Exercise 3: Sockets

Operating Systems UE
2022W

David Lung, Florian Mihola, Andreas Brandstätter,
Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-physical Systems

2022-12-06

Overview

Inter-process communication

Considered so far. . .

Exchanging data between processes on the same system

- Explicit synchronization between unrelated processes
  - Shared Memory
  - Semaphores
- Implicit synchronization between related processes
  - Blocking read- and write operations
  - Non-related processes via sockets
  - Related processes via unnamed pipes

Today. . .

Exchanging data via sockets - either on the same system or over a network

- Implicit synchronization between unrelated processes

Byte Order or Endianness

- Sequential ordering of bytes in memory

```
int i = 0x12345678; // 8 hex digits = 4 bytes
```

  - Little endian: little end first = least significant byte first

Byte address	&i	&i + 1	&i + 2	&i + 3
Byte content	0x78	0x56	0x34	0x12

  - Big endian: big end first = most significant byte first

Byte address	&i	&i + 1	&i + 2	&i + 3
Byte content	0x12	0x34	0x56	0x78

- Byte order in memory depends on processor architecture (x86 is little endian)
- When writing multiple bytes, program must take care of byte order
- Network byte order is big endian

Byte Order or Endianness

Write bytes explicitely in little endian order:

```
int i = 0x12345678;
uint8_t buf[sizeof(int)];

int pos;
for (pos = 0; pos < sizeof(int); pos++)
    buf[pos] = i >> 8 * pos;

fwrite(buf, sizeof(int), 1, out);
```

Read bytes explicitely in little endian order:

```
uint8_t buf[sizeof(int)];
fread(buf, sizeof(int), 1, in);

int i = 0;
int pos;
for (pos = 0; pos < sizeof(int); pos++)
    i |= (int)buf[pos] << 8 * pos;

// i == 0x12345678
```

Byte Order or Endianness

uint32_t **htonl**(uint32_t netlong)

- Convert a 32-bit from host byte order to network byte order

uint32_t **ntohl**(uint32_t netlong)

- Convert a 32-bit integer from network byte order to host byte order

Sockets

- What is a socket?
  - Method for interprocess communication (IPC)
  - Either on a single host or between different hosts in a network (or via internet)
- Common scenario: communication between a client and a server
- Sockets are handled like files
  - Each socket gets a file descriptor
  - Reading and writing to the associated file descriptor

