3.2.5. Kommunikation über Sockets mit TCP

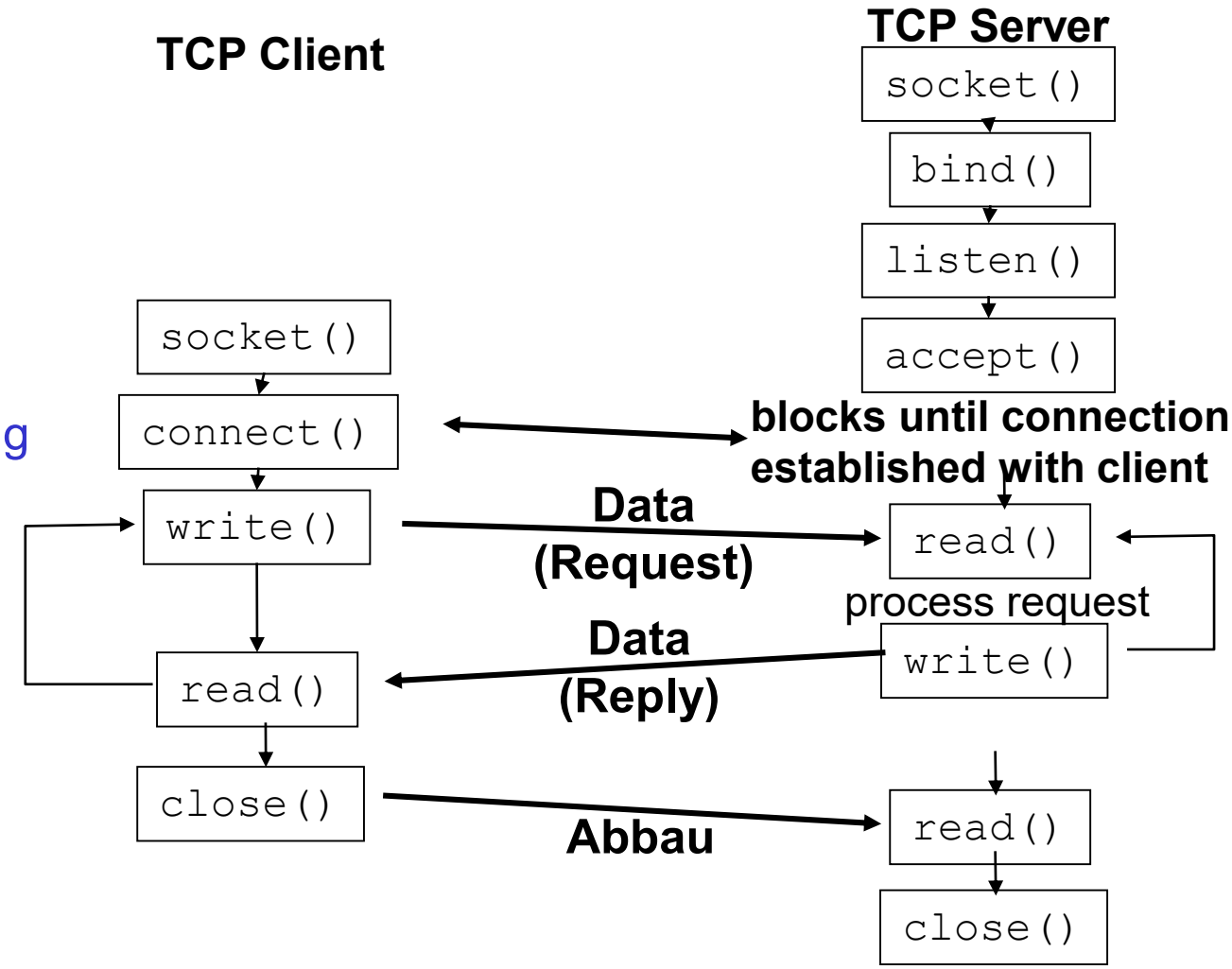
Ablaufschema:

socket() wird in jedem Fall benötigt (aber kein bind()!)

connect() initiiert Aufbau einer Verbindung

Typisch: Zyklus aus Senden, Warten, Empfangen

Typisch: close() beim Client initiiert Abbau der Verbindung



socket() und bind() kennen wir bereits, + listen() „**TCP server bereitet sich vor**“

accept() „**TCP server wartet und akzeptiert Verbindung**“

Typisch: Zyklus aus Empfangen, Verarbeiten, Beantworten

Typisch: Server erfährt vom Abbauwunsch des Clients, dann close()

Erinnerung:

TCP stellt zuverlässigen Kommunikations-Dienst zur Verfügung. Dazu wurde eine „Verbindung“ zwischen den beiden kommunizierenden Endpunkten benötigt.

Der Server wartet auf einen Verbindungsaufbauwunsch. Der Client initiiert den Verbindungsaufbau.

Servervorbereitung: socket(), bind() und listen()

- Wie bereits gesehen, wird zunächst der Socket generiert (hier `socket()` mit SOCK_STREAM für TCP) sowie konkrete Adressen an den Socket gebunden (mit `bind()`, insbesondere die Portnummer).
- Standardmäßig wird der Socket als „aktiver“ Socket generiert, d.h. es wird angenommen, dass ein Client im nächsten Schritt connect() anwendet.

Mittels `listen()` wird der Socket in einen „passiven“ Zustand versetzt:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

Parameter:

- `sockfd` Socket File Descriptor
- `backlog` Anzahl der gleichzeitig möglichen Verbindungsaufbauvorgänge (typischer Wert: 5)

Rückgabewert:

- 0 bei Erfolg
- -1 bei Fehler

listen() und Bedeutung von backlog

- Ein Server soll in der Lage sein, mehrere Verbindungen auf demselben Socket entgegen zu nehmen (Beispiel: WWW-Server, FTP-Server, …).

- Der Verbindungsaufbau beinhaltet einen sog. „Handshake“ zwischen Client und Server, der mehrere Schritte umfasst.

- Der Parameter backlog gibt nun an, wie viele Aufbauvorgänge gleichzeitig beim Server in Bearbeitung sein dürfen (diese befinden sich in der Wartezeit zwischen einem Zustand „passiv“ und Zustand „verbunden“).

Wichtig:

- `backlog` (typischer Wert = 5) beschränkt ***NICHT*** die Anzahl der möglichen gleichzeitigen TCP-Verbindungen zu einem Server-Prozess (mehr dazu bei `accept()`)

- `backlog` beschränkt die Anzahl der Verbindungen, die sich während des Aufbaus sozusagen in einem „Schwebezustand“ befinden

Client initiiert einen Verbindungsaufbau: connect()

- Der Client generiert seinen Socket (ebenfalls mit SOCK_STREAM für TCP). Dann folgt mit `connect()` der Verbindungsaufbau zum gewünschten Kommunikationspartner.

Ähnlichkeit zu UDP sendto():
- hier gibt es Zieladressen
- Damit wird der entfernte Verbindungsendpunkt festgelegt

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

Parameter:
- `sockfd` Socket File Descriptor
- `serv_addr` Zeiger auf Adress-Struktur eines Servers, zu dem eine Verbindung werden soll (→ darin konkrete IP-Adresse + Port!)
- `addrlen` Länge der Adress-Struktur

Rückgabewert:

- 0 bei Erfolg
- -1 bei Fehler

Server erwartet Verbindungsaufbau + accept()

- Der Server-Socket ist mit `listen()` in den passiven Zustand versetzt worden. Nun können mit `accept()` Verbindungsaufbauwünsche angenommen werden:

Ähnlichkeit zu UDP recvfrom():
- hier gibt es ebenfalls Adressen
- Damit kann der initiierende Verbindungsendpunkt erkannt werden

Programmsourcesequenz:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Parameter
- `sockfd` Socket File Descriptor
- `addr` Zeiger auf Adress-Struktur, in der die Adresse des Initiators des Verbindungsaufbaus abgelegt wird (→ darin konkrete IP-Adresse + Port!)
- `addrlen` Zeiger auf Länge der Adress-Struktur

Der Aufruf von `accept()` blockiert bis eine Verbindung zustande gekommen ist!

Rückgabewert:
- Integer-Wert, Descriptor eines neu kreierten Socket, über den die nun erfolgreich aufgebaute Verbindung abgewickelt wird!
- -1 bei Fehler

Beispiel von accept()

```
int sockfd, newsock, res;
sockaddr_in client_addr;
socklen_t addrlen;

…       /* the socket sockfd has been created
        and bound to a port number */

res = listen(sockfd,5);
if (res<0) { ... }

addrlen = sizeof(struct sockaddr_in);

newsock = accept(sockfd,(struct sockaddr *) &client_addr, &addrlen);

if (newsock<0) { ... }
else {
printf("Received connection from %s!\n",\
     inet_ntoa(client_addr.sin_addr));

}
```

Adress-Struktur für Quelladresse wird bereitgestellt

Standard-Aufruf von listen()

Neuer Socket-Descriptor kann ab jetzt benutzt werden!

Der alte Socket „sockfd“ ist weiterhin bereit für neue Verbindungsaufbau-wünsche!

Hilfsfunktion zur Ausgabe der IP-Adresse als Dotted-Decimal

Source

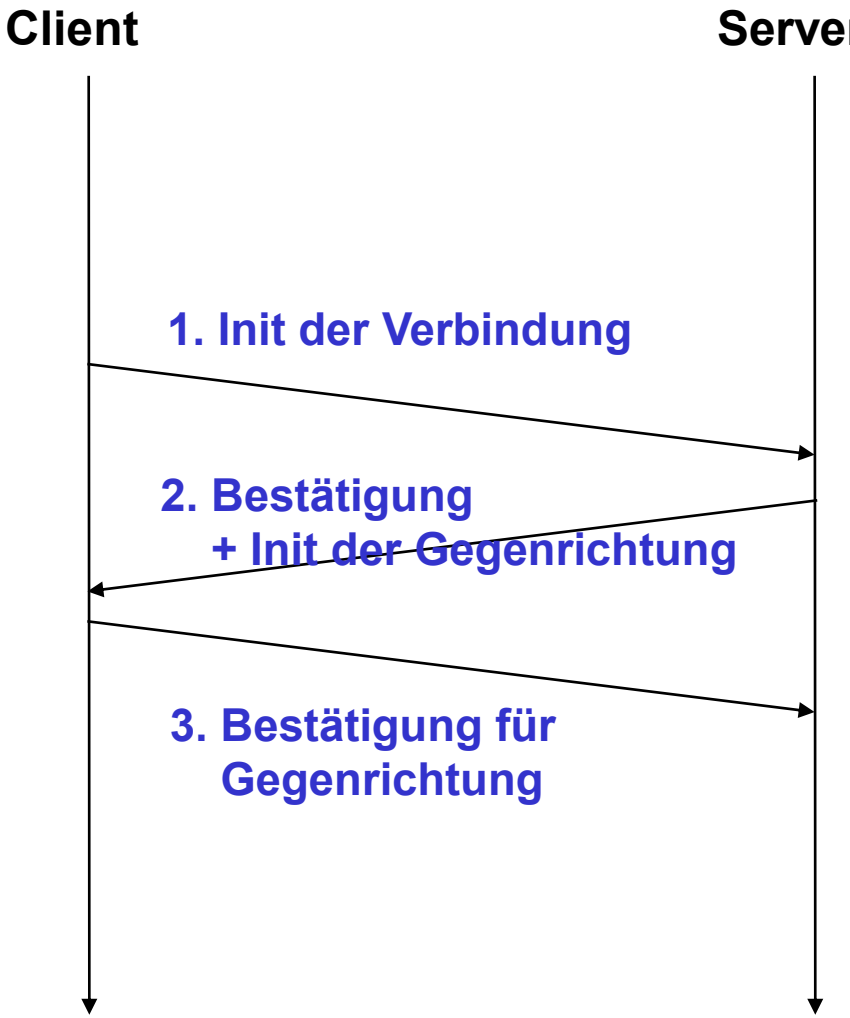
„Schwebezustand“ im Verbindungsaufbau

Ein Client initiiert mit connect() einen Verbindungsaufbau

Aufruf von connect()

Warten, connect() blockiert!

connect() kehrt zurück!



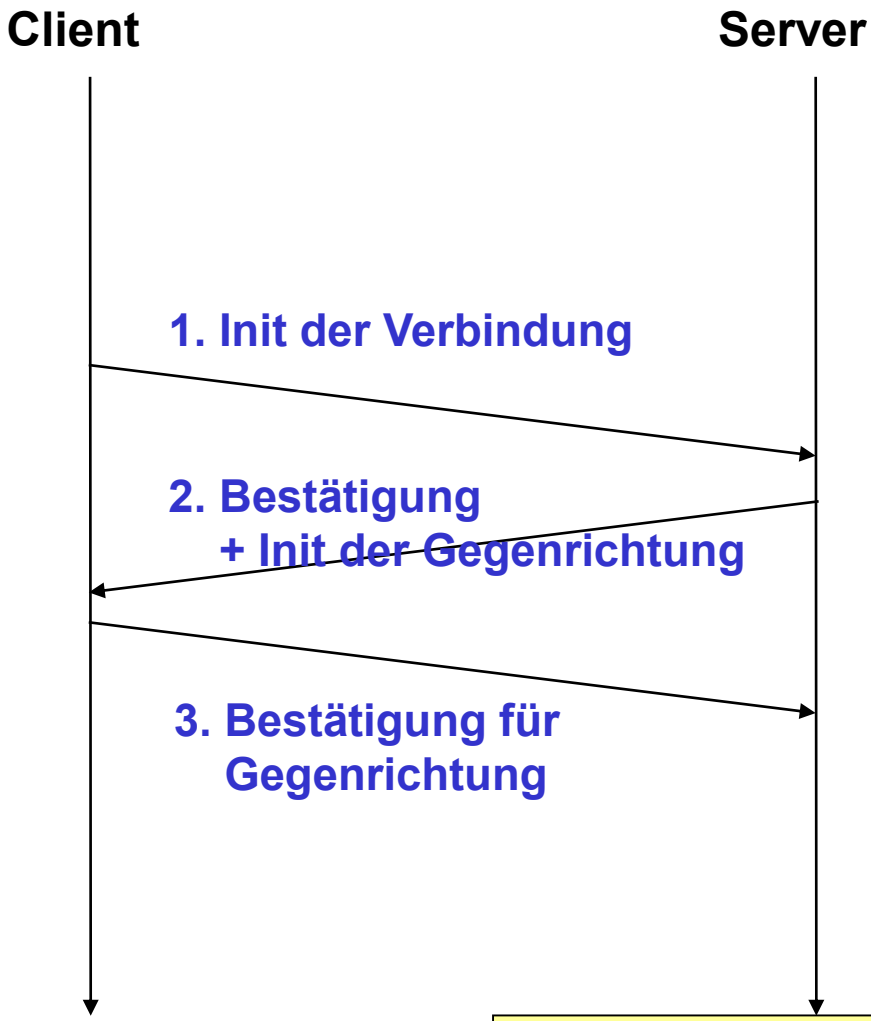
Ein Server geht mit listen() in passiven Zustand

Ein weiterer Aufbauvorgang geht in Bearbeitung!
(Parameter backlog bestimmt die maximale Anzahl der gleichzeitigen Aufbauvorgänge)

Verbindung ist aufgebaut, ein Aufruf von accept() liefert **einen neuen Socket-Descriptor (auf einen sog. connected Socket)**

Der erste Socket ist weiter verfügbar (im passiven Zustand) für weitere Verbindungsaufbauwünsche **(sog. listening Socket)**

Neuer Socket nach connect() und accept()

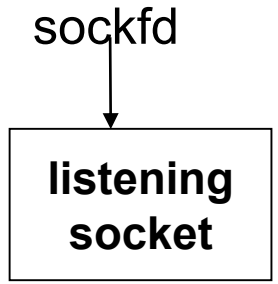


Ein Server geht mit listen() in passiven Zustand

Ein weiterer Aufbauvorgang geht in Bearbeitung!
(Parameter backlog bestimmt die maximale Anzahl)

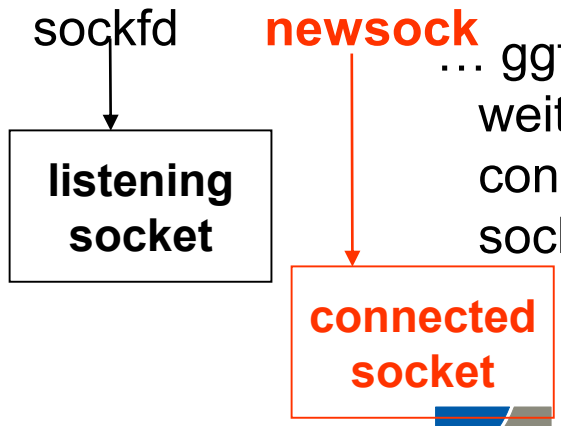
Verbindung ist aufgebaut, ein Aufruf von accept() liefert den neuen Socket-Descriptor

Vorher



… ggf. weitere connected sockets

Nachher



… ggf. weitere connected sockets

Wir werden später betrachten, wie ein Server mit mehreren Sockets, also mehreren Verbindungen umgeht!

