Datenkommunikation

Transportzugriff

Wintersemester 2011/2012

Einordnung

1	Grundlagen von Rechnernetzen, Teil 1
2	Grundlagen von Rechnernetzen, Teil 2
3	Transportzugriff
4	Transportschicht, Grundlagen
5	Transportschicht, TCP (1)
6	Transportschicht, TCP (2) und UDP
7	Vermittlungsschicht, Grundlagen
8	Vermittlungsschicht, Internet
9	Vermittlungsschicht, Routing
10	Vermittlungsschicht, Steuerprotokolle und IPv6
11	Anwendungsschicht, Fallstudien
12	Mobile IP und TCP

Überblick

1. Konzepte

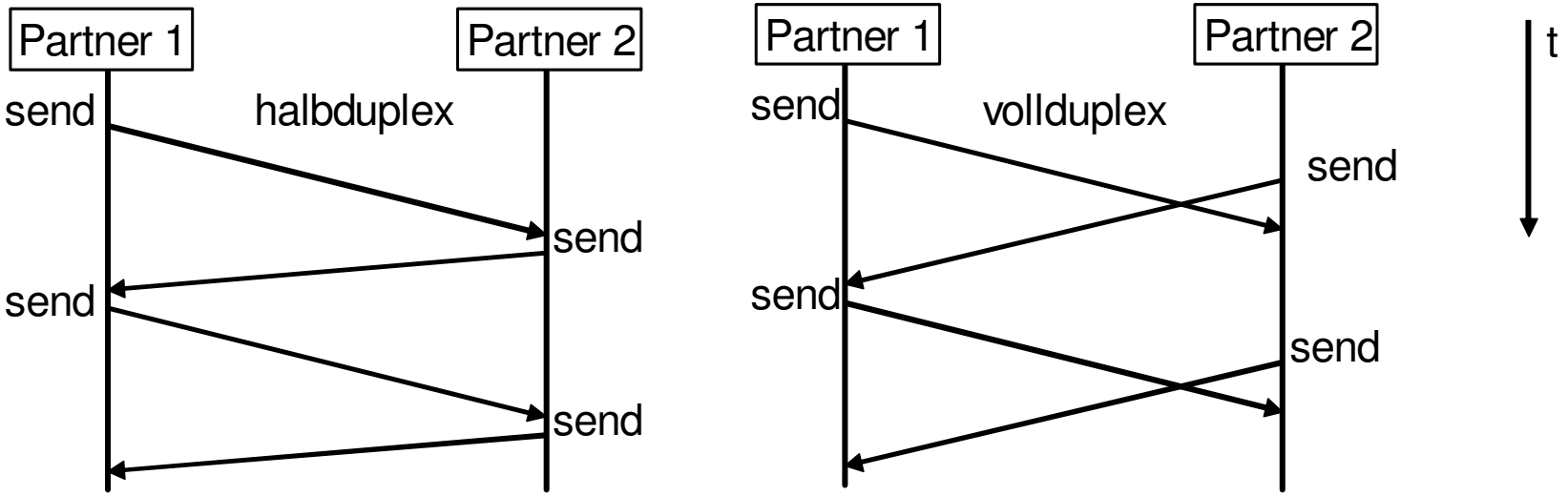
- Halbduplex, vollduplex, Empfängeradressierung
- Kommunikationsformen (Synchron vs. asynchron)
- Ablauf der Kommunikation
- Fehlersemantiken

2. Socket-Programmierung

- TCP-Sockets
- Datagram-Sockets

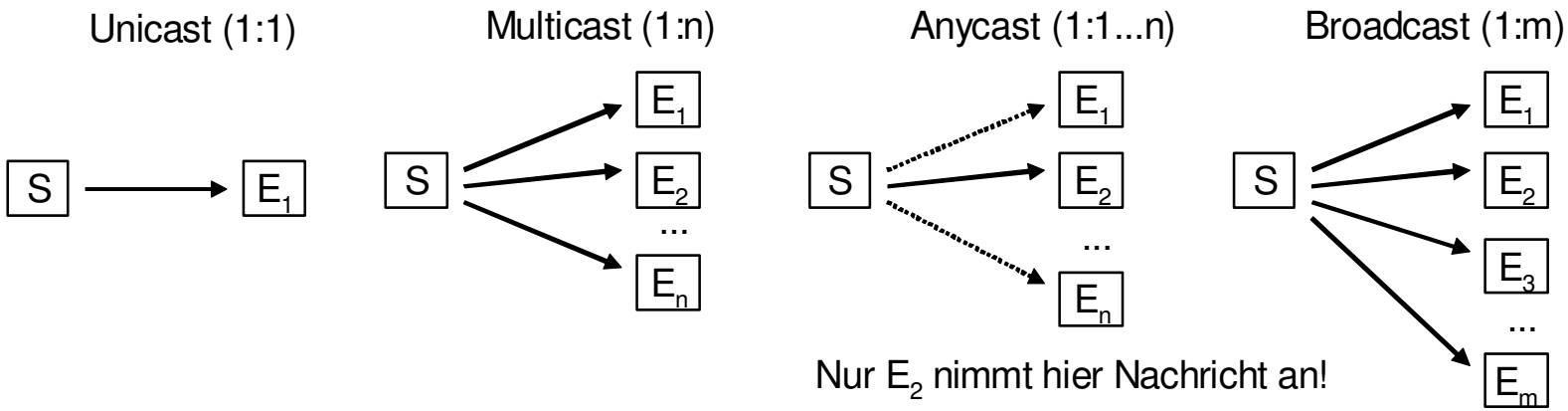
Halb- und vollduplex

- **Halbduplex**: Nur einer der Partner sendet zu einer Zeit
- **Vollduplex**: beide Partner können unabhängig voneinander senden



Empfängeradressierung

- Unicast: Nur ein Empfänger wird adressiert
- Alle anderen Varianten adressieren mehrere Empfänger



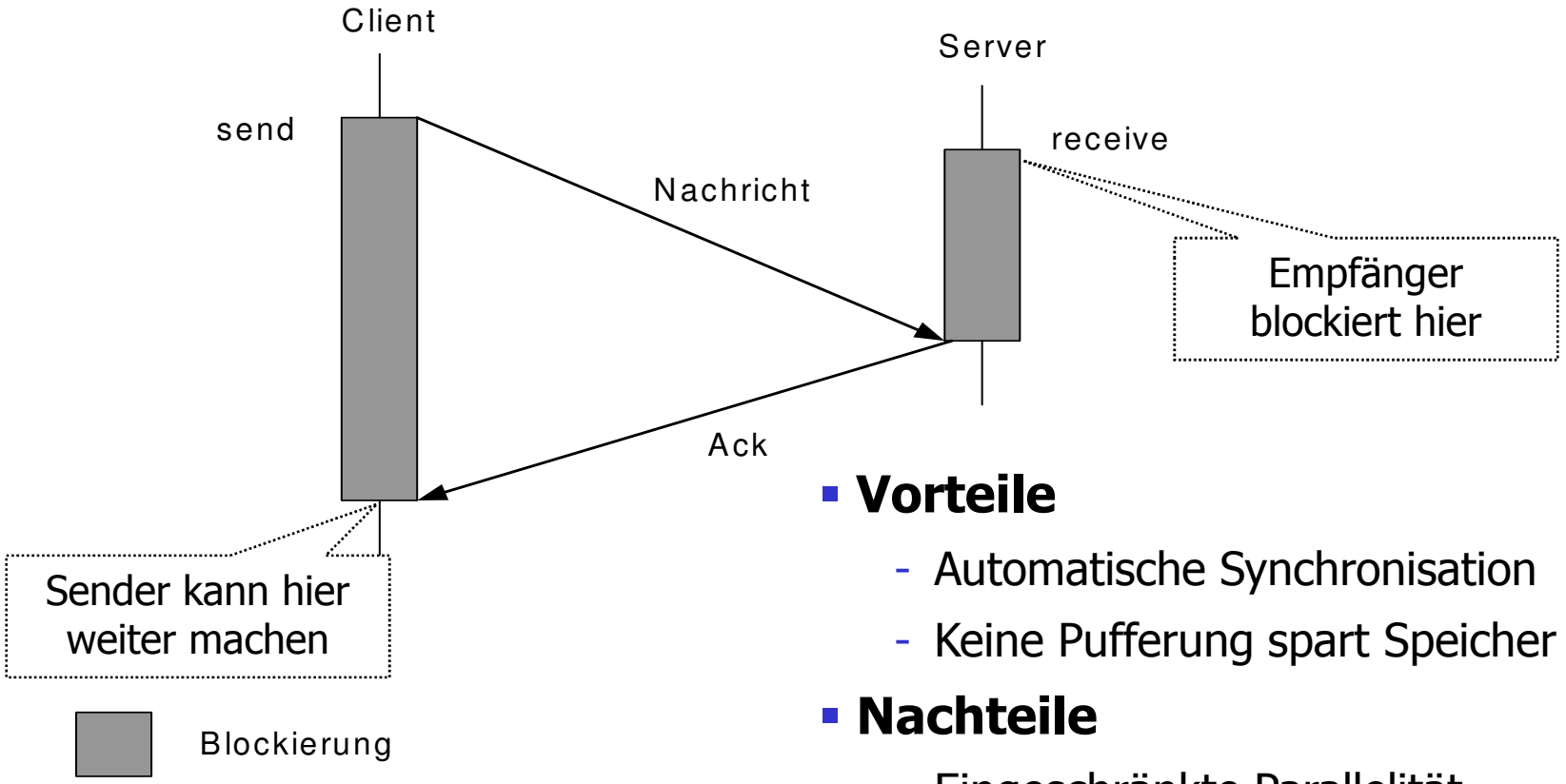
Blockierung

- Beim Nachrichtenaustausch unterscheidet man:
  - Synchrone Kommunikation
  - Asynchrone Kommunikation
- Synchron bedeutet:
  - **Blockierend**
  - Der Sender wartet, bis eine Methode send **mit einem Ergebnis** zurückkehrt
- Asynchron bedeutet:
  - **Nicht blockierend**
  - Der Sender kann weiter machen, wenn die Nachricht mit einer Methode send **in einen Transportpuffer** gelegt wurde

