networking 2

so far

building programs — Makefiles for automation, dynamic libraries

hardware support for *processes*

- kernel mode: operations for just the OS
- exceptions: way (for hardware or software) to ask OS for help
- context switches: switch active *thread* on processor
- virtual memory: let OS choose where program's memory goes
  - table of: virtual page → physical page

accounts and OS-enforced isolation

networking — layered implementation

- simulating streams of data with messages
- routing to connect local networks

last time (1)

common points of confusion re: page tables assignment

programming model in networking lab

networking layers

last time (2)

nesting layers

- higher layers implemented on interface of below
- sometimes more layering

addresses versus names

- most addresses = numbers

link layer — local network

routing (network layer)

- routing tables to know how to forward messages

port numbers (which program? transport layer)

UDP (transport without streams) v TCP (reliable streams)

lab tomorrow

code review

special-case permitted collaboration!

get/give suggestions for improving code

- better organization
- more readable
- better style
- pointing out potential problems you might not have seen

(not about others debugging/writing your code)

anonymous feedback (1)

"Myself as well as practically all of my friends in this course are extremely confused and pretty much have no idea what is going on, despite attending lectures and completing quizzes/homework assignments. It almost feels like we need a lecture to just catch up and make sure everyone is on the same page with regards to assignments and lecture content because we're all confused, and it seems like we are on an unsustainable path for the remainder of this semester."

I can't tell what such a "catch-up" lecture should cover

don't want to give a lecture that reviews just the things that aren't confusing

lack of connection to high-level goals in each topic? lack of conclusion for topics?

disconnect between assignments and lecture?

scattershot lectures from switching between review for pagetable assignment and new material?

anonymous feedback (2)

"Today in class you mentioned that we needed to test our code in parts and I was wondering how we would test page allocate? I am not sure what we should expect as an output for memory."

probably answering this too late to be useful

- anonymous feedback isn't good for quick answers

some ideas?

- make parts of page_allocate into smaller functions that can be run separately
- add some counters or similar variables to track what page allocate does and check those
- examine ptbr afterwards and look up the value of a particular page table entry (with locatoin hard-coded in test)
- manually set ptbr to something and see what page allocate does to it
- implement and test, for example, page_allocate(0)+translate(0) before more complex cases

anonymous feedback (3)

"In office hours, TAs do not always know how to allocate time correctly. I saw one TA help a student for over an hour on multiple assignments in one sitting. I ended up leaving even after waiting an hour and a half in near-empty office hours. I thought TAs were only supposed to allocate $\sim$10 minutes per student at a time. "

definitely shouldn't be happening to this extreme

some TAs report students not signing up on queue (whiteboard or online) so TAs aren't aware students need help — probably means queue isn't clear enough sometimes?

anonymous feedback (4)

"I appreciate how receptive you are to our prior knowledge and feedback! I know this is the first time this class is being taught after the pilot and the ending of last semester largely impacted what we actually know versus what we are expected to know. It is nice to know how much you care about us and our success in this class."

anonymous feedback (5)

"While sockets were a part of the CSO curriculum, due to the circumstances at the end of last semester, we didn't really learn them and the assignments related to sockets were optional/dropped. With that in mind please review more before we get into it to much <3"

"Due to the events of the latter half of last semester, we didn't cover sockets super well and in depth. It would be nice to have a short refreshed in the beginning of class."

talking with the terminal

```
printf("Name: ");
char input[1000];
fgets(input, sizeof input, stdin);
fprintf(logfile, "Got name %s\n", input);
printf("Enter command: ");
fgets(input, sizeof input, stdin);
...
```

talking with the terminal

```
printf("Name: ");
char input[1000];
fgets(input, sizeof input, stdin);
fprintf(logfile, "Got name %s\n", input);
printf("Enter command: ");
fgets(input, sizeof input, stdin);
...
```

talking with terminal w/ fread

```
/* missing below: error checking */
const char *msg = "Name: ";
fwrite(msg, 1, strlen(msg), stdout);

char input[1000] = ""; int count = 0;
do {
    count += fread(input + count, 1, 1000 - count, stdin);
} while (!strchr(input, '\n') && count < 1000);
fprintf(logfile, "Got name %s\n", first_line_of(input));

msg = "Enter command: "; fwrite(msg, 1, strlen(msg), stdout);

strcpy(input, after_first_line_of(input)); count = strlen(input);
while (!strchr(input, '\n') && count < 1000) {
    count += fread(input + count, 1, 1000 - count, stdin);
}
...
```

talking with terminal w/ fread

```
/* missing below: error checking */
const char *msg = "Name: ";
fwrite(msg, 1, strlen(msg), stdout);

char input[1000] = ""; int count = 0;
do {
    count += fread(input + count, 1, 1000 - count, stdin);
} while (!strchr(input, '\n') && count < 1000);
fprintf(logfile, "Got name %s\n", first_line_of(input));

msg = "Enter command: "; fwrite(msg, 1, strlen(msg), stdout);

strcpy(input, after_first_line_of(input)); count = strlen(input);
while (!strch
    count +=                                    stdin);
}
...
```

