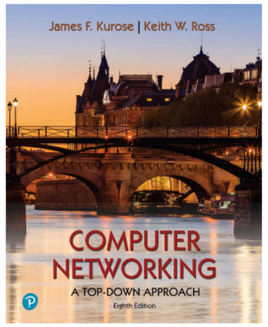
CS 241: Systems Programming
Lecture 28. Sockets II

Fall 2025

Prof. Stephen Checkoway

Slides adapted from the slides that accompany this book



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Layered Internet Protocol Stack

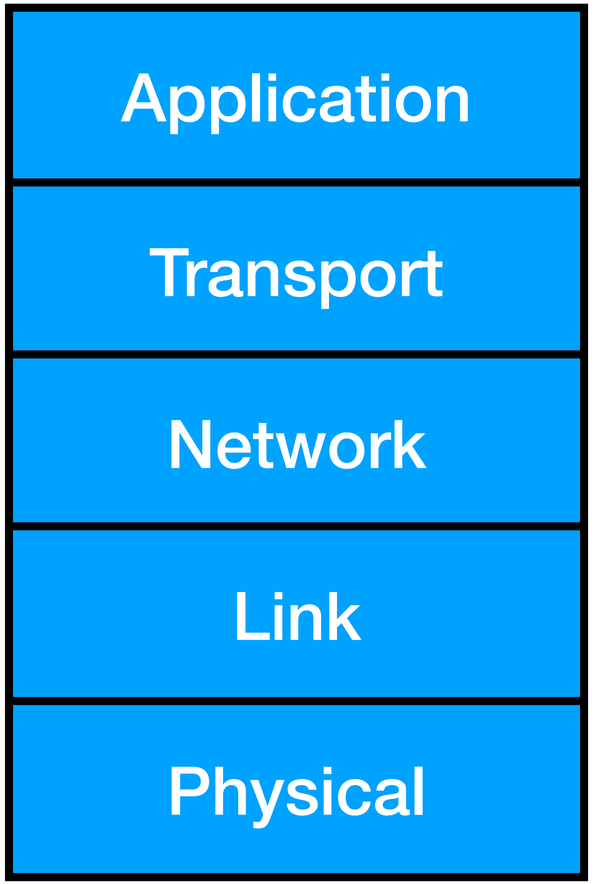
Application: supporting network applications
- e.g., HTTP

Transport: data transfer between processes on hosts
- e.g., TCP, UDP

Network: routing packets from source to destination
- e.g., IP

Link: data transfer between neighboring elements
- e.g., Ethernet, WiFi

Physical: transmit data over wires (or wireless signals)



Communicating with the transport layer

The application needs to specify:

- The destination that will receive the data → IP address + port number
- What type of transport service it wants → TCP or UDP
  - Does it need security?
  - Reliability? (e.g., no packets lost)
  - …
- The data that should be sent

The most common interface to the transport layer is the **socket** interface

TCP vs UDP

TCP: Transmission Control Protocol

TCP guarantees reliability

- All messages will get sent to the application, in order
- If a message gets lost, TCP will retransmit the message until it's received

TCP makes sure it doesn’t overwhelm receiver by sending too much, too quickly

TCP vs UDP

TCP: Transmission Control Protocol

TCP guarantees reliability

- All messages will get sent to the application, in order
- If a message gets lost, TCP will retransmit the message until it's received

TCP makes sure it doesn’t overwhelm receiver by sending too much, too quickly

UDP: User Datagram Protocol

UDP does NOT guarantee reliability

- Messages may be lost or arrive out-of-order

Because UDP doesn’t have to worry about reliability, it is much faster

For each of the following applications, choose whether you would use TCP or UDP, and justify why you would choose it. [Select any letter on your clicker]

- Online gaming
- SSH remote access
- Email
- Video conferencing
- Whatsapp

Sockets

Sockets

Process sends/receives messages to/from its socket

- Not unlike communicating between threads!

Sockets

Process sends/receives messages to/from its socket

- Not unlike communicating between threads!

Sockets are like a door

- Sending process shoves message out the door
- Sender relies on transport infrastructure at receiver door to deliver the message to socket at receiving process

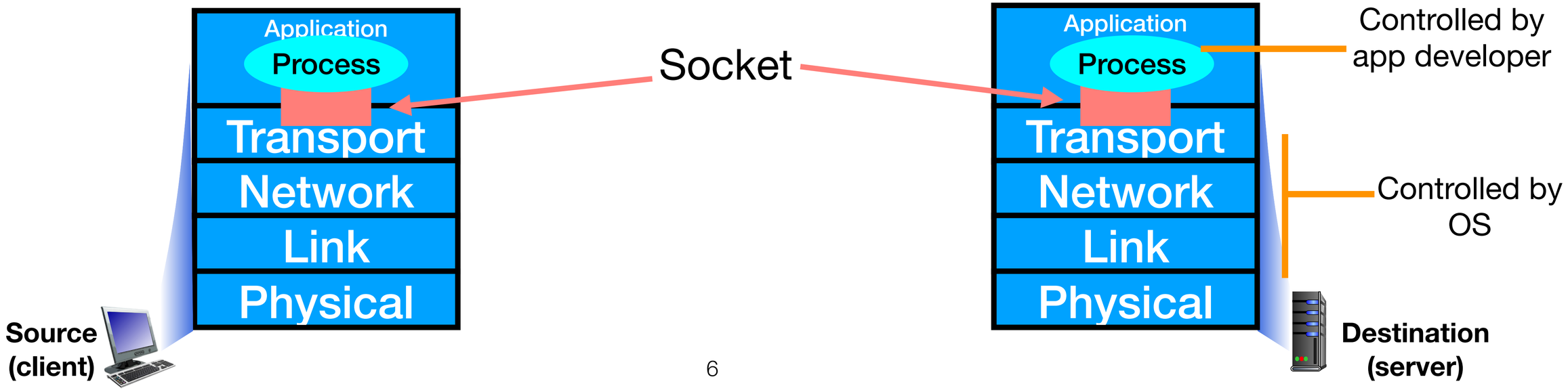
Sockets

Process sends/receives messages to/from its socket

- Not unlike communicating between threads!

