CSC209 Summer 2015 — Software Tools and Systems Programming

www.cdf.toronto.edu/~csc209h/summer/

Week 10 — July 16, 2015

Peter McCormick
pdm@cs.toronto.edu

Some materials courtesy of Karen Reid

Announcements

- **Final exam date has been determined:**
  - **Tuesday, August 11 (evening)**
  - http://www.artsci.utoronto.ca/current/exams/reminder
- No tutorial tonight

Feedbacks

Last Week Recap

- Inter-process communication techniques:
  - Low level *file descriptors* interface
  - Unidirectional *pipes*
  - Unix *signals* as software interrupt mechanism

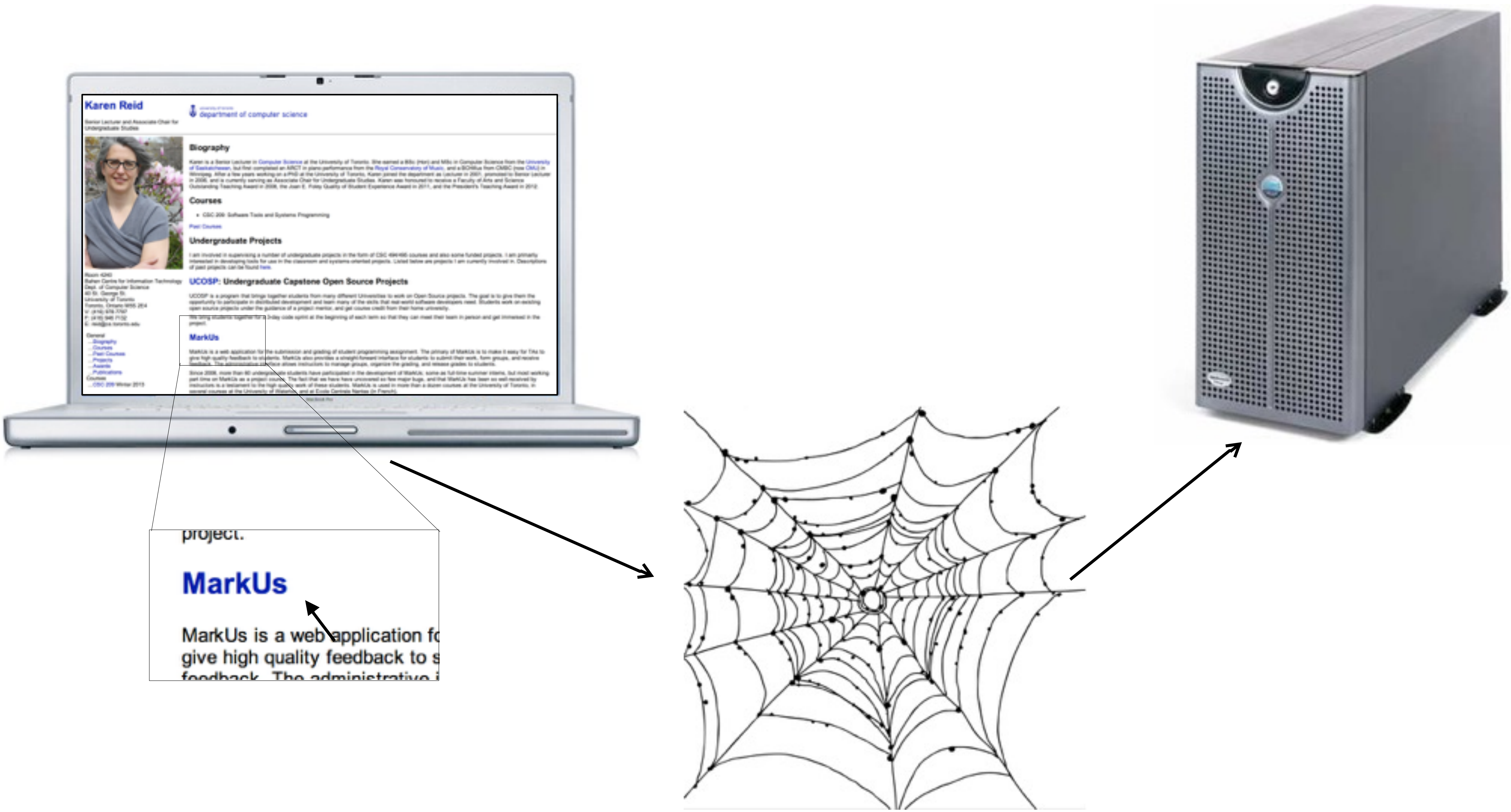
Agenda

- How the Internet works
- Programming with sockets

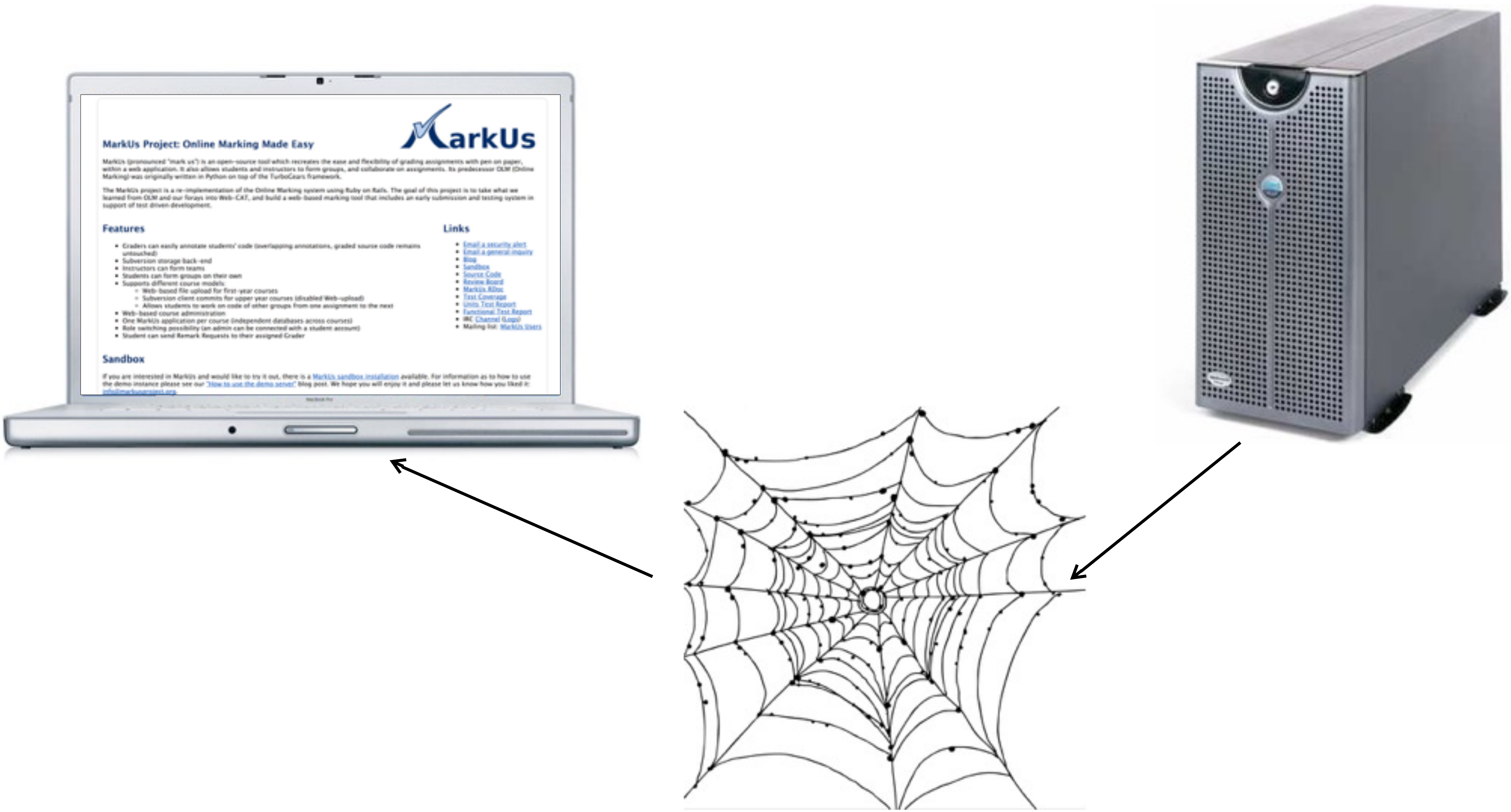
How the Internet Works

Slides courtesy of Karen Reid

Simple Web Request



Response



MarkUs
MarkUs Project: Online Marking Made Easy
Features
Links
Sandbox

The Request

- How do we ***ask*** the web server for what we want?
- How do we even ***find*** the web server in the first place?
- How do the web server and browser ***talk*** to each other?

HTTP Request

Request:

```
GET / HTTP/1.1
Host: markusproject.org
...
```

Reply:

```
HTTP/1.1 200 OK
Date: Sun, 04 Nov 2012
Server: Apache/2.2.16 (Debian)
Content-Type: text/html
```

How do we find the server?

- Every device connected to the internet speaks the *Internet Protocol (IP)*
- Each device is given an *IP Address*
  - Currently version 4 is the most common variety (referred to as *IPv4*)
  - There is also a next generation IPv6

How do we find the server?

- An IPv4 address is a 32 bit integer, and is typically displayed as 4 numbers separated by dots:

```
69.164.221.145
128.100.31.101
142.150.210.7
```

How do we find the server?

- Connected devices can *only* refer to each by their respective addresses
- Why do we usually never see addresses then?

