

# SOCKETS PROGRAMMING AND GO'S NET PACKAGE

George Porter  
Sept 28, 2022



# ATTRIBUTION

- These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
- These slides incorporate material from:
  - Alex C. Snoeren, UC San Diego
  - Michael Freedman and Kyle Jamieson, Princeton University
  - Internet Society
  - Computer Networking: A Top Down Approach
  - DK Moon, Berkeley's EE122
  - Network Programming With Go (Woodbeck)

# AGENDA FOR WEEK 2

Sockets programming using Go's net package

## ASSIGNED READINGS

“Network programming with Go” chapter 3

## PROJECTS

Project 1 due Friday at 5pm (not midnight!)



# Today's agenda

- Let's write a couple Go programs that send/receive data over the network using the TCP protocol
- Then we'll go back and take a deep dive on why they work and what is going on "under the engine hood"

# QUICK ADVICE W.R.T. PROJECT 1

- How do you read bytes of data from a file? Write data to a file?
  - The whole file? 100 bytes at a time? 10/90 bytes?
- How do you represent a 10-byte key, 90-byte value record type in Go?
- How do you maintain some kind of list/array of those records?
- There is a sort package for built-in types (e.g. ints). But what about custom record types?
  - Hint: check out sort's `Slice()` method

# BERKELEY SOCKETS API

- From UC Berkeley (1980s)
- Most popular network API
- Ported to various OSes, various languages
  - Windows Winsock, BSD, OS X, Linux, Solaris, ...
  - Socket modules in Java, Python, Perl, ...
- Similar to Unix file I/O API
  - In the form of *file descriptor* (sort of handle).
  - Can share the same `read()/write()/close()` system calls.

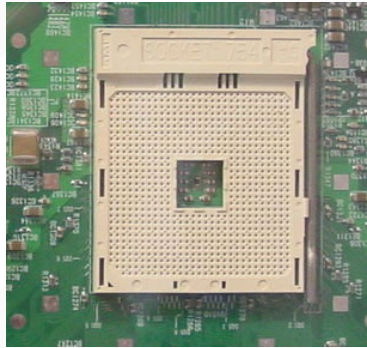
# BERKELEY SOCKETS API

- What is a socket?
  - The point where a local application attaches to the network
  - An interface between an application and the transport protocol
  - An application creates the socket
- The interface defines operations for
  - Creating a socket
  - Attaching a socket to the network
  - Sending and receiving messages through the socket
  - Closing the socket
- Sockets are where your program send and receive data



# SOCKETS

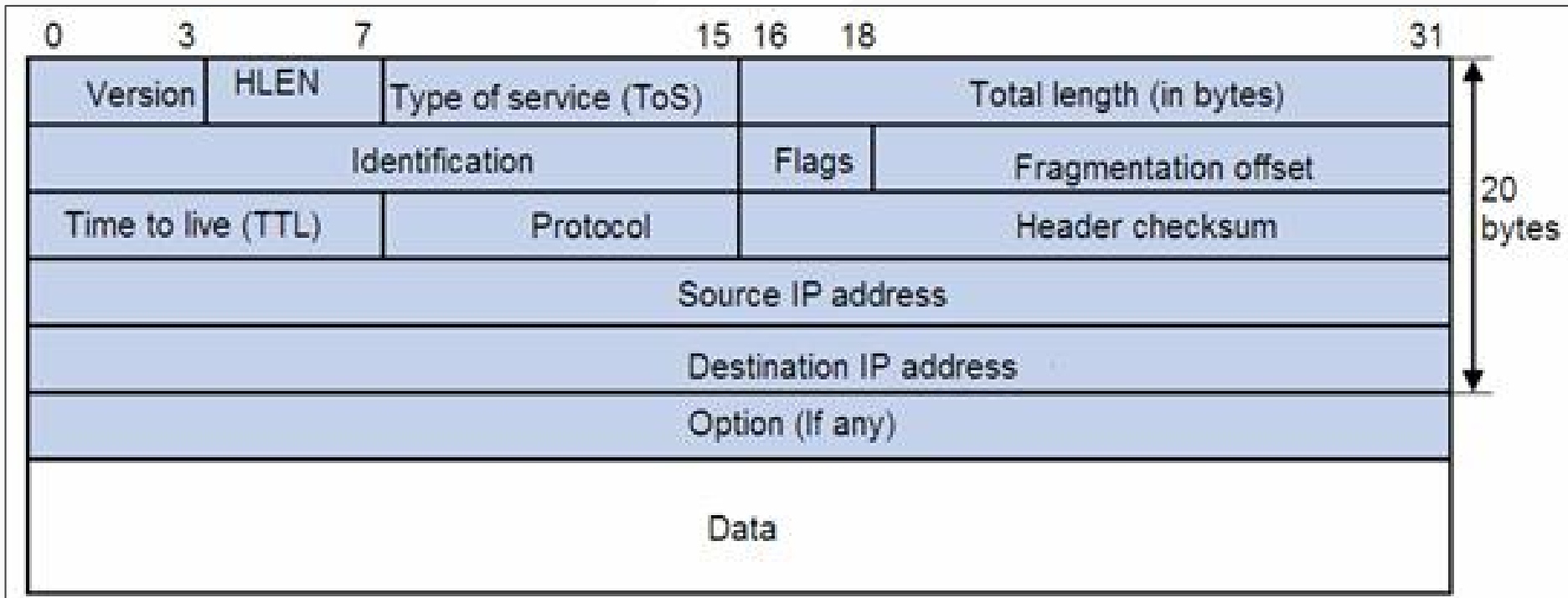
- Various sockets... Any similarity?