Sockets are like a door

- Sending process shoves message out the door
- Sender relies on transport infrastructure at receiver door to deliver the message to socket at receiving process

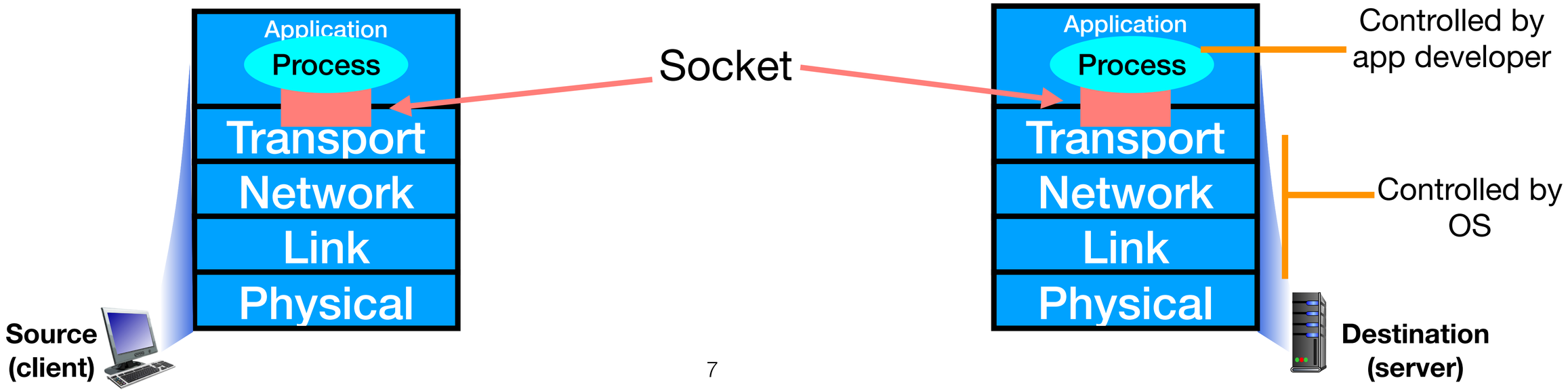


Socket Programming

Goal: build client/server applications that communicate using sockets

Two types of sockets

- TCP socket (stream)
- UDP socket (datagram)



Socket Programming

Two types of sockets

- TCP: reliable, byte stream-oriented
- UDP: unreliable datagram

Application example: [we'll implement this!]

1. Client reads a line of characters (data) from its keyboard and sends data to server
2. Server receives the data and converts the characters to uppercase
3. Server sends modified data to client
4. Client receives modified data and displays line on its screen

Socket Programming with UDP

UDP: no “connection” between client and server:

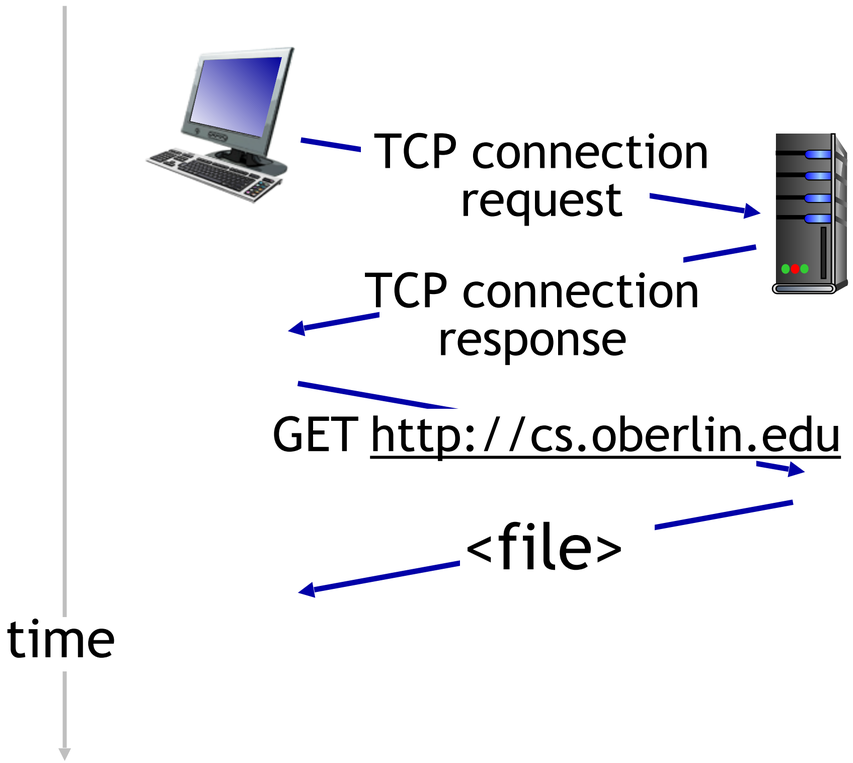
- No handshaking before sending data

Network Protocols

Network protocols are between computers (devices) instead of humans

A protocol defines:

- the ***format*** and ***order*** of messages send/received between network entities
- the ***actions*** taken upon message receipt



Socket Programming with UDP

Socket Programming with UDP

UDP: no “connection” between client and server:

- No handshaking before sending data
- Sender attaches IP destination address and port # to each packet
- Receiver extracts sender IP address and port # from received packet (so it knows where to send response)

Socket Programming with UDP

UDP: no “connection” between client and server:

- No handshaking before sending data
- Sender attaches IP destination address and port # to each packet
- Receiver extracts sender IP address and port # from received packet (so it knows where to send response)

UDP: transmitted data may be lost or received out-of-order

Socket Programming with UDP

UDP: no “connection” between client and server:

- No handshaking before sending data
- Sender attaches IP destination address and port # to each packet
- Receiver extracts sender IP address and port # from received packet (so it knows where to send response)

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server processes

Client/server socket interaction: UDP

server

client

Client/server socket interaction: UDP

server

create socket, port= x:

serverSocket = socket(…)

client

create socket:

clientSocket = socket(…)

Client/server socket interaction: UDP

server

client

create socket, port= x:

serverSocket = socket(…)

create socket:

clientSocket = socket(…)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

Client/server socket interaction: UDP

server

client 

create socket, port= x:

serverSocket = socket(…)

create socket:

clientSocket = socket(…)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
serverSocket

Client/server socket interaction: UDP

server

client

create socket, port= x:

serverSocket = socket(…)

create socket:

clientSocket = socket(…)

read datagram from serverSocket

Create datagram with serverIP address And port=x; send datagram via clientSocket

write reply to serverSocket specifying client address, port number

Client/server socket interaction: UDP

server

client

create socket, port= x:

serverSocket = socket(…)

create socket:

clientSocket = socket(…)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
serverSocket

write reply to serverSocket
specifying client address, port
number

read datagram from
clientSocket

close
clientSocket

Socket Programming with TCP

TCP: client MUST establish a connection with the server before sending data

- Server must have created a socket (door) that welcomes client's contact
- Client creates TCP socket, specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP (socket does this automatically, so application doesn't have to!)