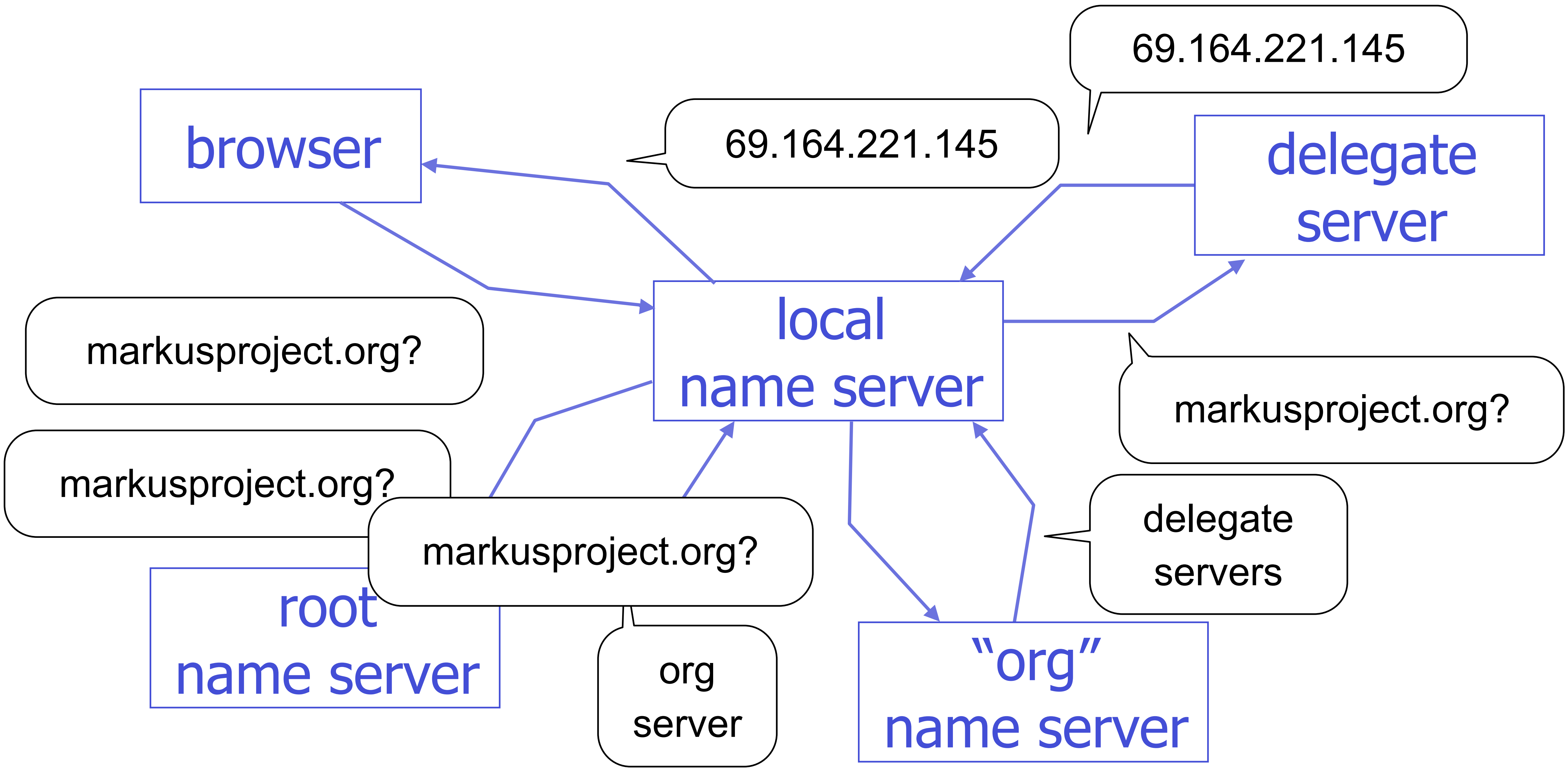
markusproject.org → `69.164.221.145`

www.cdf.toronto.edu → `128.100.31.101`

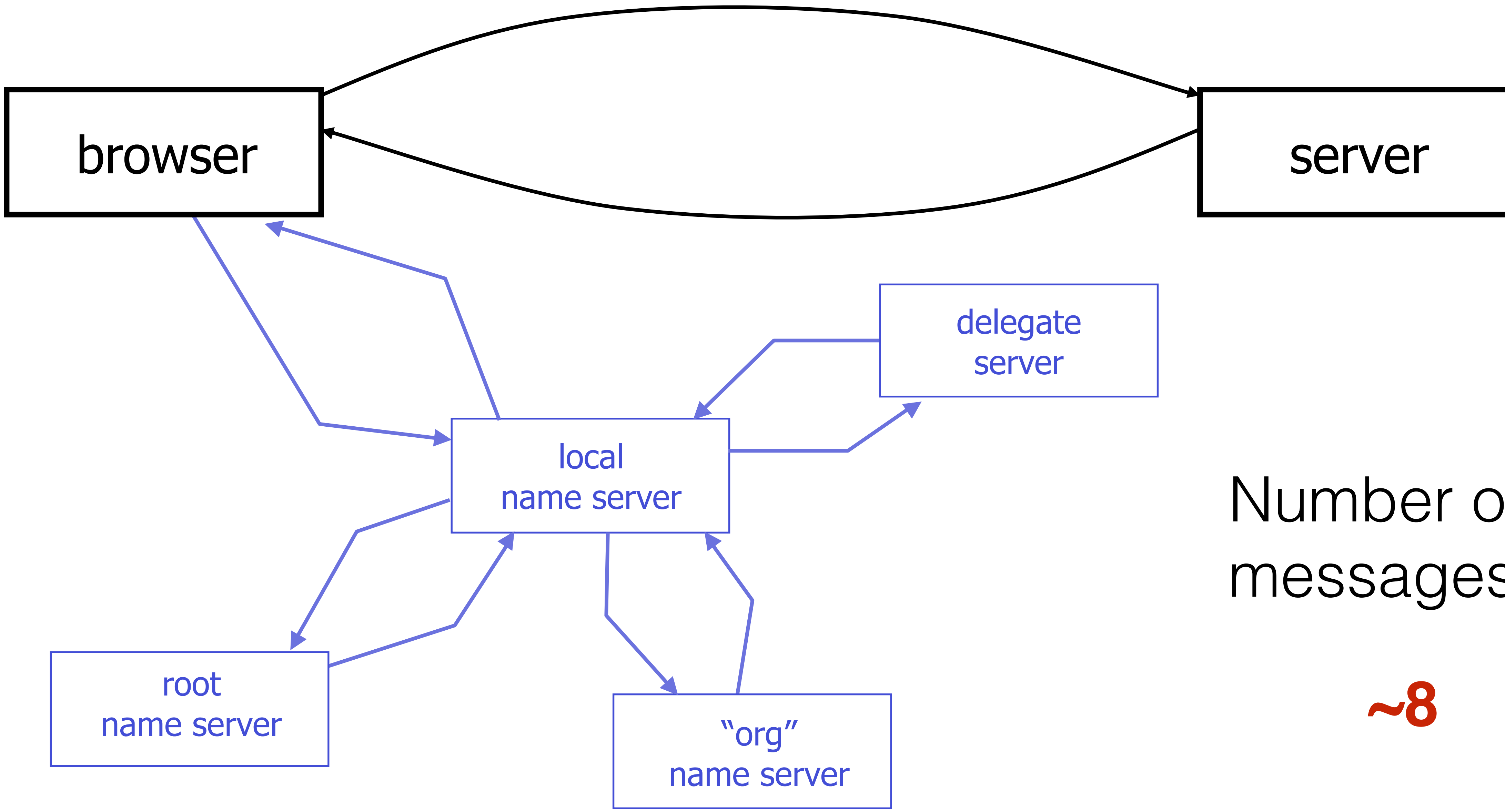
utoronto.ca → `142.150.210.7`

Domain Name System (DNS):
resolves *names* to *addresses*

Domain Name Servers



This is getting complicated!



Number of messages?

~8

Now what?

- Okay, we have the *address*.
  - What do we do with it?
- Let’s look at how two computers communicate:
- HTTP is a high-level protocol
- HTTP is specific to the web
- Computers communicate for many reasons (DNS, SSH, email, etc.)

Protocols

- Computers use several layers of general protocols to communicate.
- To understand why these layers are important, think about how a company sends you an invoice for a purchase

Protocols

Invoice:
Customer: Karen Reid
Order No: 5379

Qty:		Unit Price	Total
1	Athalon	219.00	219.00
2	128 MB	149.95	299.90
		Subtotal	518.90
		Tax	77.84
		TOTAL	596.74

Karen Reid Feb 18, 2001

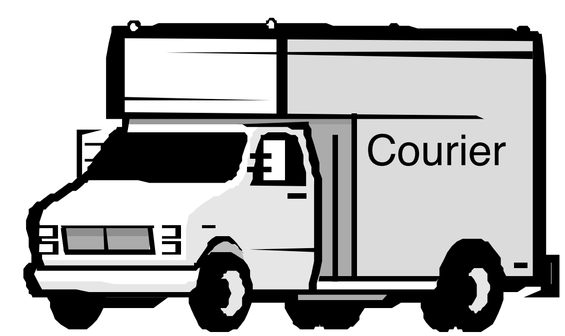
Payable to: CPUS are us $596.74
Five hundred ninety six 74/100