Der Client sendet Daten – write()

Der Aufruf ist **ähnlich** wie der von `sendto()` im Falle von UDP.

Programmsourcesequenz:

```
#include <unistd.h>
ssize_t write(int sockfd, const void *buff, size_t count);
```

Parameter:

- `sockfd` File Descriptor des Sockets
- `buff` Zeiger auf den Puffer mit den zu sendenden Daten
- `count` Länge der zu sendenden Daten in Bytes

Rückgabewert:

- >= 0 Anzahl der gesendeten Bytes
- -1 Fehler

Unterschied zu sendto():

- weniger Parameter
- Zieladressen fehlen
  (da der Verbindungsendpunkt ja fest steht, vgl. connect)

Wichtig:

- Rückgabewert von `write()` kann kleiner als `count` sein!
- d.h. es wurden weniger Bytes gesendet als gewünscht!

→ immer Rückgabewert prüfen und ggf. restliche Daten erneut senden

Der Server empfängt Daten – read()

Programmsourcesequenz:

```
#include <unistd.h>
ssize_t read(int sockfd, void *buff, size_t count);
```

Parameter:

- `sockfd` File Descriptor des Sockets
- `buff` Zeiger auf einen **Puffer, in den empfangene Daten geschrieben werden**
- `count` **Länge des Puffers**

Rückgabewert:

- > 0 Anzahl der empfangenen Bytes
- -1 Fehler
- = 0 (Sonderfall: nächste Folie!)

Unterschied zu recvfrom():

- weniger Parameter
- Quelladressen fehlen
  (da der Verbindungsendpunkt ja fest steht, vgl. accept)

Wichtig:

- Rückgabewert von `read()` kann kleiner als `count` sein!
- d.h. es wurden weniger Bytes empfangen als der Buffer aufnehmen kann!

Allgemeines zu read() und write()

Wichtig:

- der Aufruf von `read()` blockiert, bis Daten vom Socket gelesen wurden

Rückgabewert - **Sonderfall**:

- = 0 die Gegenseite kennzeichnet „end-of-file“, d.h. **Wunsch des Verb.abbaus**

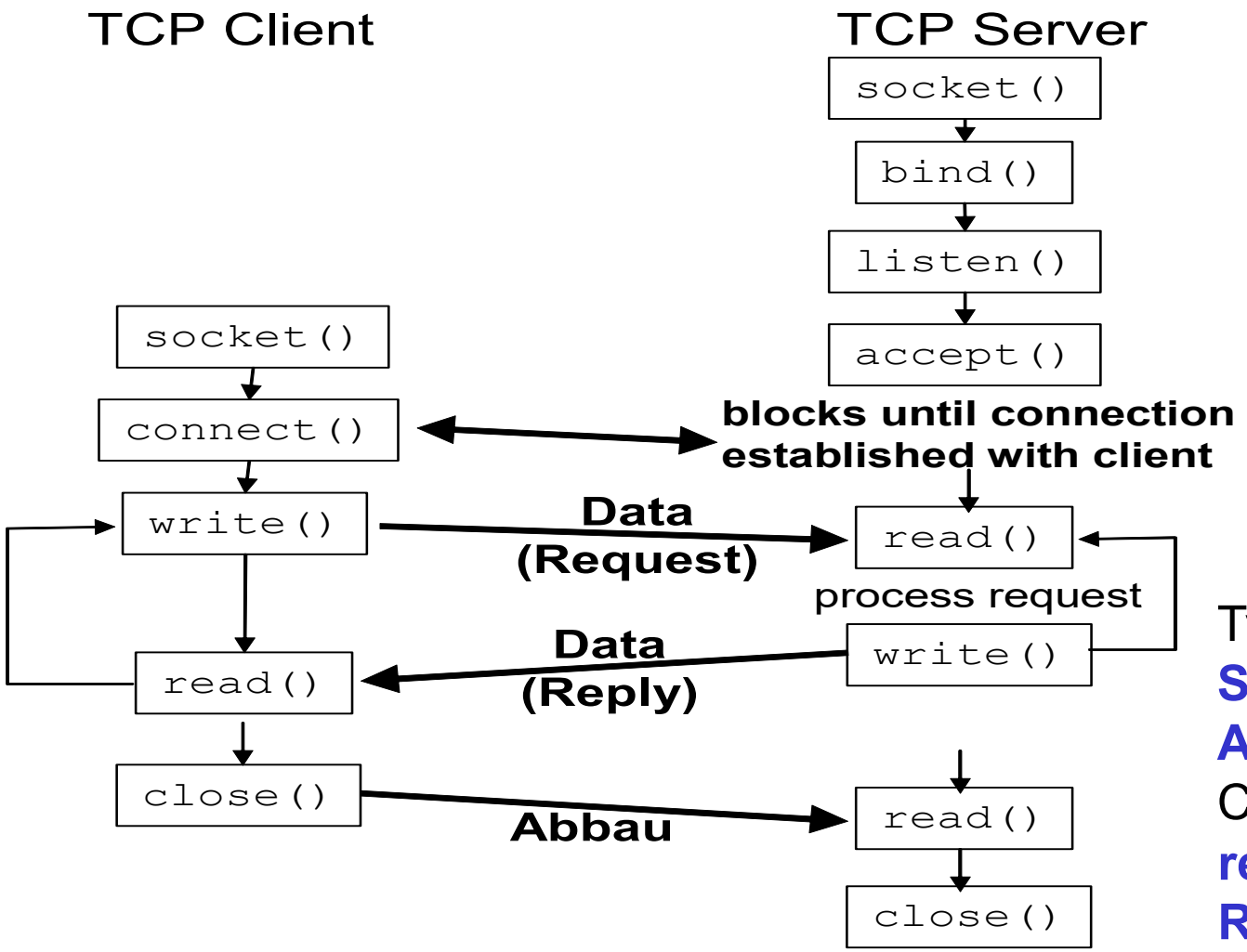
Allgemeines:

- zunächst betrachtet: Client sendet mit `write()`, Server empfängt mit `read()`

- beide Partner einer TCP-Verbindung sind gleichberechtigt, Verb. ist full-duplex
- d.h. auch Gegenrichtung mit Server `write()` und Client `read()`

Verbindungsabbau – close()

Ablaufschema:



Typisch:
close() beim Client initiiert Abbau der Verbindung

Typisch:
Server erfährt vom Abbauwunsch des Clients, dann close()
read() liefert als Rückgabewert 0

→ Aufruf von close()

Aufruf von close()

Programmsourcesequenz:

```
#include <unistd.h>
int close(int sockfd);
```

Parameter:

- `sockfd` File Descriptor des Sockets

Rückgabewert:

- 0 bei Erfolg
- -1 Fehler

Sonderfall:
`close()` markiert den Socket zum schließen.

Falls es mehrere Descriptoren auf den Socket gegeben hat, bleibt der Socket weiter im Betriebssystem bestehen, bis alle Descriptoren `close()` aufgerufen haben (mehr in Kap. 3.4.)

Bemerkungen:
Nach dem Schließen eines Sockets darf auf den Socket-Descriptor nicht mehr zugegriffen werden (weder `read()` noch `write()` möglich).

Bereits an den Socket übergebene, aber von TCP noch nicht gesendete Daten werden noch gesendet.

Alternative zu close() – shutdown()

- Der Aufruf von `close()` schließt den Socket sofort für den aufrufenden Prozess, für `write()` und `read()`– also in beiden Richtungen der Kommunikation.

- `shutdown()` erlaubt das Schließen jeweils einzeln für jede Richtung.

Programmsourcesequenz:

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

Parameter:

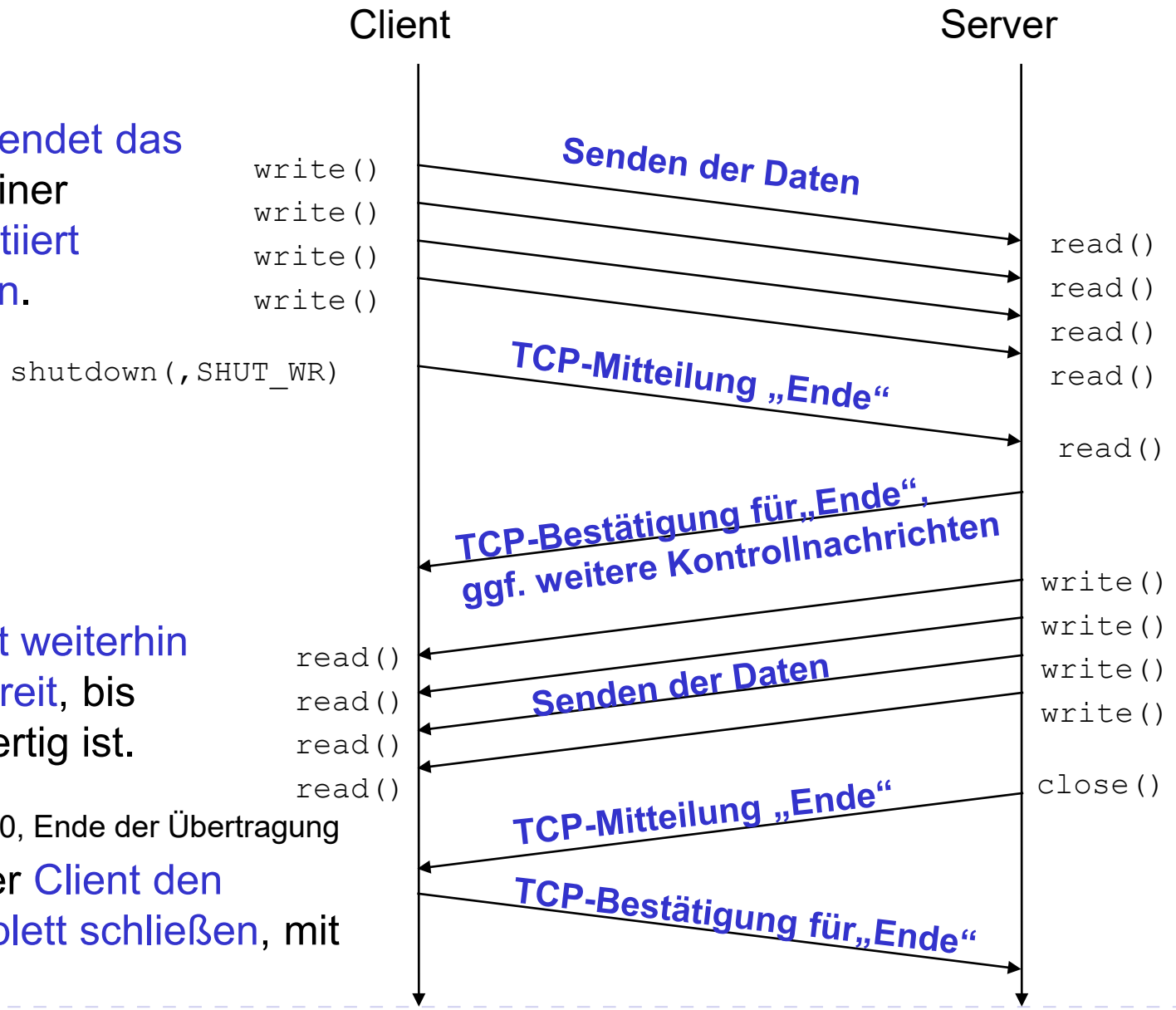
- `Sockfd` File Descriptor des Sockets
- `howto` Variante des Schließens:
  - SHUT_RD Schließen der Leserichtung
  - SHUT_WR Schließen der Schreibrichtung
  - SHUT_RDWR Schließen beider Richtungen

Rückgabewert:

- 0 bei Erfolg
- -1 Fehler

Nach dem entsprechenden Aufruf von `shutdown()` kann der Prozess nicht mehr auf die Funktionen `read()` bzw. `write()` zugreifen!

Beispiel für den Einsatz von shutdown()



Der Client beendet das Schreiben seiner Daten und initiiert den Shutdown.

Nach Aufbau der Verbindung sind Client und Server gleichberechtigte Kommunikationspartner.

Der Client ist weiterhin empfangsbereit, bis der Server fertig ist.

`read()` liefert 0, Ende der Übertragung

Nun kann der Client den Socket komplett schließen, mit close()

Der Server kann nun weiterhin Daten senden, bis er fertig ist.

Der Server schließt nun den Socket.

Rust Socket Library (nur kurz, F.85-91)

Rust bietet eine umfassende Bibliothek mit vielen Paketen

- Ähnlich zu der C-Socket Library
- Module `std::net` für Sockets (TCP, UDP, UNIX), Netzwerk IO, DNS, HTTP etc.

- Weitere Infos und Beispiele unter https://doc.rust-lang.org/stable/std/net/

3.2.5. Kommunikation über Sockets mit TCP

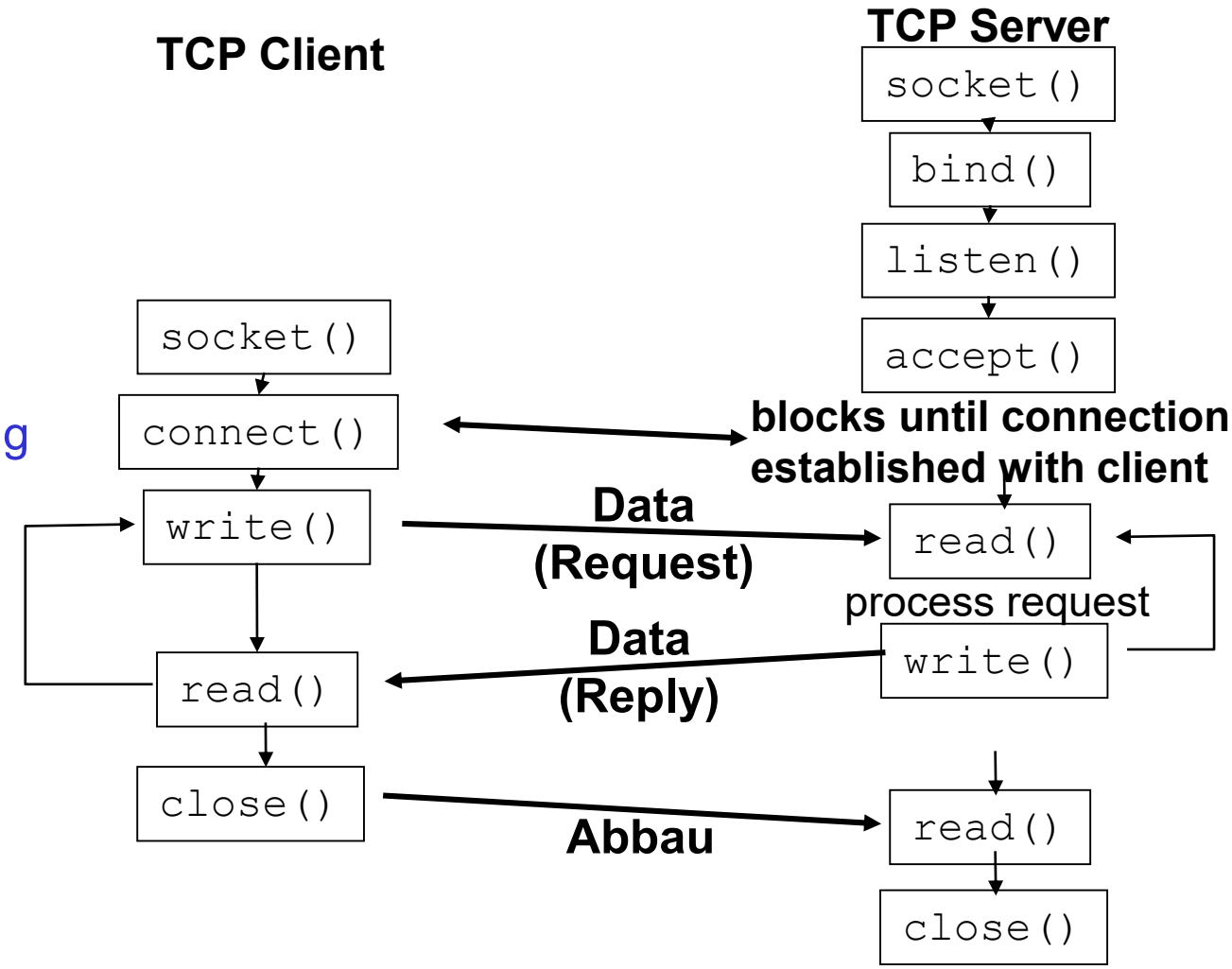
Ablaufschema:

socket() wird in jedem Fall benötigt (aber kein bind()!)

connect() initiiert Aufbau einer Verbindung

Typisch: Zyklus aus Senden, Warten, Empfangen

Typisch: close() beim Client initiiert Abbau der Verbindung



socket() und bind() kennen wir bereits, + listen() „**TCP server bereitet sich vor**“

accept() „**TCP server wartet und akzeptiert Verbindung**“

Typisch: Zyklus aus Empfangen, Verarbeiten, Beantworten

Typisch: Server erfährt vom Abbauwunsch des Clients, dann close()

Erinnerung:

TCP stellt zuverlässigen Kommunikations-Dienst zur Verfügung. Dazu wurde eine „Verbindung“ zwischen den beiden kommunizierenden Endpunkten benötigt.

