CS118 Discussion 1D, Week 1

Zengwen Yuan
Boelter Hall 2760, Friday 4:00—5:50 p.m.

TA

- Zengwen Yuan, PhD in Computer Networking
- Discussion (1D): Boelter Hall 2760, Fri 4:00 – 5:50 p.m.
- Office hours: Boelter Hall 2432 Tuesday 2 – 4 p.m.
- TA website: http://web.cs.ucla.edu/~zyuan/teaching/winter18/cs118.html
- Emails: use [CS118] in the title or may be flagged as spam

Logistics

- Submit your signed Academic Integrity Agreement
- Grade decomposition:
  - Homework: 20% (due 6 p.m. next Wednesday)
  - Project 1: 8% (due Friday, Feb. 2nd)
  - Project 2: 12% (due Friday, Mar. 18th)
  - Midterm: 30% (Thursday, Feb. 15th, in-class)
  - Final: 30% (Monday, 3–6 p.m. Mar. 19th)

Logistics: Homework

- Online submission to Gradescope only (course entry code: 9P5N5D). DEMO
- Submission guidelines:
  - 1. **Hard deadline** on submission, so submit early! You can **resubmit** multiple times before the deadline, but the system will not accept submissions after the deadline.
  - 2. Each homework problem will have a dedicated **answering box** immediately below. Do **NOT** write your answers outside the box. Any answer outside the dedicated area may not get graded.
  - 3. You are encouraged to work out the problem on the PDF file directly without altering the page layout in any ways.
  - 4. If you prefer handwriting or have to draw diagrams, you may scan the paper copy (e.g., using a smartphone app), convert it to a PDF file and then upload. It is **your** responsibility to upload a clear copy in black and white. Inaccessible answers will get low scores.

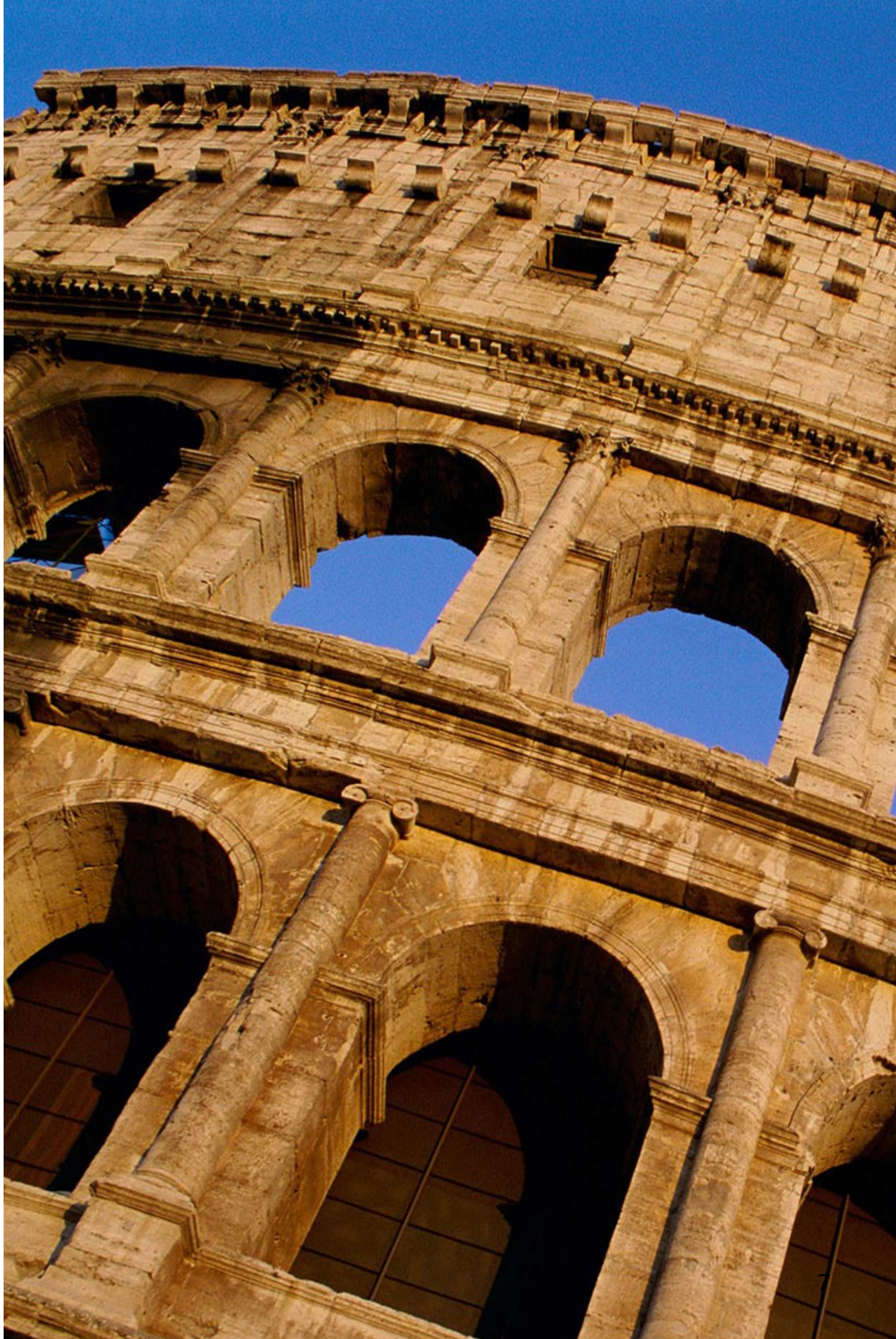
Logistics: Project

- Two projects (in C/C++):
  - A simple web server — get familiar with network programming;
  - Reliable data transfer — implement a simple user-level TCP-like transport protocol
- Form a team of 2 persons ASAP.
- Test environment:
  - (Vagrant-based?) Ubuntu virtual machine

Outline

- Intro to network programming
- Clarification of lectures

Network Programming



Network programming

- **What is the model for network programming?**
- Where are we programming?
- Which APIs can we use? How to use them?

Client-server model

- Asymmetric communication
  - Client — requests data:
    - Initiates communication
    - Waits for server’s response
  - Server (Daemon) — responds data requests:
    - Discoverable by clients (e.g. IP address + port)
    - Waits for clients connection
    - Processes requests, sends replies

Demo: telnet

```
~ telnet google.com 80
Trying 216.58.217.206...
Connected to google.com.
Escape character is '^]'.
GET / HTTP/1.1

HTTP/1.1 200 OK
Date: Fri, 12 Jan 2018 21:44:31 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2018-01-12-21; expires=Sun, 11-Feb-2018 21:44:31 GMT; path=/; domain=.google.com
Set-Cookie: NID=12…J; expires=Sat, 14-Jul-2018 21:44:31 GMT; path=/; domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
Transfer-Encoding: chunked

754a
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en"><head><meta
content="Search the world's information, including webpages, images, videos and more. Google has many
special features to help you find exactly what you're looking for." name="description">
```

