# Network Programming
## Socket Programming

Minchen Yu
SDS@CUHK-SZ
Spring 2026

香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen
SCHOOL OF DATA SCIENCE
數據科學學院

# Transport layer review

- UDP
  - Connectionless; datagrams
- TCP
  - Connection-oriented; bytestreams; sliding window/flow control; congestion control
- Programming interface: Socket

# Socket

- Simple abstraction to use the network

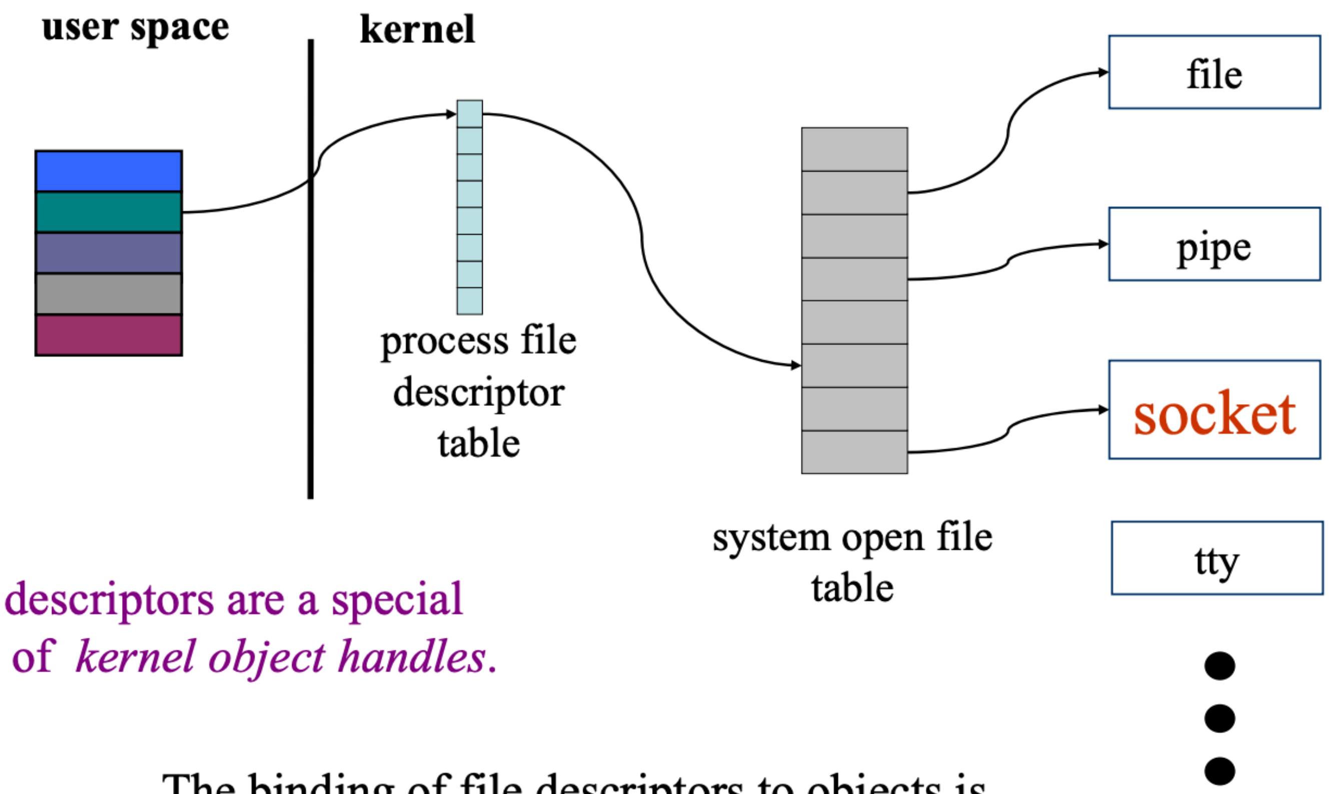
- Set of system-level functions used in conjunction with Unix I/O to build network applications

# How to use Socket

- Simple abstraction to use the network
  - 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