ugh, reading a line of input without fgets
and without doing 1 char at a time
is pretty annoying

using a connected socket

```
/* missing below: error checking */
int socket_fd = GetSocketFileDescriptorSomehow();
const char *msg = "Name: ";
write(socket_fd, msg, strlen(msg));

char input[1000]; int count = 0;
do {
    count += read(socket_fd, input + count, 1000 - count);
} while (!strchr(input, '\n') && count < 1000);
fprintf(logfile, "Got name %s\n", first_line_of(input));

msg = "Enter command: "; write(socket_fd, msg, strlen(msg));

strcpy(input, after_first_line_of(input)); count = strlen(input);
while (!strchr(input, '\n') && count < 1000) {
    count += read(socket_fd, input + count, 1000 - count);
}
...
```

using a connected socket

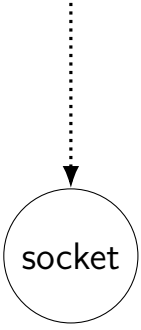
```
/* missing below: error checking */
int socket_fd = GetSocketFileDescriptorSomehow();
const char *msg = "Name: ";
write(socket_fd, msg, strlen(msg));

char input[1000]; int count = 0;
do {
    count += read(socket_fd, input + count, 1000 - count);
} while (!strchr(input, '\n') && count < 1000);
fprintf(logfile, "Got name %s\n", first_line_of(input));

msg = "Enter command: "; write(socket_fd, msg, strlen(msg));

strcpy(input, after_first_line_of(input)); count = strlen(input);
while (!strchr(input, '\n') && count < 1000) {
    count += read(socket_fd, input + count, 1000 - count);
}
...
```

using a connected socket

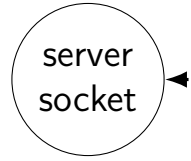
```
/* missing below: error checking */
int socket_fd = GetSocketFileDescriptorSomehow();
const char *msg = "Name: ";
write(socket_fd, msg, strlen(msg));

char input[1000]; int count = 0;
do {
    count += read(socket_fd, input + count, 1000 - count);
} while (!strchr(input, '\n') && count < 1000);
fprintf(logfile, "Got name %s\n", first_line_of(input));

msg = "Enter command: "; write(socket_fd, msg, strlen(msg));

strcpy(input, after_first_line_of(input)); count = strlen(input);
while (!strchr(input, '\n') && count < 1000) {
    count += read(socket_fd, input + count, 1000 - count);
}
...
```

sockets and server sockets

```
client:
fd = socket(…)
(rarely) bind(fd, addr, …)
```



client



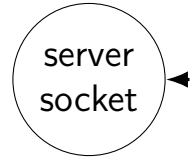

```
server:
ss_fd = socket(…)
…
bind(ss_fd, addr, …)
listen(ss_fd, …)
```

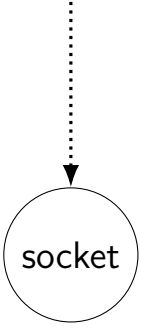
server

sockets and server sockets

```
client:
fd = socket(…)
(rarely) bind(fd, addr, …)
```

```
server:
ss_fd = socket(…)
…
bind(ss_fd, addr, …)
listen(ss_fd, …)
```





client

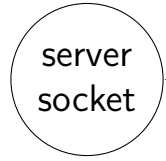
socket() function — create socket *file descriptor*
file descriptor = number that identifies a open file
(like a FILE *, but for Unix system calls)

server

sockets and server sockets

```
client:
fd = socket(…)
(rarely) bind(fd, addr, …)
```

client




```
server:
ss_fd = socket(…)
…
bind(ss_fd, addr, …)
listen(ss_fd, …)
```

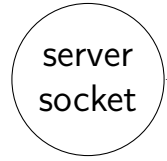
bind() — set local port number
and maybe IP address to use
if not done, OS chooses port number

server

sockets and server sockets

```
client:
fd = socket(…)
(rarely) bind(fd, addr, …)
```

