
Rechnernetze II

SoSe 2020

Roland Wismüller
Betriebssysteme / verteilte Systeme
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: 14. Juli 2020

Rechnernetze II

SoSe 2020

8 Netzwerkprogrammierung



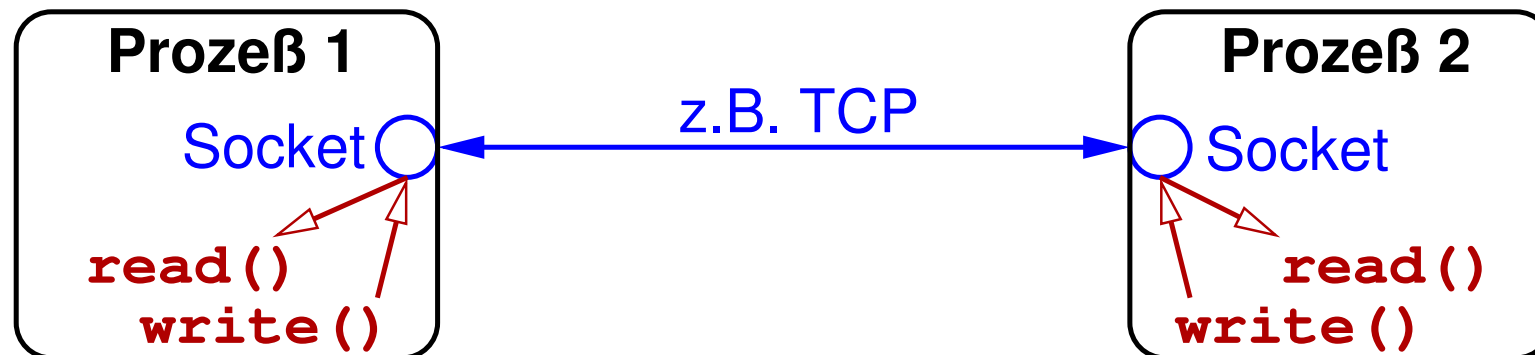
Inhalt

- ➔ Sockets
- ➔ Datagramm-Kommunikation (UDP)
- ➔ Strom-Kommunikation (TCP)
- ➔ Design von Server-Programmen

- ➔ W.R. Stevens: Programmieren von UNIX-Netzen, Hanser/Prentice Hall, 1992. Kap. 6 und 18.3
- ➔ T. Langner: Verteilte Anwendungen mit Java, Markt + Technik, 2002. Kap. 3

Socket-Schnittstelle

- ➔ API (*Application Programming Interface*) für die Interprozeß-Kommunikation
 - ➔ Prozesse auf demselben oder verschiedenen Rechnern
 - ➔ unabhängig vom Netzwerkprotokoll
 - ➔ eingeführt mit BSD 4.2 Unix (1981, *Berkeley Sockets*)
 - ➔ auch in Windows-Betriebssystemen verfügbar
- ➔ **Socket**: Abstraktion für Kommunikationsendpunkt





Erzeugung eines Sockets

➔ Systemaufruf `socket`

➔ `int fd = socket(int domain, int type, int protocol);`

➔ `domain`: Kommunikations-Bereich

➔ lokal, Internet, ...

➔ `type`: Socket-Typ

➔ Datenstrom, Datagramme, ...

➔ `protocol`: zu verwendendes Protokoll

➔ nötig, wenn Socket-Typ mehrere Protokolle unterstützt

➔ in der Regel mit 0 besetzt

➔ `fd`: Dateideskriptor des Sockets

➔ bzw. -1 bei Fehler



Kommunikations-Bereiche

- ➔ Legen fest:
 - ➔ Kommunikation lokal oder über Netzwerk
 - ➔ verwendbare Kommunikationsprotokolle
 - ➔ genauere Auswahl über `type` und `protocol` Parameter
 - ➔ Aufbau von Namen bzw. Adressen

- ➔ Unterstützte Kommunikations-Bereiche:
 - ➔ `PF_UNIX`: *UNIX Domain*, rechnerlokale Kommunikation
 - ➔ `PF_INET`: *Internet Domain*, TCP, UDP, IP
 - ➔ etliche andere (IPv6, Novell, X.25, Appletalk, ...)



Die wichtigsten Socket-Typen

➔ SOCK_STREAM: *Stream Socket*

➔ verbindungsorientierte, strombasierte Kommunikation

➔ in *Internet Domain*: TCP

```
int fd = socket(PF_INET, SOCK_STREAM, 0);
```

➔ SOCK_DGRAM: *Datagram Socket*

➔ verbindungslose Datagramm-Kommunikation

➔ in *Internet Domain*: UDP

```
int fd = socket(PF_INET, SOCK_DGRAM, 0);
```

➔ SOCK_RAW: *Raw Socket*

➔ verwendet Basisprotokoll des Kommunikations-Bereichs

➔ in *Internet Domain*: IP



Binden an eine Adresse

➔ Systemaufruf `bind`

➔ `int rv = bind(int sockfd, struct sockaddr *addr,
int addrlen);`

➔ `sockfd`: Dateideskriptor des Sockets

➔ `addr`: lokale Adresse

➔ z.B. IP-Adresse und Port

➔ `addrlen`: Länge der Adreß-Datenstruktur

➔ `rv`: Rückgabewert, 0 = OK, -1 = Fehler

➔ **Hinweis:** Ab sofort werden nur noch *Internet Domain Sockets* betrachtet



Adreß-Datenstruktur für *Internet Domain Sockets*

```
➔ struct in_addr { unsigned long s_addr; };  
struct sockaddr_in { short sin_family;  
                    u_short sin_port;  
                    struct in_addr sin_addr; }
```