CPUS are us

Karen Reid
Dept. of Computer Science
University of Toronto

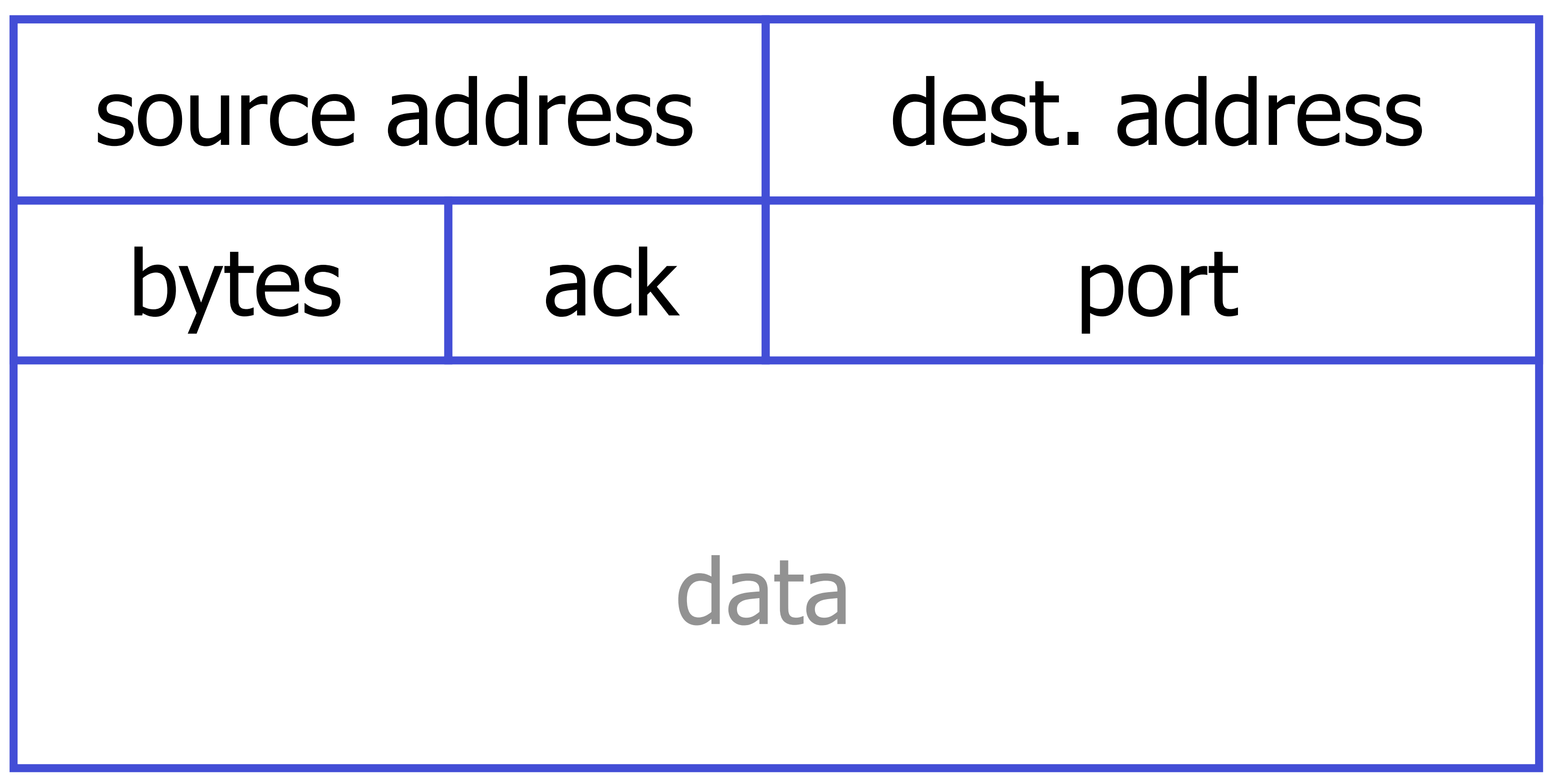
Karen Reid

CPUS are us
0 College Street
Toronto Ontario M5S 3G4

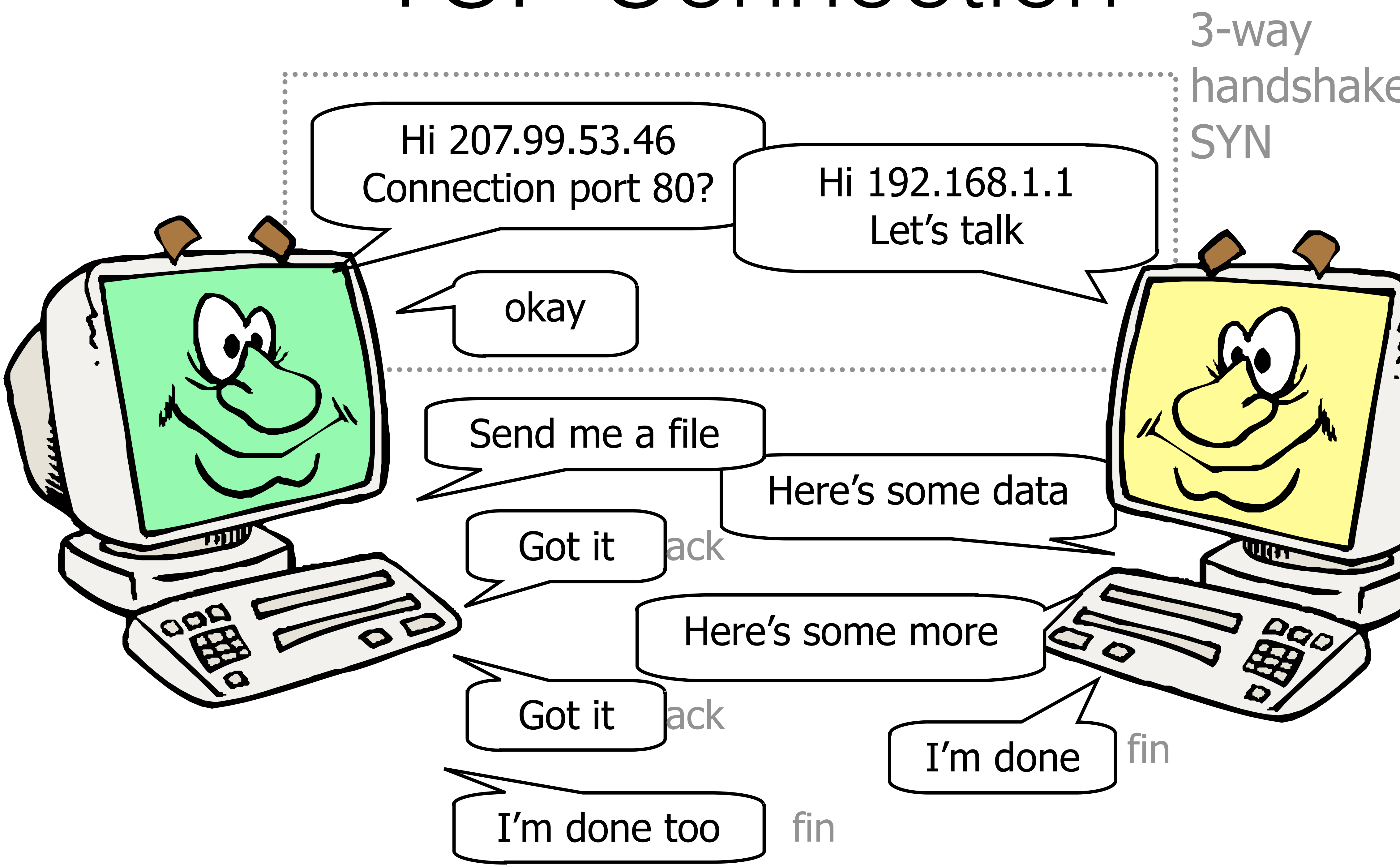


TCP/IP

- Transmission Control Protocol
- Tells us how to package up the data:



TCP Connection



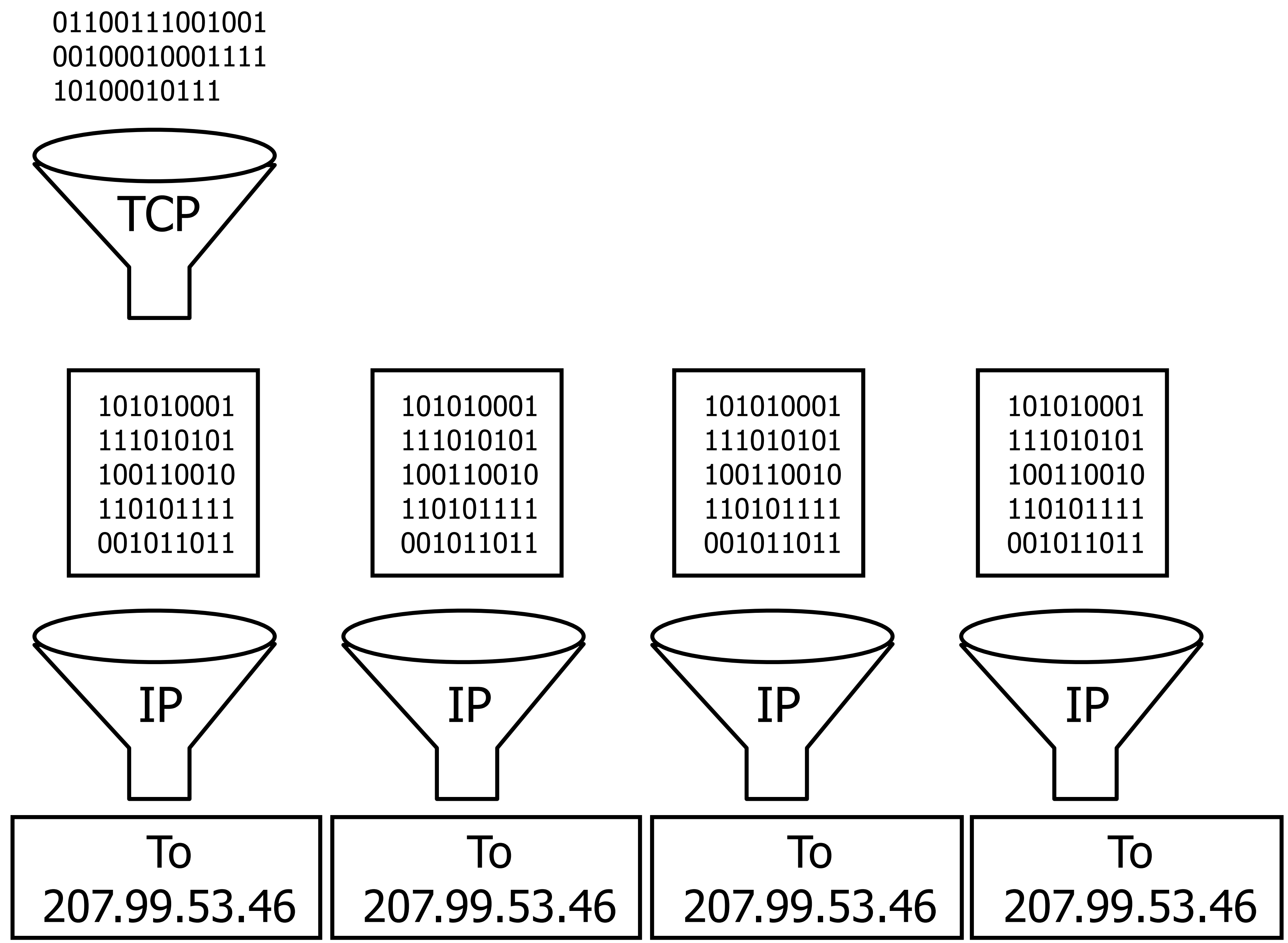
Packaging up the data

Application data

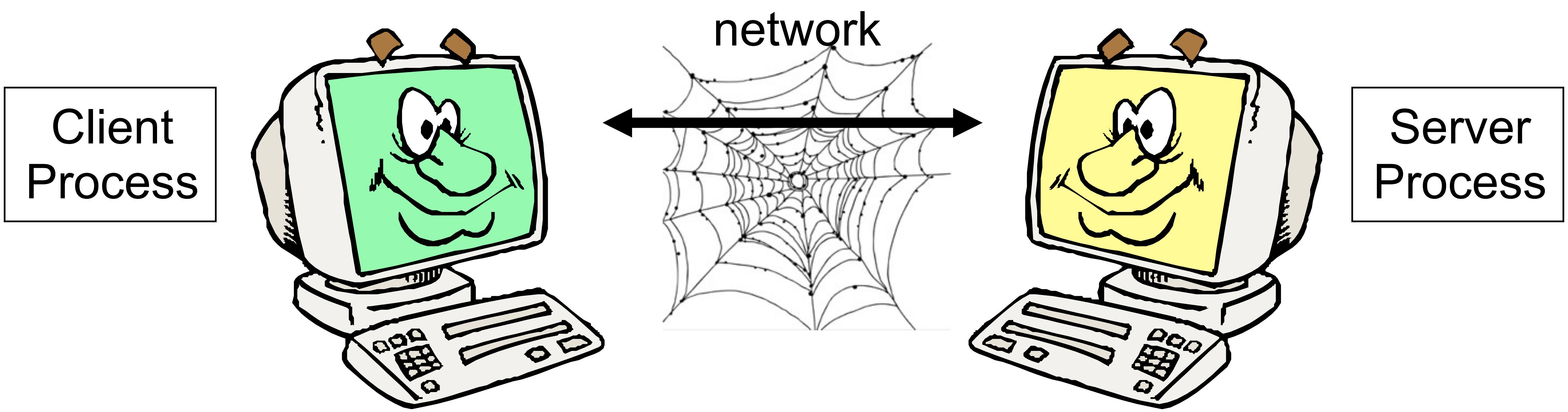
Each TCP packet is given a header:

- sequence number
- checksum

Wrapped in another IP envelope with its own header

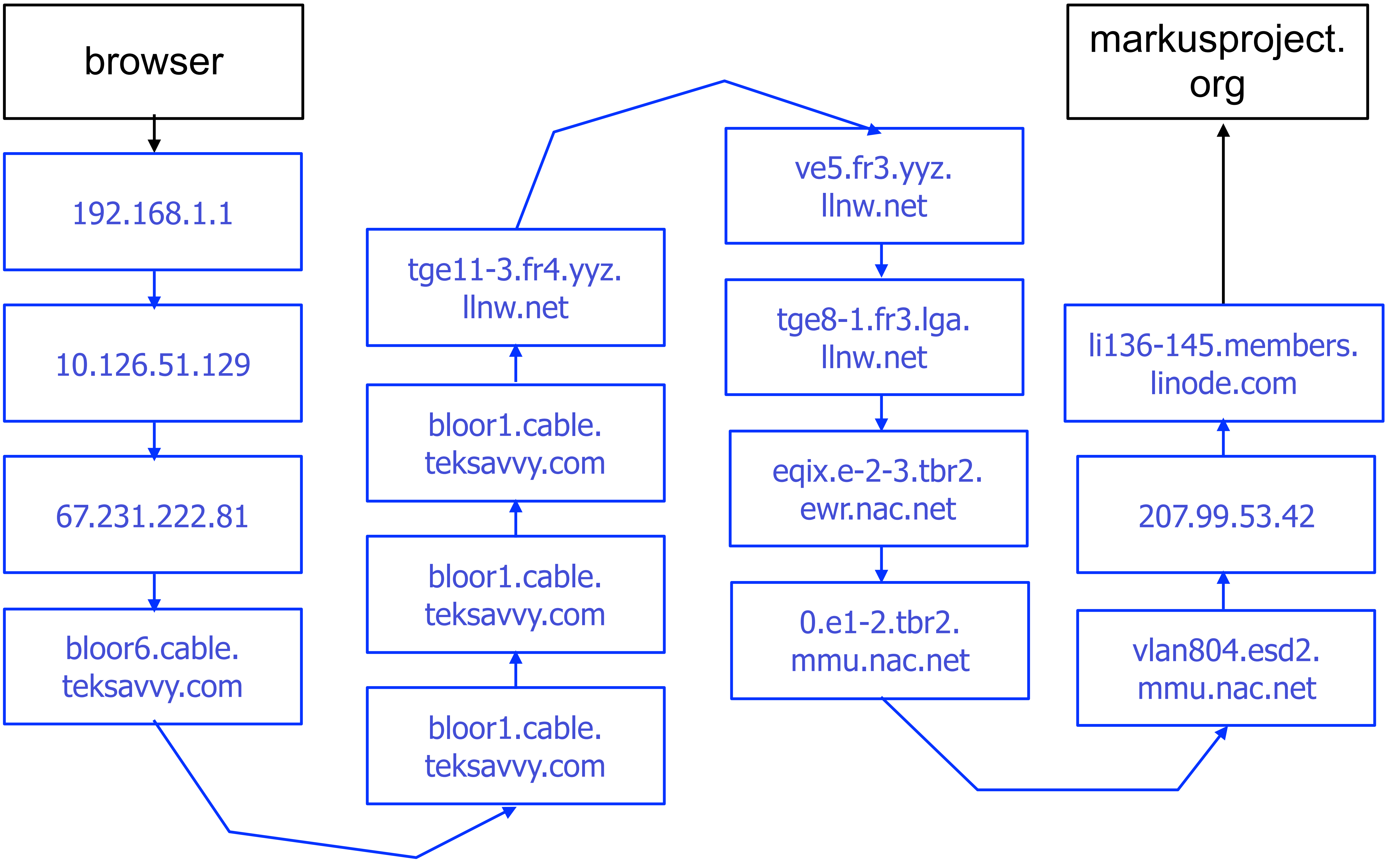


The Big Picture

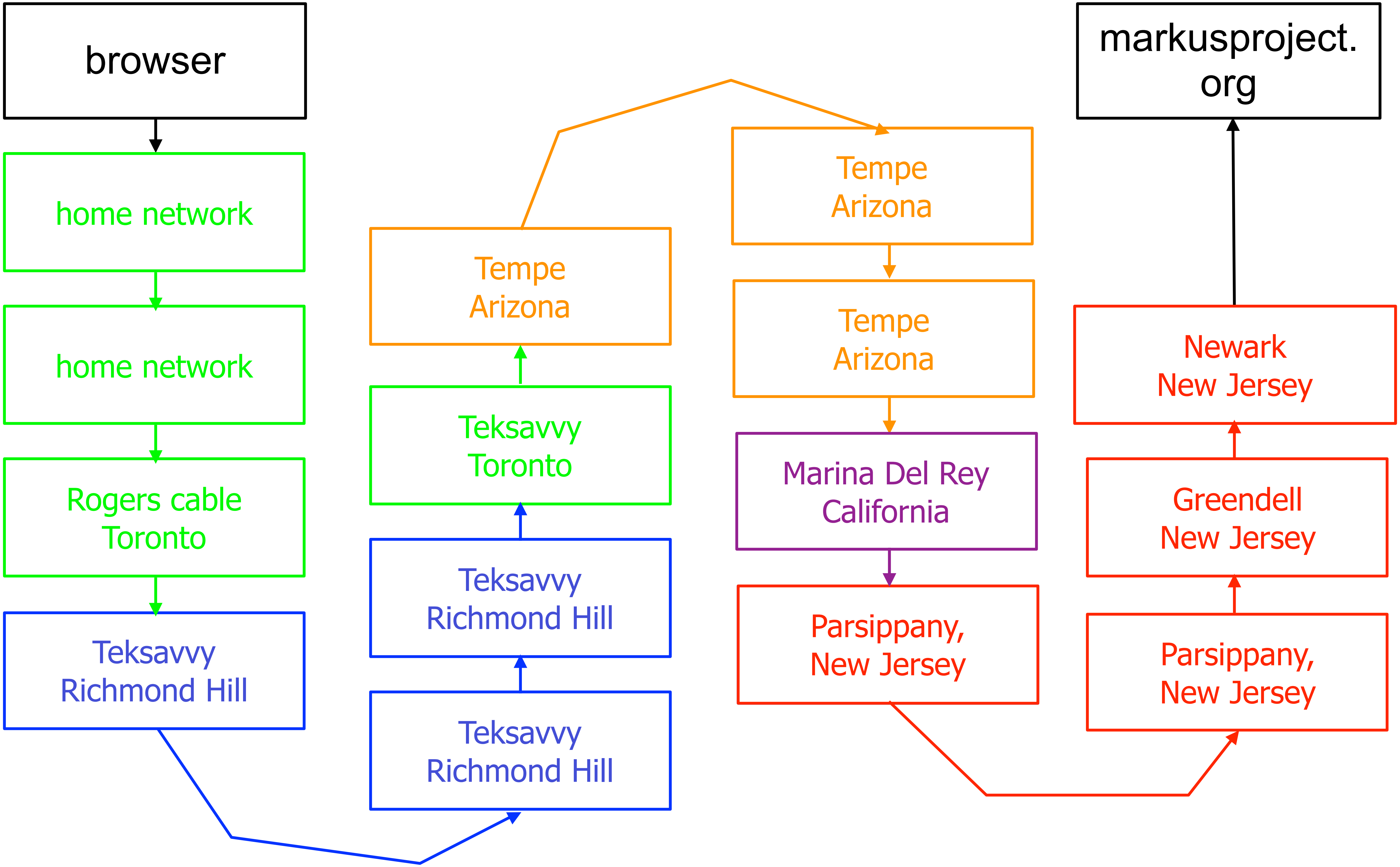


- **Client-Server model**: a *client process* wants to talk to a *server process*
- Client must first find server → DNS lookup
- Client must find process on server → ports
- Finally establish a connection so two processes can talk

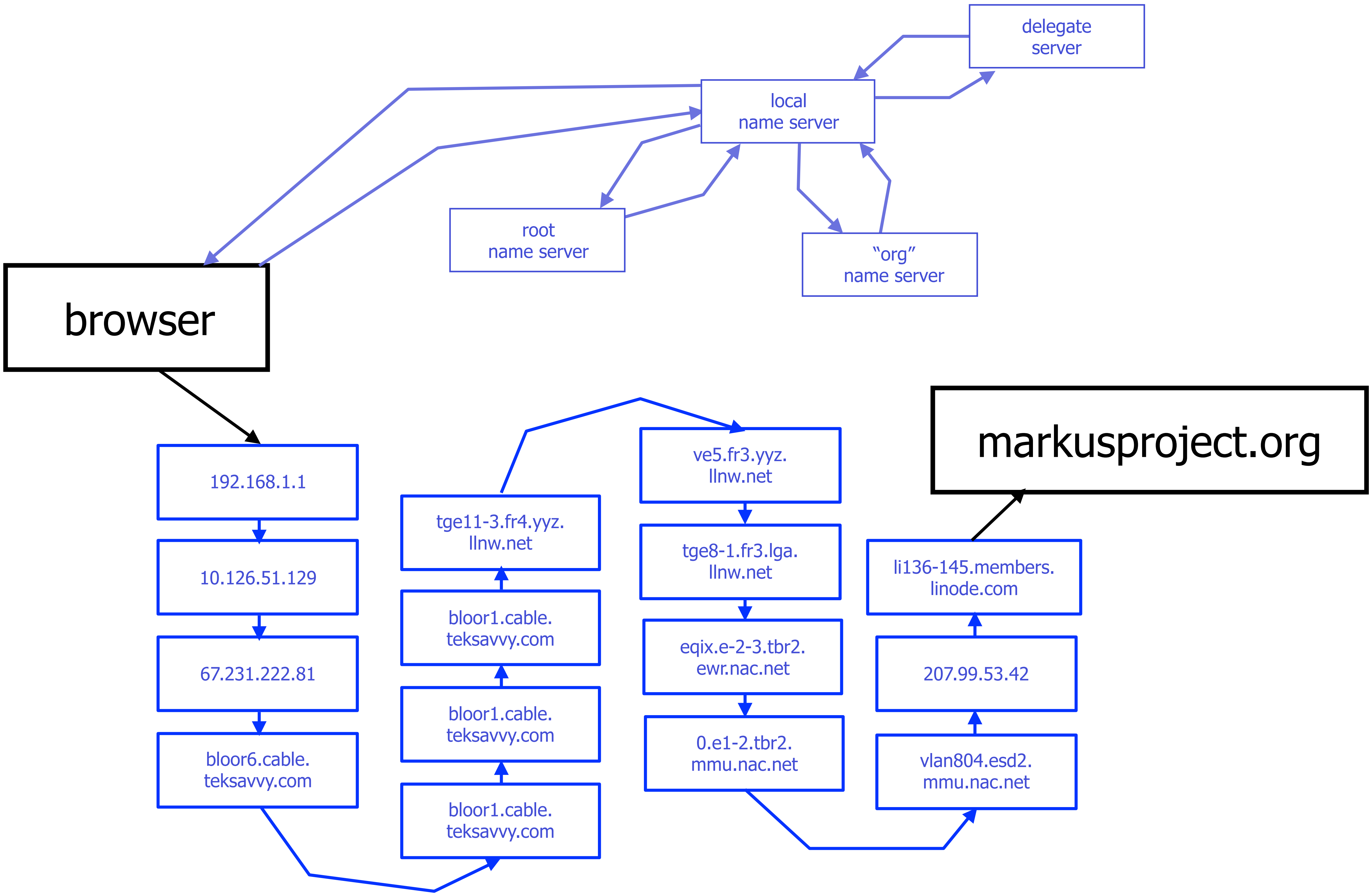
Routing (15 hops)



7 cities, 5 states/prov, 2 countries



Putting it together



How many messages?

