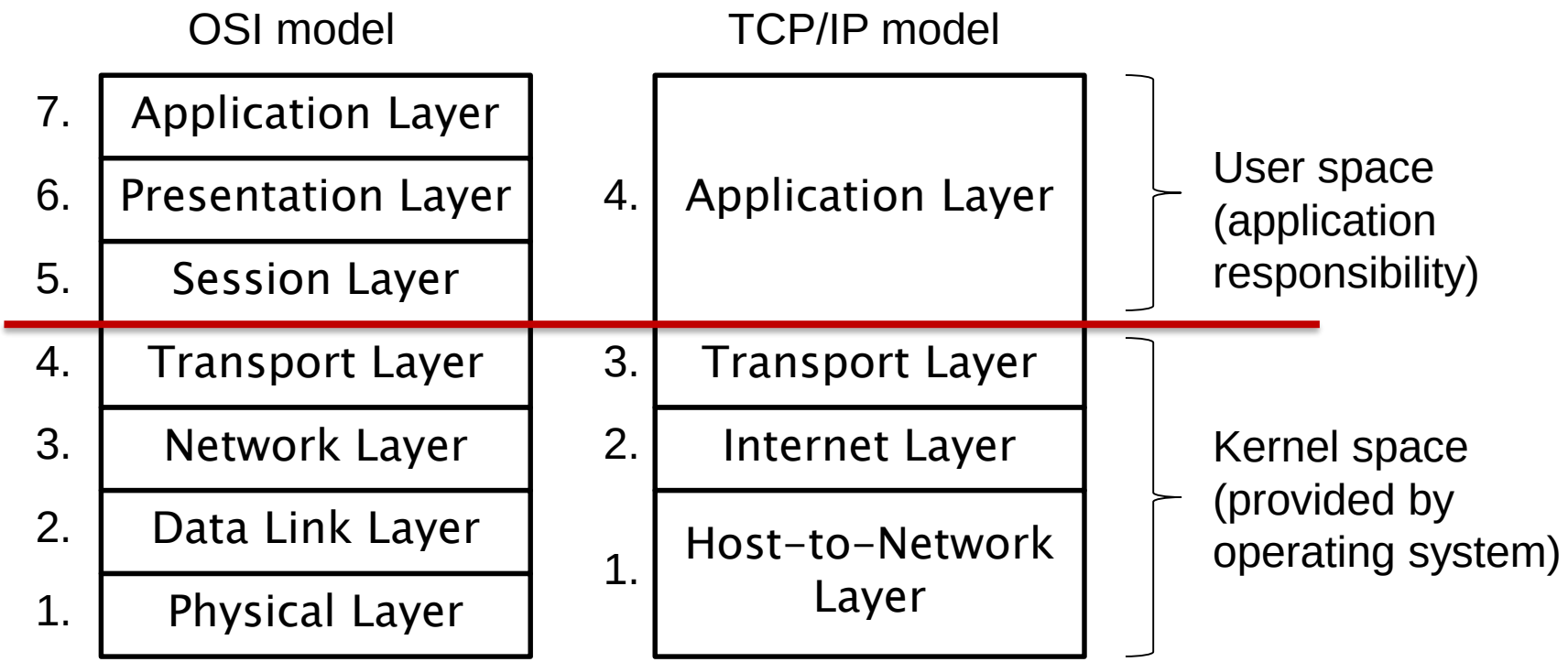
UNIVERSITÄT DUISBURG ESSEN

Internet-Technologie & Web Engineering
Sockets

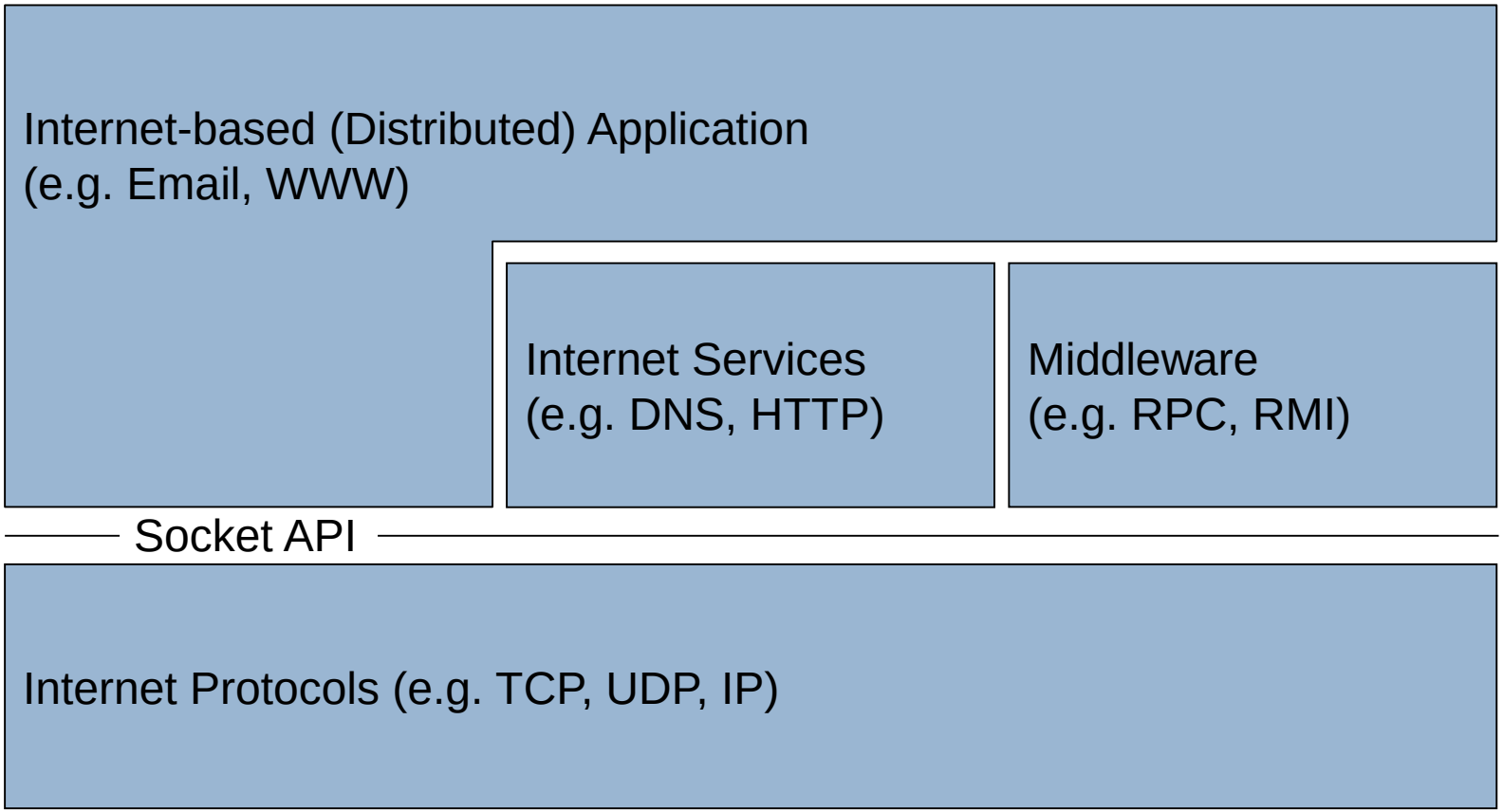
Dr.-Ing. Matthäus Wander

Universität Duisburg-Essen

Network Models



Software Architecture



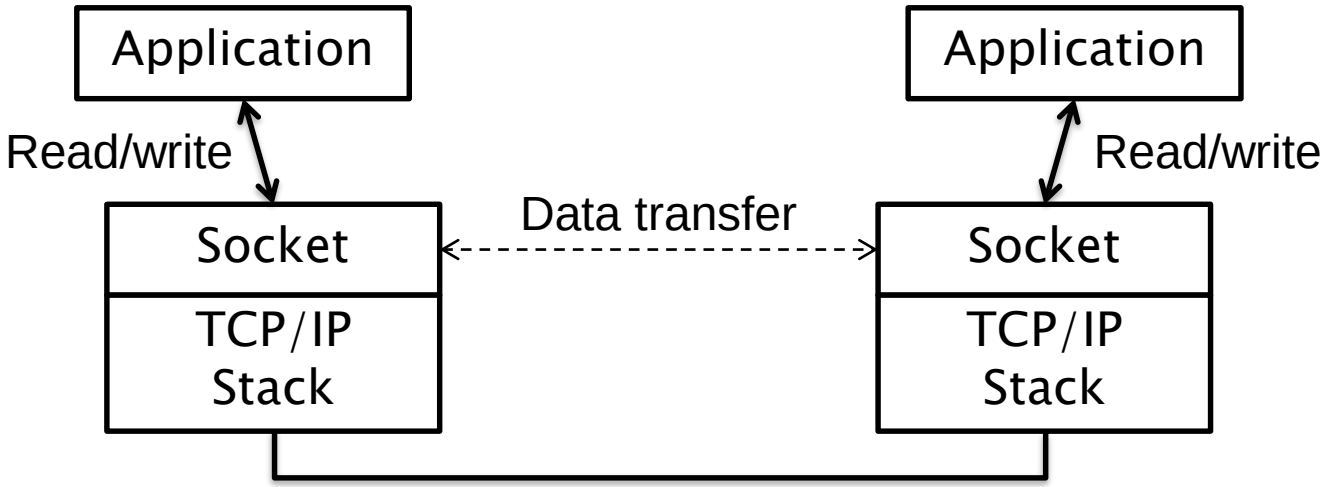
Network API Wishlist

- Generic application programming interface (API)
  - Support multiple communication protocol suites (families)
  - Address representation independence
- Support for datagram-oriented (UDP) and stream-oriented (TCP) communication
- Operating system (OS) independent
- Programming language independent

Sockets (1)

- The TCP/IP protocol suite does not include a network API definition
- Different APIs exist
  - e.g. Sockets, TLI, XTI, Winsock, MacTCP, …
- Sockets are the de-facto standard network API
  - Origin: Unix network extensions by UC Berkeley
  - Sockets behave like Unix files, pipes & FIFOs (open, read, write, close)
  - Variants of socket API are available for all major operating systems and programming languages

Sockets (2)



- Socket: abstract representation of a **communication endpoint**
- Two connected sockets appear to communicate directly with each other

