

last time (1)

general pattern for monitors

- condition variable = list of waiting threads

- always lock before accessing shared data

- while (something) wait

- if (changed something) broadcast/signal

- optimally separate condition variable for each something

network layers

- physical (send bits)

- link (machines sharing network segment [wire, radio, etc.])

- network (machines between networks)

- transport (mailbox → connection model; reliability, etc.)

- application

last time (2)

transport layer: reliable connections atop “best-effort”

- acknowledgments + timeouts

- acknowledgment lost looks like message never received

layers

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

layers

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

network limitations/failures

messages lost

messages delayed/reordered

messages limited in size

messages corrupted

network limitations/failures

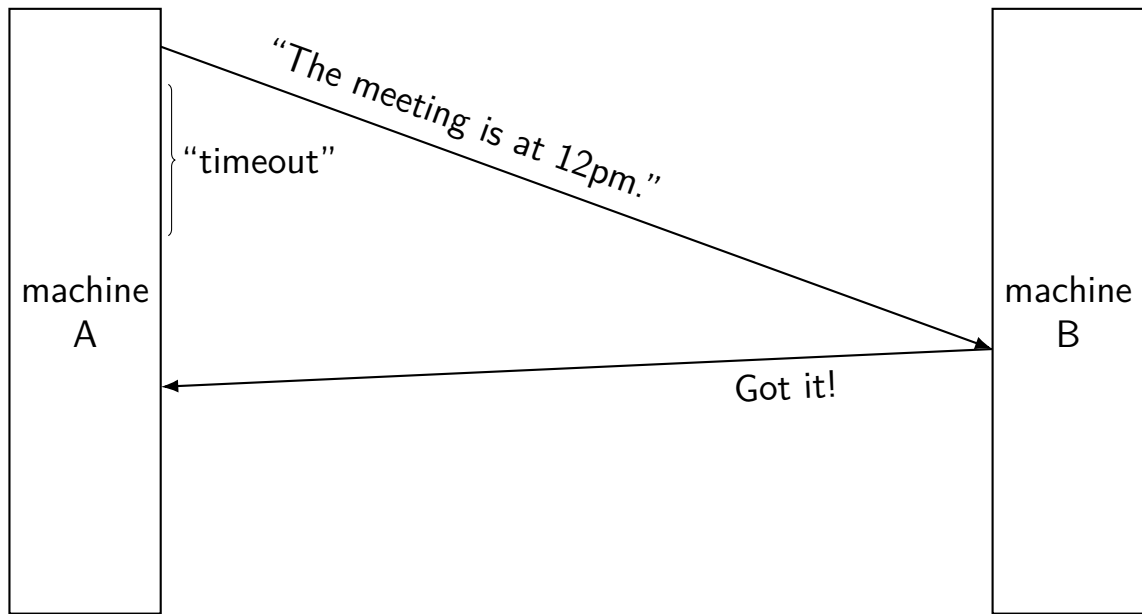
messages lost

