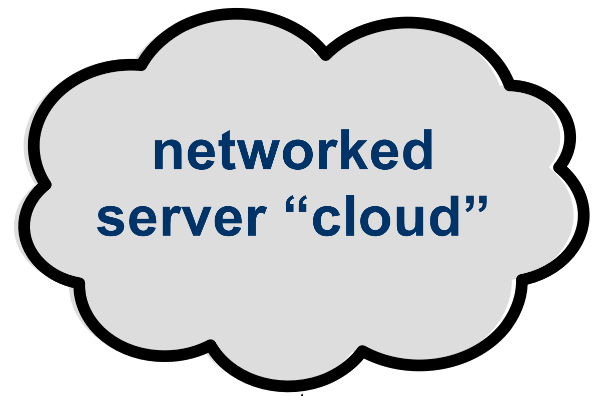
Duke Systems

Of servers and sockets

Jeff Chase
Duke University

Servers and the cloud

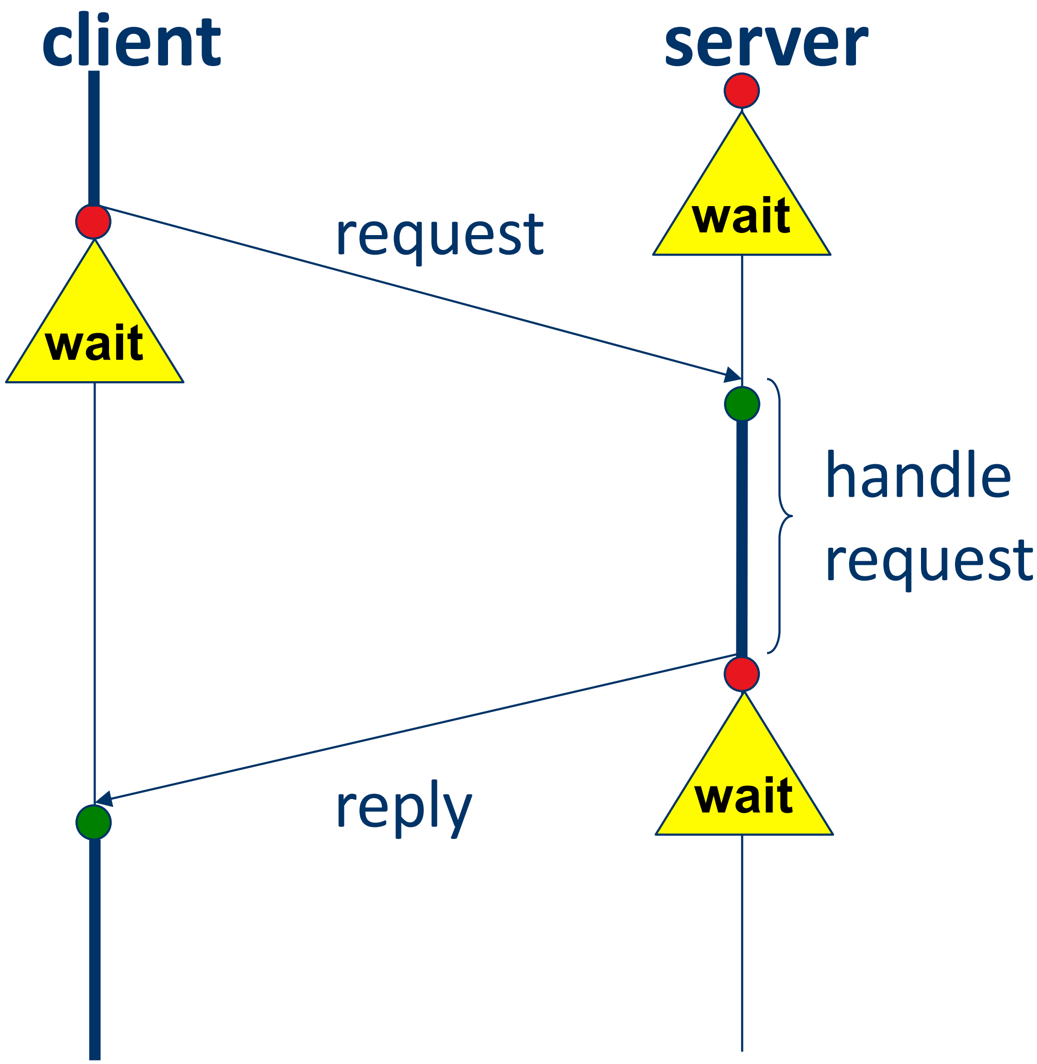
Where is your application?
Where is your data?
Where is your OS?



<u>**Cloud and Software-as-a-Service (SaaS)**</u>

Rapid evolution, no user upgrade, no user data management.
Agile/elastic deployment on clusters and virtual cloud utility-infrastructure.