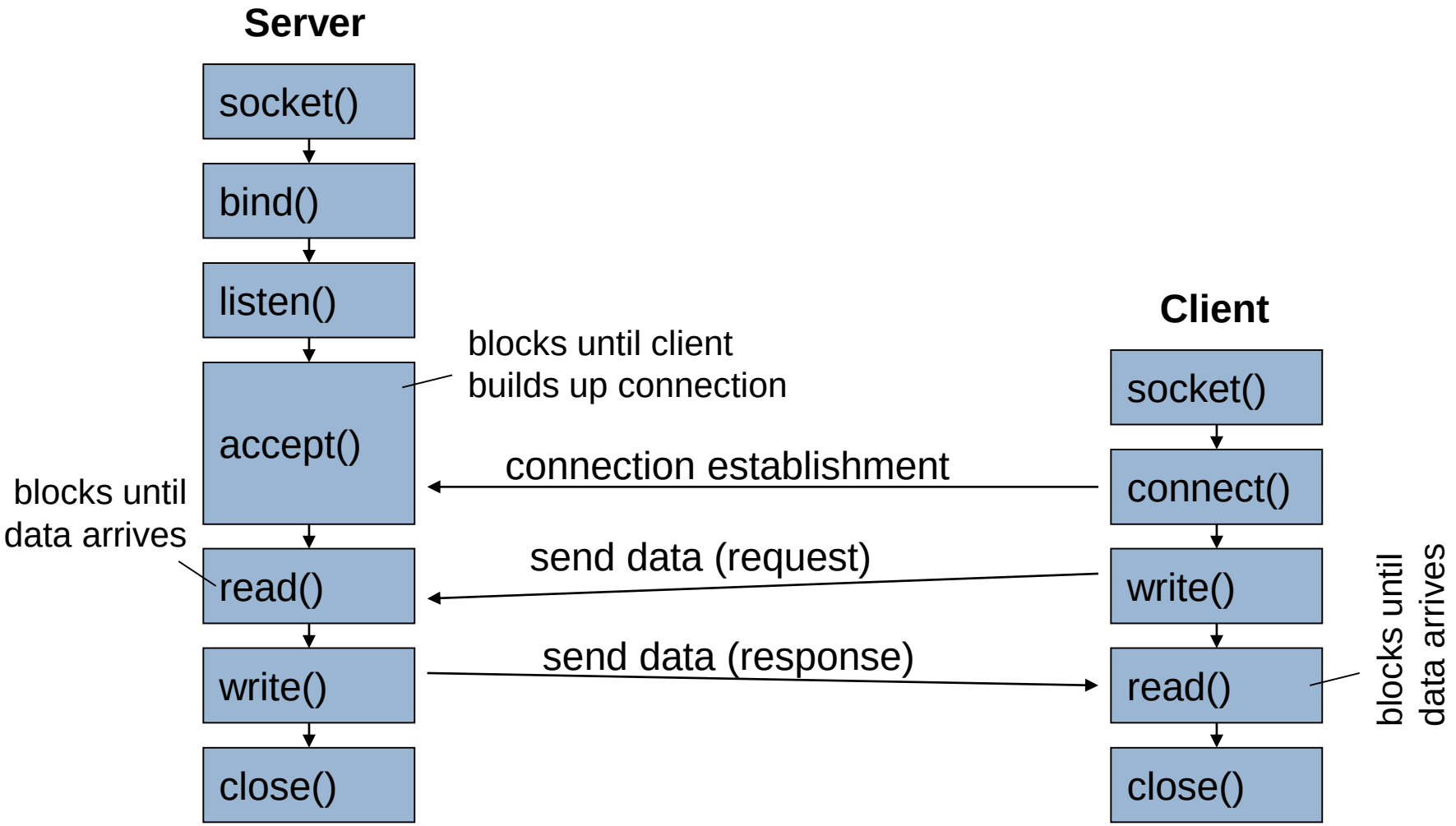
Sockets (3)

- IPv4/IPv6 support
  - Indicated via „address family“
- TCP/UDP support
  - Stream-oriented communication for TCP
  - Datagram-oriented communication for UDP
- Client/server support
  - Listening socket for servers
  - Connecting socket for clients

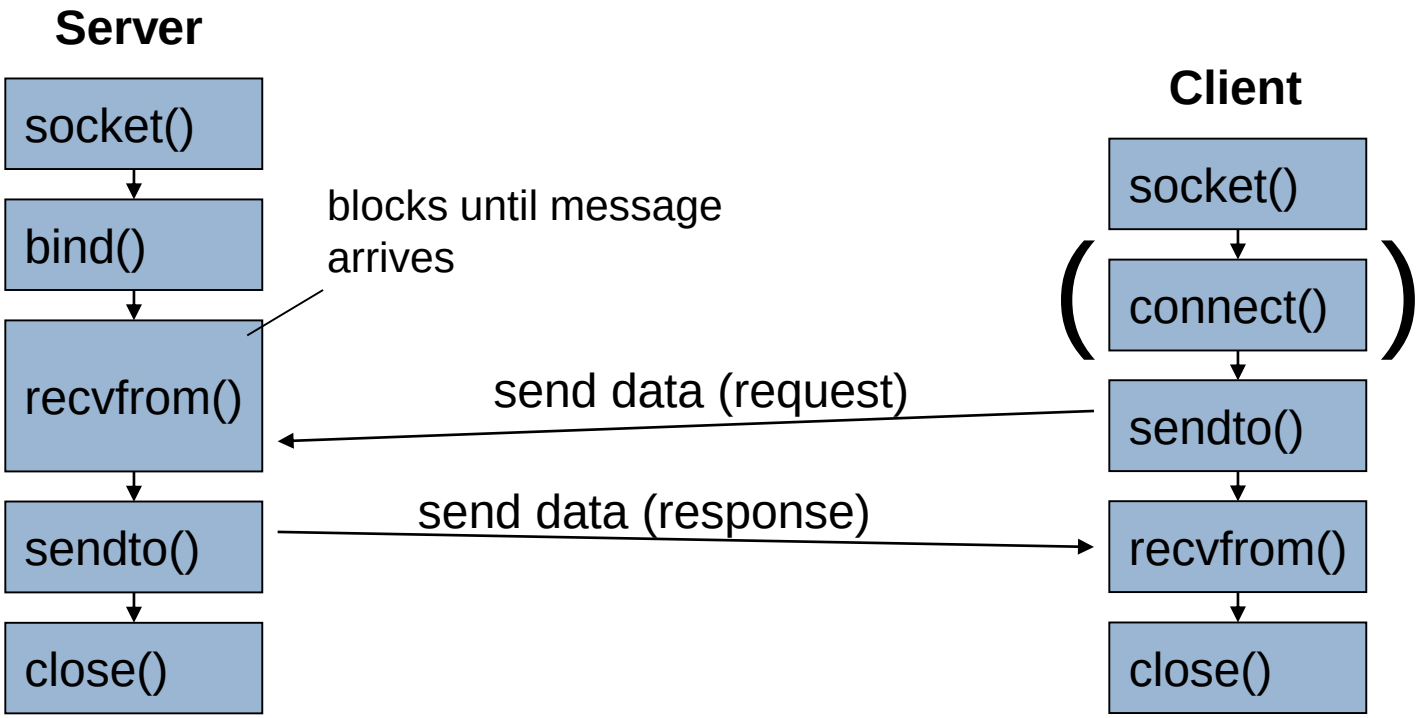
Sockets (4)