For an application, a socket is a file descriptor

# UNIX file descriptor



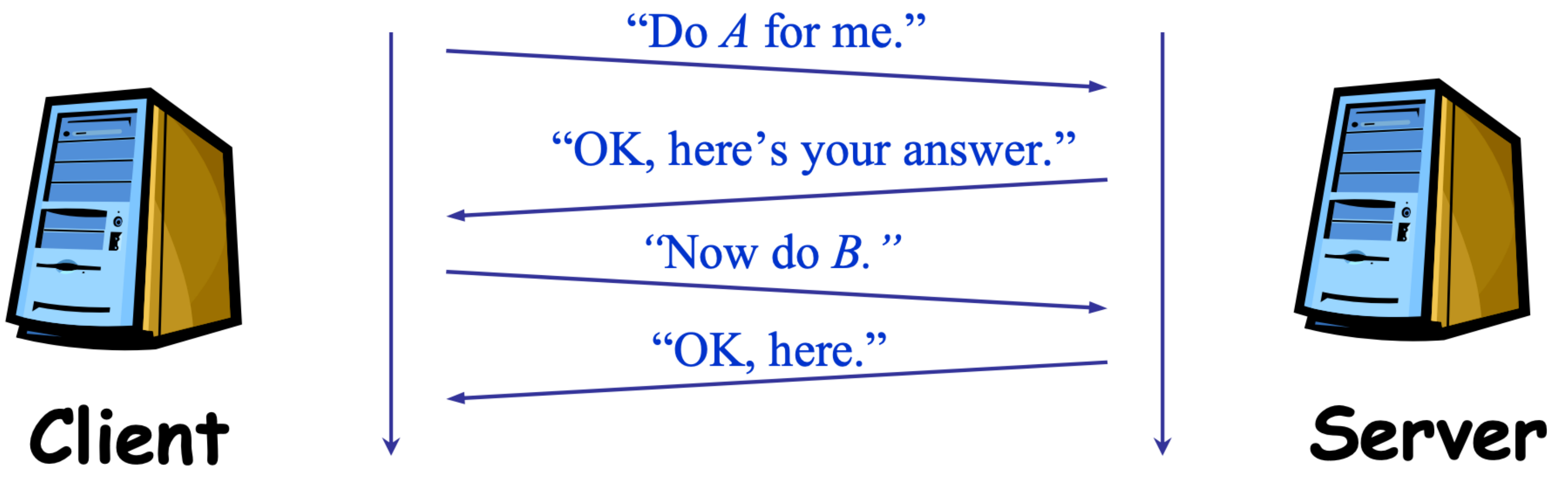

File descriptors are a special case of *kernel object handles*.

The binding of file descriptors to objects is specific to each process, like the virtual translations in the virtual address space.