Pufferung von Nachrichten

- Puffer für ankommende Nachrichten werden **in den Protokollinstanzen** (meist im Betriebssystemkern) verwaltet
- Die Instanzen **kopieren** die Nachrichten in den Adressraum der empfangenden Anwendungsprozesse
- Pufferspeicher müssen **verwaltet** werden (→ Overhead)
- Pufferspeicher **benötigen Adressraum** (Speicher)
- Pufferspeicher sind **begrenzt** (→ evtl. Verwerfen von Nachrichten, wenn sie voll sind)

Blockierung - Synchrone Kommunikation

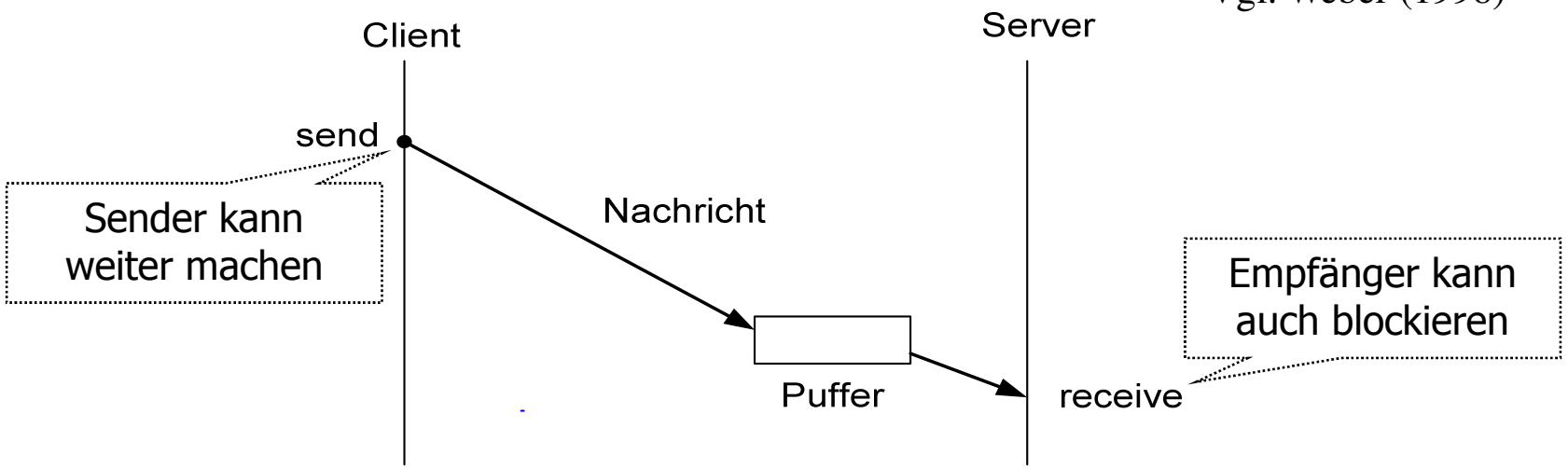


- **Vorteile**
  - Automatische Synchronisation
  - Keine Pufferung spart Speicher
- **Nachteile**
  - Eingeschränkte Parallelität
  - Evtl. langes Blockieren, wenn Empfänger keine Zeit hat

Vgl. Weber (1998)

Blockierung - Asynchrone Kommunikation

Vgl. Weber (1998)



- **Vorteile**
  - Zeitliche Entkopplung
  - Bessere Parallelarbeit möglich
  - Ereignisgesteuerte Kommunikation möglich
- **Nachteile**
  - Zwischenpufferung notwendig
  - Puffer voll führt trotzdem zum Blockieren wegen gesicherter Übertragung

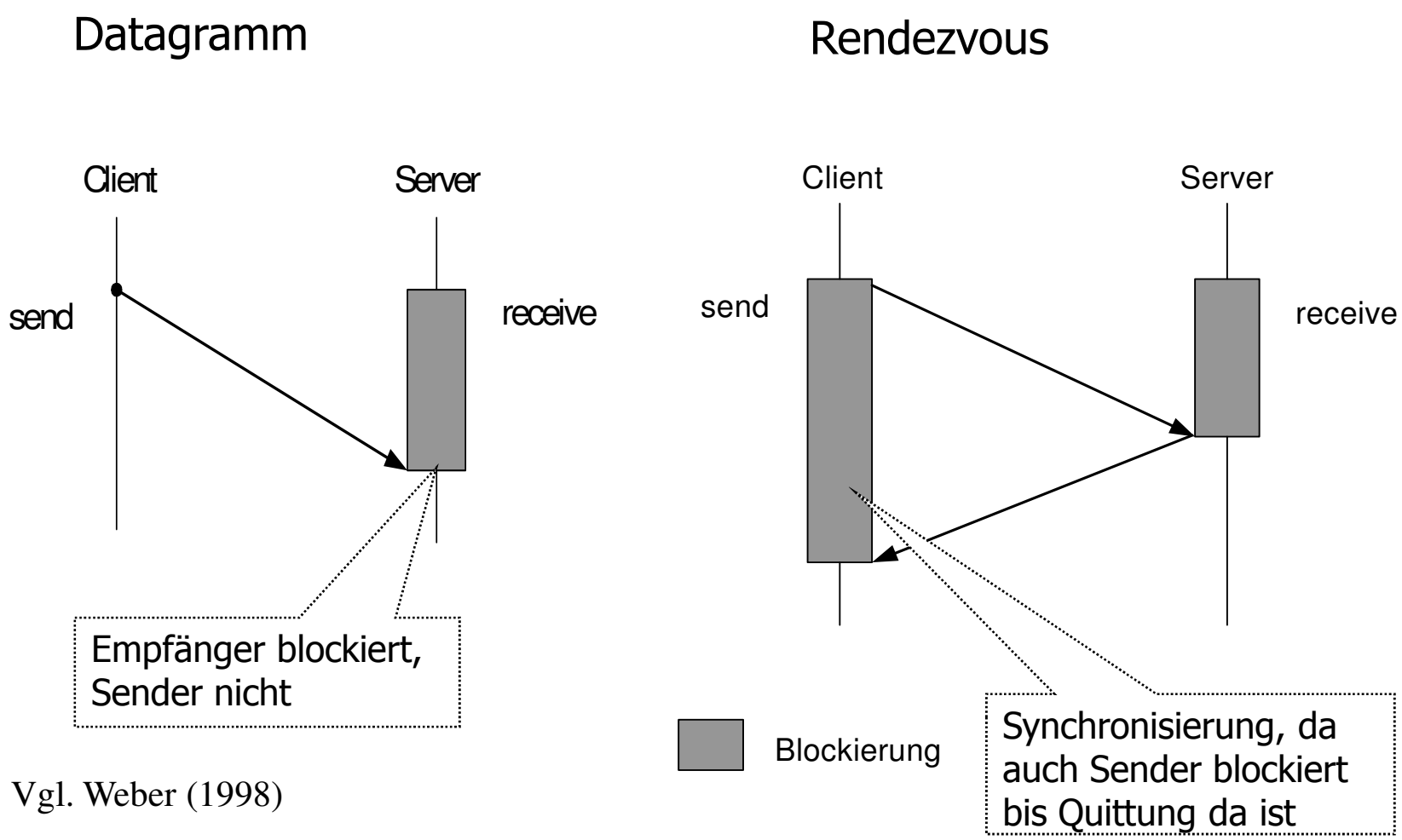
Kommunikationsformen

- Man unterscheidet weiterhin
  - **Meldungsorientierte** Kommunikation
    - Einwegnachrichten ohne Antwort
  - **Auftragsorientierte** Kommunikation
    - Request und Response (**mit Ergebnis**)
    - Entfernter Dienstaufruf

	asynchron	**synchron**
meldungsorientiert	Datagramm	Rendezvous
auftragsorientiert	asynchroner entfernter Dienstaufruf	synchroner entfernter Dienstaufruf

Vgl. Weber (1998)