- Socket API functionality
  - Create & destroy socket
  - Choose protocol
  - Specify local & remote communication endpoint
  - Initiate outgoing & handle incoming connection
  - Send & receive data
  - Return error in case when the network fails
  - Configure protocol via socket options
- Sockets are identified via descriptor (file handle)

Sockets: Stream-Oriented Communication



Sockets: Datagram-Oriented Communication



Create & Close

```
sockfd = socket(family, type, protocol);

close(sockfd);
```

- family: protocol family
  - e.g. AF_INET: IPv4
  - e.g. AF_INET6: IPv6
- type: socket type
  - e.g. SOCK_STREAM: TCP
  - e.g. SOCK_DGRAM: UDP
- protocol: usually 0
- sockfd: socket descriptor (integer chosen by OS)

Bind to local endpoint

```
bind(sockfd, *localaddr, addrlen);
```

- sockfd: socket descriptor
- localaddr:
  - Local TCP/IP endpoint to bind socket to
  - IP address (specific or any) + port number
- addrlen: length of localaddr data structure
- Important for server, optional for client
  - Without bind(), socket will be bound to random port from dynamic port range (49152-65535)

Client: Connect to remote address

```
connect(sockfd, *destaddr, addrlen);
```

- Stream socket: Establish connection to server
  - Datagram: memorize destination endpoint (optional)
- sockfd: socket descriptor
- destaddr:
  - Destination TCP/IP endpoint to connect to
  - IP address + port number
- addrlen: length of destaddr data structure

Server: Listen for incoming data

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

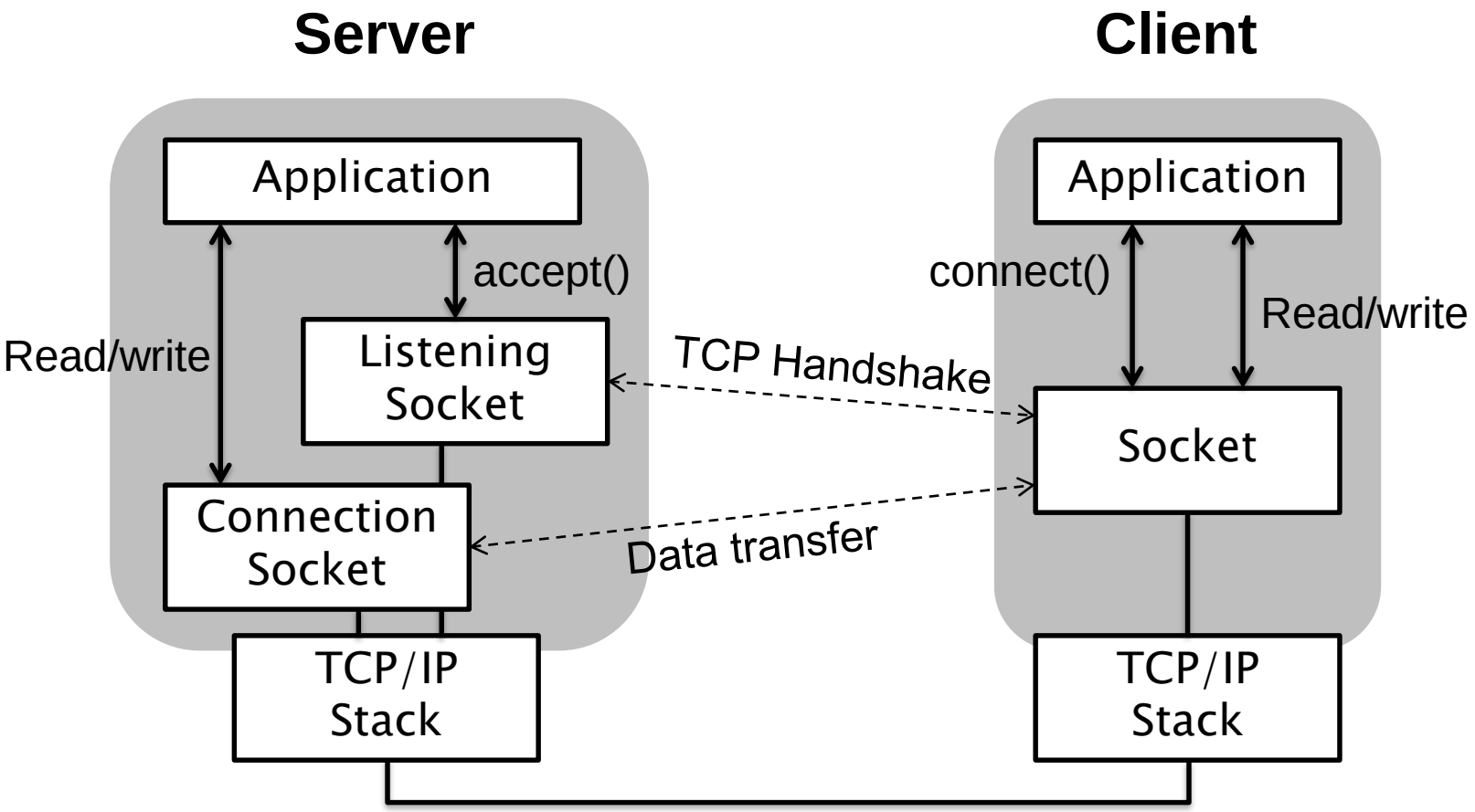
- Declare a stream socket as server socket
  - Not used for datagram sockets
- sockfd: socket descriptor
- backlog:
  - Maximum number of not fully established connections
  - High number requires more memory
  - Low number could make server temporarily unreachable