- It depends on the size of the web page
- The web page that appears for markusproject.org is less than 30 Kbytes
- If the web page is 30 Kbytes (small!) it will likely be broken up into ~20 IP packets.

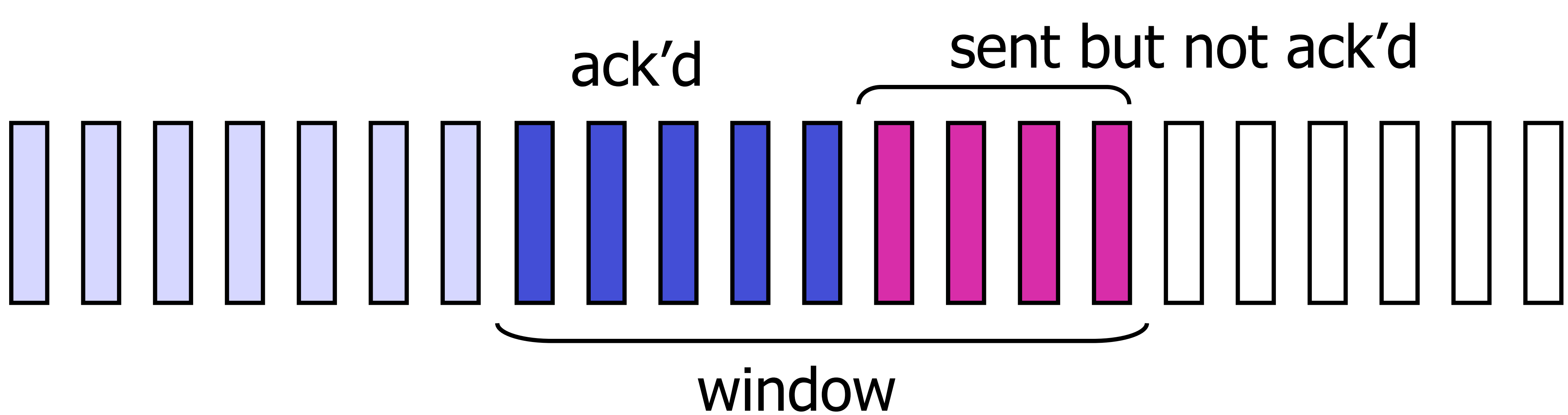
8 (DNS) + 20 * 15 hops
= 308 messages

When something goes wrong

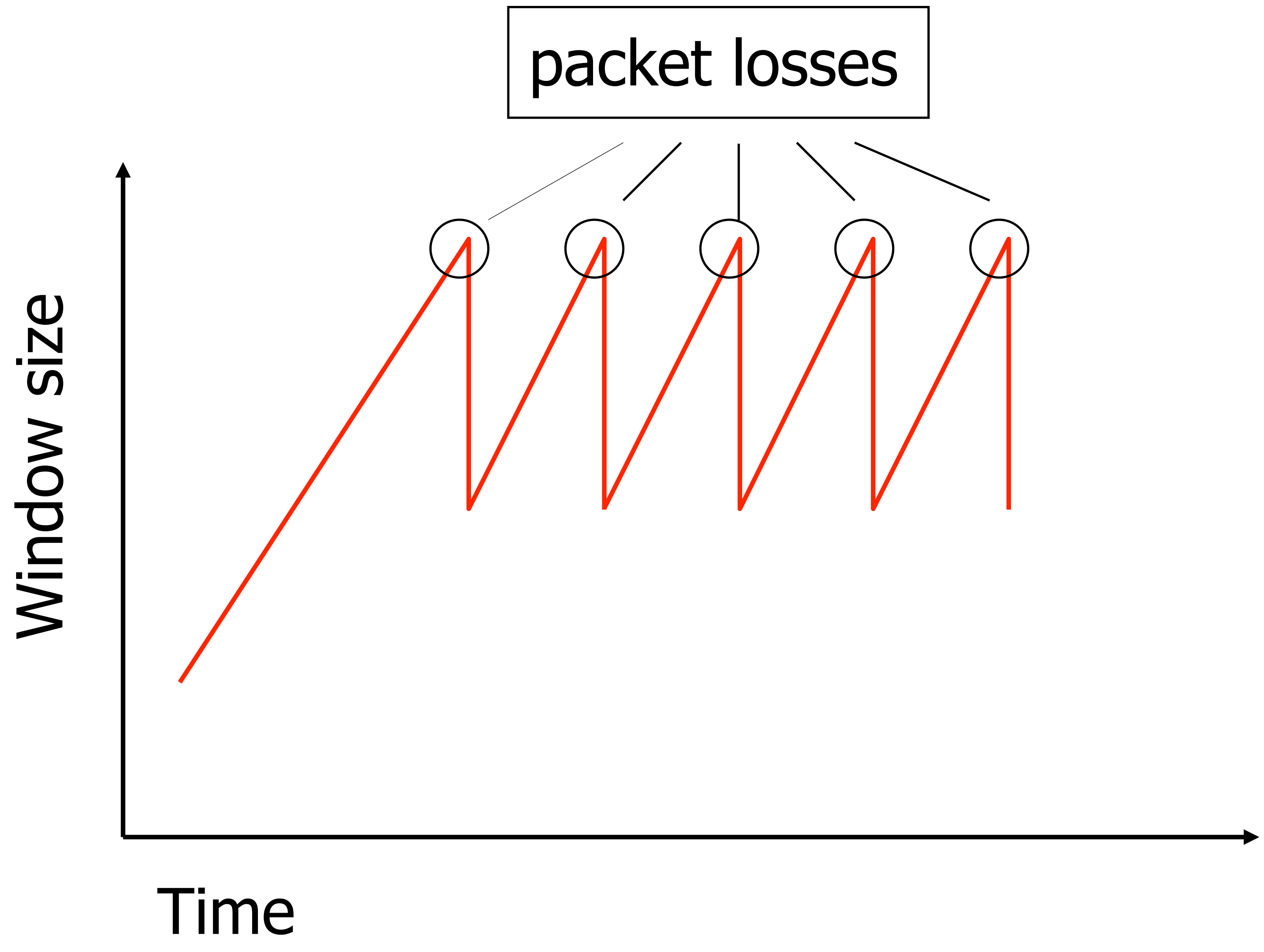
- A packet might not arrive:
  - Traffic overload
  - Bit corruption
- Receiver asks for missing packets to be resent
- Want to send data as fast as possible
- But sending too fast wastes resources

TCP Congestion Control

- Window-based:
  - Some number of packets allowed to be sent and not **`ACK`**'d
  - As successful **`ACK`**'s arrive, grow window
  - If packet loss is detected, cut window



