Sockets / RPC

last time

redo logging

- write log + "commit", then do operation
- on failure, check log
- redo anything marked committed in log

copy-on-write filesystems / snapshots

distributed systems — motivation, etc.

connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

names and addresses

name	**address**
logical identifier	location/how to locate
hostname `www.virginia.edu`	IPv4 address `128.143.22.36`
hostname `mail.google.com`	IPv4 address `216.58.217.69`
hostname `mail.google.com`	IPv6 address `2607:f8b0:4004:80b::2005`
filename `/home/cr4bd/NOTES.txt`	inode# `120800873` and device `0x2eh/0x46d`
variable `counter`	memory address `0x7FFF9430`
service name `https`	port number `443`

hostnames

typically use *domain name system* (DNS) to find machine names

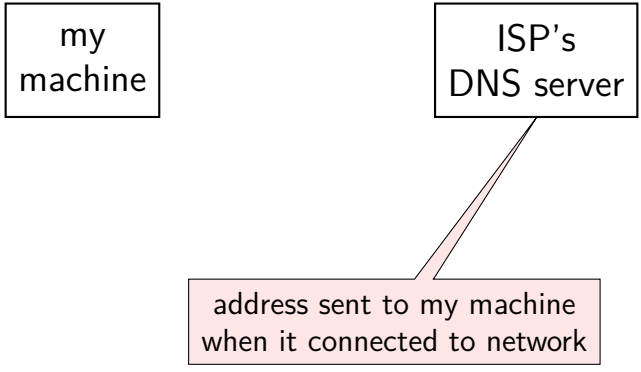
maps logical names like `www.virginia.edu`

- chosen for humans
- hierarchy of names

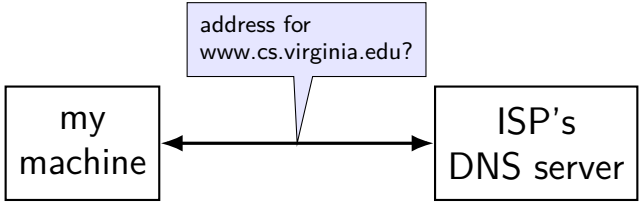
...to *addresses* the network can use to move messages

- numbers
- ranges of numbers assigned to different parts of the network
- network *routers* knows "send this range of numbers goes this way"