When contacted by client, **server TCP creates new socket for server process to communicate with that particular client**

- Allows server to establish connections with multiple clients
- Client port number and IP address used to distinguish clients

Application viewpoint:

- TCP provides reliable, in-order byte-stream transfer between client and server processes

Client/server socket interaction: TCP

server

client

Client/server socket interaction: TCP

server

client 

create socket,
port=**x**, for incoming request:

serverSocket = socket()

Client/server socket interaction: TCP

server

client 

create socket,
port=**x**, for incoming request:

serverSocket = socket()

wait for incoming
connection request

connectionSocket =
serverSocket.accept()

Client/server socket interaction: TCP

server

client

create socket,
port=**`x`**, for incoming request:

serverSocket = socket()

wait for incoming
connection request

connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **`hostid`**, port=**`x`**

clientSocket = socket()

Client/server socket interaction: TCP

server

client

create socket,
port=**x**, for incoming request:

serverSocket = socket()

wait for incoming
connection request

connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**

clientSocket = socket()

send request using
clientSocket

Client/server socket interaction: TCP

server

client

create socket,
port=**x**, for incoming request:

serverSocket = socket()

wait for incoming
connection request

connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**

clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

Client/server socket interaction: TCP

server

client

create socket,
port=**x**, for incoming request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

Client/server socket interaction: TCP

server

client

create socket,
port=**x**, for incoming request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

close
connectionSocket

Client/server socket interaction: TCP

server

client

create socket,
port=**x**, for incoming request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

Networking system calls (some arguments omitted)!

socket(domain, type) — Allocates a new socket

bind(socket, address) — Binds a socket to a specific address (IP/port)

listen(socket) — Tells the OS to accept incoming connections

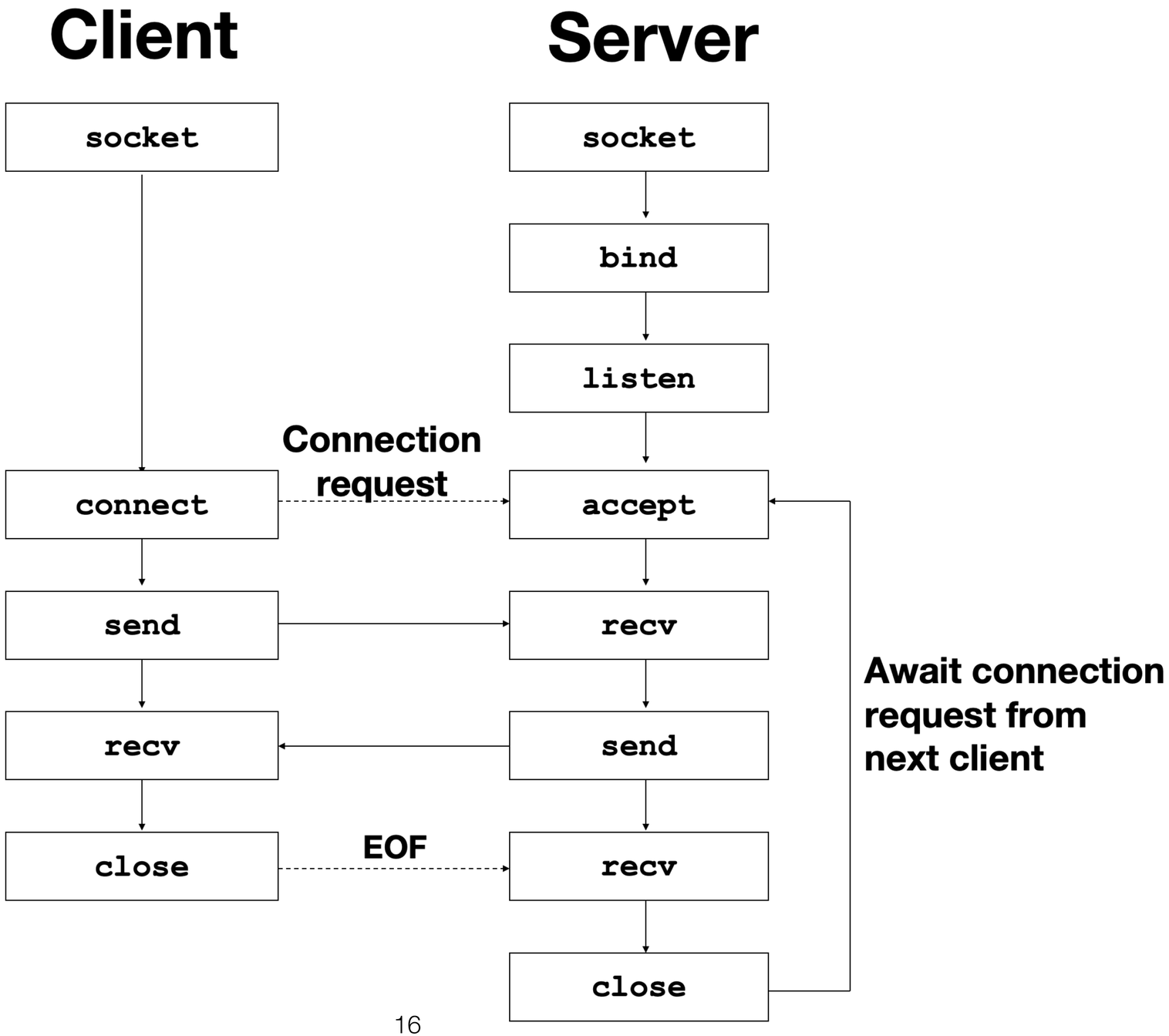
accept(socket, remote_address) — Wait for a connection on the socket

connect(socket, address) — Connect to the specified address (IP/port)

send(socket, data) — Sends data

recv(socket, data) — Receives data

close(socket) — Close the connection


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

TCP Sockets in C: socket()

```
int sockfd = socket(domain, type, protocol)
```

- `domain`: integer that specifies communication domain (i.e., type of address)
- `type`: TCP or UDP
- `protocol`: usually set to 0 (default)
- Returns a file descriptor

NOTE:

- Socket just creates the interface; it does NOT specify where the data is coming from or where it's going to

TCP Sockets in C: bind()

```
int status = bind(sockfd, &addrport, size)
```

- `sockfd`: descriptor returned by socket()
- `addrport`: struct containing address information
- `size`: size of the addrport struct
- Returns a status integer

Bind assigns an address to a socket

- Sets the IP address and reserves a port for the socket

TCP Sockets in C: connect()

```
int status = connect(sockfd, &addrport, size)
```

- `sockfd`: descriptor returned by socket()
- `addrport`: struct containing address information **of server to connect to**
- `size`: size of the addrport struct
- Returns a status integer

Client establishes connection with server using connect()

- connect() is **blocking** - program will wait until connection is either successfully established or failed

TCP Sockets in C: accept()

```
int s = accept(sockfd, &addrport, size)
```

- `sockfd`: descriptor returned by socket()
- `addrport`: struct containing address information **of client to connect to**
- `size`: size of the addrport struct
- Returns a socket to use for data transfer with client

Client establishes connection with server using connect()