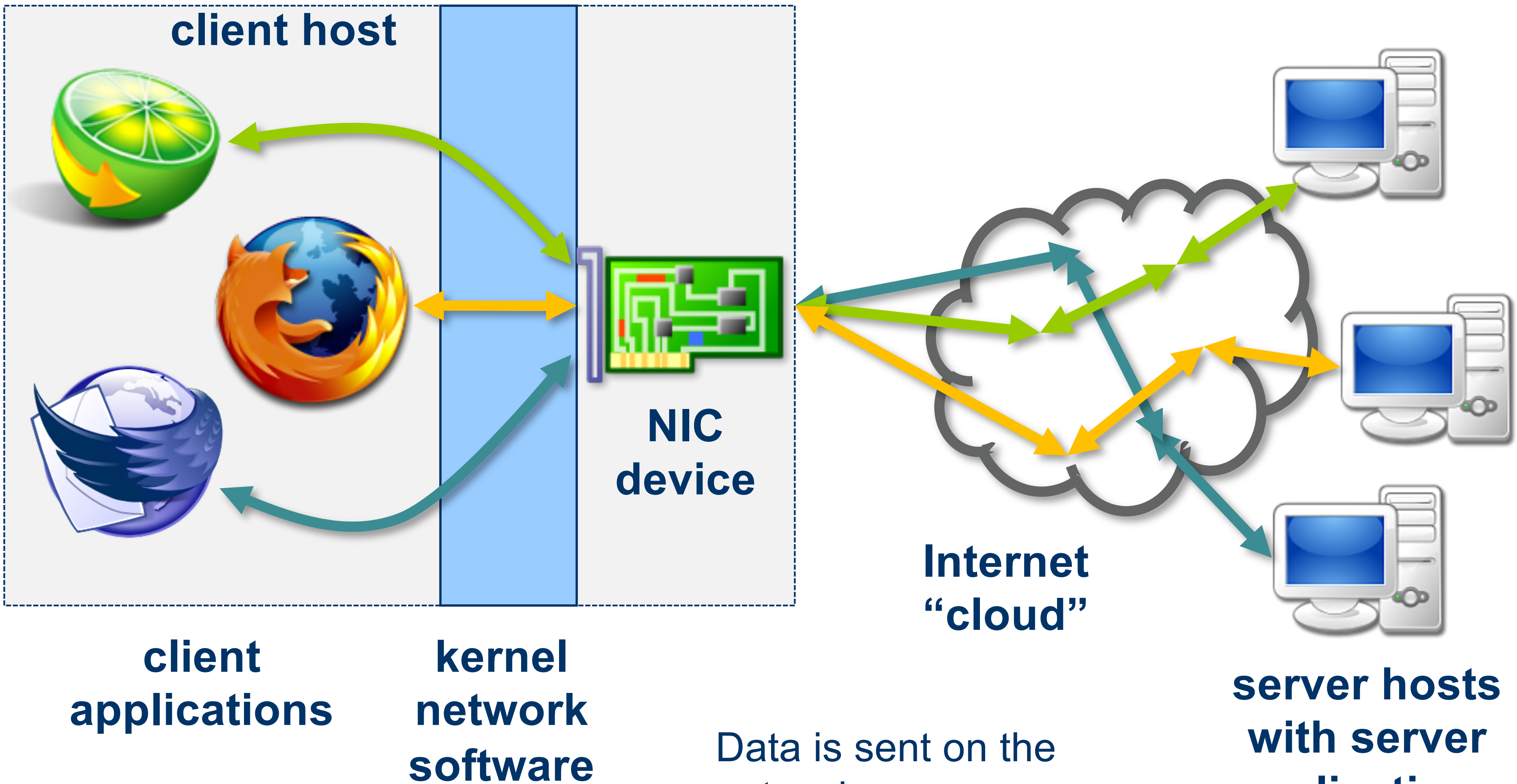
Request/reply messaging



Client initiates.
Server accepts.
Client waits.
Server replies.

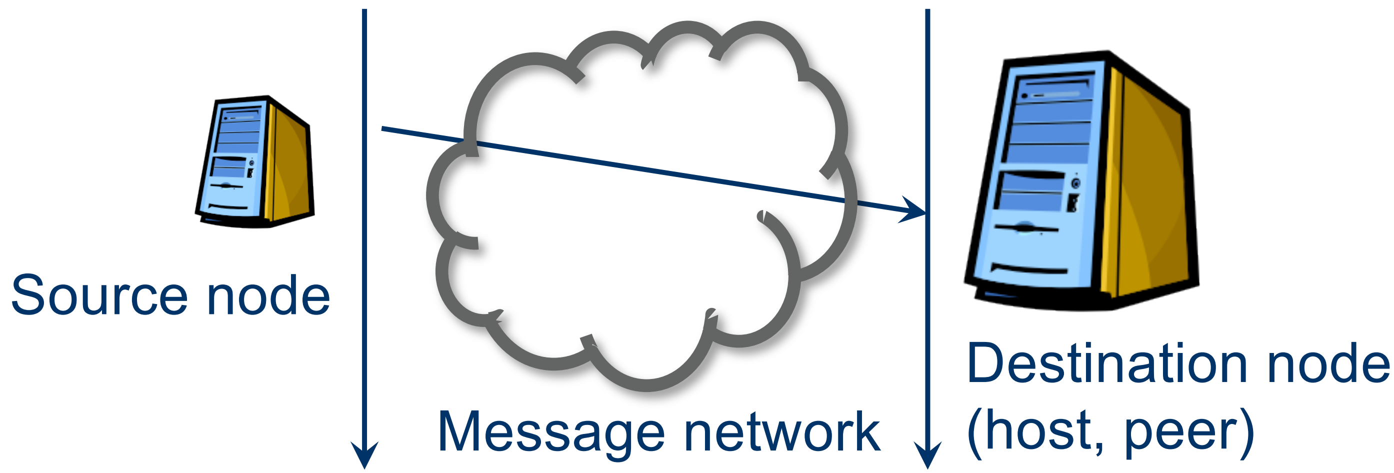
RPC, HTTP/S (web)

Networked services: big picture



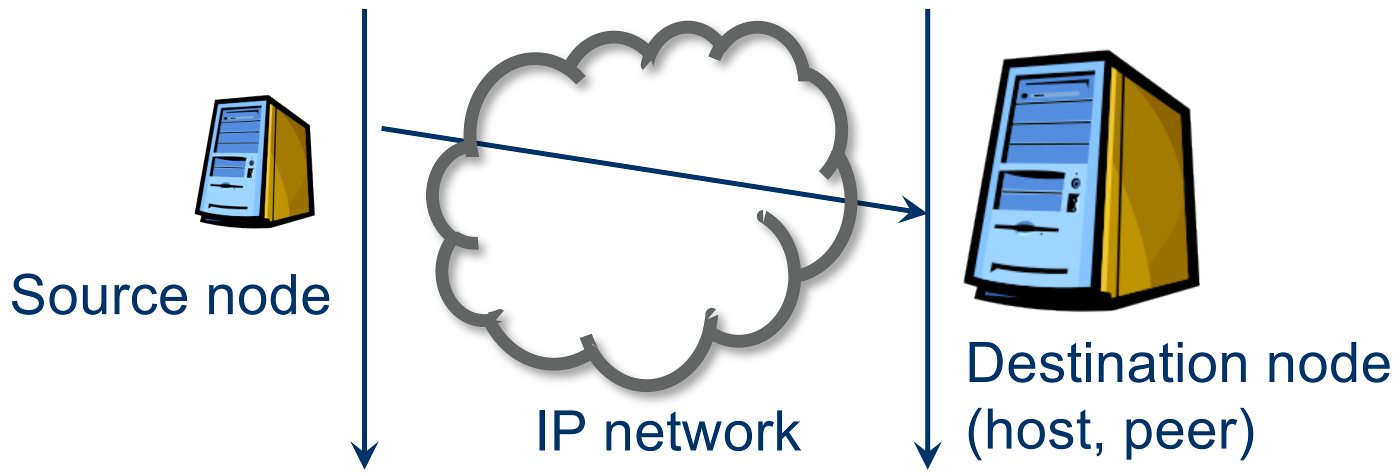
Data is sent on the network as messages called **packets**.`

Network communication



- Computers (nodes, hosts) send and receive **packets**.
- A packet has a maximum size. e.g., 1536 bytes.
- A **message** could be sent as one or more packets.
- Nodes often communicate with a **stream** (sequence, flow) of many packets over a **connection** to a peer.