Server: Accept an incoming connection

```
newsock = accept(sockfd, *remoteaddr,
  *addrlen);
```

- sockfd: descriptor of listening socket
- remoteaddr:
  - Remote TCP/IP endpoint of incoming connection
  - IP address + port number
  - Output parameter: data structure will be filled by OS
- addrlen: length of remoteaddr data structure
- newsock: descriptor of a newly created socket, which is connected with client

Connection Handling with Stream Sockets



Send data

```
len = write(sockfd, *buf, buflen);
```

- Both server and client can write (bidirectionally)
- sockfd: socket descriptor
- buf: buffer with message to be sent
- buflen: length of buf
- len: number of bytes actually sent
  - Beware: May send fewer data than requested
  - Beware: write() returns when data has been taken by OS, not when data has arrived at destination
- Beware: If buffers are full, write() will block

Send data (2)

```
len = sendto(sockfd, *buf, buflen, flags,
  *destaddr, addrlen);
```

- Datagram socket: send datagram to destination endpoint
- sockfd, buf, buflen: identical to write()
- flags: usually 0
- destaddr, addrlen: destination like in connect()
- len: number of bytes actually sent, like in write()

Receive data

```
len = read(sockfd, *buf, buflen);
```

- Both server and client can read (bidirectionally)
- sockfd: socket descriptor
- buf: buffer to be filled with message
- buflen: maximum length to read
- len: number of bytes actually read
  - Beware: May read fewer data than requested
- Beware: If no data is available, read() will block

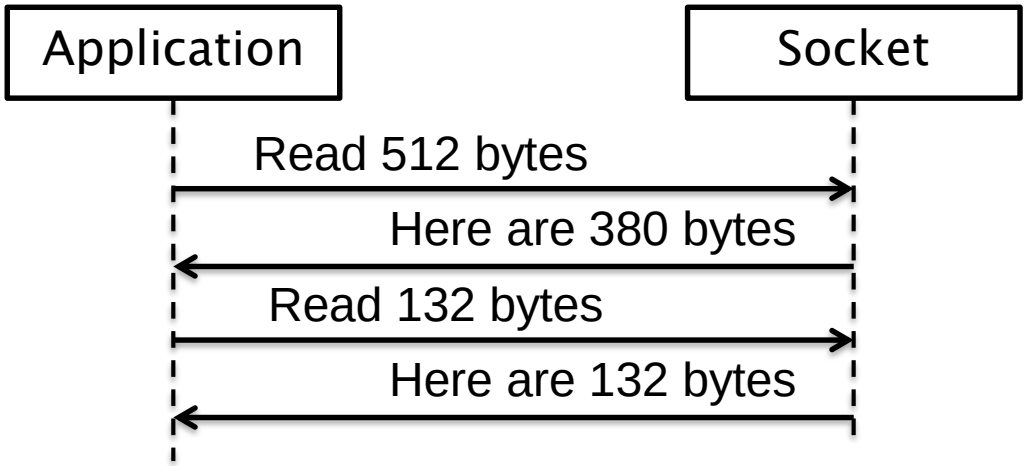
Receive data (2)

```
len = recvfrom(sockfd, *buf, buflen, flags,
  *remoteaddr, *addrlen);
```

- Datagram socket: wait for datagram from a specific remote endpoint
- sockfd, buf, buflen: identical to read()
- flags: usually 0
- remoteaddr, addrlen: endpoint like in accept()
- len: number of bytes actually read, like in read()

Reading Data with Stream Sockets

- Expect read()/write() to handle fewer bytes than requested (not an error)



- On connection close: read() returns 0 bytes
- On error: read() returns -1 (or throws Exception)

Socket Server in C

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
int main() {
    const int MAXBUF = 1023;
    const int SERVER_PORT = 10999;
    int listensock, clientsock;
    struct sockaddr_in serverSocketAddr, clientSocketAddr;
    int addrLen = sizeof(struct sockaddr);
    char buffer[MAXBUF+1];
    int msgLen;

    listensock = socket(AF_INET, SOCK_STREAM, 0);

    serverSocketAddr.sin_family = AF_INET;
    serverSocketAddr.sin_addr.s_addr = INADDR_ANY;
    serverSocketAddr.sin_port = htons(SERVER_PORT);
    memset(&(serverSocketAddr.sin_zero), '\0', 8);

    bind(listensock, (struct sockaddr *)&serverSocketAddr, addrLen);
    listen(listensock, 5);
    printf("Server running on port %d. Waiting for connections.\n",
        SERVER_PORT);
```

Socket Server in C

```
 do {
        clientsock = accept(listensock, (struct sockaddr *)
            &clientSocketAddr, &addrLen);
        printf("Accepting connection.\n");
        msgLen = read(clientsock, buffer, MAXBUF);
        buffer[msgLen] = '\0';
        printf("Received message: %s\n", buffer);

