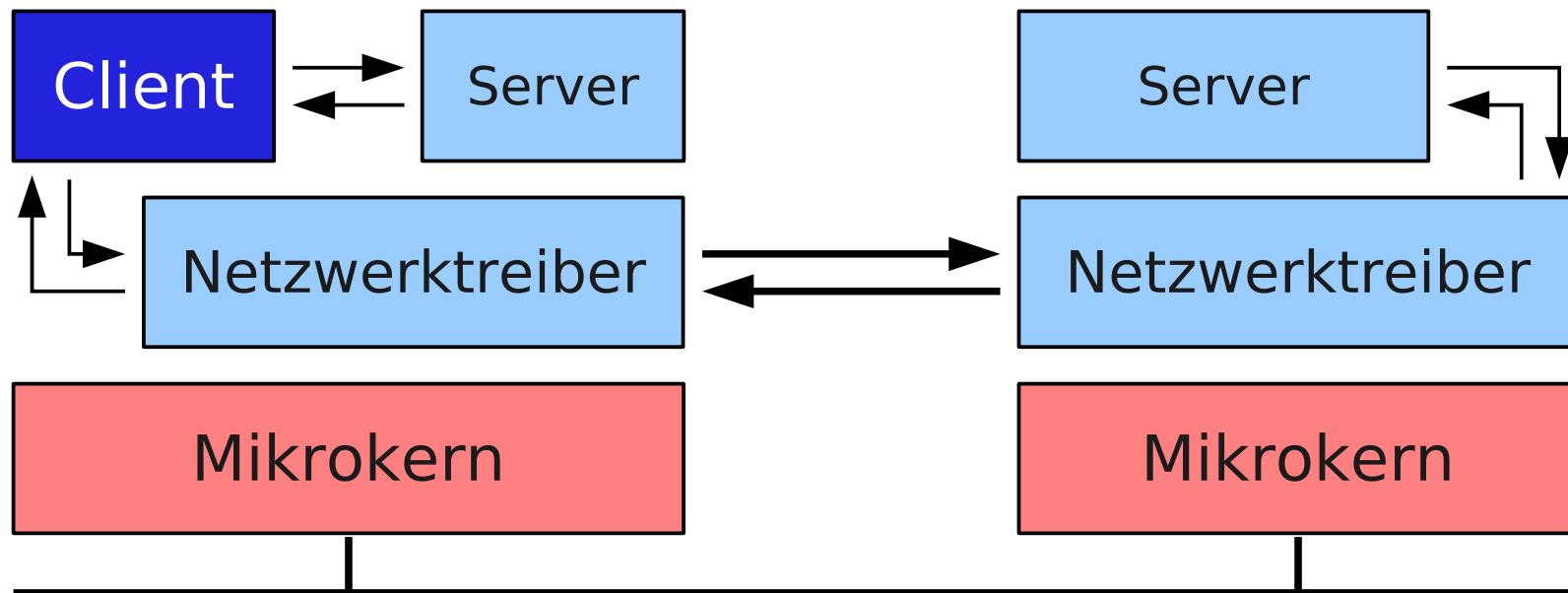
- Funktionalität grundsätzlich per Benutzerprozess (mit eigenem Adressraum) und mittels IPC
 - ◊ Dateisysteme
 - ◊ Netzwerkprotokolle
 - ◊ Treiber ...