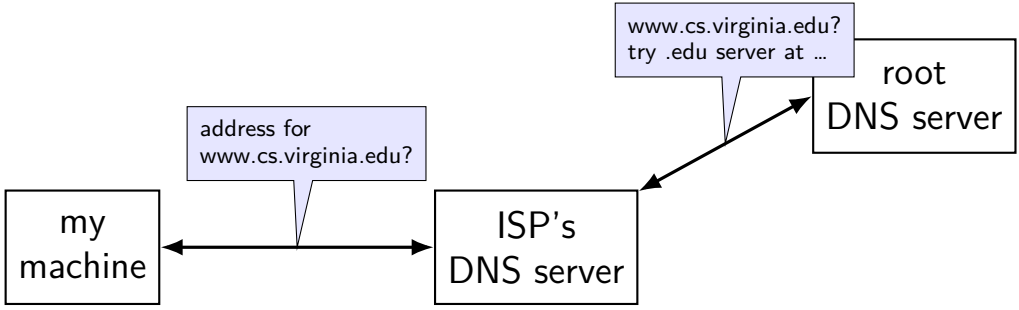
DNS: distributed database



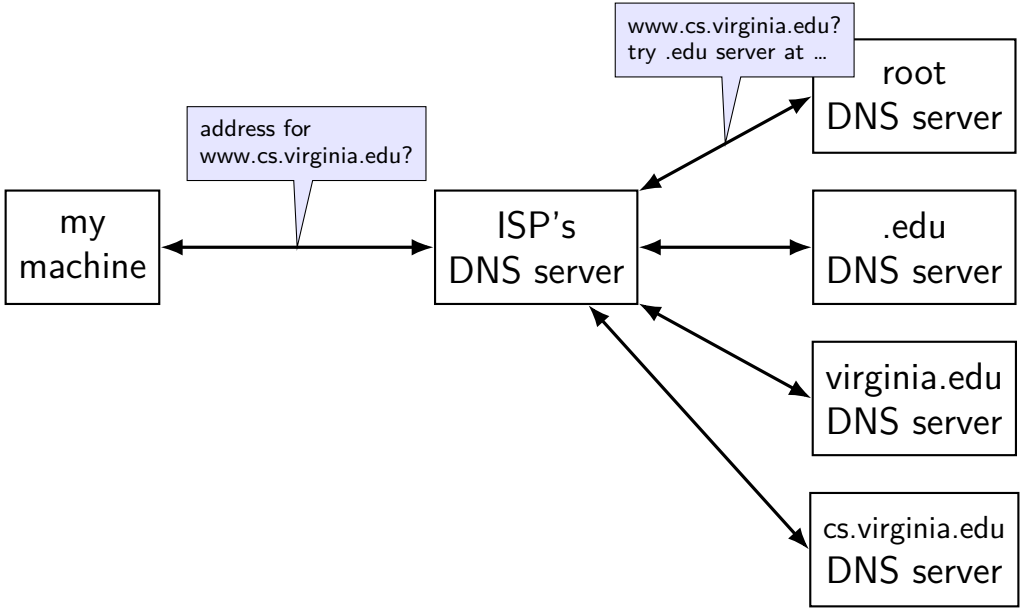
DNS: distributed database



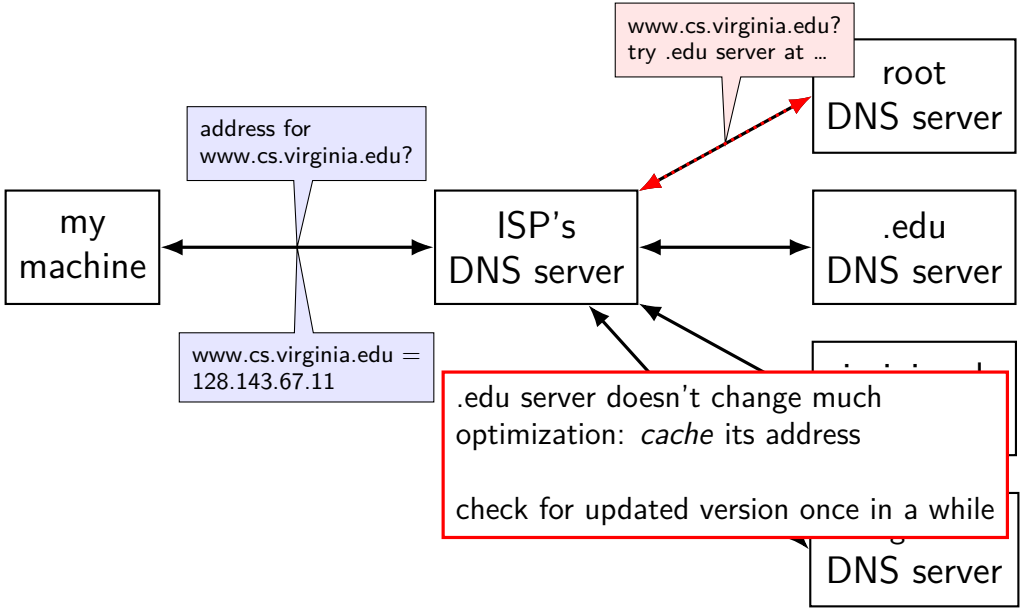
DNS: distributed database



DNS: distributed database



DNS: distributed database



IPv4 addresses

32-bit numbers

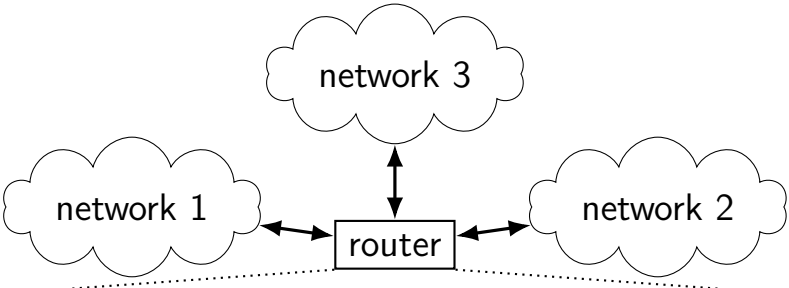
typically written like 128.143.67.11

- four 8-bit decimal values separated by dots
- first part is most significant
- same as $128 \cdot 256^3 + 143 \cdot 256^2 + 67 \cdot 256 + 11 = 2\,156\,782\,459$

organizations get blocks of IPs

- e.g. UVa has 128.143.0.0–128.143.255.255
- e.g. Google has 216.58.192.0–216.58.223.255 and 74.125.0.0–74.125.255.255 and 35.192.0.0–35.207.255.255

IPv4 addresses and routing tables



if I receive data for...	send it to...
128.143.0.0—128.143.255.255	network 1
192.107.102.0–192.107.102.255	network 1
...	...
4.0.0.0–7.255.255.255	network 2
64.8.0.0–64.15.255.255	network 2
...	...
anything else	network 3

selected special IPv4 addresses

127.0.0.0 — 127.255.255.255 — localhost

- AKA loopback
- the machine we're on
- typically only 127.0.0.1 is used

192.168.0.0–192.168.255.255 and
10.0.0.0–10.255.255.255 and
172.16.0.0–172.31.255.255

- "private" IP addresses
- not used on the Internet
- commonly connected to Internet with network address translation
- also 100.64.0.0–100.127.255.255 (but with restrictions)

169.254.0.0-169.254.255.255

- link-local addresses — 'never' forwarded by routers

network address translation

IPv4 addresses are kinda scarce

solution: *convert* many private addrs. to one public addr.

locally: use private IP addresses for machines

outside: private IP addresses become a single public one

commonly how home networks work (and some ISPs)

IPv6 addresses

IPv6 like IPv4, but with 128-bit numbers

written in hex, 16-bit parts, seperated by colons (`:`)

strings of 0s represented by double-colons (`::`)

typically given to users in blocks of 2^{80} or 2^{64} addresses

- no need for address translation?

`2607:f8b0:400d:c00::6a` =
`2607:f8b0:400d:0c00:0000:0000:0000:006a`

- `2607f8b0400d0c0000000000000006a`$_{\text{SIXTEEN}}$

selected special IPv6 addresses

`::1` = localhost

anything starting with `fe80` = link-local addresses

- never forwarded by routers

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

0–49151: typically assigned for particular services

- 80 = http, 443 = https, 22 = ssh, ...