➔ Beispiel: binde Socket an Port 80

```
struct sockaddr_in addr;  
memset(&addr, 0, sizeof(addr)); // alles auf 0  
addr.sin_family = AF_INET;  
addr.sin_port = htons(80);      // Port 80  
addr.sin_addr.s_addr = htonl(INADDR_ANY);  
if (bind(sockfd, (struct sockaddr *) &addr,  
        sizeof(addr)) < 0) ...
```

➔ Hinweis: `sockaddr_in` ist quasi Unterklasse von `sockaddr`



Binden an eine Adresse, Anmerkungen

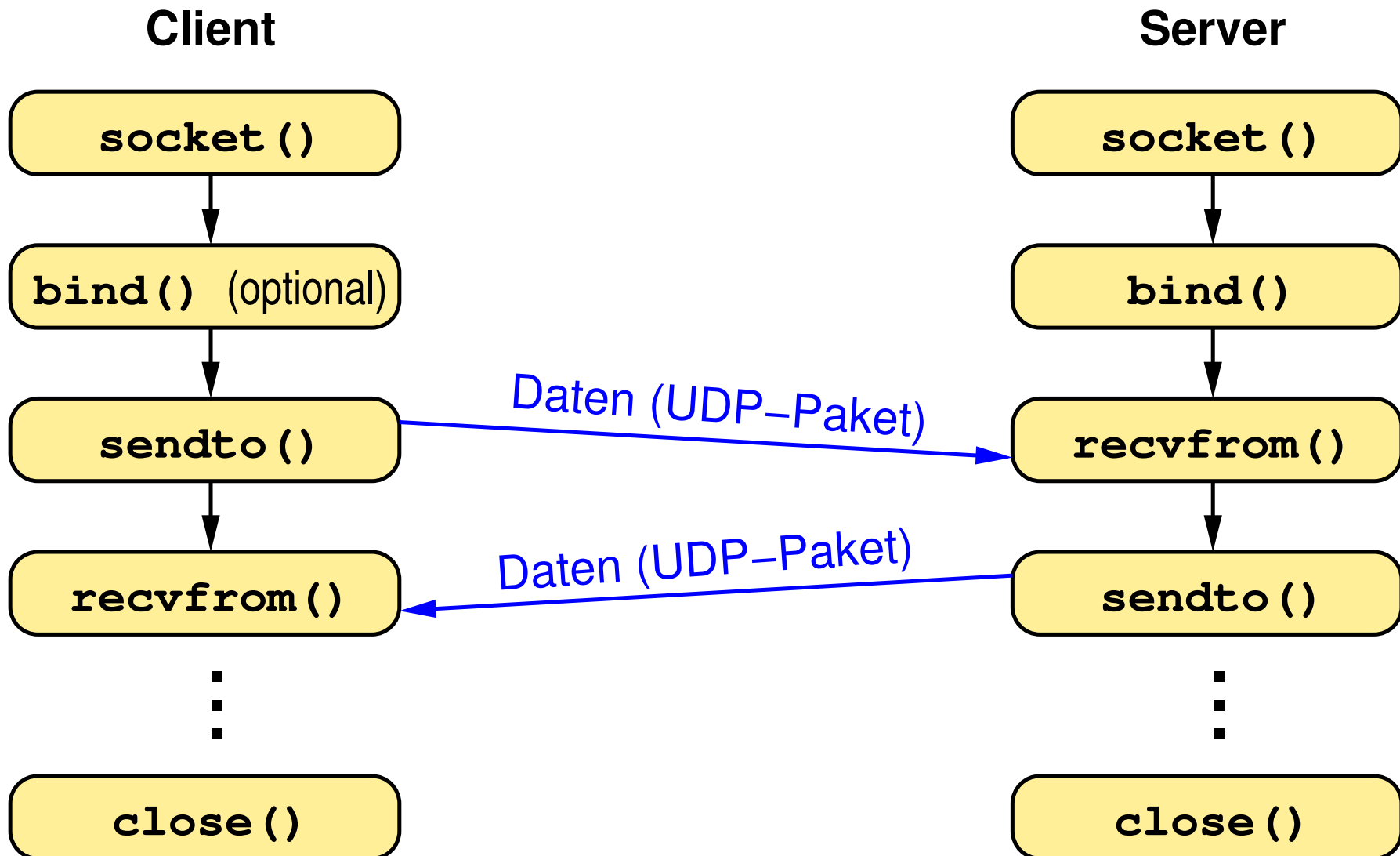
- ➔ bind ist nur für Server-Sockets erforderlich
 - ➔ Client-Sockets wird automatisch ein Port zugewiesen
- ➔ INADDR_ANY bezeichnet beliebiges lokales Netzwerk-Interface
- ➔ Verwendung einer vorgegebenen IP-Adresse, z.B.:
 - ➔ `addr.sin_addr.s_addr = inet_addr("127.0.0.1");`
- ➔ DNS-*Lookup* ist bei Bedarf über die Bibliotheksfunktion `gethostbyname` möglich
- ➔ Die Funktionen `htons` bzw. `htonl` konvertieren vom Host- in das Netzwerk-Datenformat
 - ➔ entsprechend gibt es auch `ntohs` etc.