Client-server model

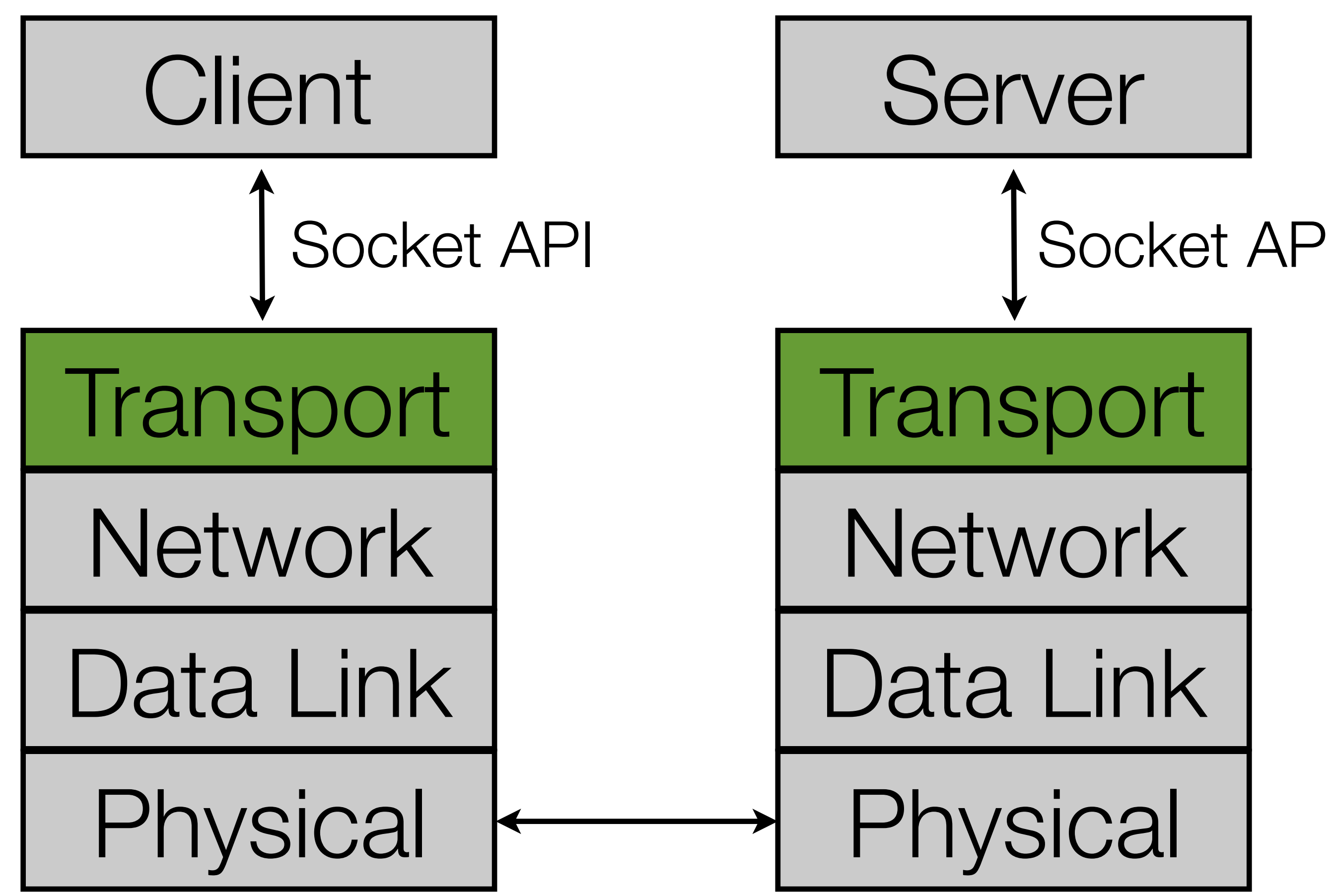
- Client and server are not disjoint
  - A client can be a server of another client
  - A server can be a client of another server
  - Example?
- Server’s service model
  - Concurrent: server processes multiple clients’ requests simultaneously
  - Sequential: server processes clients’ requests one by one
  - Hybrid: server maintains multiple connections, but responses sequentially

Network programming

- What is the model for network programming?
- **Where are we programming?**
- Which APIs can we use? How to use them?

Which layer are we at?

- “Clients” and “servers” are programs at application layer
- Transport layer is responsible for providing communication services for application layer
- Basic transport layer protocols:
  - TCP
  - UDP



TCP: Transmission Control Protocol

- A connection is set up between client and server
- Reliable data transfer
  - Guarantee deliveries of all data
  - No duplicate data would be delivered to application
- Ordered data transfer
  - If A sends data D1 followed by D2 to B, B will also receive D1 before D2
- Data transmission: full-duplex byte stream (in two directions simultaneously)
- Regulated data flow: flow control and congestion control

UDP: User Data Protocol

- Basic data transmission service
  - Unit of data transfer: datagram (in variable length)
- No reliability guarantee
- No ordered delivery guarantee
- No flow control / congestion control

Network programming

- What is the model for network programming?
- Where are we programming?
- **Which APIs can we use? How to use them?**

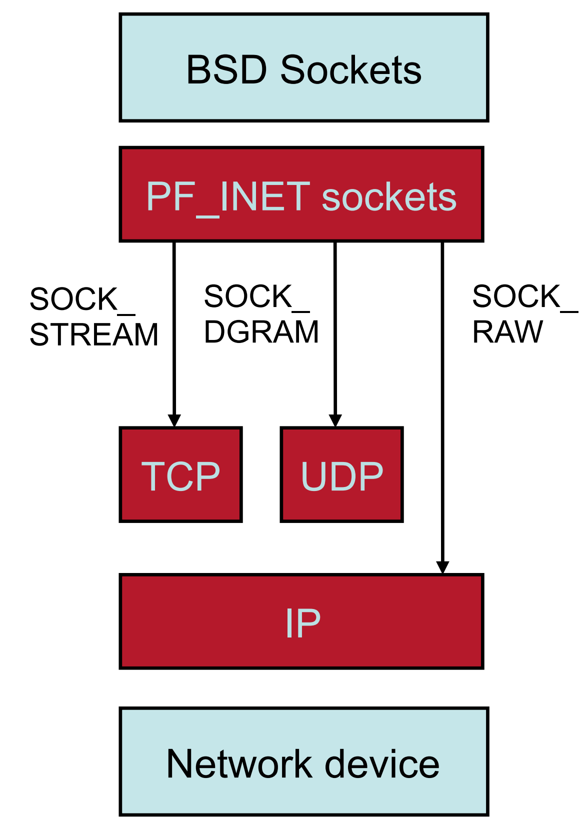
Our secret weapon: socket programming APIs

- From Wikipedia: “A network socket is an endpoint of an inter-process communication flow across a computer network”

- A socket is a tuple of **<ip_addr:port>**

- Socket programming APIs help build the communication tunnel between applications and transport/network service

- We use TCP socket in this project



Socket: port number

- Port numbers are allocated and assigned by the IANA (Internet Assigned Numbers Authority)
- See RFC 1700 or https://www.ietf.org/rfc/rfc1700.txt

1-512	• standard services (see **/etc/services**) • super-user only
513-1023	• registered and controlled, also used for identity verification • super-user only
1024-49151	• registered services/ephemeral ports
49152-65535	• private/ephemeral ports

TCP socket: basic steps

- Create service
- Establish a TCP connection
- Send and receive data
- Close the TCP connection

TCP socket: service setup

TCP Client | **TCP Server**

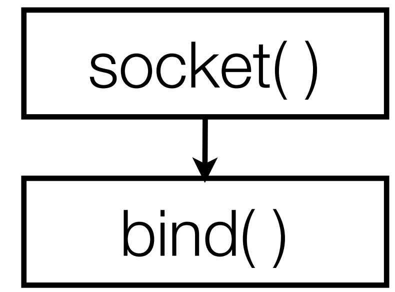
TCP socket: service setup

TCP Client	**TCP Server**
	socket( )