49152–65535: allocated on demand

- default "return address" for client connecting to server

protocols

protocol = agreement on how to comunicate

sytnax (format of messages, etc.)

semantics (meaning of messages — actions to take, etc.)

human protocol: telephone

caller: pick up phone caller: check for service caller: dial caller: wait for ringing	
	callee: “Hello?”
caller: “Hi, it’s Casey...”	
	callee: “Hi, so how about ...”
caller: “Sure, ...”	
...	...
	callee: “Bye!”
caller: “Bye!”	
hang up	hang up

layered protocols

IP: protocol for sending data by IP addresses

- mailbox model
- limited message size

UDP: send datagrams built on IP

- still mailbox model, but *with port numbers*

TCP: reliable connections built on IP

- adds port numbers
- adds resending data if error occurs
- splits big amounts of data into many messages

HTTP: protocol for sending files, etc. built on TCP

other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

SSH: secure shell (remote login) — built on TCP

SCP/SFTP: secure copy/secure file transfer — built on SSH

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

...

other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

SSH: secure shell (remote login) — built on TCP

SCP/SFTP: secure copy/secure file transfer — built on SSH

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

...

sockets

socket: POSIX abstraction of network I/O queue

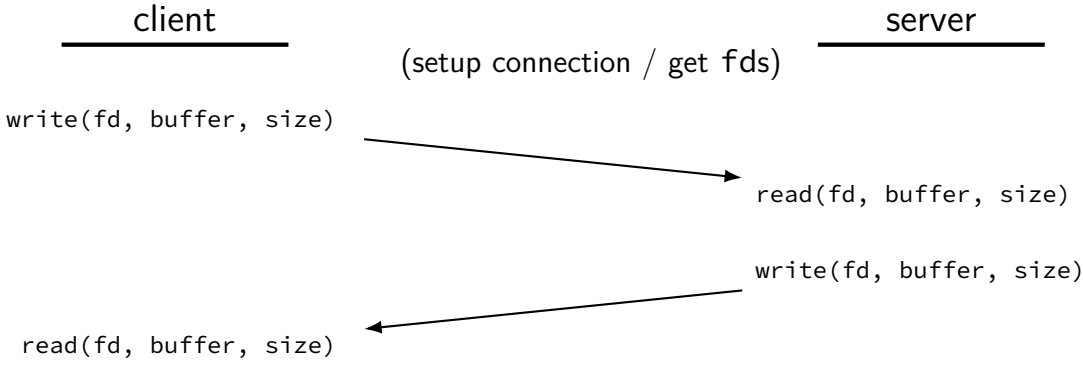
- any kind of network
- can also be used between processes on same machine

a kind of file descriptor

connected sockets

sockets can represent a connection

act like bidirectional pipe



echo client/server

```
void client_for_connection(int socket_fd) {
    int n; char send_buf[MAX_SIZE]; char recv_buf[MAX_SIZE];
    while (prompt_for_input(send_buf, MAX_SIZE)) {
        n = write(socket_fd, send_buf, strlen(send_buf));
        if (n != strlen(send_buf)) {...error?...}
        n = read(socket_fd, recv_buf, MAX_SIZE);
        if (n <= 0) return; // error or EOF
        write(STDOUT_FILENO, recv_buf, n);
    }
}
```

```
void server_for_connection(int socket_fd) {
    int read_count, write_count; char request_buf[MAX_SIZE];
    while (1) {
        read_count = read(socket_fd, request_buf, MAXSIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);
        if (read_count != write_count) {...error?...}
    }
}
```

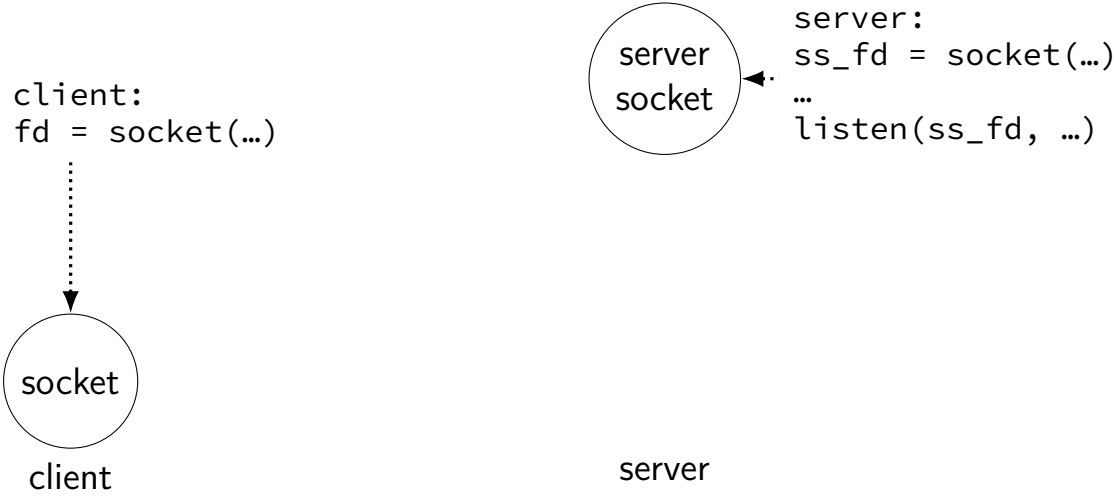
aside: send/recv

sockets have some alternate read/write-like functions:

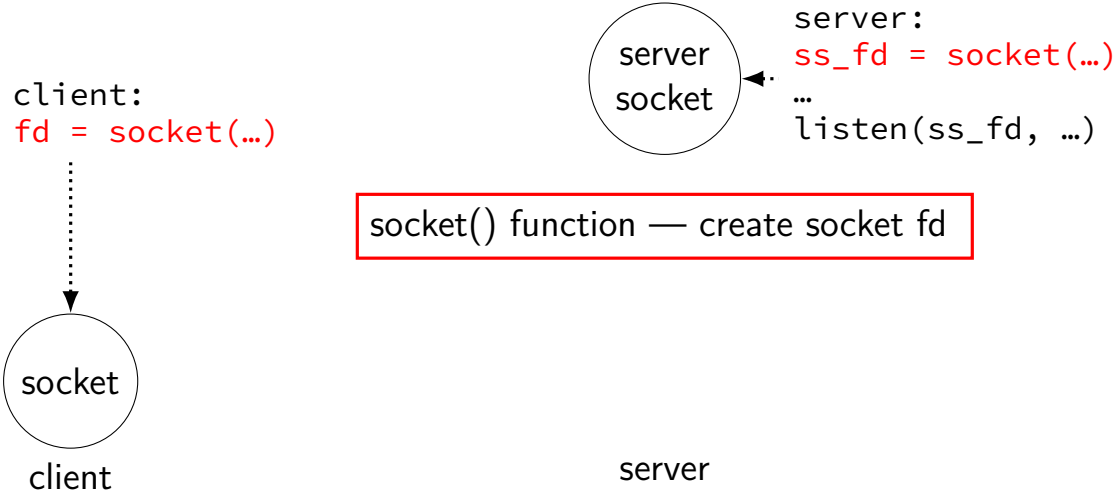
recv, recvfrom, recvmsg
send, sendmsg

have some additional options we won't need in this class

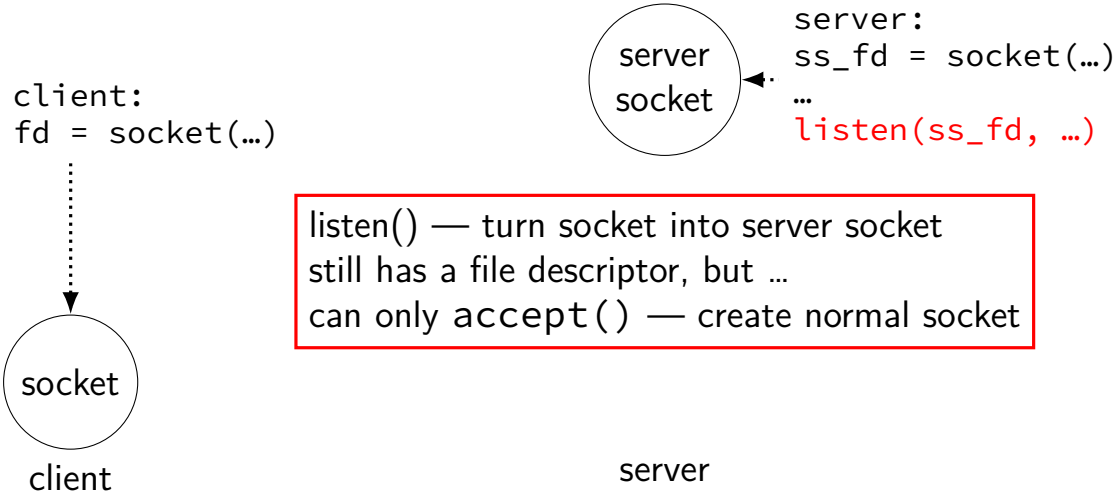
sockets and server sockets



sockets and server sockets

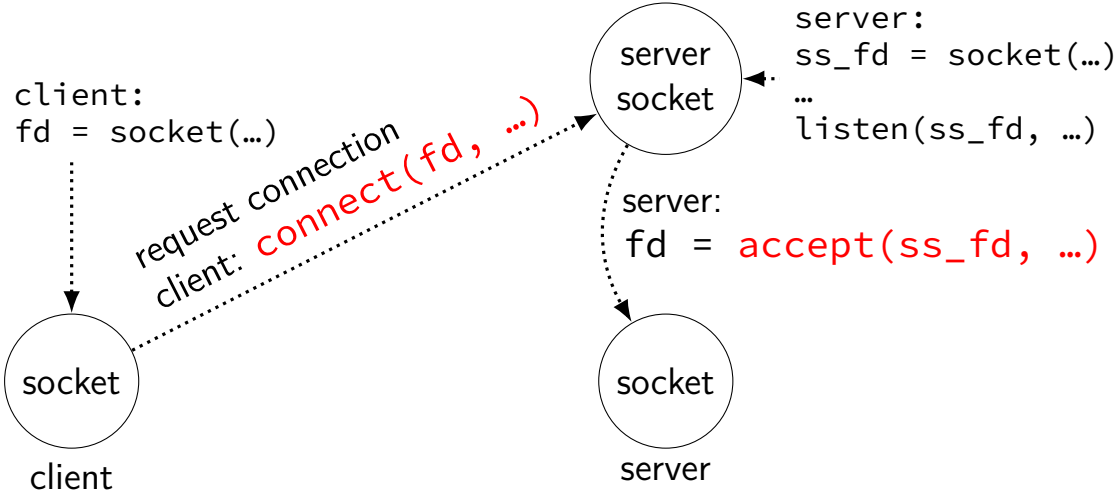


sockets and server sockets

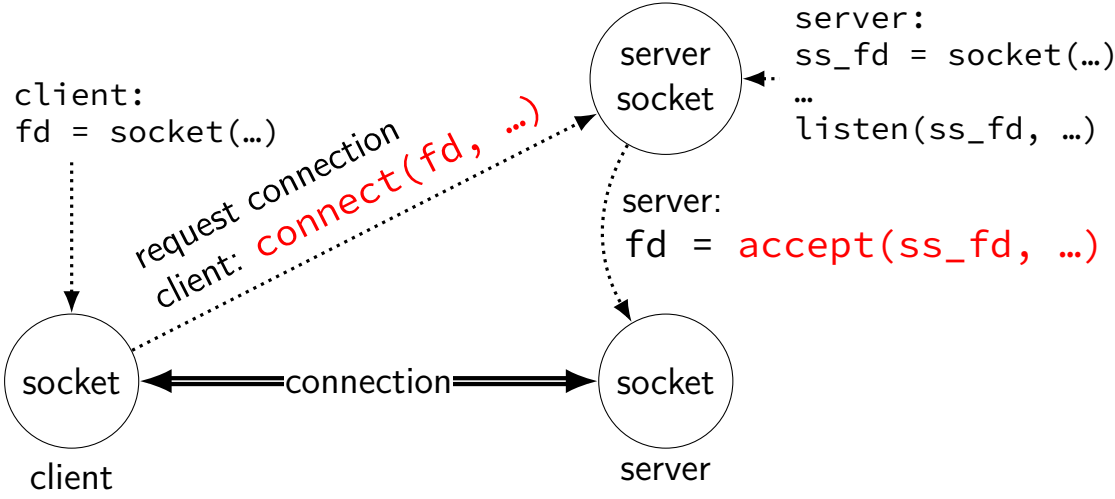


listen() — turn socket into server socket
still has a file descriptor, but ...
can only `accept()` — create normal socket

sockets and server sockets