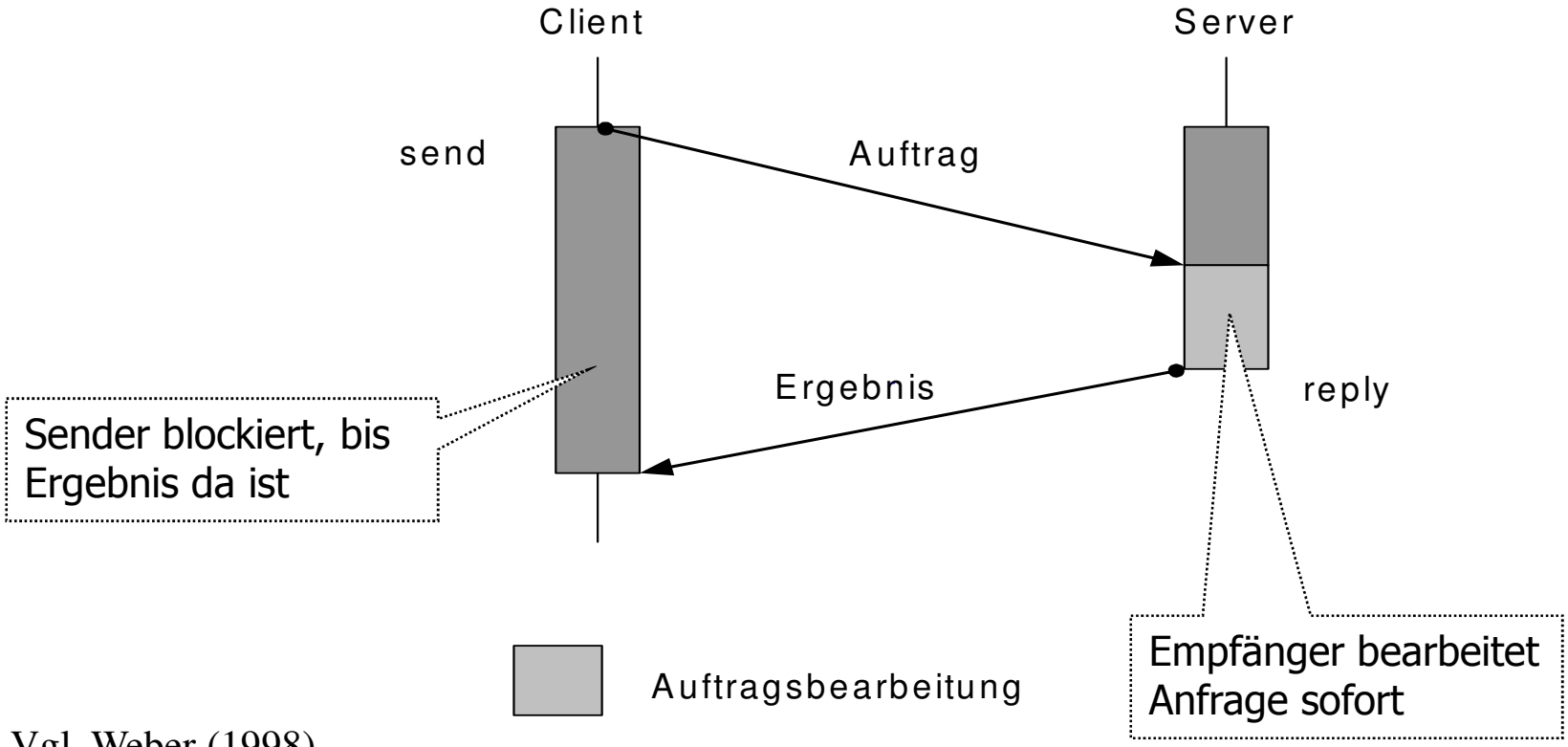
Meldungsorientiert Kommunikation



Vgl. Weber (1998)

Auftragsorientierte Kommunikation

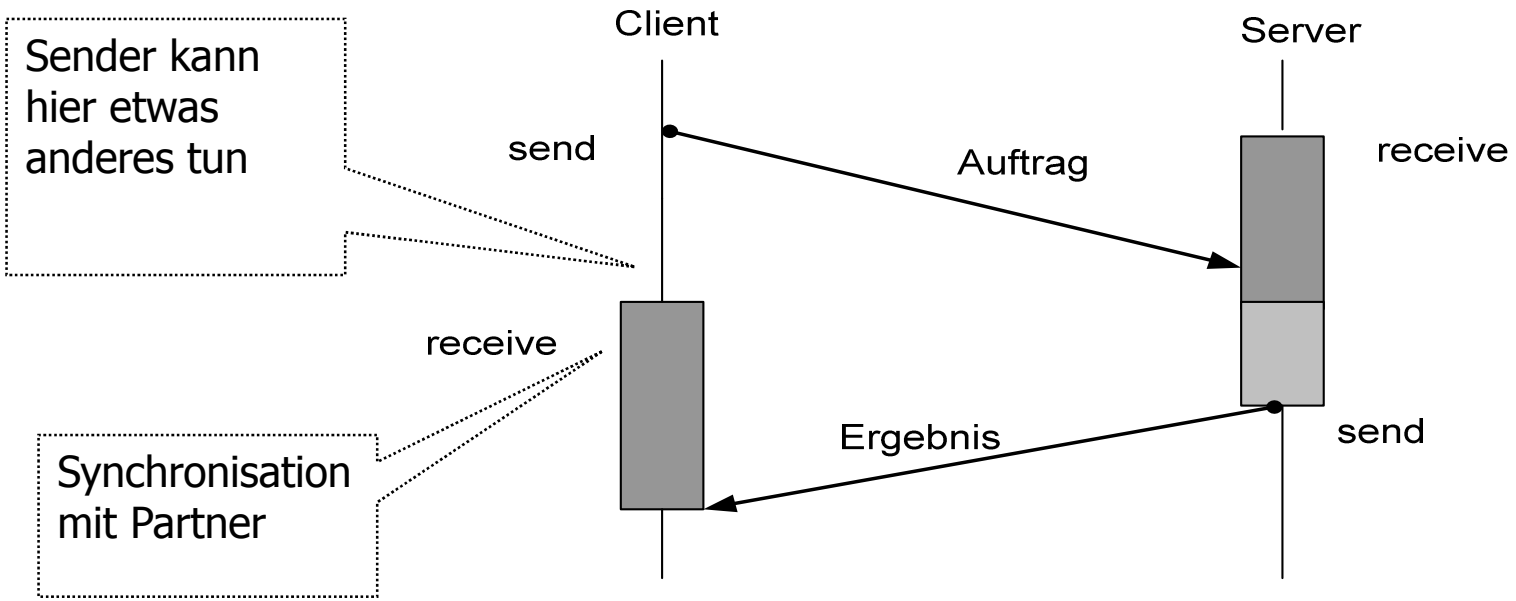
- Synchroner entfernter Dienstaufruf
  - Mit Rendezvous verwandt



Vgl. Weber (1998)

Auftragsorientierte Kommunikation

- Asynchroner entfernter Dienstaufruf



Vgl. Weber (1998)

Diskussion

- Wie kann man als Entwickler einer Kommunikations-anwendung **Threads** als Parallelisierungstechnik einsetzen?

Fehlersemantiken (1)

- Es gibt viele **Fehlerursachen**
  - Netzwerkfehler
  - Sender fällt vor Empfang des Ergebnisses aus (→ verwaiste Aufträge, Orphans)
  - Empfänger fällt während der Bearbeitung eines Requests aus
  - ...
- Ein Ausfall ist **zu jeder Zeit** möglich
- Das Kommunikationssystem kann sich hier je nach Realisierung unterschiedlich verhalten

  → **Verschiedene Fehlersemantiken** möglich

Fehlersemantiken (2)

- Lokal: Alles fällt komplett aus oder es läuft
- Verteilt: Verschiedene Ausfallsituationen zu betrachten

	Fehlerarten			
Fehlerklasse	**Fehlerfreier Ablauf**	**Nachrichten-verlust**	**Ausfall des Servers**	**Ausfall des Clients**
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0/1
At-Least-Once	Ausführung: 1 Ergebnis: 1	Ausführung: >= 1 Ergebnis: >= 1	Ausführung: >= 0 Ergebnis: >= 0	Ausführung: >= 0 Ergebnis: 0
At-Most-Once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0/1	Ausführung: 0/1 Ergebnis: 0
Exactly-Once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

