Socket Programming

Shadman Kolahzary

Kurdistan University

Lecture Today

- Motivation for sockets
- What’s in a socket?
- Working with socket
- Concurrent network applications
- Project 1

Programmer?

Python?

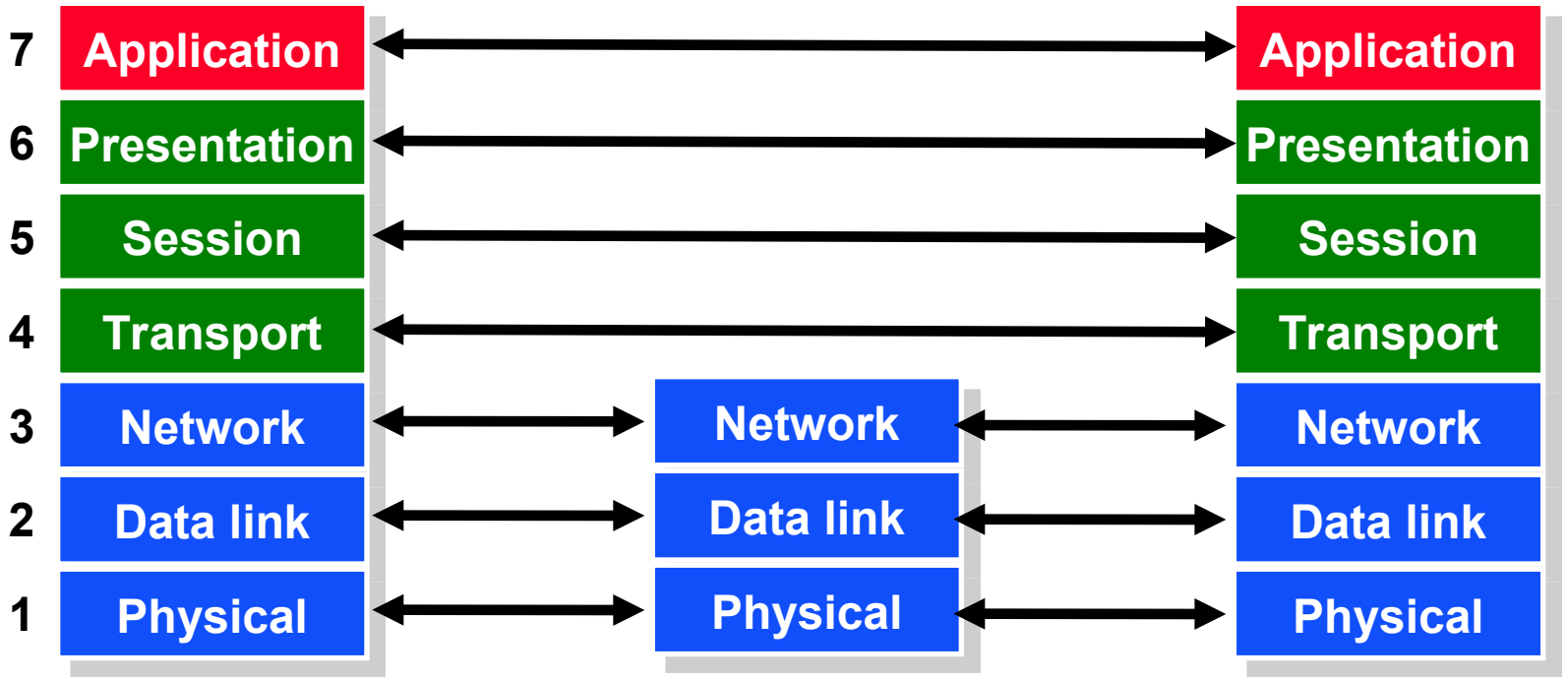
JS?

Socket?

Web?