sockets and server sockets



connections in TCP/IP

connection identified by *5-tuple*

(protocol=TCP, local IP addr., local port, remote IP addr., remote port)

how messages are tagged on the network
(other notable protocol value: UDP)

both ends always have an address+port

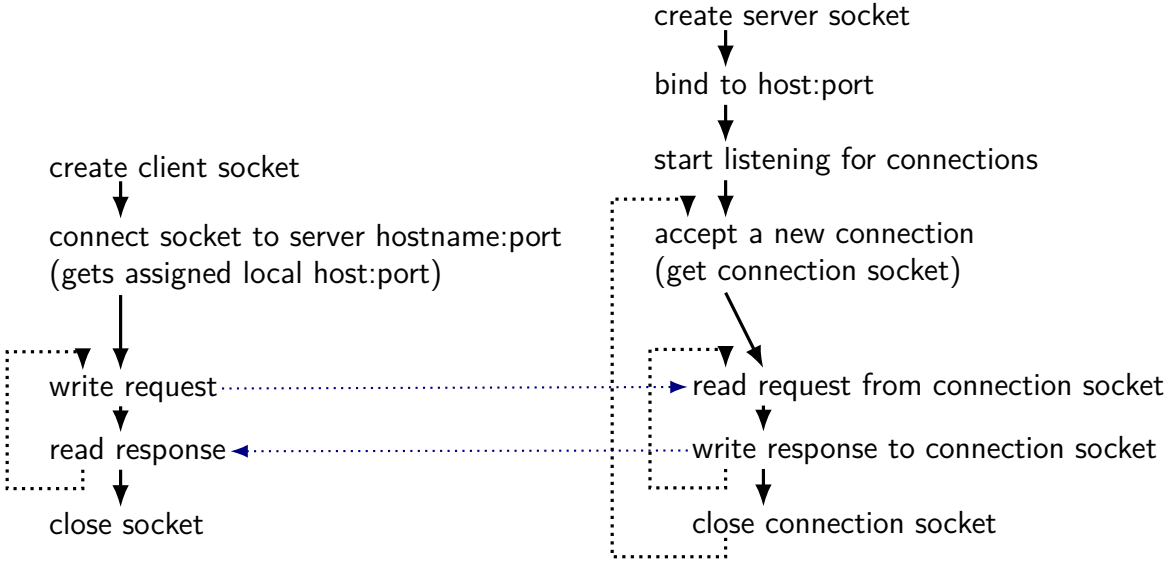
what is the IP address, port number? set with `bind()` function

typically always done for servers, not done for clients
system will choose default if you don't