- Endpoint of a connection
  - Identified by **IP address** and **Port number**
- Primitive to implement high-level networking interfaces
  - e.g., Remote procedure call (RPC), HTTP, Video streaming, etc etc.



# AN IP PACKET



# INTERNET DELIVERY MODEL

## IP

- Send and receive *packets* of data, up to 64KB in size (though 1500 bytes is the norm)
- Connection-less
- “Best-effort” delivery
  - Arbitrary order of packet delivery
  - Packets can be lost, and there is no automatic retransmission
  - Possible duplicates
  - Packets can get corrupted during transit

# TYPES OF SOCKETS

## TCP

- Connection-oriented
  - Requires connection establishment & termination
- Interface: “Infinite bytestream”
- Reliable delivery
  - In-order delivery
  - Retransmission
  - No duplicates
- High variance in latency
  - Cost of the reliable service
- E.g., HTTP, SSH, FTP, ...

## UDP

- Connection-less
- “Best-effort” delivery
  - Arbitrary order of packet delivery
  - No retransmission
  - Possible duplicates
- Low variance in latency
- Packet-like interface
  - Requires packetizing
- E.g., DNS, VoIP, VOD, ...

# UNIX "TIME" PROTOCOL

Network Working Group  
Request for Comments: 867

J. Postel  
ISI  
May 1983

## Daytime Protocol

This RFC specifies a standard for the ARPA Internet community. Hosts on the ARPA Internet that choose to implement a Daytime Protocol are expected to adopt and implement this standard.

A useful debugging and measurement tool is a daytime service. A daytime service simply sends a the current date and time as a character string without regard to the input.

### TCP Based Daytime Service

One daytime service is defined as a connection based application on TCP. A server listens for TCP connections on TCP port 13. Once a connection is established, the current date and time is sent out the connection as a single character string (and any data received is discarded away). The service closes the connection after sending the quote.

### UDP Based Daytime Service

Another daytime service service is defined as a datagram based application on UDP. A server listens for UDP datagrams on UDP port 13. When a datagram is received, an answering datagram is sent containing the current date and time as a ASCII character string (the data in the received datagram is ignored).

### Daytime Syntax

There is no specific syntax for the daytime. It is recommended that it be limited to the ASCII printing characters, space, carriage return, and line feed. The daytime should be just one line.

One popular syntax is:

Weekday, Month Day, Year Time-Zone

Example:

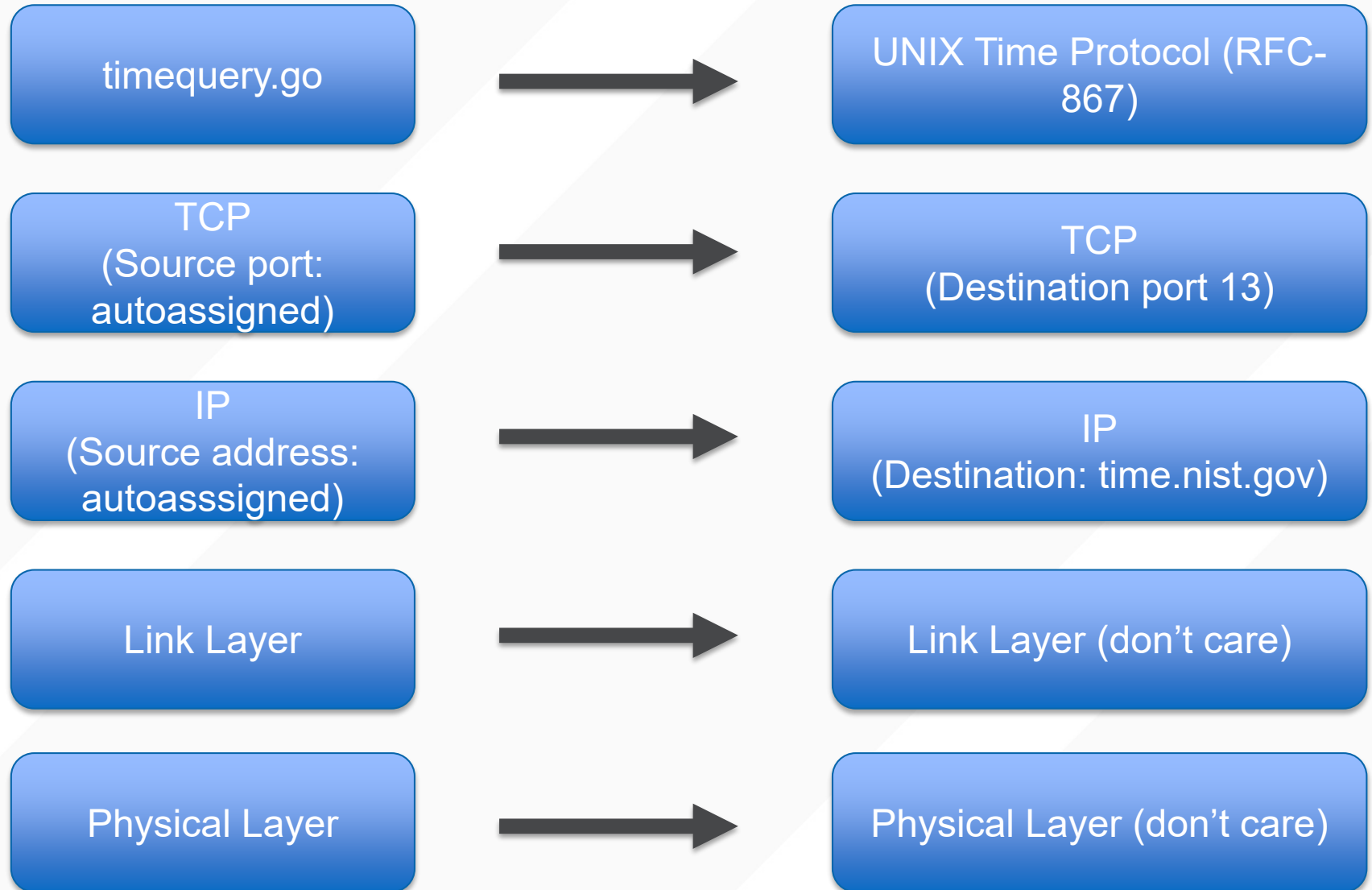
Tuesday, February 22, 1982 17:37:43-PST

TCP Transport  
Protocol

TCP Port 13

Format of the  
response

# UNIX TIME PROTOCOL AND LAYERING



# MINI DEMO: TIMEQUERY.GO

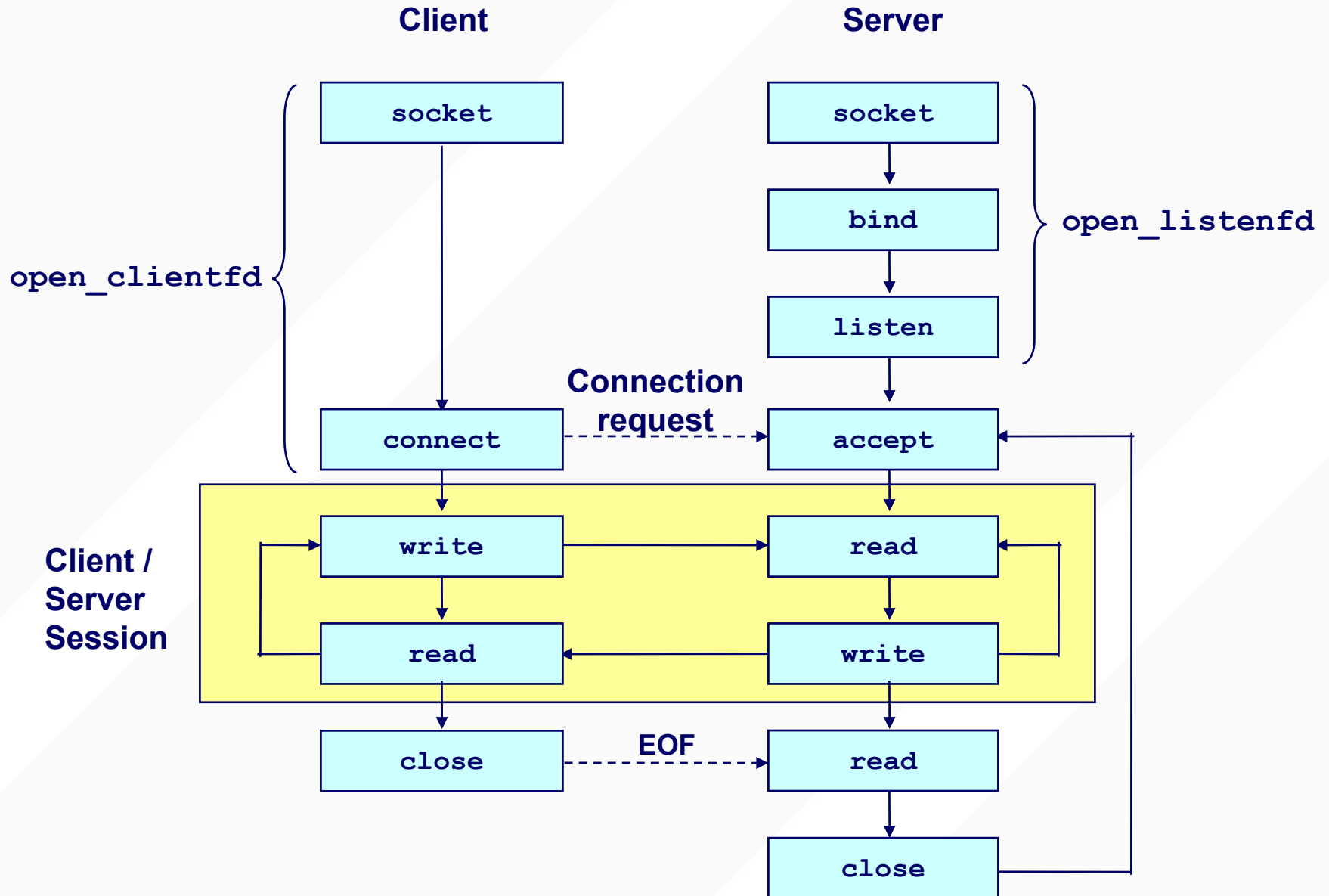
```
gmporter@navygrog lec03-demo % go run timequery.go time.nist.gov:13  
59850 22-09-28 22:33:00 50 0 0 15.1 UTC(NIST) *  
gmporter@navygrog lec03-demo %
```