client

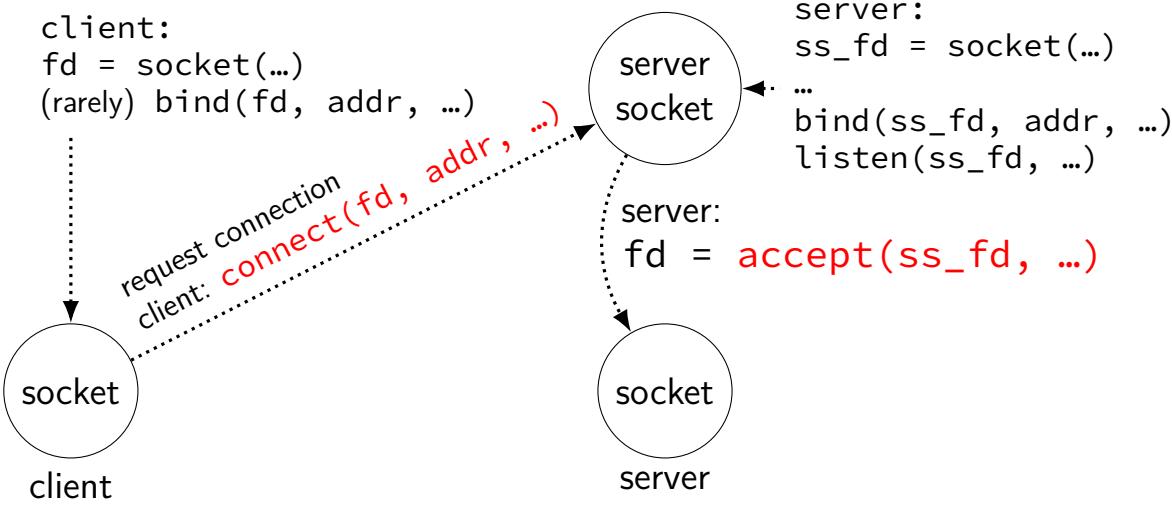



```
server:
ss_fd = socket(…)
…
bind(ss_fd, addr, …)
listen(ss_fd, …)
```

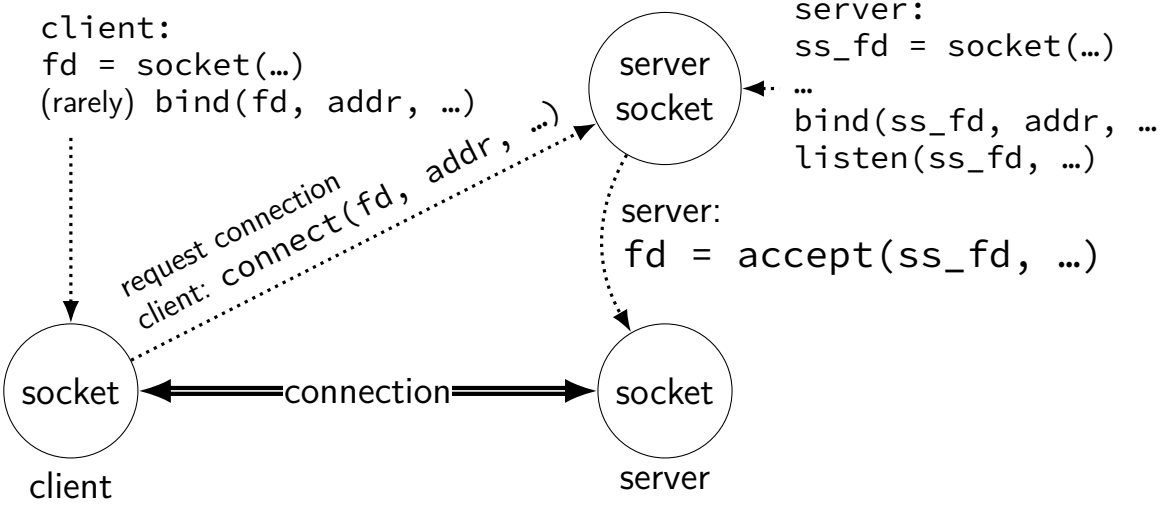
listen() — turn socket into server socket
still has a file descriptor, but …
can only `accept()` — create normal socket

server

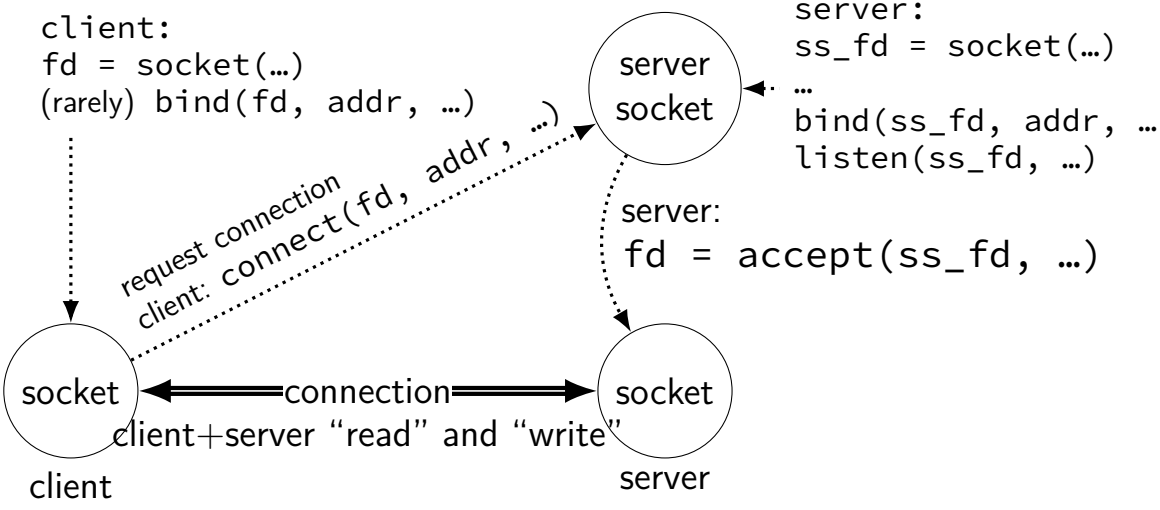
sockets and server sockets



sockets and server sockets



sockets and server sockets



layers

application	HTTP, SSH, SMTP, ...	application-defined meanings
transport	TCP, UDP, ...	reach correct program, reliablity/streams
network	IPv4, IPv6, ...	reach correct machine (across networks)
link	Ethernet, Wi-Fi, ...	coordinate shared wire/radio
physical	...	encode bits for wire/radio