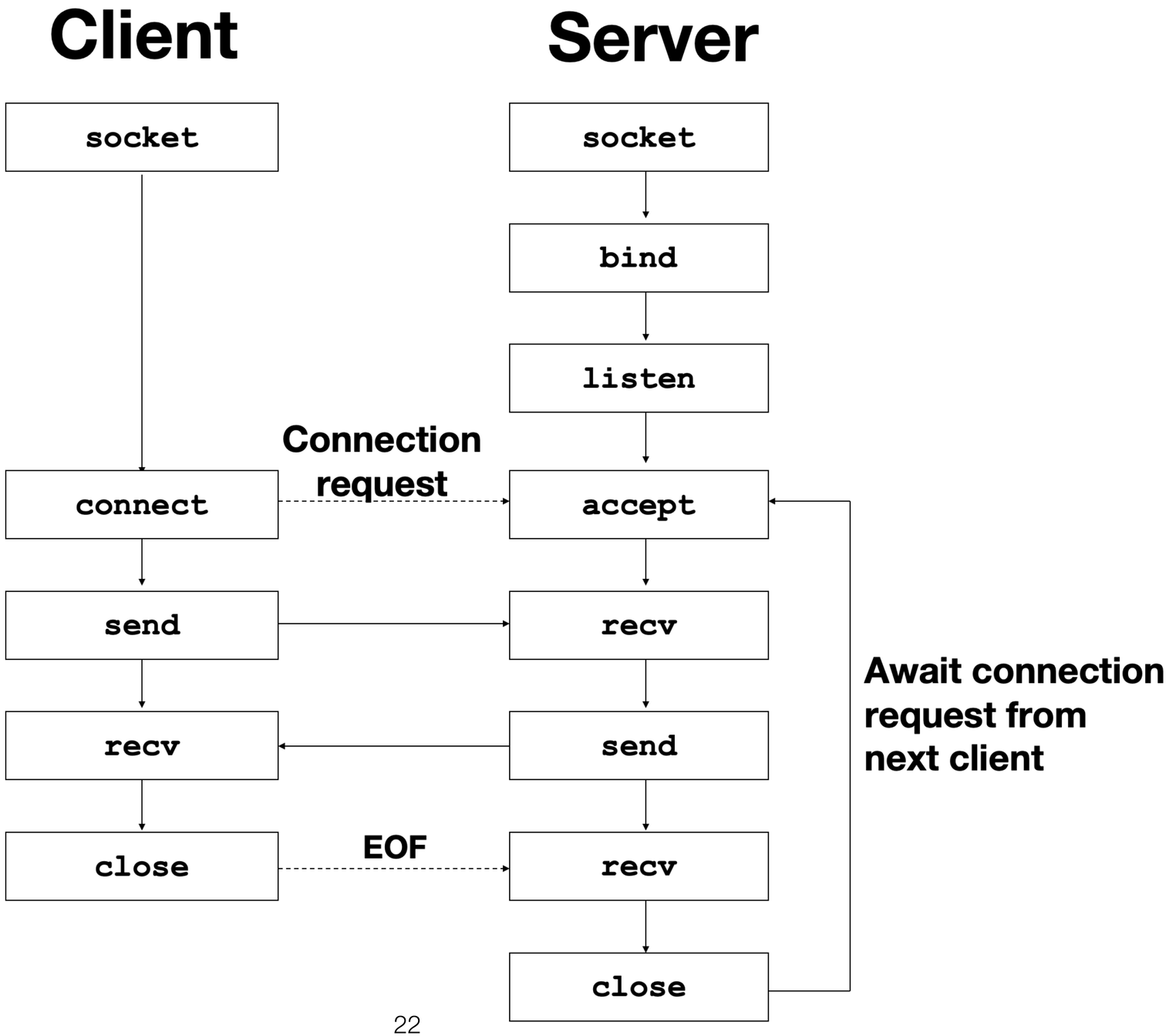
- accept() is **blocking** - program will wait until for connection before continuing

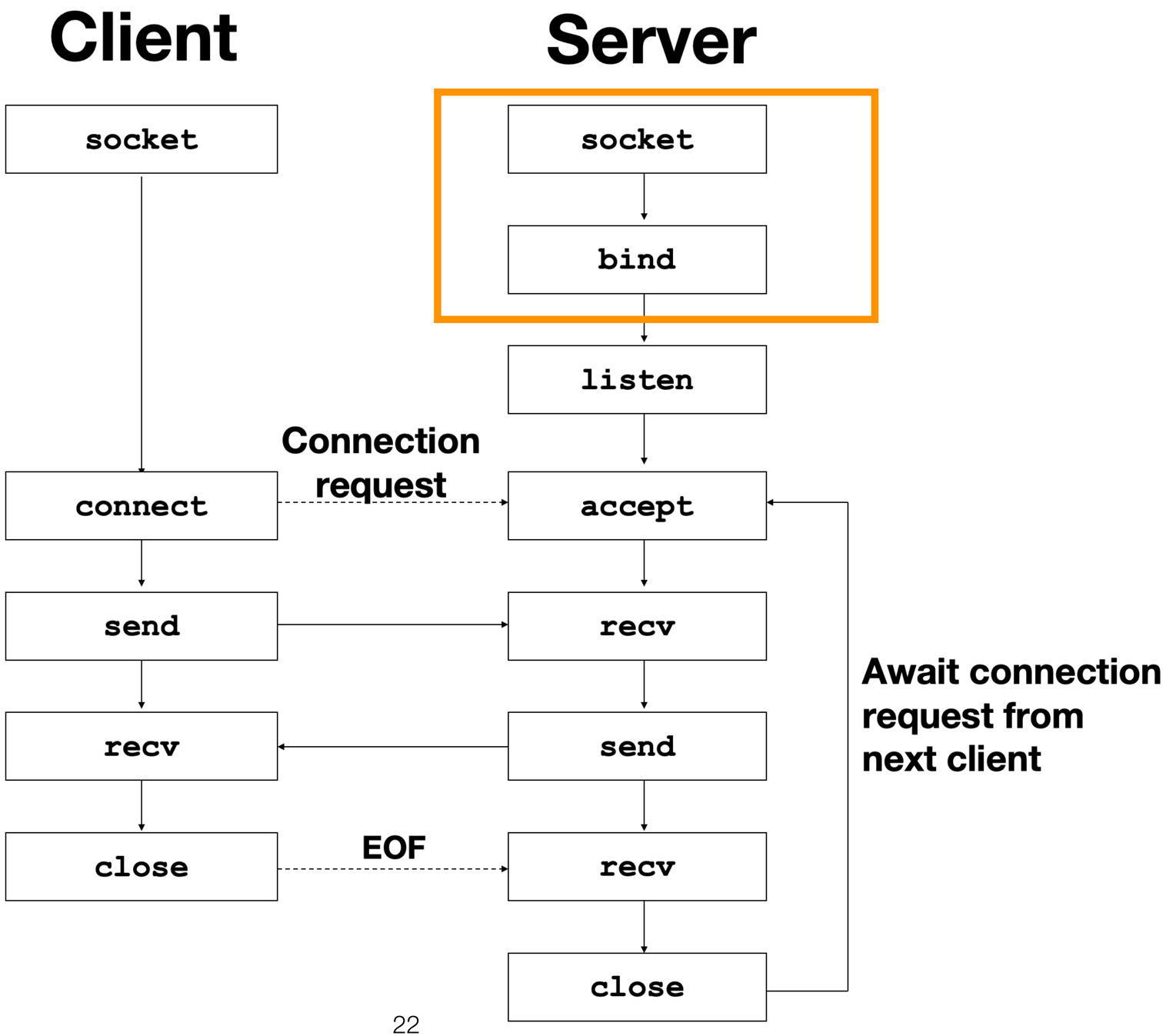
Calling accept returns a new socket because