Internet communication



- We consider networks using Internet Protocol (IP).
- IP networks may delay or discard packets arbitrarily. So
- Hosts run a **transport protocol** to communicate reliably.
- E.g. Transmission Control Protocol (TCP) for **streams**.
- "Everybody uses it"—the foundation of the Web.

TCP/IP communication

- TCP is a reliable stream protocol implemented in host software "above" IP, **end-to-end**.
  - https://www.rfc-editor.org/info/rfc793 (1981)
- A client initiates a TCP **connection** to a server.
- A TCP connection enables bi-directional **stream** communication between a pair of endpoints (IP+**port**).

Client/server connection

Using the socket syscall API

request

“`GET /images/fish.gif HTTP/1.1`”

reply

client (initiator)

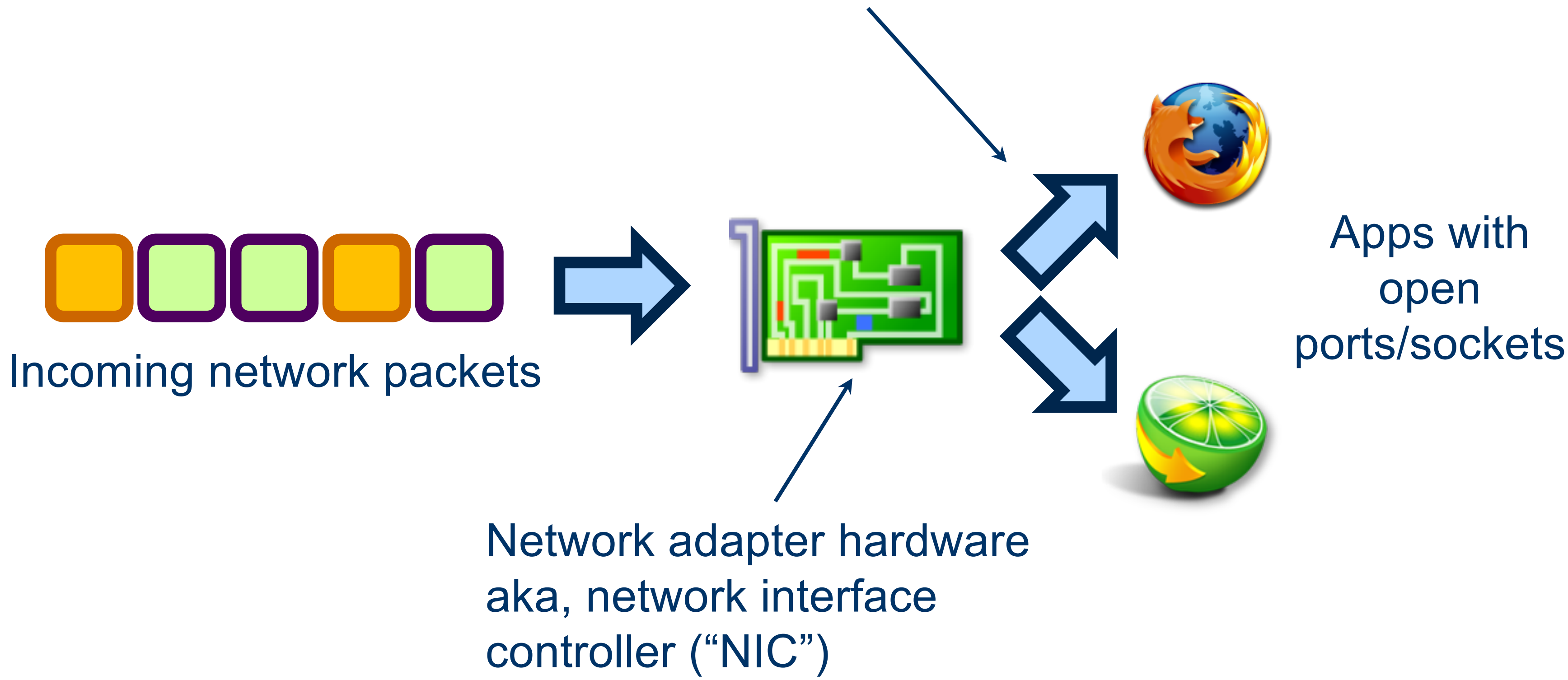
server

```
sd = socket(…);
connect(sd, name);
write(sd, request…);
read(sd, reply…);
close(sd);
```

```
s = socket(…);
bind(s, name);
listen(s, 10);
sd = accept(s);
read(sd, request…);
write(sd, reply…);
close(sd);
```

Ports and packet demultiplexing

The IP network carries data **packets** addressed to a destination node (host named by IP address) and **port**. Kernel network stack **demultiplexes** incoming network traffic: choose process/socket to receive it based on destination port and source address.



TCP/IP Ports

- Each IP transport endpoint on a host has a logical **port number** (16-bit integer) that is unique on that host.
  - Source/dest port is named in every packet.
  - Receiving kernel looks at port to demultiplex incoming traffic.
- What port number to connect to?
  - Ports 1023 and below are 'reserved' and **privileged**: generally you must be root/admin/superuser to bind to them.
  - Used for **well-known ports** for common services
  - Look at **/etc/services**
- Clients need a return port, but it can be an **ephemeral** port assigned dynamically by the kernel.

Server listens on a port

```
struct sockaddr_in socket_addr;
sock = socket(PF_INET, SOCK_STREAM, 0);
…

memset(&socket_addr, 0, sizeof socket_addr);
socket_addr.sin_family = PF_INET;
socket_addr.sin_port = htons(port);
socket_addr.sin_addr.s_addr = htonl(INADDR_ANY);

if (bind(sock, (struct sockaddr *)&socket_addr, sizeof socket_addr) < 0) {
      perror(”bind failed");
      exit(1);
}
listen(sock, 10);
```

Port number is passed as an input.