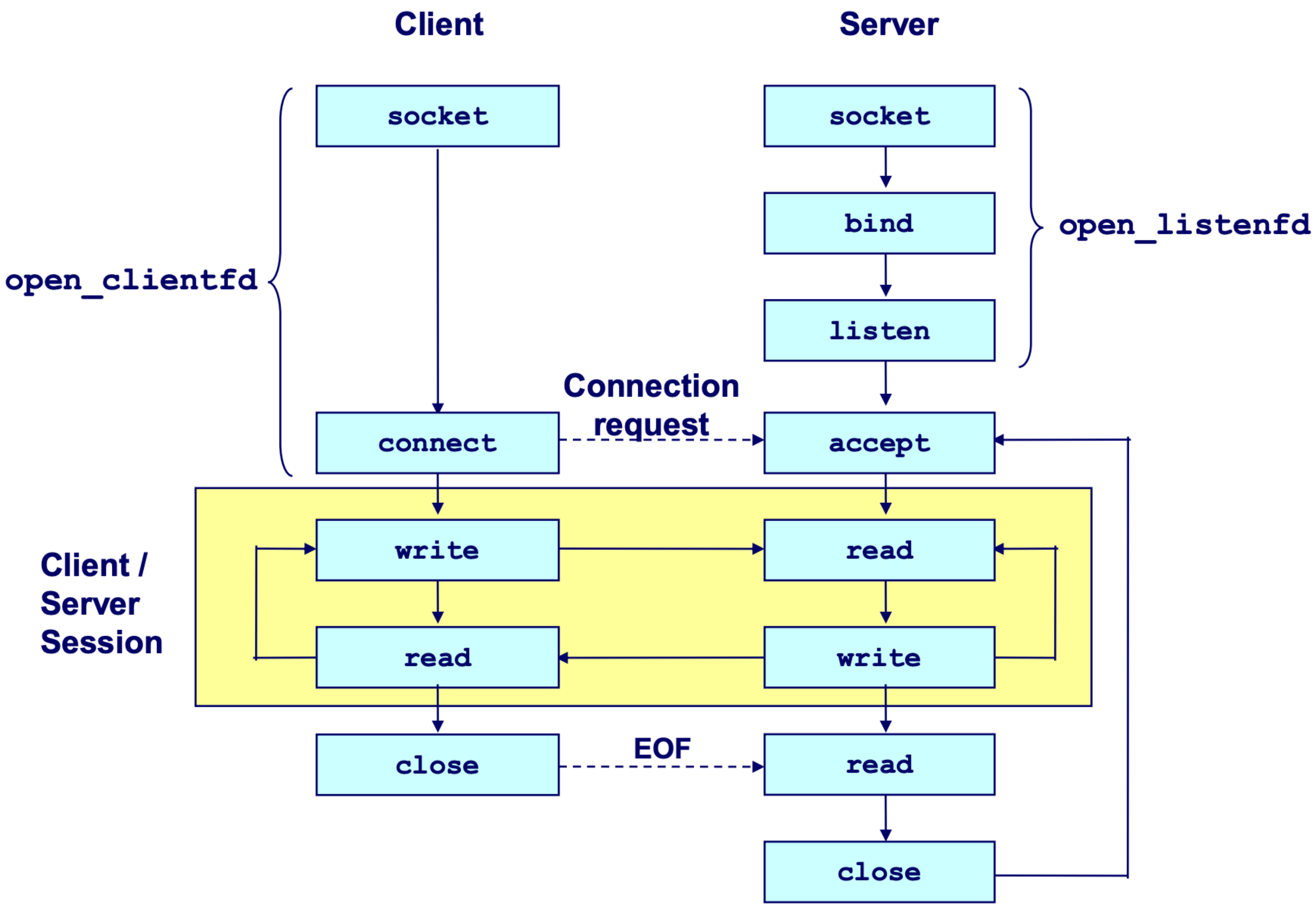
Disclaimer: this drawing is oversimplified.

# Client/Server model

- Request/response paradigm



# Client/Server model



# Setup Socket – create

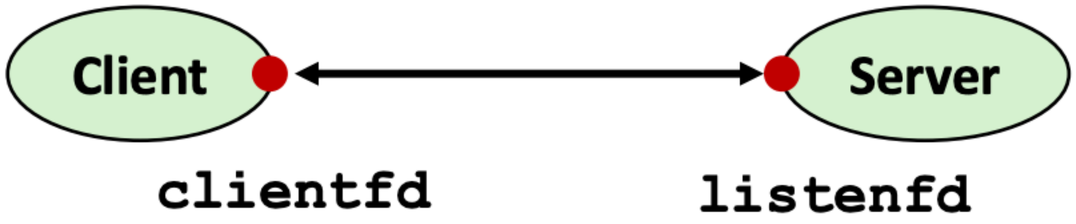
- Both client and server need to setup the socket

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

- *domain*
    - *AF_INET -- IPv4 (AF_INET6 for IPv6)*
- *type*
    - *SOCK_STREAM – TCP*
    - *SOCK_DGRAM – UDP*
- *protocol*
    - *0*

# Setup Socket – create

- Example