TCP Congestion Control

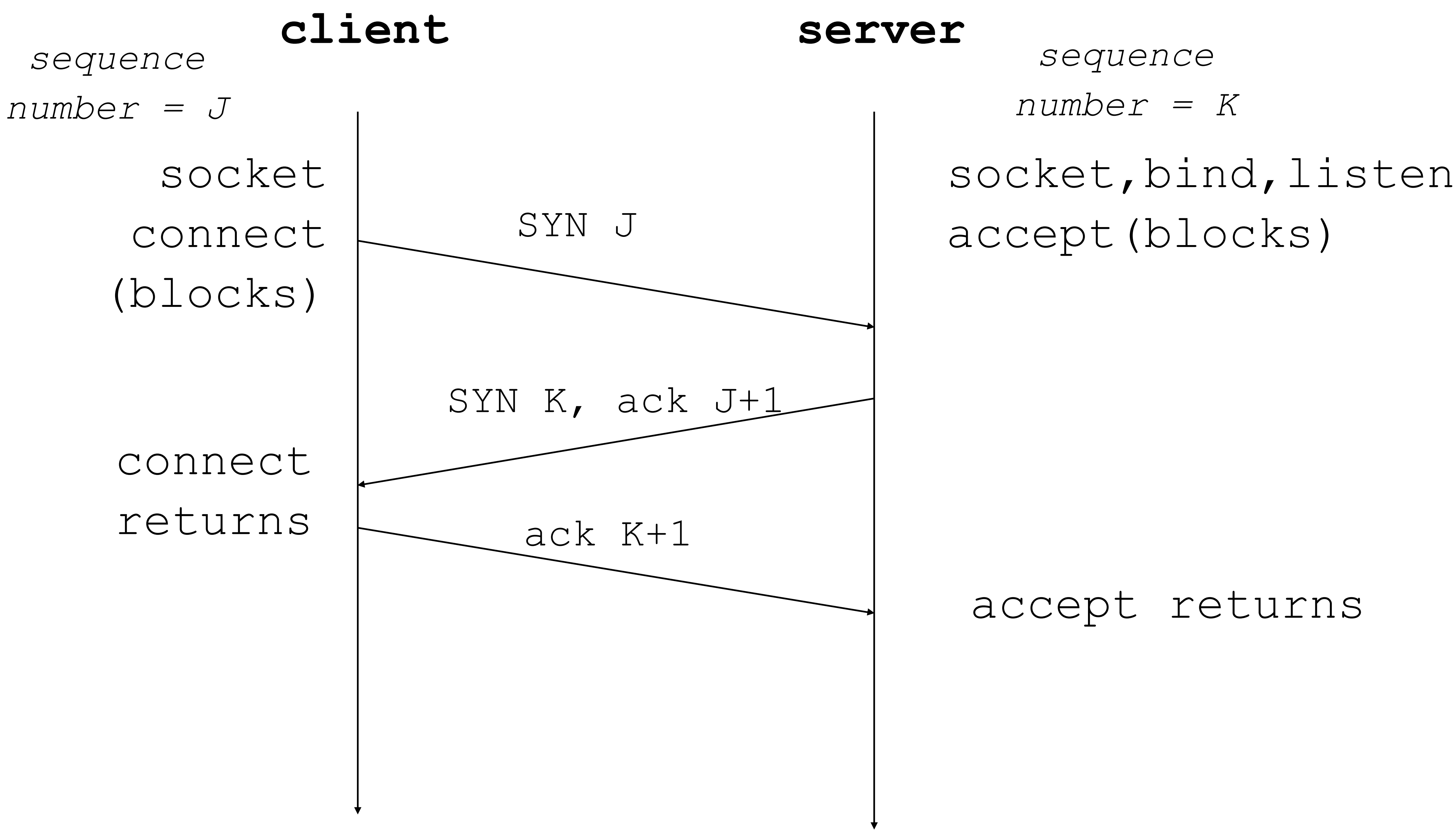


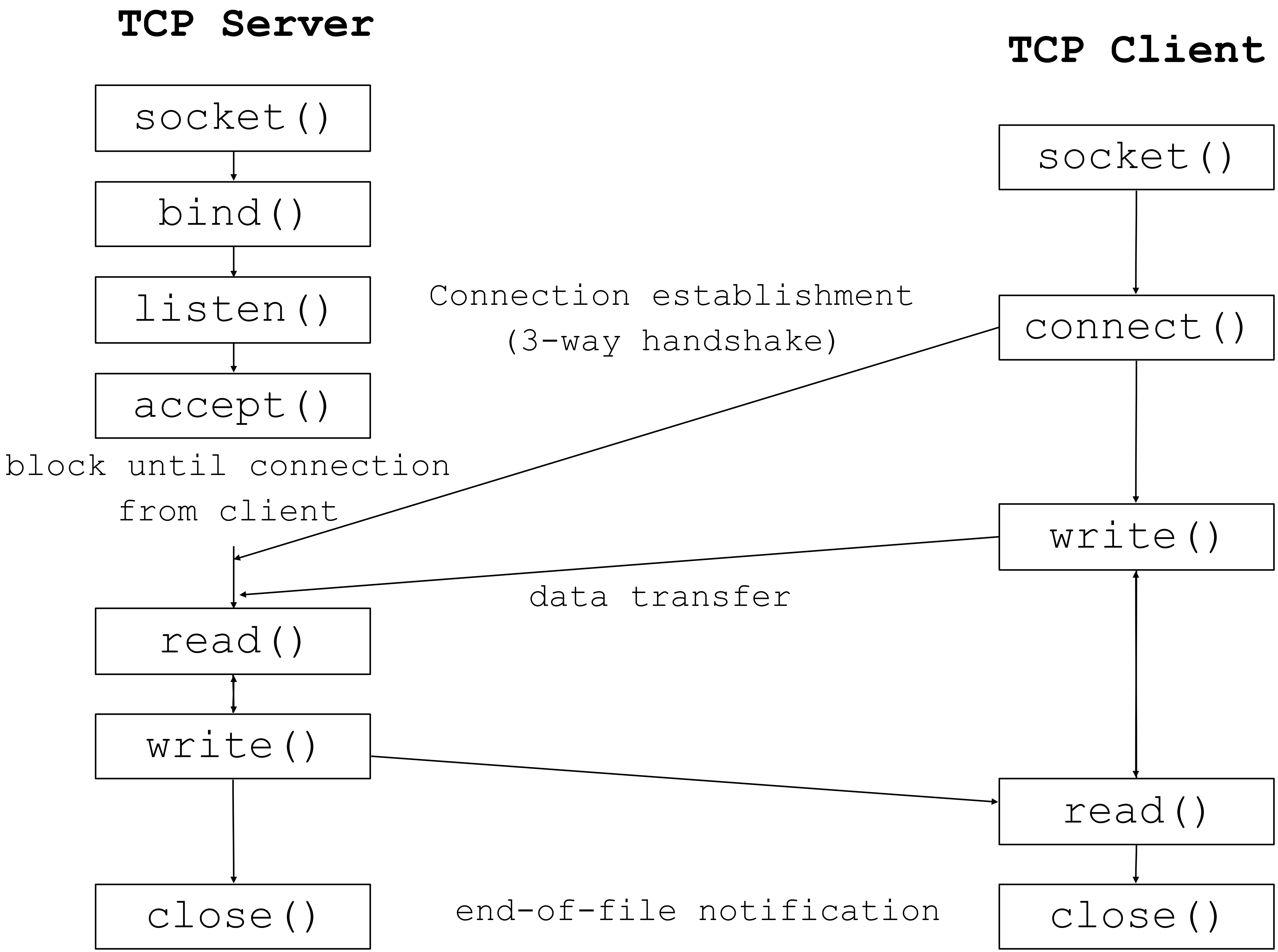
All we did was click on a link...

Take aways

- The web today is made up of complex layers of software
- No one person, organization, or company could have created it in isolation
- We can understand it because we can study one layer at a time
- We can create new things by building on top of existing layers

TCP: Three-way handshake




TCP Server
TCP Client
socket()
bind()
listen()
accept()
block until connection
from client
Connection establishment
(3-way handshake)
socket()
connect()
write()
data transfer
read()
write()
read()
close()
end-of-file notification
close()

Programming with Sockets

Kerrisk 56, 58.1-6, 59.1-7

Connection-Oriented

Server:

- Create a socket: `socket()`
- Assign a name to a socket: `bind()`
- Establish a queue for connections: `listen()`
- Get a connection from the queue: `accept()`

Client:

- Create a socket: `socket()`
- Initiate a connection: `connect()`

Socket Types

- Two main categories of sockets
  - *UNIX domain*: both processes on the same machine
  - *INET domain*: processes on different machines
- Three main types of sockets:
  - `SOCK_STREAM`: the one we will use (TCP)
  - `SOCK_DGRAM`: for connection-less sockets (UDP)
  - `SOCK_RAW`

Addresses and Ports

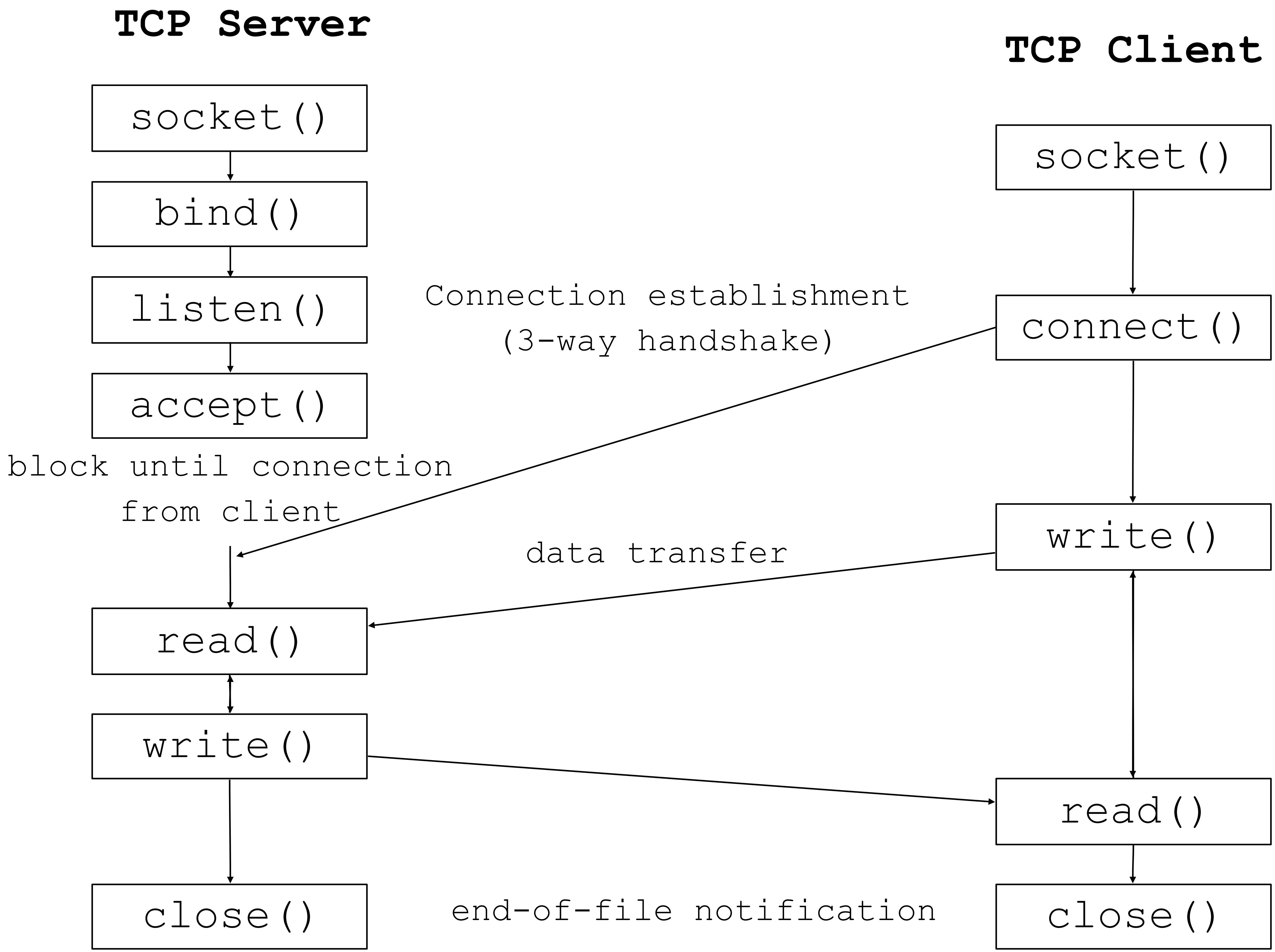
- A ***socket pair*** is the two endpoints of the connection.
- An endpoint is identified by an ***IP address and a port***.
- IPv4 addresses are 4 8-bit numbers:
  - 128.100.31.200 ← cdf.toronto.edu
- Ports
  - A way to distinguish between different processes communicating for different

`nslookup` *`dns-name`*

iana.org — Internet Assigned Numbers Authority

- *Well-known* ports: 0-1023
  - 80 = http
  - 443 = https
  - 22 = ssh
  - 21 = ftp
  - 25 = smtp (mail)
  - 194 = irc
- *Registered* ports: 1024-49151
  - 2195-2196 = Apple Push Notification
  - 3074: Xbox LIVE
  - 8000, 8080: Common alternative HTTP server
  - 23399 = Skype
- *Dynamic* (private) ports: 49152-65535

See https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers


TCP Server
TCP Client
socket()
bind()
listen()
accept()
block until connection
from client
read()
write()
close()
socket()
connect()
write()
read()
close()
Connection establishment
(3-way handshake)
data transfer
end-of-file notification

netcat (`nc`): a command line utility for acting as either a socket client or a socket server