Schließen eines Sockets

- ➔ Systemaufruf `close`
- ➔ `close(int sockfd);`
 - ➔ `sockfd`: Dateideskriptor des Sockets
- ➔ Bei TCP-Sockets: Verbindungsabbau

8.2 Datagramm-Kommunikation (UDP)





Datagramm-Transfer

➔ Senden eines Datagramms:

```
➔ int sendto(int sockfd, char *msg, int len,  
            int flags, struct sockaddr *addr,  
            int addrlen);
```

➔ msg: zu sendende Nachricht (Länge: len)

➔ flags: Optionen, normalerweise 0

➔ addr, addrlen: Zieladresse

➔ Empfangen eines Datagramms:

```
➔ int recvfrom(int sockfd, char *msg, int len,  
              int flags, struct sockaddr *addr,  
              int *addrlenptr);
```

➔ msg: Empfangspuffer (Länge: len)

➔ addr, addrlenptr: Rückgabe der Senderadresse



Beispiel-Code

➔ Siehe <http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/gen/rn2/code/udp-c.zip>



Datagramm-Sockets in Java

- ➔ Klasse DatagramSocket
 - ➔ Konstruktoren:
 - ➔ DatagramSocket()
 - ➔ DatagramSocket(int port)
 - ➔ bindet Socket an port auf lokalem Rechner, für Server
 - ➔ Methoden:
 - ➔ void send(DatagramPacket p)
 - ➔ Senden eines Datagramms
 - ➔ void receive(DatagramPacket p)
 - ➔ Empfang eines Datagramms



Datagramm-Sockets in Java ...

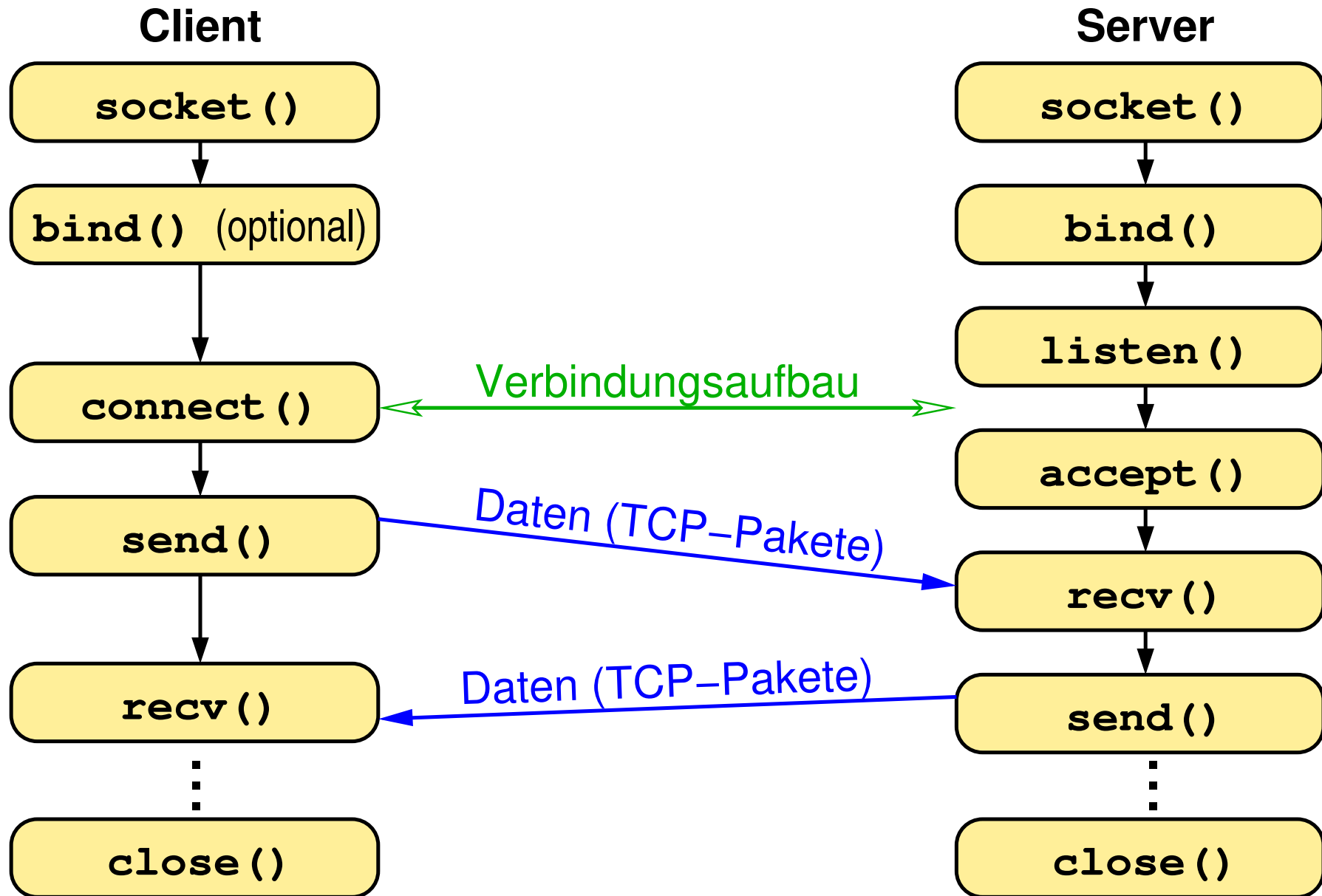
- ➔ Klasse `DatagramPacket`: Abstraktion eines UDP-Pakets
 - ➔ Konstruktoren:
 - ➔ `DatagramPacket(byte[] buf, int len, InetAddress addr, int port)`
 - ➔ für zu sendende Pakete
 - ➔ `DatagramPacket(byte[] buf, int len)`
 - ➔ für zu empfangende Pakete
 - ➔ Methoden:
 - ➔ `InetAddress getAddress()`: IP-Adresse des Pakets
 - ➔ `int getPort()`: Port-Nummer des Pakets



Datagramm-Sockets in Java ...