Socket API

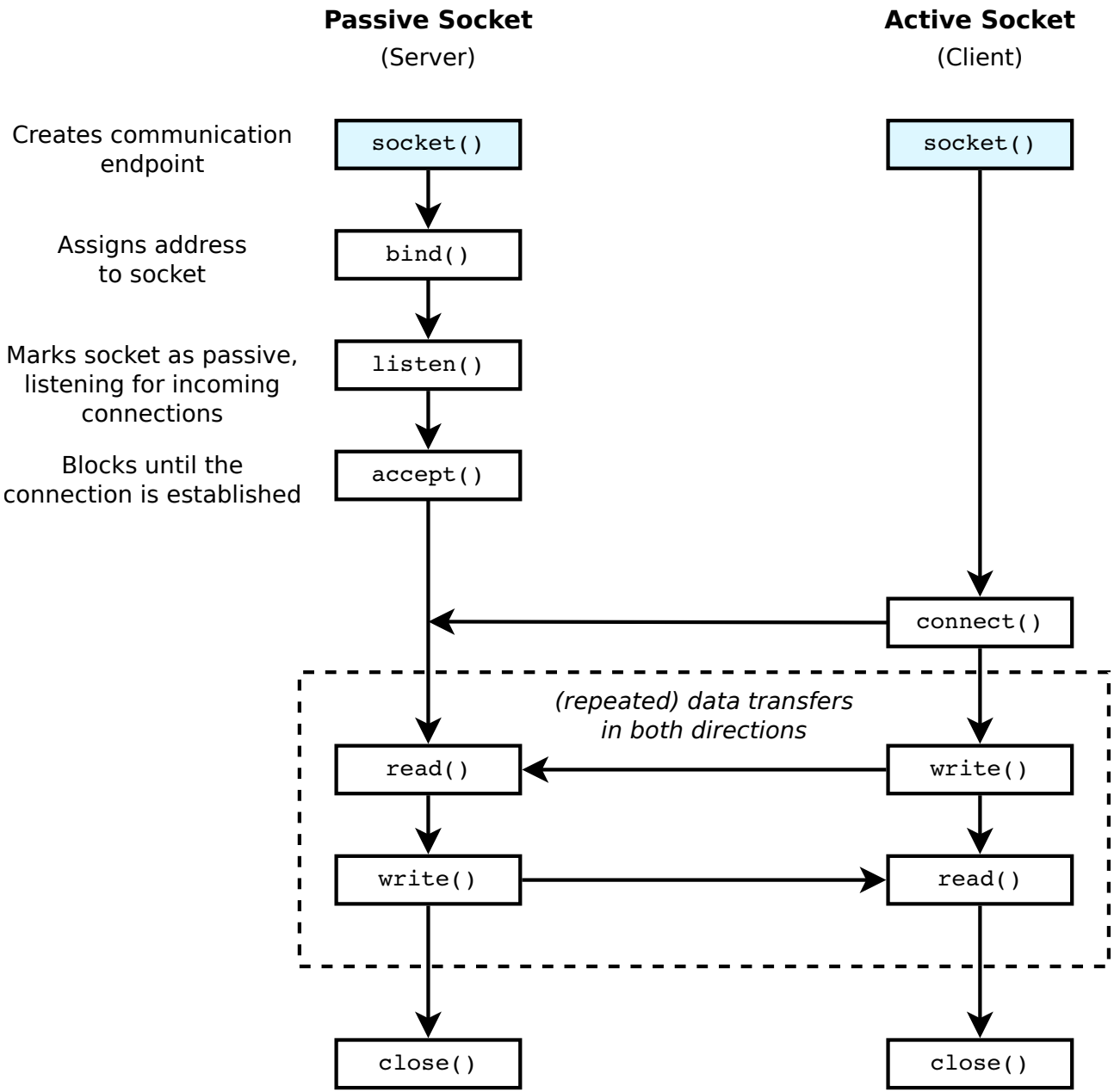
- Sockets are an interface to the transport layer of a communication protocol
  - Direct communication between client and server: no need to know the network layout
  - Sockets do not implement application protocols (HTTP, FTP, ...)
- Connection-oriented, bidirectional and reliable communication channel
- The connection is established between two endpoints
  - Endpoint on server side: Server IP + known port number
  - Endpoint on client side: Client IP + unused port number

Sockets

Address families and socket types

- Address family (network layer)
  - Internet Protocol, version 4 (IPv4)
    AF_INET → man 7 ip
  - Internet Protocol, version 6 (IPv6)
    AF_INET6 (IPv6) → man 7 ipv6
  - Unix Domain Sockets (local IPC)
    AF_UNIX → man 7 unix
- Socket type
  - Connection-oriented sockets (stream based)
    - SOCK_STREAM, default for IP is TCP
    - Connection is identified by two endpoints
  - Connection-less sockets (datagram/message based)
    - SOCK_DGRAM, default for IP is UDP

Client-Server Example



System Call: socket()