A. We can’t write to a bound port

B. Using multiple sockets is faster

C. We can continue to listen on the old socket while we use the new socket


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

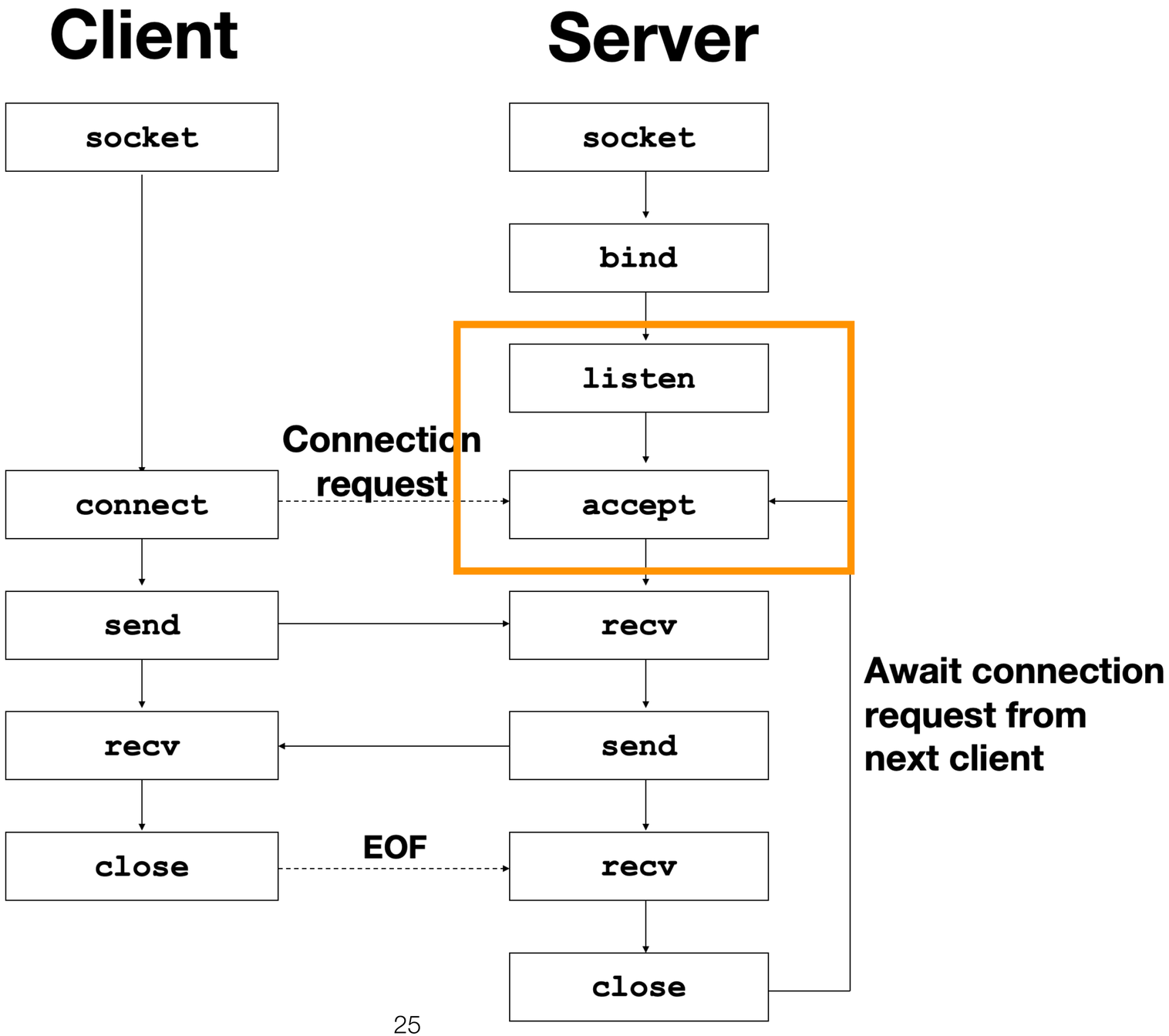
TCP Server

```rust
// main.rs
use std::net::TcpListener;

fn main() {




}
```

TCP Server

```
// main.rs
use std::net::TcpListener;

fn main() {
    // create socket, bind to address
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();




}
```

Creates a new socket and binds it to an address in form “IP_addr:port_no”


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

TCP Server

```
// main.rs
use std::net::TcpListener;

fn main() {
    // create socket, bind to address
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    // listen for incoming client connections
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        println!("Connection established!");
    }
}
```

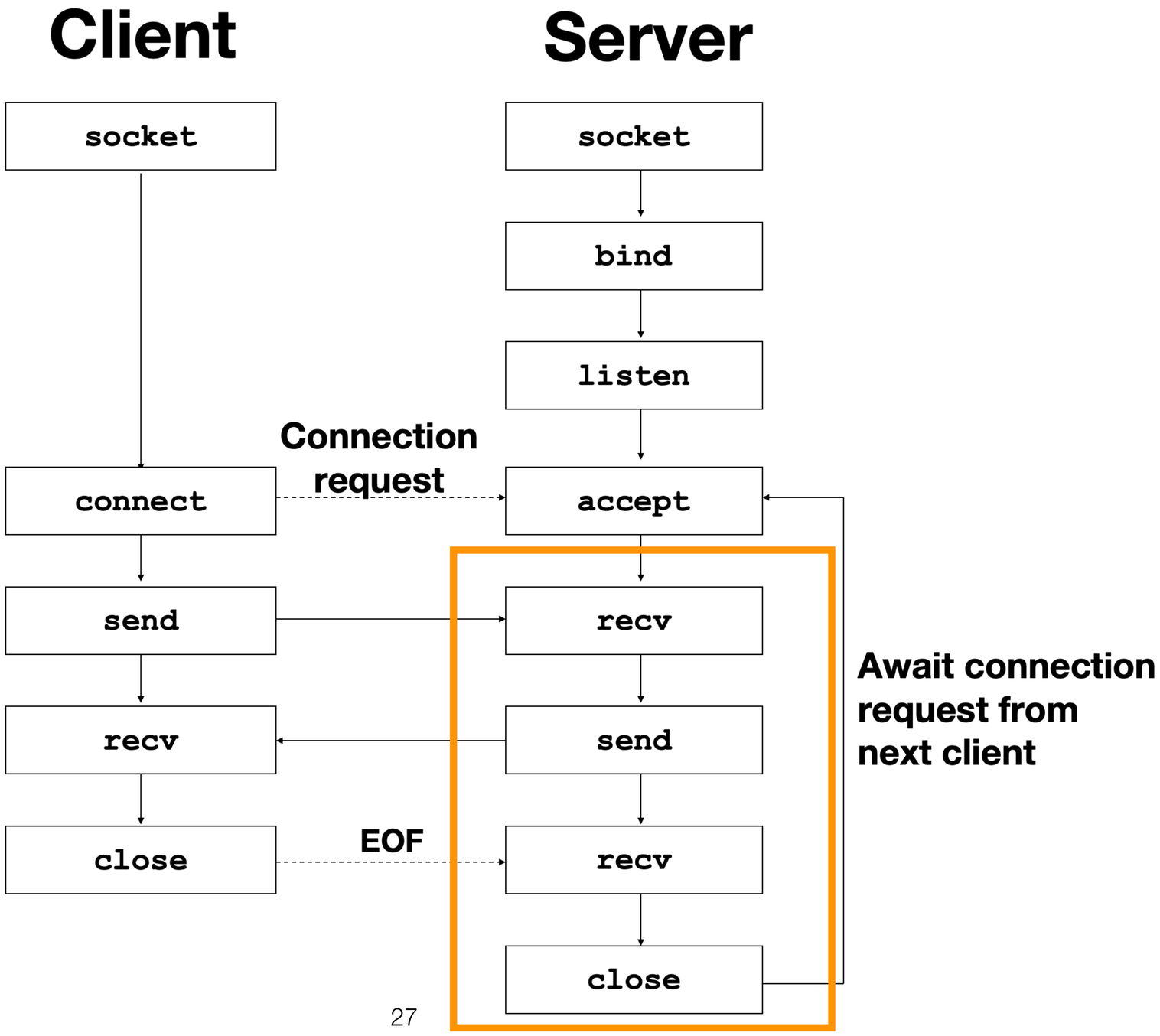
Creates a new socket and binds it to an address in form "IP_addr:port_no"