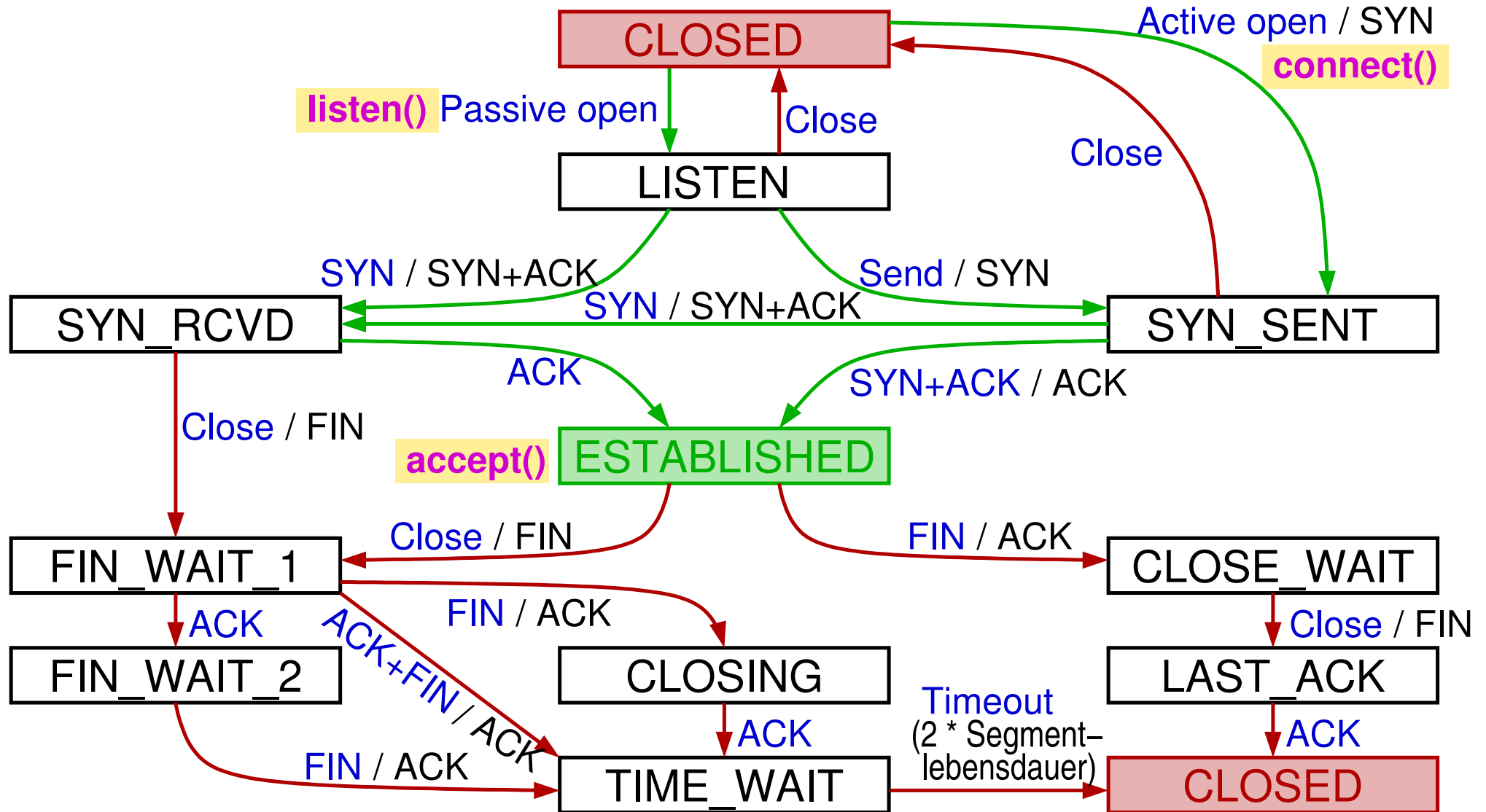
- ➔ Klasse `InetAddress`: IP-Adressen
 - ➔ statische Methoden:
 - ➔ `InetAddress getLocalHost()`
 - ➔ liefert `InetAddress`-Objekt für lokale IP-Adresse
 - ➔ `InetAddress getByName(String host)`
 - ➔ liefert `InetAddress`-Objekt für gegebenen Rechnernamen (oder IP-Adresse in String-Form)
 - ➔ `InetAddress getByAddress(byte[] addr)`
 - ➔ liefert `InetAddress`-Objekt für gegebene IP-Adresse

8.3 Strom-Kommunikation (TCP)





Erinnerung: Zustände einer TCP-Verbindung





Aktives Öffnen einer Verbindung (Client)

- ➔ Systemaufruf connect
- ➔

```
int rv = connect(int sockfd, struct sockaddr *addr,  
                int addrlen);
```

 - ➔ addr, addrlen: legt IP-Adresse und Port des Servers fest
 - ➔ rv: Rückgabewert, 0 = OK, -1 = Fehler
- ➔ Stellt Verbindung mit Socket des Servers her
 - ➔ blockiert, bis Verbindung zustandekommt



Passives Öffnen einer Verbindung (Server)

- ➔ Systemaufruf `listen`
- ➔ `int rv = listen(int sockfd, int backlog);`
 - ➔ `backlog`: legt fest, wieviele Verbindungswünsche gepuffert werden können
 - ➔ `rv`: Rückgabewert, 0 = OK, -1 = Fehler
- ➔ Teilt dem Betriebssystem mit, daß es Verbindungswünsche für diesen Socket entgegennehmen soll



Akzeptieren einer Verbindung (Server)

- ➔ Systemaufruf `accept`
- ➔

```
int fd = accept(int sockfd, struct sockaddr *addr,  
              int *addrlenptr);
```

 - ➔ `addr, addrlenptr`: Rückgabe der Clientadresse
 - ➔ `fd`: **neuer** Socketdeskriptor zur Kommunikation mit diesem Client
- ➔ Bearbeitet ersten Verbindungswunsch aus Puffer
 - ➔ blockiert, wenn kein Verbindungswunsch vorliegt
- ➔ Server kann Verbindung erst nach `accept` zurückweisen (durch `close(fd)`)



Datentransfer

➔ Senden (Schreiben)

➔ `int send(int sockfd, char *msg, int len, int flags);`

➔ `int write(int sockfd, char *msg, int len);`

➔ Empfangen (Lesen)

➔ `int recv(int sockfd, char *msg, int len, int flags);`

➔ `int read(int sockfd, char *msg, int len);`

➔ `write / read` ist äquivalent zu `send / recv` mit `flags = 0`

➔ Ergebnis: geschriebene / gelesene Bytes bzw. -1 bei Fehler

➔ Lesen blockiert, bis mindestens 1 Byte empfangen wurde

➔ Ergebnis 0 bei `read/recv`: Verbindung v. Partner geschlossen

➔ Datendarstellung muß ggf. selbst konvertiert werden!