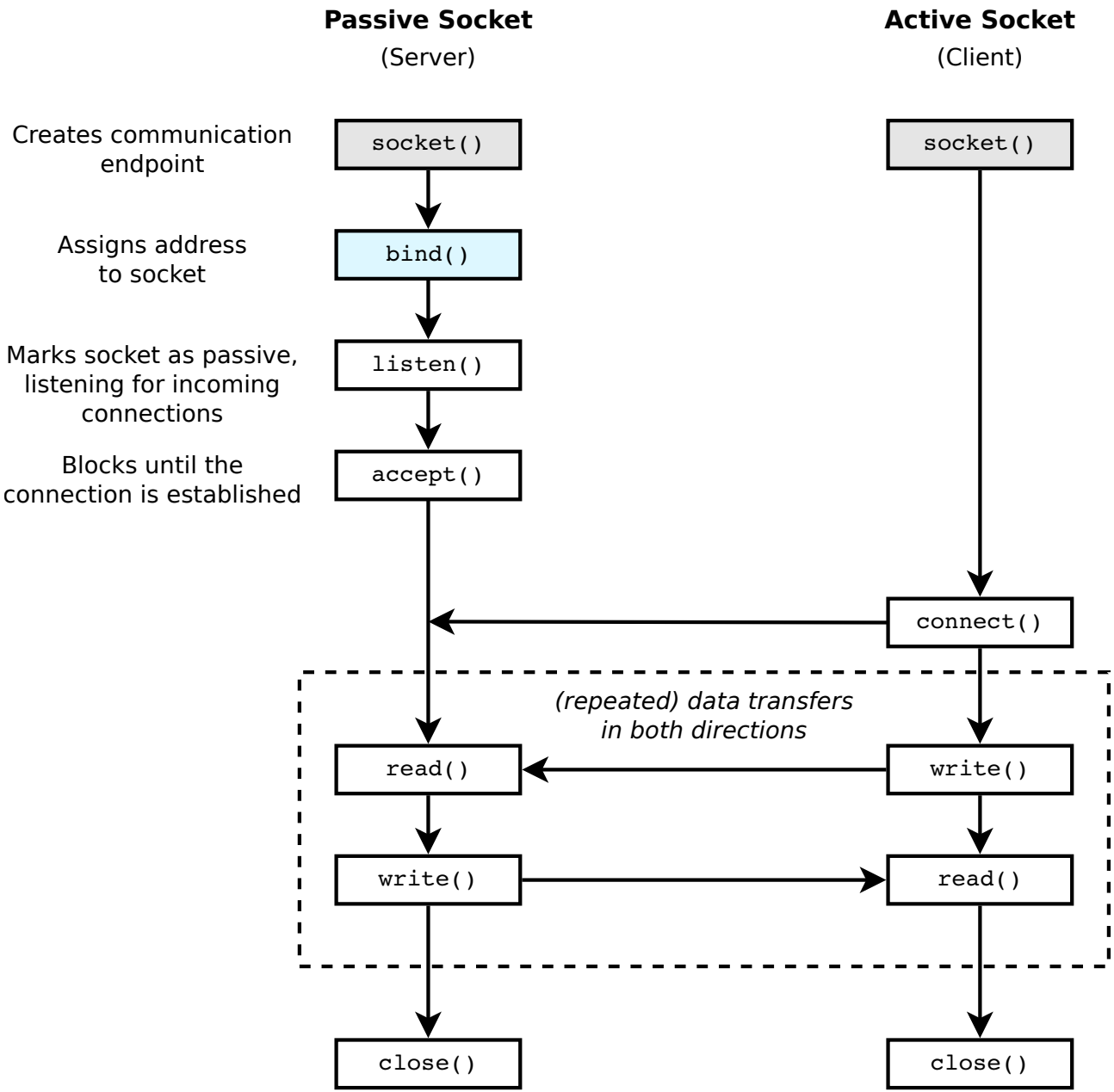
int **socket**(int family, int type, int protocol)

- Creates a communication endpoint (socket)

  family address family
  type socket type
  protocol communication protocol to be used
  - address family + type usually imply protocol
  - 0 for default-protocol
- Return value: File descriptor of the newly created socket or -1 on failure ($\rightarrow$ errno)