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

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

Indicates that we are using 32-bit IPV4 addresses

Indicates that the socket will be the end point of a reliable (TCP) connection

# Setup Socket – bind

- A server uses *bind* to ask the kernel to associate the server's socket address with a socket descriptor

```
int bind(int sockfd, SA *addr, socklen_t addrlen);
```

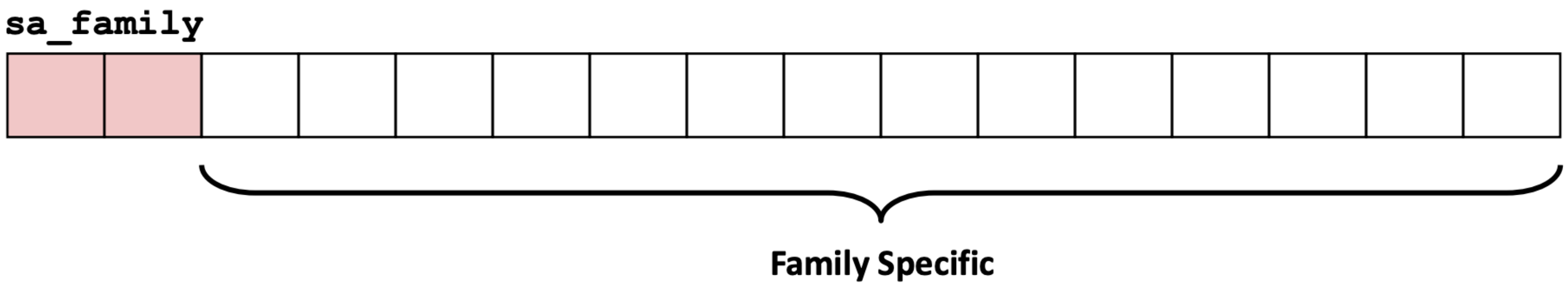
**Our convention:** `typedef struct sockaddr SA;`