names and addresses

name	address
logical identifier	location/how to locate
variable `counter`	memory address `0x7FFF9430`
DNS name `www.virginia.edu`	IPv4 address `128.143.22.36`
DNS name `mail.google.com`	IPv4 address `216.58.217.69`
DNS name `mail.google.com`	IPv6 address `2607:f8b0:4004:80b::2005`
DNS name `reiss-t3620.cs.virginia.edu`	IPv4 address `128.143.67.91`
DNS name `reiss-t3620.cs.virginia.edu`	MAC address `18:66:da:2e:7f:da`
service name `https`	port number 443
service name `ssh`	port number 22

layers

application	HTTP, SSH, SMTP, ...	application-defined meanings
transport	TCP, UDP, ...	reach correct program, reliablity/streams
network	IPv4, IPv6, ...	reach correct machine (across networks)
link	Ethernet, Wi-Fi, ...	coordinate shared wire/radio
physical	...	encode bits for wire/radio

UDP v TCP

TCP: stream to other program

- reliable transmission of as much data as you want
- "connecting" fails if server not responding
- write(fd, "a", 1); write(fd, "b", 1) = write(fd, "ab", 2)
- (at least) one socket per remote program being talked to

UDP: messages sent to program, but no reliablity/streams

- unreliable transmission of short messages
- write(fd, "a", 1); write(fd, "b", 1) $\neq$ write(fd, "ab", 2)
- "connecting" just sets default destination
- can sendto()/recvfrom() multiple other programs with one socket
  - (but don't have to)

‘connected’ UDP sockets

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in my_addr= ...;
/* set local IP address + port */
bind(fd, &my_addr, sizeof(my_addr))
struct sockaddr_in to_addr = ...;
connect(fd, &to_addr); /* set remote IP address + port */
    /* doesn't actually communicate with remote address yet */
...
int count = write(fd, data, data_size);
// OR
int count = send(fd, data, data_size, 0 /* flags */);
    /* single message -- sent ALL AT ONCE */

int count = read(fd, buffer, buffer_size);
// OR
int count = recv(fd, buffer, buffer_size, 0 /* flags */);
    /* receives whole single message ALL AT ONCE */
```

UDP sockets on IPv4

```
int fd = socket(AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in my_addr= ...;
/* set local IP address + port */
if (0 != bind(fd, &my_addr, sizeof(my_addr)))
    handle_error();
...
struct sockaddr_in to_addr = ...;
    /* send a message to specific address */
int bytes_sent = sendto(fd, data, data_size, 0 /* flags */,
    &to_addr, sizeof(to_addr));

struct sockaddr_in from_addr = ...;
    /* receive a message + learn where it came from */
int bytes_recvd = recvfrom(fd, &buffer[0], buffer_size, 0,
    &from_addr, sizeof(from_addr));
...
```

finding the read()

when message comes in,
how does OS know which `read()`/`recv()`/`recvfrom()` call its for?

connections in TCP/IP

connection identified by *5-tuple*

used by OS to lookup "where is the socket?"

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

local IP address, port number can be 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>/u/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
```

non-connection sockets

TCP servers waiting for connections +
UDP sockets with no particular remote host

Linux: OS keeps 5-tuple with “wildcard” remote address

“listening” sockets on my desktop

```
cr4bd@reiss-t3620>/u/cr4bd
$ netstat --inet --inet6 --numeric --listen
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.1:38537         0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:36777         0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:41099           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:45291           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:51949         0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:41071         0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:32881         0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:38673         0.0.0.0:*               LISTEN
....
tcp6       0      0 :::42689                :::*                    LISTEN
udp        0      0 128.143.67.91:60001     0.0.0.0:*
udp        0      0 128.143.67.91:60002     0.0.0.0:*
...
udp6       0      0 :::59938                :::*
```

TCP state machine

TIME_WAIT, ESTABLISHED, ...?

OS tracks "state" of TCP connection

- am I just starting the connection?
- is other end ready to get data?
- am I trying to close the connection?
- do I need to resend something?

standardized set of state names

TIME_WAIT

remember delayed messages?