messages delayed/reordered

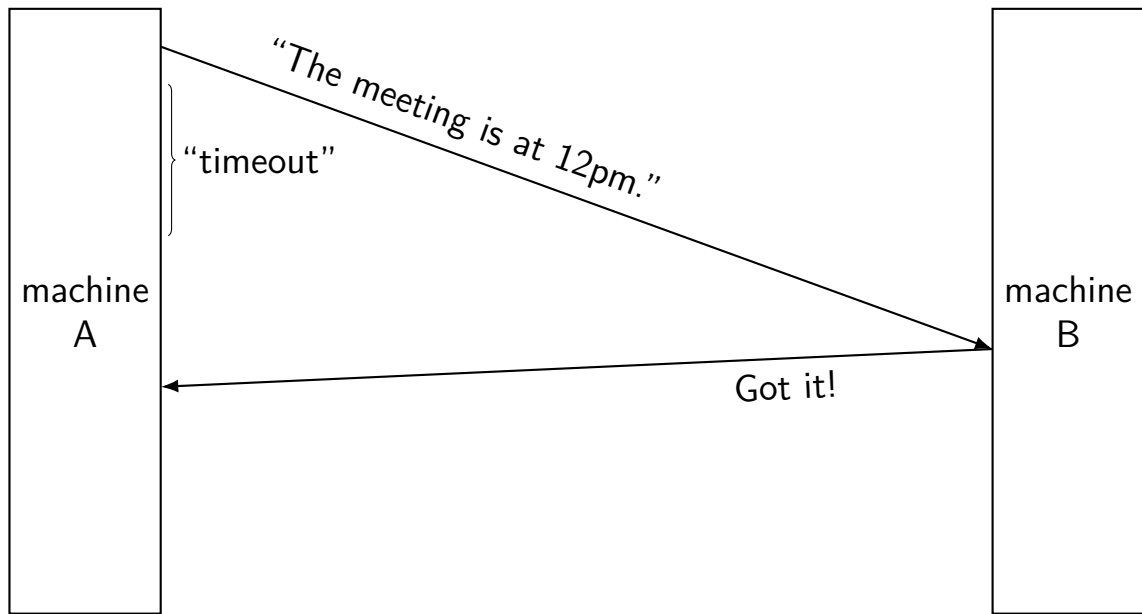
messages limited in size

messages corrupted

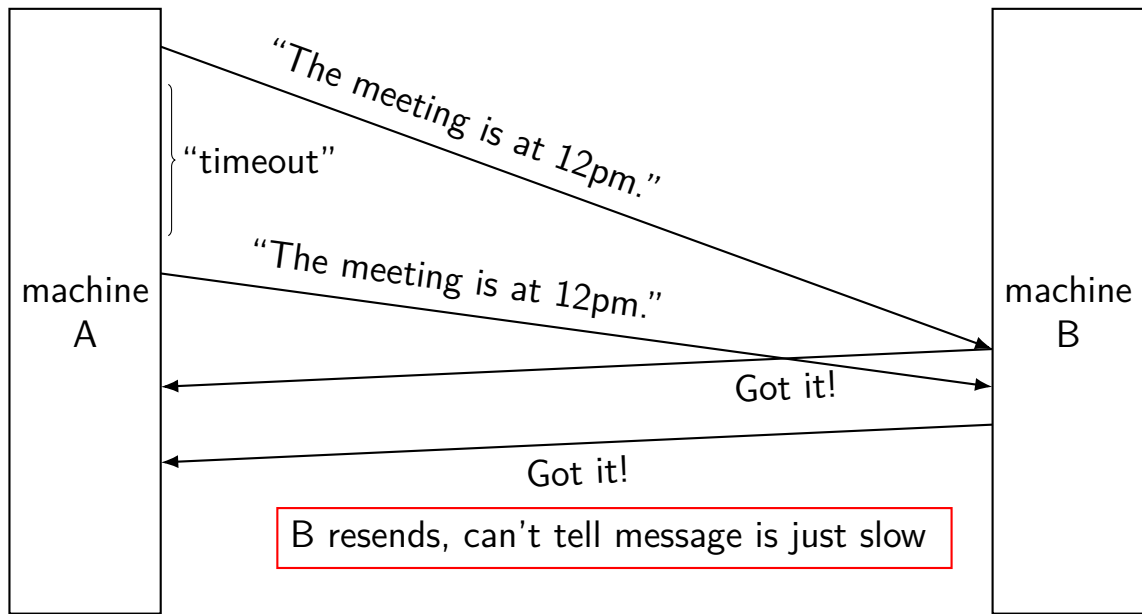
delayed message



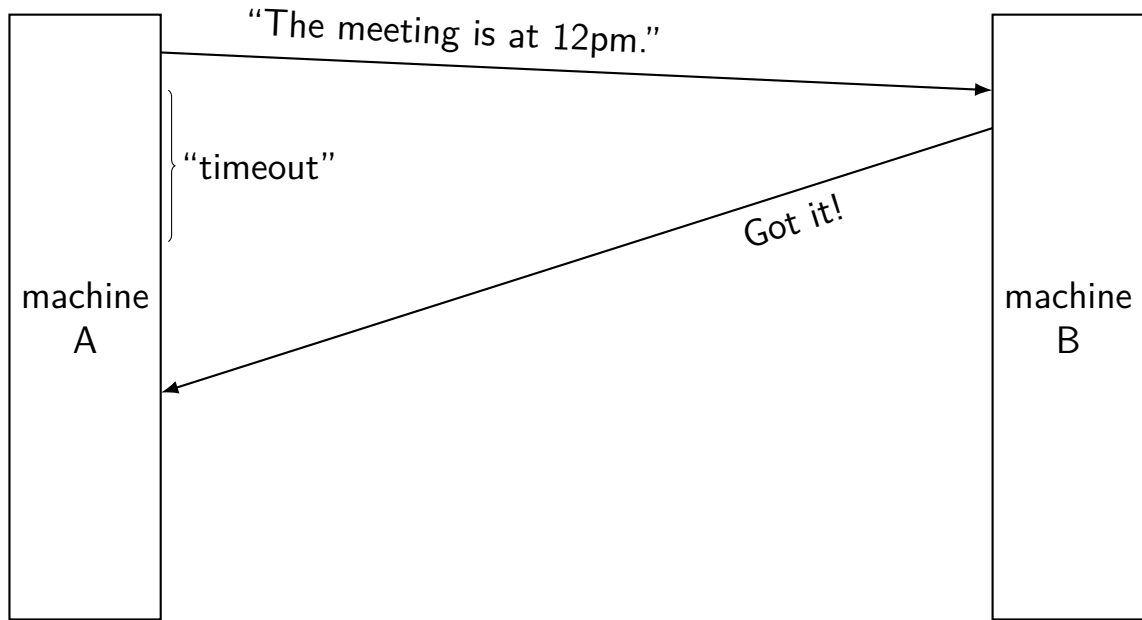
delayed message



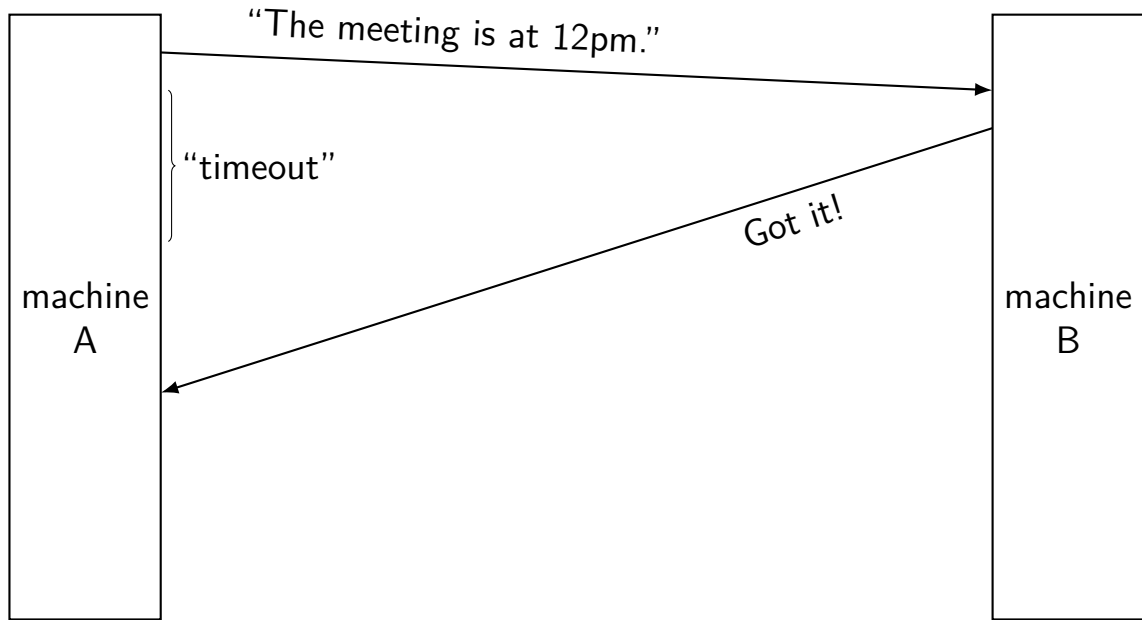
delayed message



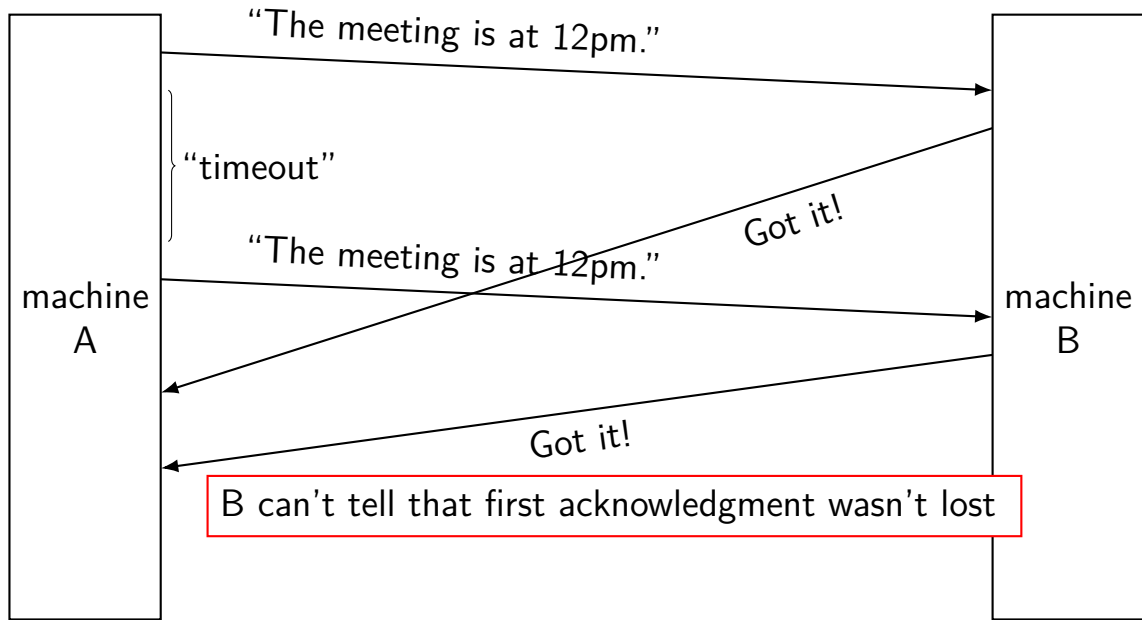
delayed acknowledgements



delayed acknowledgements



delayed acknowledgements



network limitations/failures

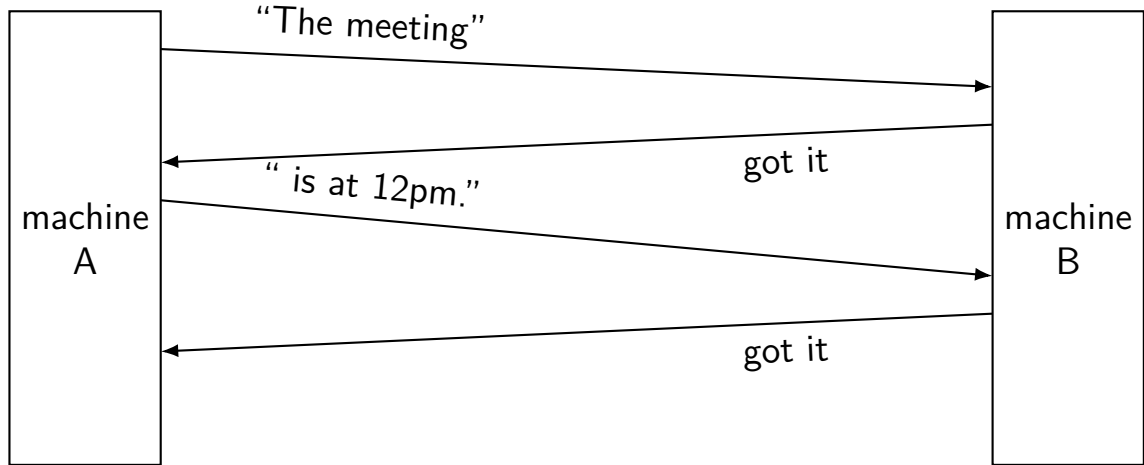
messages lost