connections on my desktop

```
cr4bd@reiss-t3620
: /zf14/cr4bd ; netstat --inet --inet6 --numeric
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 128.143.67.91:49202     128.143.63.34:22        ESTABLISHE
tcp        0      0 128.143.67.91:803       128.143.67.236:2049     ESTABLISHE
tcp        0      0 128.143.67.91:50292     128.143.67.226:22       TIME_WAIT
tcp        0      0 128.143.67.91:54722     128.143.67.236:2049     TIME_WAIT
tcp        0      0 128.143.67.91:52002     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:732       128.143.67.236:63439    TIME_WAIT
tcp        0      0 128.143.67.91:40664     128.143.67.236:2049     TIME_WAIT
tcp        0      0 128.143.67.91:54098     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:49302     128.143.67.236:63439    TIME_WAIT
tcp        0      0 128.143.67.91:50236     128.143.67.236:111      TIME_WAIT
tcp        0      0 128.143.67.91:22        172.27.98.20:49566      ESTABLISHE
tcp        0      0 128.143.67.91:51000     128.143.67.236:111      TIME_WAIT
tcp        0      0 127.0.0.1:50438         127.0.0.1:631           ESTABLISHE
tcp        0      0 127.0.0.1:631           127.0.0.1:50438         ESTABLISHE
```

client/server flow (one connection at a time)



connection setup: client — manual addresses

```
int sock_fd;

server = /* code on later slide */;
sock_fd = socket(
    AF_INET, /* IPv4 */
    SOCK_STREAM, /* byte-oriented */
    IPPROTO_TCP
);
if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client — manual addresses

```
int sock_fd;

server = /* code on later slide */;
sock_fd = socket(
    AF_INET, /* IPv4 */
    SOCK_STREAM, /* byte-oriented */
    IPPROTO_TCP
);
if
str
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

specify IPv4 instead of IPv6 or local-only sockets

specify TCP (byte-oriented) instead of UDP (‘datagram’ oriented)

connection setup: client — manual addresses

htonl/s = host-to-network long/short
network byte order = big endian

```
int sock_fd;

server = /* code
sock_fd = socket(
    AF_INET, /* IPv4 */
    SOCK_STREAM, /* byte-oriented */
    IPPROTO_TCP
);
if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client — manual addresses

```
int sock_fd
server = /*
sock_fd = s
    AF_INET
    SOCK_STREAM, /* byte-oriented */
    IPPROTO_TCP
);
if (sock_fd < 0) { /* handle error */ }

struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(2156872459); /* 128.143.67.11 */
addr.sin_port = htons(80); /* port 80 */
if (connect(sock_fd, (struct sockaddr*) &addr, sizeof(addr)) {
    /* handle error */
}
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

struct representing IPv4 address + port number
declared in <netinet/in.h>
see man 7 ip on Linux for docs

sockaddr_in

```
/* from 'man 7 ip' */
struct sockaddr_in {
   sa_family_t    sin_family; /* address family: AF_INET */
   in_port_t      sin_port;   /* port in network byte order */
   struct in_addr sin_addr;   /* internet address */
};

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

sockaddr_in6

```
/* from 'man 7 ipv6' */
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* port number */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];   /* IPv6 address */
};
```

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(
    server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
);
if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(
    server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
     //
    ser
     //
);
if (soc
if (con                                                    0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

addrinfo contains all information needed to setup socket

set by `getaddrinfo` function (next slide)

handles IPv4 and IPv6

handles DNS names, service names

connection setup: client, using addrinfo

```
int sock_fd;
struct addrinfo *server = /* code on next slide */;

sock_fd = socket(
    server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
);
if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: client, using addrinfo

```
int sock_f
struct add

sock_fd =
    server
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
);
if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

ai_addr points to a `struct sockaddr_in*` or a `struct sockaddr_in6*` (cast to a `struct sockaddr*`)

connection setup: client, using addrinfo

since addrinfo contains pointers to dynamically allocated memory, call this function to free everything

```
int
str
sock_fd = socket(
    server->ai_family,
     // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
     // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_prototcol
     // ai_protocol = IPPROTO_TCP or ...
);
if (sock_fd < 0) { /* handle error */ }
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* handle error */
}
freeaddrinfo(server);
DoClientStuff(sock_fd); /* read and write from sock_fd */
close(sock_fd);
```

connection setup: lookup address

```
/* example hostname, portname = "www.cs.virginia.edu", "443" */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; /* for IPv4 OR IPv6 */
// hints.ai_family = AF_INET4; /* for IPv4 only */

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

connection setup: lookup address

```
/* example hostname, portname = "www.cs.virginia.edu", "443" */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family
// hints.ai_fam

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

NB: pass pointer to addrinfo to fill in

connection setup: lookup address

AF_UNSPEC: choose between IPv4 and IPv6 for me
AF_INET, AF_INET6: choose IPv4 or IPV6 respectively

```
/* exam                                                    " */
const c
...
struct addrinfo *server;
struct addrinfo hints;
int rv;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC; /* for IPv4 OR IPv6 */
// hints.ai_family = AF_INET4; /* for IPv4 only */

hints.ai_socktype = SOCK_STREAM; /* byte-oriented --- TCP */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

/* eventually freeaddrinfo(result) */
```

connection setup: multiple server addresses