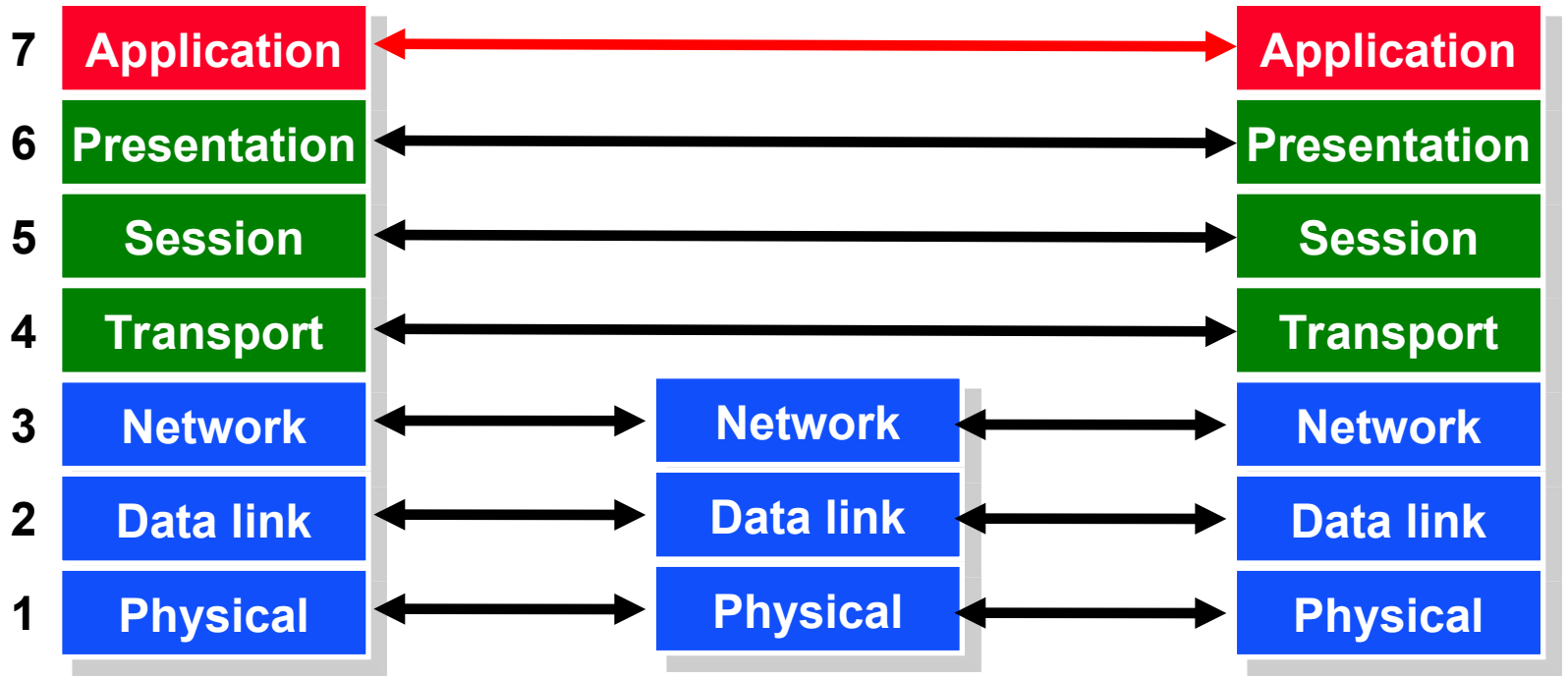
RFC?

Network Layering



Network Layering

- Why layering?



Layering Makes it Easier

- Application programmer
  - Doesn’t need to send IP packets
  - Doesn’t need to send Ethernet frames
  - Doesn’t need to know how TCP implements reliability
- Only need a way to pass the data down
  - Socket is the API to access transport layer functions

What Lower Layer Need to Know?

- We pass the data down. What else does the lower layer need to know?