Full list of servers at: <https://tf.nist.gov/tf-cgi/servers.cgi>

**INITIALLY: THE LOWER-LEVEL C INTERFACE**  
**THEN: THE SAME INTERFACE BUT IN GO**



# CLIENT AND SERVER SOCKETS (SYSTEM CALLS)



# INITIALIZATION (CLIENT AND SERVER)

```
int sock = socket(AF_INET, SOCK_STREAM, 0);  
if (sock < 0) {  
    perror("socket() failed");  
    abort();  
}
```

- **socket()** : returns a socket descriptor
- **AF\_INET**: IPv4 address family. (also OK with PF\_INET)
  - C.f. IPv6 => AF\_INET6
- **SOCK\_STREAM**: streaming socket type
  - C.f. SOCK\_DGRAM
- **perror()** : prints out an error message

# INITIALIZATION ON THE SERVER VIA BIND()

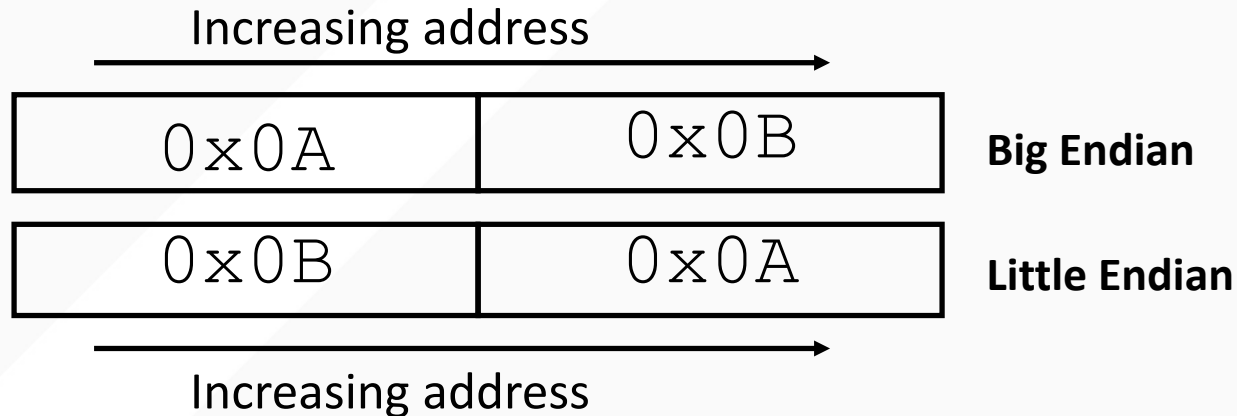
- Server needs to bind a particular port number.

```
struct sockaddr_in sin;  
memset(&sin, 0, sizeof(sin));  
sin.sin_family = AF_INET;  
sin.sin_addr.s_addr = INADDR_ANY;  
sin.sin_port = htons(server_port);  
  
if (bind(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0) {  
    perror("bind failed");  
    abort();  
}
```

- **bind()**: binds a socket with a particular port number.
  - Kernel remembers which process has bound which port(s).
  - Only one process can bind a particular port number at a time.
- **struct sockaddr\_in**: Ipv4 socket address structure. (c.f., struct sockaddr\_in6)
- **INADDR\_ANY**: If server has multiple IP addresses, binds any address.
- **htons()**: converts host byte order into network byte order.

# ENDIANESS

- Q) You have a 16-bit number: 0x0A0B. How is it stored in memory?