Let kernel fill in this host’s IP address.

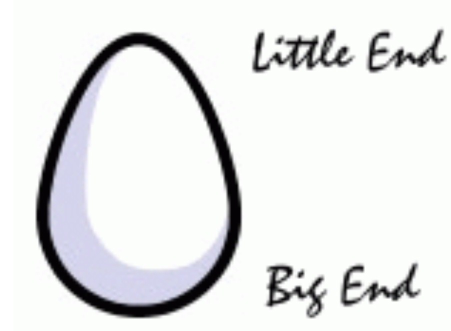
htons and htonl byte-swap to network endian.

Bind fails if server process is already running on the port.

Illustration only

What are htons() and htonl()?

Endianness



Lilliput and Blefuscu are at war over which end of a soft-boiled egg to crack.

Gulliver's Travels
1726

A silly difference among machine architectures creates a need for byte swapping when unlike machines exchange data over a network. Intel processors are little-endian, Internet is big-endian: the most significant byte is transmitted on the network first.

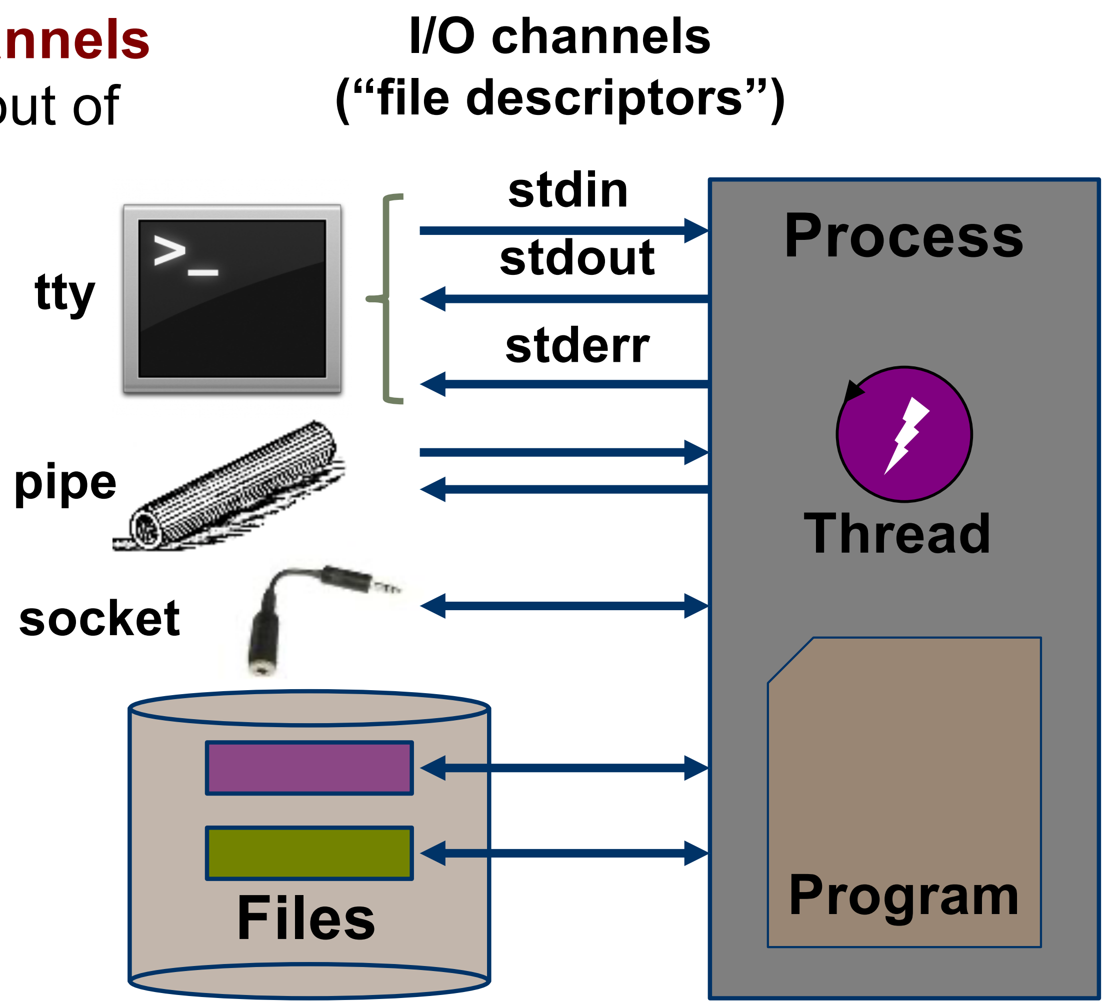
Unix process view: data

A process has multiple **channels** for data movement in and out of the process (I/O).

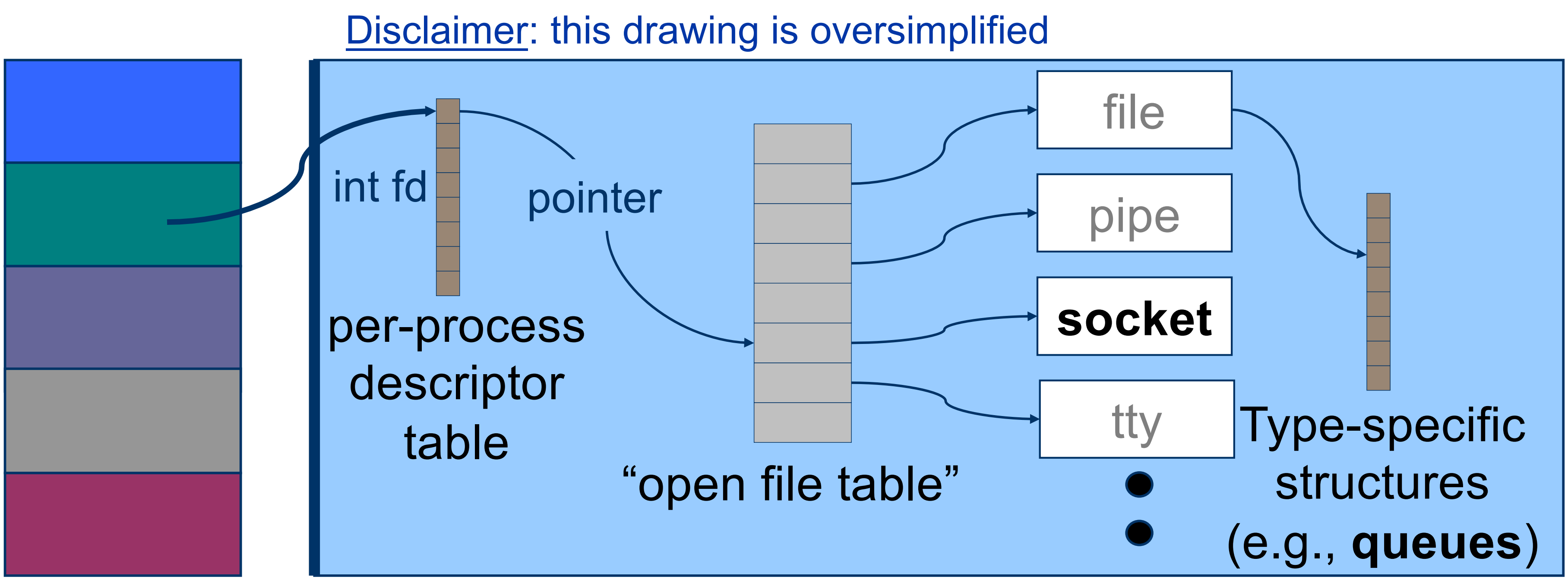
The channels are typed.