Connecting to servers (as a *client*) using *netcat*

put **`nc`** into *verbose* mode

port to connect to

```
wolf:~$ nc -v www.cdf.toronto.edu 80
```

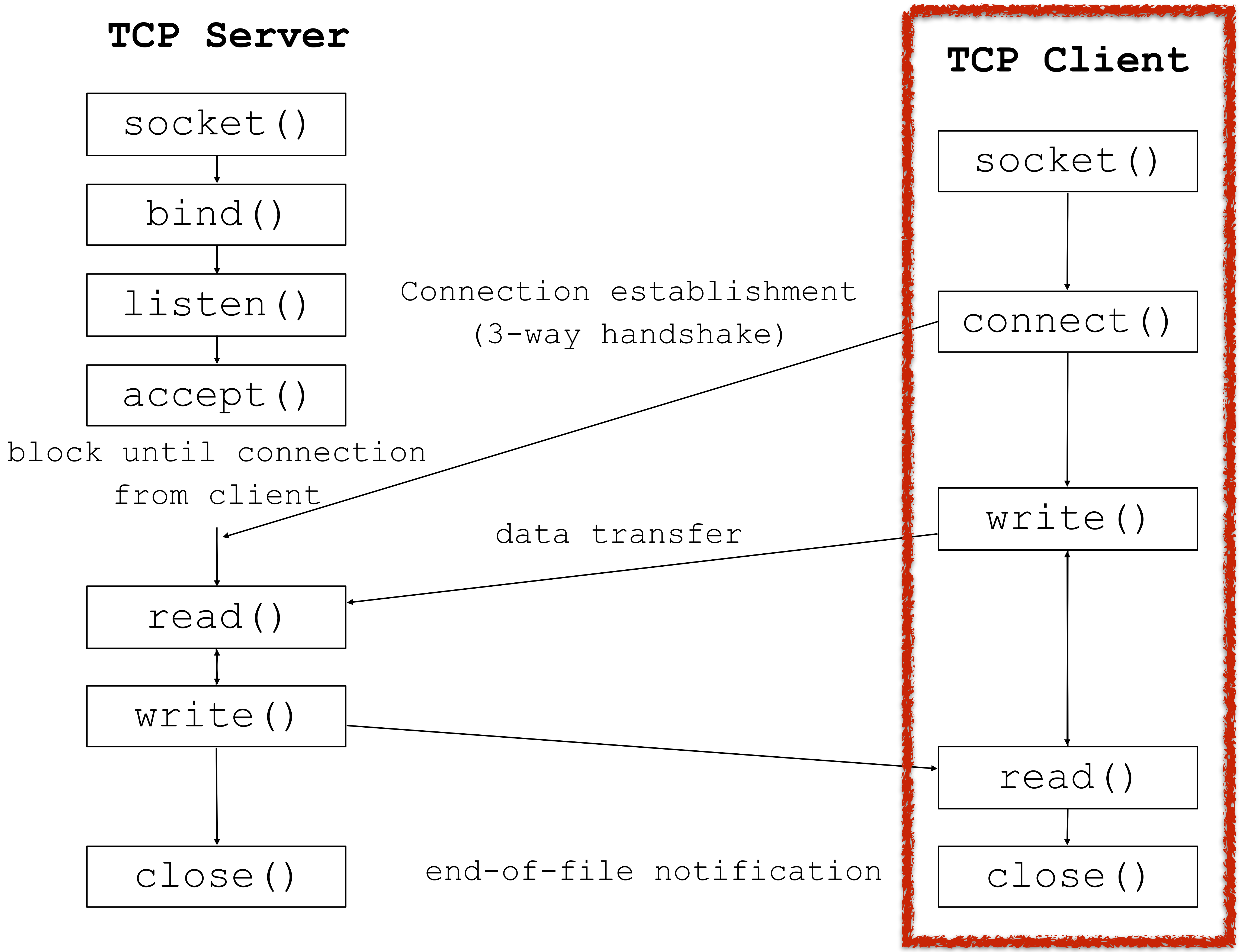
hostname to connect to

Connecting to servers (as a *client*) using *netcat*

```
wolf:~$ nc -v www.cdf.toronto.edu 80
Connection to www.cdf.toronto.edu 80 port [tcp/http] succeeded!
GET / HTTP/1.1
Host: www.cdf.toronto.edu

HTTP/1.1 200 OK
…
```

How does `nc` do this?


TCP Server
socket()
bind()
listen()
accept()
block until connection from client
read()
write()
close()
Connection establishment (3-way handshake)
data transfer
end-of-file notification
TCP Client
socket()
connect()
write()
read()
close()

`socket` — create an endpoint for communication

```
int socket(int family,
           int type,
           int protocol);
```

- `family` specifies *protocol family*:
    - `AF_INET` – IPv4
    - `AF_INET6` – IPv6
    - `AF_LOCAL` – Unix domain
    - … many others!

`socket` — create an endpoint for communication

```
int socket(int family,
           int type,
           int protocol);
```

- `type` (depends on `family`):
  - `SOCK_STREAM` — connection-oriented stream (TCP)
  - `SOCK_DGRAM` — connection-less datagrams (UDP)
  - `SOCK_RAW` — raw IP level protocol (not *underneath* TCP or UDP)

`socket` — create an endpoint for communication

```c
int socket(int family,
           int type,
           int protocol);
```

- **`protocol`**: set to 0 except for RAW sockets
- Returns -1 on error, otherwise a *socket file descriptor*

```
// Construct a TCP/IPv4 socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Construct a UDP/IPv4 socket
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

`connect` - initiate a connection on a socket

```
int connect(int sockfd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

- Connect **`sockfd`** to the server address specified in **`addr`**
    - Initiates the three-way handshake
    - The kernel will choose a dynamic port and source IP address
- Returns 0 on success, -1 on failure (setting **`errno`**)

`connect` - initiate a connection on a socket

```
int connect(int sockfd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

- Connect **`sockfd`** to the server address specified in **`addr`**
  - Initiates the three-way handshake
  - The kernel will choose a dynamic port and source IP address
- Returns 0 on success, -1 on failure (setting **`errno`**)

Specifying socket addresses

- `struct sockaddr` is a non-specific, *generic* definition
- Specific socket families (i.e. `AF_INET`) define an overlapping struct with fields specific for them (think `union`)
    - `struct sockaddr_in`

Specifying socket addresses

```
struct sockaddr_in {
    sa_family_t    sin_family; /* ==AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};

struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

See manpage `ip(7)` for more details

```c
struct sockaddr_in addr;

addr.sin_family      = AF_INET;
addr.sin_port        = ???; // network byte order?
addr.sin_addr.s_addr = ???; // network byte order?

int rc = connect(sockfd,
                 (struct sockaddr *) &addr,
                 sizeof (addr));

if (rc < 0) …
```

Byte Order

Little Endian:

209209 = `0x00 03 31 39`

A+0	A+1	A+2	A+3
39	31	03	00

(least significant bytes in lowest addresses)

Big Endian:

209209 = `0x00 03 31 39`

A+0	A+1	A+2	A+3
00	03	31	39

(most significant bytes in lowest addresses)

Specifying socket addresses

- *Network byte order* is big endian:
  - How do we specify integers in *big* endian when our x86 machines are *little* endian?

`htonl`, **`htons`**, **`ntohl`**, **`ntohs`** - convert values between host and network byte order

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- **`uint16_t`** and **`uint32_t`** types come from **`stdint.h`**
- *Host to network* short (**`htons`**)/long (**`htonl`**)
  - i.e. little endian to big endian
- *Network to host* short (**`ntohs`**)/long (**`htons`**)
  - i.e. big endian to little endian

See **`byteorder(3)`** for more

byteorders.c

```c
struct sockaddr_in addr;

short port = 80;
addr.sin_family       = AF_INET;
addr.sin_port         = htons(port);
addr.sin_addr.s_addr = ???; // network byte order?

int rc = connect(sockfd,
                 (struct sockaddr *) &addr,
                 sizeof (addr));

if (rc < 0) …
```

What about `sin_addr.s_addr` ?

```
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};

struct in_addr {
    /* address in network byte order */
    uint32_t       s_addr;
};
```

gethostbyname

```
struct hostent *gethostbyname(const char *name);
```

```
struct hostent {
    char  *h_name;       // official name of host
    char **h_aliases;    // alias list
    int    h_addrtype;   // host address type
    int    h_length;     // length of address
    char **h_addr_list;  // list of addresses
};
```

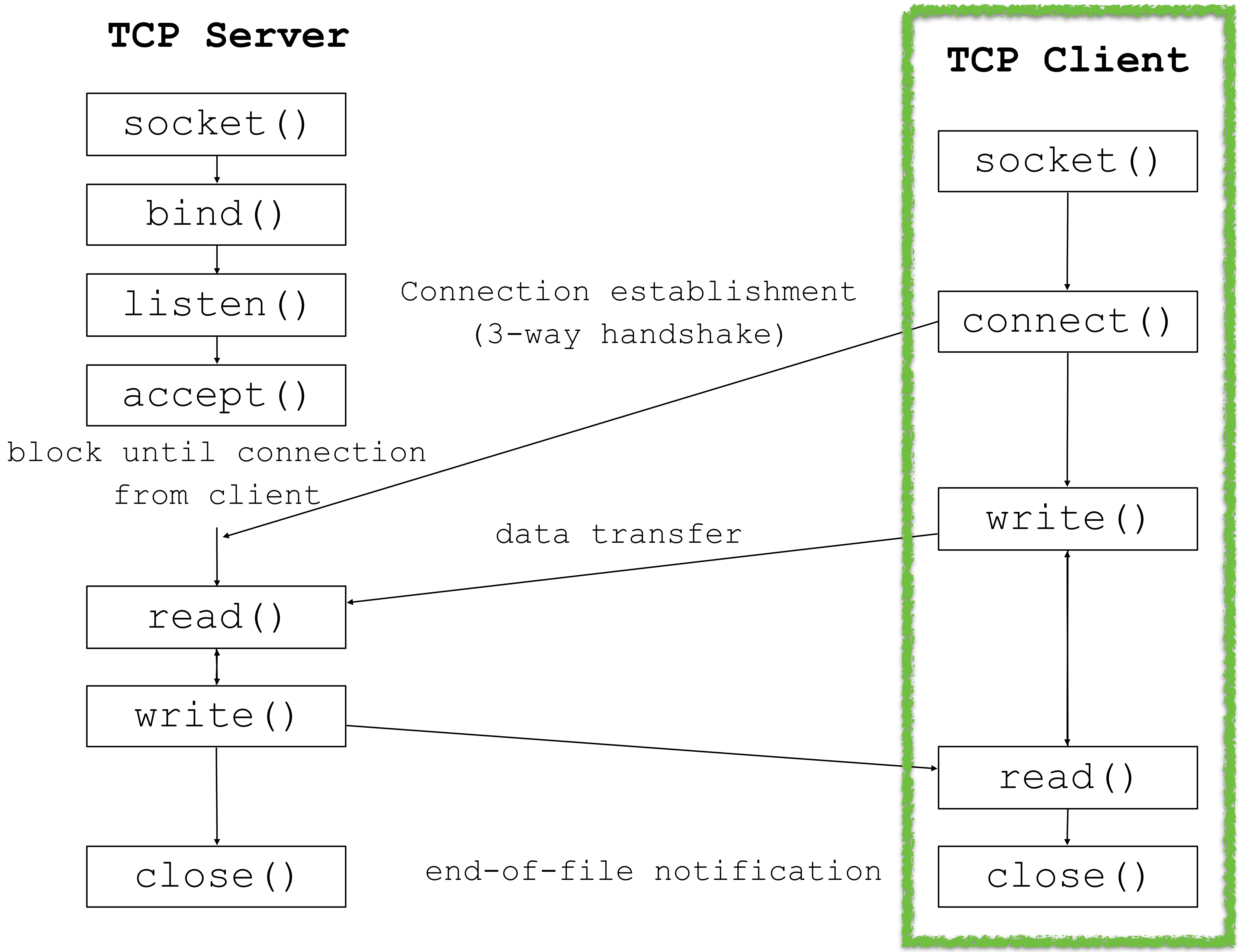
- Many fields herein…
- You can cast `h_addr_list[0]` to `struct in_addr *` for `AF_INET` addresses!

```
struct hostent *hp = gethostbyname("cdf.toronto.edu");

struct in_addr *in4 =
      (struct in_addr *) hp->addr_list[0];

addr.sin_addr = *in4;
```

gethost.c



TCP Server
socket()
bind()
listen()
accept()
block until connection
from client
read()
write()
close()
Connection establishment
(3-way handshake)
data transfer
end-of-file notification
TCP Client
socket()
connect()
write()
read()
close()

simpleget.c

Using `nc` to act *as* a server

Running your own server using *netcat*

```
wolf:~$ nc -v -l -k localhost 20209
```