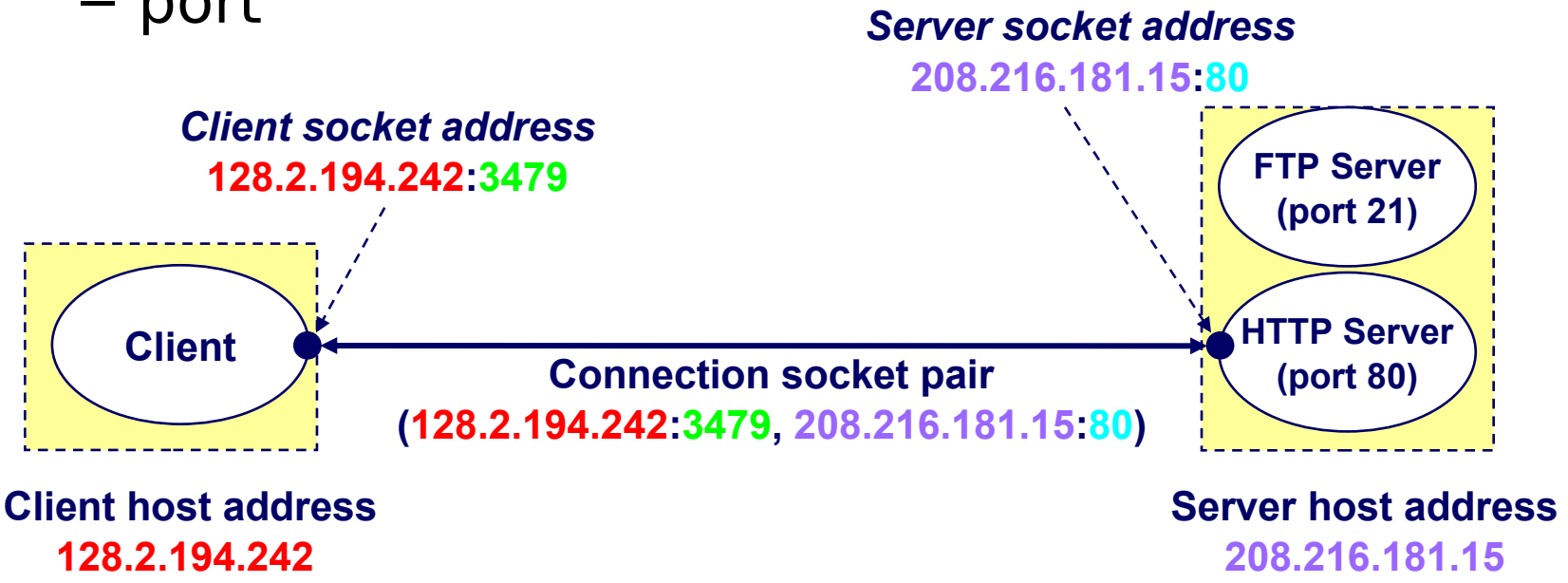
What Lower Layer Need to Know?

- We pass the data down. What else does the lower layer need to know?

- How to identify the destination process?
  - Where to send the data? (Addressing)
  - What process gets the data when it is there? (Multiplexing)