- Read bytes that arrive on the connection whose endpoint is *addr* by reading from descriptor *sockfd*
- Similarly, writes to *sockfd* are transferred along connection whose endpoint is *addr*

# Socket address

- Generic socket address
  - For address arguments to *bind*, *connect* and *accept*
  - Necessary only because C did not have generic (*void **) pointers when the sockets interface was designed

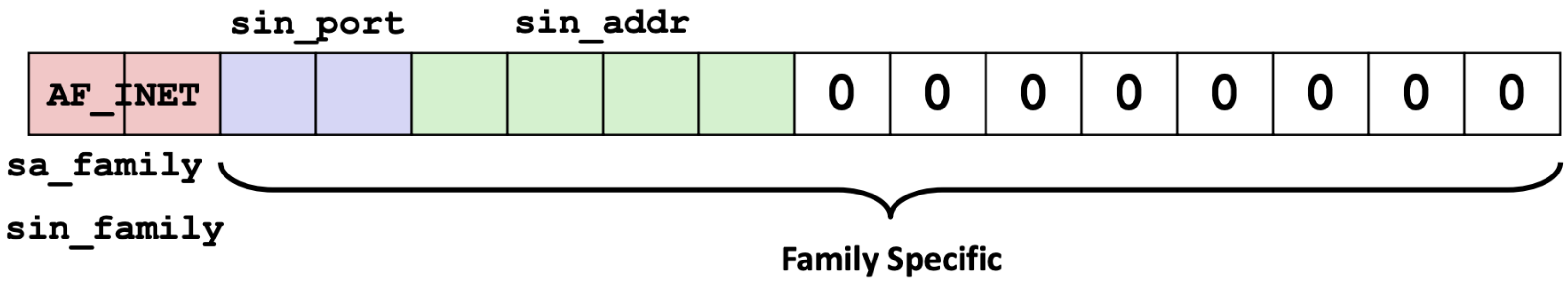
```c
struct sockaddr {
  uint16_t  sa_family;    /* Protocol family */
  char      sa_data[14];  /* Address data  */
};
```



# Socket address

- Internet (IPv4) specific socket address
  - Must cast (*struct sockaddr_in **) to (*struct sockaddr **) for functions that take socket address arguments

```
struct sockaddr_in  {
  uint16_t        sin_family;  /* Protocol family (always AF_INET) */
  uint16_t        sin_port;    /* Port num in network byte order */
  struct in_addr  sin_addr;    /* IP addr in network byte order */
  unsigned char   sin_zero[8]; /* Pad to sizeof(struct sockaddr) */
};
```



# Socket address config

- Byte ordering in network address
  - Network order is big-endian; host order can be big- or little-endian (x86 is little-endian)
  - Conversion
    - Port, addresses
    - *htons(), htonl():* host to network short/long

```
struct sockaddr_in serv_addr;
// ...

// Zero out the structure
memset(&serv_addr, 0, sizeof(serv_addr));

// Set the fields in the serv_addr struct
serv_addr.sin_family = AF_INET;         // Set the family to IPv4
serv_addr.sin_addr.s_addr = INADDR_ANY; // Bind to all interfaces
serv_addr.sin_port = htons(8080);       // Set the port number, converting it to network byte order
```

# Setup Socket – listen

- Kernel assumes that descriptor from socket function is an active socket that will be on the client end
- A server calls the listen function to tell the kernel that a descriptor will be used by a server rather than a client
- Converts *sockfd* from an active socket to a listening socket that can accept connection requests from clients

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

***backlog*** is a hint about the number of outstanding connection requests that the kernel should queue up before starting to refuse requests (128-ish by default)

# Setup Socket – accept

- Servers wait for connection requests from clients by calling accept

```
int accept(int listenfd, SA *addr, int *addrlen);
```

- Waits for connection request to arrive on the connection bound to *listenfd*, then fills in client's socket address in *addr* and size of the socket address in *addrlen*

- Returns a connected descriptor *connfd* that can be used to communicate with the client via Unix I/O routines.