messages delayed/reordered

messages limited in size

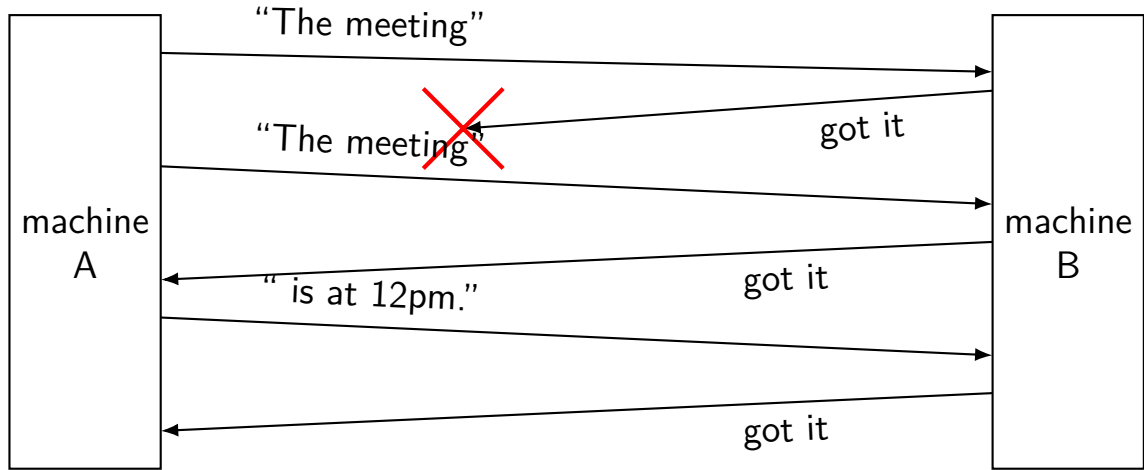
messages corrupted

splitting messages: try 1

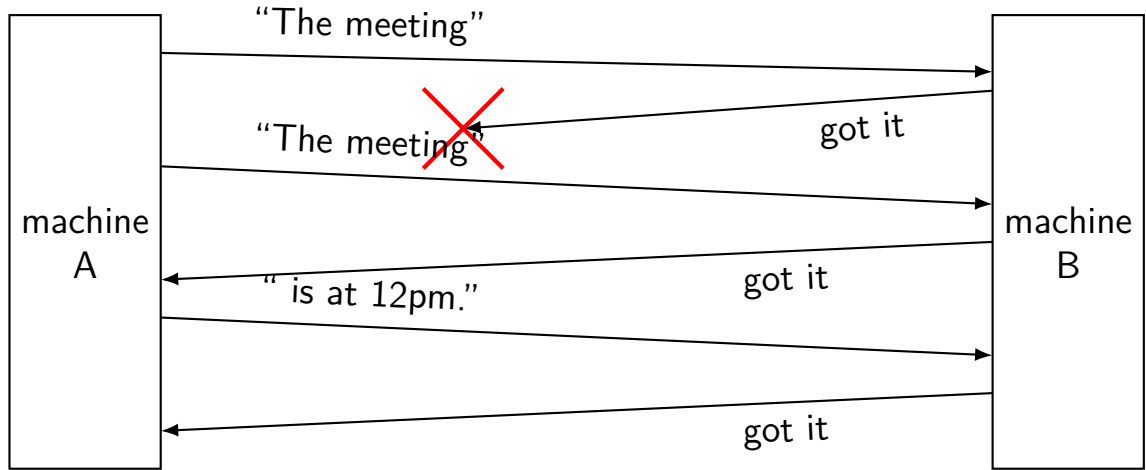


reconstructed message:
The meeting is at 12pm.

splitting messages: try 1 — problem 1



splitting messages: try 1 — problem 1



reconstructed message:

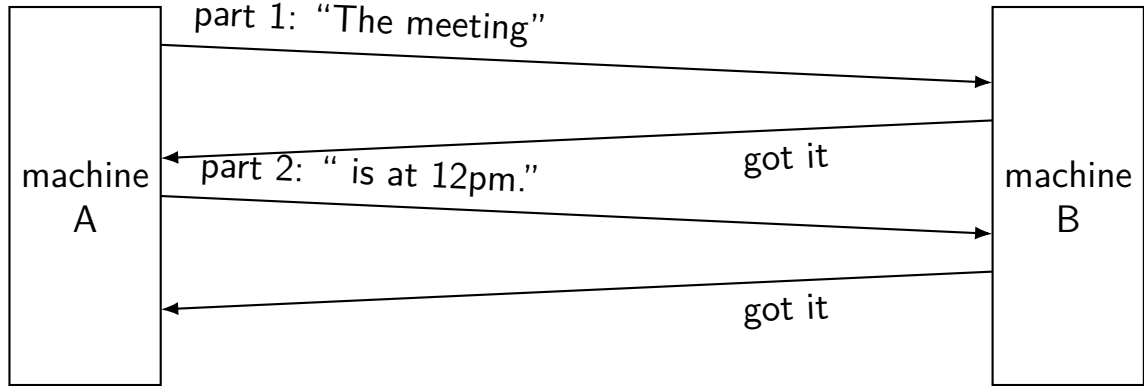
The meetingThe meeting is at 12pm.

exercise: other problems?

other scenarios where we'd also have problems?

