SOCKETS PROGRAMMING AND THE NET PACKAGE

George Porter
Jan 11 and 13, 2022

UC San Diego

ATTRIBUTION

- 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)

UC San Diego

BERKELEY SOCKETS API

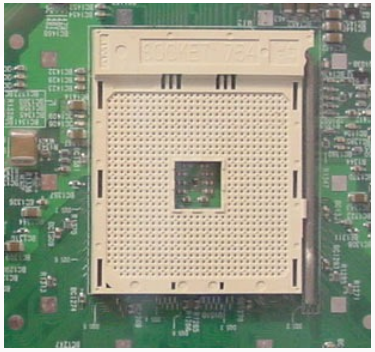
- From UC Berkeley (1980s)
- Most popular network API
- Ported to various OSes, various languages
  - Windows Winsock, BSD, OS X, Linux, Solaris, …
  - Socket modules in Java, Python, Perl, …
- Similar to Unix file I/O API
  - In the form of *file descriptor* (sort of handle).
  - Can share the same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

TYPES OF SOCKETS

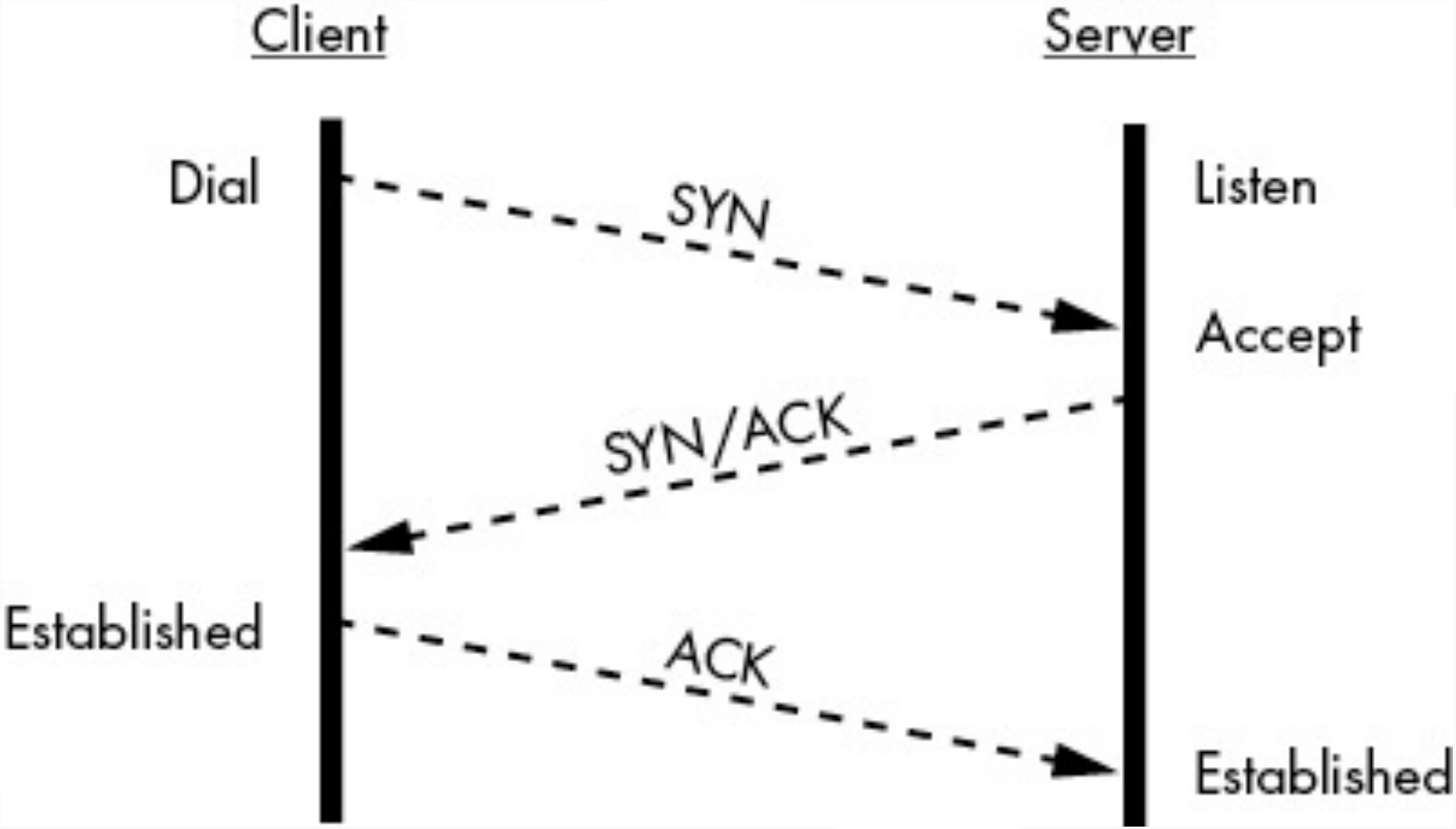
TCP

- Connection-oriented
  - Requires connection establishment & termination
- Reliable delivery
  - In-order delivery
  - Retransmission
  - No duplicates
- High variance in latency
  - Cost of the reliable service
- File-like interface (streaming)
- E.g., HTTP, SSH, FTP, …