Der Server wartet auf einen Verbindungsaufbauwunsch. Der Client initiiert den Verbindungsaufbau.

Rust Socket Library

- `connect()` um mit einer Adresse zu verbinden z.B.:
- `TcpStream::connect("127.0.0.1:34254")`
- oder
- ```
  let socket = UdpSocket::bind("127.0.0.1:3400").xy
  socket.connect("127.0.0.1:8080").xy
  ```

- Strings statt Integer-Konstanten
  ```

Rust Sockets (TCP Client)

```rust
use std::io::prelude::*;
use std::net::TcpStream;

{
    let mut stream = TcpStream::connect("127.0.0.1:34254").unwrap();

    // ignore the Result
    let _ = stream.write(&[1]);
    let _ = stream.read(&mut [0; 128]); // ignore here too
} // the stream is closed here
```

(Quelle: https://doc.rust-lang.org/stable/std/net/)

Rust Sockets (TCP Server)

```rust
use std::net::{TcpListener, TcpStream};

fn handle_client(stream: TcpStream) {
    // ...
}

fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:80").unwrap();

    // accept connections and process them serially
    for stream in listener.incoming() {
        handle_client(stream?);
    }
    Ok(())
}
```

(Quelle: https://doc.rust-lang.org/stable/std/net/)

Rust Sockets (UDP Server)

```rust
use std::net::UdpSocket;

fn main() -> std::io::Result<()> {
    {
        let mut socket = UdpSocket::bind("127.0.0.1:34254")?;

        // Receives a single datagram message on the socket. If `buf` is too small to hold
        // the message, it will be cut off.
        let mut buf = [0; 10];
        let (amt, src) = socket.recv_from(&mut buf)?;

        // Redeclare `buf` as slice of the received data and send reverse data back to origin.
        let buf = &mut buf[..amt];
        buf.reverse();
        socket.send_to(buf, &src)?;
    } // the socket is closed here
    Ok(())
}
```

(Quelle: https://doc.rust-lang.org/stable/std/net/)

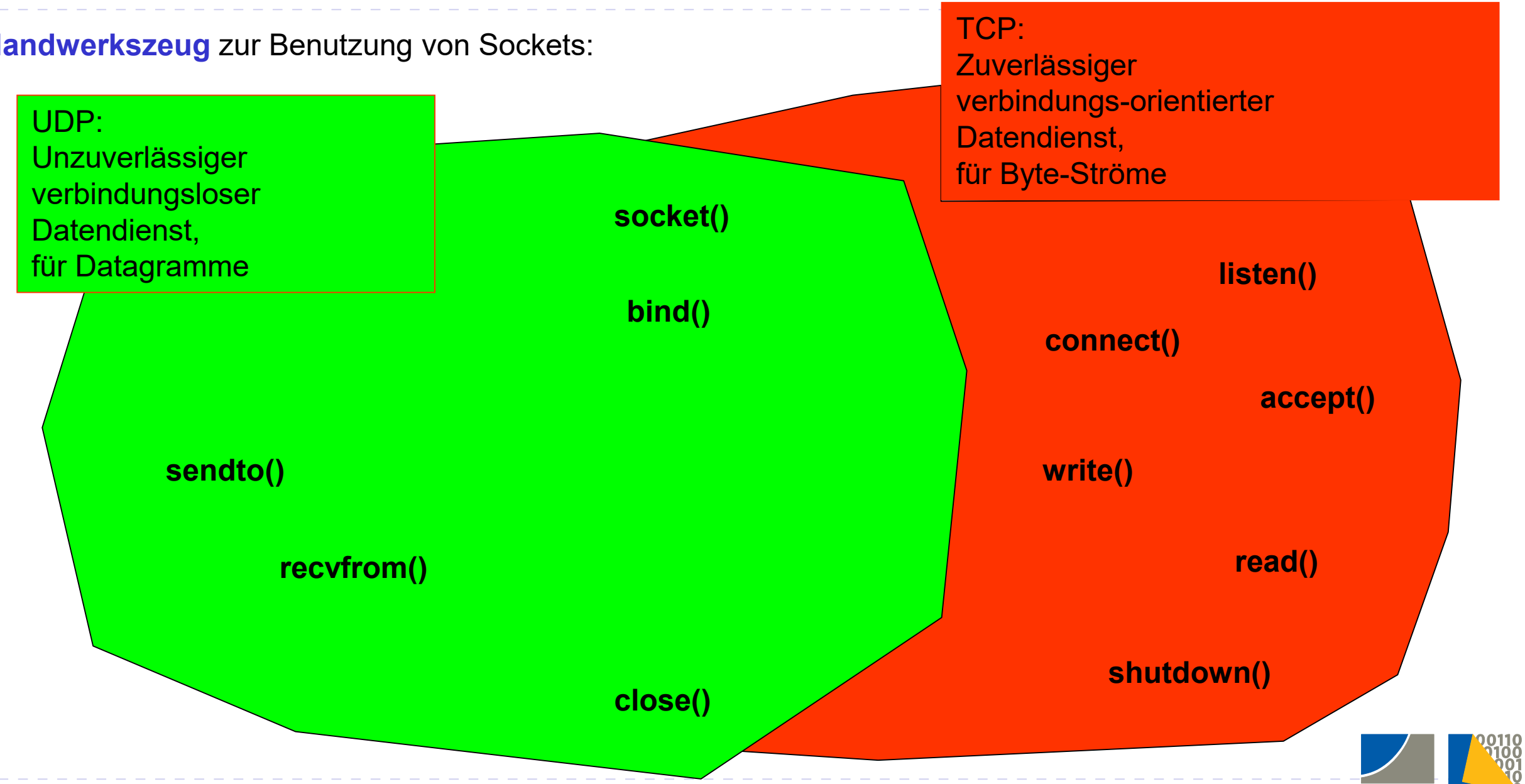
Rust in Action

```
extern crate reqwest;

use std::io::Read;

fn run() -> Result<()> {
    let mut res = reqwest::get("http://httpbin.org/get")?;
    let mut body = String::new();
    res.read_to_string(&mut body)?;

    println!("Status: {}", res.status());
    println!("Headers:\n{}", res.headers());
    println!("Body:\n{}", body);