# Setup Socket – connect

- A client establishes a connection with a server by calling connect

```
int connect(int clientfd, SA *addr, socklen_t addrlen);
```

- Attempts to establish a connection with server at socket address *addr*

- If successful, then *clientfd* is now ready for reading and writing.

# Server socket config

- Recall the network byte order
  - Convert port with ***htons()***
  - Convert string SERVER_IP (e.g., ”127.0.0.1”) with *inet_pton()*
    - *int inet _pton(int af, const char *src, void *dst)*

```
// Set the fields in the server_addr struct
server_addr.sin_family = AF_INET; // Set the family to IPv4
server_addr.sin_port = htons(SERVER_PORT); // Set the port number, converting it to network byte order

// Convert the IP address from text to binary form
if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
// error handling
}
```

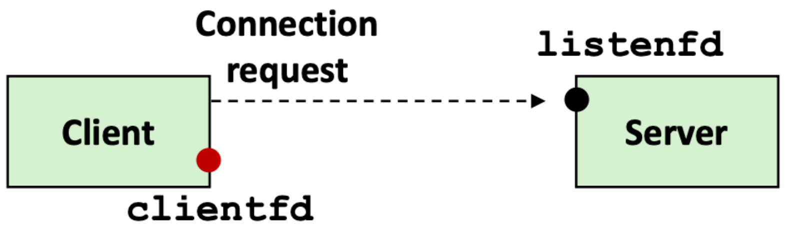
***Address family***

***Pointer to IP string***

***Pointer to the field of IP address***

# Connect/Accept


*1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`*




*2. Client makes connection request by calling and blocking in `connect`*