Vorteile

- Offenheit
- Fehlerisolierung

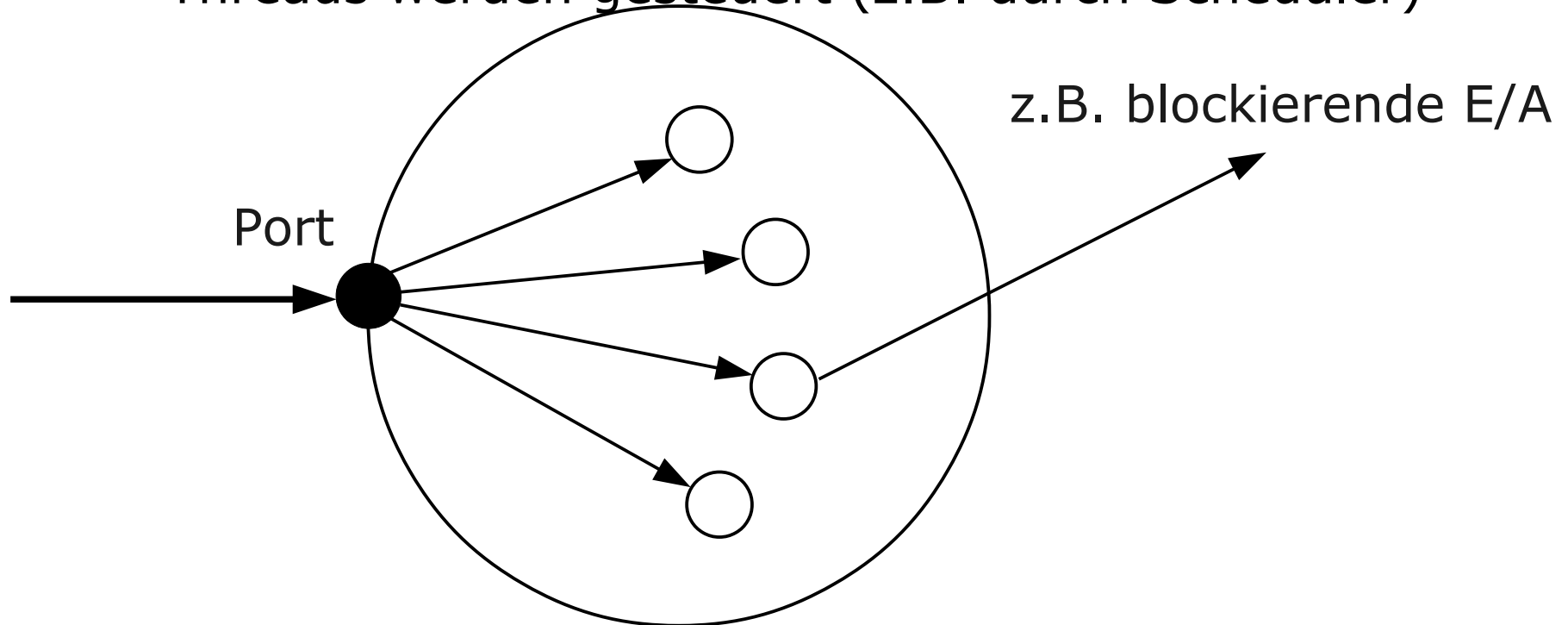
Nachteil

- Effizienz



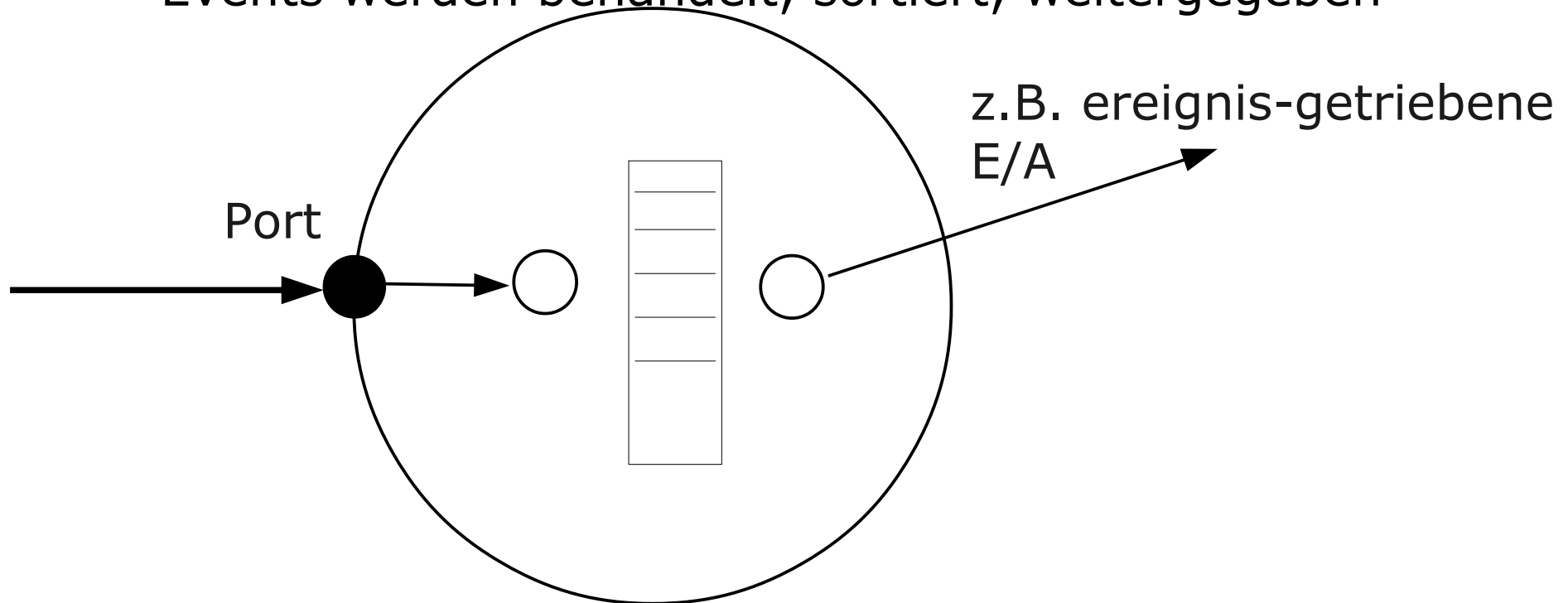
Einsatz vieler Threads

- Server müssen in der Lage sein, viele RPCs überlappend auszuführen
- alle Betriebsmittel eines Servers in einem Adressraum
- Grundidee:
 - viele Threads pro Adressraum
 - Threads werden gesteuert (z.B. durch Scheduler)

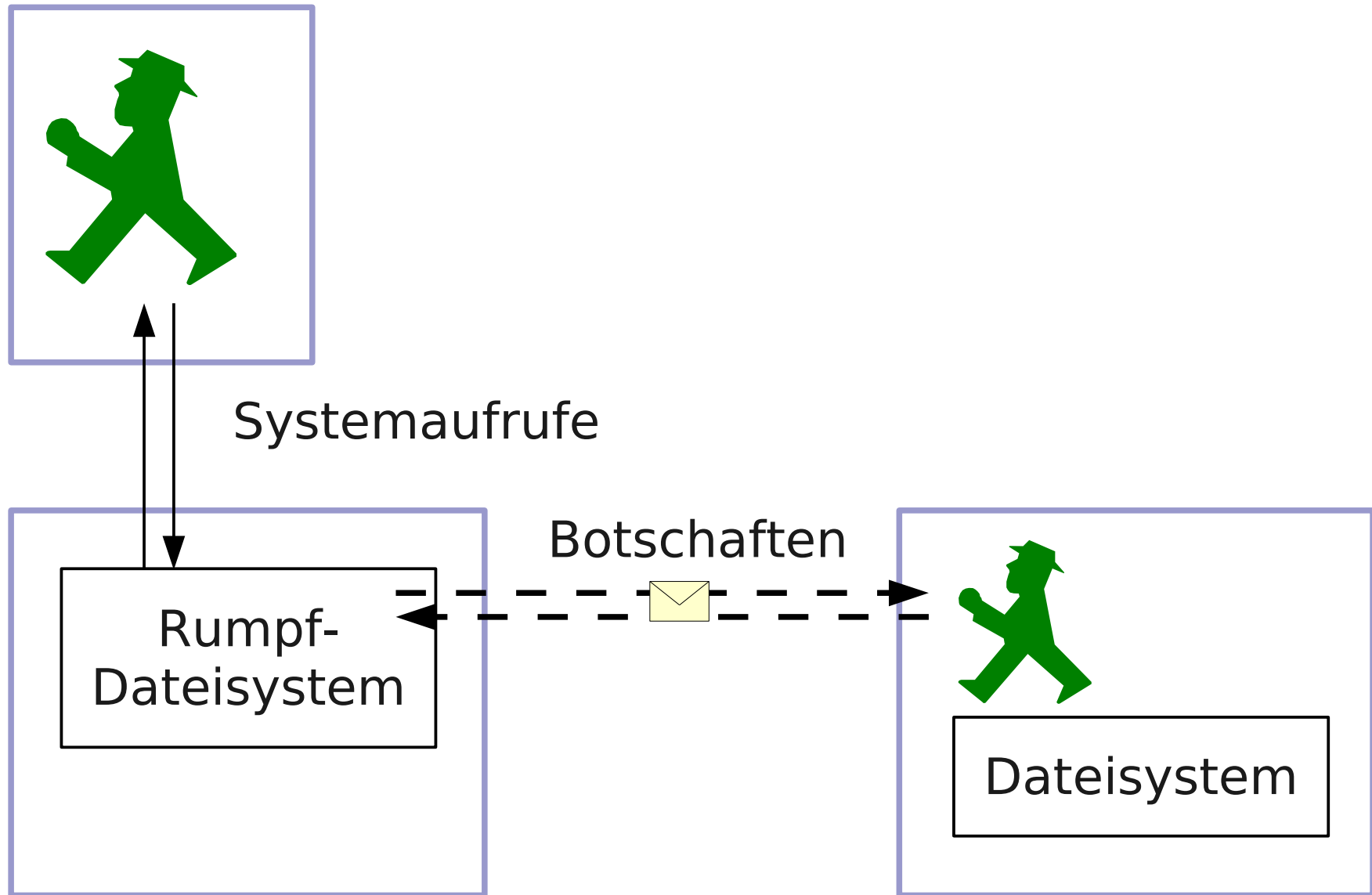


Abarbeitung per Ereignissen (Events)