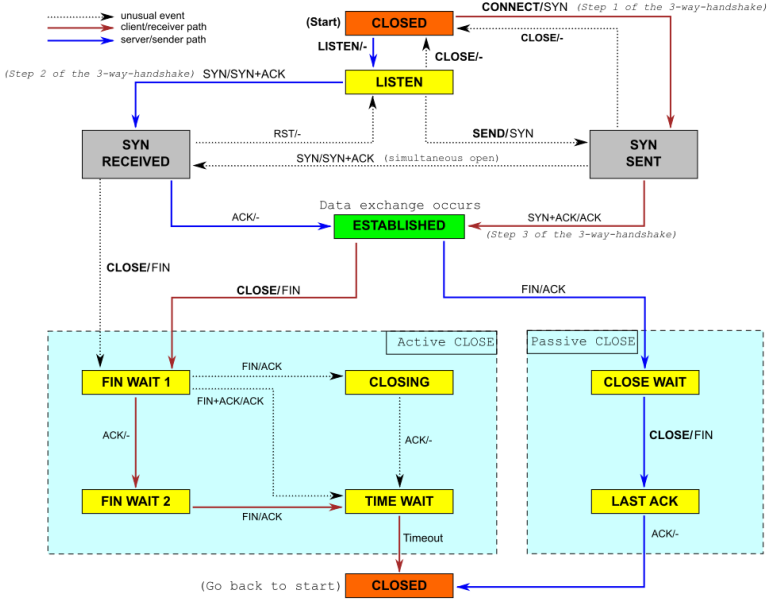
problem for TCP ports

if I reuse port number, I can get message from old connection

solution: TIME_WAIT to make sure connection really done

- done after sending last message in connection

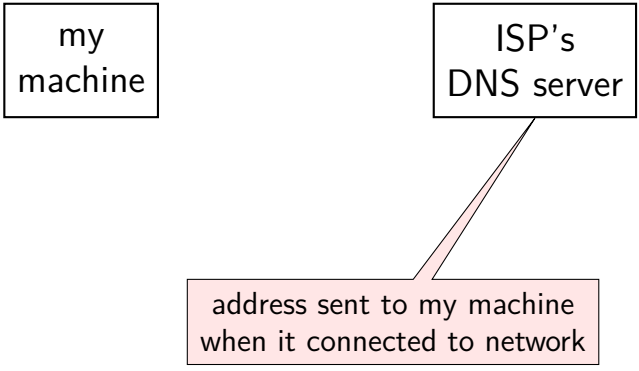
TCP state machine picture



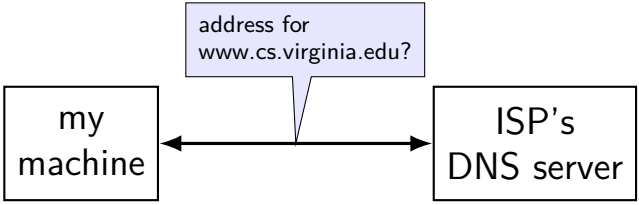
names and addresses

name	address
logical identifier	location/how to locate
variable `counter`	memory address `0x7FFF9430`
DNS name `www.virginia.edu`	IPv4 address `128.143.22.36`
DNS name `mail.google.com`	IPv4 address `216.58.217.69`
DNS name `mail.google.com`	IPv6 address `2607:f8b0:4004:80b::2005`
DNS name `reiss-t3620.cs.virginia.edu`	IPv4 address `128.143.67.91`
DNS name `reiss-t3620.cs.virginia.edu`	MAC address `18:66:da:2e:7f:da`
service name `https`	port number 443
service name `ssh`	port number 22