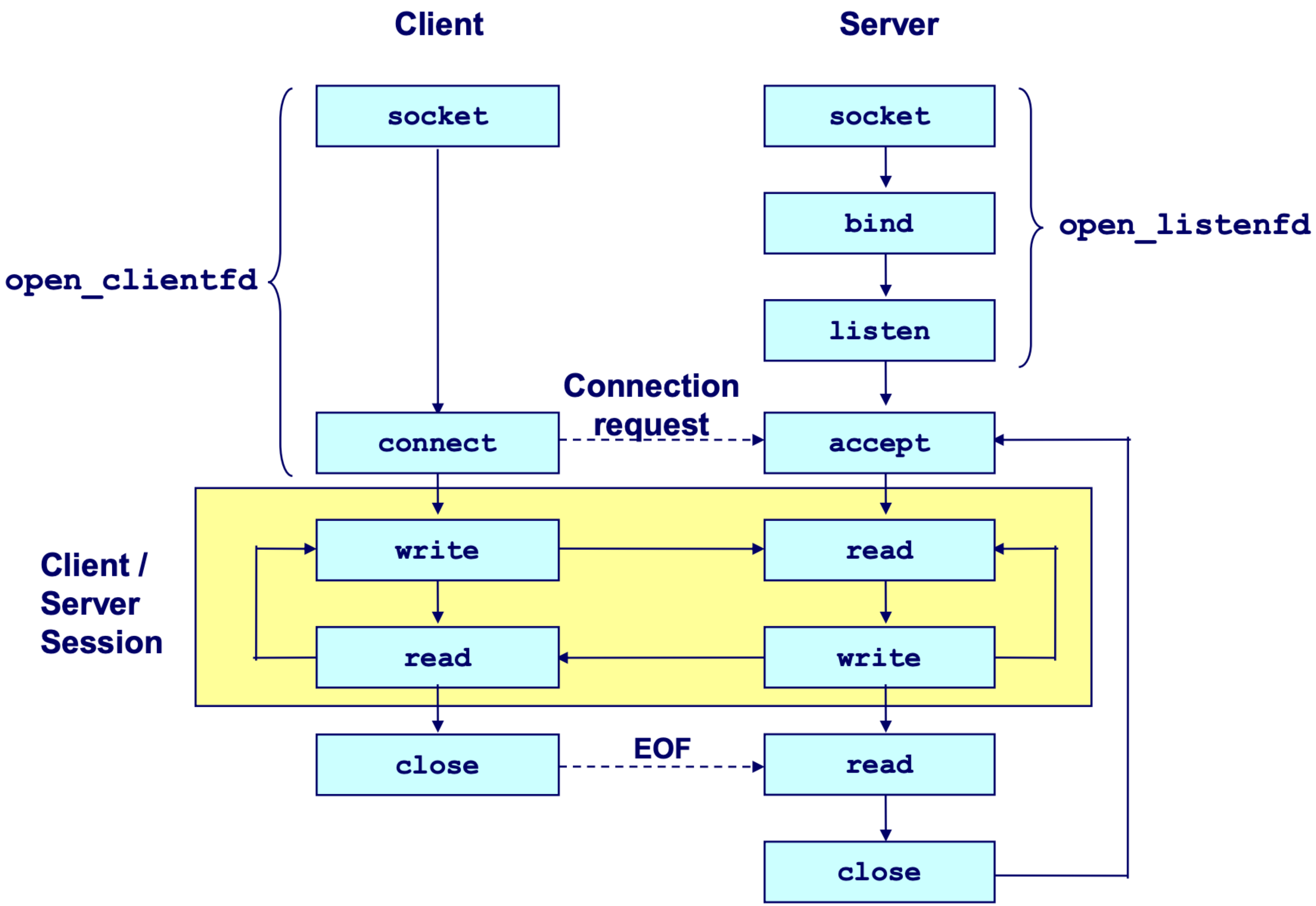

*3. Server returns `connfd` from `accept`. Client returns from `connect`. Connection is now established between `clientfd` and `connfd`*

# Connected vs. listening descriptors

- Listening descriptor
  - End point for client connection requests
  - Created once and exists for lifetime of the server
- Connected descriptor
  - End point of the connection between client and server
  - A new descriptor is created each time the server accepts a connection request from a client
  - Exists only as long as it takes to service client

Allows for concurrent servers that can communicate over many client connections simultaneously

# Client/Server model



# Send and receive

- Send and receive data after connection established
  - *ssize_t read(int fd, void *buf, size_t len);*
  - *ssize_t write(int fd, const void *buf, size_t len);*

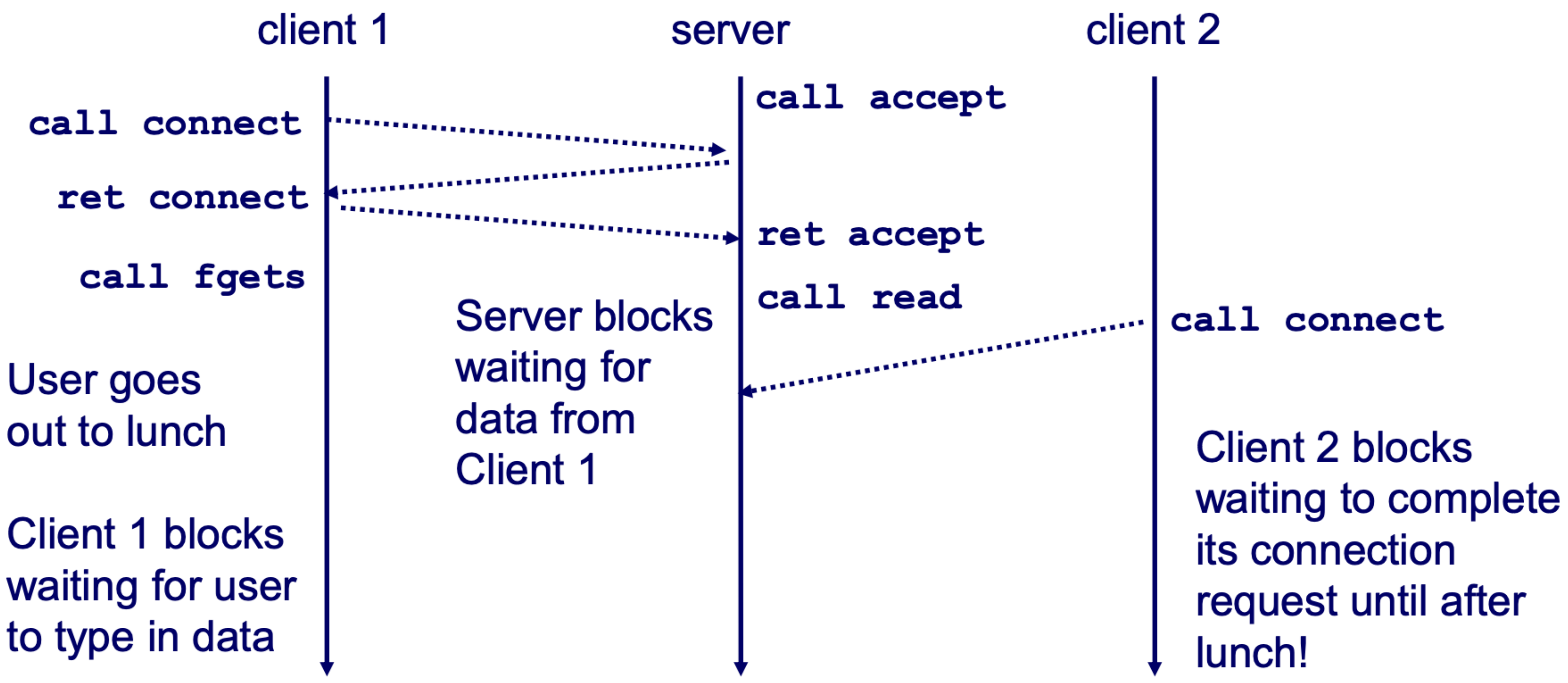
```
// Read data from the connected socket
ssize_t bytes_read = read(new_socket, buffer, 1024);
printf("Received message from client: %s\n", buffer);

// Send data to the socket
write(new_socket, buffer, strlen(buffer));
printf("Message sent to client\n");
```