TCP Server

```
// main.rs
use std::net::TcpListener;

fn main() {
    // create socket, bind to address
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    // listen for incoming client connections
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        println!("Connection established!");
    }
}
```

Creates a new socket and binds it to an address in form "IP_addr:port_no"

Returns an iterator that gives us a sequence of streams


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

The server is connected, now let's handle client data

```
// main.rs
use std::net::TcpListener;

fn main() {
    // create socket, bind to address
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();

    // listen for incoming client connections
    for stream in listener.incoming() {
        let stream = stream.unwrap();
        println!("Connection established!");

        // receive client data
        handle_connection(stream);
    }
}
```

Creates a new socket and binds it to an address in form "IP_addr:port_no"

Returns an iterator that gives us a sequence of streams

The server is connected, now let's handle client data

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {









}
```

The server is connected, now let's handle client data

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {
    // reader for client data
    let buf_reader = BufReader::new(&stream);









}
```

The server is connected, now let's handle client data

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {
    // reader for client data
    let buf_reader = BufReader::new(&stream);
    // read in data
    let client_data: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap())
        .take_while(|line| !line.is_empty())
        .collect();
}
```

Socket Programming

Two types of sockets

- TCP: reliable, byte stream-oriented
- UDP: unreliable datagram

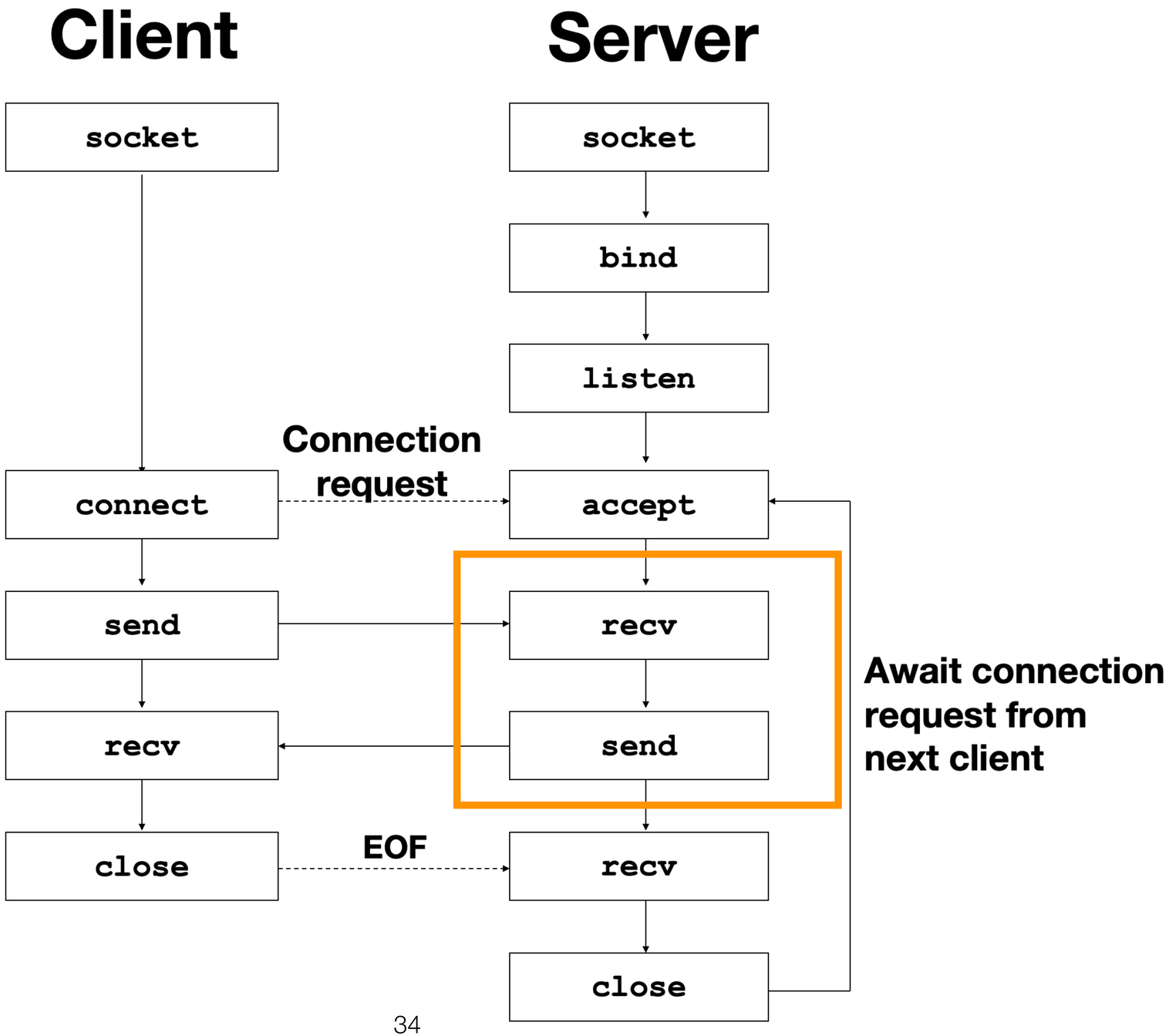
Application example: [we'll implement this!]

1. Client reads a line of characters (data) from its keyboard and sends data to server
2. Server receives the data and converts the characters to uppercase
3. Server sends modified data to client
4. Client receives modified data and displays line on its screen

Convert client data to all uppercase

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {
    // reader for client data
    let buf_reader = BufReader::new(&stream);
    // read in data
    let client_data: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap().to_uppercase())
        .take_while(|line| !line.is_empty())
        .collect();
}
```