        /* this is the place where you could do something useful */

        write(clientsock, buffer, msgLen);
        close(clientsock);
        printf("Closed connection.\n");
    } while(1); // endless loop
}
```

Socket Client in C

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
int main() {
    const int MAXBUF = 1023;
    const int PORT = 10999;
    const char DESTADDR[] = "127.0.0.1";
    int clientsock;
    struct sockaddr_in serverSocketAddr;
    int addrLen = sizeof(struct sockaddr);
    char buffer[MAXBUF+1];
    int msgLen;

    clientsock = socket(AF_INET, SOCK_STREAM, 0);

    serverSocketAddr.sin_family = AF_INET;
    serverSocketAddr.sin_port = htons(PORT);
    inet_pton(AF_INET, DESTADDR, &serverSocketAddr.sin_addr);
    memset(&(serverSocketAddr.sin_zero), '\0', 8);
```

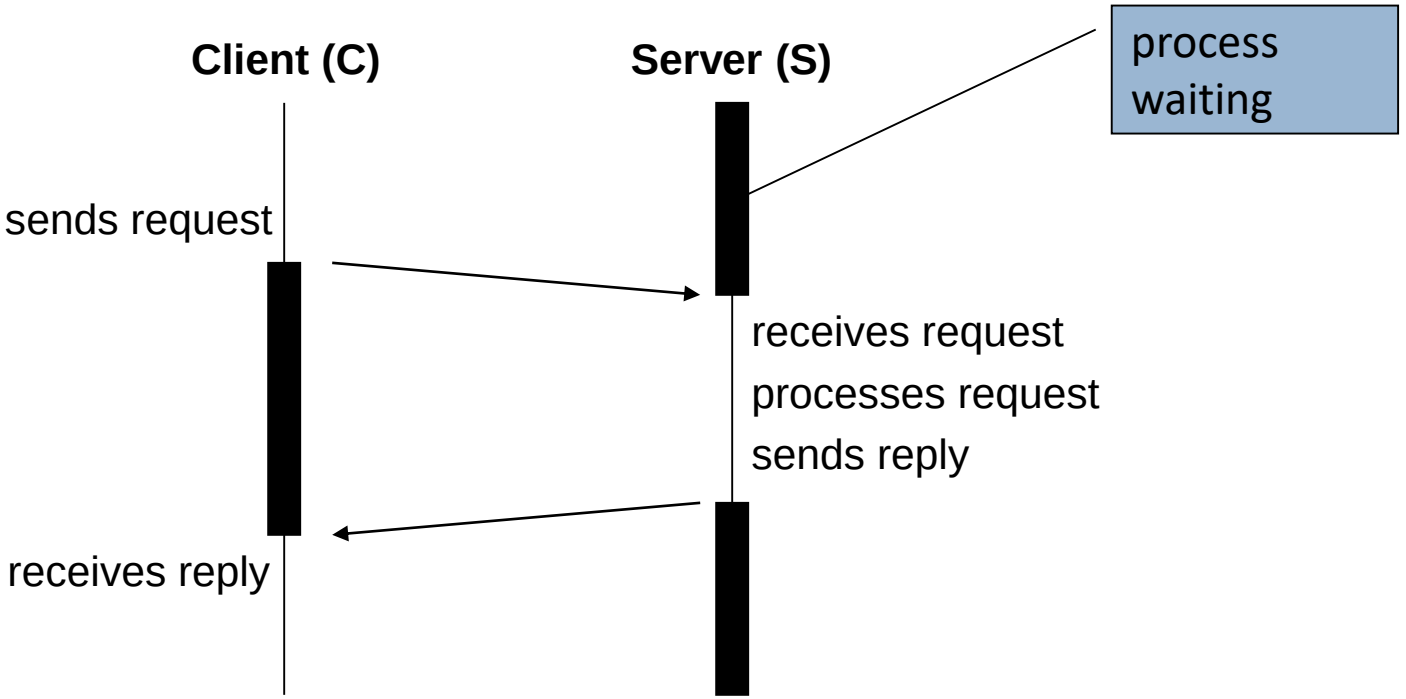
Socket Client in C

```
    connect(clientsock, (struct sockaddr*)&serverSocketAddr,
        addrLen);

    write(clientsock, "Hello World", 11); // send 11 bytes

    msgLen = read(clientsock, buffer, MAXBUF);
    buffer[msgLen] = '\0';
    printf("Received the following message from server: %s\n",
        buffer);

    close(clientsock);
}
```

Sockets in Java

- Class *java.net.Socket*
  - Object-oriented interface to Socket API
  - *java.net.ServerSocket* for listening stream Sockets
- Read/write with InputStream/OutputStream
  - Processes single `byte` or `byte[]`
- How to read/write text?
  - `byte[] buf = "foo".getBytes();`
  - `String foo = new String(buf);`
  - Or use e.g. InputStreamReader/OutputStreamWriter