Each channel is named by an I/O **descriptor** (called a “file descriptor”).

A file descriptor is an integer value assigned by the kernel (e.g., at **open**).

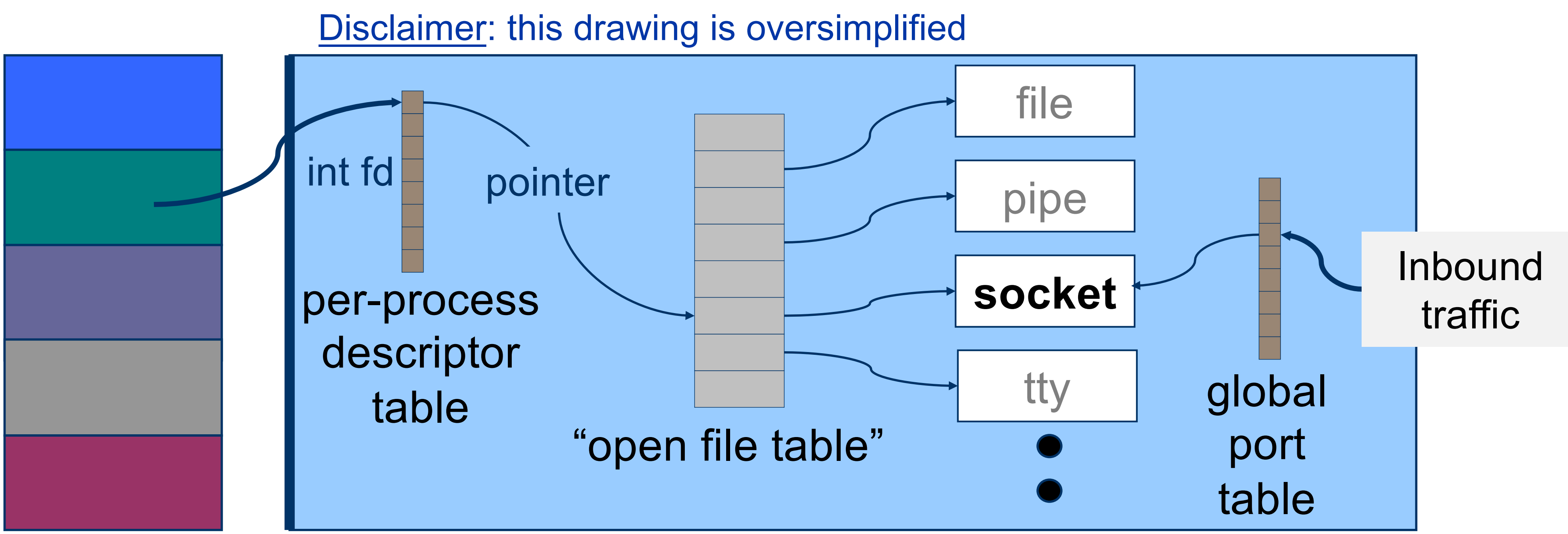


“File” (I/O) descriptors in Unix



An **I/O descriptor** is a positive integer value in an integer variable (e.g., fd). The user program passes it to the kernel to identify the channel for each I/O syscall. The kernel uses the fd to index into an internal per-process table to find a typed object for the channel to operate on it.

Socket descriptors in Unix



There's no magic here: processes use **read/write** (and similar syscalls called **send* and recv***) to operate on sockets.

Deeper in the kernel, sockets are handled differently from files, pipes, etc. Sockets are the entry/exit point for the **network protocol stack**.

Server accept loop

A trivial example

```
while (1) {
    int acceptsock = accept(sock, NULL, NULL);
    char *input = (char *)malloc(1024*sizeof (char));
    recv(acceptsock, input, 1024, 0);
    int is_html = 0;
    char *contents = handle(input,&is_html);
    free(input);

    …send response…

    close(acceptsock);
}
```

Accept returns **another** socket for connection.