Applies the `to_uppercase` method to every item in the vector

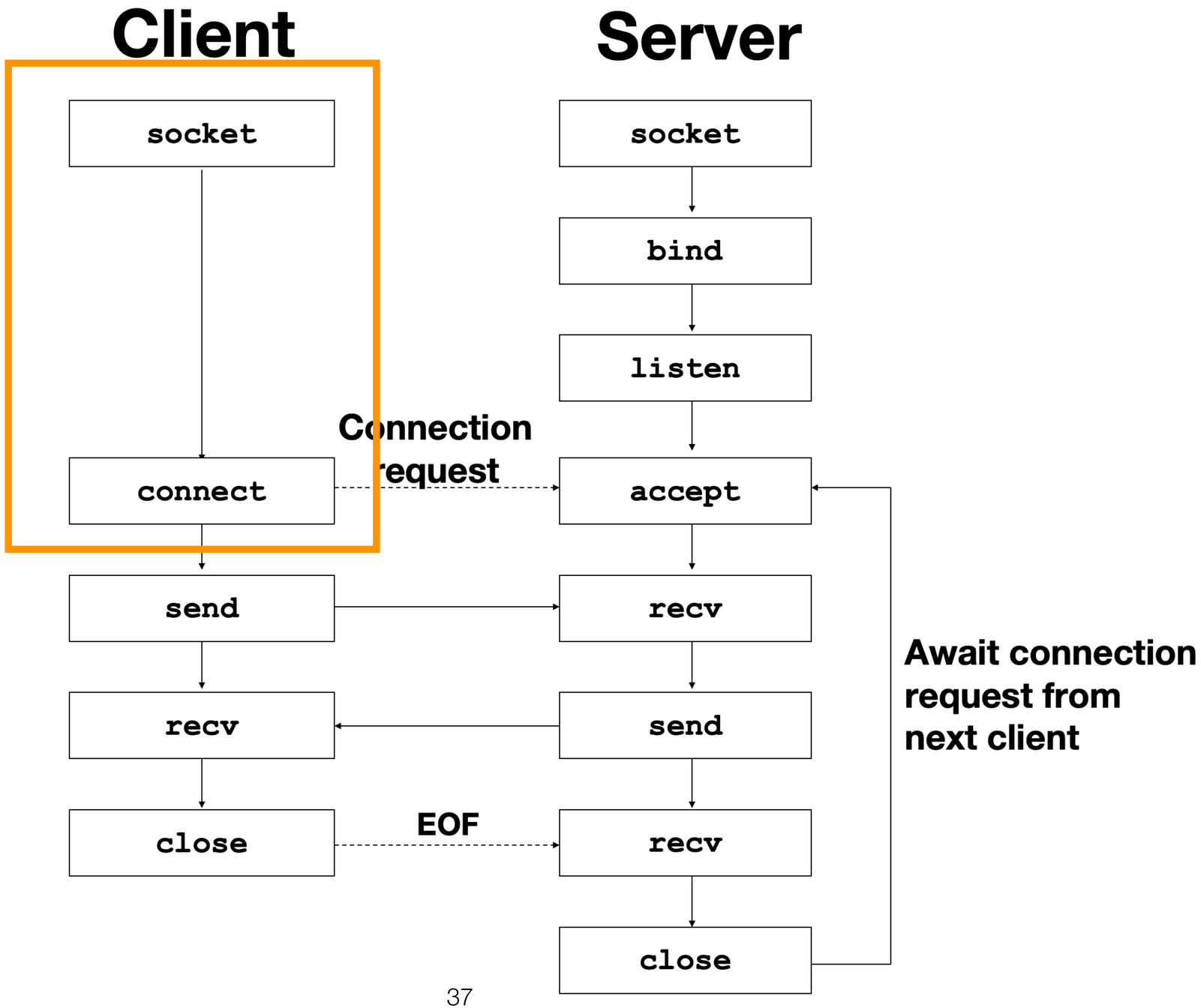

Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

Send the data to the client!

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {
    // reader for client data
    let buf_reader = BufReader::new(&stream);
    // read in data
    let client_data: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap().to_uppercase())
        .take_while(|line| !line.is_empty())
        .collect();
    // convert response data into byte (type &[u8])
    let response = …;

}
```

Send the data to the client!

```
// main.rs
use std::{
    io::{prelude::*, BufReader},
    net::{TcpListener, TcpStream}, };

fn handle_connection(mut stream: TcpStream) {
    // reader for client data
    let buf_reader = BufReader::new(&stream);
    // read in data
    let client_data: Vec<_> = buf_reader
        .lines()
        .map(|result| result.unwrap().to_uppercase())
        .take_while(|line| !line.is_empty())
        .collect();
    // convert response data into byte (type &[u8])
    let response = …;
    stream.write_all(response).unwrap();
}
```


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

Let's build the client application! We need to connect to the server, using `TcpStream::connect()`, which takes as input the address of the server to connect to. What's input should we use to connect to our server?

A. "127.0.0.1:8080"

B. "127.0.0.1:7878"

C. "127.0.0.1"

D. More than 1 of the above (which ones?)

Connect to the server

```
// main.rs (client-side)
use std::net::TcpStream;