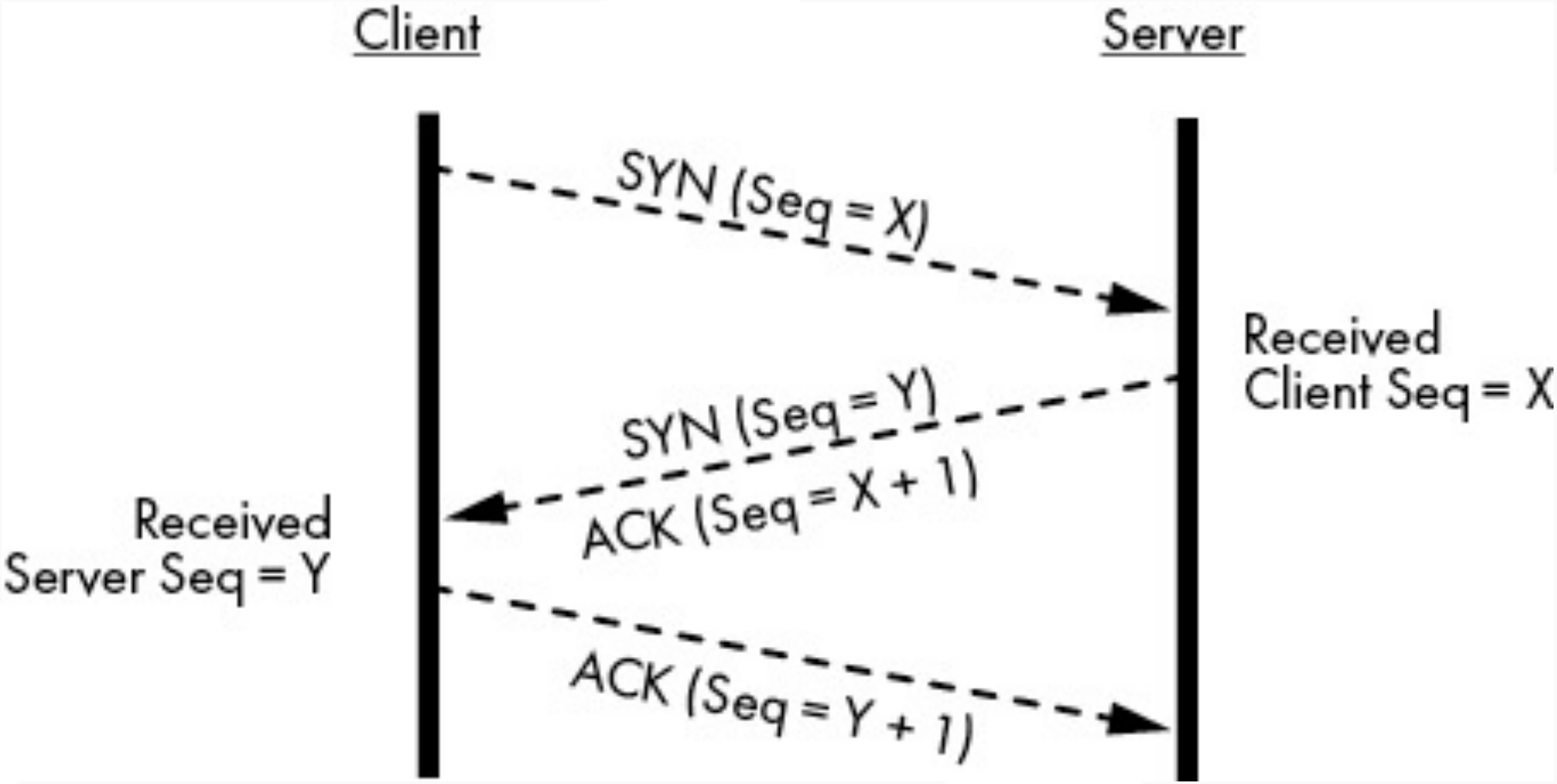
UDP

- 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, AOD, …

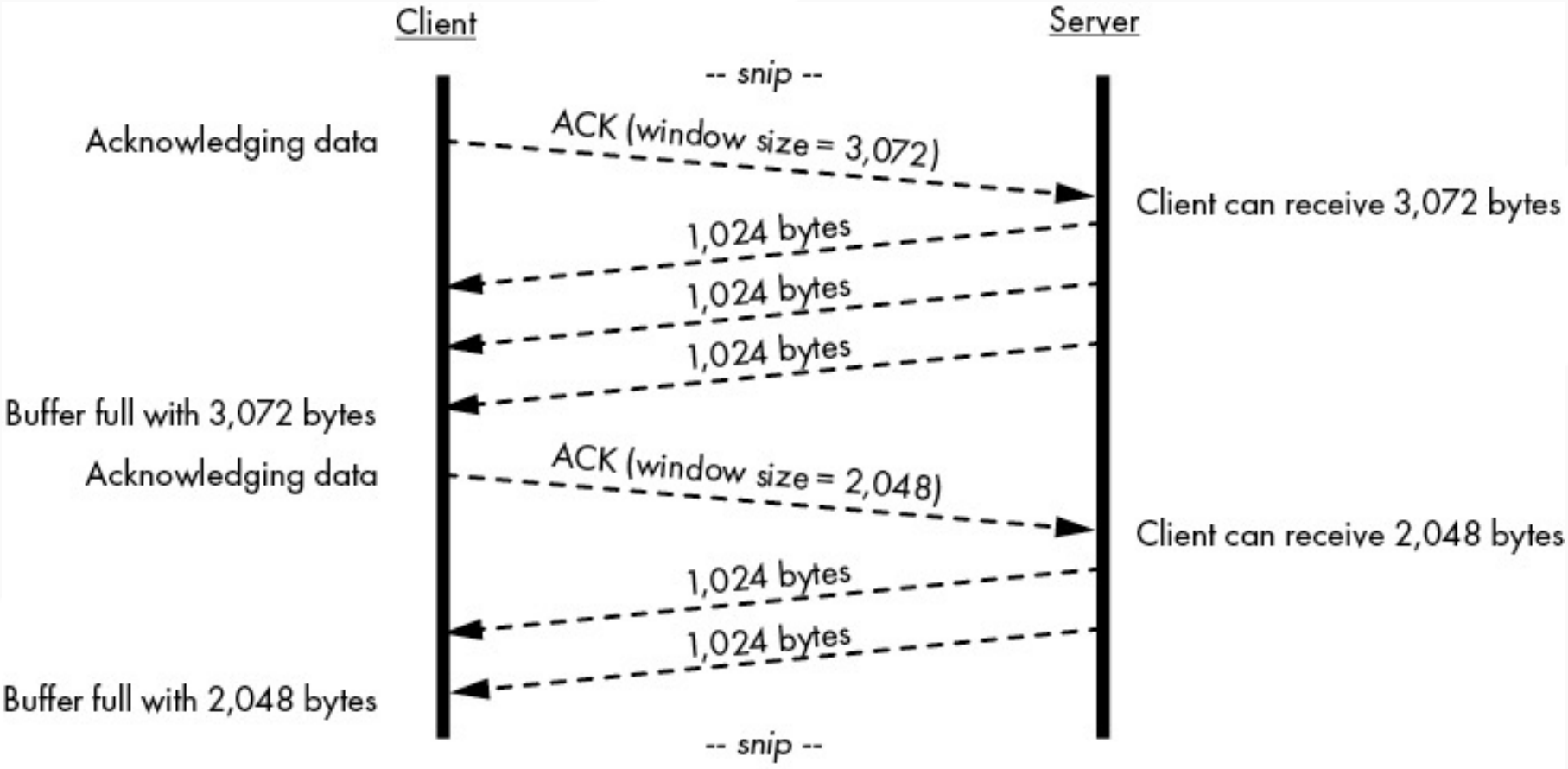