- Host byte order is not uniform
  - Some machines are Big endian, others are Little endian
- Communicating between machines with different host byte orders is problematic
  - Transferred \$256 (0x0100), but received \$1 (0x0001)
- For Internet, we standardize on big-endianness
  - hton() and ntohs()

# ENDIANESS (CON'T)

- Network byte order: Big endian
  - To avoid the endian problem
- We must use network byte order when sending 16bit, 32bit, 64bit numbers.
- Utility functions for easy conversion

```
uint16_t htons(uint16_t host16bitvalue);  
uint32_t htonl(uint32_t host32bitvalue);  
uint16_t ntohs(uint16_t net16bitvalue);  
uint32_t ntohl(uint32_t net32bitvalue);
```

- Hint: **h**, **n**, **s**, and **l** stand for host byte order, network byte order, short(16bit), and long(32bit), respectively

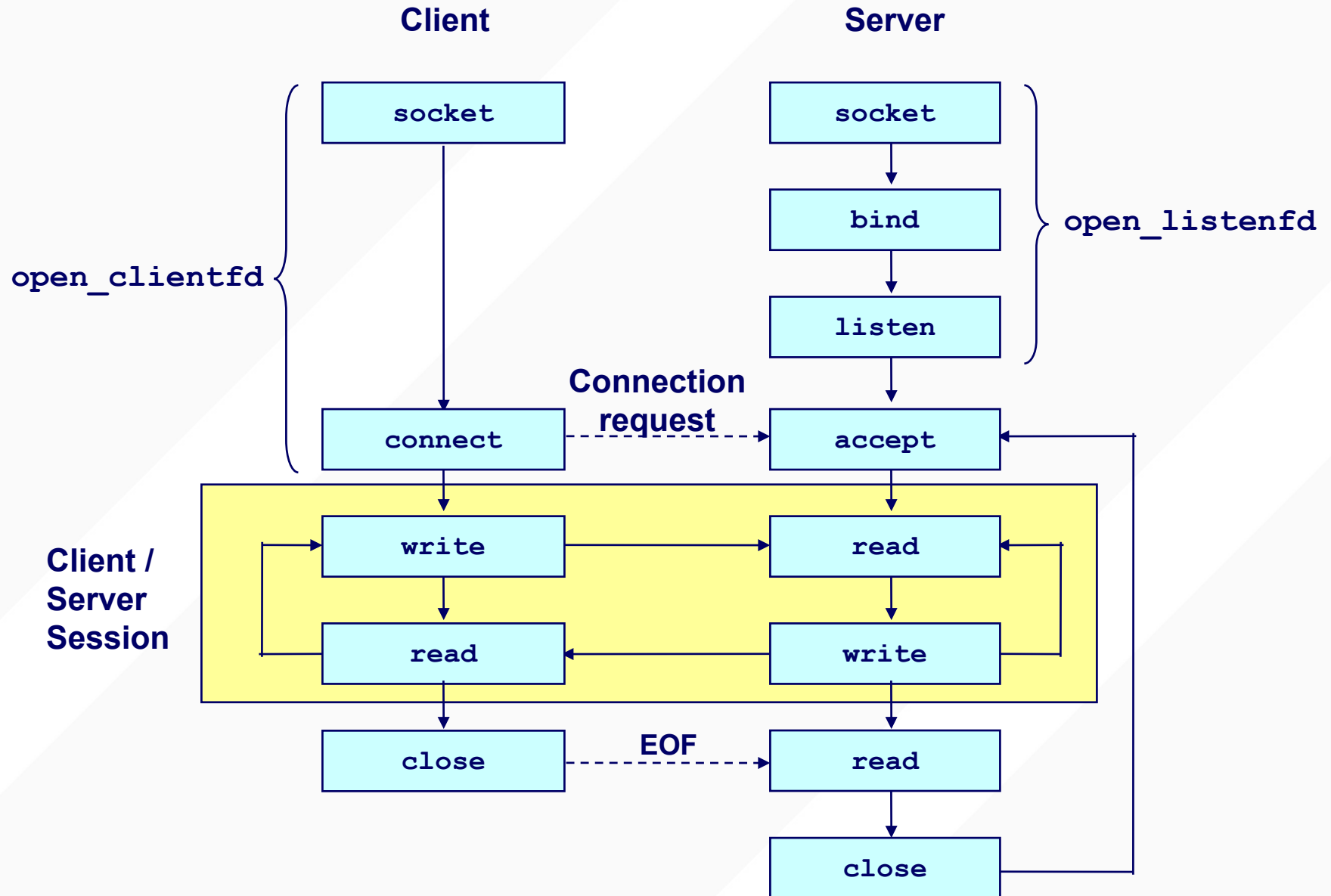
# INITIALIZATION, SERVER VIA LISTEN()

- Socket is active, by default
- We need to make it passive to get connections.

```
if (listen(sock, back_log) < 0) {  
    perror("listen failed");  
    abort();  
}
```

- **listen()**: converts an active socket to passive
- **back\_log**: connection-waiting queue size. (e.g., 32)
  - Busy server may need a large value (e.g., 1024, ...)

# CLIENT AND SERVER SOCKETS (SYSTEM CALLS)





# CONNECTION ESTABLISHMENT (ON THE CLIENT)

```
struct sockaddr_in sin;  
memset(&sin, 0, sizeof(sin));  
  
sin.sin_family = AF_INET;  
sin.sin_addr.s_addr = inet_addr("128.32.132.214");  
sin.sin_port = htons(80);  
  
if (connect(sock, (struct sockaddr *) &sin, sizeof(sin)) < 0) {  
    perror("connection failed");  
    abort();  
}
```

- **Connect()**: waits until connection establishes/fails
- **inet\_addr()**: converts an IP address string into a 32bit address number (network byte order).