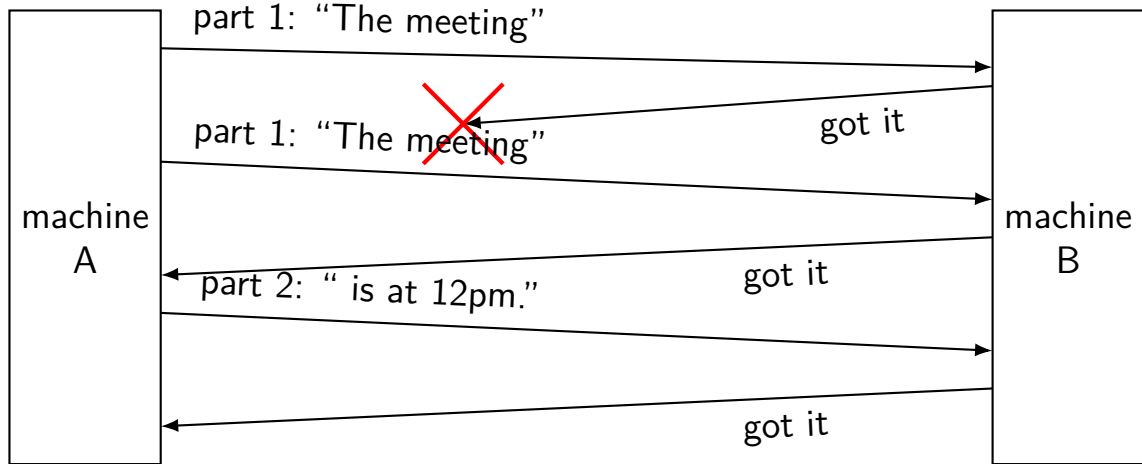
1. message (instead of acknowledgment) is lost
2. first message from machine A is delayed a long time by network
3. acknowledgment of second message lost instead of first

splitting messages: try 2



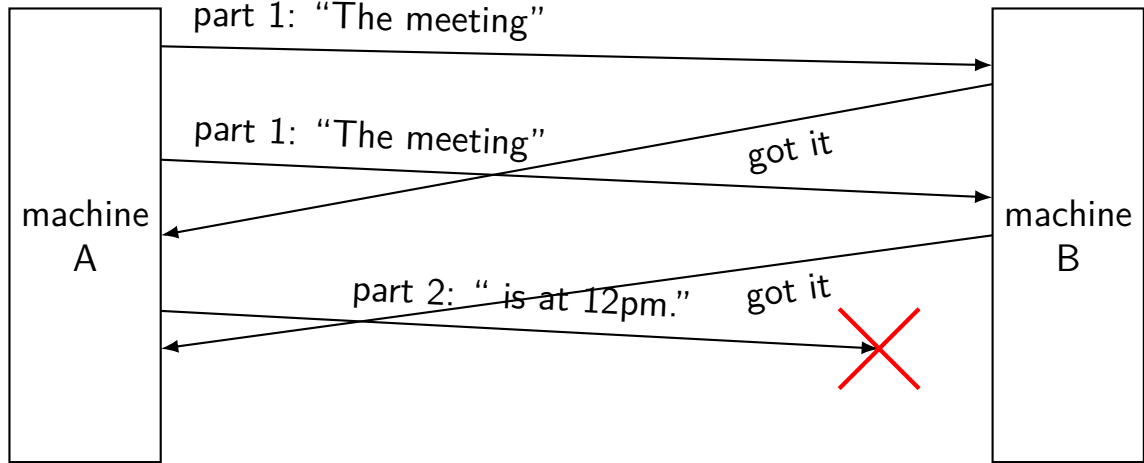
reconstructed message:
The meeting is at 12pm.