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

if (sockfd < 0)
    // error
```

Client-Server Example



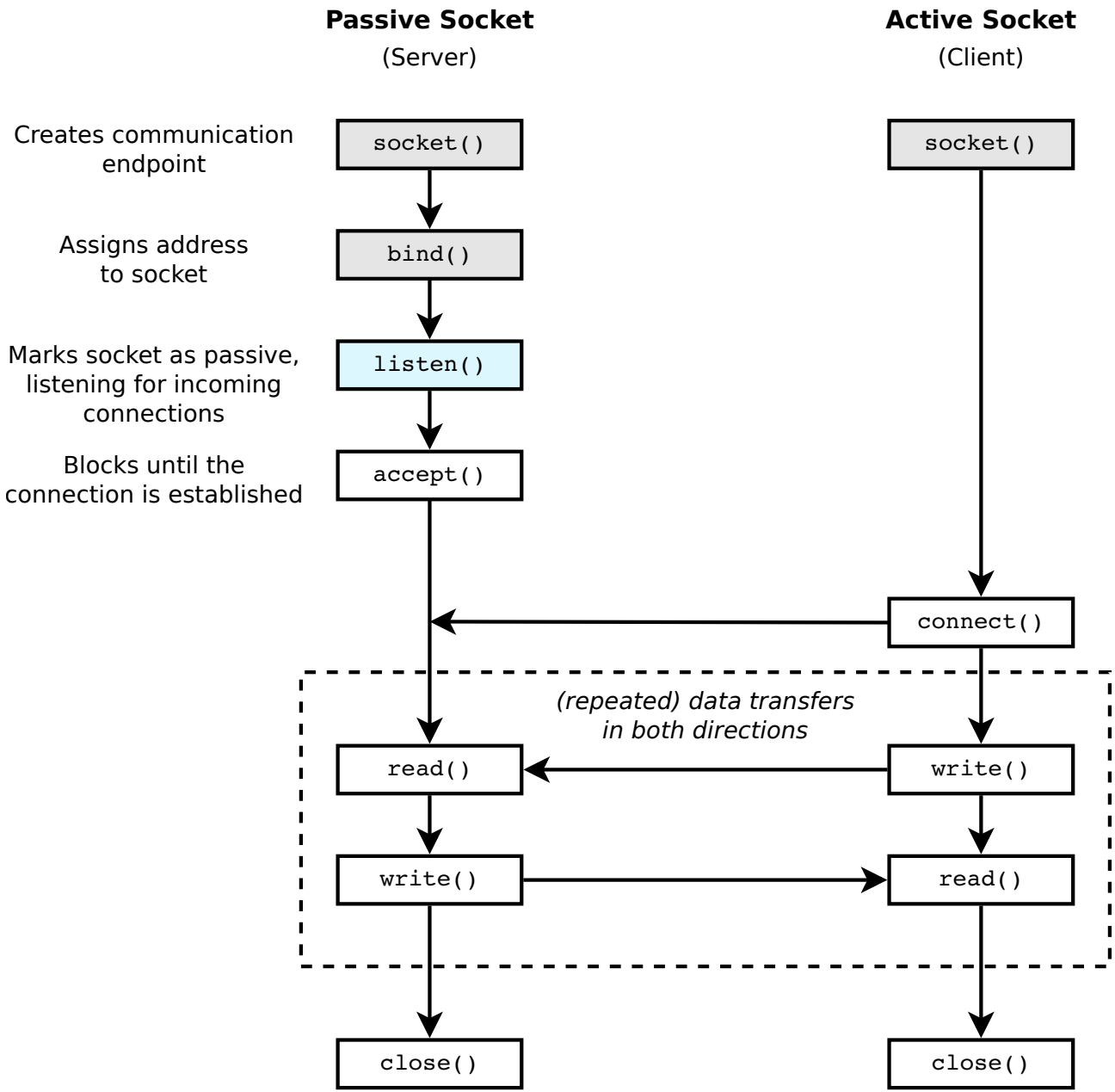
System Call: bind()

int **bind**(int socket, struct sockaddr *address,
socklen_t addr_len)

- Assigns the specified address to a socket
  - socket: file descriptor of the socket
  - address: data structure with the desired address
  - addr_len: size of the address data structure
- Return value: 0 on success, -1 on failure (→ errno)

```
struct sockaddr_in *sa;
...