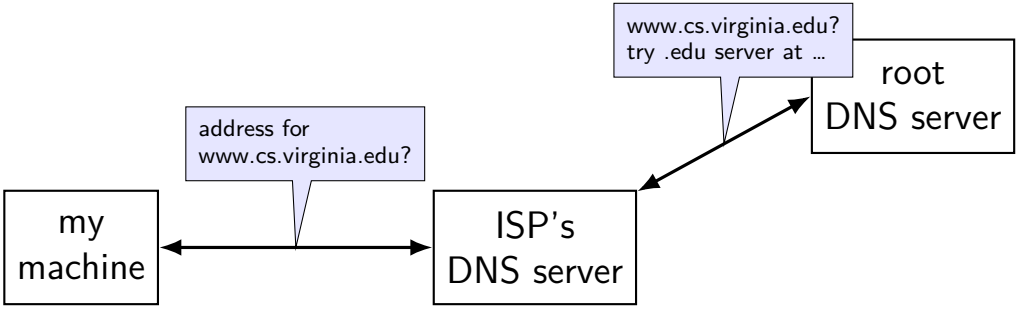
DNS: distributed database



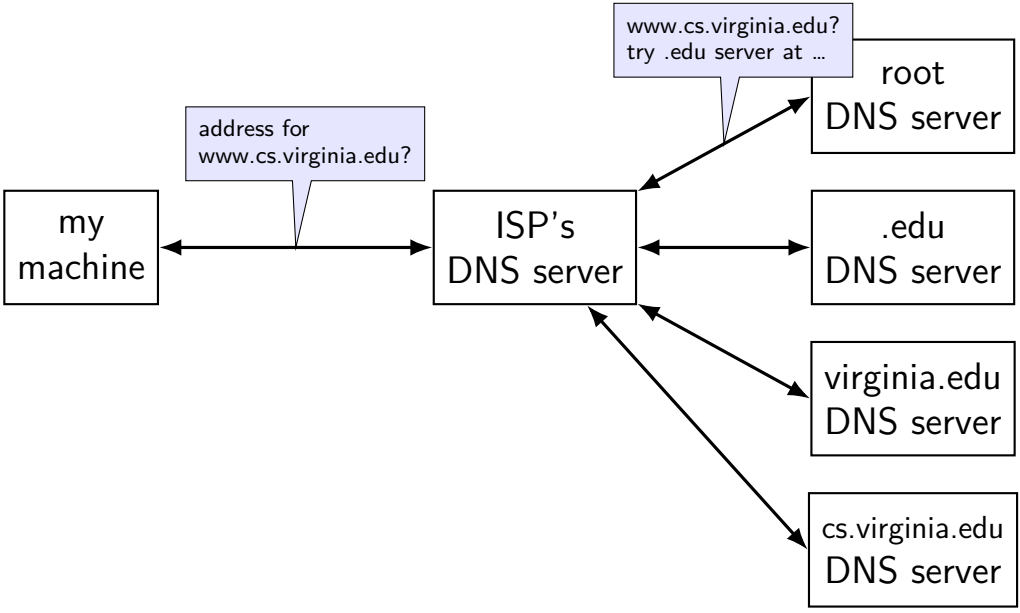
DNS: distributed database



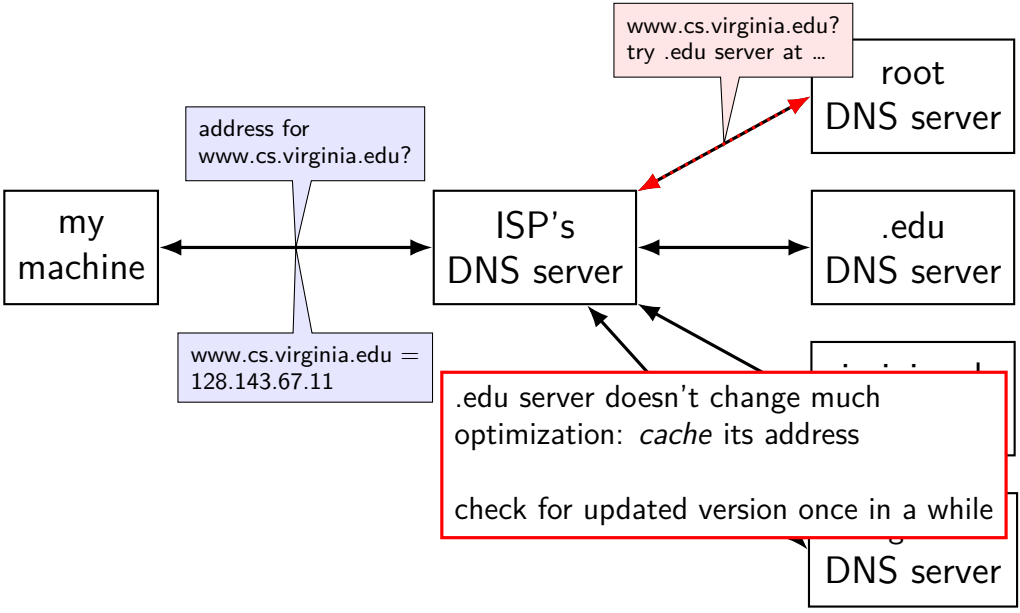
DNS: distributed database



DNS: distributed database



DNS: distributed database



querying the root

```
$ dig +trace +all www.cs.virginia.edu
...
edu.                    172800      IN      NS      b.edu-servers.net.
edu.                    172800      IN      NS      f.edu-servers.net.
edu.                    172800      IN      NS      i.edu-servers.net.
edu.                    172800      IN      NS      a.edu-servers.net.
...
b.edu-servers.net.      172800      IN      A       191.33.14.30
b.edu-servers.net.      172800      IN      AAAA       2001:503:231d::2:30
f.edu-servers.net.      172800      IN      A       192.35.51.30
f.edu-servers.net.      172800      IN      AAAA       2001:503:d414::30
...
;; Received 843 bytes from 198.97.190.53#53(h.root-servers.net) in 8 ms
...
```

querying the edu

```
$ dig +trace +all www.cs.virginia.edu
...
virginia.edu.                172800        IN        NS        nom.virginia.edu.
virginia.edu.                172800        IN        NS        uvaarpa.virginia.edu.
virginia.edu.                172800        IN        NS        eip-01-aws.net.virginia.edu.
nom.virginia.edu.        172800        IN        A        128.143.107.101
uvaarpa.virginia.edu.        172800        IN        A        128.143.107.117
eip-01-aws.net.virginia.edu. 172800 IN        A        44.234.207.10
;; Received 165 bytes from 192.26.92.30#53(c.edu-servers.net) in 40 ms
...
```