splitting messages: try 2 — missed ack



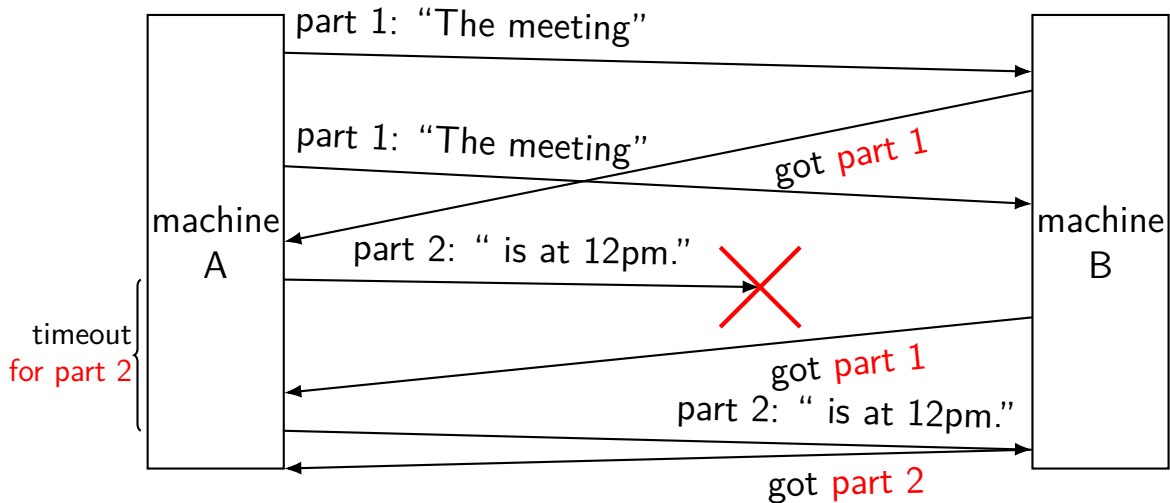
reconstructed message:
The meeting is at 12pm.

splitting messages: try 2 — problem



A thinks: part 1 + part 2 acknowledged!

splitting messages: version 3



network limitations/failures

messages lost

messages delayed/reordered

messages limited in size

messages corrupted

message corrupted

instead of sending “message”

say $\text{Hash}(\text{“message”}) = 0x\text{ABCDEF12}$

then send “0xABCDEF12,message”

when receiving, recompute hash

pretend message lost if does not match

“checksum”

these hashes commonly called “checksums”

in UDP/TCP, hash function: treat bytes of messages as array of integers; then add integers together

going faster

so far: send one message, get acknowledgments

pretty slow

instead, can send a bunch of parts and get them acknowledged together

need to do *congestion control* to avoid overloading network

upcoming lab

request + receive message split into pieces

you are responsible for:

- requesting parts in order

- resending requests if messages lost/corrupted

“acknowledge” receiving part X to request part $X+1$

upcoming lab

request + receive message split into pieces

you are responsible for:

- requesting parts in order

- resending requests if messages lost/corrupted

“acknowledge” receiving part X to request part $X+1$

protocol

GET0 — start

other end acknowledges by giving data
if they don't acknowledge, you need to send again

ACK n

request message $n + 1$ by acknowledging message n
not quite same purpose as acknowledgments in prior examples
(in lab, the response is your 'acknowledgment' of your request;
you retry if you don't get it)

callback-based programming (1)

```
/* library code you don't write */
/* in the lab: part of waitForAllTimeouts() */
void mainLoop() {
    while (true) {
        Event event = waitForAndGetNextEvent();
        if (event.type == RECIEVED) {
            recvd(...);
        } else if (event.type == TIMEOUT) {
            (event.timeout_function)(...);
        }
        ...
    }
}
```

callback-based programming (2)

```
/* your code, called by library */
void recvd(...) {
    ...
    setTimeout(..., timerCallback, ...);
}

void timerCallback(...) {
    ...
}

int main() {
    send(.../* first message */);
    ... /* other initial setup */
    waitForAllTimeouts(); // runs mainLoop()
}
```

callback-based programming

writing scripts in a webpage

many graphical user interface libraries

sometimes servers that handle lots of connections

layers

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

more than four layers?

sometimes more layers above 'application'

e.g. HTTPS:

HTTP (app layer) on TLS (another app layer) on TCP (network) on ...

e.g. DNS over HTTPS:

DNS (app layer) on HTTP on on TLS on TCP on ...

e.g. SFTP:

SFTP (app layer??) on SSH (another app layer) on TCP on ...

e.g. HTTP over OpenVPN:

HTTP on TCP on IP on OpenVPN on UDP on different IP on ...

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

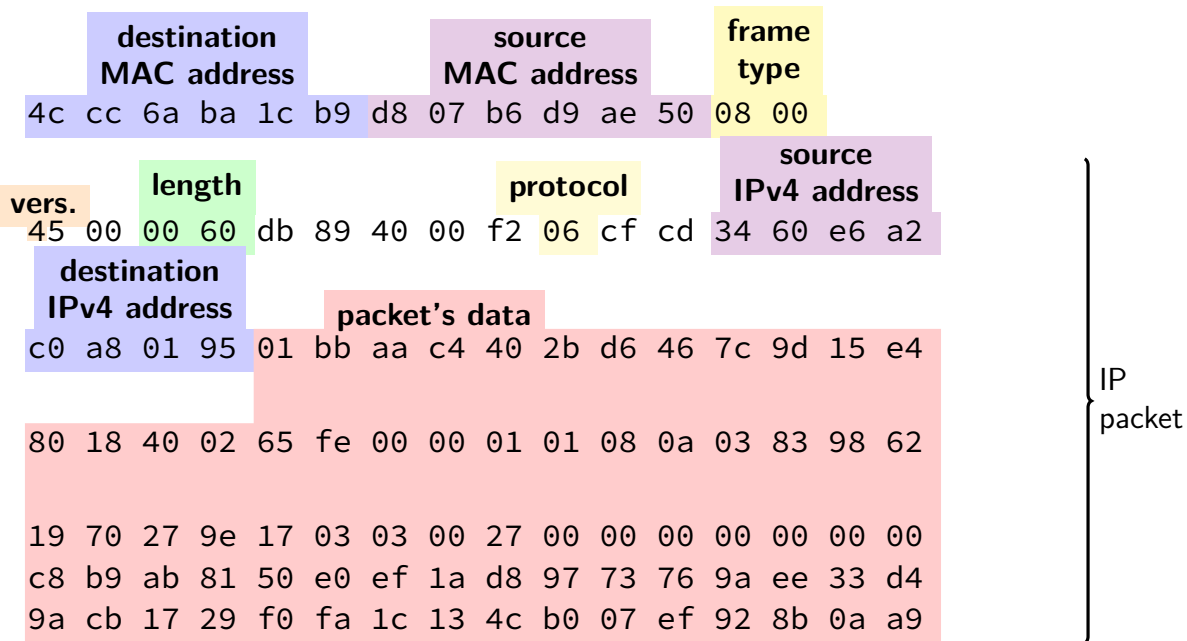
application	HTTP, SSH, SMTP, ...	application-defined meanings
transport	TCP, UDP, ...	reach correct program, reliability/streams
network	IPv4, IPv6, ...	reach correct machine (across networks)
link	Ethernet, Wi-Fi, ...	coordinate shared wire/radio
physical	...	encode bits for wire/radio