if (bind(sockfd, sa, sizeof(struct sockaddr_in)) < 0)
    // error
```

Client-Server Beispiel



System Call: listen()

int **listen**(int socket, int backlog)

- Listen for connections on a socket (= mark it as passive)
- For connection-oriented protocols only

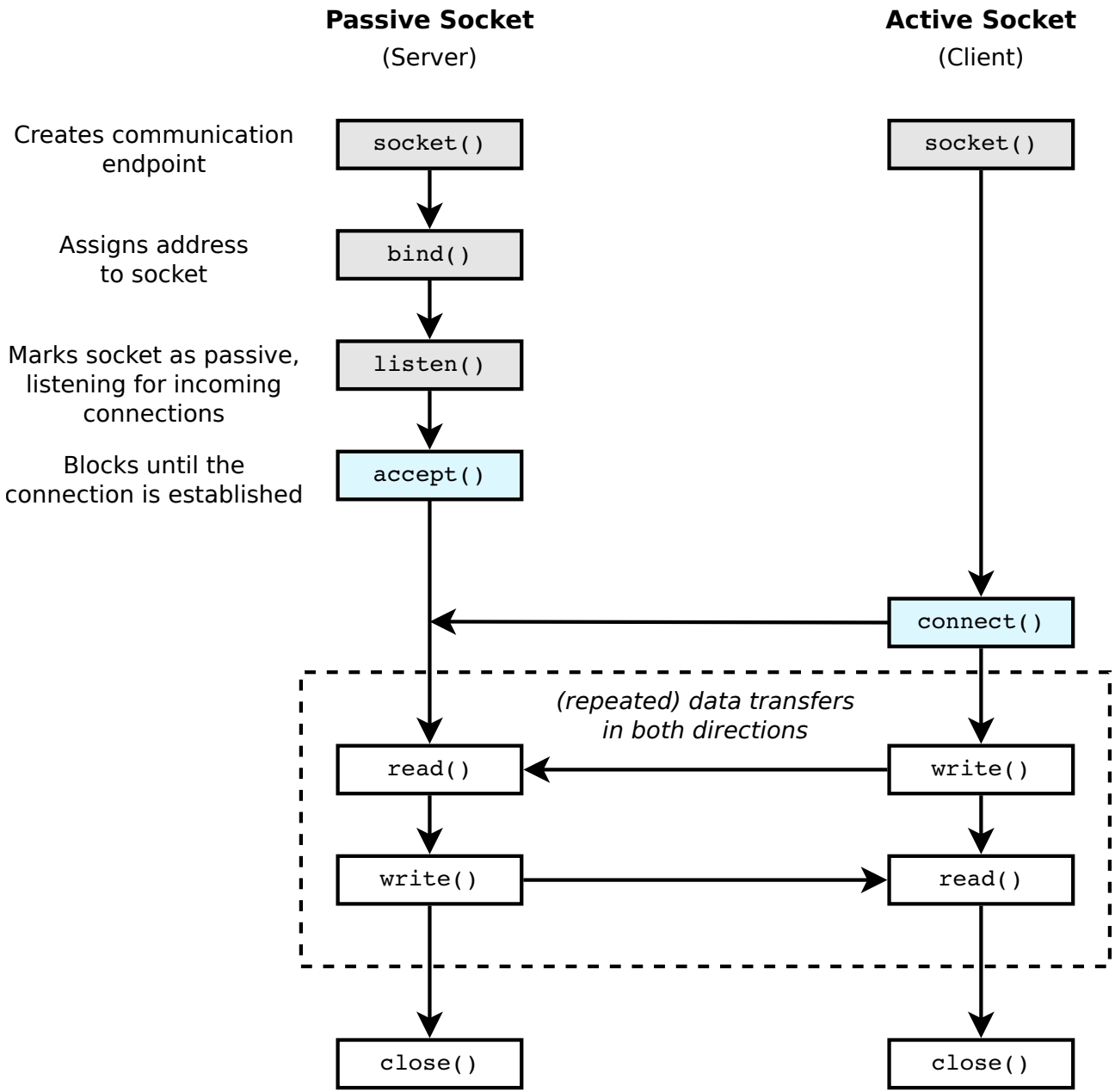
  socket — socket file descriptor

  backlog — number of connection requests, which are managed in a queue by the OS, until the server accepts them

- Return value: 0 on success, -1 on failure (→ errno)

```
if (listen(sockfd, 1) < 0)
    // error
```

Client-Server Example



System Call: accept()

int **accept**(int socket, struct sockaddr *address,
socklen_t *addr_len)

- Accept a new connection on a socket (passive, server)

  socket — socket file descriptor

  address — pointer to a sockaddr structure where the address of the connecting socket is returned (actual type depends on protocol, e.g. `sockaddr_in`), `NULL` possible

  addr_len — pointer to the size of the structure in address
- Blocks if there is no pending request
- Returns a new socket (file descriptor) for the first pending connection or -1 on error ($\rightarrow$ errno)

```
int connfd = accept(sockfd, NULL, NULL);

if (connfd < 0)
    // error
```

System Call: connect()

int **connect**(int socket, const struct sockaddr *address,
socklen_t addr_len)

- Initiate a connection (active, client)

socket	socket file descriptor
address	address of the server (destination)
addr_len	size of the address structure

- Returns after the connection has been established
- The operating system of the client selects an arbitrary, unused port

```
struct sockaddr_in server_addr;
...