Nach Schill (2007),

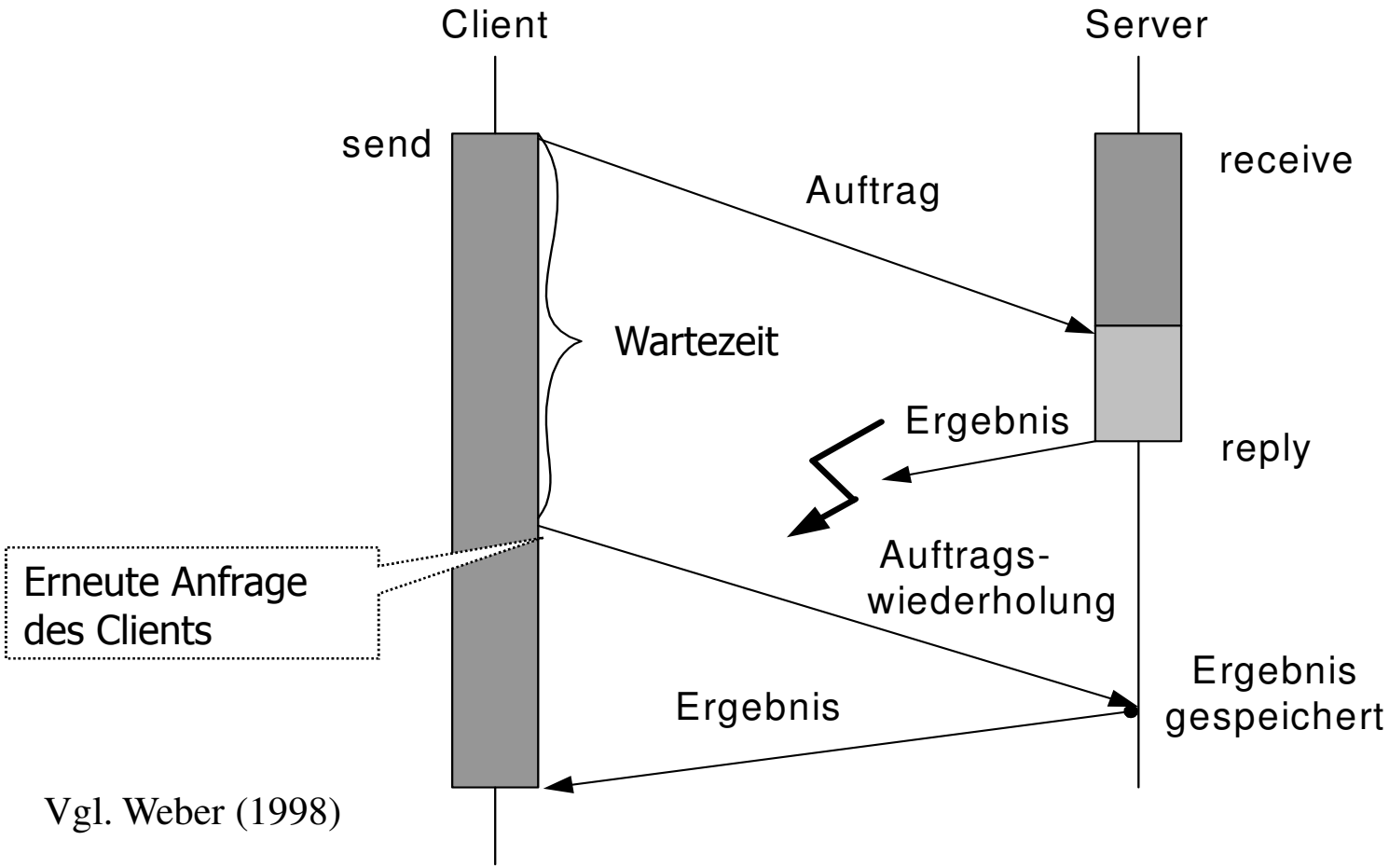
Wiederanlauf und Rücksetzmechanismen vorhanden → Transaktionen

Fehlersemantiken (3)

- Beispiel für eine Fehlersituation mit entsprechender Reaktion
  - Der Client erhält Antwort vom Server nicht und reagiert mit einem Timeout
  - Die Antwort des Servers ging verloren
  - Der Server speichert die Antwort
  - Der Client wiederholt den Auftrag nach dem Timeout
  - Der Server kann nun das gespeicherte Ergebnis senden
- Implementierungsaufwand!

Fehlersemantiken (4)

- Möglicher Ablauf im Beispiel

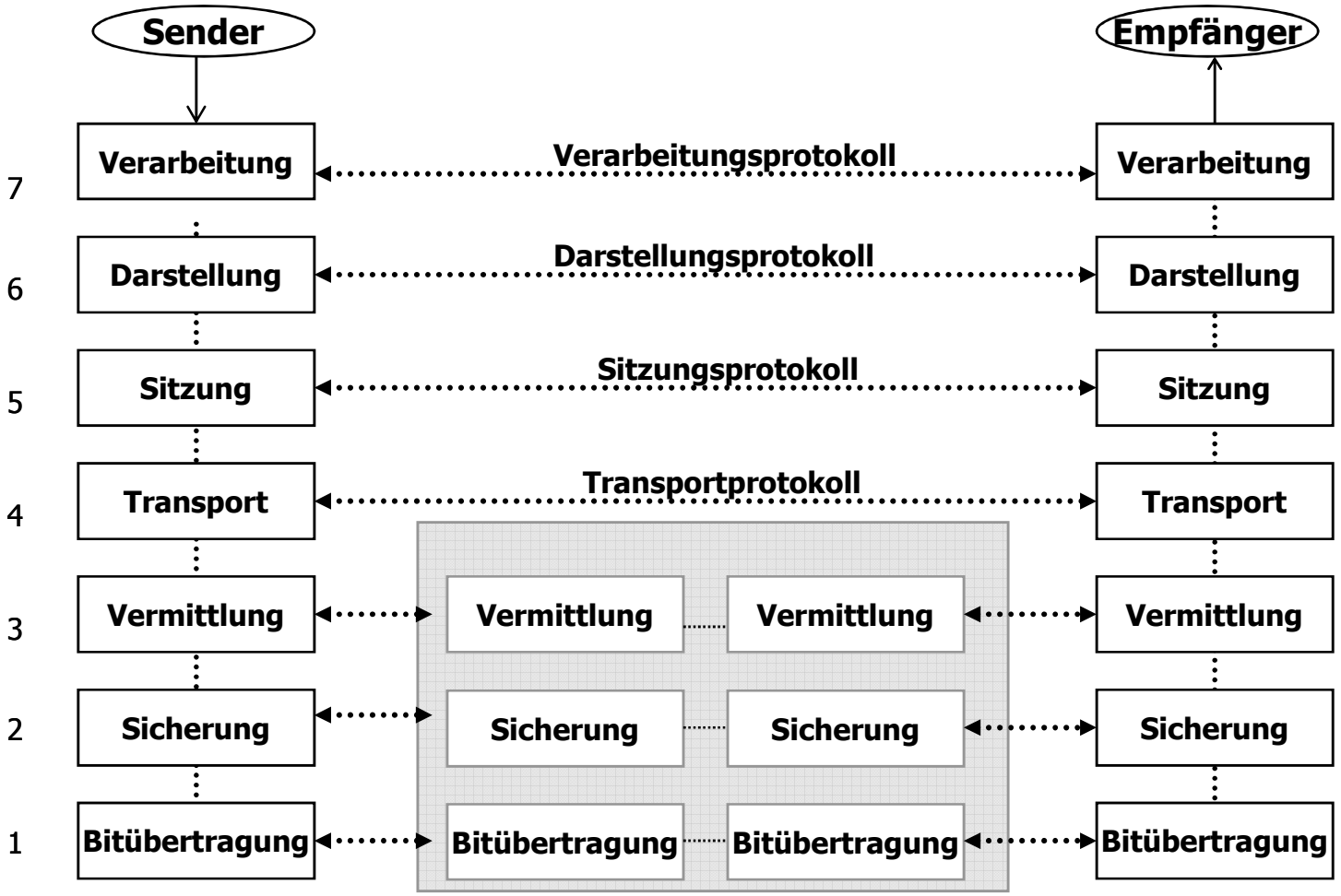


Vgl. Weber (1998)

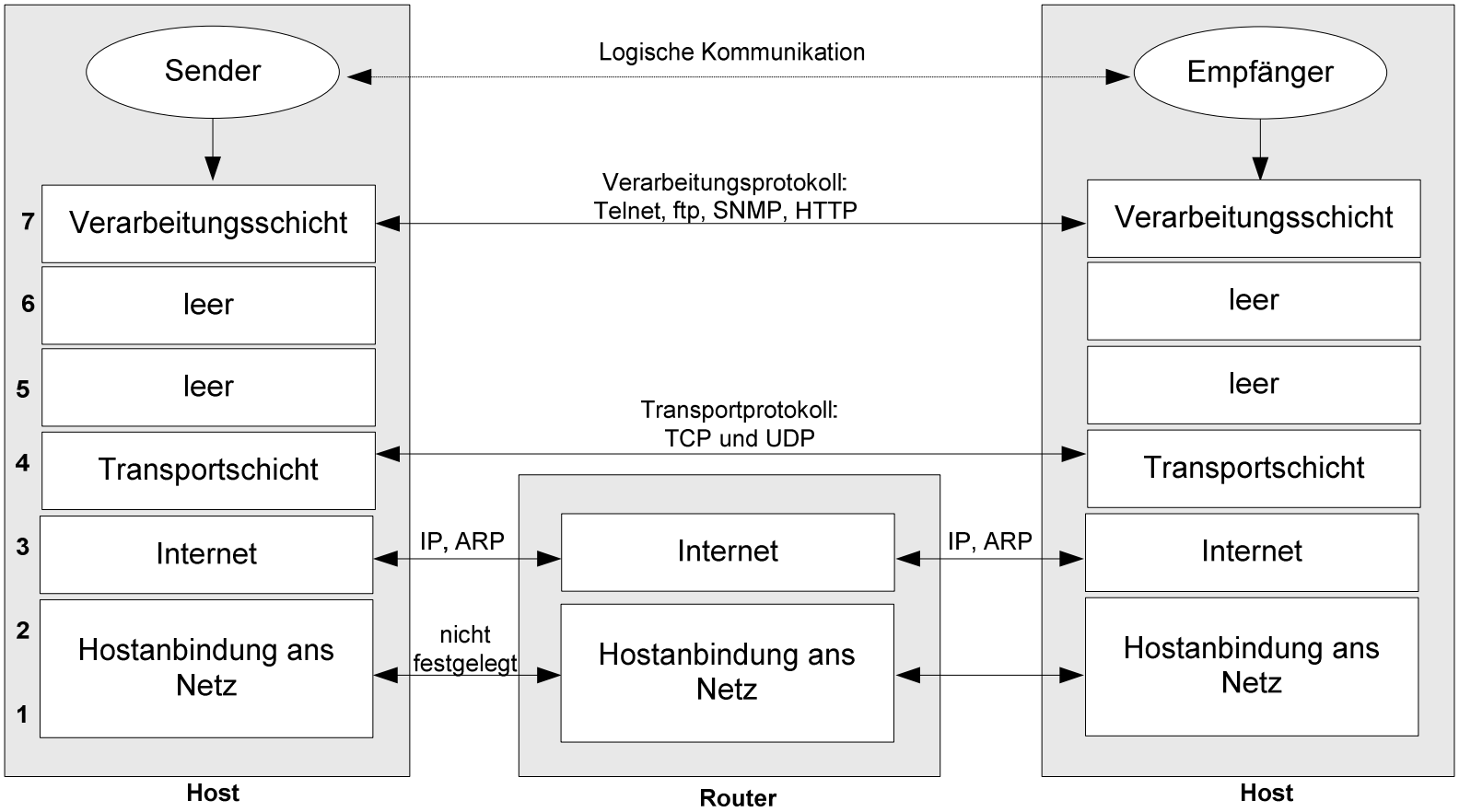
Überblick

1. Konzepte
   - Halbduplex, vollduplex, Empfängeradressierung
   - Kommunikationsformen
   - Ablauf der Kommunikation
   - Fehlersemantiken
2. **Socket-Programmierung**
   - Sockets: Programmiermodell, Systemeinbettung
   - TCP-Sockets
   - Datagram-Sockets