    Ok(())
}
```

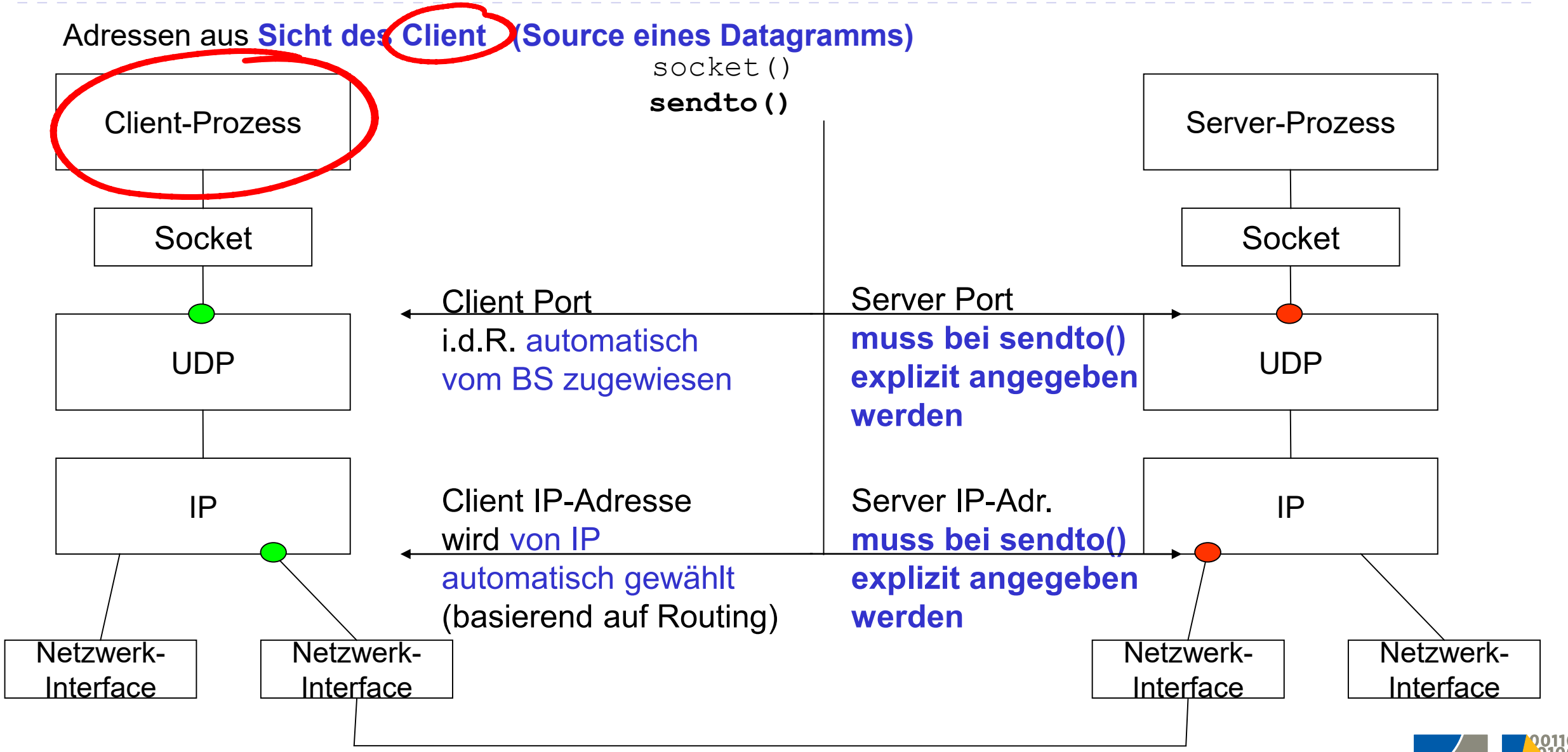
(Quelle: https://doc.rust-lang.org/stable/std/net/)

3.2.6. Zwischenresümee: Was haben wir bis jetzt gesehen?

Handwerkszeug zur Benutzung von Sockets:

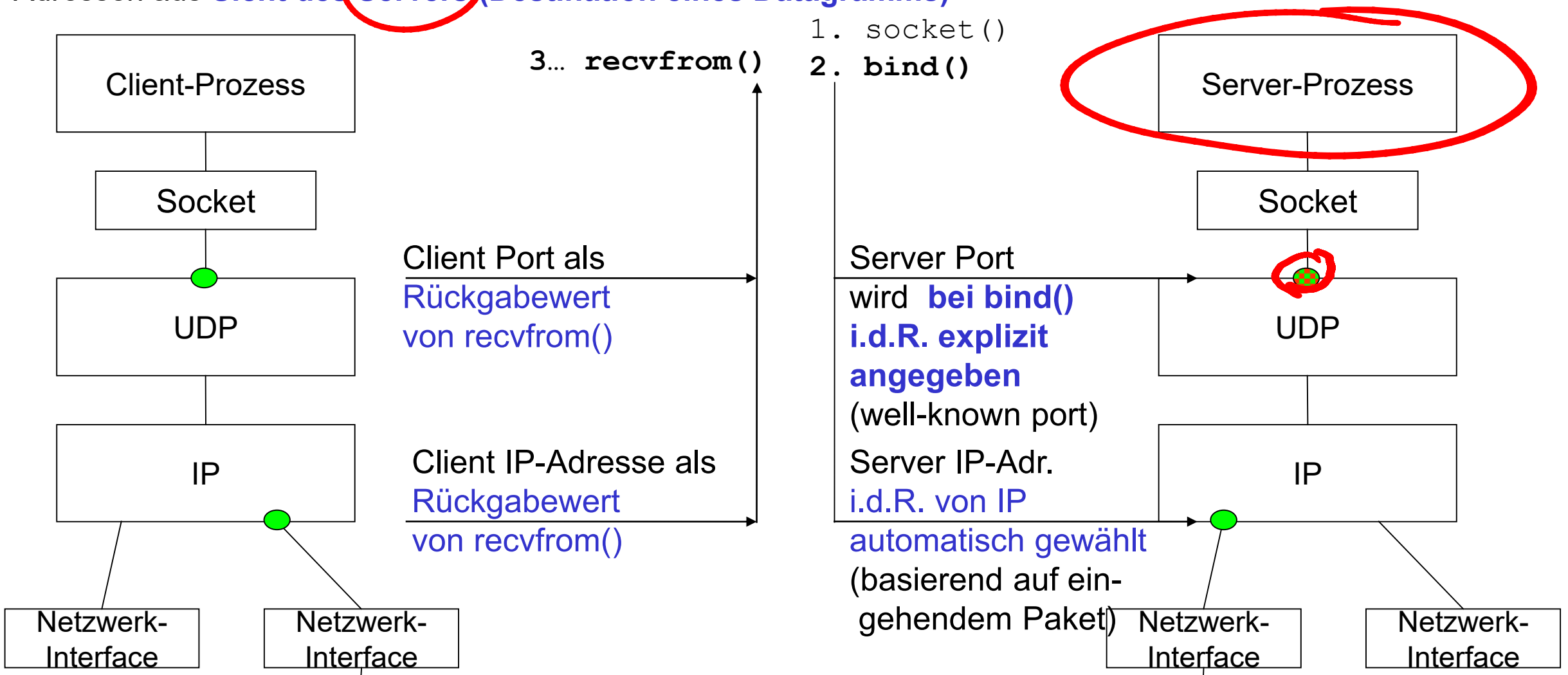


Übersicht: Adressen bei UDP



Übersicht: Adressen bei UDP

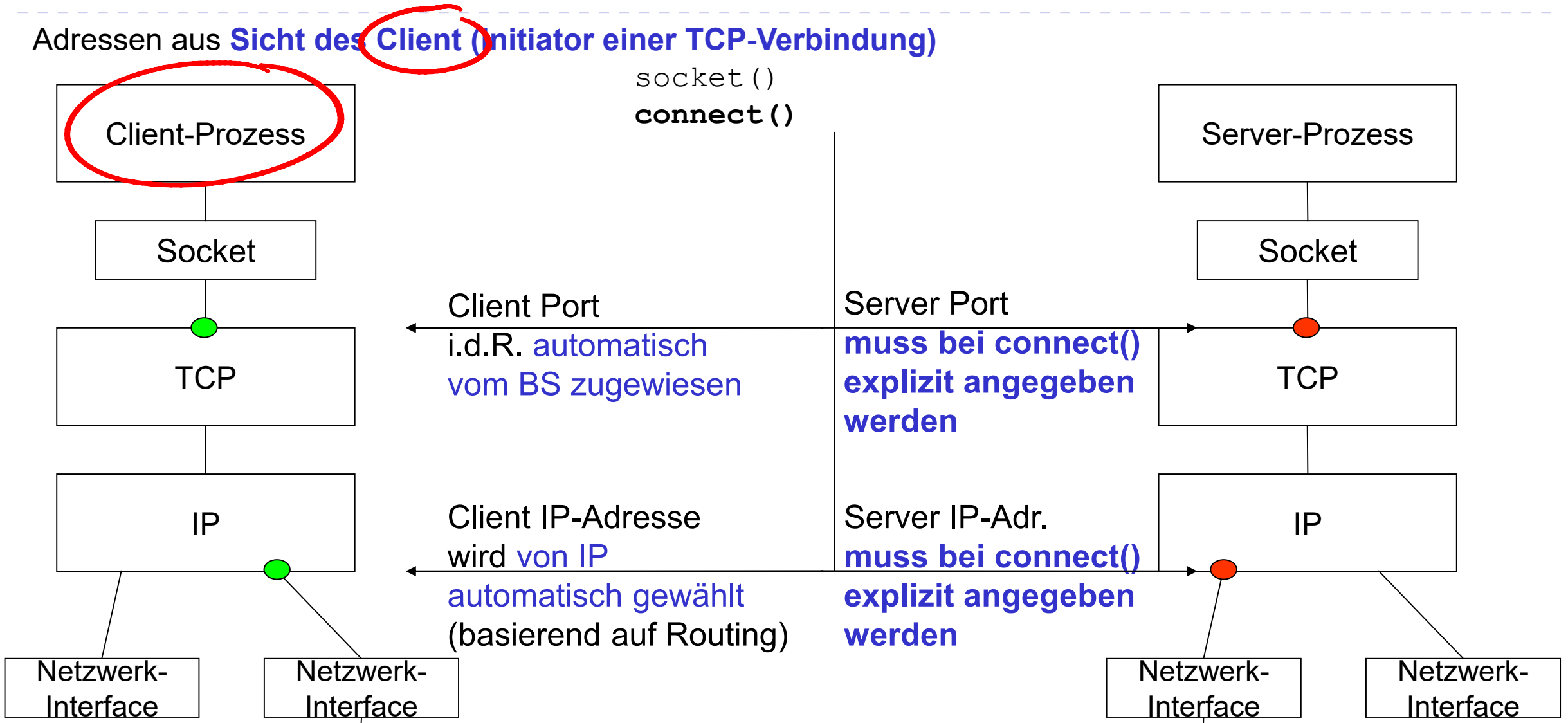
Adressen aus **Sicht des Servers (Destination eines Datagramms)**



= (grün) Adresse i.d.R. automatisch bestimmt = (rot) Adresse muss explizit angegeben werden

Übersicht: Adressen bei TCP

Adressen aus **Sicht des Client (Initiator einer TCP-Verbindung)**

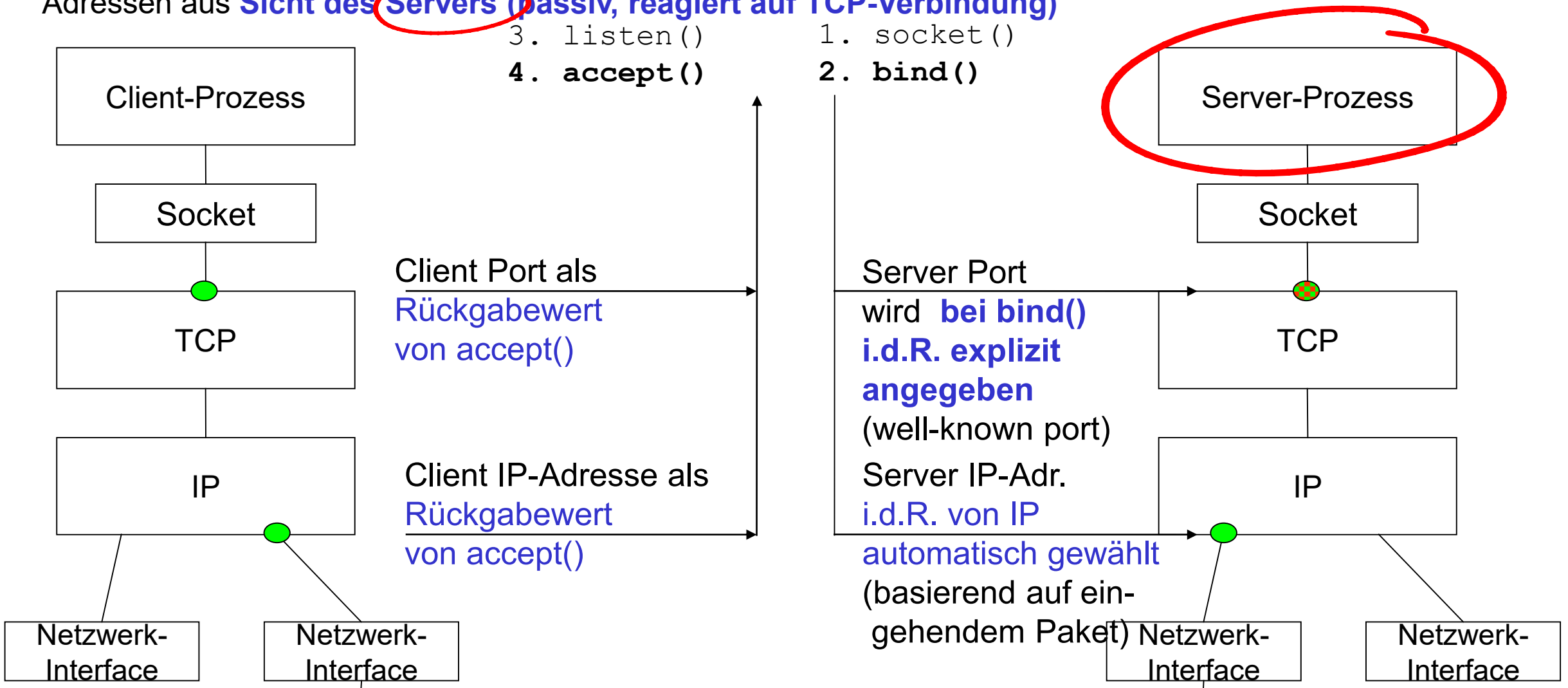


= (grün) Adresse i.d.R. automatisch bestimmt

= (rot) Adresse muss explizit angegeben werden

Übersicht: Adressen bei TCP

Adressen aus **Sicht des Servers (passiv, reagiert auf TCP-Verbindung)**



= (grün) Adresse i.d.R. automatisch bestimmt = (rot) Adresse muss explizit angegeben werden

Hilfsfunktionen für TCP und UDP

Sowohl bei UDP als auch TCP werden die **Adressen in der Regel automatisch bestimmt**:

- Portnummer beim Client (dynamisch aus verfügbarer Menge)
- IP-Adresse beim Client (abhängig vom besten Weg zum Ziel, wenn mehrere Netzwerkinterfaces vorhanden sind)
- IP-Adresse beim Server (abhängig vom eingehenden Paket, wenn mehrere Netzwerkinterfaces vorhanden sind)
- ggf. Portnummer beim Server (auch dynamisch möglich, wenn kein well-known Port genutzt werden kann oder soll)

Wie kann nun ein Client oder Server herausfinden, welches die konkreten Adressen sind, die das Betriebssytem automatisch gewählt hat:

- `recvfrom()` teilt dem UDP-Server Adressen des Absenders eines Datagramms mit
- `accept()` teilt dem TCP-Server Adressen des Initiators der Verbindung mit

Zwei weitere Funktionen, mit denen **ein Endpunkt seine eigenen (dynamischen) Adressen erfragen kann**:

- `getsockname()`
- `recvmsg()`

Hilfsfunktionen für TCP und UDP

Übersicht aller **Hilfsunktionen zur Adressbestimmung**:

Gewünschte Information aus dem IP-Datagramm des Clients (Quelle= Client, Destination = Server)	**TCP Server**	**UDP Server**
Quell-IP-Adresse	`accept()` oder auch `getpeername()`	`recvfrom()`
Quell-Portnummer	`accept()` oder auch `getpeername()`	`recvfrom()`
Destination-IP-Adresse	`getsockname()`	`recvmsg()`
Destination-Portnummer	`getsockname()`	`getsockname()`

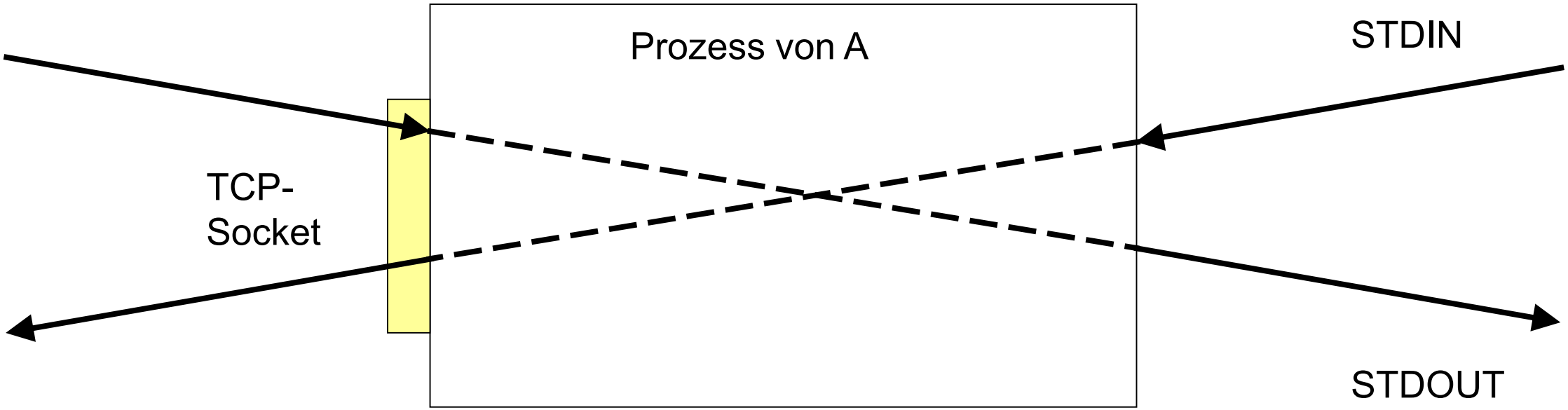
Das Kleingedruckte:

- wir betrachten hier getsockname(), getpeername() und recvmsg() nicht weiter im Detail (bei Bedarf siehe R. Stevens et al. 2003)
- recvmsg() ist eine der allgemeinsten I/O-Funktionen, recvfrom() und auch read() könnten durch Aufrufe von recvmsg() mit bestimmter Parameterausprägung ersetzt werden
- die Wahl von „name“ in getsockname() und getpeername() ist unglücklich und hat nichts mit Internet-Namen und DNS zu tun. Sie liefern den „Namen“ eines Sockets, im Falle von AF_INET eine Adress-Struktur mit IP-Adresse und Portnummer

3.3. Input/Output (I/O) Multiplexing

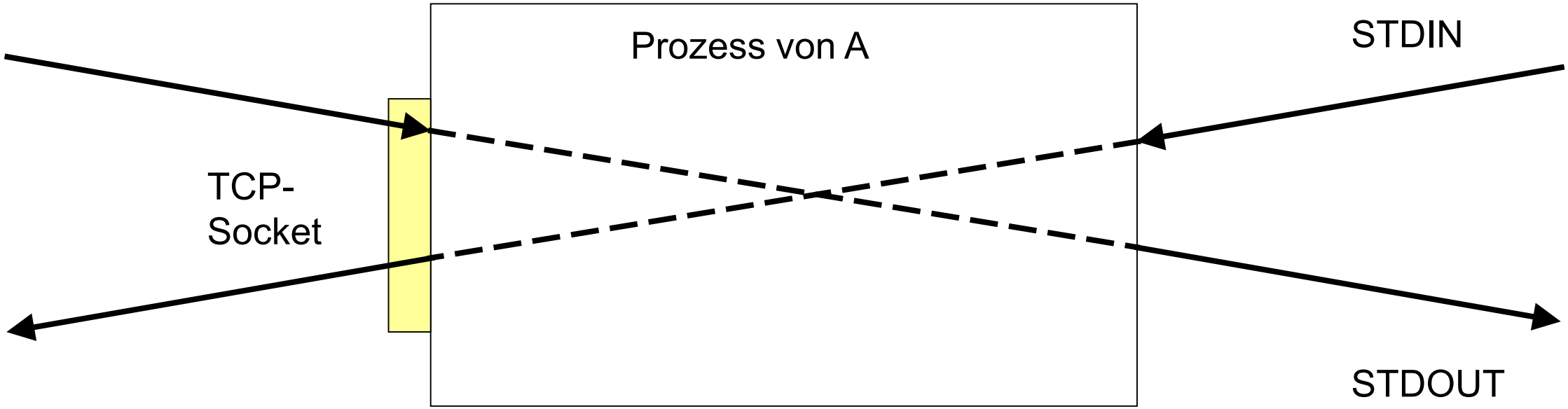
Motivation:

Wir wollen ein Programm A schreiben, das gleichzeitig auf Eingaben von verschiedenen Quellen reagieren kann:



Konkret:
Das Programm erwartet Eingaben vom TCP-Socket und vom Nutzer (via STDIN) und soll entsprechend Ausgaben auf STDOUT bzw. auf den TCP-Socket erzeugen.

Motivation



Wie können wir die Verwaltung mehrerer Eingabequellen realisieren?

Herausforderung:

- die Eingabefunktionen von Sockets blockieren typischerweise,
  d.h. der Funktionsaufruf kehrt erst zurück, wenn eine Eingabe vorliegt
  - `recvfrom()` bei UDP nach Ankunft eines Datagrammes
  - `accept()` bei TCP nach Beendigung eines Verbindungsaufbaus
  - `read()` bei TCP nach Ankunft von Daten (mind. 1 Byte oder EOF)
- ebenso blockiert `fgets( , , stdin)` beim Einlesen von der Tastatur

Weitere Beispiele für Bedarf an I/O-Multiplexing

Das angegebene **Beispiel stellt einen generischen TCP-Client** dar,
Einsatz z.B. wie bei Telnet, Rlogin, …

Weitere Beispiele:

- ein Server soll gleichzeitig Anfragen über TCP und UDP entgegen nehmen können
- ein Programm/Prozess soll gleichzeitig Daten von mehreren Sockets entgegen nehmen können (Bsp.: ein Web-Client, HTML-Seite mit eingebetteten Grafiken, nutze mehrere TCP-Verbindungen parallel)
- ein TCP-Server soll gleichzeitig seinen listening Socket (für Verb.aufbau) und seine connected Sockets bedienen
- ein Serverprozess soll gleichzeitig mehrere Dienste und mehrere Protokolle bedienen können

I/O-Multiplexing ist nicht nur auf Netzwerkprogrammierung beschränkt!

Modelle und Lösungen für I/O-Multiplexing

- **Übersicht über verschiedene Ansätze**:

- blockierender I/O (unsere gegebene Situation)

- nicht-blockierender I/O

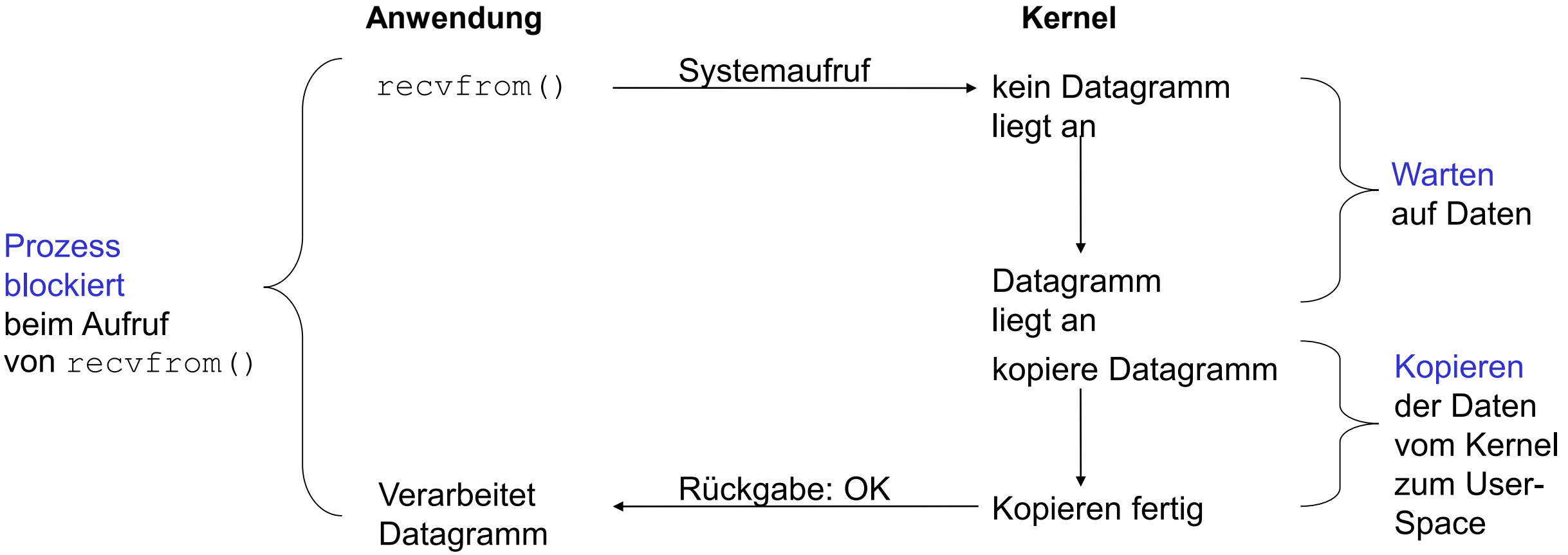
- Signal-gesteuerter Ablauf mit Interrupt

- I/O Multiplexing mit speziellen Hilfsfunktionen

Modell des blockierenden I/O

Eine **Eingabeoperation besteht typischerweise aus 2 Phasen**:

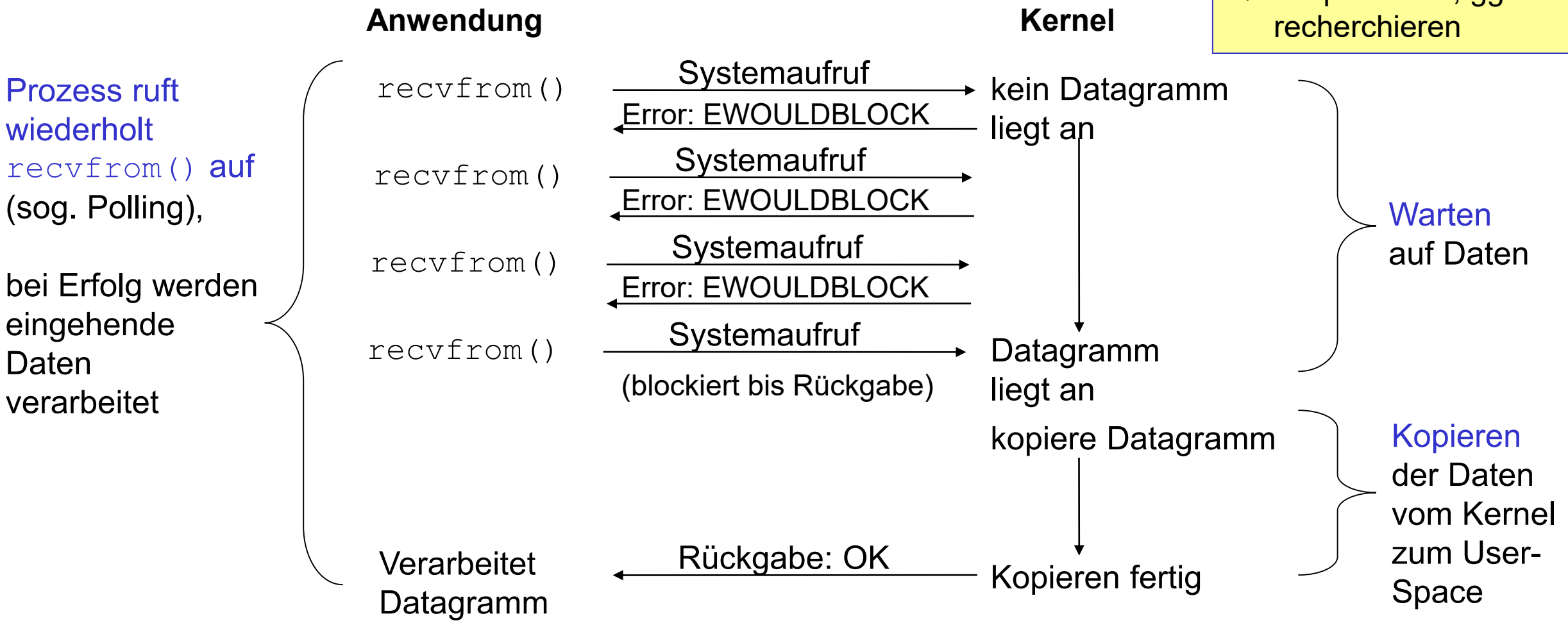
1. Warten auf das „Eintreffen“ von Daten (Tastatur, Netzwerk, …)
2. Kopieren der Daten vom Kernel zum Prozess



Modell des nicht-blockierenden I/O

EWOULDBLOCK ist auf den meisten Systemen synonym zu **EAGAIN**
⇒ ausprobieren, ggf. recherchieren

Zur Vorbereitung muss zunächst ein Socket (oder ein allg. File Descriptor) in den sog. non-blocking Mode gesetzt werden.



Beispiel zum non-blocking I/O

Nutzung der Systemfunktion `fcntl()` (File Control) um Socket in den non-blocking Mode zu setzen:

Programmsourcesequenz:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

int fcntl(int fildes, int cmd, /* arg */ ...);
```