```
struct addrinfo *server;
...
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

for (struct addrinfo *current = server; current != NULL;
      current = current->ai_next) {
    sock_fd = socket(current->ai_family, current->ai_socktype, curre
    if (sock_fd < 0) continue;
    if (connect(sock_fd, current->ai_addr, current->ai_addrlen) == 0
        break;
    }
    close(sock_fd); // connect failed
}
freeaddrinfo(server);
DoClientStuff(sock_fd);
close(sock_fd);
```

connection setup: multiple server addresses

```
struct addrinfo *server;
...
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }

for (struct addrinfo *current = server; current != NULL;
       current = current->ai_next) {
    sock_fd = socket(current->ai_family, current->ai_socktype, curre
    if (sock_fd < 0) continue;
    if (connect(sock_fd, current->ai_addr, current->ai_addrlen) == 0
        break;
    }
    clos
}
freeaddr
DoClient
close(so
```

addrinfo is a linked list

name can correspond to multiple addresses

example: redundant copies of web server

example: an IPv4 address and IPv6 address

example: wired + wireless connection on one machine

connection setup: old lookup function

```
/* example hostname, portnum= "www.cs.virginia.edu", 443*/
const char *hostname; int portnum;
...
struct hostent *server_ip;
server_ip = gethostbyname(hostname);

if (server_ip == NULL) { /* handle error */ }

struct sockaddr_in addr;
addr.s_addr = *(struct in_addr*) server_ip->h_addr_list[0];
addr.sin_port = htons(portnum);
sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
connect(sock_fd, &addr, sizeof(addr));
...
```

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen
...
int socket_fd = accept(server_socket_fd, NULL);
```

INADDR_ANY: accept connections for any address I can!

alternative: specify specific address

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
list
...
int socket_fd = accept(server_socket_fd, NULL);
```

bind to 127.0.0.1? only accept connections from same machine

what we recommend for FTP server assignment

connection setup: server, manual

```
int server_socket_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY; /* "any address I can use" */
    /* or: addr.s_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
    /* or: addr.s_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

choose the number of unaccepted connections

aside: on server port numbers

Unix convention: must be `root` to use ports 0–1023

root = superuser = 'adminstrator user' = what sudo does

so, for testing: probably ports > 1023

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags = AI_PASSIVE;

rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }
```

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.a                                    care */
hints.ai_flags =

rv = getaddrinfo(                                   ver);
if (rv != 0) { /* handle error */ }
```

hostname can be NULL
means “use all possible addresses”
only makes sense for servers

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *hostname; const char *portname;
...
struct addrinfo *server;
struct addrinfo hints;
int rv;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* don't care */
hints.ai_flags = AI_PASSIVE;

rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }
```

portname can also be NULL
means “choose a port number for me”
only makes sense for servers

connection setup: server, address setup

```
/* example (hos                                          3") */
const char *hos
...
struct addrinfo *server;
struct addrinfo hints;
int rv;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET; /* for IPv4 */
/* or: */ hints.ai_family = AF_INET6; /* for IPv6 */
/* or: */ hints.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags = AI_PASSIVE;

rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) { /* handle error */ }
```

AI_PASSIVE: “I’m going to use bind”

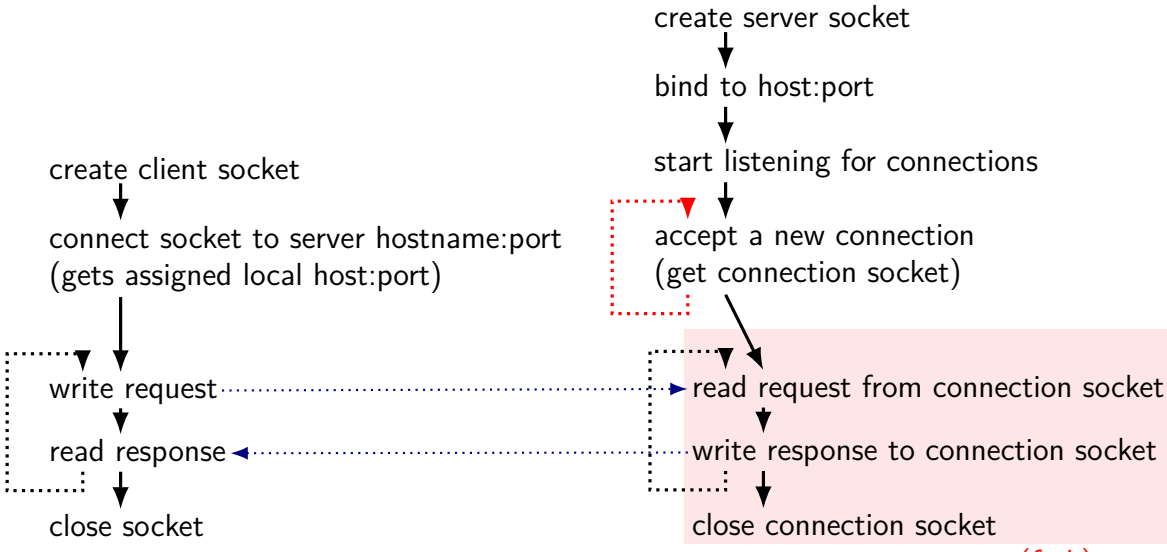
connection setup: server, addrinfo

```
struct addrinfo *server;
... getaddrinfo(...) ...