querying virginia.edu+cs.virginia.edu

```
$ dig +trace +all www.cs.virginia.edu
...
cs.virginia.edu.        3600        IN        NS        coresrv01.cs.virginia.edu.
coresrv01.cs.virginia.edu. 3600        IN        A        128.143.67.11
;; Received 116 bytes from 44.234.207.10#53(eip-01-aws.net.virginia.edu) in 72 ms

www.cs.Virginia.EDU.        172800        IN        A        128.143.67.11
cs.Virginia.EDU.        172800        IN        NS        coresrv01.cs.Virginia.EDU.
coresrv01.cs.Virginia.EDU. 172800 IN        A        128.143.67.11
;; Received 151 bytes from 128.143.67.11#53(coresrv01.cs.virginia.edu) in 4 ms
```

querying typical ISP's resolver

```
$ dig www.cs.virginia.edu
...
;; ANSWER SECTION:
www.cs.Virginia.EDU.          7183        IN        A        128.143.67.11
..
```

cached response

valid for 7183 more seconds

after that everyone needs to check again

DNS time-to-live

don't want DNS entries cached forever

solution: time-to-live

"www.cs.virginia.edu is 128.148.67.11 for next 86400 seconds"

DNS exercise (1)

"www.cs.virginia.edu is 128.148.67.11 for next 86400 seconds"

(given record above) if sysadmin changes IP address DNS server returns for www.cs.virginia.edu, then what will happen to machines accessing website?

A. they'll start using the new address after 86400 seconds, and use the old one before then.

B. different machines will use the new address at different times, but no longer than 86400 seconds from when it changes

C. machines will start using the new address almost immediately, but after some small delay after it is changed

D. machines may keep using the old address until they are rebooted

E. something else?

DNS exercise (2)

if sysadmin wants to change the IP address of www.cs.virginia.edu, how do they do this without downtime?

they can change the IP address the server returns and/or the time-to-live?

what should they change and when to smoothly transition to a new address?

names and addresses

name	address
logical identifier	location/how to locate
variable `counter`	memory address `0x7FFF9430`
DNS name `www.virginia.edu`	IPv4 address `128.143.22.36`
DNS name `mail.google.com`	IPv4 address `216.58.217.69`
DNS name `mail.google.com`	IPv6 address `2607:f8b0:4004:80b::2005`
DNS name `reiss-t3620.cs.virginia.edu`	IPv4 address `128.143.67.91`
DNS name `reiss-t3620.cs.virginia.edu`	MAC address `18:66:da:2e:7f:da`
service name `https`	port number 443
service name `ssh`	port number 22

two types of addresses?

MAC addreses: on link layer

IP addresses: on network layer

how do we know which MAC address to use?

a table on my desktop

my desktop:

```
$ arp -an
? (128.143.67.140) at 3c:e1:a1:18:bd:5f [ether] on enp0s31f6
? (128.143.67.236) at <incomplete> on enp0s31f6
? (128.143.67.11) at 30:e1:71:5f:39:10 [ether] on enp0s31f6
? (128.143.67.92) at <incomplete> on enp0s31f6
? (128.143.67.5) at d4:be:d9:b0:99:d1 [ether] on enp0s31f6
```

...

how is that table made?

ask machines on local network (same switch)

“Who has 128.148.67.140”

the correct one replies

what about non-local machines?

when configuring network specify:

range of addresses to expect on local network

- 128.148.67.0-128.148.67.255 on my desktop
- "netmask"

gateway machine to send to for things outside my local network

- 128.143.67.1 on my desktop
- my desktop looks up the corresponding MAC address

routes on my desktop

```
$ /sbin/route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         128.143.67.1    0.0.0.0         UG    100    0        0 enp0s31f6
128.143.67.0    0.0.0.0         255.255.255.0   U     100    0        0 enp0s31f6
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s31f6
```

network configuration says:

(line 2) to get to 128.143.67.0–128.143.67.255, send directly on local network

> "genmask" is mask (for bitwise operations) to specify how big range is

(line 3) to get to 169.254.0.0–169.254.255.255, send directly on local network

(line 1) to get anywhere else, use "gateway" 128.143.67.1

URL / URIs

Uniform Resource Locators (URL)

tells how to find “resource” on network

Unifrom Resources Identifiers

superset of URLs

URI examples

```
https://kytos02.cs.virginia.edu:443/cs3130-spring2023/
                quizzes/quiz.php?qid=02#q2

https://kytos02.cs.virginia.edu/cs3130-spring2023/
                quizzes/quiz.php?qid=02

https://www.cs.virginia.edu/

sftp://cr4bd@portal.cs.virginia.edu/u/cr4bd/file.txt

tel:+1-434-982-2200

//www.cs.virginia.edu/~cr4bd/3130/S2023/
/~cr4bd/3130/S2023
```

scheme and/or host implied from context

URI generally

`scheme://authority/path?query#fragment`

scheme: — what protocol

//authority/

authoirty = user@host:port OR host:port OR user@host OR host

path

which resource

?query — usually key/value pairs

#fragment — place in resource

most components (sometimes) optional

URLs and HTTP (1)