#include wichtig für Auslesen der Fehlernummer

Typ des dritten Arguments abhängig von cmd

Bespielverwendung:

```
int flags, err;
   /* sockfd sei Descriptor eines TCP-Sockets */

flags = fcntl(sockfd, F_GETFL, 0);
err = fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
```

(Parameter-Schlüssel sind in fcntl.h bzw. sys/fcntl.h definiert)

Beispiel zum non-blocking I/O

Der Zugriff auf den Socket könnte nun wie folgt stattfinden:

Programmsourcesequenz:

```
while (! done) {
…
  if ( (n=read(STDIN_FILENO,…)<0))
{
 if (errno != EWOULDBLOCK)
        /* ERROR */
}
  else write(sockfd,…)
…
  if ( (n=read(sockfd,…)<0))
{
 if (errno != EWOULDBLOCK)
        /* ERROR */
}
  else write(STDOUT_FILENO,…)
…
}
```

Prüfe `read()` auf Fehler

`errno` sagt genaue Fehlernummer

`EWOULDBLOCK/EAGAIN` ist gewünscht, d.h. Rückkehr ohne Daten
→ Weiterlaufen des Prozesses mit anderen Anweisungen

Erfolgreiches `read()`

Dito für nächste Eingabequelle

Anmerkung:
Auch STDIN muss zuerst auf non-blocking Mode gesetzt werden

Diskussion zu non-blocking I/O

- **Nachteil von nicht-blockierendem Input**:

- non-blocking I/O realisiert ein sog. „Busy Waiting“

- im Bsp. läuft die While-Schleife endlos, solange bis an einer der Eingabequellen Daten anliegen

- das „Busy Waiting“ verbraucht unnötig CPU-Zeit (der Prozess bleibt aktiv)

Anders bei blockierendem Input:

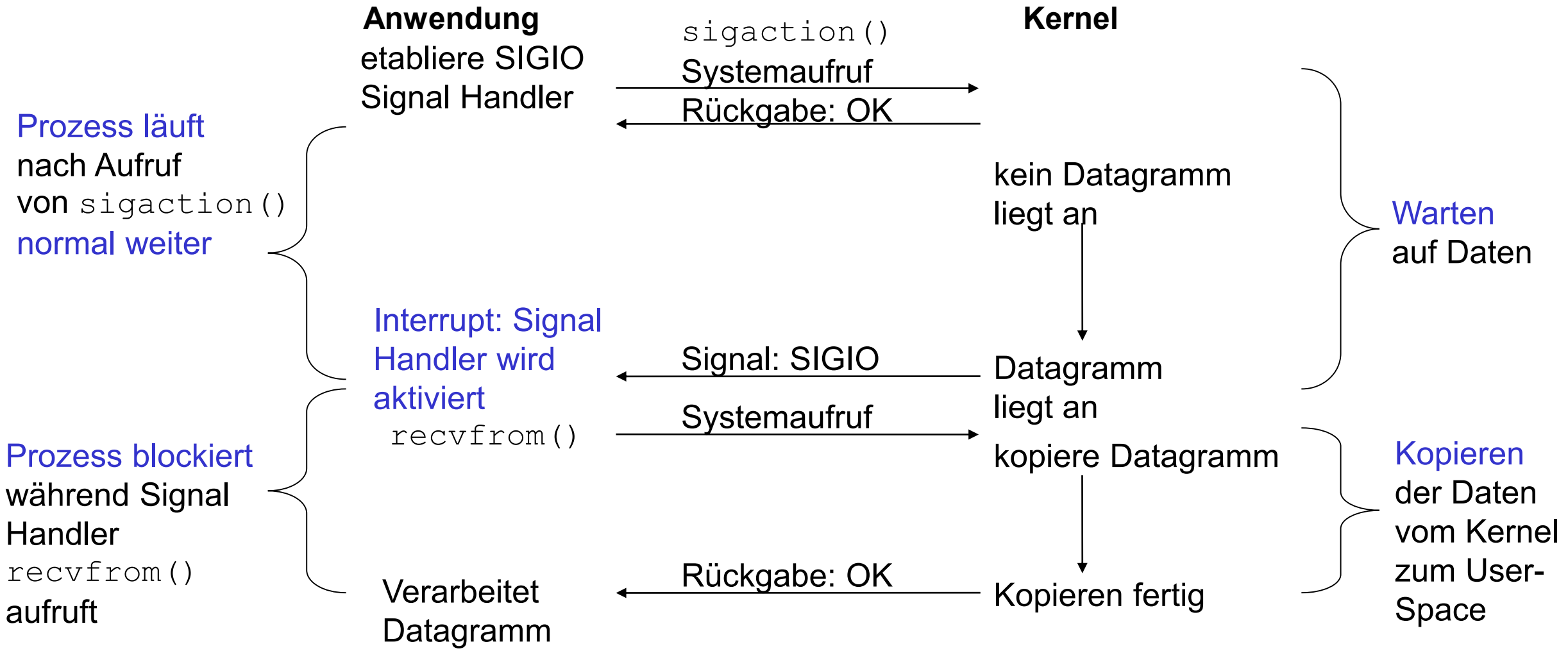
- beim blockierenden Aufruf einer Eingabefunktion wie `read()` wird der Prozess vom Betriebssystem in den sog. Sleep-Modus versetzt

- der Prozess verbraucht keine CPU-Zeit

- wenn Eingabedaten anliegen, wird der Prozess vom BS geweckt und der Funktionsaufruf kehrt zurück

Modell des Signal-gesteuerten I/O

Zur Vorbereitung muss zunächst der Socket für signal-driven I/O vorbereitet und ein sog. Signal Handler für das Signal SIGIO aktiviert werden.



Diskussion zu Signal-gesteuertem I/O

Zwei Alternativen:

1. der Signal Handler ruft `recvfrom()` auf und übergibt (irgendwie) das Datagramm an die Hauptschleife
2. der Signal Handler teilt (irgendwie) der Hauptschleife mit, dass sie mit `recvfrom()` ein Datagramm vom Socket holen kann

Keine weiteren Details in unserer Vorlsg.

Vorteile:

- der Prozess kann weiterarbeiten, Datenankunft wird per Signal-Interrupt mitgeteilt
- einfach für UDP: Signal SIGIO => 1 Datagramm ist angekommen

Nachteile:

- wenn der Prozess sonst nichts zu tun hat: „Busy Waiting“ wie bei non-blocking I/O
- sehr komplex für TCP: Signal SIGIO kann vieles bedeuten

Reales Beispiel für signal-driven I/O:
NTP-Server (Network Time Protocol): Prozess hat „viel“ zu tun, aber Datagramm benötigt bei Ankunft exakten Zeitstempel => Interrupt mit SIGIO (weitere Details für Interessierte: Stevens Ch. 25.2)

I/O-Multiplexing mit Hilfsfunktion select()

Die Hilfsfunktion `select()` stellt universelle Funktionalität für I/O-Multiplexing mehrerer Eingabequellen zur Verfügung.
Es lässt sich für `select()` sowohl ein blockierender als auch nicht-blockierender Aufruf realisieren.

Programmsourcesequenz:

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Parameter:

- `maxfdp1` die Nummer des höchsten File Descriptors plus 1 „max fd p 1“ = Anzahl der zu prüfenden File Descriptoren
- `readfds` Liste von File Descriptoren, die zum Lesen überprüft werden
- `writefds` Liste von File Descriptoren, die zum Schreiben überprüft werden
- `exceptfds` Liste von File Descriptoren, die auf Exceptions überprüft werden

„*fds“ = file descriptor set

I/O-Multiplexing mit Hilfsfunktion select()

Programmsourcesequenz:

```
#include <sys/select.h>
#include <sys/time.h>

int select(int maxfdp1, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, const struct timeval *timeout);
```

Parameter: (fortgesetzt)

- `timeout` Zeitintervall, nach dem `select()` in jedem Fall zurückkehrt

Rückgabewert:

- >0 Anzahl der Descriptoren im Zustand „ready for I/O“
- = 0 Timeout (ohne bereite Descriptoren)
- = -1 Fehler

- zu beachten: `readfds, writefds, exceptfds` werden manipuliert!
  (Parameter sind Zeiger)

Timeout-Funktion von select()

Parameter:

- `timeout` Zeitintervall, nach dem `select()` in jedem Fall zurückkehrt

Zugehörige Typdefinition:

```
struct timeval {
  long tv_sec  /* seconds */
  long tv_usec /* microseconds */

};
```

Insider-Tipp:
Aufruf von `select()` mit leeren fd-sets und **konkretem Timeout** realisiert eine wesentlich genauere Timer-Funktion als mit der System-funktion `sleep()` !

Drei Möglichkeiten der Nutzung:

1. Unendlich warten (= blockierender Aufruf), Aufruf von `select()` mit NULL-Zeiger
2. Warte gar nicht (= nicht-blockierender Aufruf), Aufruf von `select()` mit Werten sec = 0 und usec = 0
3. Warte eine spezifische Zeit, Aufruf von select() mit entsprechenden konkreten Werten sec + usec

Manipulation der File-Descriptor Mengen (fd sets)

Hilfsmakros zur Manipulation der fd-sets: (definiert in sys/select.h)

```
FD_ZERO (fd_set *set);   /* clears all bits */
FD_SET (int fd, fd_set *set); /* turns on bit fd */
FD_CLR (int fd, fd_set *set); /* turns off bit fd */
FD_ISSET(int fd, fd_set *set);    /* checks if bit fd is set */
```

Der Typ `fd_set` als Menge der File-Descriptoren wird als Bit-Flag implementiert.

Zur Vermeidung von Überraschungen unbedingt mit `FD_ZERO()` initialisieren!

Zu überprüfende File-Descriptoren entsprechend setzen.

Nach Rückkehr von select() sind die fd sets verändert:
- lese-/schreib-bereite **fd-Bits bleiben gesetzt**
- nicht-bereite fd-Bits sind auf 0 gesetzt worden

Ablaufschema für die Benutzung von select()

Dieses Ablaufschema muss für jede Benutzung von select() erneut und komplett durchgeführt werden:

1. Löschen der Menge (Bit-Flags) mit `FD_ZERO`

2. zu überprüfende Descriptoren hinzufügen mit `FD_SET`

3. eigentlicher Aufruf von `select()`

4. nach Rückkehr von `select()` Rückgabewert überprüfen (>0, 0, -1)
und ggf. `mit FD_ISSET` ein oder mehrere bereite File Descriptoren bearbeiten

Im Falle von TCP kann ein listening Server-Socket mit `select()` auch auf den Eingang von Verbindungsaufbauwünschen überprüft werden.
(danach kann der Aufruf von `accept()` erfolgen)

Wann/wie wird ein Socket File Descriptor bereit?

1. **Ein Socket wird bereit zum Lesen** (eingetragen in Menge `readfds`)

- es liegen Daten zum Lesen an (1 Datagramm UDP, x Bytes TCP)
- der Leseteil eines TCP-Sockets wurde geschlossen
  d.h. der Kommunikationspartner hat seinen Schreibteil geschlossen
  => Aufruf von `read()` wird 0 zurückliefern (vgl. vorne)
- ein TCP listening Socket hat einen komplettierten Verbindungsaufbauwunsch
  => Aufruf von `accept()` wird erfolgreich sein
- ein Socket-Error liegt vor, `read()` liefert -1, `errno` gibt weiteren Aufschluss
  (im Falle eines Socket-Errors wird readable und writeable markiert, s.u.)

Wann/wie wird ein Socket File Descriptor bereit?

2. **Ein Socket wird bereit zum Schreiben** (eingetragen in Menge `writefds`)

- der Socket Sendepuffer hat genügend Platz zum Schreiben weiterer Daten
  (aus diesem Puffer werden zu sendende Daten an den Kernel übergeben. Bei nicht-blockierendem Schreiben könnte man diesen Puffer überfluten!)
- der Schreibteil eines Sockets ist geschlossen
- nach einem nicht-blockierenden Aufruf von `connect()` wurde der Verbindungsaufbau abgeschlossen, erfolgreich oder fehlerhaft
- ein Socket-Error liegt vor, `write()` liefert -1, `errno` gibt weiteren Aufschluss
  (im Falle eines Socket-Errors wird readable und writeable markiert, s.u.)