Wiederholung: ISO/OSI-Referenzmodell



Wiederholung: TCP/IP-Referenzmodell



Überblick über Sockets

- Die Socket-Schnittstelle ist eine API, mit der man Kommunikationsanwendungen entwickeln kann
- Die Socket-Schnittstelle ist eine **Transportzugriffs-schnittstelle**
- Sockets wurden in der Universität von Berkeley entwickelt (BSD-Version von UNIX), **erste Version 4.1cBSD-System** für die VAX aus dem Jahre 1982
- Die **Originalversion** der Socket-Schnittstelle stammt von Mitarbeitern der **Firma BBN** (ARPA-Projekt, 1981)
- Sockets sind heute der **De-facto-Standard**

Überblick über Sockets

- Die Socket API **unterstützt** vor allem **Client-Server-Anwendungen**, was aus dem Programmiermodell hervorgeht:
  - Aktiver Partner = Client
  - Passiver Partner = Server
- Sockets sind **Kommunikationsendpunkte** innerhalb der Applikationen, die in der Initialisierungsphase miteinander verbunden werden
- Dabei spielt es keine Rolle, auf welchen Rechnern die miteinander kommunizierenden Prozesse laufen

Protokollmechanismen

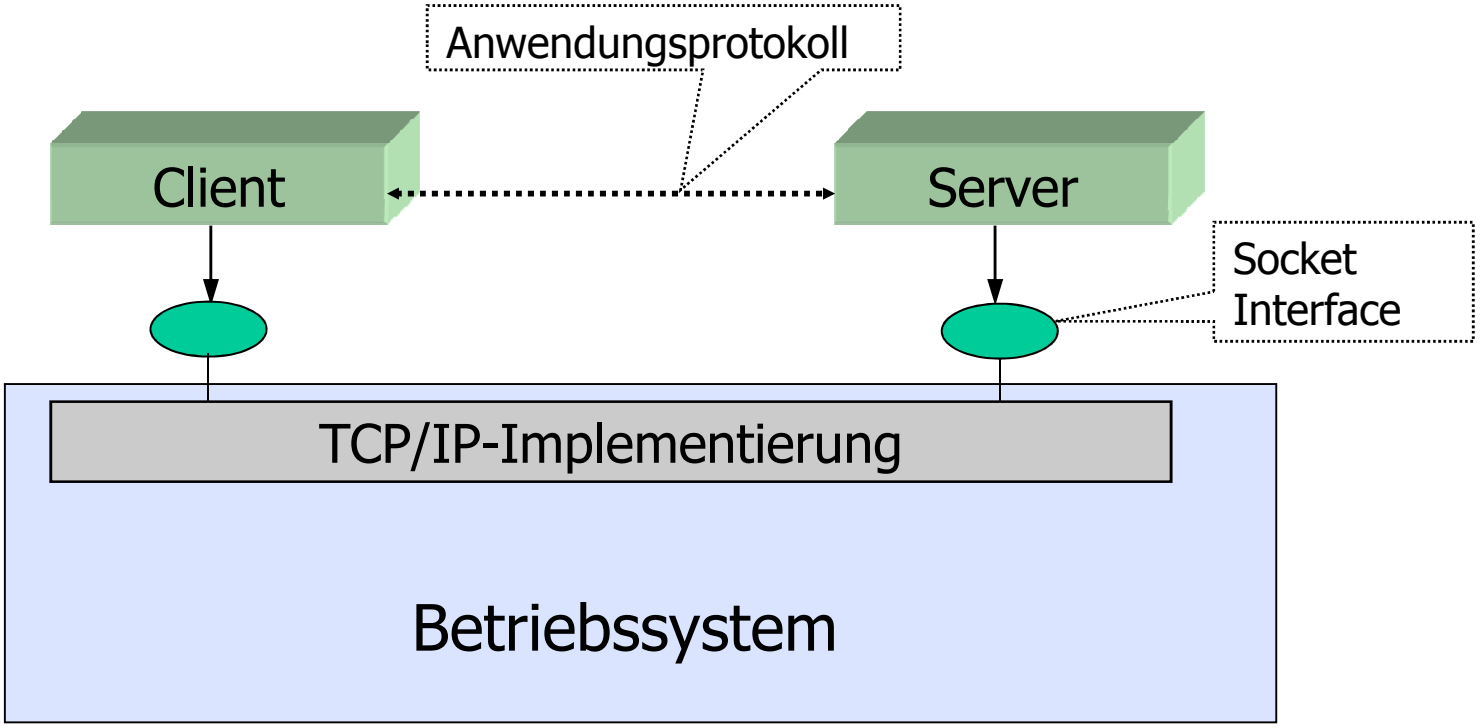
- Die TCP-Socket-Schnittstelle ist **stream-orientiert:** Es wird ein Datenstrom eingerichtet
- Die UDP-Socket-Schnittstelle ist nachrichten-orientiert
- Es ist **verbindungsorientierte** (TCP-basierte) und **verbindungslose** (UDP-basierte) Kommunikation möglich
- Die Adressierung der Kommunikationspartner erfolgt über die Kommunikationsendpunkte über das Tupel (IP-Adresse, Portnummer)

Überblick über Verbindungsaufbau

- Aufbau der Kommunikationsbeziehung bei verbindungsorientierter Kommunikation:
  - Serveranwendung wartet auf ankommende Verbindungsaufbauwünsche (listen) an einem TCP-Port
  - Clientanwendung erzeugt eine Connect.Request-PDU
  - Serveranwendung nimmt den Verbindungswunsch entgegen
  - Serveranwendung beantwortet Verbindungswunsch mit einer Connect.Response-PDU

Überblick über Systemeinbettung

- Systemeinbettung von Sockets



Phasen der Kommunikation über Sockets

- Die Verwendung von Sockets in einer Applikation erfolgt programmiertechnisch ist in **3 Phasen**:
  - Initialisierungsphase
    - Diese Phase ist für UDP symmetrisch, d.h. es gibt keine Unterscheidung zwischen Sender- und Empfänger-Sockets
    - Für TCP ist diese Phase asymmetrisch zwischen Client- und Server-Socket, d.h. ein Server-Socket wird anders initialisiert als ein Client-Socket
  - Lese- und Schreibphase
    - Sowohl für UDP als auch für TCP ist diese Phase symmetrisch, d.h. jeder Socket kann gleichermaßen senden und empfangen
  - Aufräumphase
    - Die benötigten Ressourcen werden freigegeben
    - Diese Phase ist sowohl für TCP als auch für UDP symmetrisch