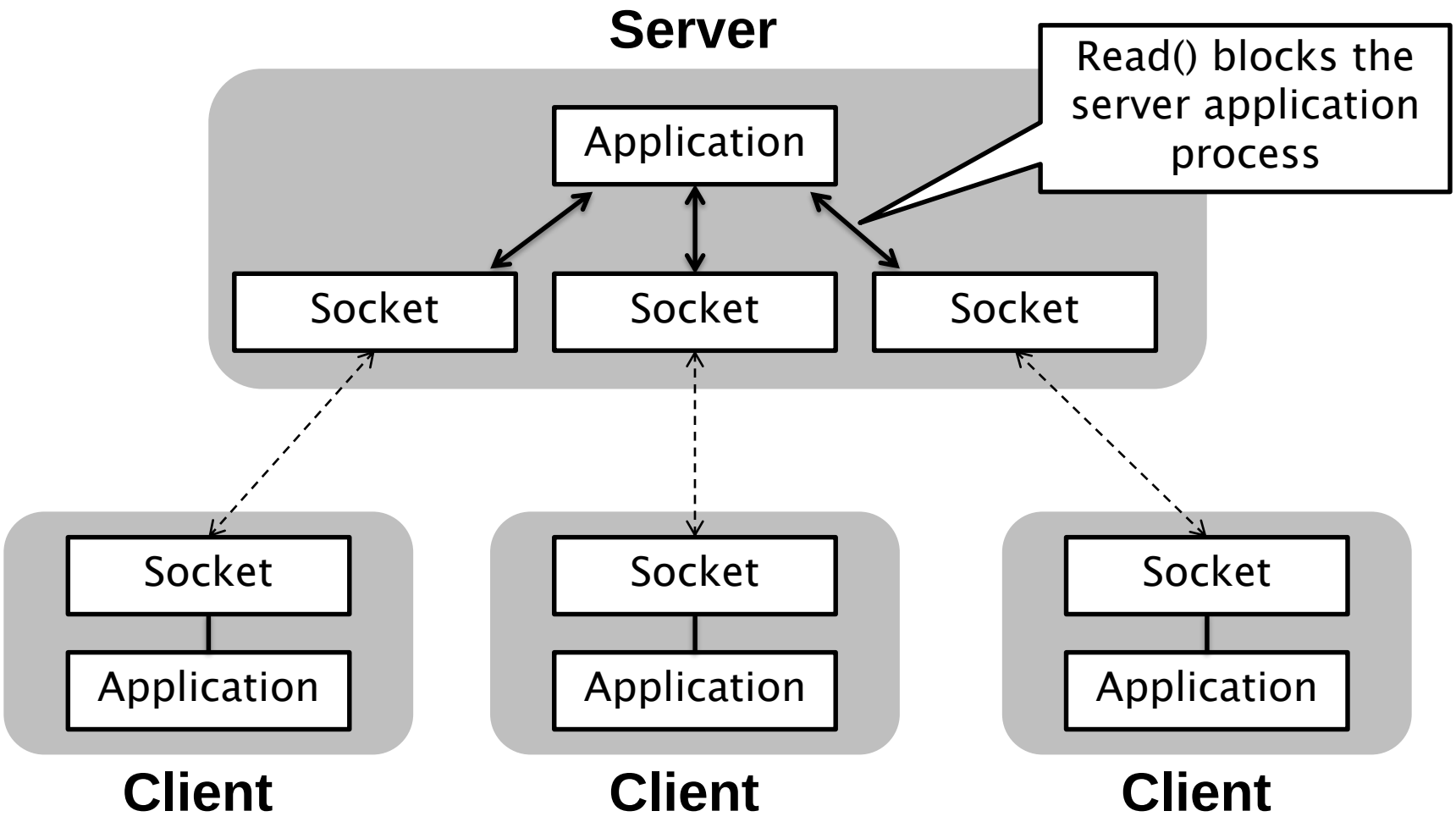
Client/Server Model



Stateless vs. Stateful

- Stateless server: no client state kept across requests
  - Simple for server, easier to handle lots of clients
  - But: all necessary information must be included into request
- Stateful server: client state kept across requests
  - Typically easier for clients
  - State handling necessary at server (e.g. when to delete state?)
  - Server may run out of memory for lots of (concurrent) clients

Dealing with Multiple Clients



Input/Output Models

- Blocking I/O: read() blocks until data is available
  - Default for a socket, easy to program
- Multithreading: create new thread per client
  - Can use blocking I/O in a thread
  - Not scalable for large number of clients (stack memory per thread, context switching overhead)
- Non-blocking I/O: read() returns instantly
  - EWOULDBLOCK error if no data available
  - Loop over all clients (cumbersome for stateful server)

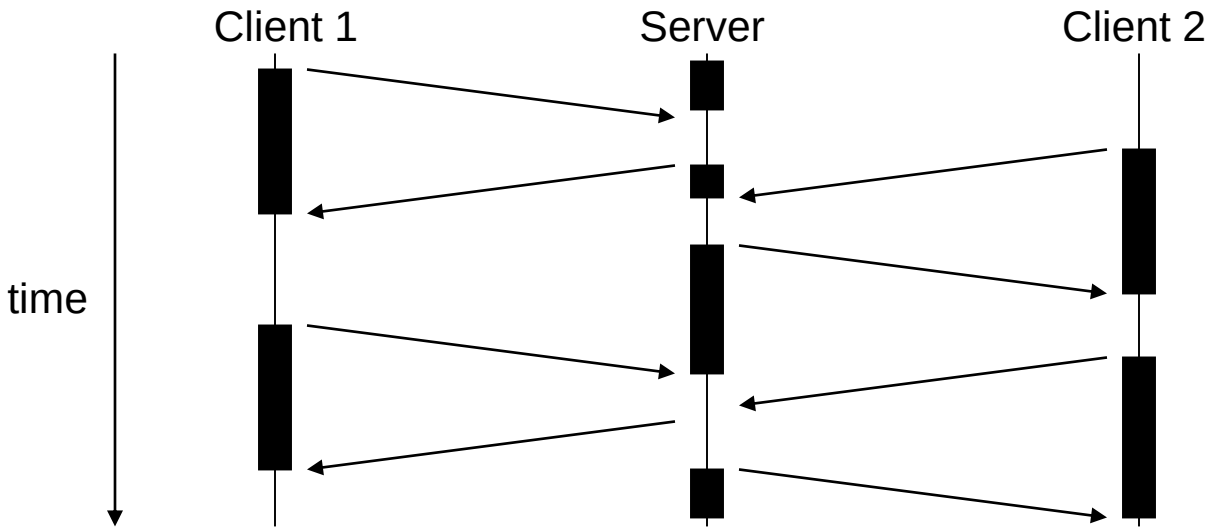
Input/Output Models

- Asynchronous I/O: pass callback to read()
  - Process data in event handler
  - Cumbersome to program for stateful servers
- I/O multiplexing: block until **any** socket is ready
  - Return first socket that has data available for reading
  - OS implementations: select, poll, epoll, kqueue