3. **Beim Socket liegt eine Ausnahmebedingung vor**, Exception
   (eingetragen in Menge `exceptionfds`)

Selten benutzer Fall:

- am TCP-Socket liegen sog. „Out-of-band“ Daten an
  (wichtige Daten, die „normale“ Daten überholen sollen)
  (mehr Informationen für Interessierte im Stevens, Ch. 24)

Beispiel für select()

```
int res, fd1=5, fd2=8;
char buf[1024];
fd_set rset;

while (1) {
    FD_ZERO(&rset);
    FD_SET(fd1, &rset);
    FD_SET(fd2, &rset);

    res = select(9, &rset, NULL, NULL, NULL);
    if (res<=0) { ... }
    if (FD_ISSET(fd1,&rset)) {
        bzero(buf,1024);
        res = read(fd1,buf,1024);
        if (res<0) { ... }
        if (res==0) printf("Received EOF on fd1!\n");
        else printf("Received data on fd1: %s\n", buf);
    }
    if (FD_ISSET(fd2,&rset)) { ... }
}
```

File-Descriptoren hier manuell gesetzt.

Jedes Mal fd-sets neu initialisieren!

Blockierender Aufruf von select(), nur Abfrage auf lesebereite Sockets.

Error von select()

Error von read()

Jeder mögliche Socket muss überprüft werden.

Source

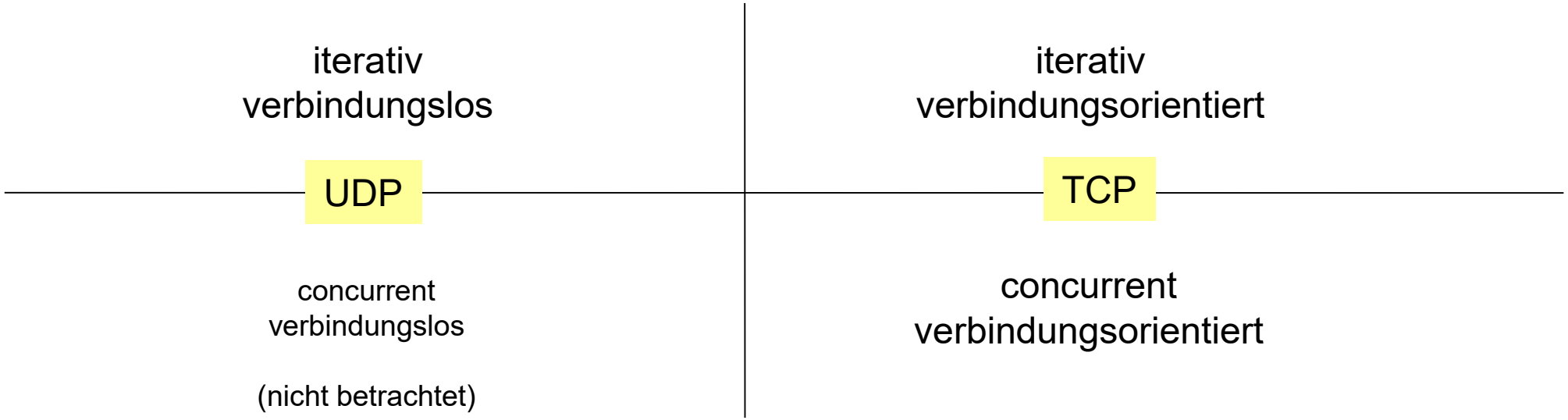
3.4. Server-Strukturen

Motivation:

Der Programmierer soll einen Kommunikations-Server entwerfen und programmieren, der in der Lage ist, „viele“ (… tausende!) Anfragen entgegen zu nehmen!

Hierzu sind **verschiedene Eigenschaften** auszuwählen und zu realisieren:

- iterativ (hintereinander) vs. concurrent (gleichzeitig, nebenläufig)
- verbindungsorientiert (TCP) vs. verbindungslos (UDP)



Iterativ vs. nebenläufig

- ein **iterativer Server** bearbeitet zu einem Zeitpunkt genau eine Client-Anfrage
  d.h. alle Client-Anfragen werden hintereinander bearbeitet

- ein **nebenläufiger (concurrent) Server** kann mehrere Client-Anfragen „gleichzeitig“ bearbeiten

Nebenläufig (concurrent)

- typisch für längere Anfragen (mehrere Datenpakete) oder Anfragen mit variabler Länge

- „schwieriger“ zu programmieren

- benötigt typischerweise mehr System-ressourcen (Sockets mit Buffer, Prozesse, Threads, …)

Iterativ

- typisch für kurze Anfragen bzw. Anfragen mit fester Länge

- **einfach zu programmieren**

Verbindungslos vs. verbindungsorientiert

Verbindungslos

- weniger Overhead
  (Verbindungsauf- u. Abbau,
  Verwaltung von Sockets, …)
- keine Begrenzung der Anzahl
  von Clients

Verbindungsorientiert

- **einfach zu programmieren**
- das Transportprotokoll (TCP) kümmert
  sich um das „Schwierige“
  (zuverlässige Übertragung von
  Daten-Strömen)
- benötigt aber getrennte Sockets für
  jede aktive Verbindung
  (d.h. Anzahl ist „begrenzt“)

→ Wähle

Iterativ (weil **einfach zu programmieren**)
+
Verbindungsorientiert (weil **einfach zu programmieren**)

???

Die einfachste Lösung
ist nicht automatisch
die beste Lösung!

Statelessness vs. Statefullness

- State = Zustand: Information, die ein Server über den Status einer aktuellen Client-Interaktion aufrecht erhält

- Statelessness = Zustandslosigkeit, der Server merkt sich keinerlei Information über Client-Interaktionen (z.B. typisch bei iterativ und verbindungslos)

- Statefullness = Zustandsbehaftung, der Server merkt sich den Zustand der Interaktionen mit einem Client (z.B. einfach realisierbar bei verbindungsorientierten Servern für die Dauer einer Verbindung)

Aber:
Verbindungslose Server mit Verwaltung von Statusinformationen (Statefullness) müssen besonders vorsichtig + sorgfältig konzipiert werden!

- ein Client kann jederzeit ausfallen
- ein Client kann jederzeit neu starten (also mehrmals aus Sicht des Servers)
- Nachrichten im Netz können verloren gehen
- Nachrichten im Netz können dupliziert werden

Design-Alternativen für Server

Übersicht über verschiedene Ansätze:

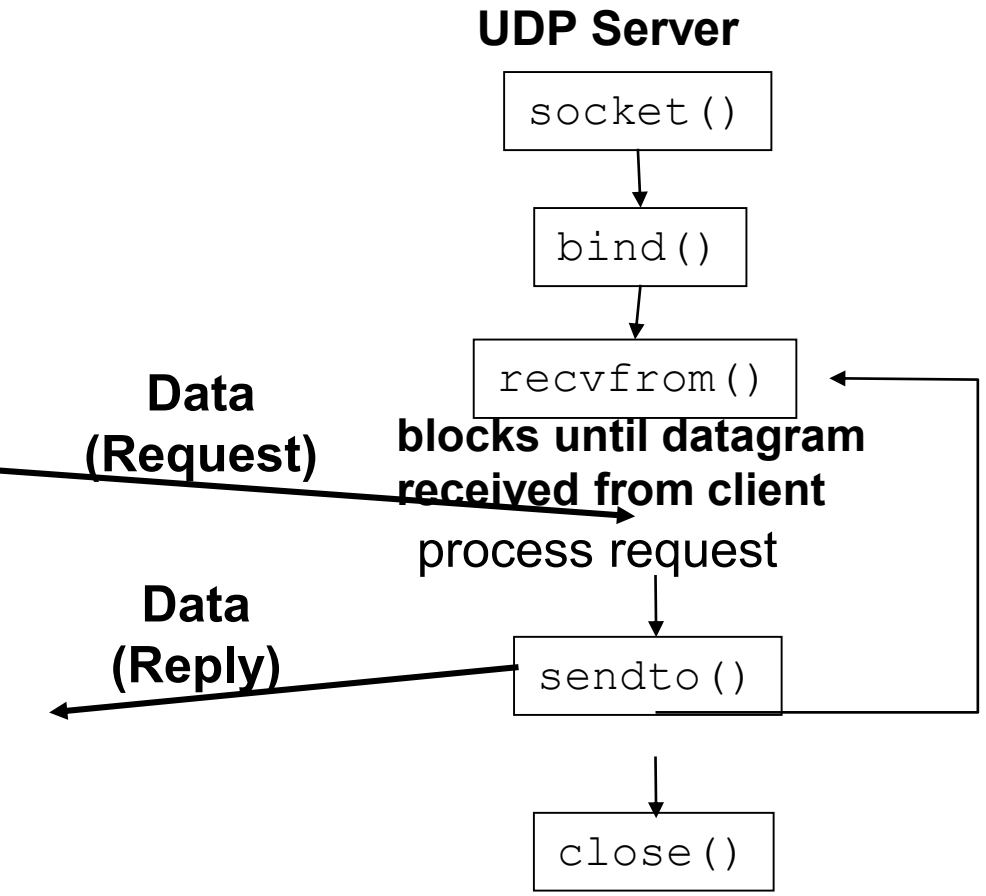
- Iterativ, verbindungslos oder verbindungsorientiert

Verbindungsorientiert:

- Je ein Kind-Prozess pro Client (statt Prozessen auch Threads)
- Prefork (auch Prethreaded), Prozesse/Threads vorbereitet vor Client-Anfrage
- Select Loop
- … weitere Varianten

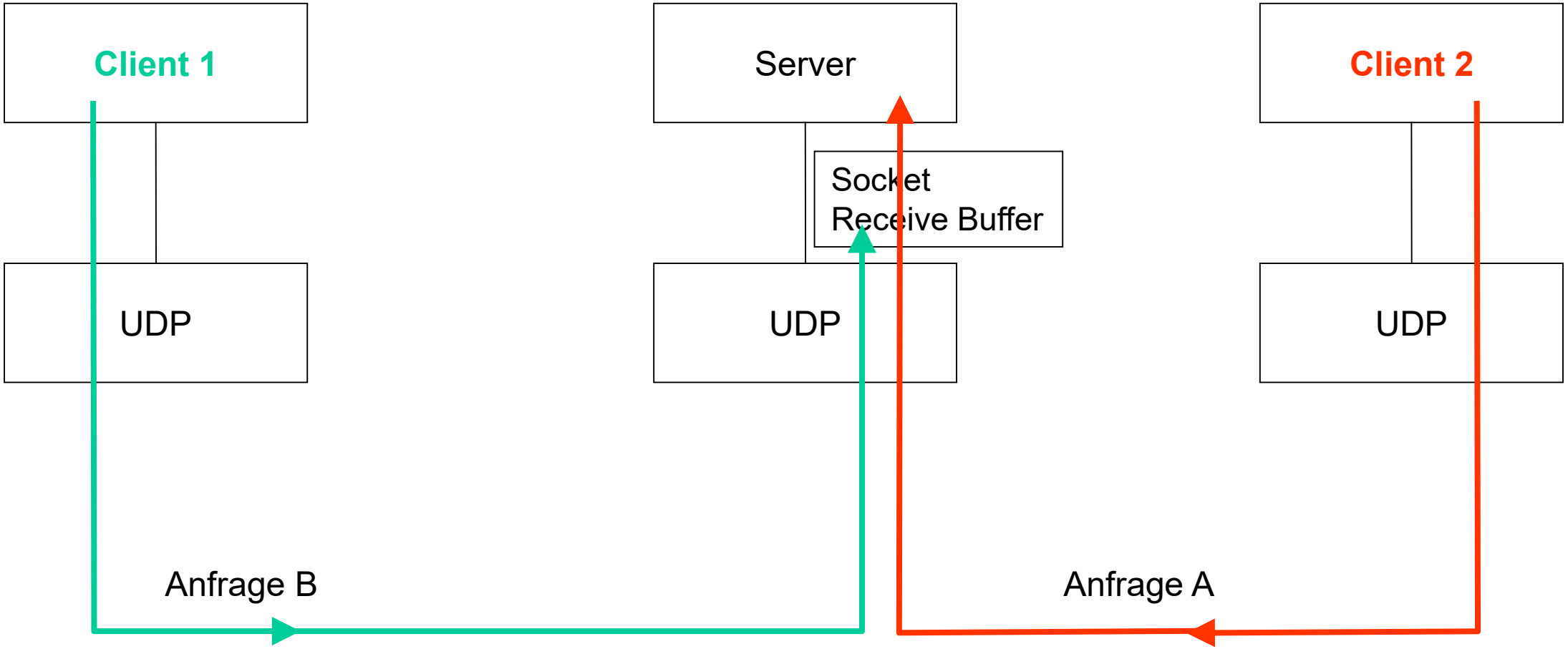
UDP Server, iterativ

Typischer Beispielablauf vom UDP-Server (vgl. Folie 61)



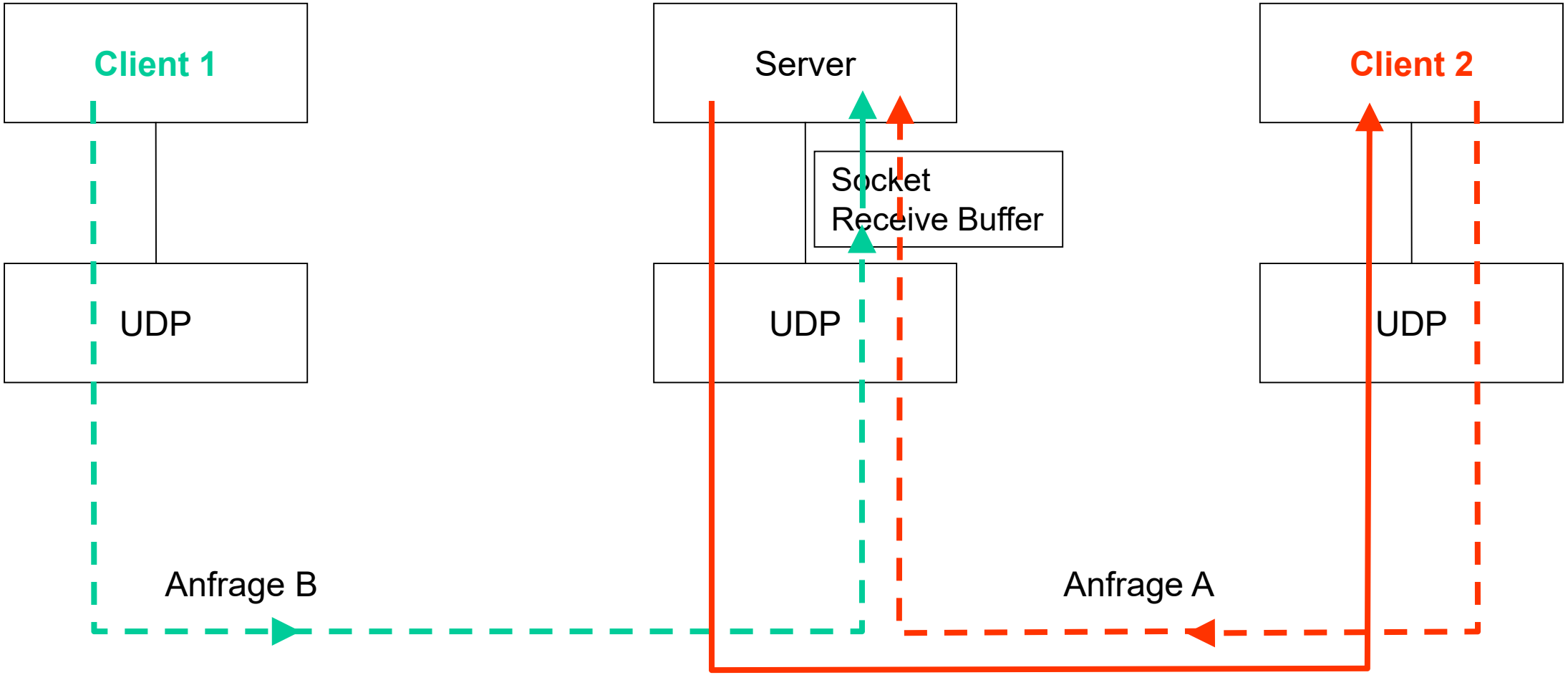
1. Blockieren bis Client-Anfrage empfangen (Client-Adresse aus recvfrom())
2. Verarbeiten der Anfrage
3. Senden einer Antwort an die entsprechende Client-Adresse
4. Weiter bei 1., warte auf nächste Anfrage

Iterativer UDP-Server mit 2 Clients (1)