Identify the Destination

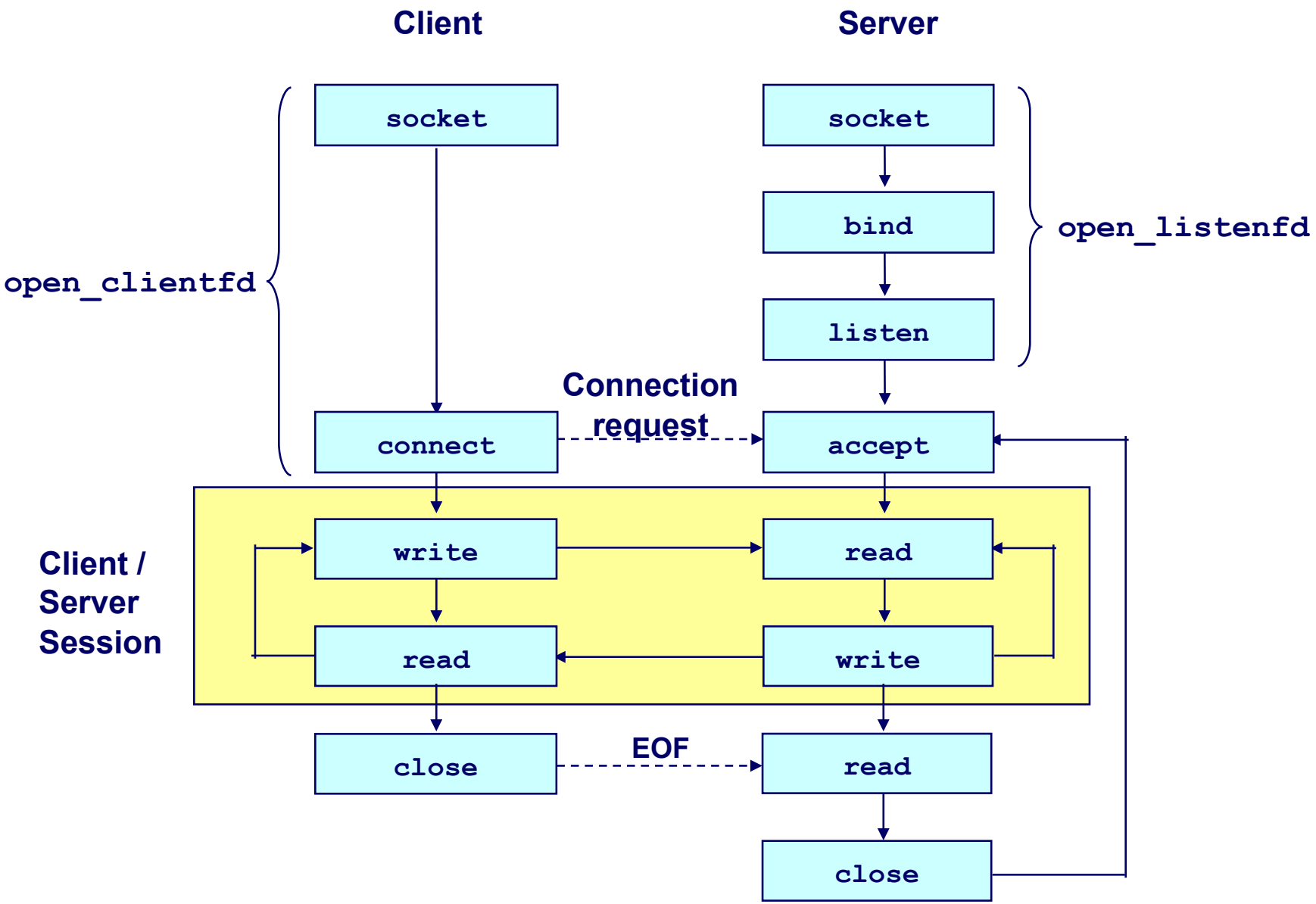
- Addressing
  - IP address
  - hostname (resolve to IP address via DNS)
- Multiplexing
  - port



Sockets

- How to use sockets
  - Setup socket
    - Where is the remote machine (IP address, hostname)
    - What service gets the data (port)
  - Send and Receive
    - Designed just like any other I/O in unix
    - send -- write
    - recv -- read
  - Close the socket

Overview


Client
Server
socket
socket
bind
listen
open_listenfd
open_clientfd
Connection request
connect
accept
Client / Server Session
write
read
read
write
close
EOF
read
close

Step 1 – Setup Socket

- **Both client and server need to setup the socket**

server = socket.socket(address_family, protocol)

- *Address family*
  - AF_INET -- IPv4 (AF_INET6 for IPv6)
- *protocol*
  - SOCK_STREAM -- TCP
  - SOCK_DGRAM -- UDP
- For example:

- server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Step 2 (Server) - Binding

- **Only server need to bind**
  - *server.bind((ip, port))*
- *IP*
  - IP address to listen on
- *Port*
  - Port to listen on
- Example
  - server.bind(('localhost',5960))