# CONNECTION ESTABLISHMENT (ON THE SERVER)

```
struct sockaddr_in client_sin;  
int addr_len = sizeof(client_sin);  
int client_sock = accept(listening_sock,  
                        (struct sockaddr *) &client_sin,  
                        &addr_len);  
  
if (client_sock < 0) {  
    perror("accept failed");  
    abort();  
}
```

- **accept()** : returns a new socket descriptor for a client connection in the connection-waiting queue.
  - This socket descriptor is to communicate with the client
  - The passive socket (**listening\_sock**) is not to communicate with a client

# SENDING DATA (BOTH CLIENT AND SERVER)

```
char *data_addr = "hello, world";
int data_len = 12;

int sent_bytes = send(sock, data_addr, data_len, 0);
if (sent_bytes < 0) {
    perror("send failed");
}
```

- **send()**: sends data, returns the number of sent bytes
  - Also OK with `write()`, `writew()`
- **data\_addr**: address of data to send
- **data\_len**: size of the data
- With blocking sockets (default), `send()` blocks until it sends all the data.
- With non-blocking sockets, **sent\_bytes** may not equal to **data\_len**
  - If kernel does not have enough space, it accepts only partial data
  - You must retry for the unsent data

# RECEIVING DATA (BOTH CLIENT AND SERVER)

```
char buffer[4096];
int expected_data_len = sizeof(buffer);

int read_bytes = recv(sock, buffer, expected_data_len, 0);
if (read_bytes == 0) { // connection is closed
    ...
} else if (read_bytes < 0) { // error
    perror("recv failed");
} else { // OK. But no guarantee read_bytes == expected_data_len
    ...
}
```

- **recv()**: reads bytes from the socket and returns the number of read bytes.
  - Also OK with **read()** and **readv()**
- **read\_bytes** may not equal to **expected\_data\_len**
  - If no data is available, it blocks
  - If only partial data is available, `read_bytes < expected_data_len`
  - On socket close, `expected_data_len` equals to 0 (not error!)
  - If you get only partial data, you should retry for the remaining portion.

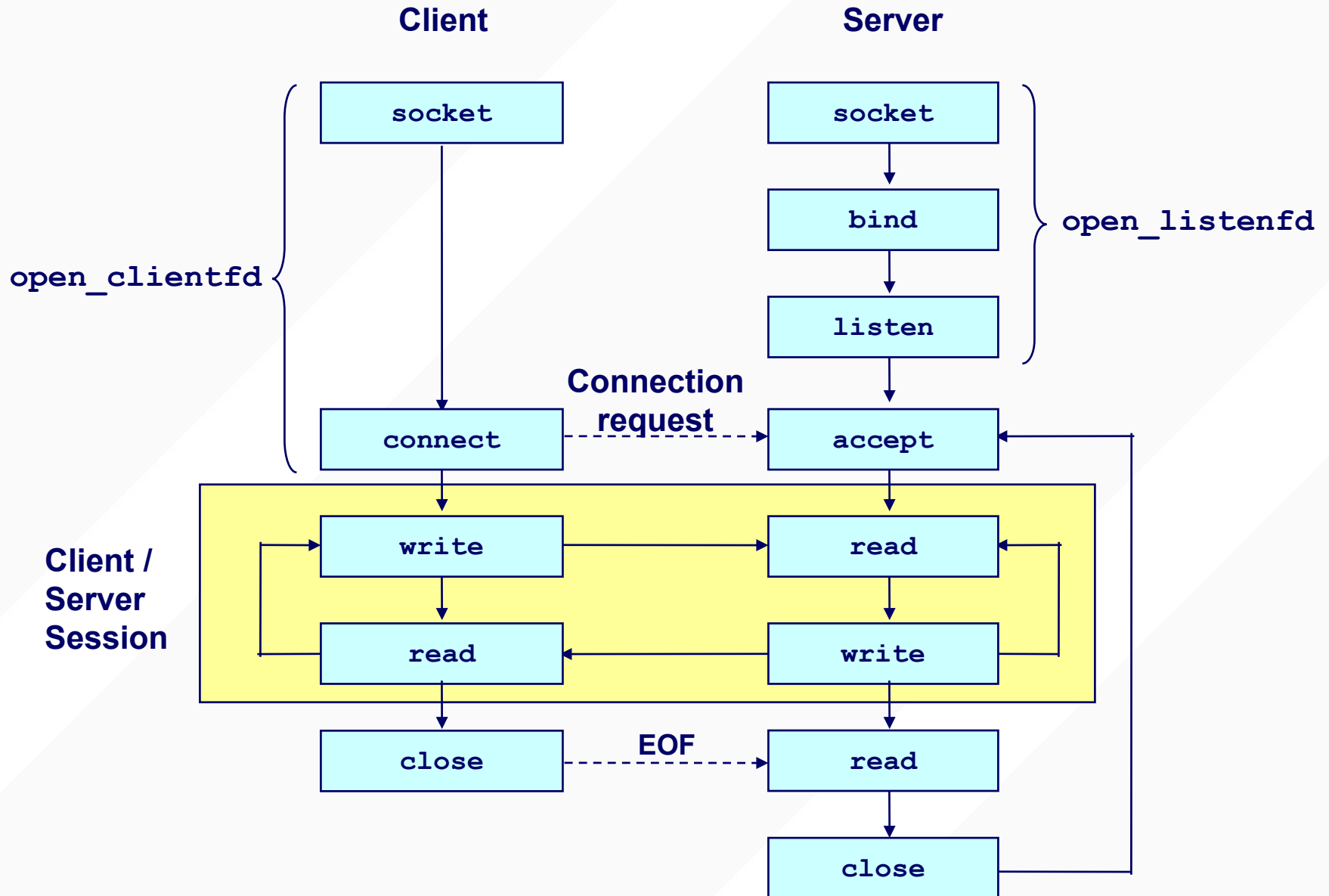
# CLOSING CONNECTION (BOTH CLIENT AND SERVER)

```
// after use the socket  
close(sock);
```