TCP socket: service setup

TCP Client

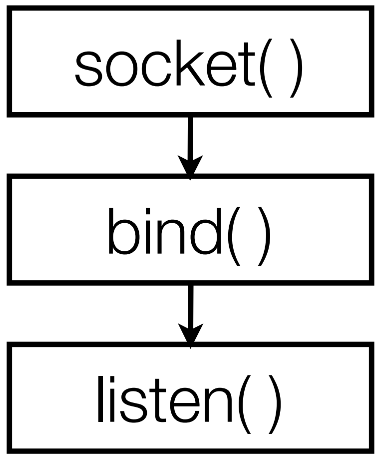
TCP Server



TCP socket: service setup

TCP Client

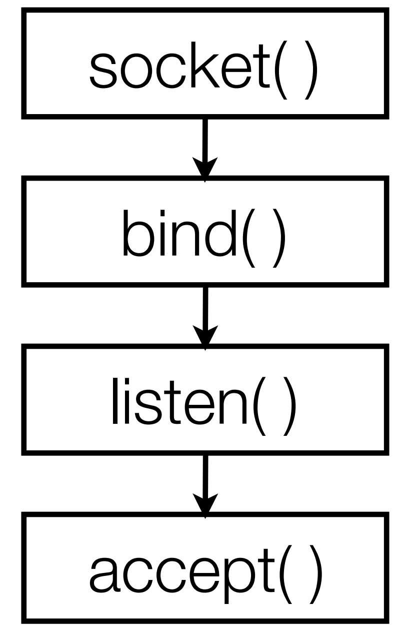
TCP Server



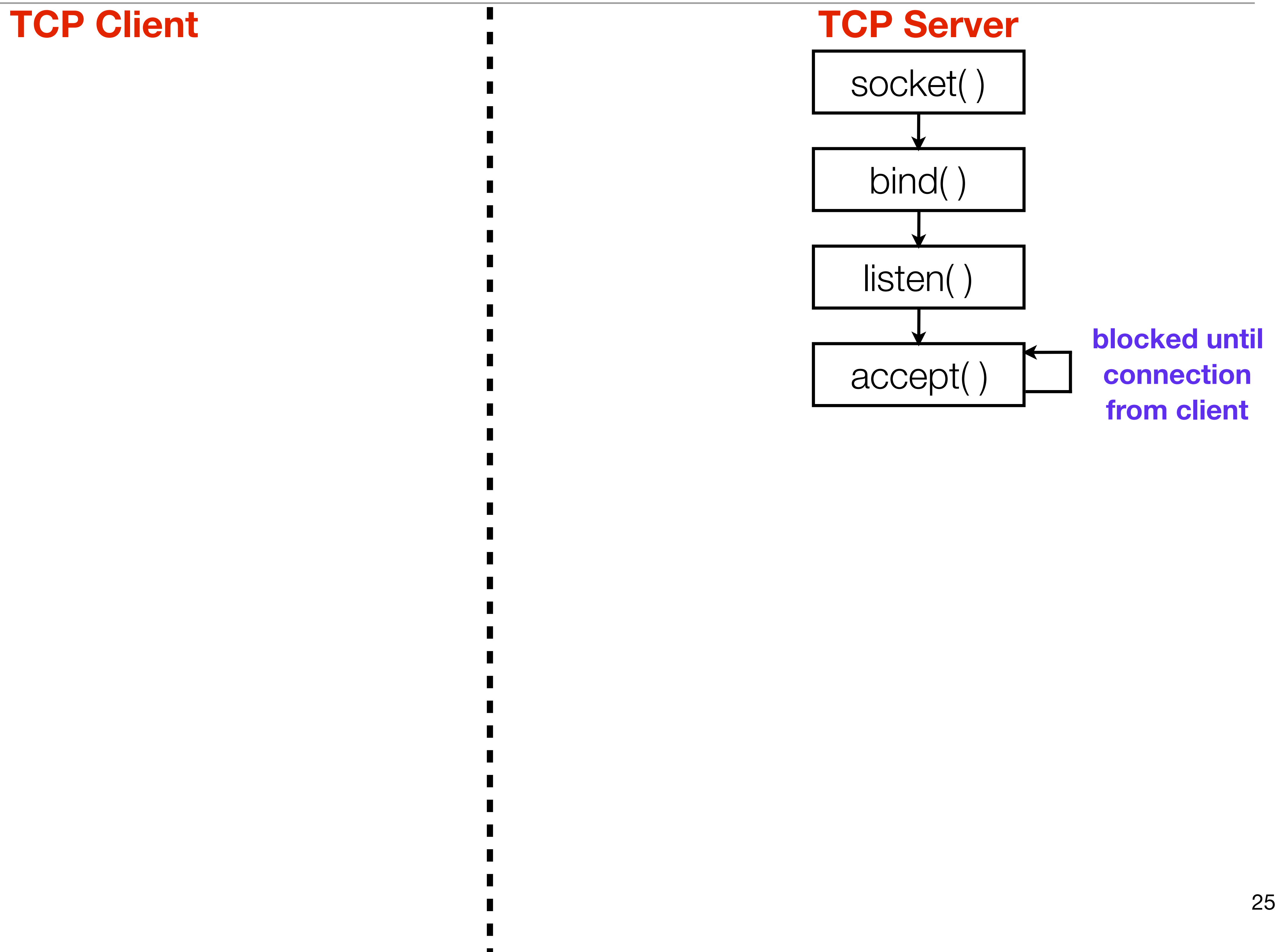
TCP socket: service setup

TCP Client

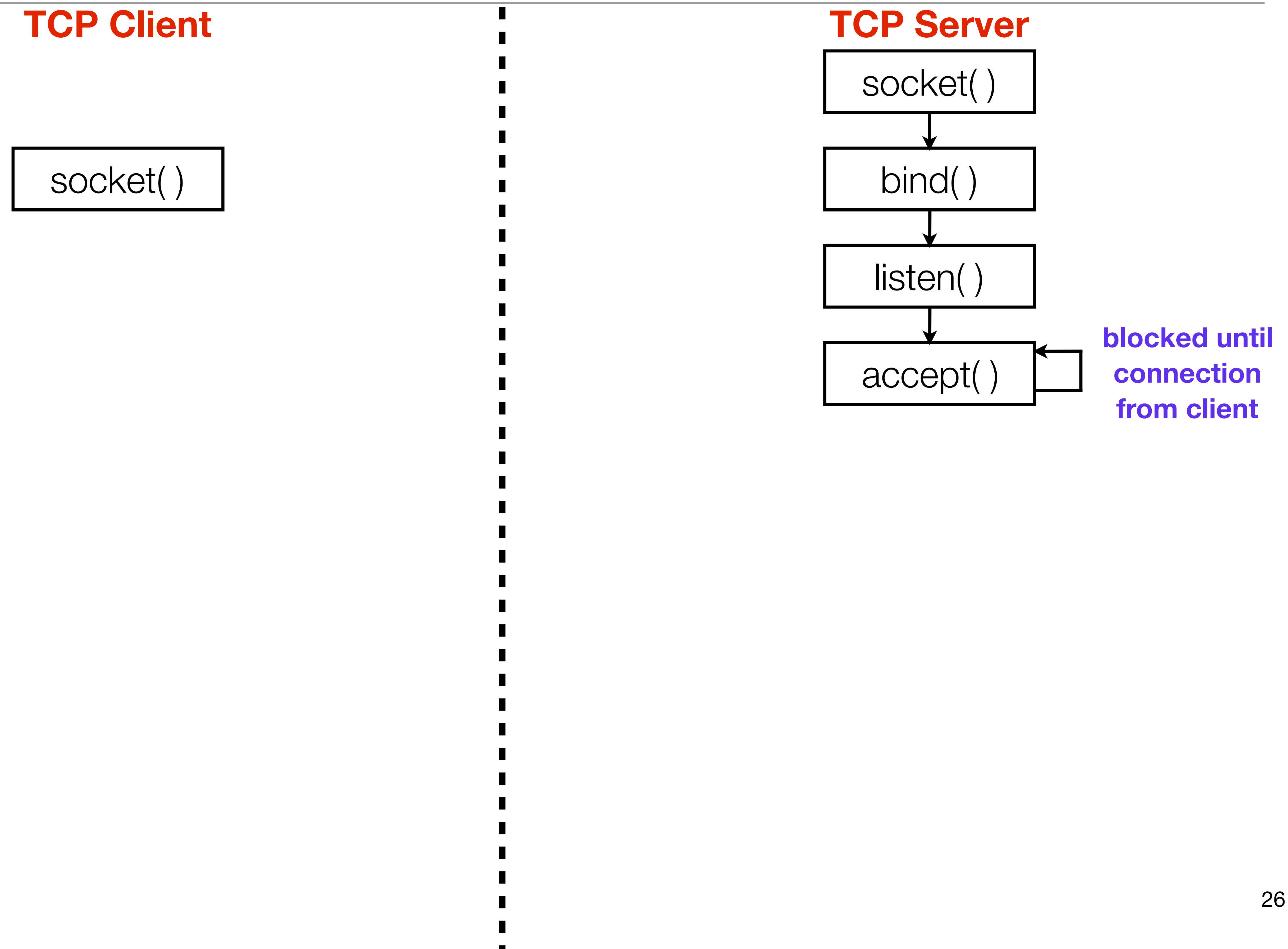
TCP Server



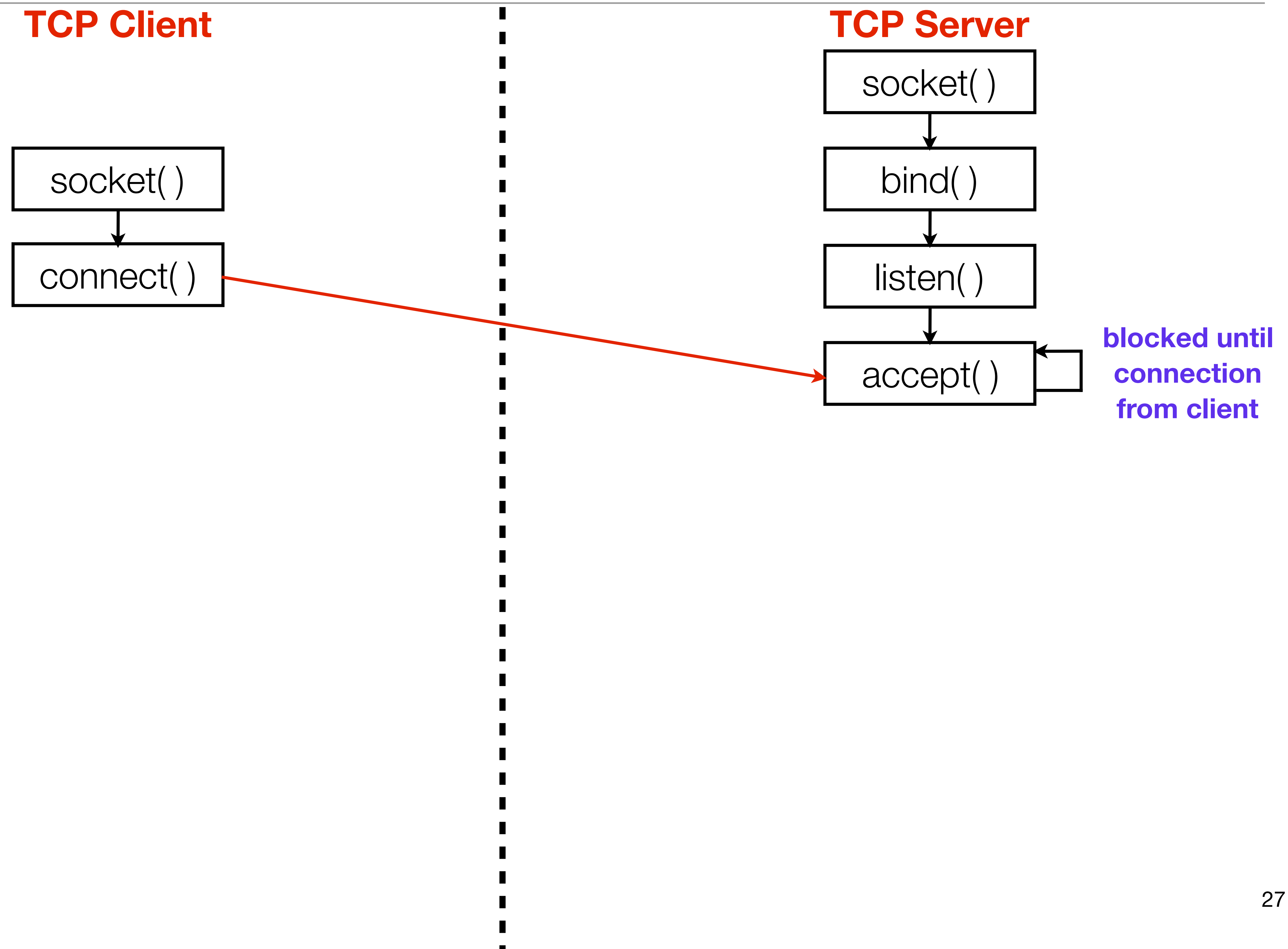
TCP socket: service setup



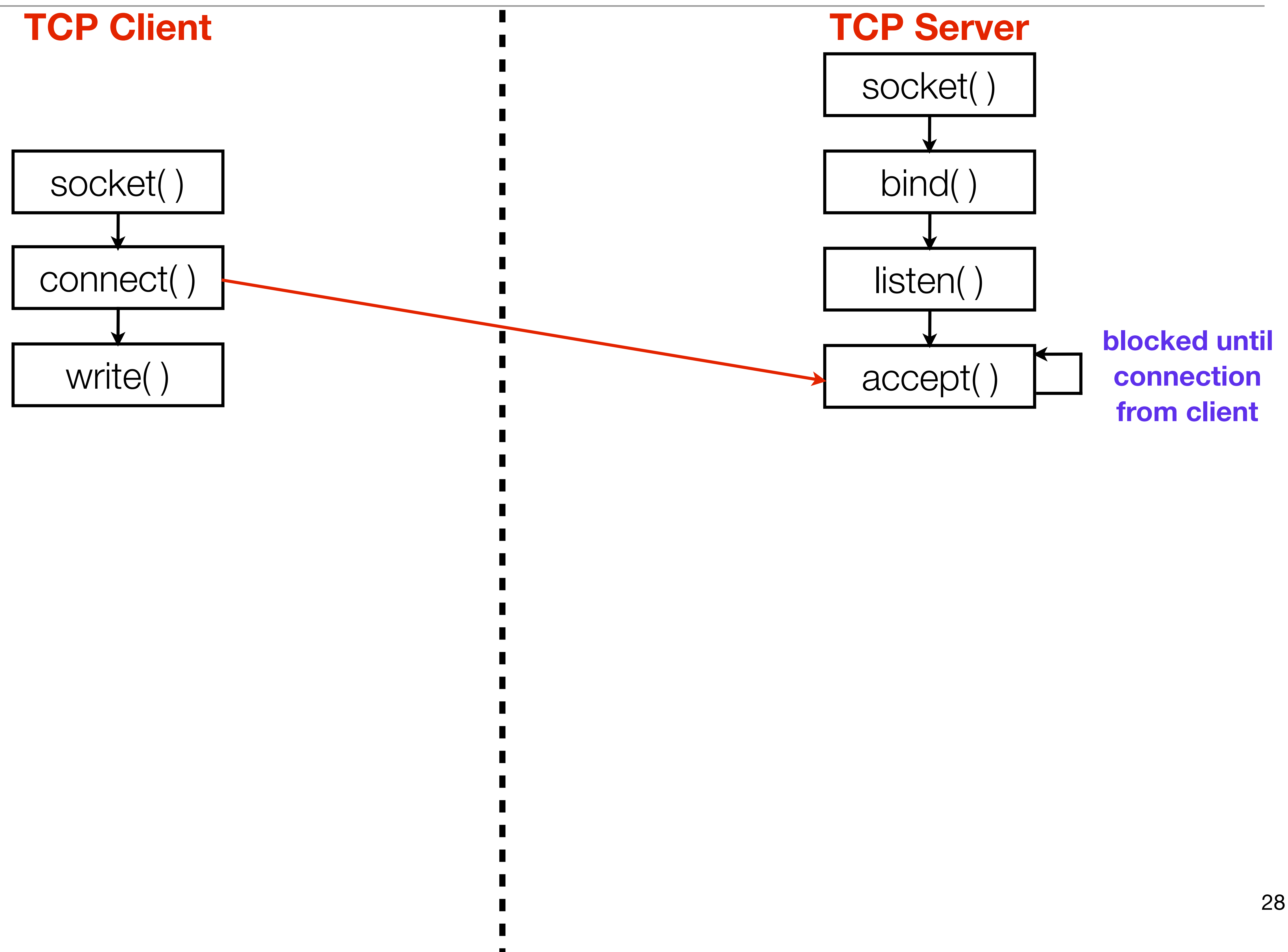