an Ethernet frame

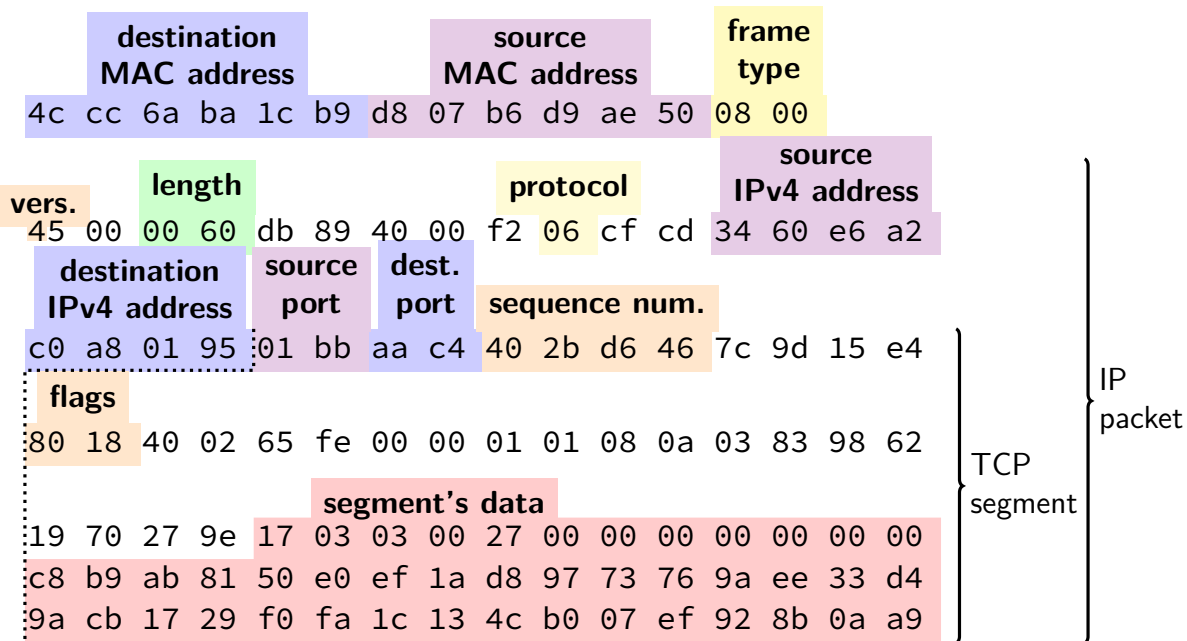
destination MAC address						source MAC address						frame type	
4c	cc	6a	ba	1c	b9	d8	07	b6	d9	ae	50	08	00

frame's data															
45	00	00	60	db	89	40	00	f2	06	cf	cd	34	60	e6	a2
c0	a8	01	95	01	bb	aa	c4	40	2b	d6	46	7c	9d	15	e4
80	18	40	02	65	fe	00	00	01	01	08	0a	03	83	98	62
19	70	27	9e	17	03	03	00	27	00	00	00	00	00	00	00
c8	b9	ab	81	50	e0	ef	1a	d8	97	73	76	9a	ee	33	d4
9a	cb	17	29	f0	fa	1c	13	4c	b0	07	ef	92	8b	0a	a9

an Ethernet frame



an Ethernet frame



the link layer

Ethernet, Wi-Fi, Bluetooth, DOCSIS (cable modems), ...

allows send/recv messages to machines on “same” network segment

- typically: wireless range+channel or connected to a single switch/router
- could be larger (if *bridging* multiple network segments)
- could be smaller (switch/router uses “virtual LANs”)

typically: source+destination specified with MAC addresses

- MAC = media access control

- usually manufacturer assigned / hard-coded into device
- unique address per port/wifi transmitter/etc.

can specify destination of “anyone” (called *broadcast*)

messages usually called “frames”

the link layer

Ethernet, Wi-Fi, Bluetooth, DOCSIS (cable modems), ...

allows send/recv messages to machines on “same” network segment

- typically: wireless range+channel or connected to a single switch/router
- could be larger (if *bridging* multiple network segments)
- could be smaller (switch/router uses “virtual LANs”)

typically: source+destination specified with MAC addresses

- MAC = media access control

- usually manufacturer assigned / hard-coded into device

- unique address per port/wifi transmitter/etc.

can specify destination of “anyone” (called *broadcast*)

messages usually called “frames”

link layer jobs

divide raw bits into messages

identify who message is for on shared radio/wire

handle if two+ machines use radio/wire at same time

drop/resend messages if corruption detected

resending more common in radio schemes (wifi, etc.)

link layer reliability?