Step 3 (Server) - Listen

- **Now we can listen**
  - *server.listen()*

Step 4 (Server) - Accept

- **Server must explicitly accept incoming connections**
  - *conn, addr = server.accept()*
- *Example:*

```
def handle_connection(conn, addr):
    print(f'New connection from: {addr}')

while True:
    conn, addr = server.accept()
    thread = threading.Thread(target=handle_connection, args= (conn, addr))
    thread.start()
```

Put Server Together

server-non-concurrent.py > ...

```python
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server.bind(('localhost',5960))

server.listen()
print('listening on port 5960...')

conn, addr = server.accept()

print(f'New connection from: {addr}')
conn.send('Welcome, You are connected to the server'.encode('utf-8'))

receiving = True
while receiving:
    msg = conn.recv(2048).decode('utf-8')
    if msg:
        print(f'Message received from {addr} -> {msg}')
    if msg == '/bye':
        print('bye signal received')
        receiving = False

conn.close()
print('End')
```

What about client?

- Client doesn’t need bind, listen, and accept
- **All client need to do is to connect**
  - *server.bind((ip, port))*
- For example,
  - *client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)*
  - *client.connect(('localhost',5960))*

Put Client Together

client.py > wait_for_receive

```python
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('localhost',5960))

def send(msg):
    encoded = msg.encode('utf-8')
    client.send(encoded)

def wait_for_receive():
    # TODO: use a loop to receive multiple messages
    message = client.recv(2048).decode('utf-8')
    print(f'Received message: {message}')

send('Hello World')
wait_for_receive()
send('Heya! I got your message!')
# or any other messages here

send('/bye')
client.close()