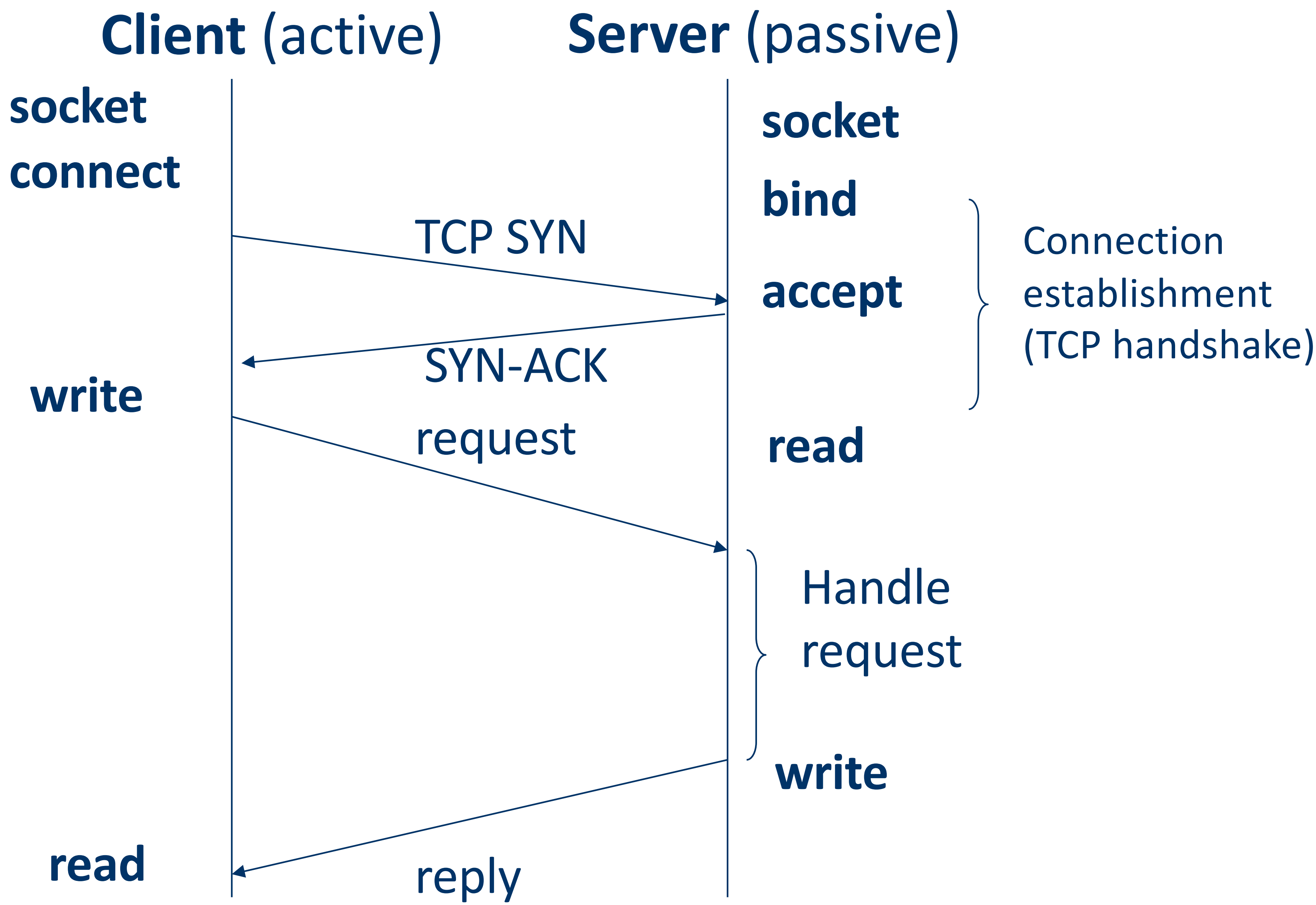
Accept blocks to wait for a connection.

Use **read/write** or **send/recv** syscalls for I/O on connection.

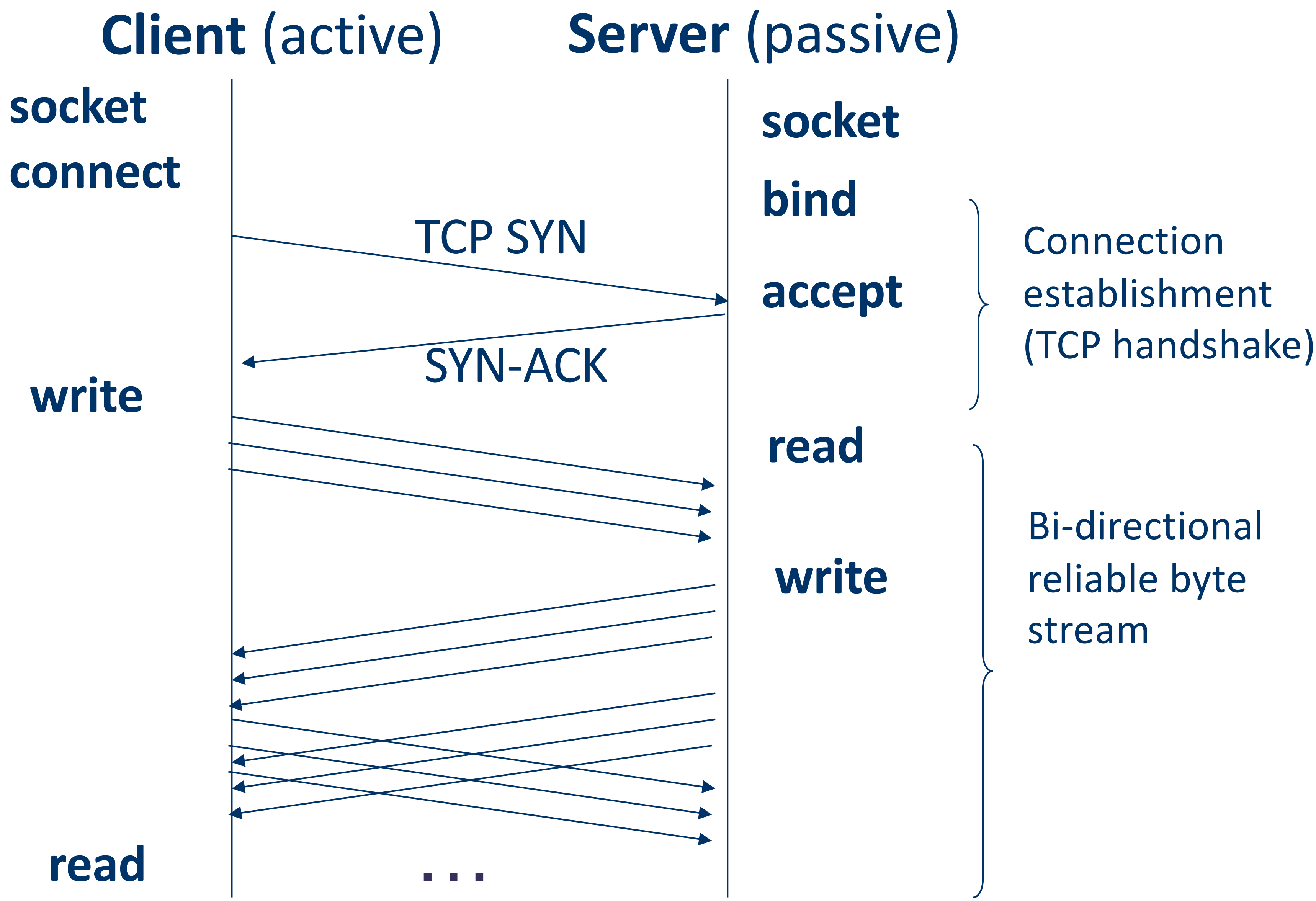
Always **close** descriptor when done with it, e.g., else client waits.