- **close()**: closes the socket descriptor
- We cannot open files/sockets more than 1024\*
  - We must release the resource after use

\* Super user can overcome this constraint, but regular user cannot.

# CLIENT AND SERVER SOCKETS (SYSTEM CALLS)



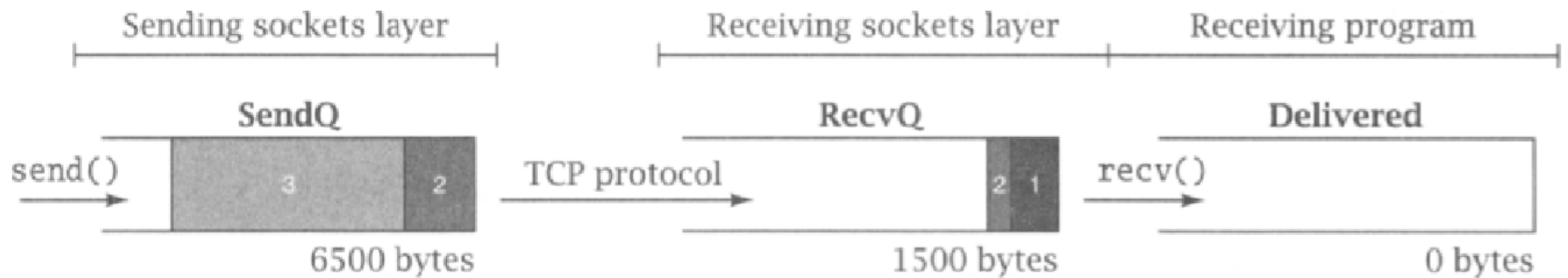
**GO DEMO: [DAYTIMESERVER.GO](https://daytimeserver.go)**



# DIGGING INTO SEND() A BIT MORE

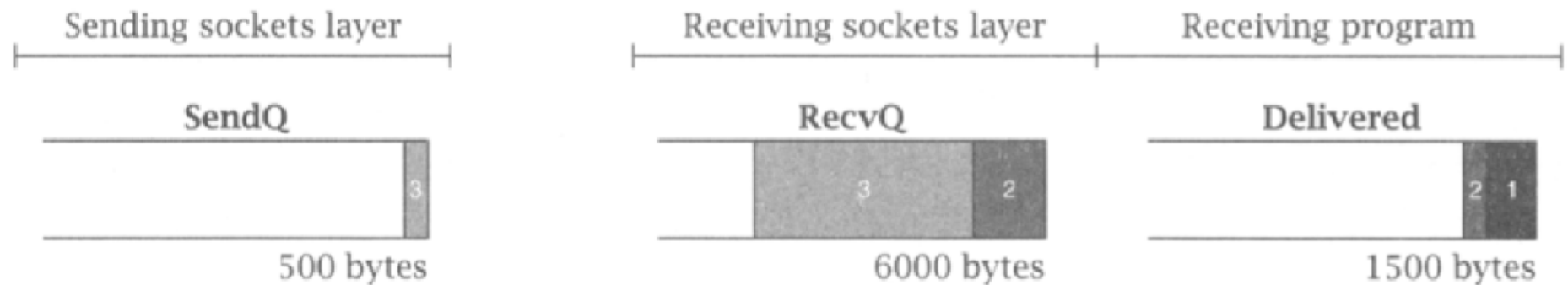
```
rv = connect(s,...);  
:  
:  
rv = send(s,buffer0,1000,0);  
:  
:  
rv = send(s,buffer1,2000,0);  
:  
:  
rv = send(s,buffer2,5000,0);  
:  
:  
close(s);
```