TCP socket: service setup



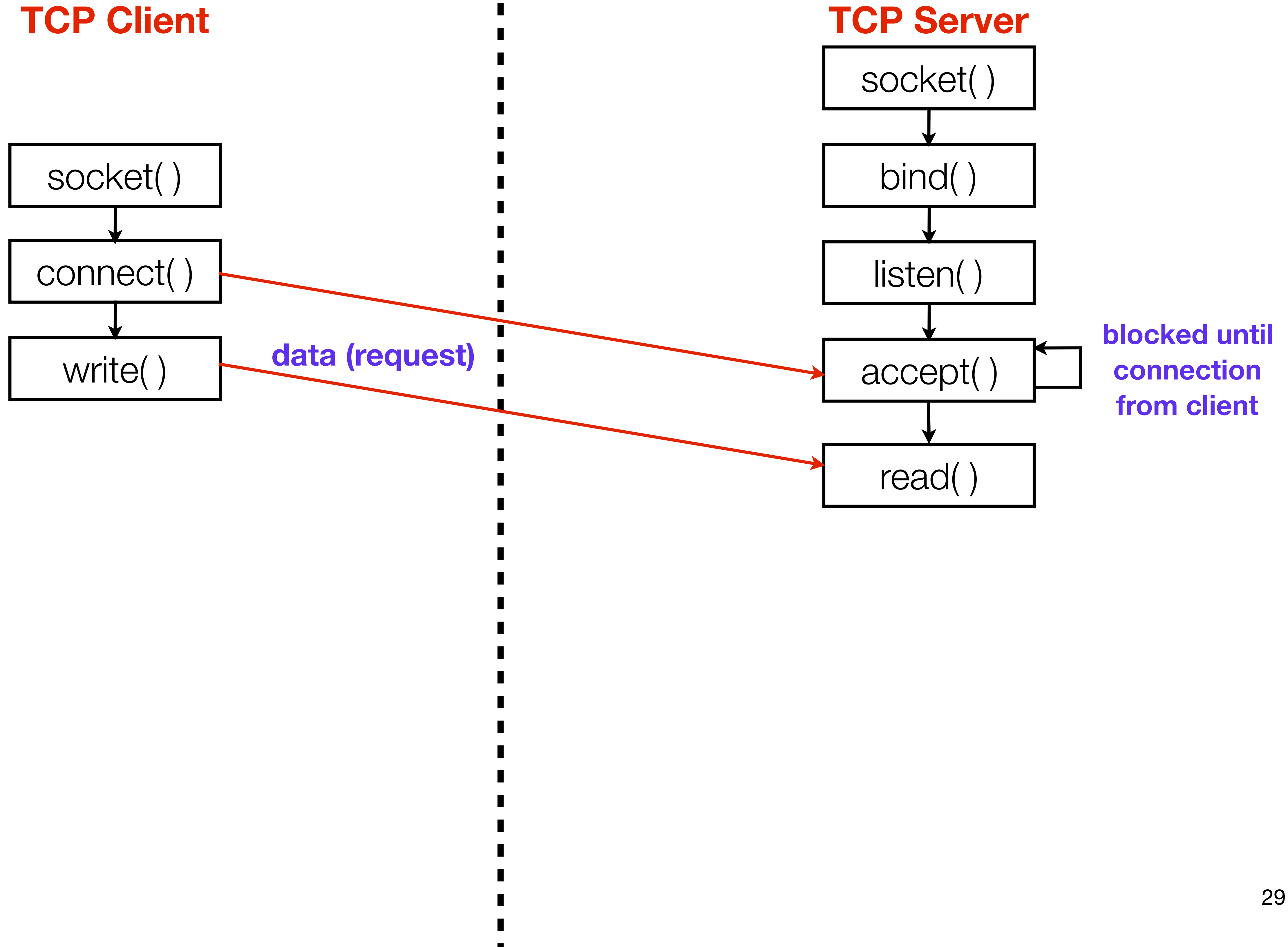
TCP socket: establish connection



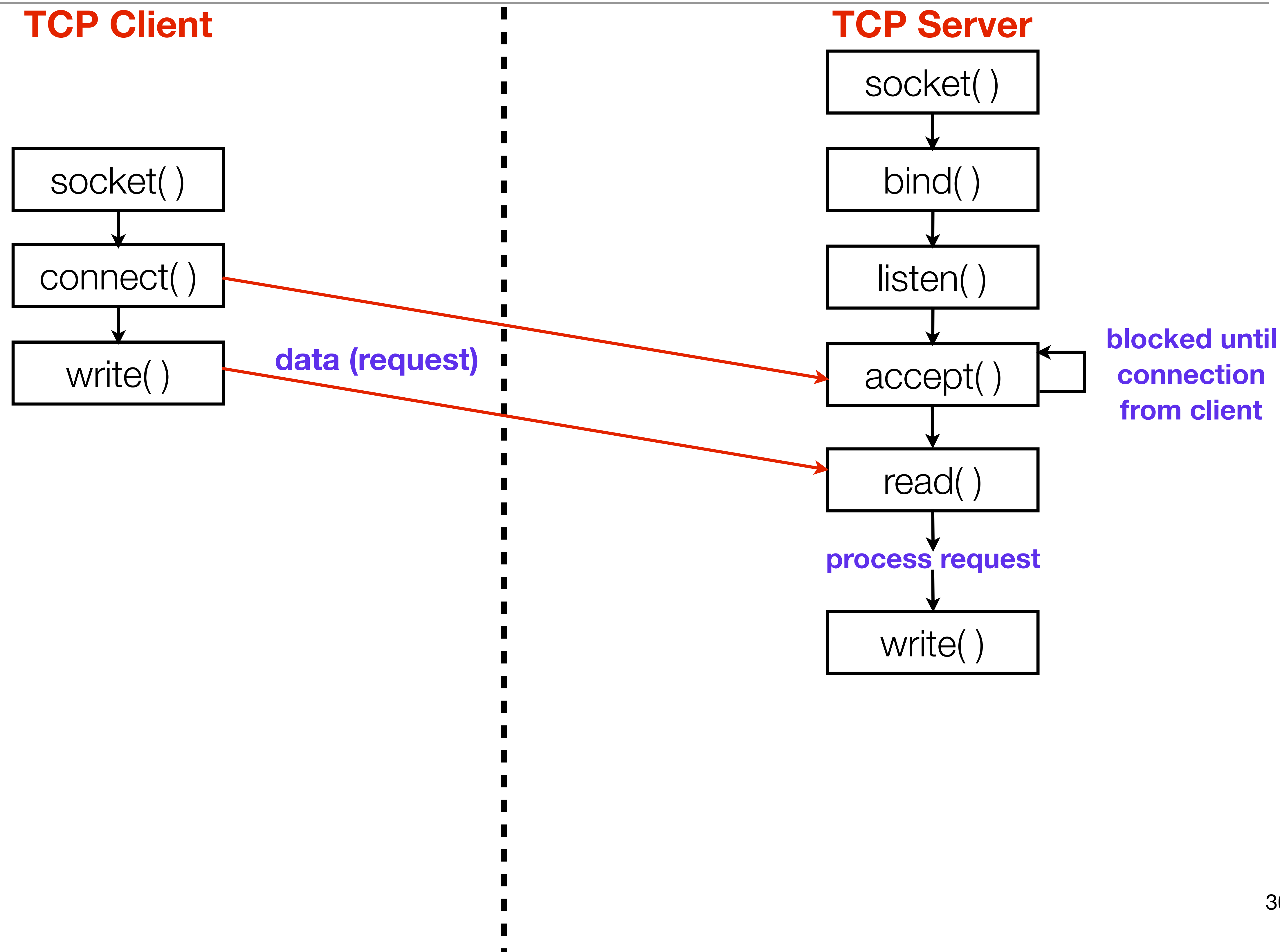
TCP socket: send and receive data



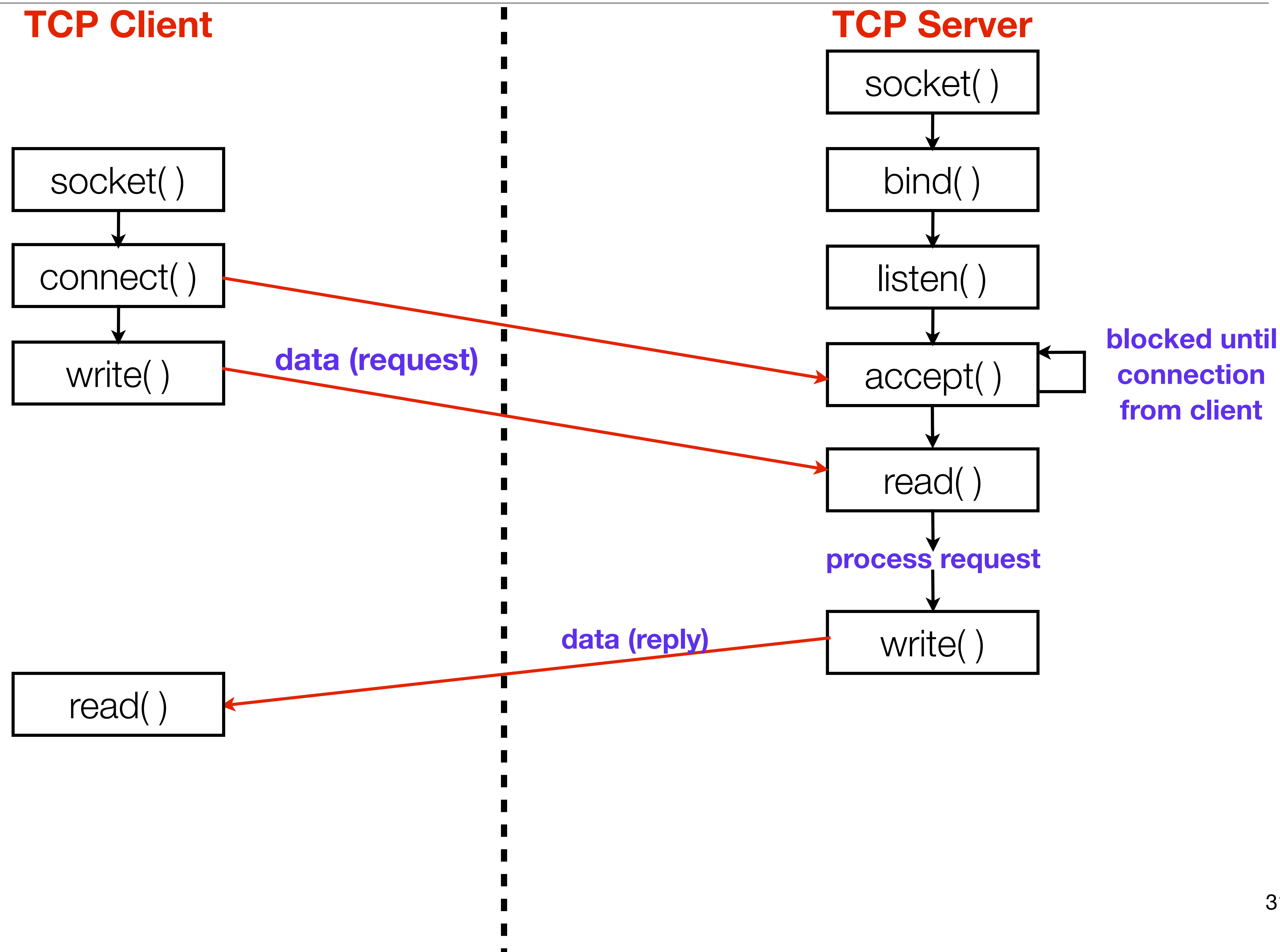
TCP socket: send and receive data



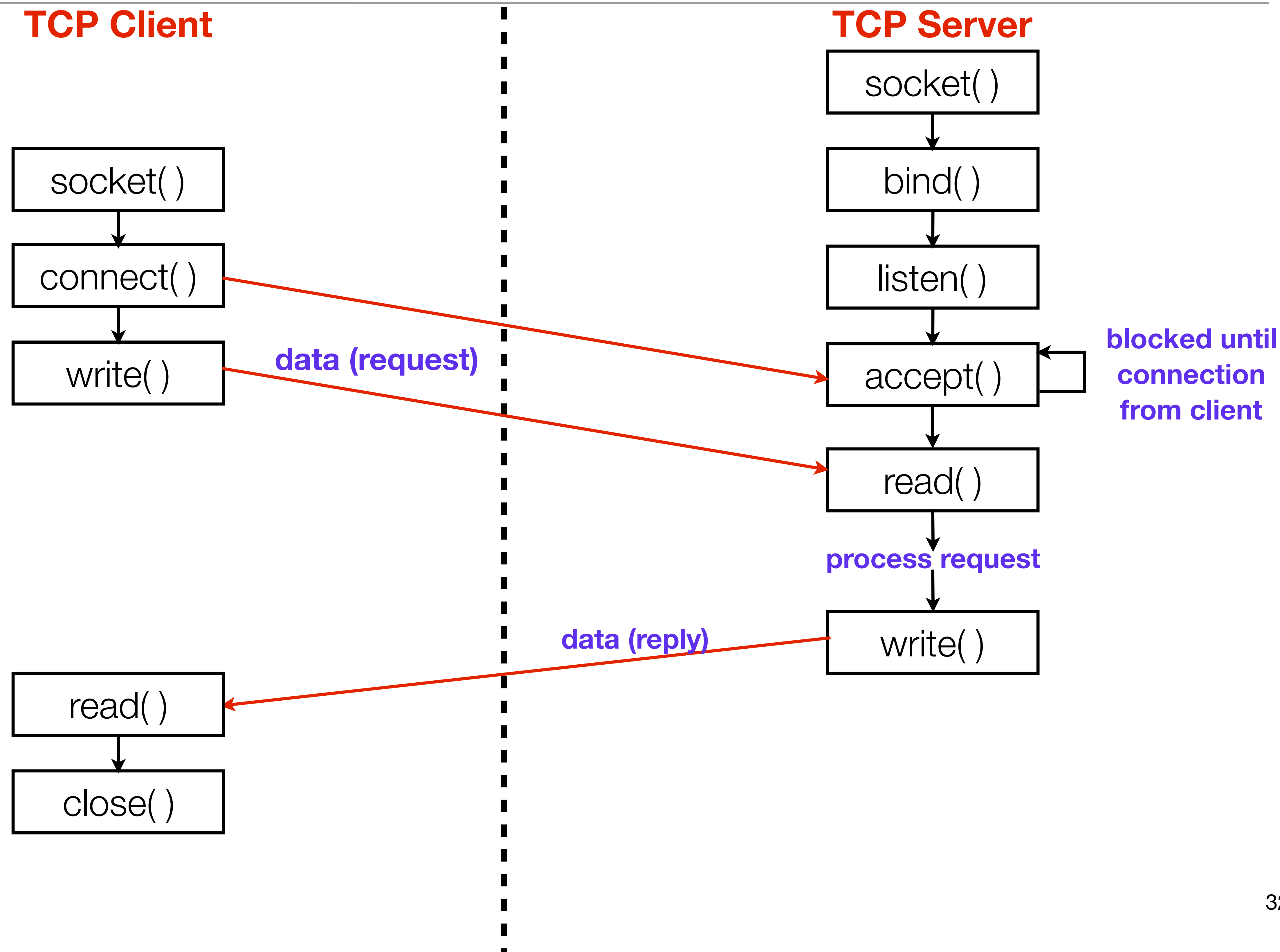
TCP socket: send and receive data



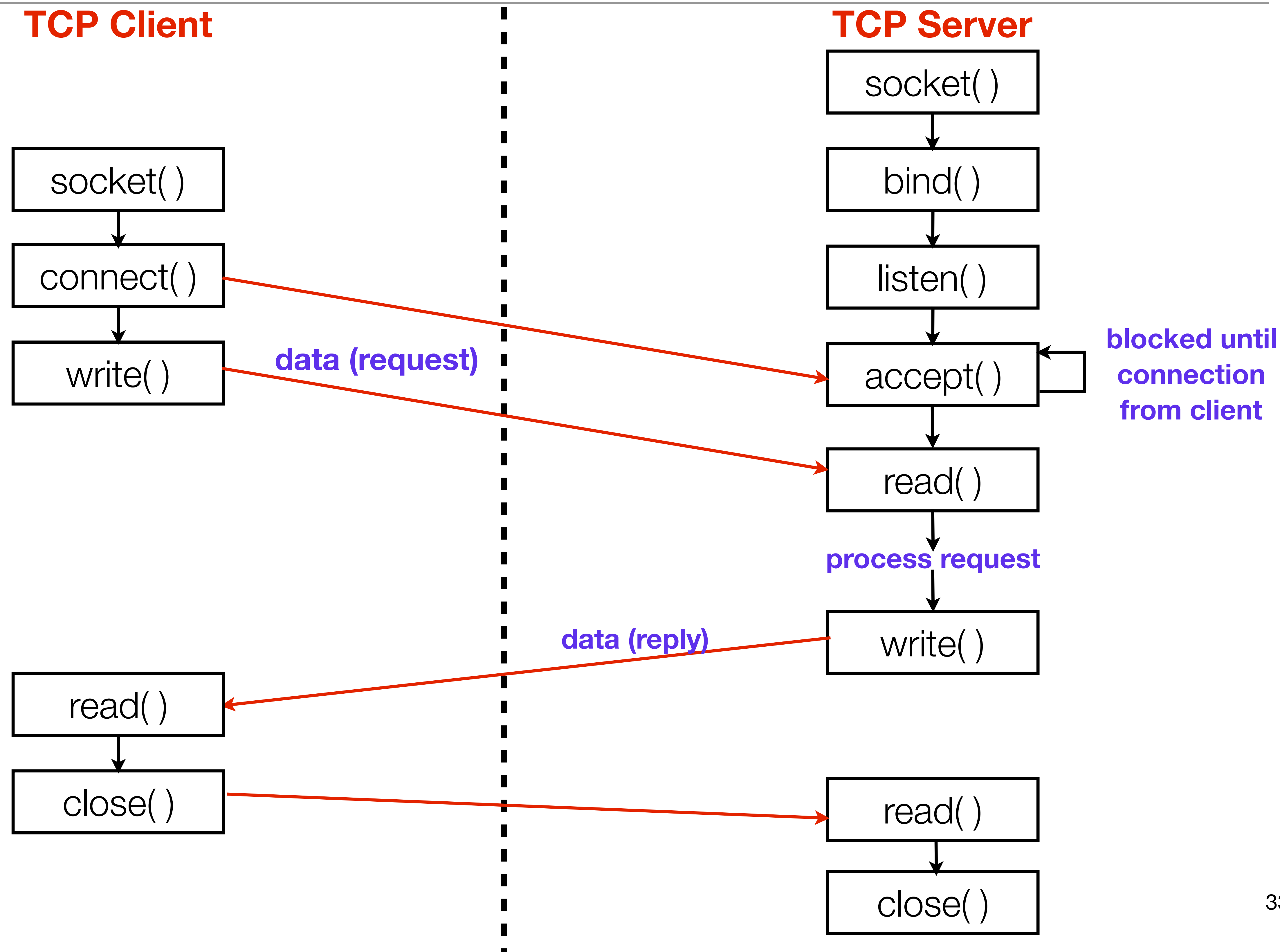
TCP socket: send and receive data