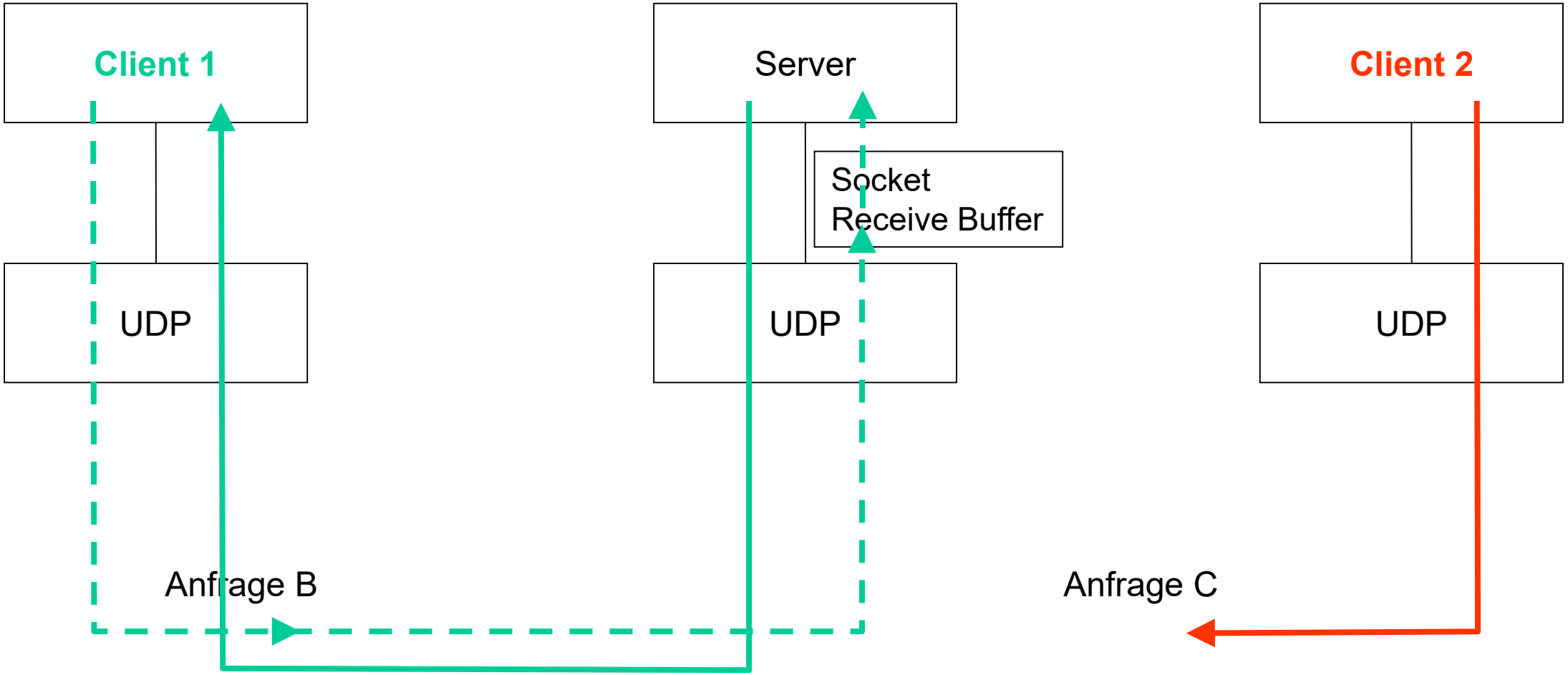
- Anfrage A erreicht Socket Receive Buffer und Server-Prozess
- Anfrage B erreicht Socket Receive Buffer

Iterativer UDP-Server mit 2 Clients (2)



- Server bearbeitet Anfrage A
- Server sendet Antwort an Client 2
- Anfrage B erreicht Server aus Socket Receive Buffer

Iterativer UDP-Server mit 2 Clients (3)



- Server bearbeitet Anfrage B
- Server sendet Antwort an Client 1

Iterativer UDP-Server: Fazit

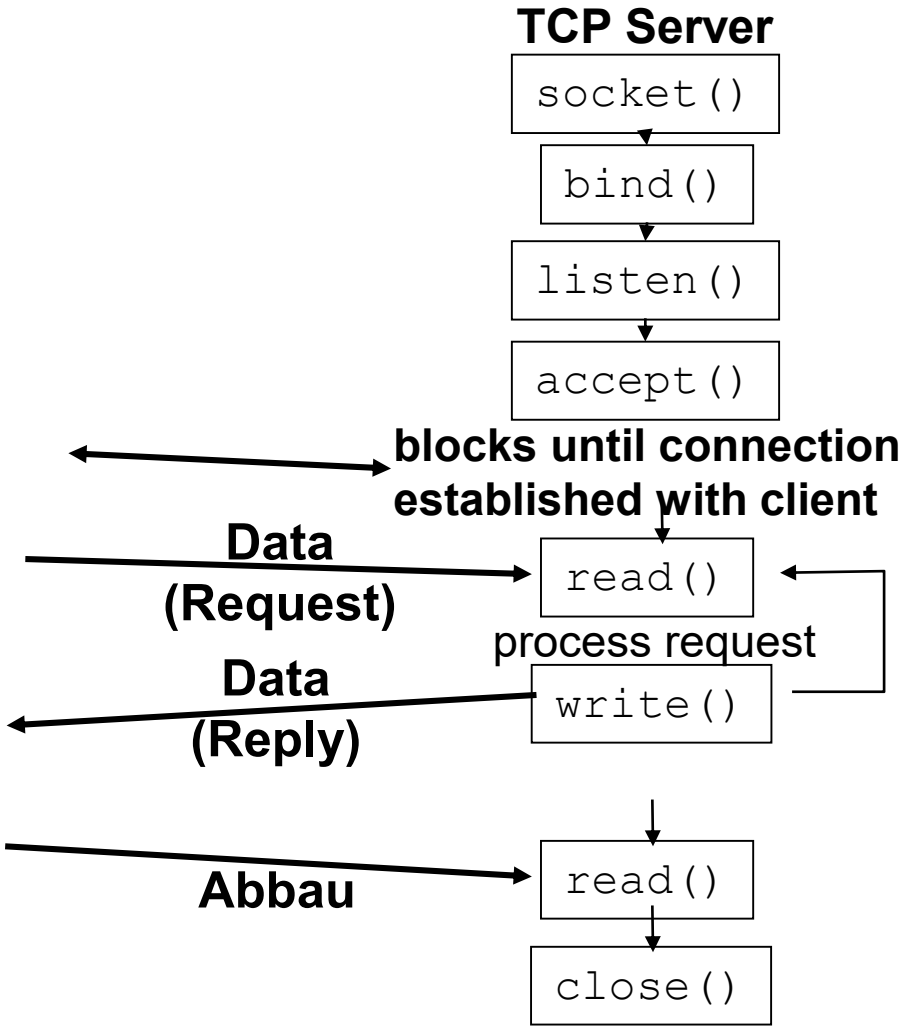
- Der Server bearbeitet eine Anfrage und ist direkt danach bereit für die nächste Anfrage (iterativ).
- Die Client-Adresse einer Anfrage wird direkt von der Anfrage in die Antwort übernommen, typischerweise wird keine Zustandsinformation gespeichert
  (d.h. aufeinander folgende Anfragen werden unabhängig beantwortet).
- In dieser Hinsicht ist der Server unabhängig von der Anzahl Clients (diese ist somit unbegrenzt).
- Während der Bearbeitung ankommende Anfragen werden im Socket Receive Buffer zwischengespeichert.

Achtung!
Bei langer Bearbeitungszeit kann der Socket Receive Buffer nur eine begrenzte Zahl von Anfragen zwischenpuffern (=> Anfragen können verloren gehen!)

„Lange“ Anfragen (größer als ein UDP/IP-Datagramm) können nicht von diesem Server-Typ unterstützt werden!

Iterativer TCP-Server

Typischer Beispielablauf vom TCP-Server (vgl. Folie 69)



„**TCP server bereitet sich vor**“
Socket im „listen“ Modus

accept() „**TCP server wartet und akzeptiert Verbindung**“

Typisch:
Zyklus aus Empfangen, Verarbeiten, Beantworten

Typisch:
Server erfährt vom Abbauwunsch des Clients, dann close()

accept() liefert neuen Socket-Descriptor für die gerade aufgebaute Verbindung.

Der Descriptor des listening Sockets bleibt erhalten, weiterhin im „listen“ Modus.

Der Descriptor der aktuellen Verbindung wird geschlossen.

Iterativer TCP-Server

Beispielprogrammsequenz für einen iterativen TCP-Server:

```
int sock_listen_fd, newfd;

…          /* sock_listen_fd vorbereitet */

while (1) {
  newfd = accept(sock_listen_fd, ...);

  treat_request(newfd);
  close(newfd);
}
```

Listening Socket „wie üblich“ vorbereitet.

accept() kehrt zurück, nachdem neue Verbindung aufgebaut ist.

In treat_request() wird die komplette Anfrage/TCP-Verbindung bis zum Abbauwunsch des Clients bearbeitet.

Der iterative TCP-Server funktioniert analog zum iterativen UDP-Server:

- ein Client-Verbindungsaufbauwunsch wird mit `accept()` angenommen
- diese Client-Verbindung wird komplett bearbeitet (bis Abbauwunsch)
- diese Verbindung wird geschlossen
  → Iteration: nächster Client-Verbindungsaufbauwunsch kann angenommen werden

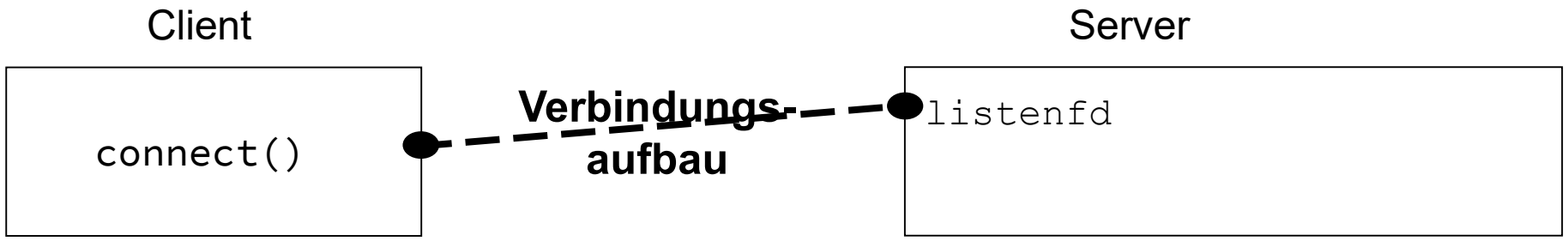
Iterativer TCP-Server: Fazit

- Dieser Server ist **einfach zu programmieren**.
- Die Anzahl der aktiven Verbindungen ist auf 1 begrenzt (wg. Iteration), die Anzahl der „wartenden“ Verbindungen ist ebenfalls begrenzt (vgl. backlog von `listen()`, Folie 72 ff.).
- Bei langer Bearbeitungszeit einer Anfrage müssen ggf. Verbindungsaufbauwünsche abgewiesen werden.
- Möglicherweise niedrige Auslastung der Server-Ressourcen (CPU, Platten), wenn `treat_request(newfd)` den Server allein nicht voll auslastet.
- Die Warte- und Bearbeitungszeit von Anfragen erhöht sich dadurch.

→ insbesondere aufgrund der Ressourcen-Auslastung ist die nebenläufige Bearbeitung von Client-Anfragen/TCP-Verbindungen sinnvoll.

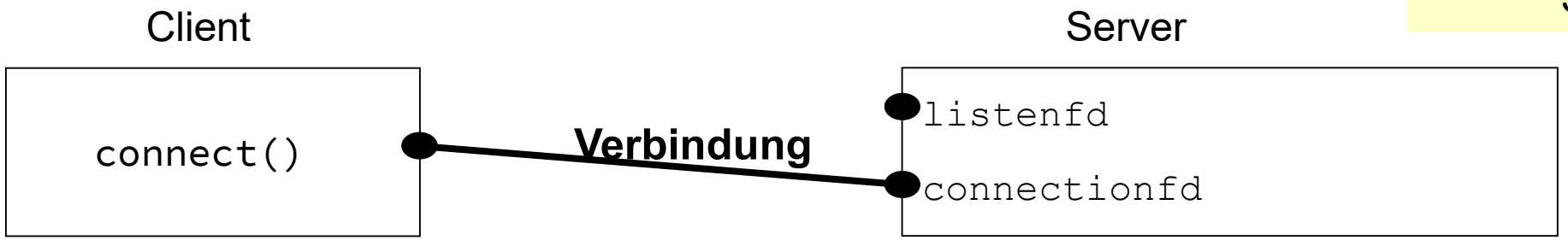
Nebenläufiger Server mit TCP mit fork()

Beim nebenläufigen Server mit TCP und `fork()` wird für jede Verbindung zu einer Client-Anfrage ein neuer Prozess (sog. Kind-Prozess bzw. Child-Process) generiert, der sich um die Bearbeitung kümmert.



1. der Client initiiert `connect()`, der Server wartet auf Rückkehr von `accept()`

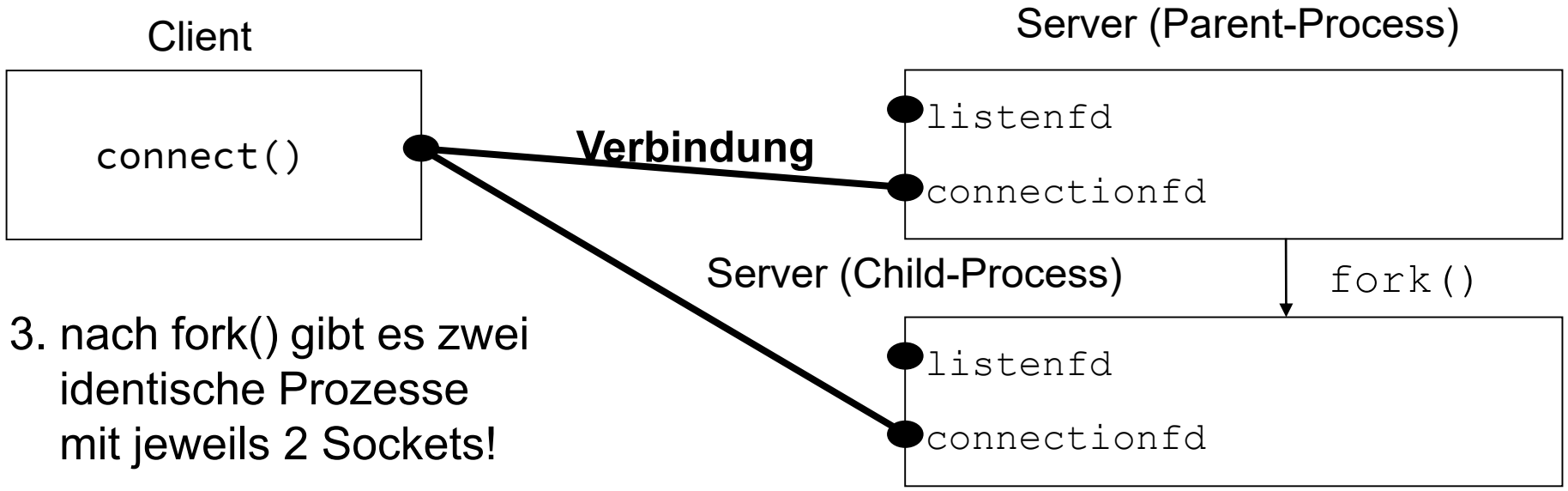
Vgl. Kap 2.1. Prozesse SysProg



2. nach Rückkehr von `accept()` läuft die Verbindung auf dem neuen Socket

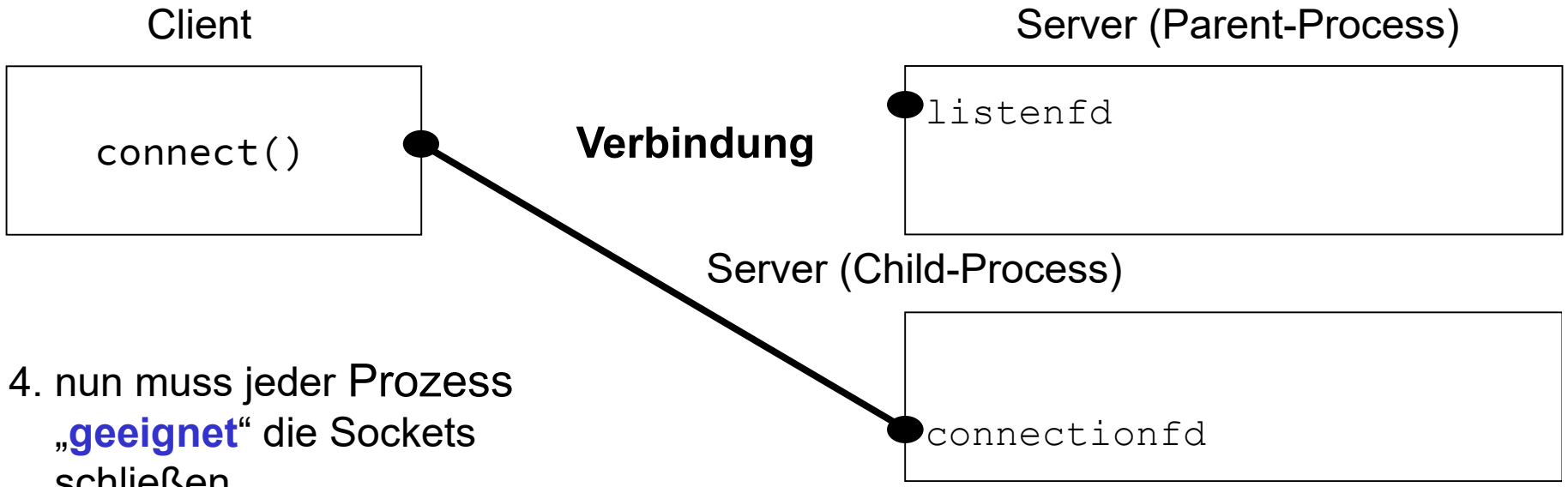
Bis hierher ist alles identisch zum iterativen TCP-Server!
Der Server wäre nun mit dem Socket `connectionfd` beschäftigt und kann keine weiteren eingehenden Verb.aufbauwünsche an `listenfd` entgegen nehmen!