`http://www.foo.com:80/foo/bar?quux#q1`

lookup IP address of www.foo.com

connect via TCP to port 80:

```
GET /foo/bar?quux HTTP/1.1
Host: www.foo.com:80
```

URLs and HTTP (1)

`http://www.foo.com:80/foo/bar?quux#q1`

lookup IP address of www.foo.com

connect via TCP to port 80:

```
GET /foo/bar?quux HTTP/1.1
Host: www.foo.com:80
```

URLs and HTTP (1)

`http://www.foo.com:80/foo/bar?quux#q1`

lookup IP address of www.foo.com

connect via TCP to port 80:

```
GET /foo/bar?quux HTTP/1.1
Host: www.foo.com:80
```

exercise: why include the Host there?

autoconfiguration

problem: how does my machine get IP address

otherwise:

- have sysadmin type one in?
- just choose one?
- ask machine on local network to assign it

autoconfiguration

problem: how does my machine get IP address

otherwise:

- have sysadmin type one in?
- just choose one?
- ask machine on local network to assign it

autoconfiguration

problem: how does my machine get IP address

otherwise:

- have sysadmin type one in?
- just choose one?
- ask machine on local network to assign it

often local router machine runs service to assign IP addresses

- knows what IP addresses are available
- sysadmin might configure in mapping from MAC addresses to IP addresses

DHCP high-level

protocol done over UDP

but since we don't have IP address yet, use `0.0.0.0`

and since we don't know server address, use `255.255.255.255`
= "everyone on the local network"

local server replies to request with address + time limit

later: can send messages to local server to renew/give up address

DHCP high-level

protocol done over UDP

but since we don't have IP address yet, use `0.0.0.0`

and since we don't know server address, use `255.255.255.255`
= "everyone on the local network"

local server replies to request with address + time limit

later: can send messages to local server to renew/give up address

exercise: why time limit?

DHCP “lease”

rather than getting address forever

but DHCP has way of releasing taken address

why impose a time limit

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)

implementing NAT

remote host + port	outside local port number	inside IP	inside port number
128.148.17.3:443	54033	192.168.1.5	43222
11.7.17.3:443	53037	192.168.1.5	33212
128.148.31.2:22	54032	192.168.1.37	43010
128.148.17.3:443	63039	192.168.1.37	32132

table of the translations

need to update as new connections made

NAT and layers

previously: network layer responsible for get to right machine

now: network + transport layer

- because we use port numbers

also, NAT needs to know about connections (transport layer)

- to know how to setup/remove table entries

backup slides

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

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,
     // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_protocol
     // ai_protocol = 0 (default) or ...
);
if (sock_fd < 0) {
if (connect(sock_fd, server->ai_addr, server->ai_addrlen) < 0) {
    /* 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_fd;
struct addrinfo *server; /* pointer to linked list */

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);
```

ai_addr points to struct representing address
type of struct depends whether IPv6 or IPv4

connection setup: client, using addrinfo

```
int sock_fd;
st
so
    // 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);
```

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

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 = 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) */
```

NB: pass pointer *to pointer* to addrinfo to fill in

connection setup: lookup address

```
/* example hostname, portname = "www.cs.virginia.edu", "443" */
const
...
struct
struct
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) */
```

AF_UNSPEC: choose between IPv4 and IPv6 for me

AF_INET, AF_INET6: choose IPv4 or IPV6 respectively

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.ai_family = AF_UNSPEC; /* I don't care */
hints.ai_flags =

rv = getaddrinfo                                    rver);
if (rv != 0) { /
```

hostname could also 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; /* I don't care */
hints.ai_flags

rv = getaddrin                                    er);
if (rv != 0) {
```

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

connection setup: server, address setup

```
/* example (hostname, portname) = ("127.0.0.1", "443") */
const char *ho
...
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);
```

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 (sock_fd < 0) { /* handle error */ }
st
ad
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

```
int sock_fd;

server = /* coo
sock_fd = socke
    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);
```

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

connection setup: client — manual addresses

```
int sock_fd;

server = /
sock_fd =
    AF_INE
    SOCK_S
    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

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 */
}
lister
...
int so
```

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 */
}
lis
...
int
```

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(serv
...
int socket_fd = accept(server_socket_fd, NULL);
```

choose the number of unaccepted connections

writing files?

```
write(file, "H", 1);
write(file, "i", 1);
write(file, "\n", 1);
```

```
write(file, "Hi\n", 3);
```

---

with files/the terminal: both do the same thing

- can read back result in same way

also: don't need to worry about data being lost/reordered

stream sockets: same kind of interface

alternative: datagram

alternative: datagram sockets

send "datagrams"

- individual messages
- if too long — too bad
- can be lost/corrupted/etc.

interface for using UDP