TCP CONNECTION ESTABLISHMENT



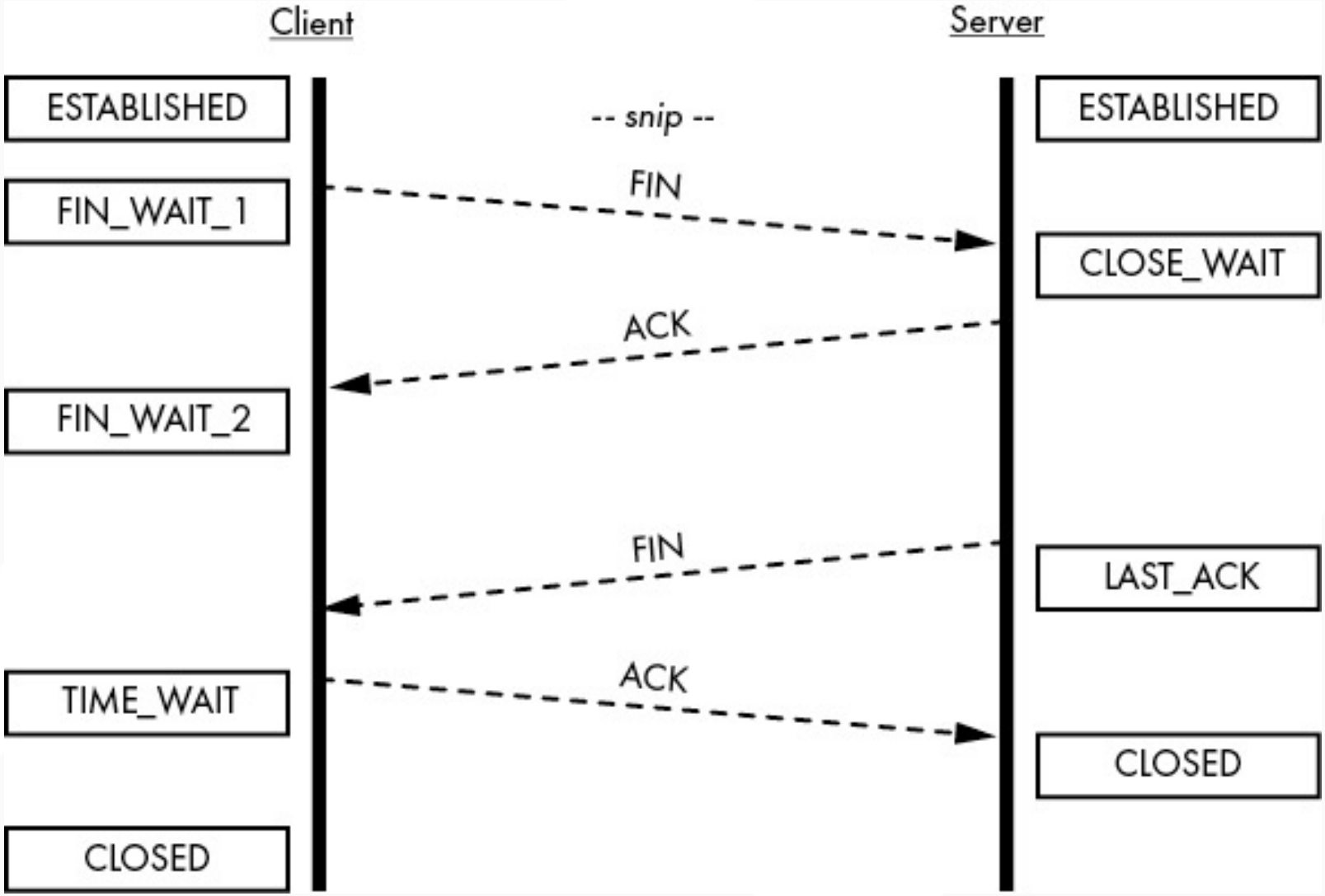
SEQUENCE NUMBERS



RECEIVER-WINDOW BASED FLOW CONTROL