Nebenläufiger Server mit TCP mit fork()



3. nach fork() gibt es zwei identische Prozesse mit jeweils 2 Sockets!

Aufruf von `fork()` erzeugt eine identische Kopie des aktuellen Prozesses!



4. nun muss jeder Prozess „**geeignet**“ die Sockets schließen

Der Eltern-Prozess schließt `connectionfd` und ist bereit für neue Client-Verb.-Anfragen.

Der Kind-Prozess schließt `listenfd` und bearbeitet nur noch die Verbindung.

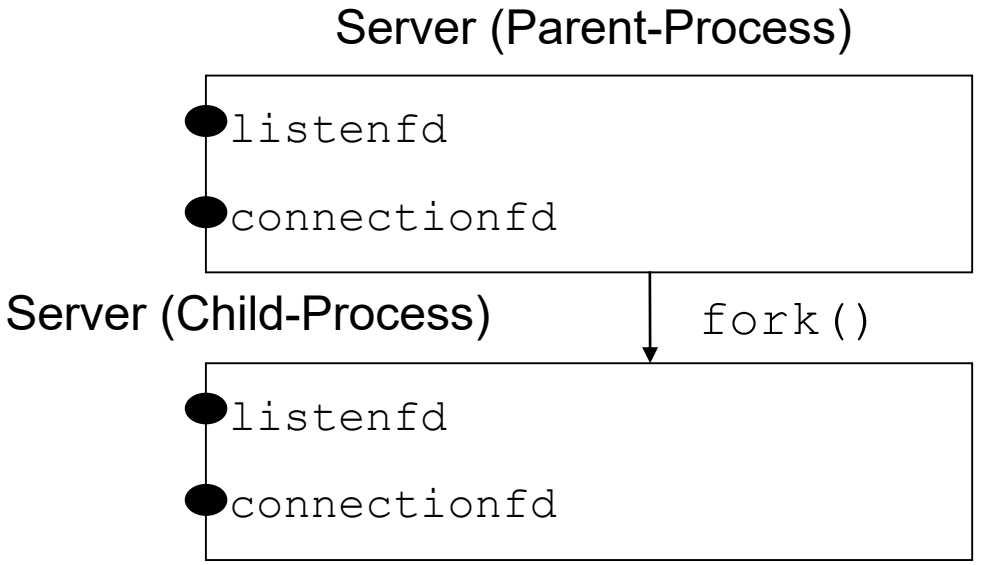
Systemfunktion fork()

Programmsourcesequenz:

```
#include <unistd.h>

pid_t fork(void);
```

Parameter:

- keine



Aufruf von `fork()` erzeugt eine identische Kopie des aktuellen Prozesses!

Rückgabewert:

- = 0 für den Child-Process
- >0 Prozess-ID des Child-Process (für den Parent-Process)
- -1 bei Fehler

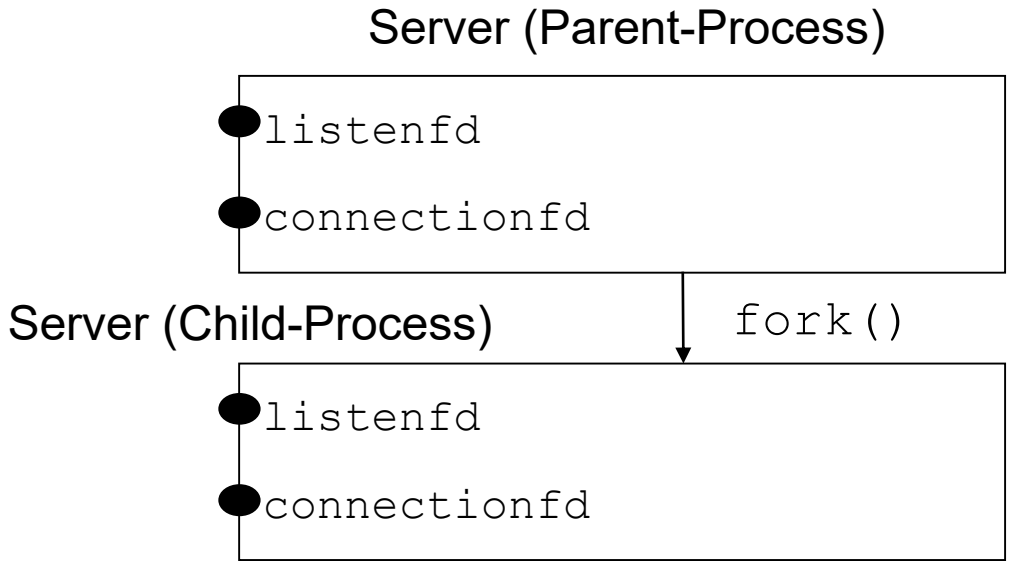
Zu beachten:

- `fork()` wird **einmal im Parent-Process** aufgerufen
- `fork()` kehrt **zweimal zurück**, jeweils **im Parent-** sowie **im Child-Prozess** und der jeweilige Prozess läuft im Programmcode hinter `fork()` weiter
- Parent sowie Child werden durch den Rückgabewert unterschieden

Prozess-IDs mit fork()

Rückgabewert:

- = 0 für den Child-Process
- >0 Prozess-ID des Child-Process (für den Parent-Process)
- -1 bei Fehler



Child-Prozess:

- erkennt sich selbst an Rückgabewert 0 von `fork()`
- kann Prozess-ID des Parent mit `getppid()` abfragen
- kann eigene Prozess-ID mit `getpid()` abfragen

Parent-Prozess:

- erkennt sich selbst an Rückgabewert >0 von `fork()`
- Rückgabewert >0 ist Prozess-ID des Child, diese muss bei Bedarf gemerkt werden
- kann eigene Prozess-ID mit `getpid()` abfragen

Beispielablauf mit fork()

```
int main() {
  int sock_listen_fd, newfd, child_pid;

…          /* sock_listen_fd vorbereitet */

  while (1) {
  newfd = accept(sock_listen_fd, ...);

  if (newfd<0)      /* Fehlerbehandlung */

  child_pid = fork();

  if (child_pid==0) {
  close(sock_listen_fd);
  treat_request(newfd);
  close(newfd);
  exit(0);
    }
  else { close(newfd); }
  }
}
```

Listening Socket „wie üblich“ vorbereitet.

accept() kehrt zurück, nachdem neue Verbindung aufgebaut ist.

Aufruf von fork(), danach **zwei** Prozesse !!!

Child schließt listening Socket.

In treat_request() wird die komplette Anfrage/TCP-Verbindung bis zum Abbauwunsch des Clients bearbeitet.

Child schließt Connection-Socket und beendet sich selbst!

Parent schließt Connection-Socket und ist danach sofort wieder emfpangsbereit auf listening Socket.

Abfrage `child_pid == -1` für Fehler fehlt!

Source

Bemerkungen zum Beispielablauf

- Nach Rückkehr von `fork()` existieren je zwei Descriptoren auf die Sockets `sock_listen_fd` und `newfd` (vgl. Folie 132 ff.)

- Der jeweilige Aufruf von `close()` in Child und Parent dekrementiert die Anzahl aktiver Descriptoren und schließt somit noch nicht den Socket (vgl. Folie 82)

- Nach dem letzten `close()` und `exit(0)` im Child bleibt der Child-Prozess noch bestehen (sog. „Zombie“). Dies muss der Parent-Prozess „geeignet“ aufräumen!

- Was passiert, wenn der Child-Prozess in `treat_request(newfd)` eine globale Variable verändern möchte??? (z.B. ein globaler Request-Counter, der die jeweils aktuelle Anzahl nebenläufig bearbeiteter Anfragen beinhalten soll???)

- Systemaufruf von `fork()` benötigt „viel“ Zeit/Ressourcen im Betriebssystem
- die Anzahl der gleichzeitigen Anfragen lässt sich nicht (einfach) begrenzen

Alternativen:

- verwende Threads anstelle von „teuren“ Prozess-Kopien mit `fork()`
- verwende sog. „Preforked Server“

Preforked Server

Eigenschaften:

- grundsätzlich gleiches Prinzip wie bei `fork()`, d.h. eine Client-Anfrage und die dazugehörige Verbindung wird von einem eigenen Prozess bearbeitet
- Unterschied: der „teure“ Aufruf von `fork()` wird vom Parent-Server-Prozess bereits mehrmals vorab ausgeführt
- die Anzahl der Child-Prozesse (somit die Anzahl möglicher nebenläufiger Client-Anfragen) kann dadurch begrenzt werden
- jeder Child-Prozess ruft `accept()` auf, der nächste eingehende Verbindungsaufbauwunsch eines Clients wird dann vom BS einem der Prozesse zugewiesen

Beispielablauf Preforked Server

```
#define NB_PROC 10

void recv_requests(int fd) { /* An iterative server */
  int f;

  while (1) {
    f=accept(fd,...);
    treat_request(f);
    close(f);
  }
}


int main() {
  int fd;
…          /* Listening Socket fd vorbereitet */

  for (int i=0;i<NB_PROC;i++) { /* Create NB_PROC children */
    if (fork()==0) recv_requests(fd);
  }
  while (1) pause(); /* The parent process does nothing */
}
```

Anzahl der Preforked Children (fest programmiert)

accept() kehrt zurück, nachdem neue Verbindung aufgebaut ist.

In treat_request() wird die komplette Anfrage/TCP-Verbindung bis zum Abbauwunsch des Clients bearbeitet.

Child schließt Connection-Socket und ist bereit für neue Verbindung

Parent generiert 10 Child-Prozesse

Child-Prozess ruft jeweils `recv_requests(fd)` auf

Parent macht danach gar nichts mehr!

Bemerkungen zum Preforked Server

- alle Child-Prozesse rufen „gleichzeitig“ die Systemfunktion `accept()` auf

- in manchen Betriebssystemen läuft dieser gleichzeitige (blockierende) Zugriff **nicht**, dann müssen entsprechende Synchronisationsmechanismen eingesetzt werden (z.B. Lock/Unlock Mutex)

- Wie groß sollte die Anzahl der Preforked Children sein???
  Kompromiss aus Gewinn durch vorgezogenen Aufruf von `fork()` und ggf. Anzahl unnötig erzeugter Child-Prozesse

Alternativen:
- realisiere die Anzahl der Preforked Children dynamisch (d.h. ggf. werden spätere weitere Child-Prozesse generiert oder wieder beendet)
- auch hier ist der Einsatz von Threads anstelle von Prozess-Kopien denkbar

Anmerkung:
Der weitverbreitete **Apache Web Server** (ab Version 2.0) benutzt Preforking.
(„This Multi-Processing Module (MPM) implements a non-threaded, pre-forking web server …“)
Mehr für (freiwillig) Interessierte: http://httpd.apache.org/docs-2.0/mod/prefork.html
(URL getestet 12.01.2024)

Ein „Exot“: Select Loop

Ein einziger Prozess könnte auch alle Sockets (Listening und n x Connection) mit Hilfe von `select()` bedienen!

- extrem schwierig, das ganze korrekt zu implementieren!
- eine Client-Anfrage (u.U. länger dauernde TCP-Verbindung mit mehreren `read()` `write()` -Zyklen) muss korrekt verarbeitet werden
- der Prozess muss komplexe Datenstrukturen für alle nebenläufigen Anfragen verwalten (Status, „lokale“ Daten, …)

Anmerkung:
Die „**Squid WWW Cache** Application“ benutzt das Konzept des Select Loop!
(„Squid is implemented as a single, non-blocking process based around a BSD select() loop.”)

Mehr für (freiwillig) Interessierte:
http://www.squid-cache.org/Doc/code
(URL getestet 12.01.2024)

Ein „Exot“: Select Loop

Ein einziger Prozess könnte auch alle Sockets (Listening und n x Connection) mit Hilfe von `select()` bedienen!

Weiteres Beispiel: Node.js

Aus: The Hacker's Guide to Scaling Python
Von: Julien Danjou
2017

If you combine multiple sources of events in a `select` call, it is easy to see how your program can become *event-driven*. The `select` loop becomes the main control flow of the program, and everything revolves around it. As soon as some file descriptor or socket is available for reading or writing, it is possible to continue operating on it.

This kind of mechanism is at the heart of any program that wants to handle, for example, thousands of connections at once. It is the base technology leveraged by tools such as really fast HTTP servers like NGINX or Node.js.

`select` is an old but generic system call, and it is not the most well performing out there. Different operating systems implement various alternative and optimizations, such as `epoll` in Linux or `kqueue` in FreeBSD. As Python is a high-level language, it implements and provides an abstraction layer known as *asyncio*.

Zusammenfassung: Server Design Choices

Welche ist die beste Server-Struktur für meine Anwendung ???

Antwort abhängig von versch. Faktoren:

- erwartete Anzahl gleichzeitiger Client-Anfragen
- Größe der Anfrage (=> benötigte Zeit für die Bearbeitung/ggf. „Nachschlagen“)
- Schwankungen der Anfrage-Größe
- Verfügbare System-Ressourcen (Hauptspeicher, # CPUs, CPU-Leistung, Festplatte, …)

A: Iterativ, UDP
B: Iterativ, TCP

C: fork() mit UDP (nicht besprochen)
D: fork() mit TCP

E: Threads statt Prozess-fork()
F: Preforked/Prethreaded

G: Select Loop

Es ist wichtig, diese Aspekte und deren Optionen bei der Anwendungsentwicklung zu kennen und zu verstehen!

U.U. kann ein Vergleichstest verschiedener Alternativen helfen, die beste Lösung zu finden!

3.5. Zusammenfassung

- Wie funktioniert das Internet?
- Adressierung im Internet mit IP-Adressen und Portnummern (und Protokoll TCP oder UDP)
- IP ist ein unzuverlässiges Netzwerkprotokoll: Adressierung und Routing
- UDP ist ein unzuverlässiges Transportprotokoll (nur in den Endgeräten)
- TCP ist ein zuverlässiges Transportprotokoll (nur in den Endgeräten)

- mit Sockets kann der Programmierer auf TCP oder UDP zugreifen
- abhängig von TCP oder UDP werden unterschiedliche Systemfunktionen genutzt
  TCP: socket(), bind(), listen(), accept(), connect(), read(), write(), close(), shutdown()
  UDP: socket(), bind(), recvfrom(), sendto(), close()
- die sog. Host Byte Order kann unterschiedlich zur Network Byte Order sein!

- im Normalfall blockiert eine Lesefunktion auf einem Socket
- es gibt die Variante des nicht-blockierenden Aufrufs derartiger Funktionen
- Variante: signal-gesteuerter Input/Output
- die select()-Funktion stellt die komfortabelste Hilfe beim I/O-Multiplexing dar

- Beim Design/Programmieren eines Servers sind diverse Entscheidungen zu fällen: iterativ/nebenläufig, TCP/UDP, stateless/statefull
- verschiedene Realisierungen: fork() mit Prozessen, Verwendung von Threads, Preforked/Prethreaded Server, Select Loop