if (connect(sockfd, &server_addr, sizeof(server_addr)) < 0)
    // error
```

getaddrinfo(3)

int **getaddrinfo**(const char *node, const char *service,
const struct addrinfo *hints, struct addrinfo **res)

- Create a suitable socket address with getaddrinfo(3)
  - node: Hostname (e.g. “localhost”, “173.194.44.232”, “google.com”) or NULL (for usage with bind())
  - service: port no. or name of service (e.g. “80”, “http”)
  - hints: Selection criteria
  - res: Destination address for the resulting `addrinfo` structure (filled by getaddrinfo)
- Returns 0 on success or an error code **(no use of errno!)**
- See also gai_strerror(3) and freeaddrinfo(3)

```
struct addrinfo hints, *ai;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

int res = getaddrinfo("localhost", "1280", &hints, &ai);
if (res != 0)
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(res));
```

Example: getaddrinfo()

Client

```
struct addrinfo hints, *ai;
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;


int res = getaddrinfo("localhost", "1280", &hints, &ai);
if (res != 0) {
    // error
}

int sockfd = socket(ai->ai_family, ai->ai_socktype,
                    ai->ai_protocol);
if (sockfd < 0) {
    // error
}

if (connect(sockfd, ai->ai_addr, ai->ai_addrlen) < 0) {
    // error
}

freeaddrinfo(ai);
```

Example: getaddrinfo()

Server

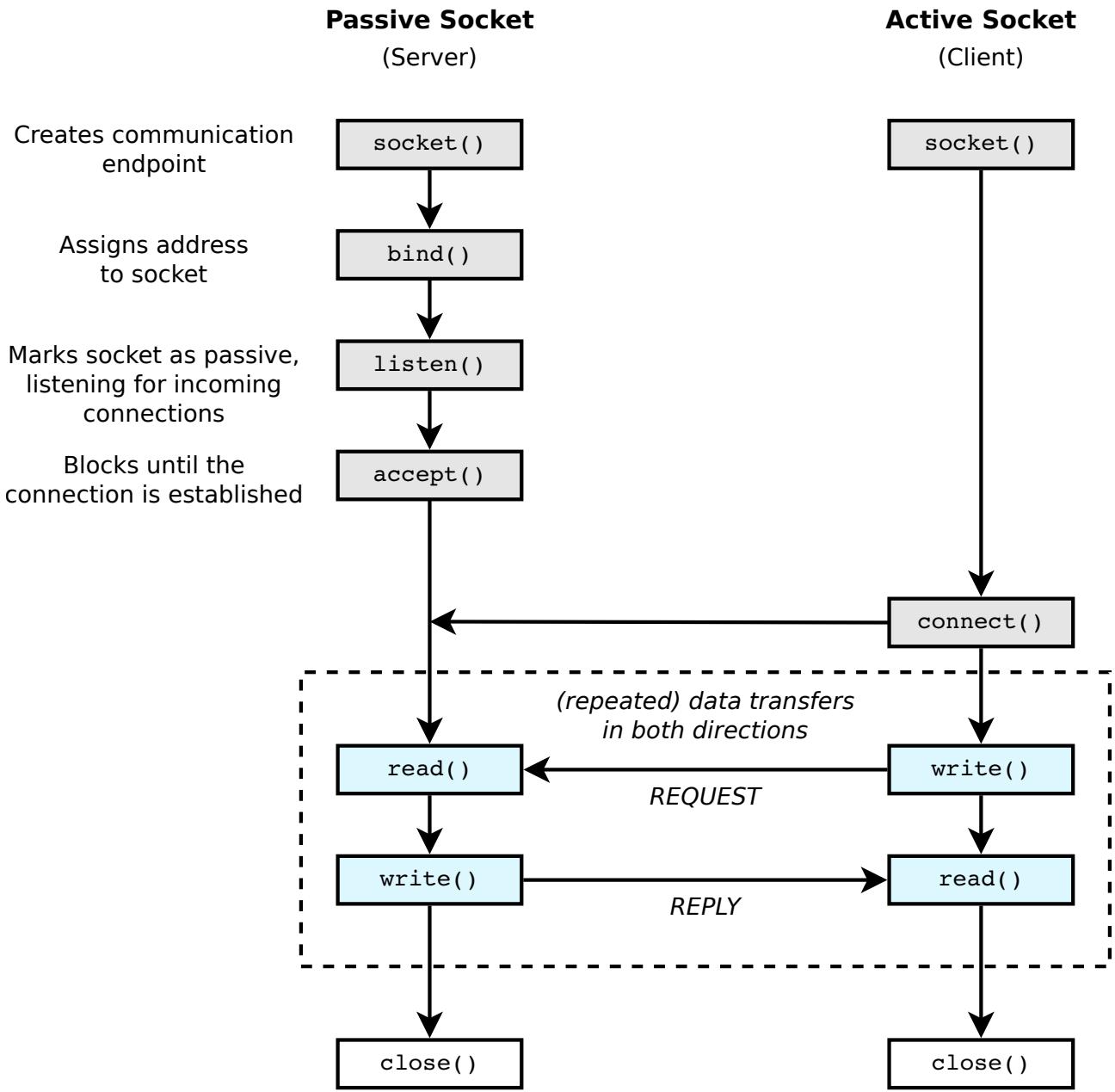
```
struct addrinfo hints, *ai;
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