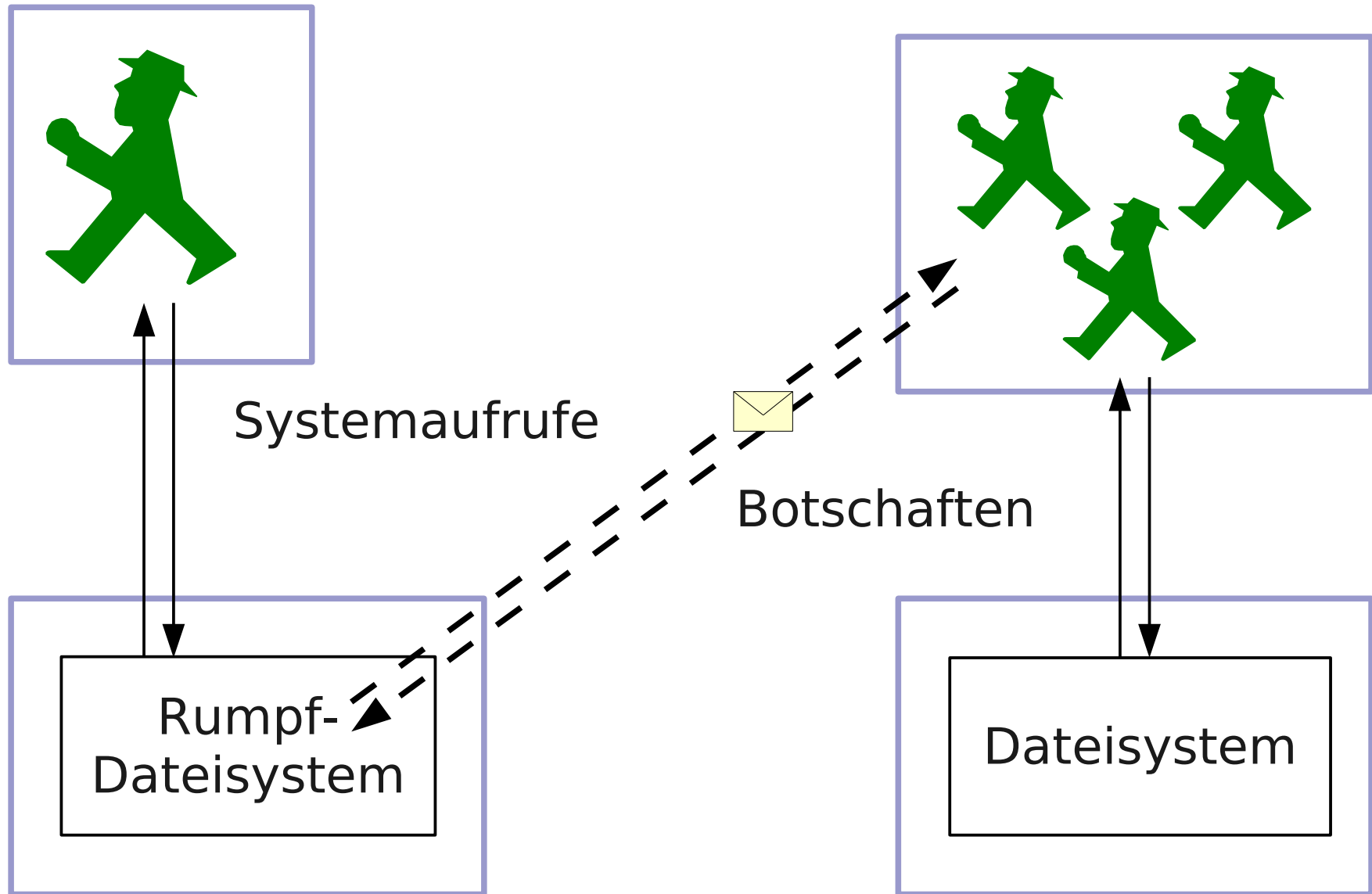
- Server müssen in der Lage sein, viele RPCs überlappend auszuführen
- alle Betriebsmittel eines Servers in einem Adressraum
- Grundidee:
 - ◊ wenige Threads
 - ◊ Events werden behandelt, sortiert, weitergegeben