➔ Funktionen `htons`, `ntohs`, `htonl`, ...



Datentransfer ...

➔ send / recv (und write / read) können zurückkehren, bevor die Daten komplett gesendet / empfangen wurden

➔ Daher: immer Verwendung in einer Schleife, z.B.:

```
char *msg = "HELLO cs.tum.edu\n";
int len = strlen(msg);
int written = 0, res;
while (written < len) {
    res = write(fd, &msg[written], len-written);
    if ((res < 0) && (errno != EINTR)) {
        perror("write"); exit(1);
    }
    if (res >= 0) written += res;
}
```




Adreß-Information

➔ `int getsockname(int sockfd, struct sockaddr *addr,
int *addrlenptr)`

➔ lokale IP-Adresse und Port des Sockets

➔ `int getpeername(int sockfd, struct sockaddr *addr,
int *addrlenptr)`

➔ IP-Adresse und Port des Kommunikationspartners

Beispiel-Code

➔ Siehe <http://www.bs.informatik.uni-siegen.de/web/wismueller/v1/gen/rn2/code/tcp-c.zip>



Stream Sockets in Java

➔ Klasse Socket

➔ Konstruktor: `Socket sock = new Socket();`

➔ zur Erzeugung eines Client-Sockets

➔ auch serverseitig verwendet (nach `accept()`)

➔ Serverseitig: Klasse `ServerSocket`

➔ Konstrukturen:

➔ `ServerSocket sock = new ServerSocket(int port);`

➔ oder `new ServerSocket(int port, int backlog);`

➔ Konstruktoren führen auch `bind()` und `listen()` aus

➔ Methode `Socket accept()`

➔ Client-Adresse durch Socket-Methoden zu erfragen



Stream Sockets in Java ...

- ➔ Verbinden eines Sockets (clientseitig)
 - ➔ über Methode `connect` der Klasse `Socket`:
 - ➔

```
byte[] b = { (byte)217, 72, (byte)195, 42 };  
InetAddress addr = InetAddress.getByAddress(b);  
sock.connect(new InetSocketAddress(addr, port));
```
 - ➔ bzw. mit DNS-Lookup, z.B.:

```
sock.connect(new InetSocketAddress("www.web.de",  
                                   80));
```
 - ➔ oder direkt über Konstruktor der Klasse `Socket`:
 - ➔

```
Socket sock = new Socket(addr, port);
```
 - ➔

```
Socket sock = new Socket("www.web.de", 80);
```



Stream Sockets in Java ...

- ➔ Weitere wichtige Methoden von Socket:
 - ➔ `void close()`
 - ➔ Schließen des Sockets (TCP-Verbindungsabbau)
 - ➔ `InetAddress getLocalAddress()` u. `int getLocalPort()`
 - ➔ liefern lokale IP-Adresse / Port
 - ➔ `InetAddress getInetAddress()` und `int getPort()`
 - ➔ liefern IP-Adresse / Port des Kommunikationspartners
 - ➔ `InputStream getInputStream()`
 - ➔ liefert Eingabestrom zum Lesen vom Socket
 - ➔ `OutputStream getOutputStream()`
 - ➔ liefert Ausgabestrom zum Schreiben auf den Socket



Stream Sockets in Java ...

➔ Lesen vom Socket (typisch):