int res = getaddrinfo( NULL      , "1280", &hints, &ai);
if (res != 0) {
    // error
}

int sockfd = socket(ai->ai_family, ai->ai_socktype,
                    ai->ai_protocol);
if (sockfd < 0) {
    // error
}

if (   bind(sockfd, ai->ai_addr, ai->ai_addrlen) < 0) {
    // error
}

freeaddrinfo(ai);
```

gethostbyname(3)

getaddrinfo replaces the obsolete function gethostbyname

- gethostbyname does not support IP version 6 and is obsolete
- Most of the C socket examples that can be found online still use the old gethostbyname
- **You must not use gethostbyname and related functions (i.e. gethostbyaddr, gethostbyname2, gethostent_r, gethostbyaddr_r, gethostbyname_r, gethostbyname2_r, . . .) during the exercises or the exams!**

Client-Server Example



Send and Receive

write(2) and read(2)

- After the connection has been established, the file descriptor of the socket is used to read and write data
- Use read and write the same way as with files

```
char buf[80];
int pos, cnt;

for (pos = 0; pos < sizeof(buf); ) {
    cnt = read(sockfd, buf + pos, sizeof(buf) - pos);

    if (cnt < 0) {
        if (errno != EINTR)
            // other error than EINTR
    } else
        pos += cnt;
}
```

- You can also use the Stream I/O with fdopen() (take care with buffering, use fflush() to send the data!)

Send and Receive

Stream I/O - Example without error handling

```
struct addrinfo hints, *ai;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("neverssl.com", "http", &hints, &ai);

int sockfd = socket(ai->ai_family, ai->ai_socktype,
                    ai->ai_protocol);
connect(sockfd, ai->ai_addr, ai->ai_addrlen);

FILE *sockfile = fdopen(sockfd, "r+");

fputs("GET / HTTP/1.1\r\nHost: neverssl.com\r\n\r\n",
      sockfile);
fflush(sockfile); // send all buffered data