Hilfsthread im Kern



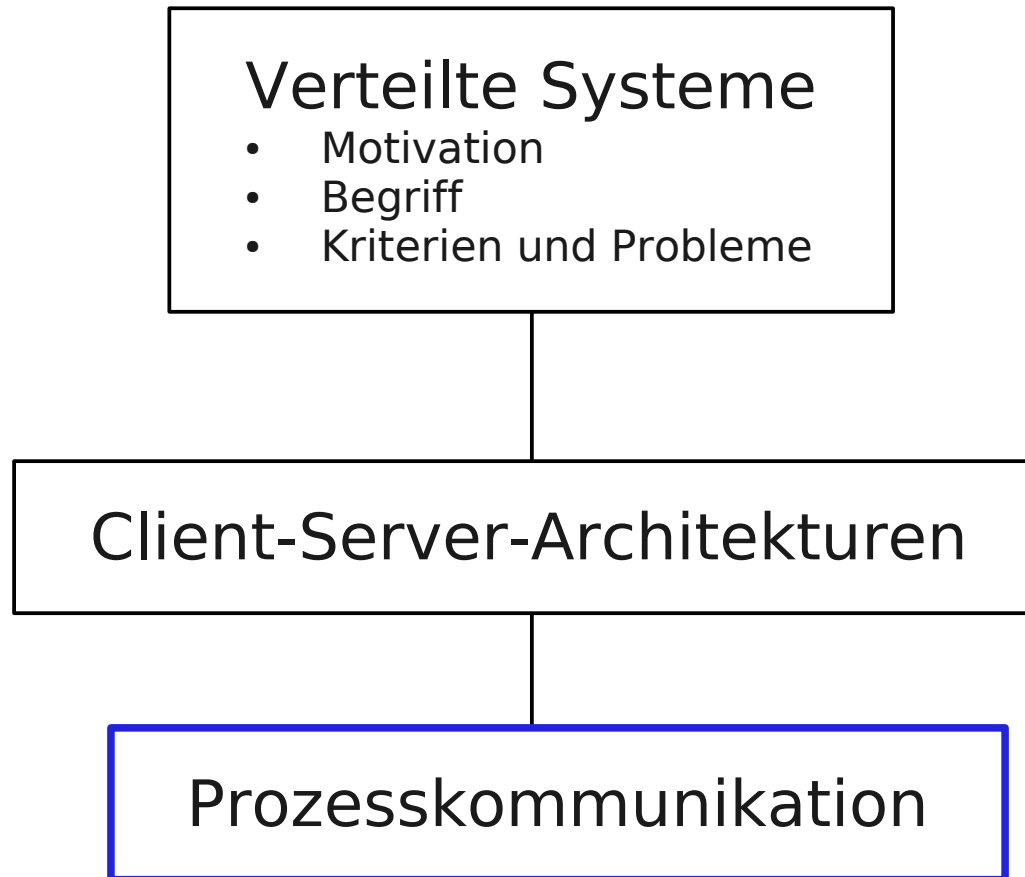
Hilfsprozess



Peer-To-Peer Systeme

- Ziel und Charakteristika:
 - Nutzung / Kooperation zahlreicher Rechner im Internet
 - keine ausgewiesene und (wohl-)administrierte Server
 - zuverlässige, gemeinsame Nutzung („sharing“) verteilter und potentiell unzuverlässiger Rechner
- Anwendungen:
 - gemeinsame Nutzung von Dateien („File-Sharing“)
 - Web-Caches
- Problembereiche:
 - sehr große Zahlen beteiligter Rechner
 - Lastverteilung
 - große Dynamik in der Verfügbarkeit der beteiligten Rechner
 - Sicherheit („Security“)

Wegweiser



Wegweiser

Fernaufruf (Remote Procedure Call RPC)

- Vorgehen
- Implementation
- Fehlerbehandlung

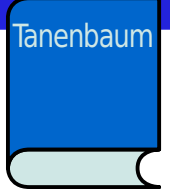
Gruppenkommunikation

Anhang: SUN RPC

Einige Begriffe und Abkürzungen

IPC	Inter Process Communication
RPC	Remote Procedure Call
lokal	auf einem Rechner
entfernt (remote)	über Rechengrenze
unicast	ein Partner
broadcast	an alle (z. B. an alle in einem Teilnetz)
multicast	an einige (z. B. Mitglieder einer Gruppe)

Basismechanismen und Kommunikationsformen



- send (A, M) überträgt Daten an Adressaten
- receive (M) empfängt Daten

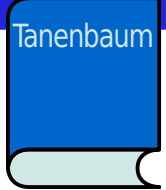
• Datenströme (z. B. Pipes)	Sender	Empfänger
	send (M_1)	receive (M_1)
	send (M_2)	receive (M_2)

	send (M_n)	receive (M_k)

- Fernaufruf (RPC)

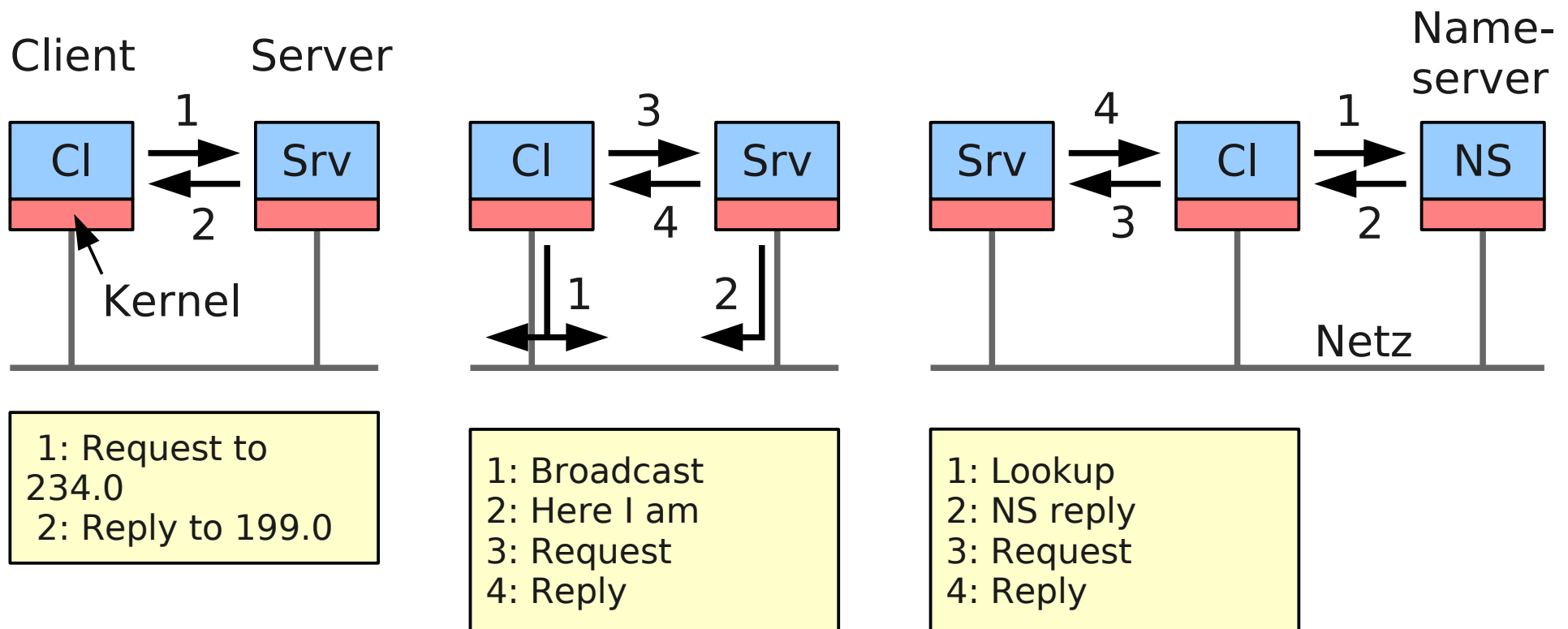
→ Rechnernetze-Vorlesung

Adressierung in Client-Server Systemen



Prinzipielle Alternativen:

- Maschine.Prozess (IP-Adresse.Port)
- Broadcast, z. B. große Zahlen als Adressen
- Nameserver