TCP socket: close connection



TCP socket: close connection



Socket programming API: syscalls

- **int socket(int domain, int type, int protocol);**
  - Create a socket
  - returns the socket descriptor or -1(failure). Also sets errno upon failure
  - **domain**: protocol family
    - **PF_INET** for IPv4, **PF_INET6** for IPv6, **PF_UNIX** or **PF_LOCAL** for Unix socket, **PF_ROUTE** for routing
  - **type**: communication style
    - **SOCK_STREAM** for TCP (with **PF_INET**)
    - **SOCK_DGRAM** for UDP (with **PF_INET**)
  - **protocol**: protocol within family, which is typically set to 0

Socket programming API: essential structs

- sockfd — socket descriptor. Just a regular `int`.
- sockaddr — socket address info
- sockaddr_in — yet another struct for the ‘internet’

```
struct sockaddr {
    unsigned short sa_family;   // addr family, AF_xxx
    char           sa_data[14]; // 14 bytes of proto addr
};
struct sockaddr_in { // used for IPv4 only
    short          sin_family;  // addr family, AF_INET
    unsigned short sin_port;    // port number
    struct in_addr sin_addr;    // internet address
    unsigned char  sin_zero[8]; // zeros, same size as sockaddr
};
struct in_addr { // used for IPv4 only
    uint32_t       sin_port;    // 32-bit IPv4 address
};
```