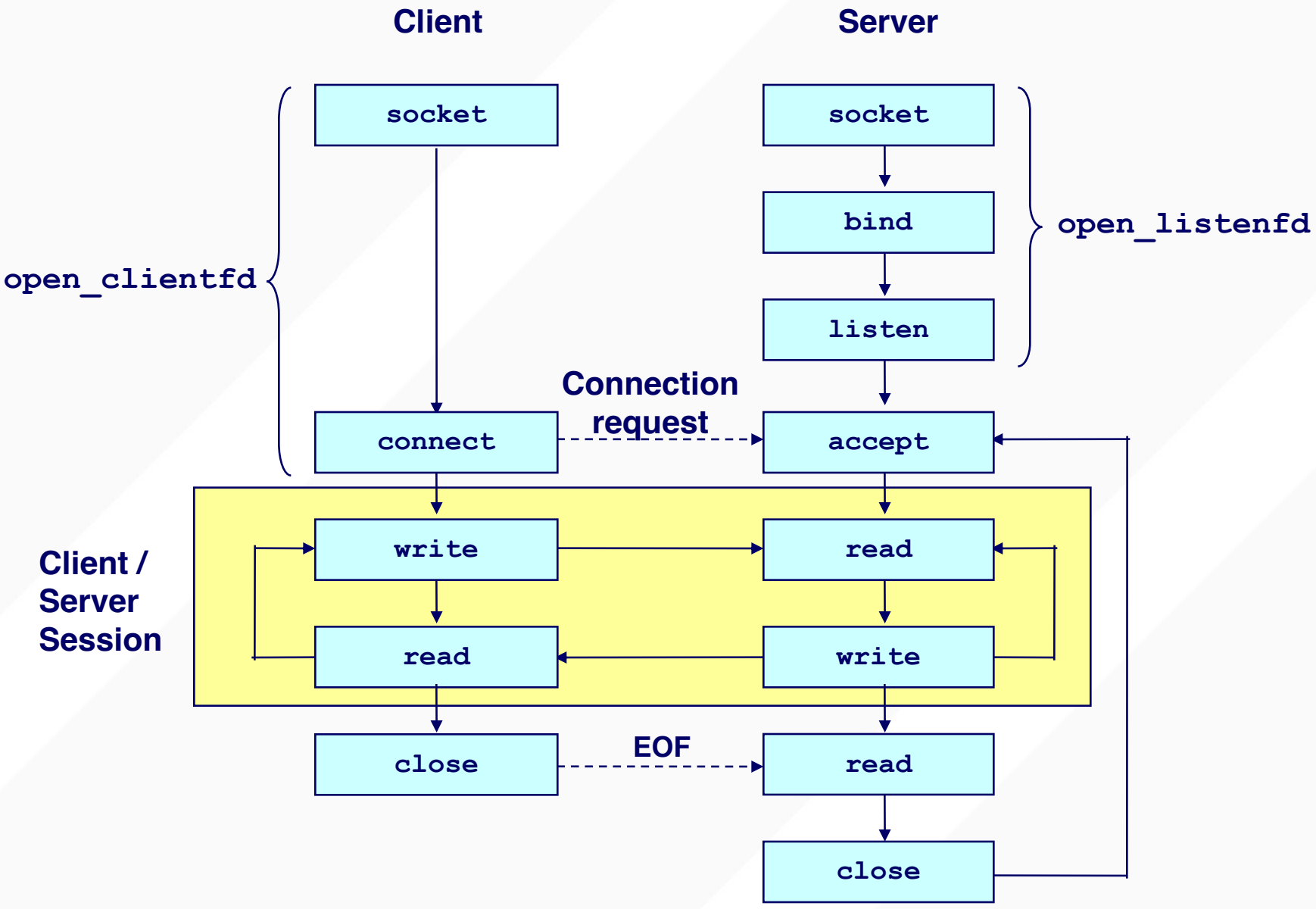


TCP STATE.MACHINE STAGES



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.