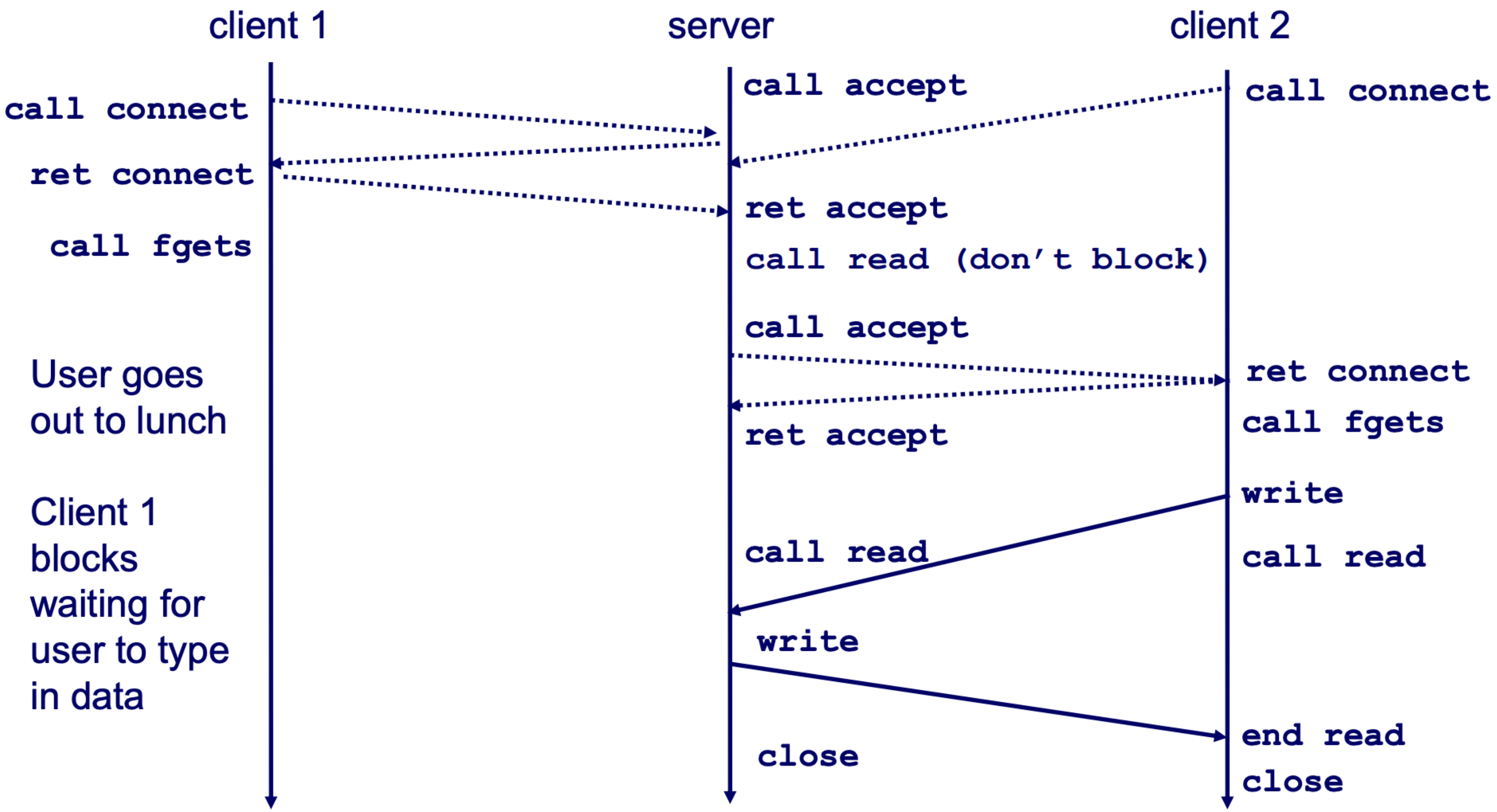
# Close the Socket

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

# Other problem? Concurrency!



# Concurrent server



# Concurrency

- Threading
  - Easier to understand
  - Race conditions increase complexity
- *select()/poll()*
  - Explicit control flows, no race conditions
  - Explicit control more complicated

# I/O multiplexing

- I/O multiplexing: to be notified, by kernel, if one or more I/O conditions are ready
- Example in network applications
  - A server handling a listening socket and its connected sockets
  - A server handling multiple services and protocols
  - A client handling multiple sockets and/or descriptors (e.g., stdio)

# Poll

- A system call for monitoring multiple file descriptors.
- Advantages
  - Handles multiple file descriptors concurrently
  - Simple API for tracking and responding to I/O events
  - No limit on the number of file descriptors (compared with *select*)

# How Poll works

- Initialize a list of *pollfd* structures, one per file descriptor
- Specify the events to monitor (e.g., *POLLIN* for incoming data)
- Call *poll* and wait for events
- Check ***revents*** to see what events occurred
- Handle the events accordingly

```
struct pollfd {
    int fd;         // File descriptor
    short events;   // Requested events
    short revents;  // Returned events
};
```

# Example

- Allow address reuse for multiple connections
  - *setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opts, sizeof(opts));*
  - *opts=1*

```c
// Create a master socket
if ((master_socket = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("socket failed");
    exit(EXIT_FAILURE);
}

// Set master socket to allow multiple connections
if (setsockopt(master_socket, SOL_SOCKET, SO_REUSEADDR, (char *)&opt, sizeof(opt)) < 0) {
    perror("setsockopt");
    exit(EXIT_FAILURE);
}
```

# Example

- *pollfd* initialization

```
struct pollfd poll_fds[MAX_CLIENTS + 1]; // +1 for the master socket
// Set up the initial listening socket
poll_fds[0].fd = master_socket;
poll_fds[0].events = POLLIN;

// Initialize client sockets...
```

# Example

- Poll in main loop
  - *int poll(struct pollfd *fds, nfds_t nfds, int timeout)*
  - *nfds*: The number of items in the *fds* array.
  - *timeout*: The number of milliseconds that *poll()* should block waiting for a file descriptor to become ready. The call will block indefinitely if this argument is -1

```c
while (1) {
    // Prepare the poll_fds array
    // ...
    
    // Wait for events
    int event_count = poll(poll_fds, max_clients + 1, -1);

    // Handle new connections or IO on existing ones
    // ...
}
```

# Example

- Handle events
  - Check for events and proceed to handling logic

```
if (poll_fds[0].revents & POLLIN) {
    // Accept new connection
    // ...
}

for (int i = 0; i < max_clients; i++) {
    if (poll_fds[i + 1].revents & POLLIN) {
        // Handle incoming data
        // ...
    }
}
```

# Review

- Socket
  - Setup
  - I/O
  - close
- Client: *socket()----------------------->connect()->I/O->close()*
- Server: *socket()->bind()->listen()->accept()--->I/O->close()*

- Concurrency: *poll()*

# Credits

- Some slides are adapted from course slides of 15-213 in CMU