Socket programming API: syscalls

- **int bind(int sockfd, struct sockaddr* myaddr, int addrlen);**
  - Bind a socket to a local IP address and port number
  - returns 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor returned by socket ( )
  - **myaddr**: includes IP address and port number
    - **NOTE**: sockaddr and sockaddr_in are of same size, use sockaddr_in and convert it to socketaddr
    - **sin_family**: protocol family, e.g. AF_INET
    - **sin_port**: port number assigned by caller
    - **sin_addr**: IP address
    - **sin_zero**: used for keeping same size as sockaddr
  - **addrlen**: sizeof(struct sockaddr_in)

```
struct sockaddr {
    short sa_family;
    char sa_data[14];
};

struct sockaddr_in {
    short sin_family;
    ushort sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

a pointer to a struct `sockaddr_in` can be cast to a pointer to a struct `sockaddr` and vice-versa

What's the difference between PF_INET and AF_INET???

Socket programming API: syscalls

- **int listen(int sockfd, int backlog);**
  - Put socket into passive state (wait for connections rather than initiating a connection)
  - returns 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor returned by socket( )
  - **backlog**: the maximum number of connections this program can serve simultaneously

Socket programming API: syscalls

- **int accept(int sockfd, struct sockaddr* client_addr, int* addrlen);**
  - Accept a new connection
  - Return client's socket file descriptor or -1. Also sets errno on failure
  - **sockfd**: socket file descriptor for server, returned by socket( )
  - **client_addr**: IP address and port number of a client (returned from call)
  - **addrlen**: length of address structure = pointer to **int** set to **sizeof(struct sockaddr_in)**
  - **NOTE: client_addr and addrlen are result arguments**
    - i.e. The program passes empty client_addr and addrlen into the function, and the kernel will fill in these arguments with client's information **(why do we need them?)**

Socket programming API: syscalls

- **int connect (int sockfd, struct sockaddr* server_addr, int addrlen);**
  - Connecter to another socket (server)
  - Return 0 on success, -1 and sets errno on failure
  - **sockfd**: socket file descriptor (returned from socket)
  - **server_addr**: IP address and port number of the server
    - server's IP address and port number should be known in advance
  - **addrlen**: sizeof(struct sockaddr_in)

Socket programming API: syscalls

- **int write(int sockfd, char* buf, size_t nbytes);**
  - Write data to a TCP stream
  - Return the number of sent bytes or -1 on failures
  - **sockfd**: socket file descriptor from socket ( )
  - **buf**: data buffer
  - **nbytes**: the number of bytes that caller wants to send

Socket programming API: syscalls

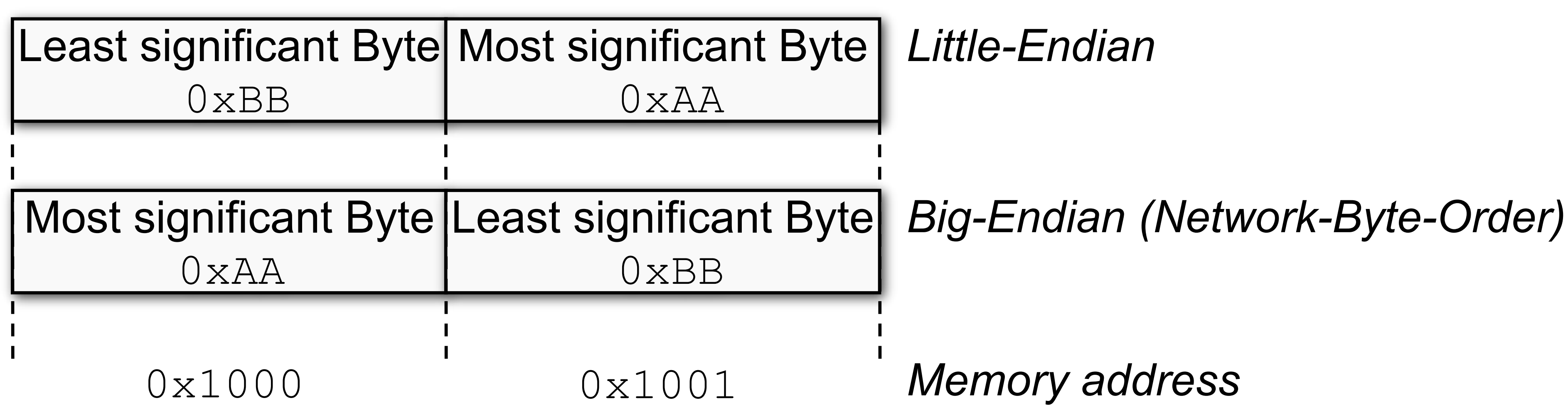
- **int read(int sockfd, char* buf, size_t nbytes);**
  - Read data from TCP stream
  - Return the number of bytes read or -1 on failures
  - Return 0 if socket is closed
  - **sockfd**: socket file descriptor returned from socket ( )
  - **buf**: data buffer
  - **nbytes**: the number of bytes that caller can read (usually set as buffer size)

Socket programming API: syscalls

- **int close(int sockfd);**
  - close a socket
  - return 0 on success, or -1 on failure
  - After close, sockfd is no longer valid

Caveat: byte ordering matters

- Little Endian: least significant byte of word is stored in the lowest address
- Big Endian: most significant byte of word is stored in the lowest address
- Hosts may use different orderings, so we need byte ordering conversion
- **Network Byte Order = Big Endian**



Caveat: byte ordering matters

- Byte ordering functions: used for converting byte ordering
- Example:

```
int m, n;
short int s,t;

m = ntohl (n)   net-to-host long (32-bit) translation
s = ntohs (t)   net-to-host short (16-bit) translation
n = htonl (m)   host-to-net long (32-bit) translation
t = htons (s)   host-to-net short (16-bit) translation
```

- Rule: for every int or short int
- Call htonl( ) or htons( ) before sending data
- Call ntohl( ) or ntohs( ) before reading received data

Address util functions

- All binary values are network byte ordered
- **struct hostent* gethostbyname (const char* hostname);**
  - Translate host name (e.g. “localhost”) to IP address (with DNS working)
- **struct hostent* gethostbyaddr (const char* addr, size_t len, int family);**
  - Translate IP address to host name
- **char* inet_ntoa (struct in_addr inaddr);**
  - Translate IP address to ASCII dotted-decimal notation (e.g. “192.168.0.1”)
- **int gethostname (char* name, size_namelen);**
  - Read local host’s name

FYI: struct hostent

`char *h_name`	The real canonical host name.
`char **h_aliases`	A list of aliases that can be accessed with arrays—the last element is `NULL`
`int h_addrtype`	The result's address type, which really should be `AF_INET` for our purposes.
`int length`	The length of the addresses in bytes, which is 4 for IP (version 4) addresses.
`char **h_addr_list`	A list of IP addresses for this host. Although this is a `char**`, it's really an array of `struct in_addr*`s in disguise. The last element is `NULL`.
`h_addr`	A commonly defined alias for `h_addr_list[0]`. If you just want any old IP address for this host (they can have more than one) just use this field.

Address util functions (cont'd)

- **in_addr_t inet_addr (const char* strptr);**
  - Translate dotted-decimal notation to IP address (network byte order)

```
struct sockaddr_in ina;
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

- **int inet_aton (const char* strptr, struct in_addr *inaddr);**
  - Translate dotted-decimal notation to IP address

```
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;          // host byte order
my_addr.sin_port = htons(MYPORT);      // short, network byte order
inet_aton("10.12.110.57",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct
```

How to write a server: headers

```c
/* PLEASE include these headers */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#define MYPORT 5000   /* Avoid reserved ports */
#define BACKLOG 10 /* pending connections queue size */
```

How to write a server: body (I)

```
int main()
{
    int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;  /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    int sin_size;

    /* create a socket */
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
```

How to write a server: body (II)

```
    // ...
    /* set the address info */
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPORT); /* short, network byte order */
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    /* INADDR_ANY allows clients to connect to any one of the host’s IP
address. Optionally, use this line if you know the IP to use:
       my_addr.sin_addr.s_addr = inet_addr(“127.0.0.1”);
     */
    memset(my_addr.sin_zero, '\0', sizeof my_addr.sin_zero);

    /* bind the socket */
    if (bind(sockfd, (struct sockaddr *) &my_addr,
                      sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }
```

How to write a server: body (III)

```
    // ...
        if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    while (1) { /* main accept() loop */
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr*)
                                     &their_addr, &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n",
               inet_ntoa(their_addr.sin_addr));
        close(new_fd);
    }
}
```

How to write a client?

```
/* include all the headers */
int main() {
    int sockfd, new_fd; /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;   /* my address */
    struct sockaddr_in their_addr; /* connector addr */
    struct hostent* he;
    int sin_size;

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror ("socket");
        exit (1);
    }

    their_addr.sin_family = AF_INET; /* interp'd by host */
    their_addr.sin_port = htons (PORT);
    their_addr.sin_addr = *((struct in_addr*) he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if(connect(sockfd, (struct sockaddr*) &their_addr, sizeof(struct sockaddr)) == -1) {
        perror ("connect");
        exit (1);
    }
    return 0;
}
```

Summary: what we have learned today

- What is the model for network programming?
  - **Client-Server model**
- Where are we programming?
  - **TCP and UDP in a nutshell**
- Which APIs can we use? How to use them?
  - **Socket programming**

Further Reading

- Stevens, W. Richard, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming: The Sockets Networking API*. Vol. 1. Addison-Wesley Professional, 2004.
- Beej's Guide to Network Programming (http://beej.us/guide/bgnet)
- Socket Programming from Dartmouth, http://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html
- C/C++ reference: http://en.cppreference.com

Q&A on lectures

See you next time!

- TA: Zengwen Yuan
- OH: BH2432 Tue 2–4p
- Website:
- http://web.cs.ucla.edu/~zyuan/teaching/winter18/cs118.html