Fernaufruf (Remote Procedure Call)

Ziel: Client-Server-Architekturen handhabbar machen

Weg: Simulation des lokalen Prozeduraufrufes mittels
Botschaften-System

Client

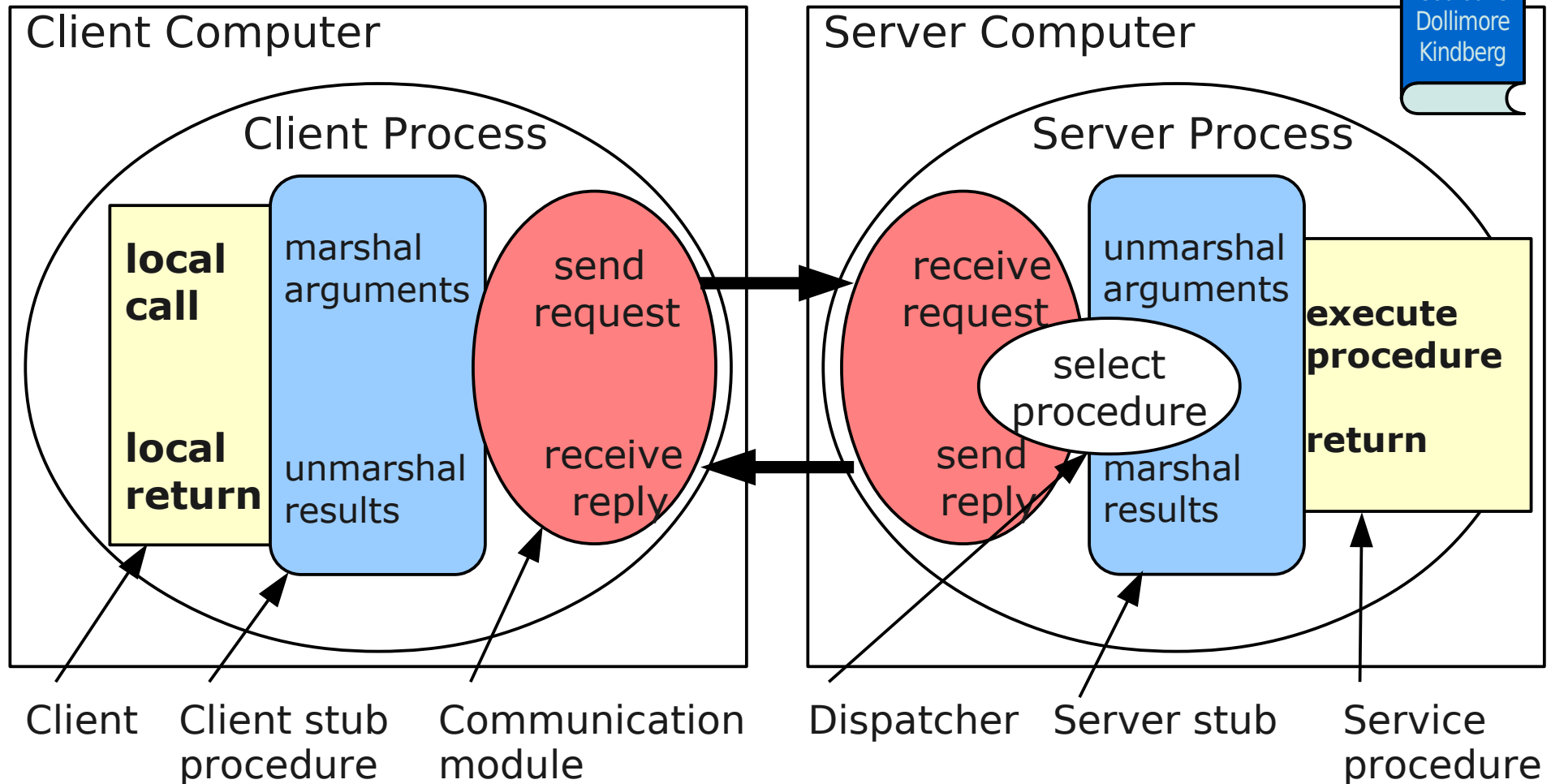
```
send (S, M);  
receive (M2);
```

Server

```
while (1) {  
    wait (M);  
    switch (M.op) {  
        case procl : ...  
  
        send (Client, M2);  
        break; ...  
    }  
}
```

RPC – Implementierung (aus Coulouris et al.)

Coulouris
Dollimore
Kindberg



Aufgaben: Parametereaufbereitung (Marshalling)
 Binden (Auffinden des zuständigen Servers)
 Kommunikation

Ablauf RPC

- Client ruft Client-Stub-Prozedur auf
- Client steckt Parameter in Nachricht, verzweigt in Kern
- Kern sendet Nachricht an Server (*send*)
- Client-Stub: *receive* wird blockiert
- entfernter Kern gibt Nachricht an Server-Stub
- Server-Stub packt Parameter aus, ruft Prozedur auf
- Server-Stub packt Resultat in Nachricht, verzweigt in Kern
- Server-Stub: *send; receive* wird blockiert
- Client-Kern gibt Nachricht an Client-Stub
- Client-Stub packt Resultat aus, übergibt an Client

Unterschiede lokaler RPC – Prozedur

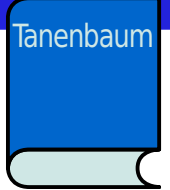
- Parameter
 - Call by Reference (unüblich, nicht unmöglich)
 - statt dessen: copy in, copy out Semantik
- globale Variable
 - anderer Adressraum bei RPC
- komplexe Datenstrukturen als Parameter
- Fehlerisolation
- Sicherheit
- Effizienz
 - ◊ RPC-lokal: ca. 100 Takte auf ALPHA 21164
 - ◊ Prozedur: 0 Takte (bei offenem Einbau)