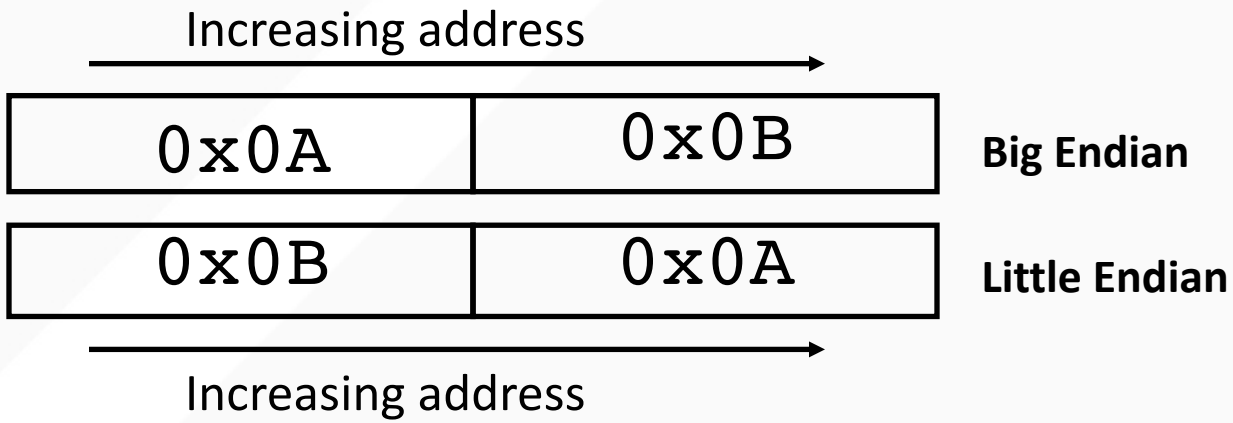
```
struct sockaddr_in sin;
memset(&sin, 0, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

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
  - htons() and ns

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()

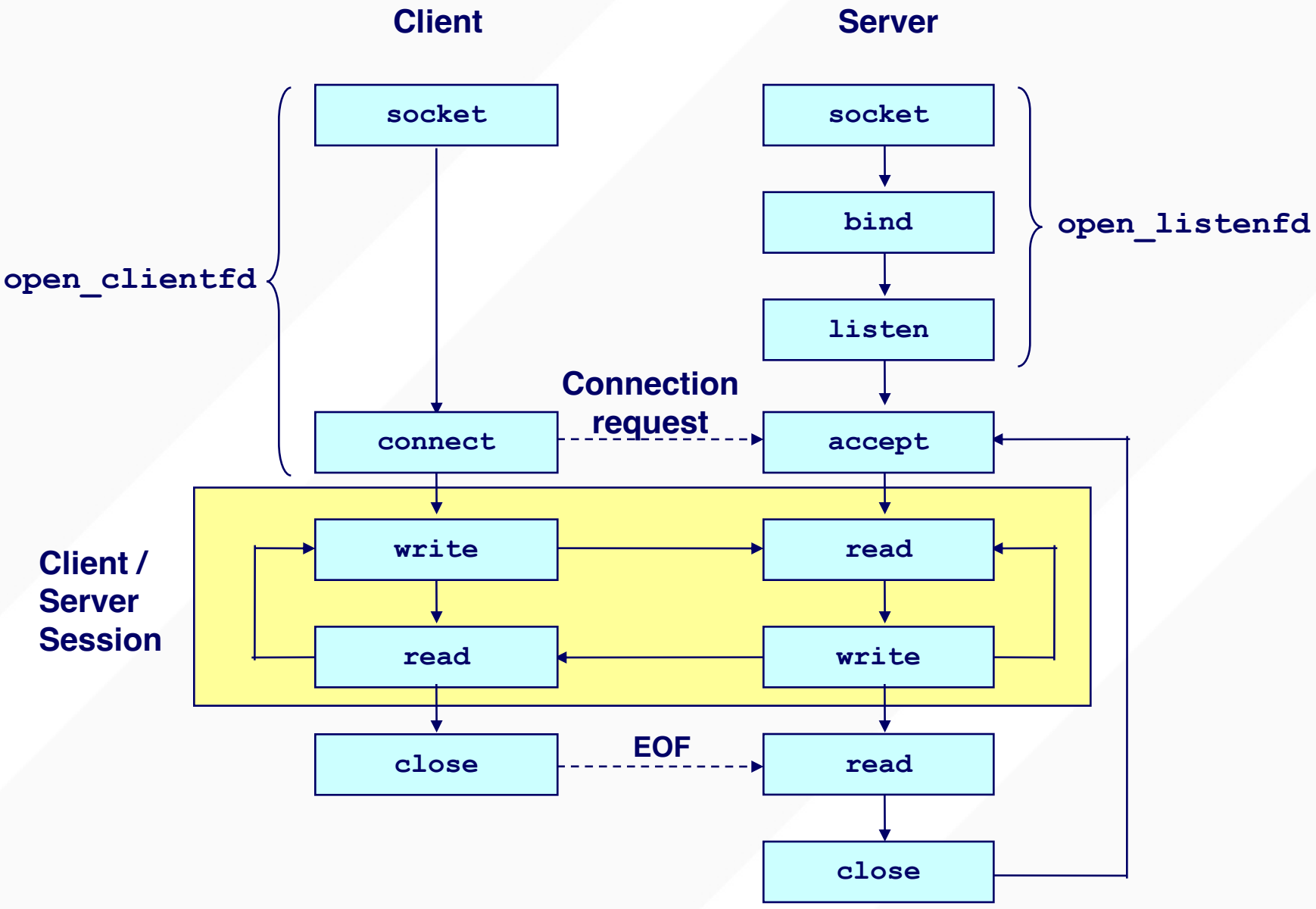
- Socket is active, by default
- We need to make it passive to get connections.