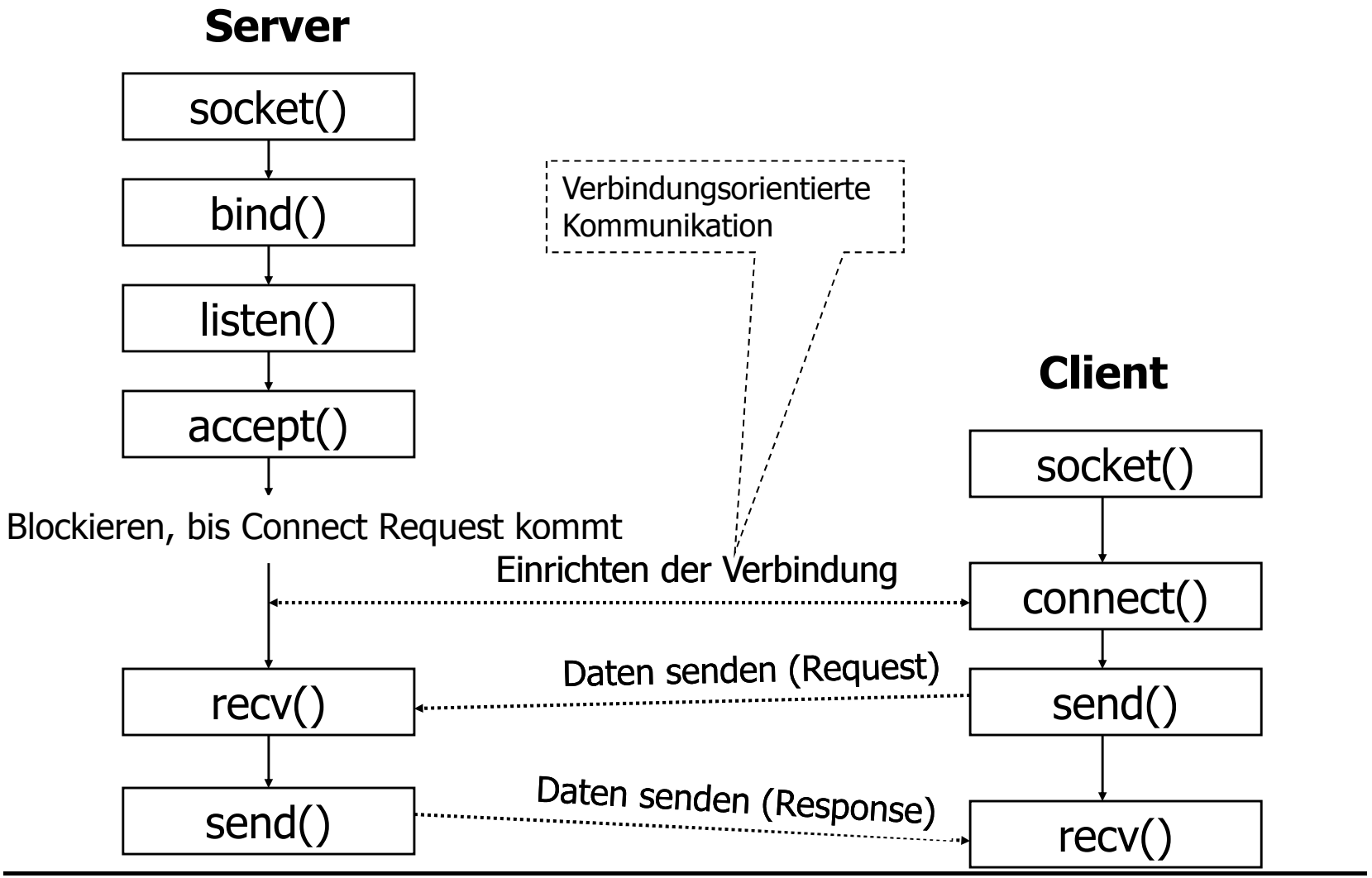
Socket-Programmierung in C

- C-Socket-Schnittstelle wird in nahezu jedem Betriebssystem in einer Funktionsbibliothek bereitgestellt
- Mit der Funktionsbibliothek lässt sich „relativ“ aufwändig programmieren
- Die meisten Socket-basierten Kommunikations-anwendungen sind heute in C programmiert
- Im Folgenden sind einige Ausschnitte aus einfachen Beispielanwendungen skizziert

Die wichtigsten C-Funktionen

- **socket()** - Initialisiert einen Socket
- **bind()** - Ordnet einem Socket eine lokale Adresse zu
- **connect()** - Baut eine Verbindung vom Client zum Server auf
- **listen()** - Setzt den Socket in einen passiven, d.h. auf ankommenden Verbindungswünsche wartenden Zustand
- **accept()** - Wird bei TCP-Verbindungen verwendet und gibt die nächste ankommende, aufgebaute Verbindung aus der Warteschlange zurück
- **close()** - Schließt eine Verbindung
- **recv()** - Liest Daten aus dem spezifizierten Socket und gibt die Anzahl der tatsächlich gelesenen Byte zurück
- **send()** - Sendet Daten über den spezifizierten Socket und gibt die Anzahl der tatsächlich gesendeten Byte zurück

Nutzung der C-Funktionen



C-Programmausschnitt für TCP-Sockets, Server

```
#include "inet.h"
#define SERV_TCP_PORT 5999
char *pname;...
main(int argc, char argv[])
{
   int sockfd, newsockfd, clilen, childpid;
   struct sockaddr_in cli_addr, serv_addr;
   pname = argv[0];

   // Socket öffnen und bind
   if ((sockfd =
      socket(AF_INET, SOCK_STREAM, 0) ) < 0)
         // Fehlermeldung ausgeben...
   // Lokale Adresse binden, vorher Socket sockaddr
      belegen mit IP-Adresse und Port
   bzero((char *) &serv_addr, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr =
      htonl(INADDR_ANY);
   serv_addr.sin_port = htons(SERV_TCP_PORT);
```

Protokollfamilie

Sockettyp

```
if (bind(sockfd, (struct sockaddr *)&serv_addr,
      sizeof(serv_addr) < 0)
      // Fehlermeldung ausgeben

listen(sockfd, 5);
for (;;) {
      // Auf Verbindungsaufbauwunsch warten
      clilen = sizeof(cli_addr);
      newsockfd = accept(sockfd,
        (struct sockaddr *) &cli_addr,  &clilen);
      if ((childpid = fork()) < 0)
        // Fehlermeldung ausgeben
      else if (childpid == 0) { // Sohnprozess
        close(sockfd);
        // Client-Request bearbeiten
        exit(0);
      }
      close (newsockfd); // Elternprozess
   }
}
```

Länge der Anfragequeue

Sohnprozess terminieren

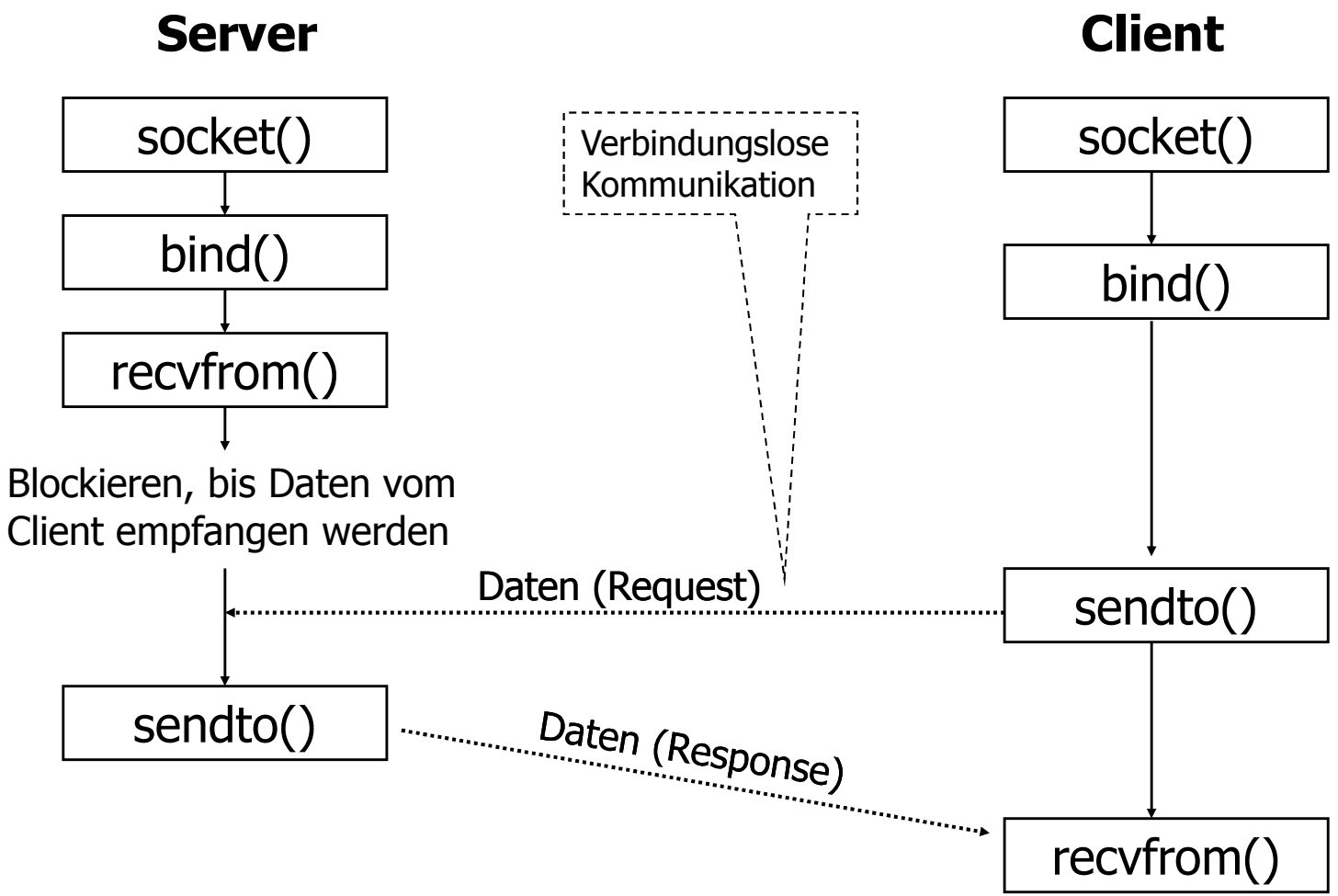
Vgl. Stevens

C-Programmausschnitt für TCP-Sockets, Client

```
#include ...
#include "inet.h"
#define SERV_TCP_PORT 5999

// Serveradresse
#define SERV_HOST_ADDR "192.43.235.6"

char *pname;

main(int argc, char argv[])
{
      int sockfd;
      struct sockaddr_in serv_addr;
      pname = argv[0];

      // Serveradresse belegen...
      bzero((char *) &serv_addr, sizeof(serv_addr));
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr =
      inet_addr(SERV_HOST_ADDR);
      serv_addr.sin_port = htons(SERV_TCP_PORT);
```

```
      // Socket öffnen
      if ((sockfd =socket(AF_INET,
        SOCK_STREAM, 0) < 0)
        // Fehlermeldung ausgeben

      // Verbindung zum Server aufbauen

      if (connect(sockfd; (struct sockaddr *)
      &serv_addr,  sizeof(serv_addr)) < 0)
         // Fehlermeldung ausgeben
         // Verarbeitung: Request senden ...

      close(sockfd);
      exit(0);
}
```