```

We Are Connected

- Server accepting connections and client connecting to servers

Send and receive data

- *client.recv(size).decode(encoding)*
- *conn.send(message.encode(encoding))*

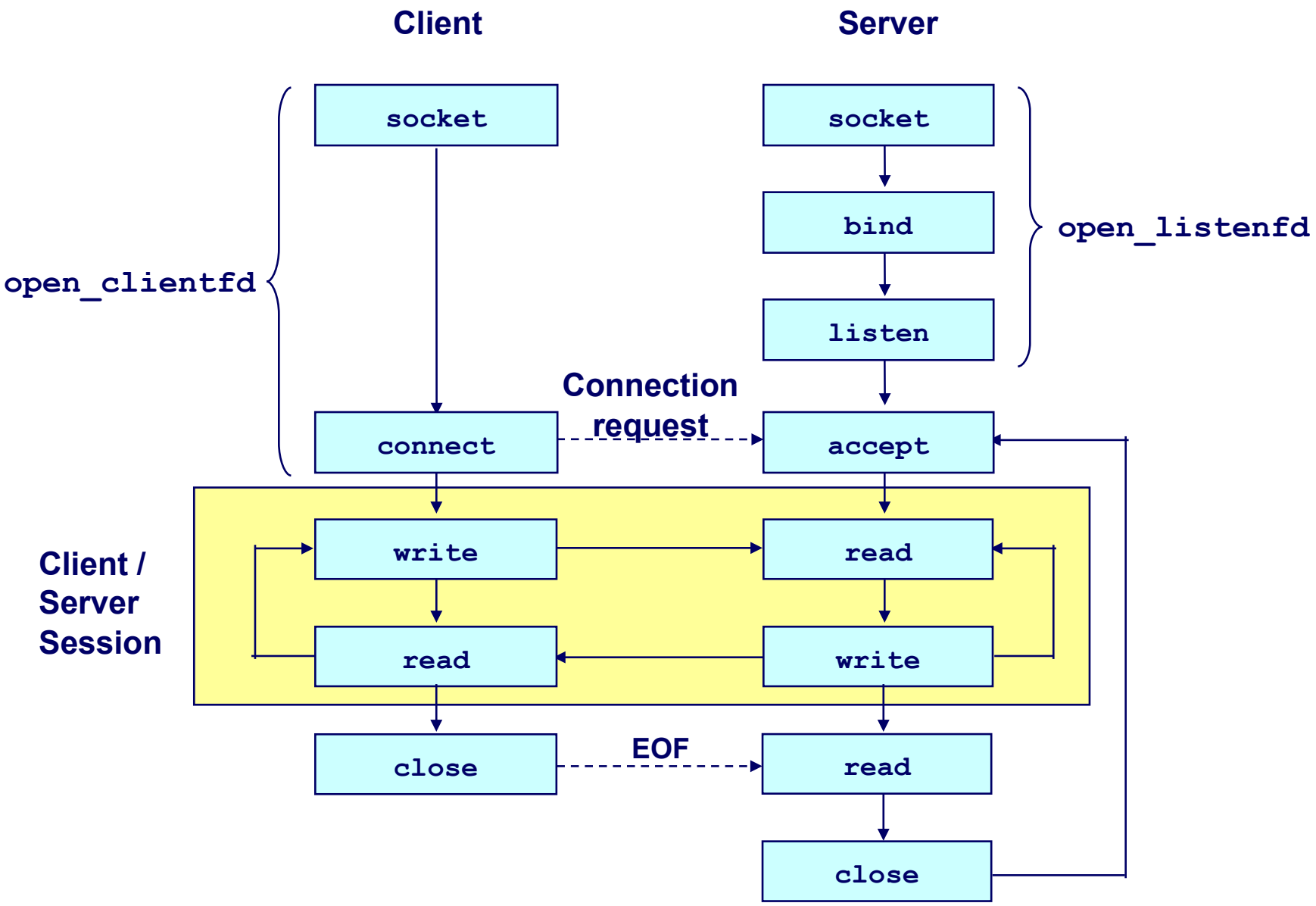
For example,

- *client.recv(2048).decode('utf-8')*
- *conn.send('Welcome, You are connected to the server'.encode('utf-8'))*

TCP Framing

- TCP does NOT guarantee message boundaries
  - IRC commands are terminated by a newline
  - But you may not get one at the end of read(), e.g.
    - One Send “Hello\n”
    - Multiple Receives “He”, “llo\n”
  - If you don’t get the entire line from one read(), use a buffer