Fehlerbehandlungen



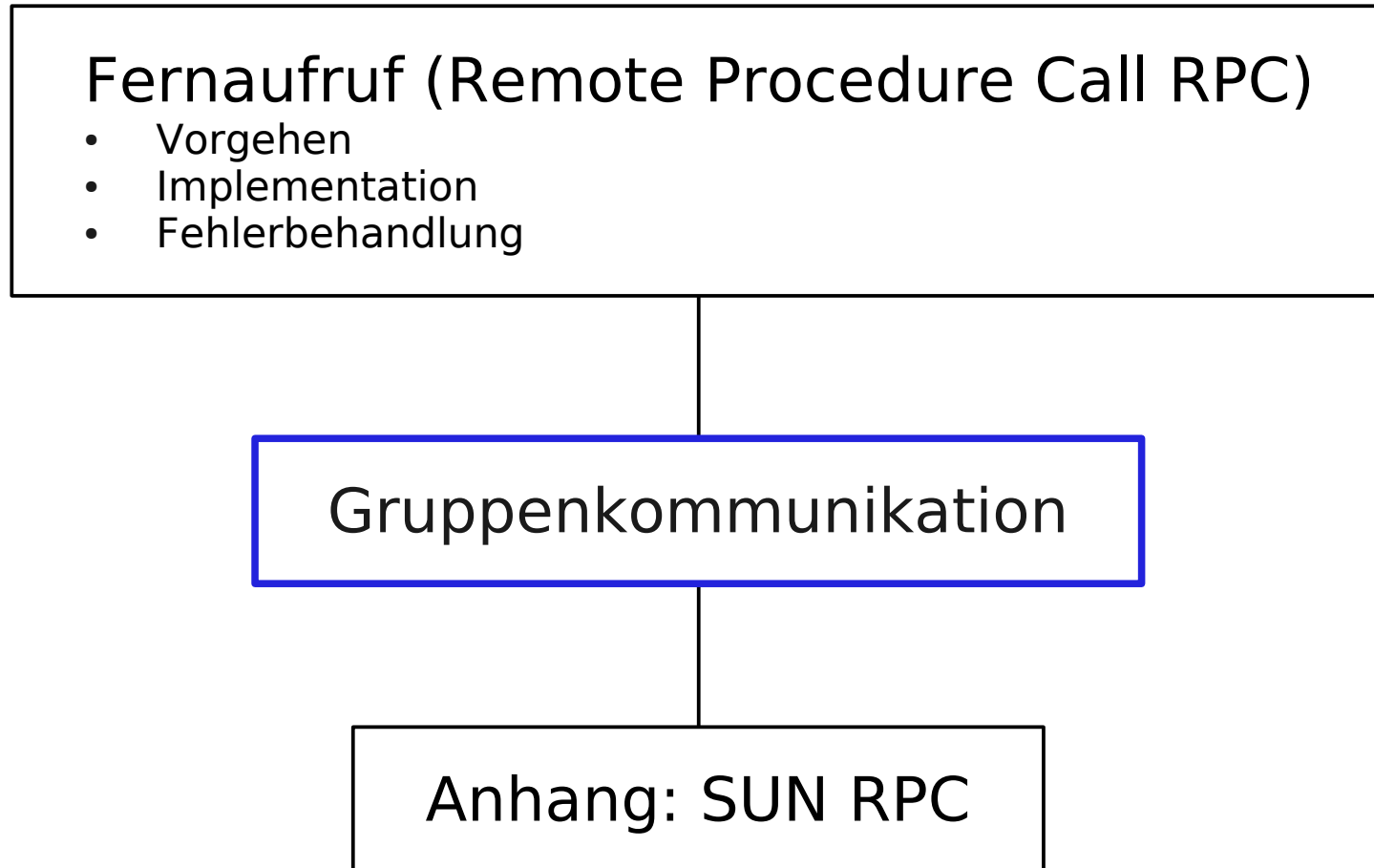
- Fehlersemantik Prozeduraufruf:
exakt ein Aufruf („exactly once“)
- verfälschte Botschaften
erkennbar und korrigierbar durch redundante Codierung
- verlorene Request-Botschaft
 - Timeout
 - Wiederholung Request
- Verlorene Reply-Botschaft
 - Timeout bei Request
 - Wiederholung Request
 - ja: „at least once“ / Duplikation
 - nein: „at most once“

RPC-Fehlersemantik



- May Be
Nachricht genau einmal versenden, keine Kontrolle
- At Least Once (SUN RPC)
 - wiederholtes Senden der Request-Botschaft bis zu einem Reply
 - nur für Anwendungen mit idempotenter Semantik
- At Most Once (Corba)
 - Senden einer Fehlernachricht (Duplikat-Erkennung / Reply-Wiederholung)
 - Birell/Nelson: wenn Server nicht ausfällt und ein Reply kommt, dann „exactly once“-Semantik, sonst unklar

Wegweiser



Gruppen-IPC (multicast)

Gruppe:

Menge von Prozessen, die auf eine vom System oder von Nutzern festgelegte Art zusammenarbeiten

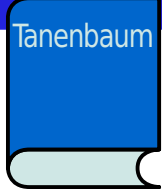
Multicast:

1 Botschaft eines Prozesses an alle Prozesse einer Gruppe

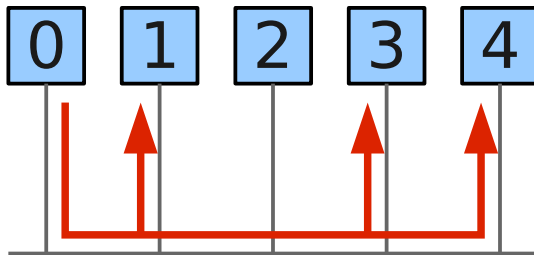
Anwendungen:

- Fehlertoleranz auf Basis replizierter Server
- Auffinden von Objekten in verteilten Anwendungen
- höhere Leistungsfähigkeit auf der Basis replizierter Server
- Konsistenz von Kopien (multiple update)
- Bandbreitenreduzierung (1 mal übertragen statt n mal)