- Ask **`nc`** to *listen* (**`-l`**) on address localhost, port 20209
- Optional **`-k`** is for keep-alive, i.e. stay listening for more connections even after the first one ends

Run a chat server *and* client

Server (*listening*):

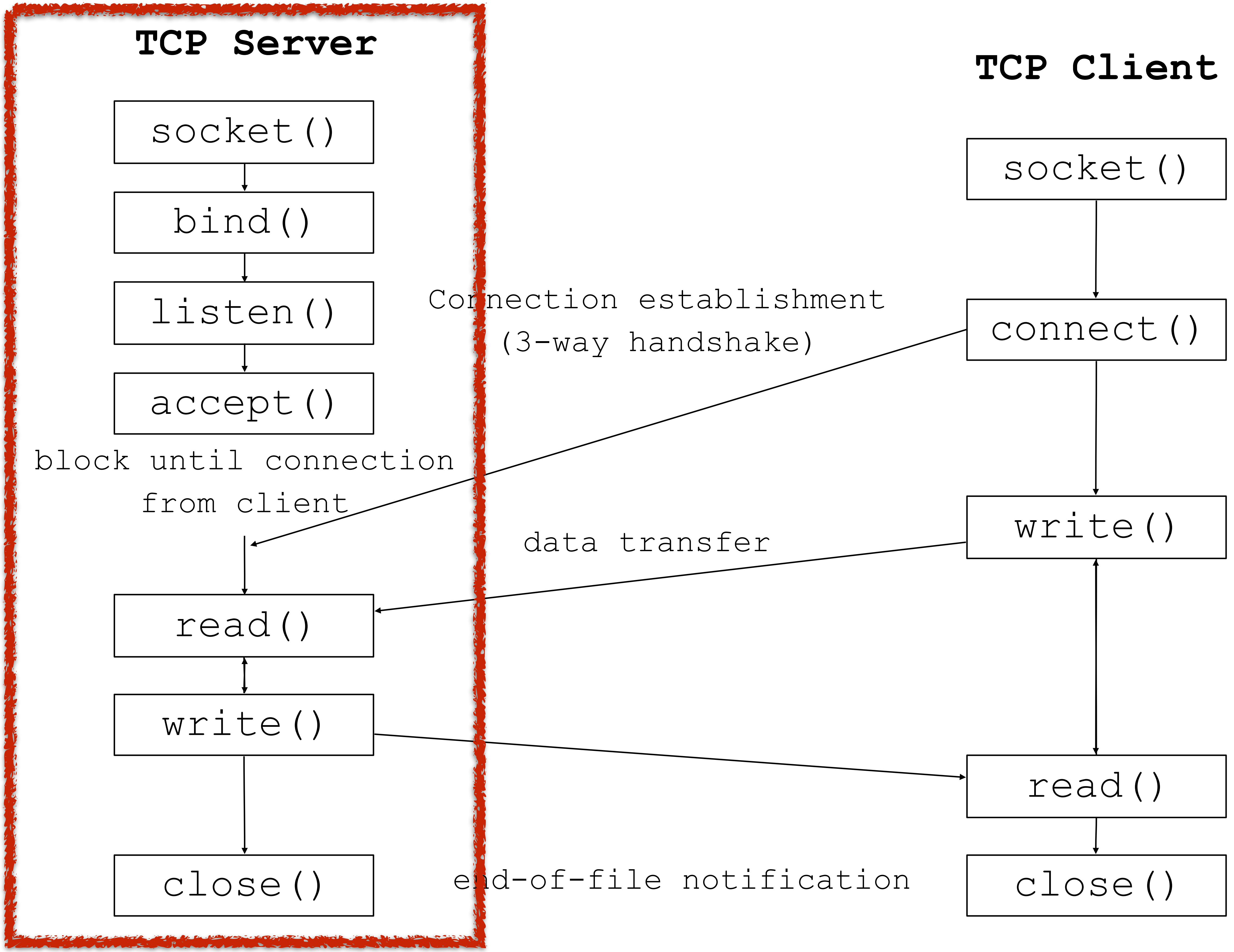
```
wolf:~$ nc -vlk localhost 20209
```

Client (*connecting*):

```
wolf:~$ nc -v localhost 20209
```

NB: Other students may be using the same port number so if necessary find one that is free!

`simpleget.c` *and* `nc`


TCP Server
socket()
bind()
listen()
accept()
block until connection from client
read()
write()
close()
TCP Client
socket()
connect()
write()
read()
close()
Connection establishment (3-way handshake)
data transfer
end-of-file notification

`bind` — bind a name to a socket

```
int bind(int sockfd,
         const struct sockaddr *addr,
         socklen_t addrlen);
```

- *Assigns* (binds) the specified address & port (*name*) to the socket
- **`sin_addr.s_addr`** can be set to `INADDR_ANY` to bind to any network interface
  - Also, `INADDR_LOOPBACK` refers to the loopback device (adrdess `127.0.0.1`)
  - But you must use **`htonl`**! (these constants are in host byte order)

```
struct sockaddr_in addr;

addr.sin_family      = AF_INET;
addr.sin_port        = htons(port);
addr.sin_addr.s_addr = htonl(IPADDR_LOOPBACK);

int rc = bind(sockfd,
              (struct sockaddr *) &addr,
              sizeof (addr));

if (rc < 0) …
```

bind.c

`listen` — listen for connections on a socket

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

- Prepare the kernel-maintained queue of connections that are waiting to be accepted (the 3-way handshake)
- Once prepared, the socket is ready to accept connections
- **`backlog`** is the maximum size of that queue (default upper bound on Linux is 128)

`accept` — accept a connection

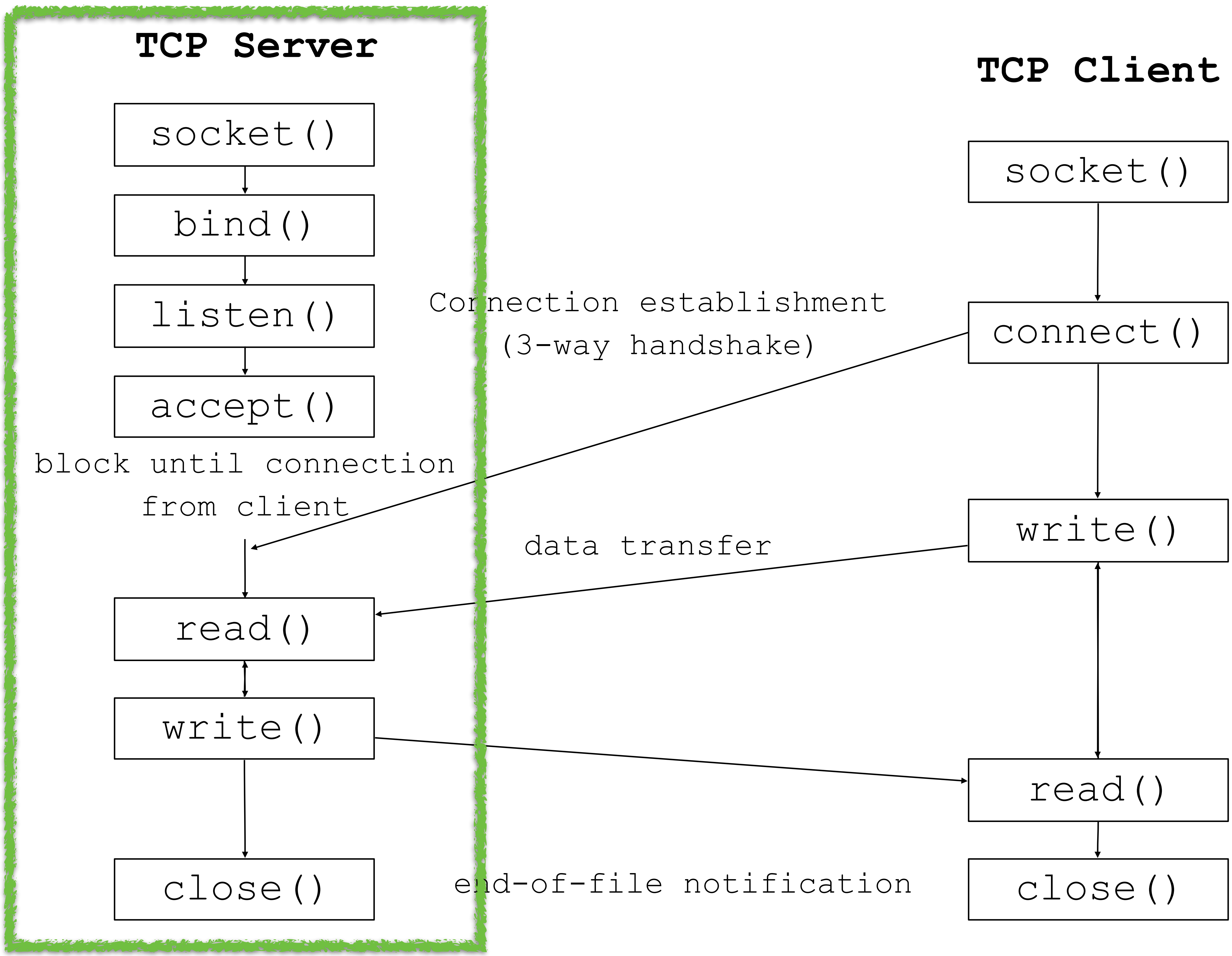
```
int accept(int sockfd,
           struct sockaddr *addr,
           socklen_t *addrlen);
```

- Blocks waiting for a new connection from the kernels' queue
- Returns a *new socket file descriptor* with refers to the newly connected TCP client
  - **`read`** & **`write`** to that descriptor to communicate with the client!

`accept` — accept a connection

```
int accept(int sockfd,
           struct sockaddr *addr,
           socklen_t *addrlen);
```

- **`addr`** will contain address information about the newly connected client
- Before the call, **`*addrlen`** must contain the length of the structure that **`addr`** points at, and aftewards, that length will be updated to be the specific length of the structure returned


TCP Server
socket()
bind()
listen()
accept()
block until connection from client
read()
write()
close()
TCP Client
socket()
connect()
write()
read()
close()
Connection establishment (3-way handshake)
data transfer
end-of-file notification

acceptclose.c

`send`, `recv` — like `write`, `read` but with more socket-y *`flags`* goodness

ssize_t recv(*int* sockfd, *void* *buf, *size_t* len,
int flags);

ssize_t send(*int* sockfd, *const void* *buf, *size_t* len,
int flags);

- `send`: when `flags==0`, then behaves exactly like `write`
  - `flags`: `MSG_OOB`, `MSG_DONTROUTE`, `MSG_DONTWAIT`
- `recv`:
  - `flags`: `MSG_OOB`, `MSG_WAITALL`, `MSG_PEEK`

More variations for *datagrams* (UDP) and raw packet family:

`sendto`, `recvfrom`

Socket System Calls

- `socket`
- `connect`
- `bind`
- `listen`
- `accept`
- `send`
- `recv`

Suggested Exercises

https://github.com/pdmccormick/csc209-summer-2015/blob/master/lectures/week10/README.md

Next Week

- Regularly scheduled office hour on Tuesday
- A3 due on Wednesday
- Lecture: *Advanced socket server programming*