Illustration only

Request/reply over a connection



Streams over a TCP/IP connection



Send HTTP/HTML response

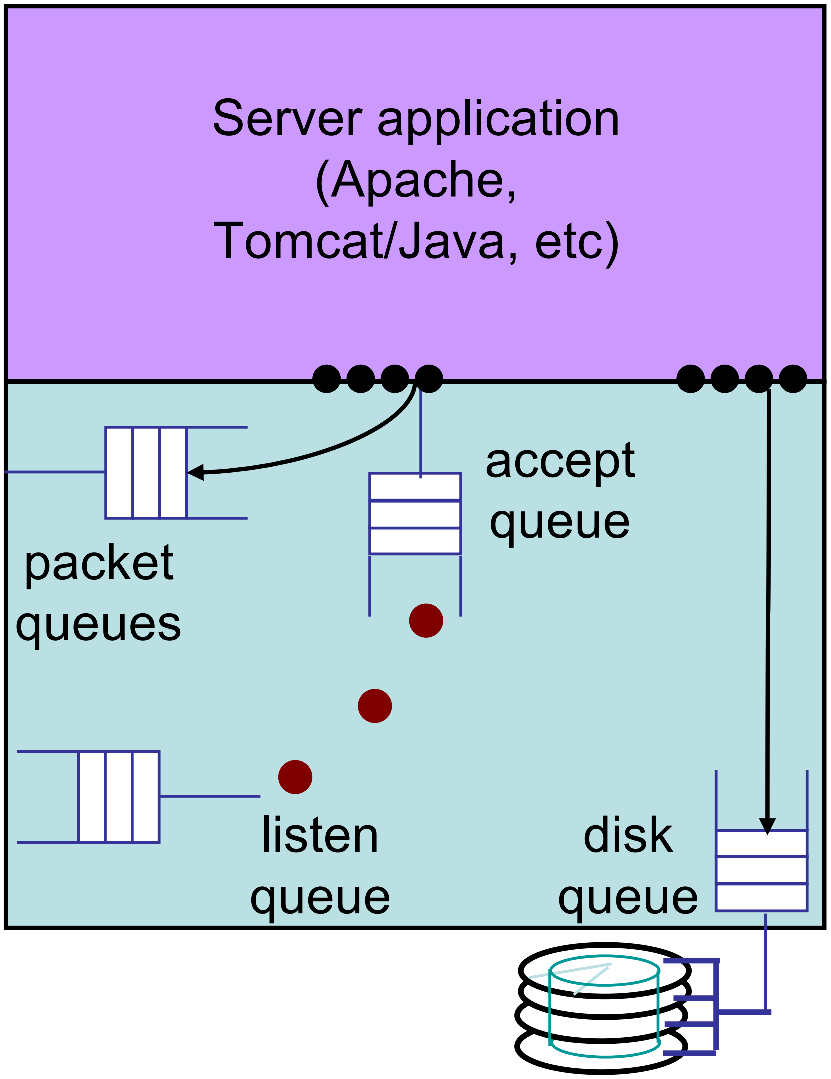
```
const char *resp_ok = "HTTP/1.1 200 OK\nServer: BuggyServer/1.0\n";
const char *content_html = "Content-type: text/html\n\n";

send(acceptsock, resp_ok, strlen(resp_ok), 0);
send(acceptsock, content_html, strlen(content_html), 0);
send(acceptsock, contents, strlen(contents), 0);
send(acceptsock, "\n", 1, 0);

free(contents);
```

Illustration only

Know the socket system calls

- **Socket**. Create/open a **socket** that can be one side of a connection. **Does not block.**
- **Bind**. Advertise an open socket as an open **port** on the network for clients to connect to (a ServerSocket). **Does not block.**
- **Connect** an open socket to a server port. Used by client. Blocks waiting for an accept reply from server.
- **Accept** a connect request from a client. Used by server. Blocks waiting for next client connect request.
- **Read/write** or **send/receive**. Write data into a connected socket, or read data from a connected socket. Blocks waiting for data or for space. (soda machine)

Inside your Web server



Server operations

create **socket**(s)

bind to port number(s)

listen to advertise port

wait for client to arrive on port

accept client connection

read or **recv** request

write or **send** response

close client connection

Not tested

The listen queue

Why do we have both a listen queue and an accept queue?
This is part of the next level of detail on TCP socket behavior to know about. It explains some of the behaviors we observe. E.g., why you can connect() to a server process even if the server process does not accept() or respond to input, e.g., because you smashed it.

- After listen(), the kernel handles the connection handshake for incoming connect requests, independent of the user process.