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(  
        socket.getInputStream()  
    ));  
String request = in.readLine();
```

➔ Schreiben auf einen Socket (typisch):

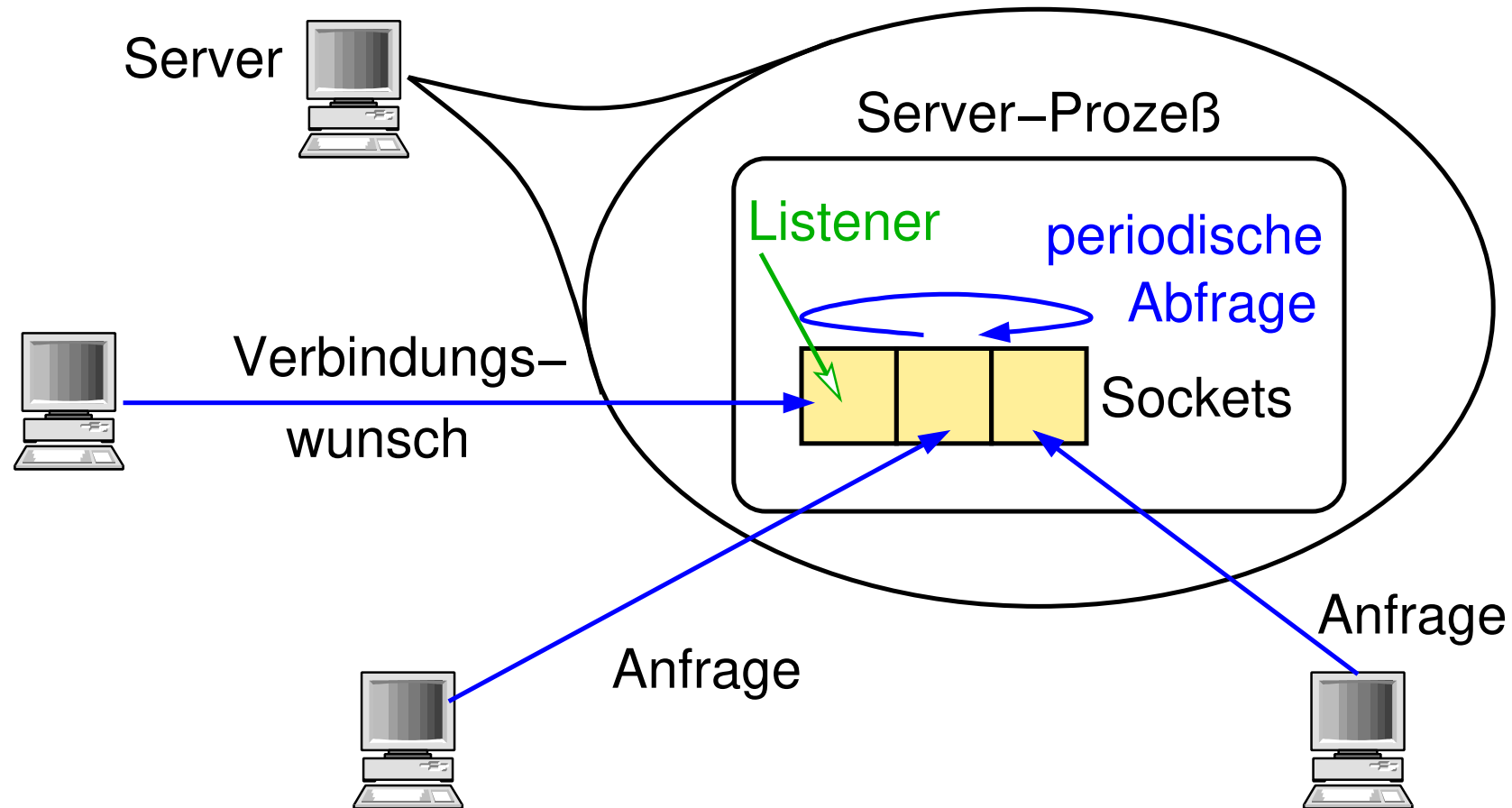
```
PrintWriter out = new PrintWriter (  
    socket.getOutputStream()  
);  
out.println("This is the reply!");  
out.flush();
```



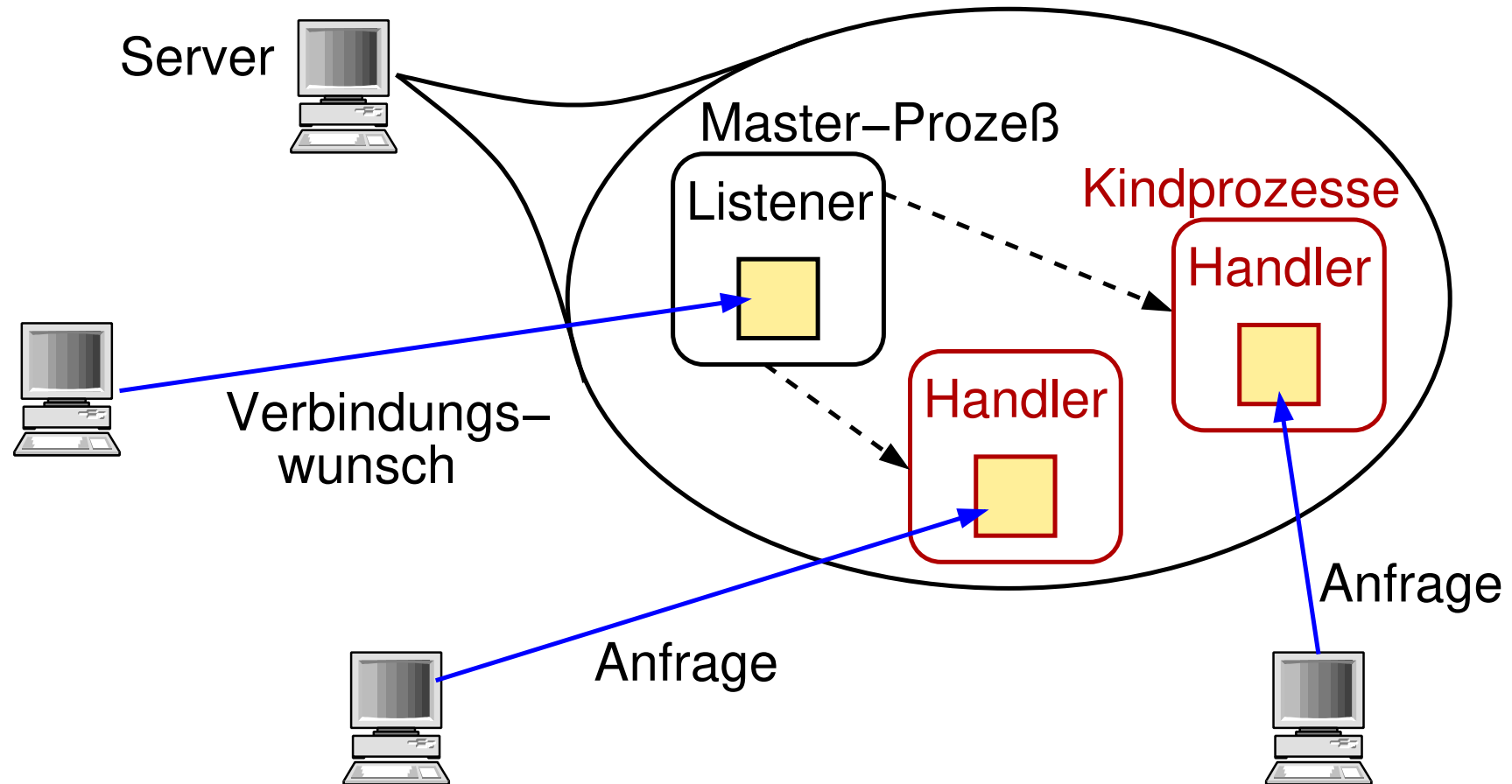
- ➔ Server sollte mehrere Clients gleichzeitig bedienen können
 - ➔ d.h. Server muß gleichzeitig mehrere Sockets auf eingehende Nachrichten abfragen können

- ➔ Alternativen:
 - ➔ *Polling*: Sockets nacheinander nichtblockierend abfragen
 - ➔ durch Optionen des Sockets bzw. von `recv()`
 - ➔ für jeden Client einen neuen Prozeß erzeugen
 - ➔ UNIX: Systemaufruf `fork()` erzeugt Prozeßkopie
 - ➔ für jeden Client einen neuen Thread erzeugen
 - ➔ z.B. in Java
 - ➔ blockierendes Warten an einer Menge von Sockets