char buf[1024];
while (fgets(buf, sizeof(buf), sockfile) != NULL)
    fputs(buf, stdout);
```

Add error handling to this code!

Send and Receive

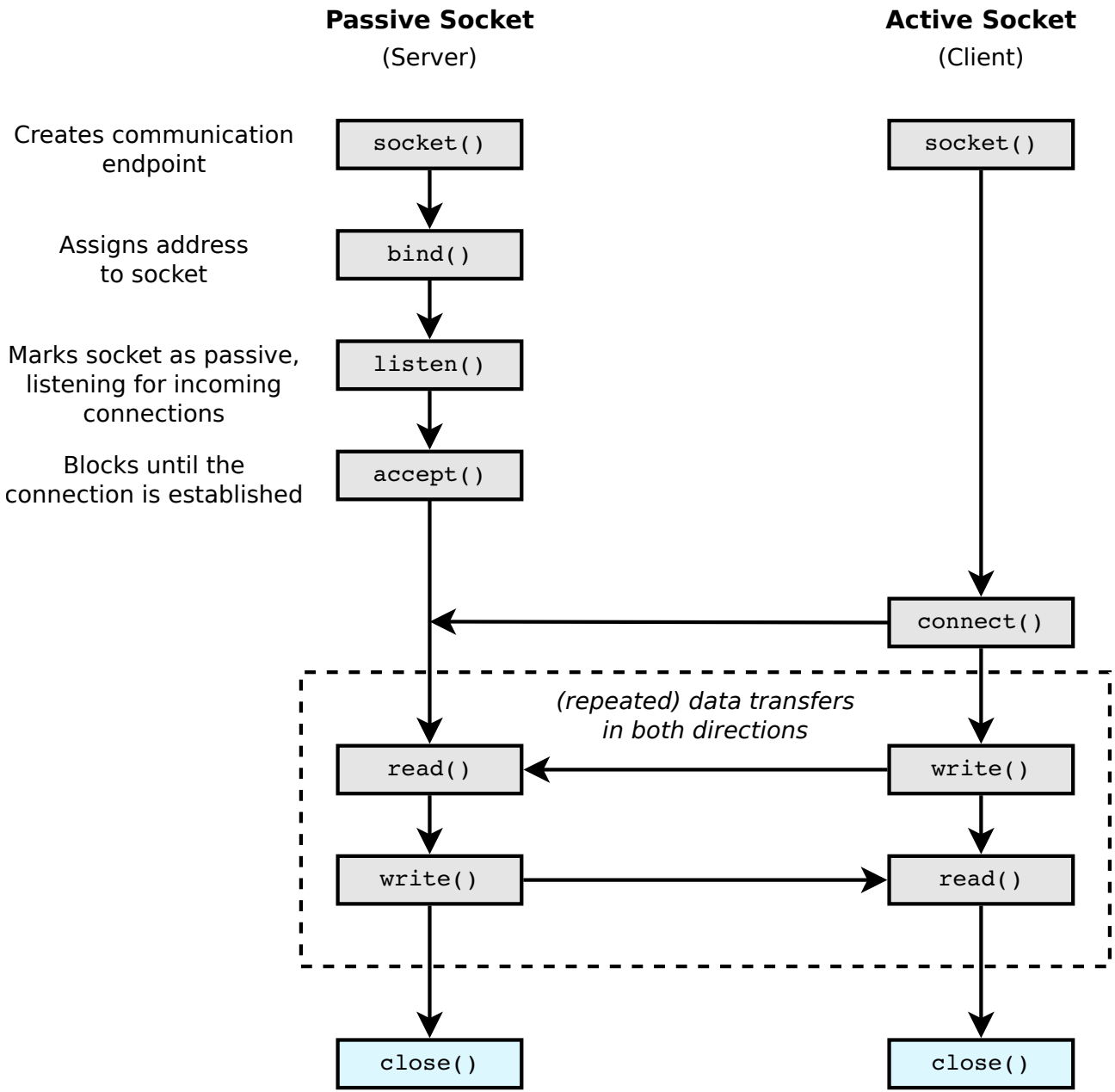
send(2) and recv(2)

int **send**(int socket, const void *msg, size_t msg_len, int flags)

int **recv**(int socket, void* buf, size_t buf_len, int flags)

- Spezializations of write und read for sockets
- Return value and first three arguments same as for write und read
- Additional argument: flags
  - MSG_DONTWAIT – Non-blocking send/receive
  - MSG_WAITALL – Block until all data was received (exceptions: error, signal received)

Client-Server Example



Socket Options

int **setsockopt**(int socket, int level, int option_name,
const void *option_value, socklen_t option_len)

- Set options on a socket (see man page for full list: setsockopt(2), socket(7), ip(7))
- Useful to avoid the error "Address already in use" (EADDRINUSE) with bind upon restarting your server program (otherwise the port remains unusable for approximately 1 min after the server was terminated)

```
int optval = 1;
setsockopt(serverfd, SOL_SOCKET, SO_REUSEADDR, &optval,
           sizeof optval);
```

Exercise 3

Client and server for HTTP

- 3A: Client
- 3B: Server
- IPC via stream-oriented sockets
- Implement a subset of the HTTP (HyperText Transfer Procotol), used for requesting websites
- Your server can serve files to a web browser (e.g. Firefox)
- Your client can request files from webservers (unfortunately most webservers require HTTPS)
  - `http://pan.vmars.tuwien.ac.at/osue/`
  - `http://neverssl.com/`
  - `http://www.nonhttps.com/`

Material

- OSUE-Wiki: Sockets
  `http://wiki.vmars.tuwien.ac.at/sockets`
- The GNU C Library Reference Manual,
  Ch. 12 (Stream I/O), Ch. 16 (Sockets)
  `http://www.gnu.org/software/libc/manual/html_node/`
- Beej's Guide to Network Programming
  `http://beej.us/guide/bgnet/`