Systemcalls bei UDP-Sockets



C-Programmausschnitt für UDP-Sockets, Server (1)

```
#include „inet.h“;
#define SERV_UDP_PORT 5999
 ...
main(int argc, char argv[])
{
      int sockfd;
      struct sockaddr_in serv_addr, cli_addr;
      pname = argv[0];

      // UDP-Socket oeffnen
      if (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
        // Error handling
      }
      bzero((char *) &serv_addr, sizeof(serv_addr));
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
      serv_addr.sin_port = htons(SERV_UDP_PORT);

      // Binde lokale Adresse
      if (bind(socketfd, &serv_addr,...) ...
      dg_echo(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
}
```

C-Programmausschnitt für UDP-Sockets, Server (2)

```
#include <sys/types.h>
#include <sys/socket.h>
#define MAXMESG 2048

 /* Jede Nachricht wird zum Client zurueckgesendet, echo */

dg_echo(int sockfd, sockaddr pcli_addr, int maxclilen)
{
      int n, clilen;
      char mesg[MAXMESG];

      for (;;) {
        clilen = maxclilen;
        n = recvfrom(sockfd, mesg, MAXMESG, 0, pcli_addr, &clilen);
        if (n < 0) {
          /* Error handling */
        }
        if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n) {
          /* Error handling */
        }
      }
}
```

C-Beispielprogramm für UDP-Sockets, Client (1)

```
#include „inet.h"; ...
main(int argc, char argv[])
{
      int sockfd;
      struct sockaddr_in serv_addr, cli_addr;
      pname = argv[0];
      bzero((char *) &serv_addr, sizeof(serv_addr());
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
      serv_addr.sin_port = htons(SERV_UDP_PORT);

      // UDP-Socket oeffnen und lokale Adresse binden
      if (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
        /* Error handling */ }
      if (bind(socketfd, (struct sockaddr *) &cli_addr,...) ...
      dg_cli(stdin, sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        /* Error handling */ }
      close(sockfd); exit(0);
}
```

vgl. Stevens

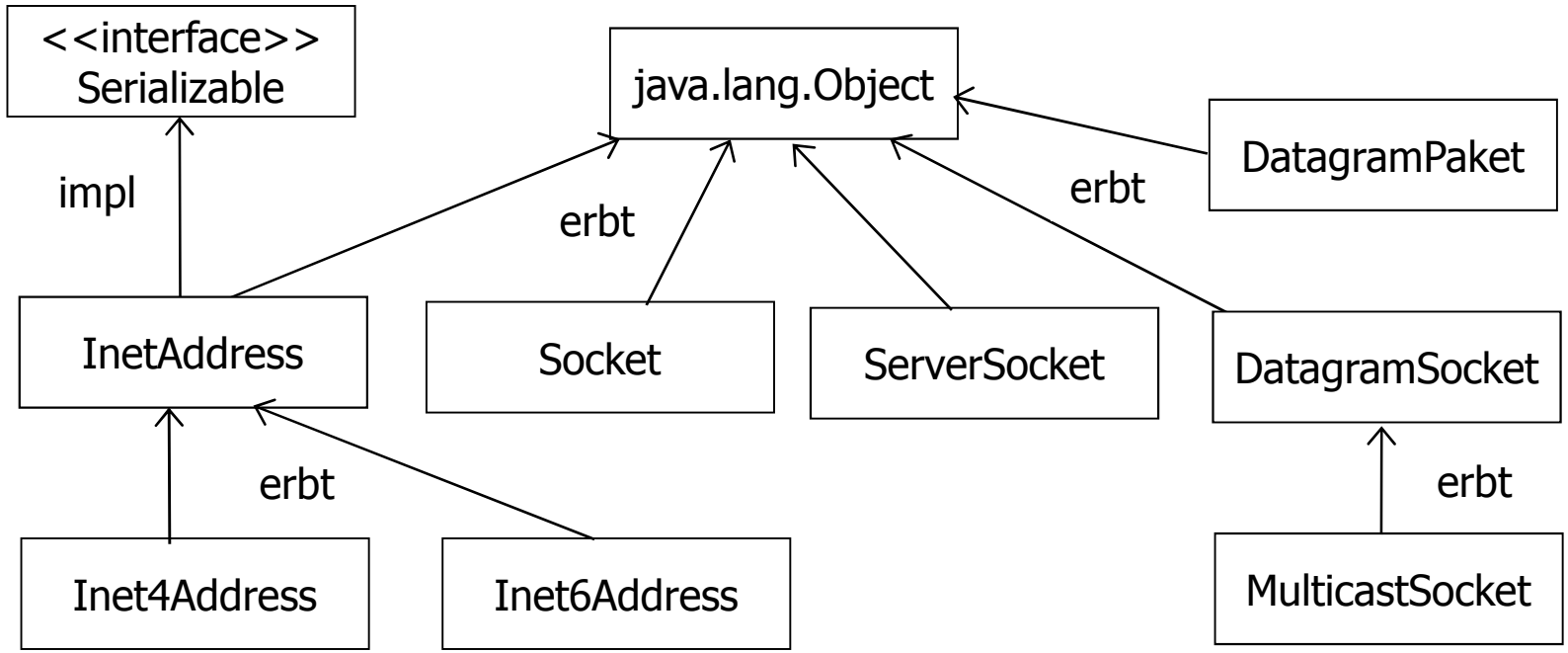
C-Beispielprogramm für UDP-Sockets, Client (2)

```
#include <stdio.h>
#include <sys/socket.h>
#define MAXLINE 512

 /* Nachricht ueber stdin einlesen, zum Server senden, wieder empfangen und auf
    stdout ausgeben */
dg_cli(FILE fd, int sockfd, struct sockaddr *pser_addr, int servlen)
{
      int n;
      char sendline[MAXMESG]; recvline[MAXLINE+1];

      /* Nachricht von stdin einlesen */

      if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n) {
        /* Error handling */ }

      n = recvfrom(sockfd, recvlen, MAXLINE, 0, (struct sockaddr *) 0, (int *) 0);
      if (n < 0) {
         /* Error handling */ }

      /* Nachricht auf stdout ausgeben */ }
}
```

Wichtige Java-Klassen für TCP-und UDP-Sockets

- In der Java-API ist das Package **java.net** für TCP-Sockets vorgesehen. Wichtige Klassen sind:
  - InetAddress
  - Socket (Client)
  - ServerSocket
- Das Package java.net enthält auch Klassen zur Bearbeitung von UDP-Sockets. Wichtige Klassen sind:
  - DatagramSocket
  - DatagramPaket

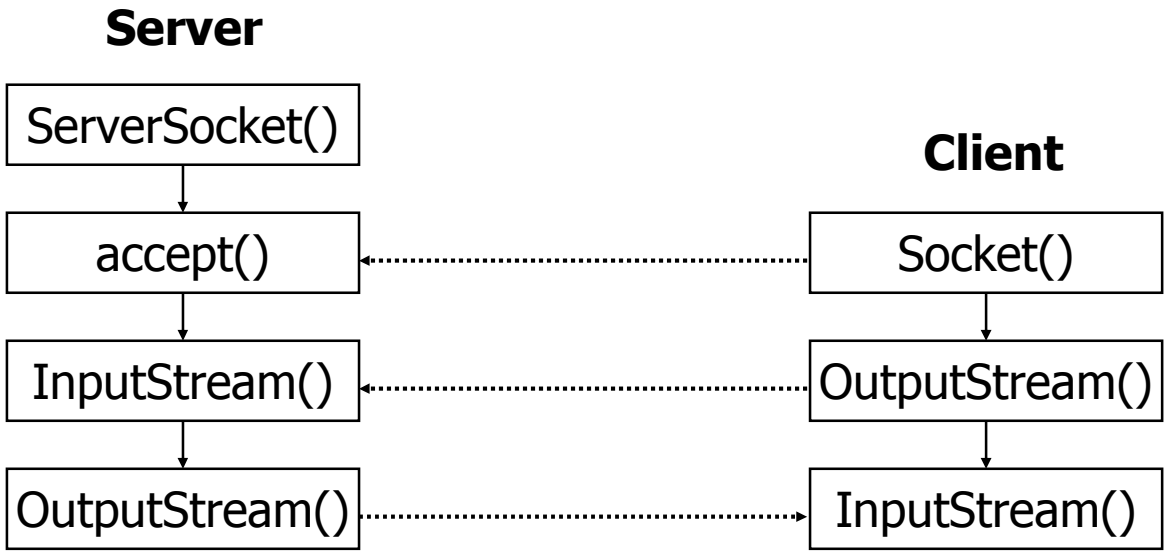
Package java.net, wichtige Klassen



Vgl. Java™ 2 Platform Standard Edition 1.4x, 1.5x, 1.6

Java-TCP-Sockets

- **java.net** stellt eine höherwertige Schnittstelle für Sockets zur Verfügung
  - objektorientierte Schnittstelle
  - Implementierungsdetails gut gekapselt
- Programmierung wird durch Input- und Output-Streams, die den Sockets zugeordnet werden, vereinfacht

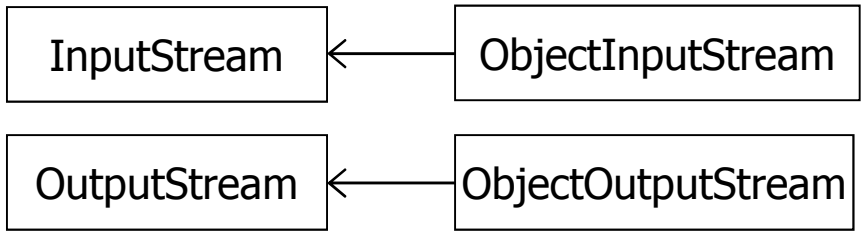


UDP-Sockets allgemein

- Paket-basierte Kommunikation im Gegensatz zu Streams-basiert
- Keine Empfangsgarantie
- Keine Ordnung der Pakete
- Duplikate sind möglich
- Vorteile:
  - Separate Nachrichten können gesendet werden
  - Ein Fehler in einer Nachricht wirkt sich nicht auf die folgenden Nachrichten aus
  - Es werden immer nur ganze Nachrichten empfangen

Weitere wichtige Klassen zur Socket-Programmierung

- Bei TCP-Sockets werden zum Lesen und Schreiben mit Objektströmen verwendet:
  - ObjectInputStream
  - ObjectOutputStream



Package java.net, SocketExceptions

- java.lang.Object
  - java.lang.Throwable
    - java.lang.Exception
      - java.io.IOException
        - java.net.ProtocolException
        - java.net.UnknownHostException
        - java.net.UnknownServiceException
        - java.net.SocketException
          - java.net.ConnectException
          - java.net.BindException
          - java.net.NoRouteToHostException
          - ...