int server_socket_fd = socket(
    server->ai_family,
    server->ai_sockttype,
    server->ai_protocol
);

if (bind(server_socket_fd, ai->ai_addr, ai->ai_addr_len)) < 0) {
    /* handle error */
}
listen(server_socket_fd, MAX_NUM_WAITING);
...
int socket_fd = accept(server_socket_fd, NULL);
```

client/server flow (multiple connections)



incomplete writes

`write` might write less than requested

- error after writing some data
- if blocking disabled with `fcntl()`, buffer full

`read` might read less than requested

- error after reading some data
- not enough data got there in time

handling incomplete writes

```
bool write_fully(int fd, const char *buffer, ssize_t count) {
    const char *ptr = buffer;
    const char *end = buffer + count;
    while (ptr != end) {
        ssize_t written = write(fd, (void*) ptr, end - ptr);
        if (written == -1) {
            return false;
        }
        ptr += written;
    }
    return true;
}
```

on filling buffers

```
char buffer[SIZE];
ssize_t buffer_used = 0;

int fill_buffer(int fd) {
    ssize_t amount = read(
        fd, buffer + buffer_used, SIZE - buffer_used
    );
    if (amount == 0) {
        /* handle EOF */ ???
    } else if (amount == -1) {
        return -1;
    } else {
        buffer_used += amount;
    }
}
```

reading lines

(note: code below is not tested)

```
int read_line(int fd, const char *p_line, size_t *p_size) {
    const char *newline;
    while (1) {
        newline = memchr(buffer, '\n', buffer_used);
        if (newline != NULL || buffer_used == SIZE) break;
        fill_buffer();
    }
    memcpy(p_line, buffer, newline – buffer);
    *p_size = newline – buffer;
    memmove(newline, buffer, buffer + SIZE – newline);
    buffer_end –= (newline – buffer);
}
```

aside: getting addresses

on a socket fd: `getsockname` = local addresss

- `sockaddr_in` or `sockaddr_in6`
- IPv4/6 address + port

on a socket fd: `getpeername` = remote address

addresses to string

can access numbers/arrays in `sockaddr_in`/`in6` directly

another option: `getnameinfo`

- supports getting W.X.Y.Z form or looking up a hostname

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR "socket option"

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn't check for reading multiple lines at once

example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

- OS kills program trying to write to closed socket/pipe

set the SO_REUSEADDR "socket option"

- default: OS reserves port number for a while after server exits
- this allows keeps it unreserved
- allows us to bind() immediately after closing server

client handles reading until a newline

- but doesn't check for reading multiple lines at once

reading and writing at once

so far assumption: alternate between reading+writing

- sufficient for FTP assignment
- how many protocols work

"half-duplex"

don't have to use sockets this way, but tricky

threads: one reading thread, one writing thread *OR*

event-loop: use *non-blocking I/O* and select()/poll()/etc. functions

- non-blocking I/O setup with fcntl() function
- non-blocking write() fills up buffer as much as possible, then returns
- non-blocking read() returns what's in buffer, never waits for more

beyond threads: event based programming

writing server that servers multiple clients?

- e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

- one network, not that fast

idea: one thread handles multiple connections

beyond threads: event based programming

writing server that servers multiple clients?

e.g. multiple web browsers at a time

maybe don't really need multiple processors/cores

one network, not that fast

idea: one thread handles multiple connections

issue: read from/write to multiple streams at once?

event loops

```
while (true) {
    event = WaitForNextEvent();
    switch (event.type) {
    case NEW_CONNECTION:
        handleNewConnection(event); break;
    case CAN_READ_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleRead(connection);
        break;
    case CAN_WRITE_DATA_WITHOUT_WAITING:
        connection = LookupConnection(event.fd);
        handleWrite(connection);
        break;
        ...
    }
}
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + com
                     sizeof(command) -
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length - 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

some single-threaded processing code

```
void ProcessRequest(int fd) {
    while (true) {
        char command[1024] = {};
        size_t comamnd_length = 0;
        do {
            ssize_t read_result =
                read(fd, command + com
                     sizeof(command) -
            if (read_result <= 0) handle_error();
            command_length += read_result;
        } while (command[command_length - 1] != '\n');
        if (IsExitCommand(command)) { return; }
        char response[1024];
        computeResponse(response, commmand);
        size_t total_written = 0;
        while (total_written < sizeof(response)) {
            ...
        }
    }
```

```
class Connection {
  int fd;
  char command[1024];
  size_t command_length;
  char response[1024];
  size_t total_written;
  ...
};
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

as event code

```
handleRead(Connection *c) {
    ssize_t read_result =
        read(fd, c->command + command_length,
             sizeof(command) - c->command_length);
    if (read_result <= 0) handle_error();
    c->command_length += read_result;

    if (c->command[c->command_length - 1] == '\n') {
        computeResponse(c->response, c->command);
        if (IsExitCommand(command)) {
          FinishConnection(c);
        }
        StopWaitingToRead(c->fd);
        StartWaitingToWrite(c->fd);
    }
}
```

POSIX support for event loops

`select` and `poll` functions

- take list(s) of file descriptors to read and to write
- wait for them to be read/writeable without waiting
- (or for new connections associated with them, etc.)

many OS-specific extensions/improvements/alternatives:

- examples: Linux epoll, Windows IO completion ports
- better ways of managing list of file descriptors
- do read/write when ready instead of just returning when reading/writing is okay