 - ➔ UNIX: Systemaufruf `select()` erlaubt Warten an einer Menge von Dateideskriptoren

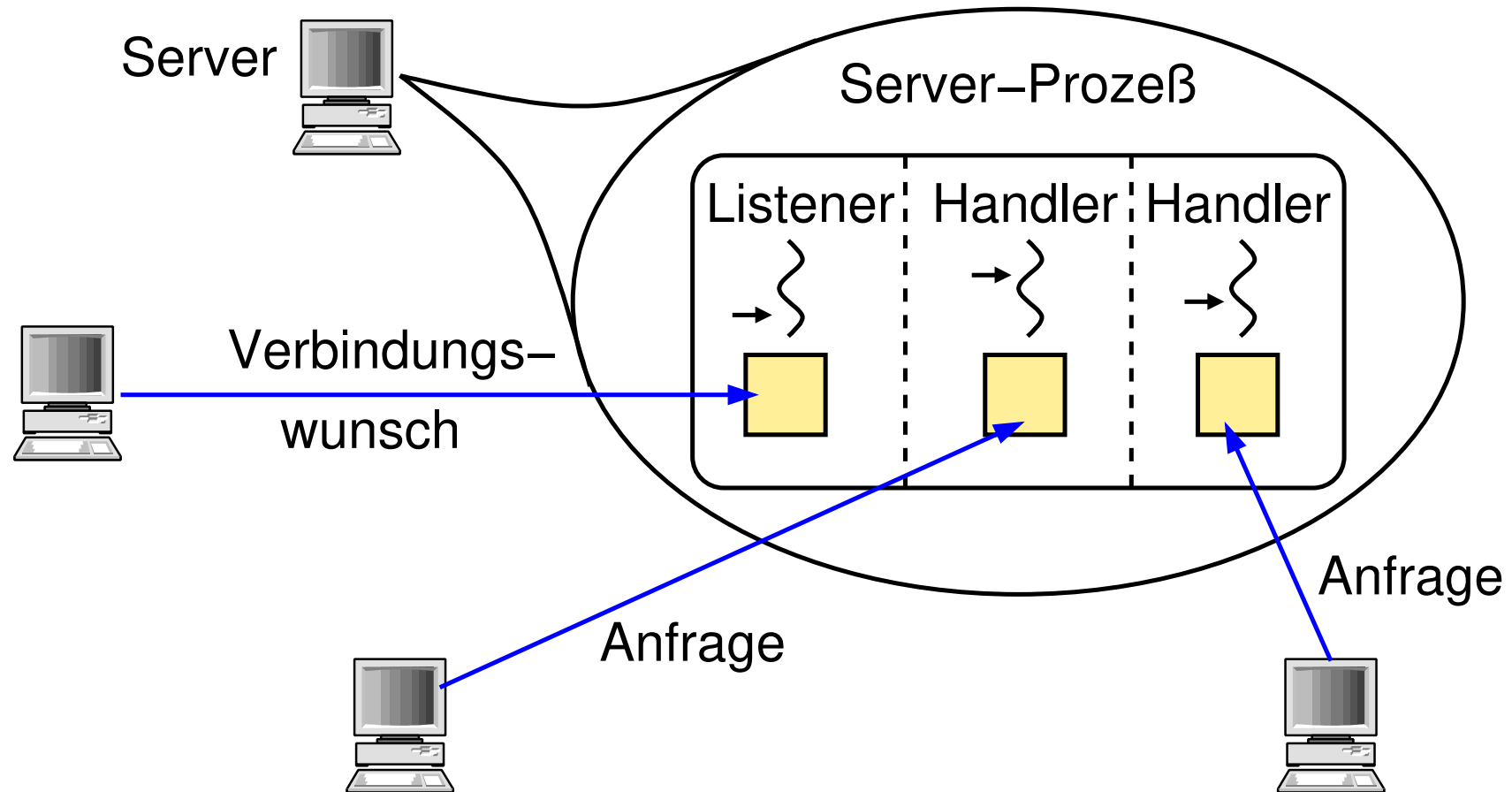
Polling Server



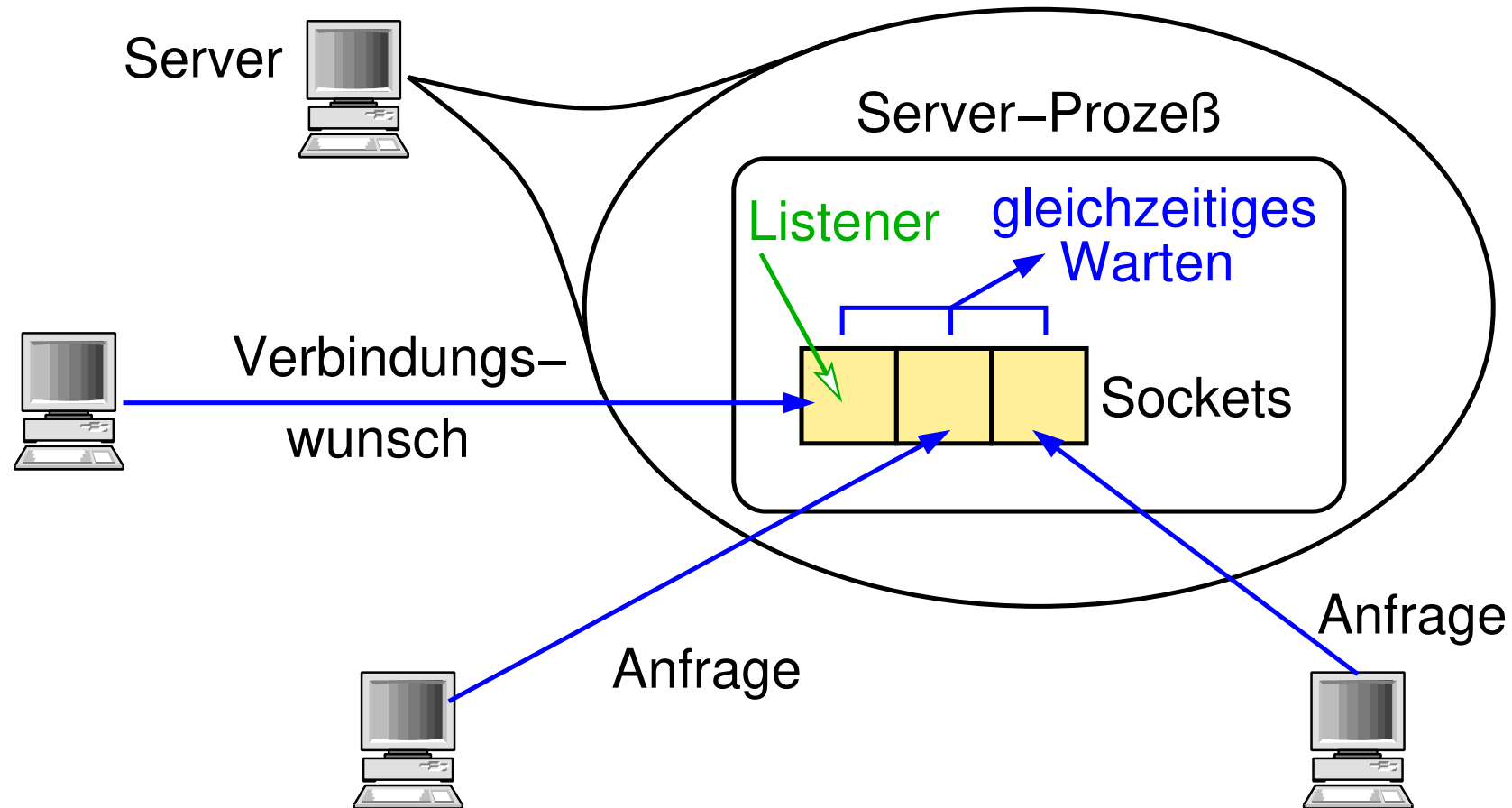
Prozeß pro Client



Thread pro Client



Select-basierter Server





Diskussion

- ➔ *Polling*: verschwendet CPU-Zeit
- ➔ Prozeß pro Client:
 - ➔ Problem: Ressourcenverbrauch bei vielen Clients
 - ➔ Schutz der einzelnen Server-Prozesse gegeneinander
- ➔ Thread pro Client:
 - ➔ weniger ressourcenintensiv als Prozeß pro Client
 - ➔ einfache Nutzung gemeinsamer Daten zw. den Threads
- ➔ Select-basierter Server:
 - ➔ benötigt keine zusätzlichen Ressourcen
 - ➔ Aufträge werden rein sequentiell verarbeitet (keine Synchronisation, aber langer Auftrag blockiert alle folgenden)



Socket-Programmierung

- ➔ Socket: Abstraktion für Kommunikationsendpunkt
- ➔ Socket-API unabhängig von Netzwerk und Protokollen
- ➔ Datagramm- und *Stream* Sockets (mit IP: UDP / TCP)
- ➔ Datagramm-Kommunikation