# AFTER 3 SEND() CALLS



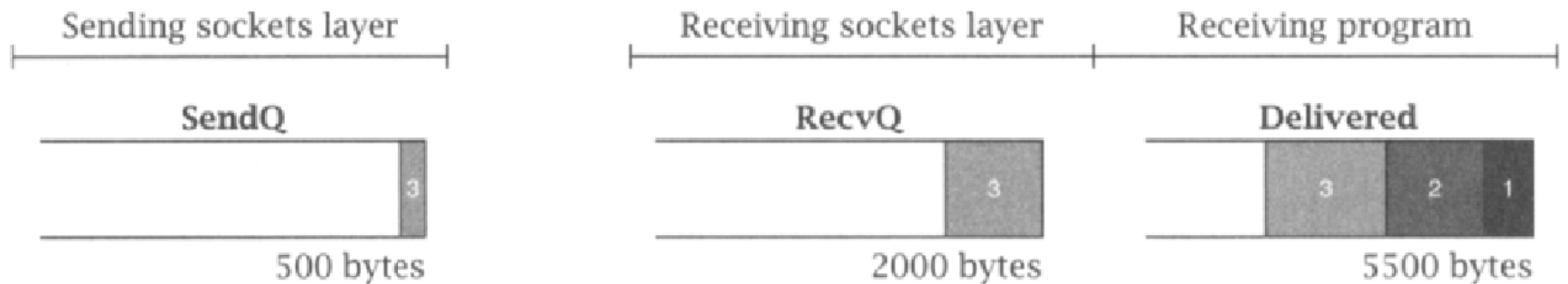
- 1 First send call (1000 bytes)
- 2 Second send call (2000 bytes)
- 3 Third send call (5000 bytes)

# AFTER FIRST RECV()



- 1 First send call (1000 bytes)
- 2 Second send call (2000 bytes)
- 3 Third send call (5000 bytes)

# AFTER ANOTHER RECV()

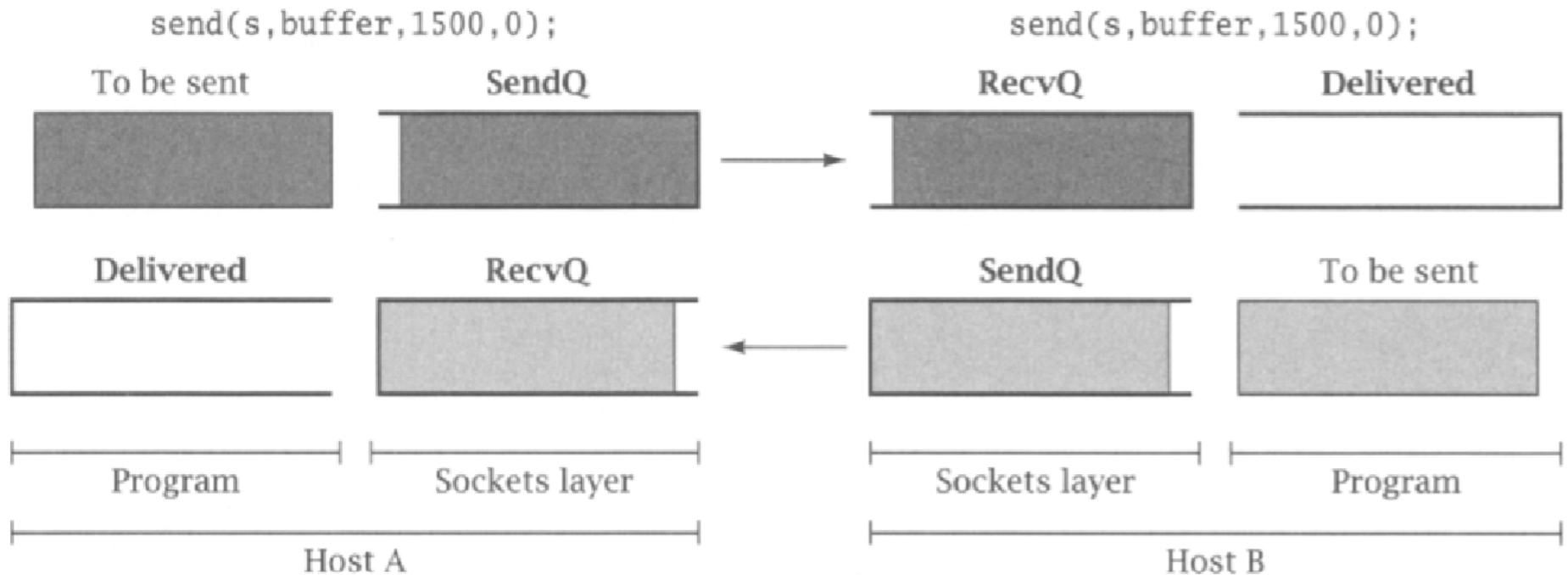


- 1 First send call (1000 bytes)
- 2 Second send call (2000 bytes)
- 3 Third send call (5000 bytes)

# WHEN DOES BLOCKING OCCUR?

- SendQ size: **SQS**
- RecvQ size: **RQS**
- `send(s, buffer, n, 0);`
  - $n > \text{SQS}$ : blocks until  $(n - \text{SQS})$  bytes xfered to RecvQ
  - If  $n > (\text{SQS} + \text{RQS})$ , blocks until receiver calls `recv()` enough to read in  $n - (\text{SQS} + \text{RQS})$  bytes
- How does this lead to deadlock?
  - Trivial cause: both sides call `recv()` w/o sending data

# MORE SUBTLE REASON FOR DEADLOCK



- SendQ size = 500; RecvQ size = 500