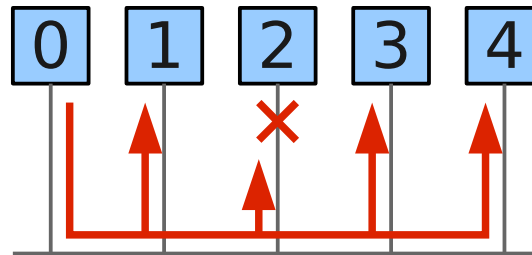
Adressierung von Gruppen



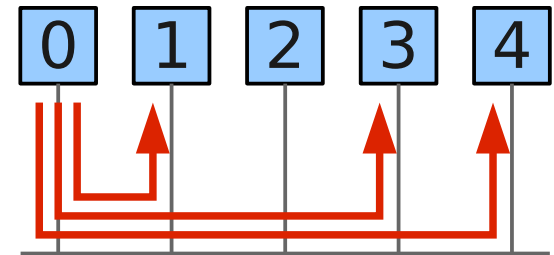
Multicast



Broadcast



Unicast



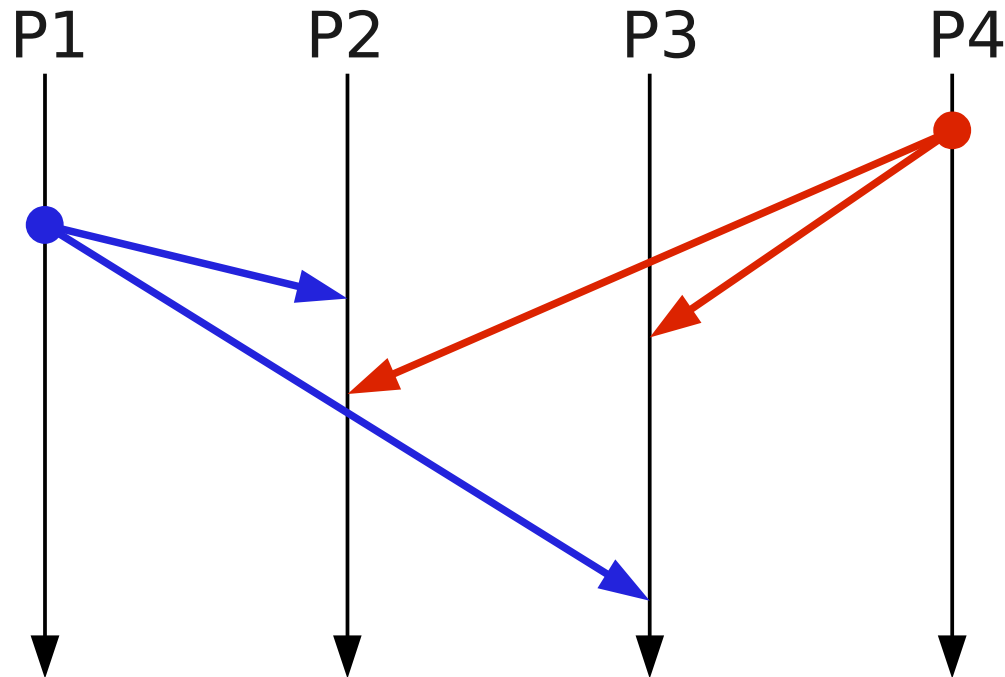
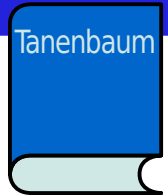
Netzwerknutzung:

nur eine Botschaft über Netz

Atomarität:

alle oder kein Mitglied der Gruppe empfangen Botschaft

Botschaften-Reihenfolge (1)



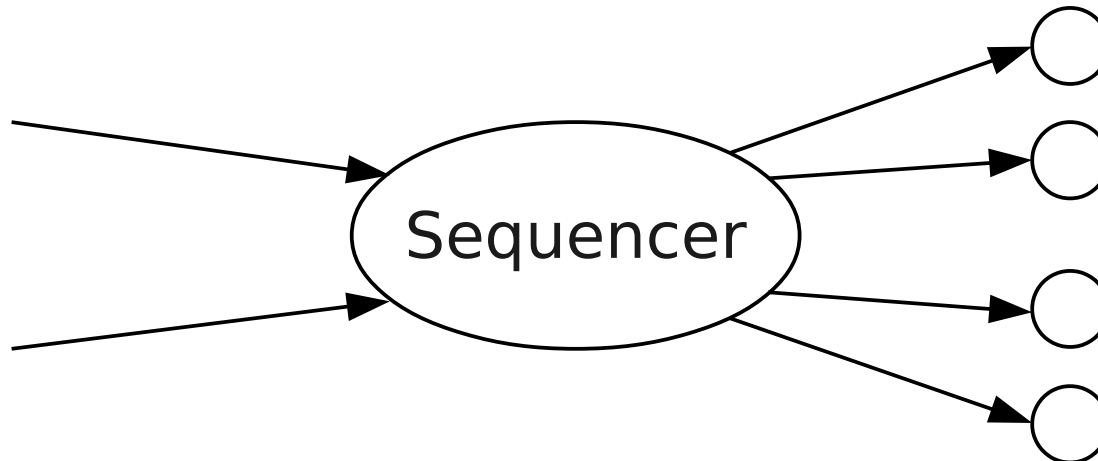
- P1, P4: Klienten
- P2, P3: replizierte Server
- Botschaften : Schreib-Operationen führen zu Konsistenz-Problem

Botschaften-Reihenfolge (2)



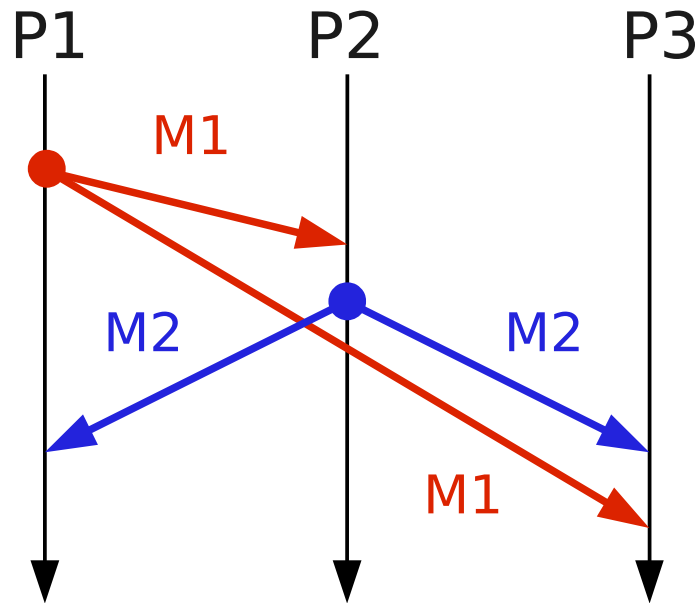
„Totally Ordered Multicast“

- Werden mehrere Botschaften an eine Gruppe gesendet, so werden sie von allen Mitgliedern der Gruppe in gleicher Reihenfolge empfangen
- Mögliche Implementierung:
ein Prozess („Sequencer“) ist für Zustellung zuständig



Botschaften-Reihenfolge (3)

Beispiel für nicht
kausal geordnet:



P1, P2, P3: Teilnehmer an News-Group

M1: Anfrage

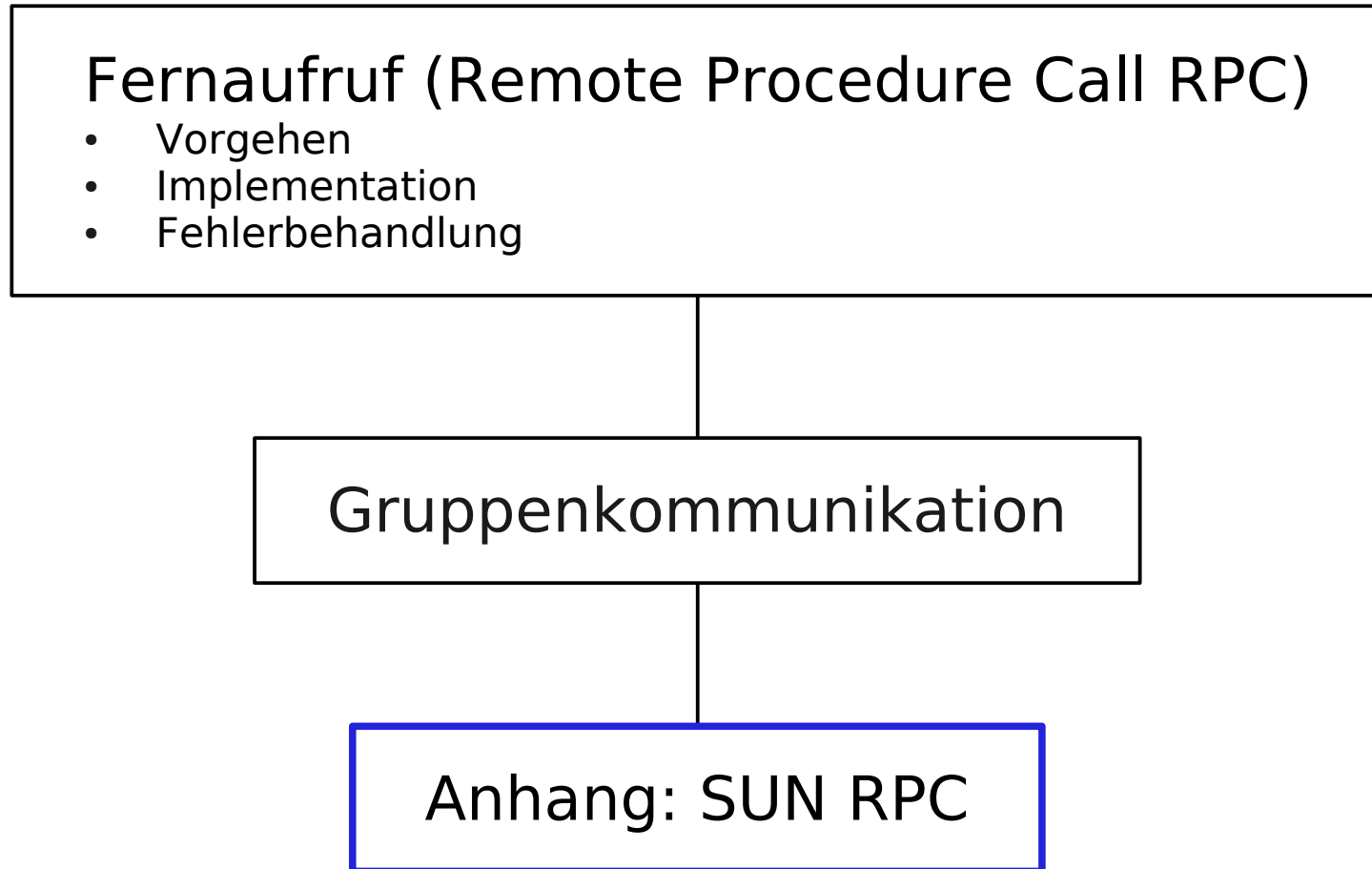
M2: Antwort auf Anfrage

- Kausal geordnete Zustellung („Causally Ordered Multicast“)
→ Aufrechterhaltung einer potentiellen kausalen
Abhängigkeit zwischen Ereignissen

Zusammenfassung

- IPC - Grundlage für offene Systeme
- Client-Server-Architekturen
- Wichtige Werkzeuge
 - ◊ Bibliotheken für Threads und
 - ◊ RPC
- Fehlersemantik und Anwendung

Wegweiser



Werkzeuge für RPC

→ Ziel:

- Botschaften-Systeme für Client-Server-Architekturen handhabbar machen

durch

- Simulation des lokalen Prozeduraufrufes durch Botschaften-System

Fall-Beispiel SUN-RPC

Interface Definition Language

- Programm-, Versions-Nummer und Prozedurnummern
- typedefs, constants, structs, programs
- rpcgen erzeugt
 - ◊ client: stub
 - ◊ server: main, dispatcher, stub
 - ◊ marshalling und unmarshalling
 - ◊ header file
- lokaler Binding-Service ("port mapper")

File interface in Sun XDR (1)

Coulouris
Dollimore
Kindberg

```
/* FileReadWrite service interface definition  
   in file FileReadWrite.x */
```

```
const MAX = 1000;
```

```
typedef int FileIdentifier;
```

```
typedef int FilePointer;
```

```
typedef int Length;
```

File interface in Sun XDR (2)

Coulouris
Dollimore
Kindberg

```
struct Data {
    Length length;
    char buffer[MAX];
};

struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};
```

```
program FILEREADWRITE

    version VERSION {

        void WRITE(writeargs) =1;
        Data READ(readargs) =2;

    } = 2; //version number
           //=> write_2
           //=> read_2

    } = 9999; //program number
```

C program for client in Sun RPC (1)

Coulouris
Dollimore
Kindberg

```
// File: C.c - Simple client of the
// FileReadWrite service.

#include <stdio.h>
#include <rpc/rpc.h>
#include <FileReadWrite.h>

int main(int argc, char **argv) {

    CLIENT *clientHandle;
    char *serverName = "coffee";
    readargs a;
    Data *data;

    clientHandle = clnt_create(serverName, FILEREADWRITE,
                              VERSION, "udp");
    //creates socket + a client handle
```

C program for client in Sun RPC (2)

Coulouris
Dollimore
Kindberg

```
if (clientHandle == NULL) {
    clnt_pcreateerror(serverName);
    // unable to contact server
    exit(1);
}

a.f = 10;
a.position = 100;
a.lenght = 1000;
data = read_2(&a, clientHandle);
// call to remote read procedure

...

clnt_destroy(clientHandle); //closes socket

return 0;
}
```

C program for server procedures in Sun RPC

Coulouris
Dollimore
Kindberg

```
// File S.c - server procedures for the
// FileReadWriteservice
#include <stdio.h>
#include <rpc/rpc.h>
#include <FileReadWrite.h>

void *write_2(writeargs *a) {

    // do the writing to the file
}

Data *read_2(readargs *a) {

    static Data result; //must be static
    result.buffer = ... //do the reading from the file
    result.length = ... //amount read from the file
    return &result;
}
```