```
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()`, `writev()`
- **`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)

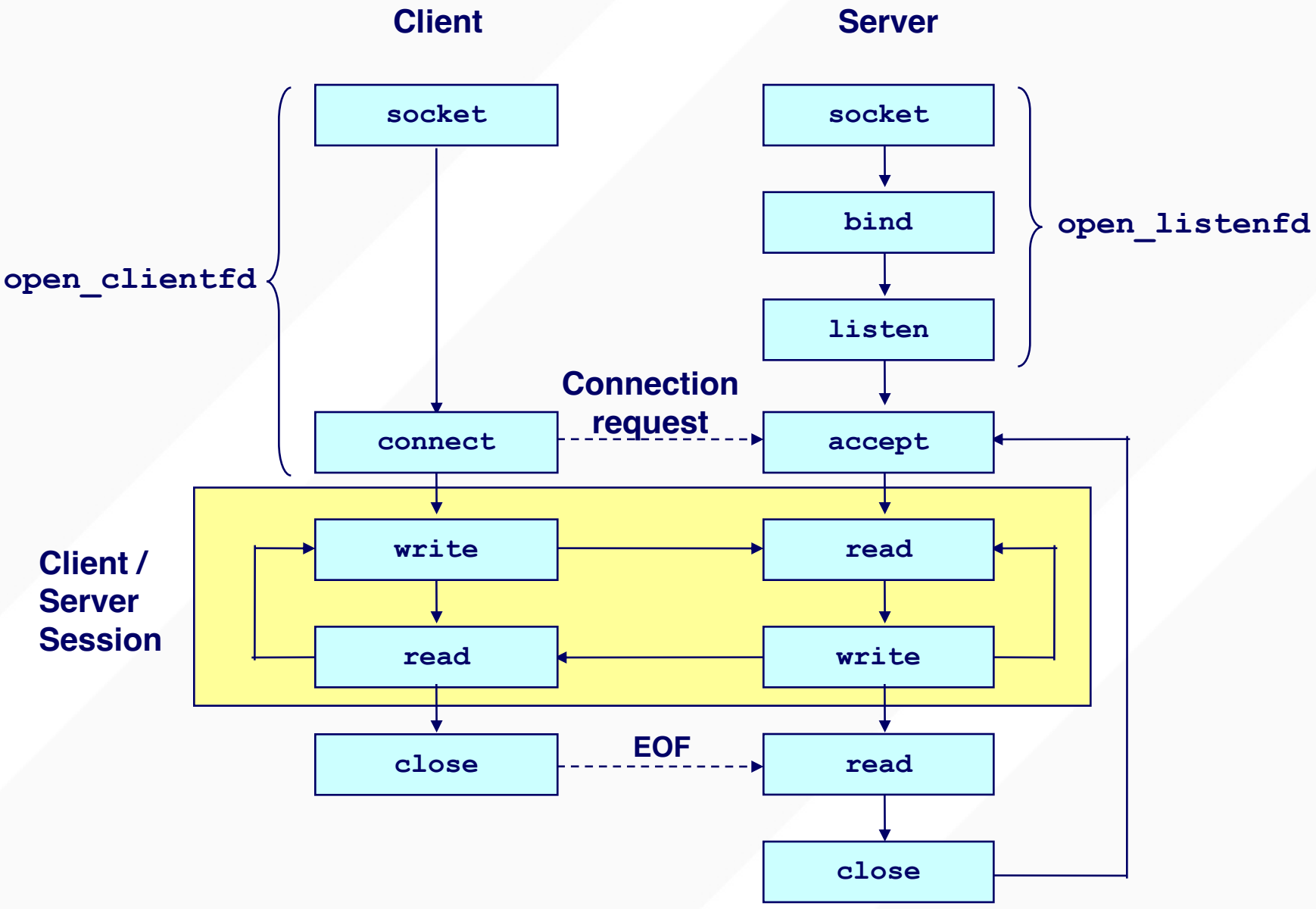
```
// after use the socket
close(sock);
```

- **`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)



TCP SOCKET EXAMPLE IN GO

```
package main

import (
	"flag"
	"log"
	"net"
	"strconv"
)

func main() {
	port := flag.Int("port", 3333, "Port to accept connections on.")
	host := flag.String("host", "127.0.0.1", "Host or IP to bind to")
	flag.Parse()

	l, err := net.Listen("tcp", *host+":"+strconv.Itoa(*port))
	if err != nil {
		log.Panicln(err)
	}
	log.Println("Listening at '"+*host+"' on port", strconv.Itoa(*port))
	defer l.Close()

	for {
		conn, err := l.Accept()
		if err != nil {
			log.Panicln(err)
		}

		go handleRequest(conn)
	}
}
```

Code courtesy of @golergka

TCP SOCKET EXAMPLE IN GO (CON'T)

```
func handleRequest(conn net.Conn) {
	log.Println("Accepted new connection.")
	defer conn.Close()
	defer log.Println("Closed connection.")

	for {
		buf := make([]byte, 1024)
		size, err := conn.Read(buf)
		if err != nil {
			return
		}
		data := buf[:size]
		log.Println("Read new data from connection", data)
		conn.Write(data)
	}
}
```