Revisited

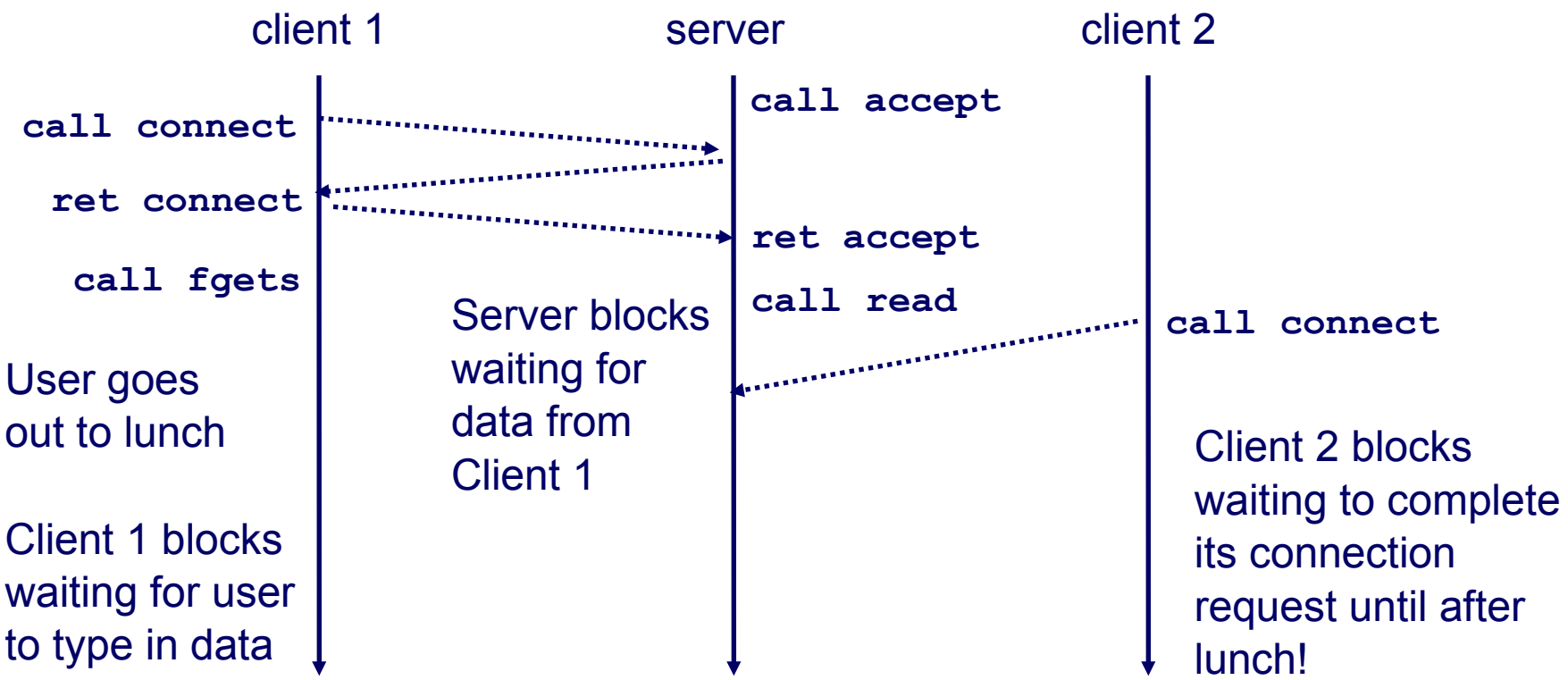

Client
Server
socket
socket
bind
listen
open_listenfd
open_clientfd
Connection request
connect
accept
Client / Server Session
write
read
read
write
close
EOF
read
close

Close the Socket

- Don’t forget to close the socket descriptor, like a file
  - *socket.close()*

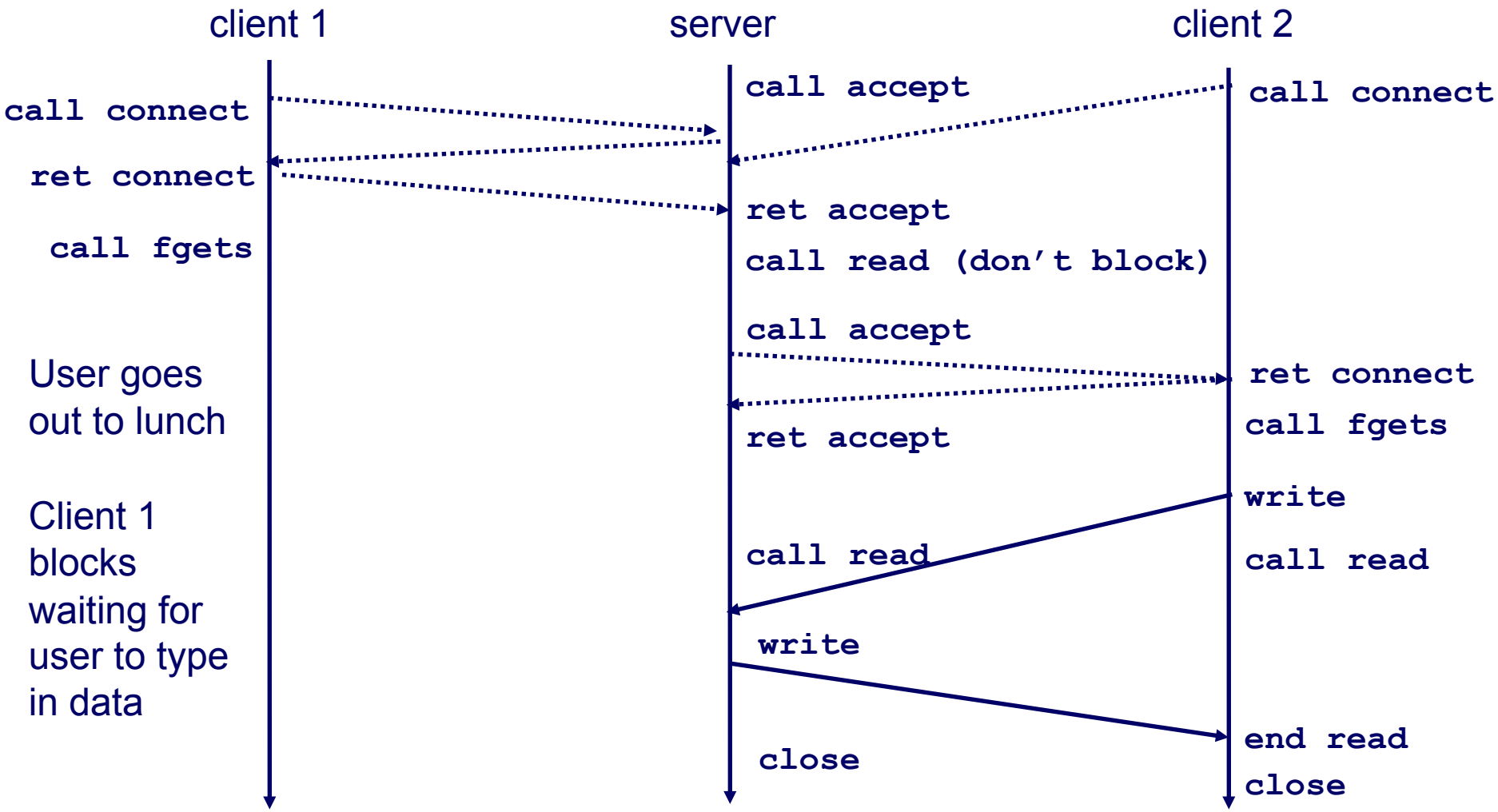
- Now server can loop around and accept a new connection when the old one finishes
- What’s wrong here?