Beispielprogramm mit Java TCP-Sockets

- Der folgende Programmrahmen zeigt einen Server, der Client-Requests nur sequentiell abarbeiten kann
- Der Server übernimmt die Verbindung und arbeitet den Request ab
- Erst nach der Bearbeitung kann er wieder neue Requests empfangen
- Alternativ: Für die Bearbeitung nebenläufiger Requests kann man Java-Threads nutzen
  - Verbindungen werden über eigene Threads angenommen und bearbeitet

Beispielprogramm - Programmausschnitt

socket(), bind() und listen() im Konstruktor

socket() und bind() im Konstruktor

Server

```
...
ServerSocket server = new
    ServerSocket(Portnummer) {
while (true) {
    Socket verbindung =
      server.accept();
    InputStream in =
      verbindung.getInputStream();
    OutputStream out =
      verbindung.getOutputStream();

    // Empfangen über Inputstream
    // Senden über OutputStream
    // ...

    // Stream und Verbindung schließen
    // Schleife beenden
}
...
```

Client

```
...
Socket client = new Socket (Host,
    Portnummer);
InputStream in =
    client.getInputStream();
OutputStream out =
    client.getOutputStream();

// Senden über OutputStream
// Empfangen über Inputstream
...

// Stream und Verbindung schließen

...
```

UDP-Sockets in Java (Datagram-Sockets)

- Sockets vom Typ DatagramSocket für Datagramme
  - analog TCP kann dem Konstruktor eine Portnr. > 5000 vorgegeben werden
  - ohne spezifizierte Portnummer → freie Portnummer im Bereich 1024 – 5000
- Methoden der Klasse DatagramSocket für die nachrichtenorientierte Kommunikation
  - void **send**(DatagramPacket p) zum Senden und
  - void **receive**(DatagramPacket p) zum Empfangen von Datagrammen
  - receive blockiert den Aufrufer bis ein Datagramm eingeht

UDP-Sockets in Java

- Datagramme vom Typ DatagramPacket können instanziiert werden:
  - zum Empfangen, gibt man dem Konstruktor ein Byte-Array mit, in das die zu empfangenden Daten eingetragen werden sollen.
  - zum Versenden, gibt man dem Konstruktor neben den Daten (Byte-Array) die IP-Adresse und den Port des Empfängers mit.
- DatagramPacket bietet ferner Methoden zum Lesen und Schreiben der Daten, der IP-Adresse und der Portnummer des Senders und des Empfängers
- → Vgl. Java™ 2 Platform Standard Edition für genaue Information zu den Methoden

Ablauf einer Kommunikation über einen UDP-Socket

- DatagramSocket erzeugen
- Zum Senden eines Datagramms:
  - Objekt vom Typ DatagramPacket erzeugen;
    - dabei die zu sendenden Daten und die Adresse des Empfängers angeben.
  - Das Datagramm mit der Methode send am DatagramSocket verschicken

Ablauf einer Kommunikation über einen UDP-Socket

- Zum Empfangen eines Datagramms:
  - Objekt vom Typ DatagramPacket erzeugen
  - Byte-Array angeben, das Platz für die zu empfangenden Daten bereitstellt
  - Daten mit receive am DatagramSocket empfangen
  - Die eigentlichen Daten und die Senderadresse mit Hilfe der Methoden getData bzw. getAddress am DatagramPacket auslesen
- DatagramSocket mit close schließen

Beispiel EchoServer (1)

```
package UDPEchoExample;
import java.net.*;
import java.io.*;

public class UDPEchoServer {
     protected DatagramSocket socket;
     public UDPEchoServer (int port) throws IOException
     {
       socket = new DatagramSocket (port);
     }
     public void execute () throws IOException
     {
        while (true) {
                DatagramPacket packet = receive();
                sendEcho (packet.getAddress (), packet.getPort (),
                                packet.getData (), packet.getLength ());
       }
}
```

Beispiel EchoServer (2)

```
protected DatagramPacket receive () throws IOException {
    byte buffer[] = new byte[65535];
    DatagramPacket packet = new DatagramPacket (buffer, buffer.length);
    socket.receive (packet);
    System.out.println ("Received " + packet.getLength () + " bytes.");
    return packet;
  }
  protected void sendEcho (InetAddress address, int port, byte data[], int
      length) throws IOException {
    DatagramPacket packet = new DatagramPacket (data, length, address, port);
    socket.send (packet);
    System.out.println („Response sent");
  }
  public static void main (String args[]) throws IOException {
    if (args.length != 1)
      throw new RuntimeException ("Syntax: UDPEchoServer <port>");
    UDPEchoServer echo = new UDPEchoServer (Integer.parseInt (args[0]));
    echo.execute ();
  }
}
```

Beispiel EchoClient (1)

```
package UDPEchoExample;
import java.net.*;
import java.io.*;
public class UDPEchoClient {
  protected DatagramSocket socket;
  protected DatagramPacket packet;
 public UDPEchoClient (String message, String host, int port) throws
     IOException {
    socket = new DatagramSocket ();
    packet = buildPacket (message, host, port);
    try {
      sendPacket ();
      receivePacket ();
    } finally {
      socket.close ();
    }
  }
 protected void sendPacket () throws IOException {
   socket.send (packet); ...
 }
```

Beispiel EchoClient (2)

```
  protected void receivePacket () throws IOException {
    byte buffer[] = new byte[65535];
    DatagramPacket packet = new DatagramPacket (buffer, buffer.length);
    socket.receive (packet);
    ByteArrayInputStream byteI = new ByteArrayInputStream (packet.getData (),
     0, packet.getLength ());
    DataInputStream dataI = new DataInputStream (byteI);
    String result = dataI.readUTF ();
  }
  public static void main (String args[]) throws IOException {
    if (args.length != 3)
      throw new RuntimeException („EchoClient <host> <port> <message>");
    while (true) {
      new UDPEchoClient (args[2], args[0], Integer.parseInt (args[1]));
      try {
        Thread.sleep (1000);
      } catch (InterruptedException ex) {...}
    }
  }
}
```

Package java.net, MulticastSockets

- Die Klasse MulticastSockets dient zum Senden und Empfangen von IP-Multicast-Datagrammen
- Ein MulticastSocket ist ein UDP-DatagramSocket zur IP-basierten Gruppenkommunikation
  - Siehe IP-Adressen 224.* - 239.*
  - Methode **joinGroup(InetAddress groupAddr)** zum Anbinden an eine Gruppe
  - Methode **leaveGroup(InetAddress mcastaddr)** zum Verlassen der Gruppe
- Ein gesendetes Datagramm empfangen alle Gruppenmitglieder

Zusammenfassung: Sockets (1)

- Wie andere Programmiersprachen bietet Java Sockets für die TCP/IP- bzw. UDP/IP-Kommunikation an
- **Socketadressen** bestehen aus zwei Komponenten, der IP-Adresse bzw. dem logischen DNS-Namen des betreffenden Rechners und einer Portnummer
- Für TCP-Verbindungen gibt es **Server-Sockets und Client-Sockets**:
  - Wenn sich ein Client-Socket an einen Server-Socket anbindet, wird auf Server-Seite ein neuer Socket erzeugt, der den Endpunkt dieser Verbindung darstellt
  - Das eigentliche Senden und Empfangen von Daten geschieht über Schreib- und Leseströme, die man mit einem Socket assoziieren kann

Zusammenfassung: Sockets (2)

- Datagram-Sockets = Endpunkte zum Senden und Empfangen von Datagrammen
  - UDP ist verbindungslos
  - UDP ist nicht zuverlässig
  - UDP garantiert keine Reihenfolgetreue
- Das Anwendungsprogramm muss diese Dinge regeln
- Bei Nutzung von UDP-Sockets sollte man berücksichtigen:
  - 16 Bits für Paketlänge, also Datenlänge begrenzen, kein Stream
  - Vermeidung von Fragmentierung wichtig, wenn ein Fragment (IP) verloren geht, wird die ganze Nachricht verworfen

Überblick

1. Konzepte
   - Halbduplex, vollduplex, Empfängeradressierung
   - Kommunikationsformen
   - Ablauf der Kommunikation
2. Socket-Programmierung
   - Sockets: Programmiermodell, Systemeinbettung
   - TCP-Sockets
   - Datagram-Sockets