fn main() {




}
```

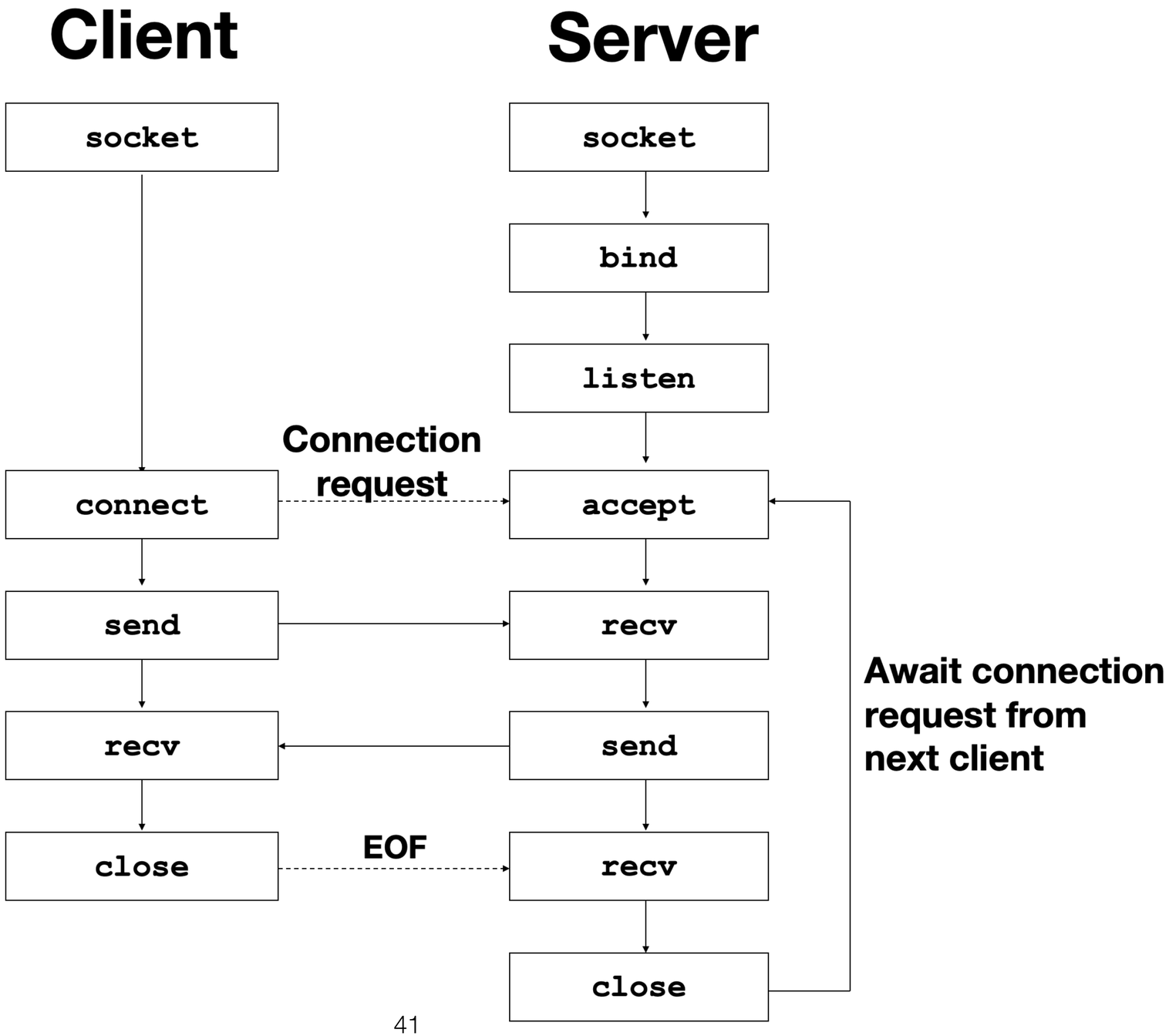
Connect to the server

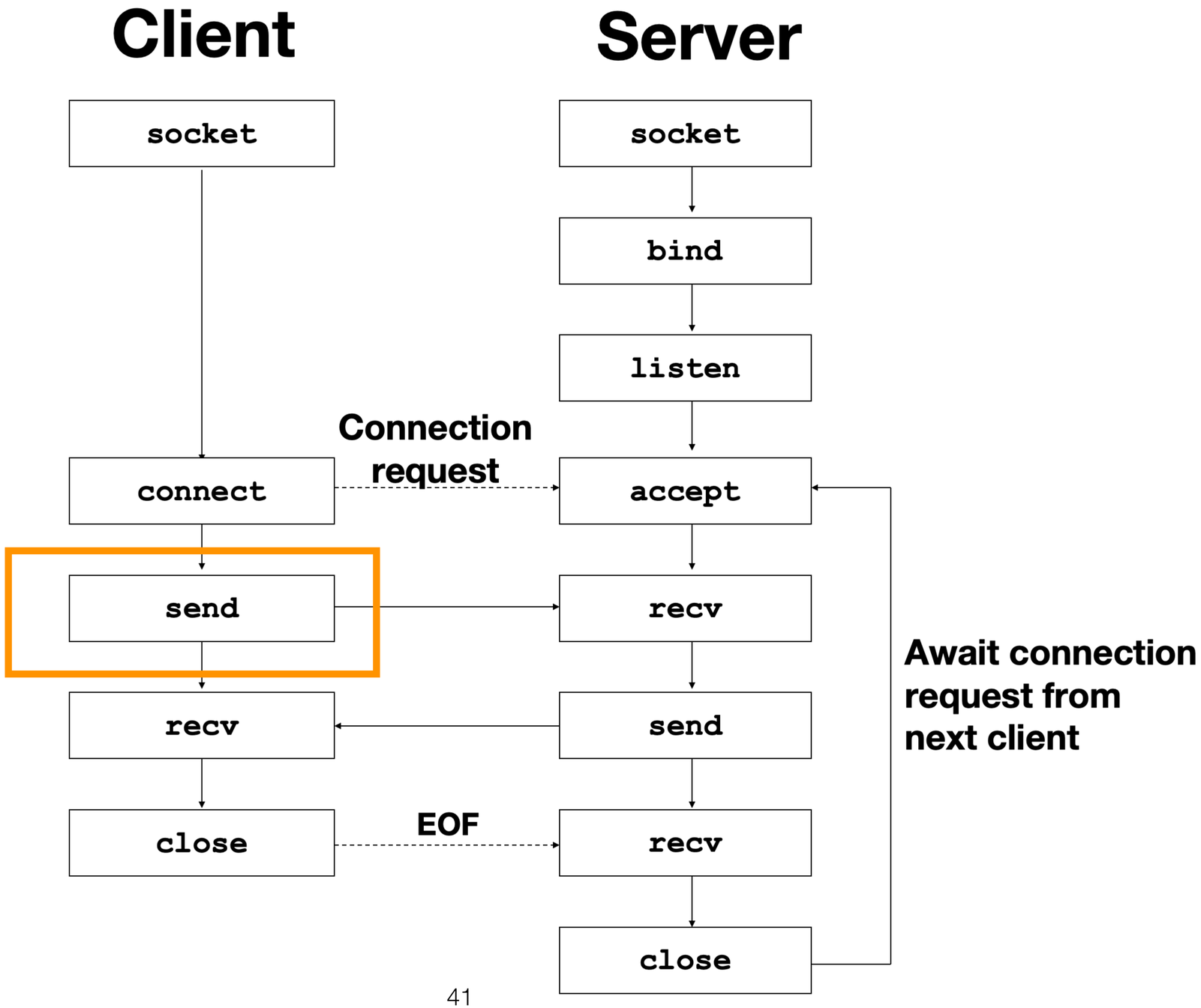
```
// main.rs (client-side)
use std::net::TcpStream;

fn main() {
    // connect to server address
    let mut stream = TcpStream::connect("127.0.0.1:7878").unwrap();









}
```

Connect to the server

```
// main.rs (client-side)
use std::net::TcpStream;

fn main() {
    // connect to server address
    let mut stream = TcpStream::connect("127.0.0.1:7878").unwrap();




}
```

Stream needs to be mutable because reading/writing modifies its internal state


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client


Client
Server
socket
socket
bind
listen
Connection request
connect
accept
send
recv
recv
send
close
EOF
recv
close
Await connection request from next client

Connect to the server

```
// main.rs (client-side)
use std::net::TcpStream;

fn main() {
    // connect to server address
    let mut stream = TcpStream::connect("127.0.0.1:7878").unwrap();
    // send a message
    let message = b"Hello, world!";
    stream.write_all(message);
    // read in server response
    …
}
```

Stream needs to be mutable because reading/writing modifies its internal state