Server Flaw



Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides

Concurrent Servers



Taken from D. Murray, R. Bryant, and G. Langale 15-441/213 slides

Concurrency

- Threading
  - Easier to understand
  - Race conditions increase complexity
- EventLoop (NodeJs, etc.)
  - Callbacks
  - Async/Await

The Server

```python
import socket
import threading

def handle_connection(conn, addr):
    print(f'New connection from: {addr}')
    print(f'Active connections: {threading.active_count() - 1}')
    conn.send('Welcome, You are connected to the server'.encode('utf-8'))
    receiving = True
    while receiving:
        msg = conn.recv(2048).decode('utf-8').rstrip()
        if msg:
            print(f'Message received from {addr} -> {msg}')
        if msg == '/bye':
            print('bye signal received')
            receiving = False
    conn.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('localhost',5960))
server.listen()
print('listening on port 5960...')

while True:
    conn, addr = server.accept()
    thread = threading.Thread(target= handle_connection, args= (conn, addr))
    thread.start()

```

What about checking clients?

- The main loop only tests for incoming connections
  - There are other reasons the server wakes up
  - Clients are sending data, pending data to write to buffer, clients closing connections, etc.
- Store all client file descriptors
  - in pool
- Keep the while(1) loop thin
  - Delegate to functions
- Come up with your own design

Summary

- Sockets
  - socket setup
  - I/O
  - close
- Client: socket()---------->connect()->I/O->close()
- Server: socket()->bind()->listen()->accept()--->I/O->close()

- Concurrency
  - threading

About Project 1

- Chat
  - Checkpoint 1: Single Client Chat (You got it!)
    - Inputs from server console should be printed on client console
    - Inputs from client console should be printed on server console
  - Checkpoint 2: Multi-User Chat (You’re a great student)
    - First take own username from user
    - Then take target username
    - Create a dictionary of username → connection on server
    - Users should be able to start a bi-directional chat together
  - Checkpoint 3: Users list (You’re a pro programmer!)
    - Create a specific request which returns list of online usernames
    - Create a UI to show users list and start a chat after clicking on a username

Suggestions

- Start early!
  - Work ahead of checkpoints
- Read the documentation
- Try to implement it in another language
- Email (shadman.ko@gmail.com)