- Once we have sorted out how to handle I/O, we need to sort out how to process requests

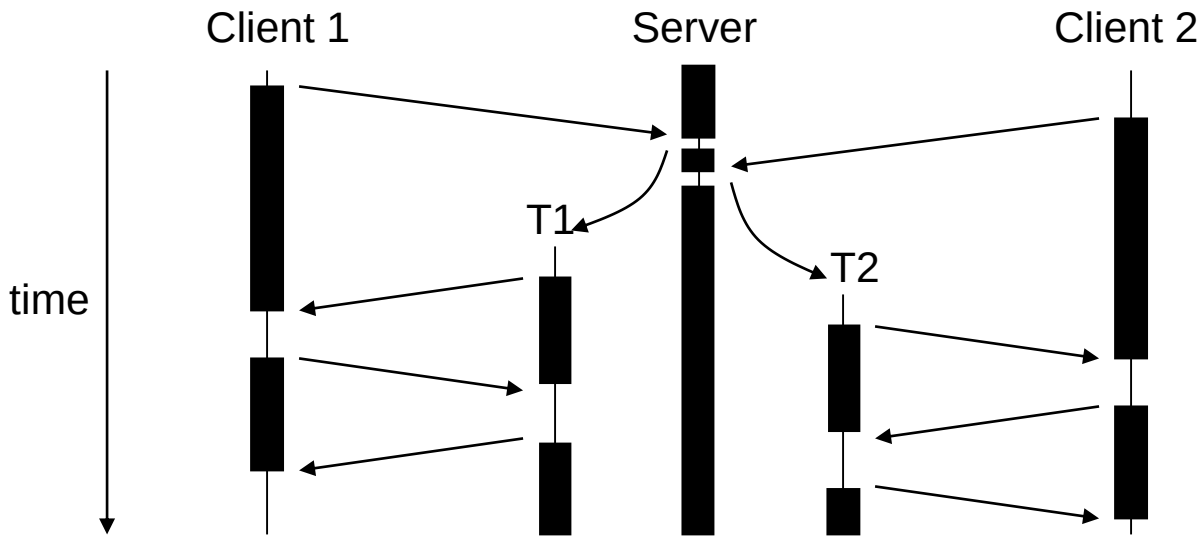
Sequential Processing

- Sequential handling of client requests (datagram or stream-oriented)
  - One process/thread for all server processing



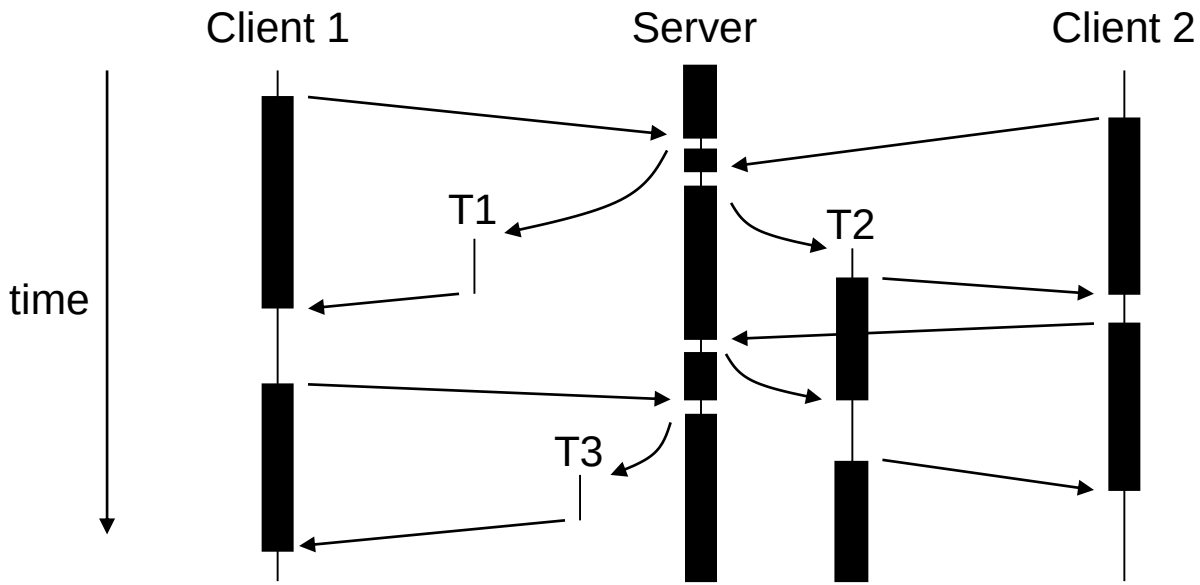
Parallel Processing (1)

- Parallel handling of clients (stream-oriented)
  - New thread for each new connection
  - Concurrent processing of requests in separate threads



Parallel Processing (2)

- Parallel handling of clients (datagram-oriented)
  - Server starts a pool of threads
  - Distributes requests to an available thread from pool



Conclusion

- Sockets are an API for the transport layer
- Follows Unix I/O principle („everything is a file“)
- Transmits plain bytes
- No serialization of data (presentation layer)
- Programming cumbersome
- Can be used by middleware as foundation for higher communication models, e.g. RPC, RMI