Code courtesy of @golergka

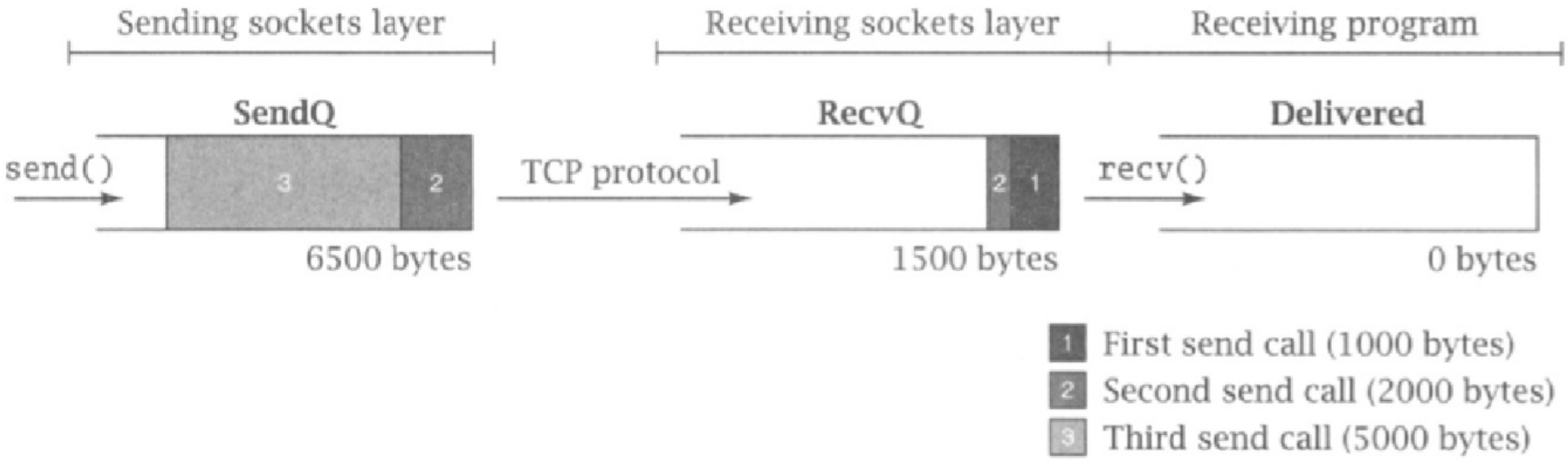
TURING SENDER/RECEIVER

- Based on ch03/dial_test.go

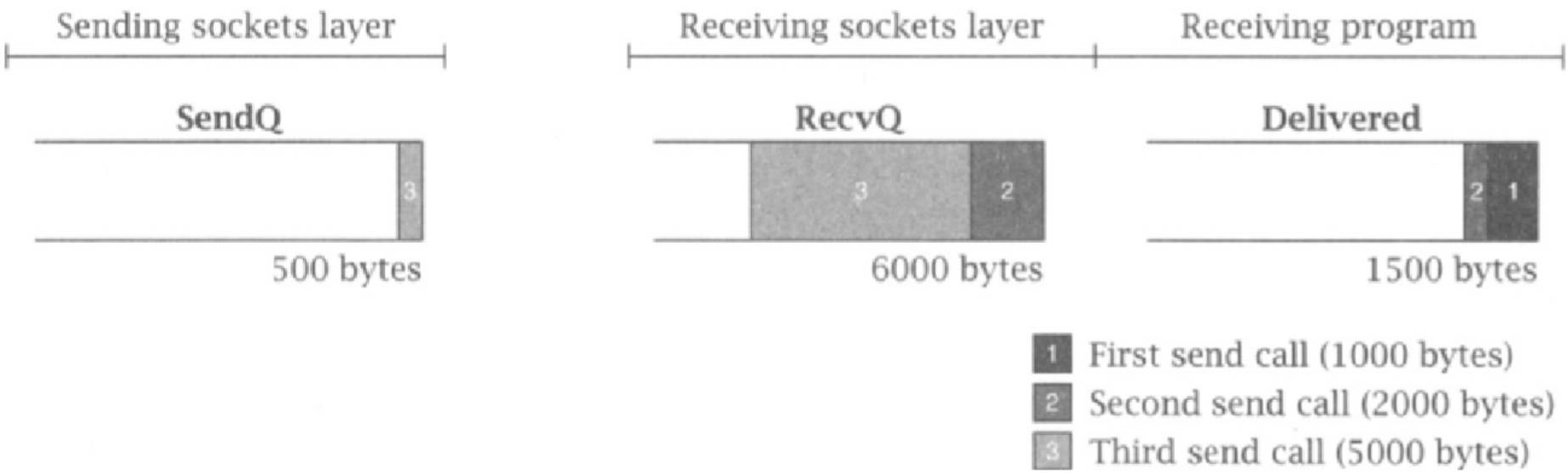
DIGGING INTO SEND() A BIT MORE

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

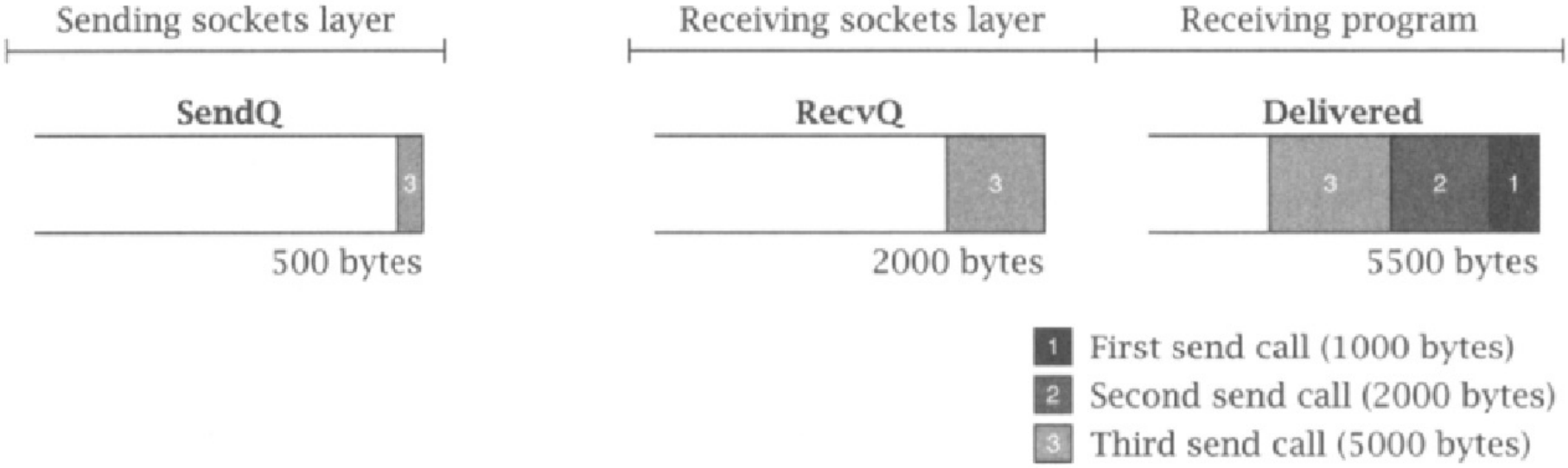
AFTER 3 SEND() CALLS



AFTER FIRST RECV()