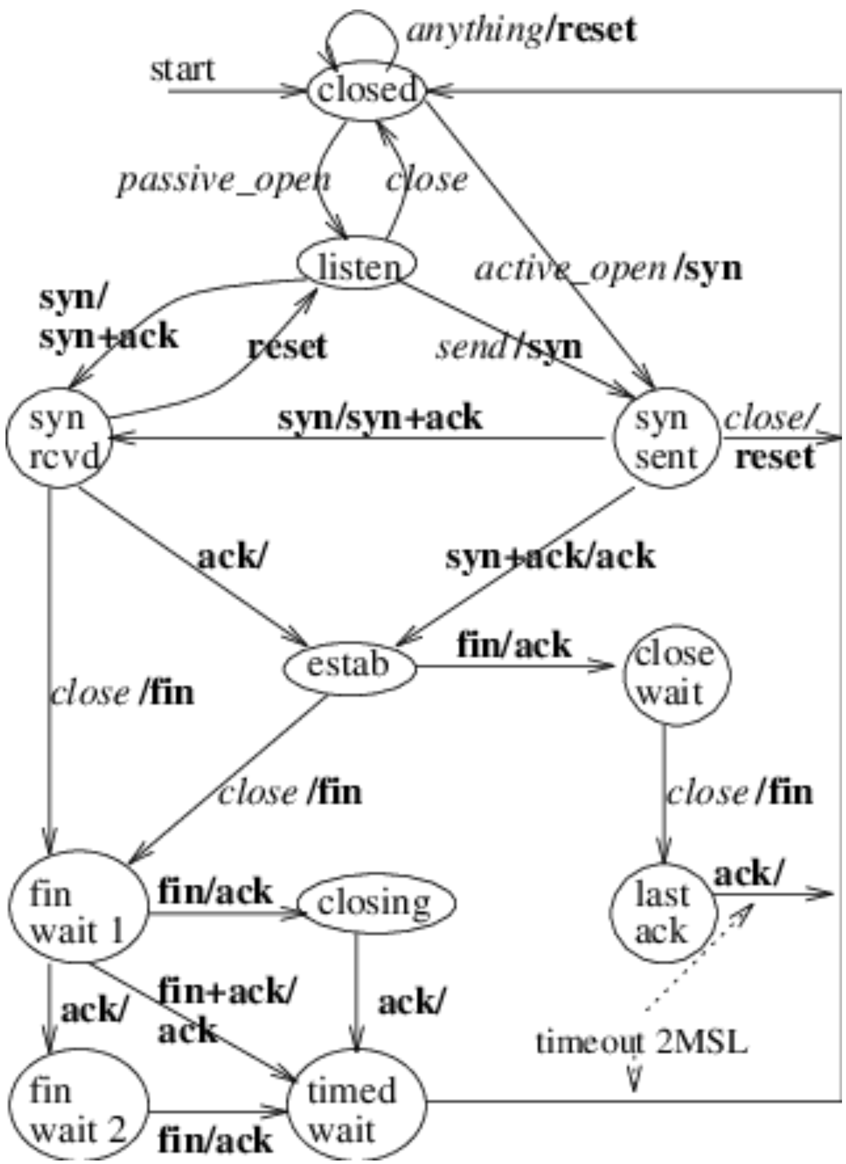
- Established connections go on the **accept queue** until (on Linux) the accept queue is full up to the specified listen() backlog (e.g., 10). If full, the kernel rejects connect requests until an accept() makes space on the queue.

- While handshake is in progress a connection is partially established ("half-open") Those go on the **listen queue**, which is also of bounded size, but separate. Putting them on the accept queue could allow a **denial-of-service** attacker to jam the server process with half-open connnections.

http://veithen.github.io/2014/01/01/how-tcp-backlog-works-in-linux.html

Not tested

TCP connection state is a FSM



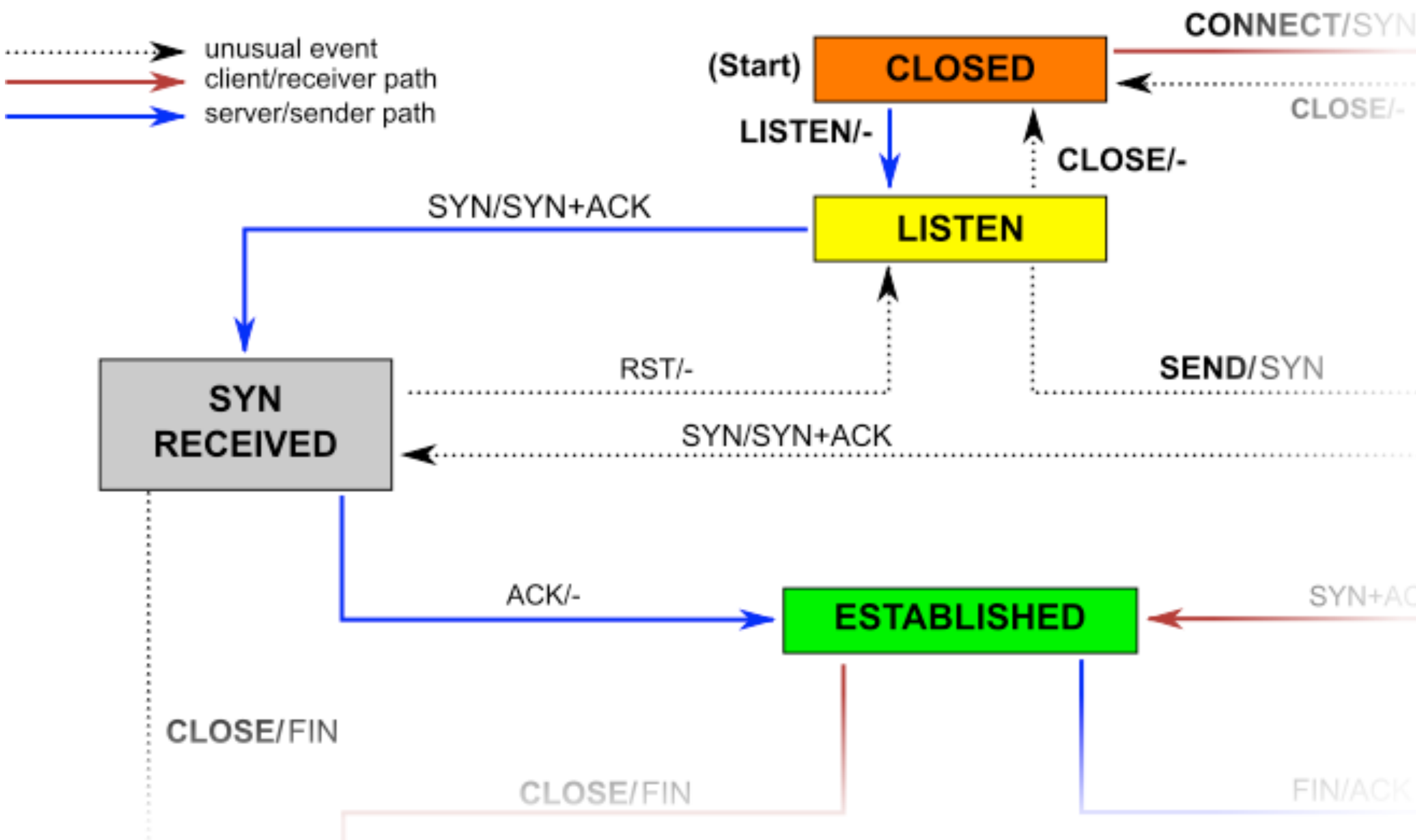
Both peers of a TCP connection track its state as a finite state machine.

Connections in **syn-sent** or **syn-rcvd** states are “half open”.

Connections in *wait* or closing states are “half closed”. These can cause a bind() for the port to fail after server restart.

Not tested

Listen



http://veithen.github.io/2014/01/01/how-tcp-backlog-works-in-linux.html