 - ➔ `socket()`: Erzeugen eines Sockets
 - ➔ `bind()`: binden an IP-Adresse / Port (für Server)
 - ➔ `sendto()`: Senden eines Datagramms
 - ➔ `recvfrom()`: Empfang eines Datagramms
- ➔ Java-Schnittstelle für Datagramm-Sockets
 - ➔ Klasse `DatagramSocket`
 - ➔ Methoden `send()` und `receive()`



Socket-Programmierung ...

➔ *Stream-Sockets*

- ➔ `listen()`: passives Öffnen des TCP-Ports durch Server
- ➔ `connect()`: TCP-Verbindungsaufbau durch Client
- ➔ `accept()`: Server erhält neuen Socket für akzeptierte TCP-Verbindung
- ➔ `send()` oder `write()` zum Senden von Daten
- ➔ `recv()` oder `read()` zum Empfangen

➔ Java-Schnittstelle für *Stream-Sockets*

- ➔ Klasse `Socket`
 - ➔ Methoden `getInputStream()` und `getOutputStream()`
- ➔ Klasse `ServerSocket`
 - ➔ Konstruktor erledigt `bind()` und `listen()`



Server-Design

- ➔ (Polling), Prozeß pro Client, Thread pro Client, Select-basiert
- ➔ Optimales Design abhängig vom Anwendungsfall