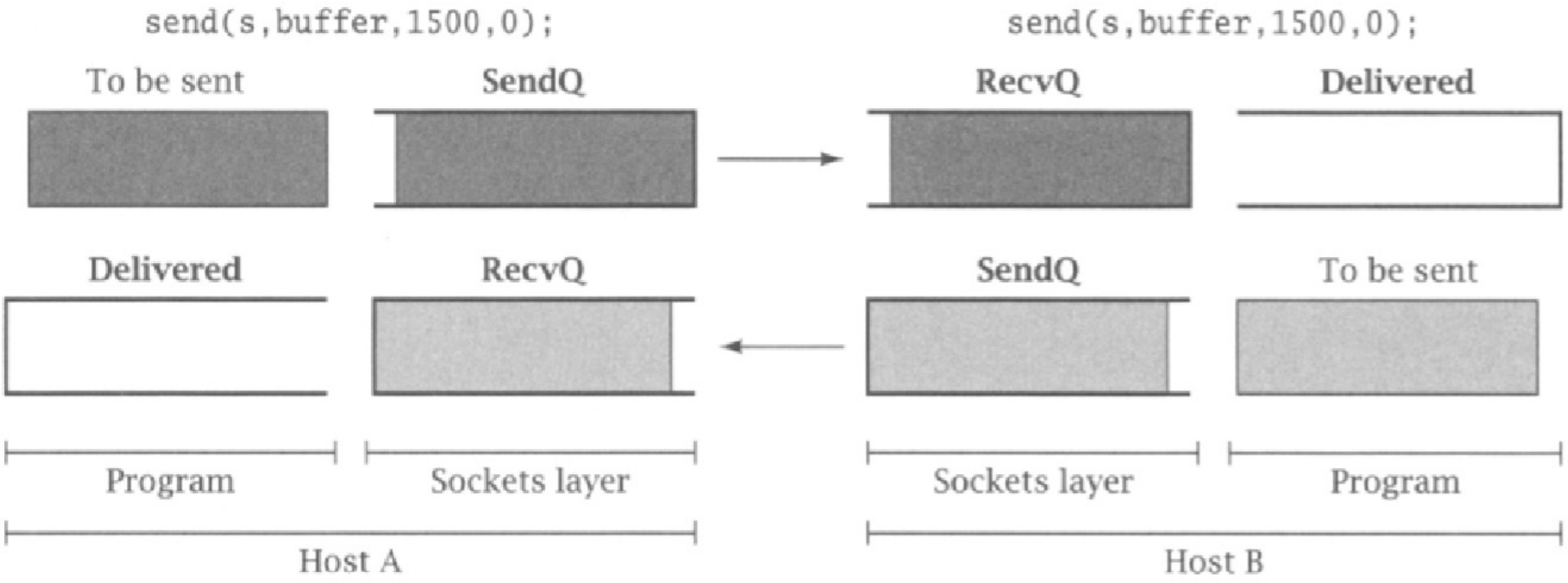
AFTER ANOTHER RECV()



WHEN DOES BLOCKING OCCUR?

- SendQ size: SQS
- RecvQ size: RQS
- send(s, buffer, **n**, 0);
  - n>SQS: blocks until (n – SQS) bytes xfered to RecvQ
  - If n>(SQS+RQS), blocks until receiver calls recv() enough to read in n-(SQS+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

UC San Diego