Ethernet + Wifi have checksums

Q1: Why doesn't this give us uncorrupted messages?

Why do we still have checksums at the higher layers?

Q2: What's a benefit of doing this if we're also doing it in the higher layer?

layers

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

the network layer

the Internet Protocol (IP) version 4 or version 6

there are also others, but quite uncommon today

allows send messages to/recv messages from other networks

“internetwork”

messages usually called “packets”

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

some IPs reserved for non-Internet use (127.*, 10.*, 192.168.*)

IPv6 addresses

IPv6 like IPv4, but with 128-bit numbers

written in hex, 16-bit parts, separated 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

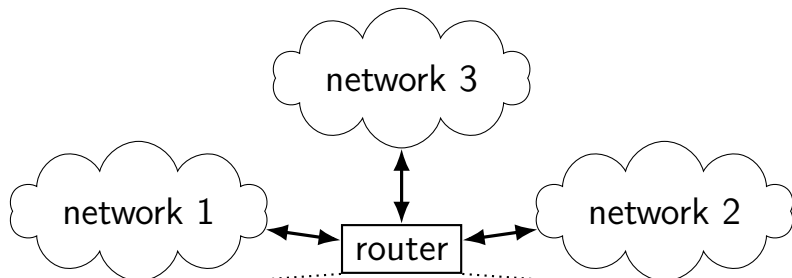
2607f8b0400d0c0000000000000000006a_{SIXTEEN}

selected special IPv6 addresses

`::1` = localhost

anything starting with `fe80` = link-local addresses
never forwarded by routers

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

layers

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

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

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 reliability/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)

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 reliability/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)

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 reliability/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)

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	ESTABLISH
tcp	0	0	128.143.67.91:803	128.143.67.236:2049	ESTABLISH
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	ESTABLISH
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	ESTABLISH
tcp	0	0	127.0.0.1:631	127.0.0.1:50438	ESTABLISH

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

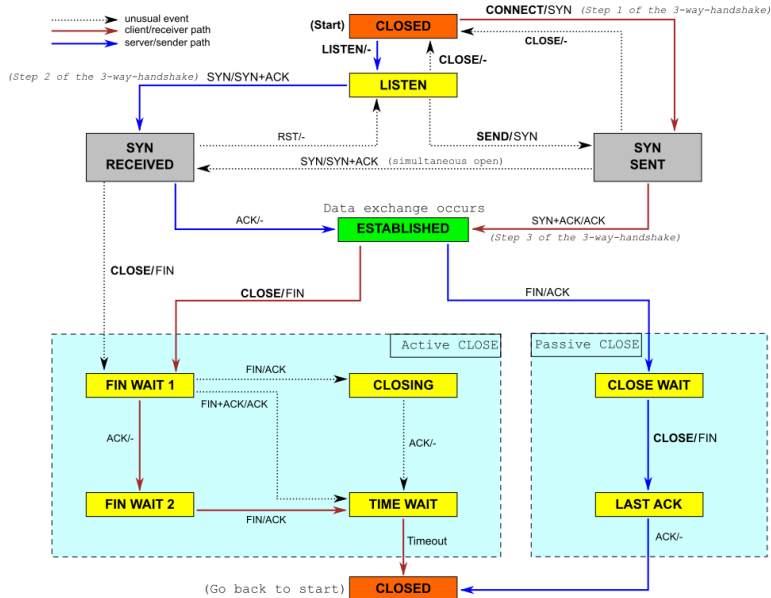
remember delayed messages?

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

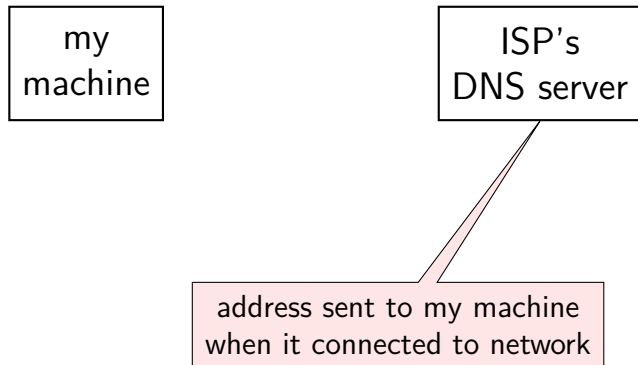
TCP state machine picture



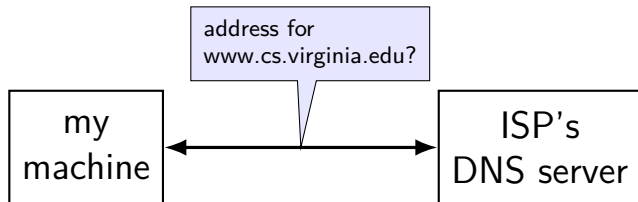
names and addresses

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

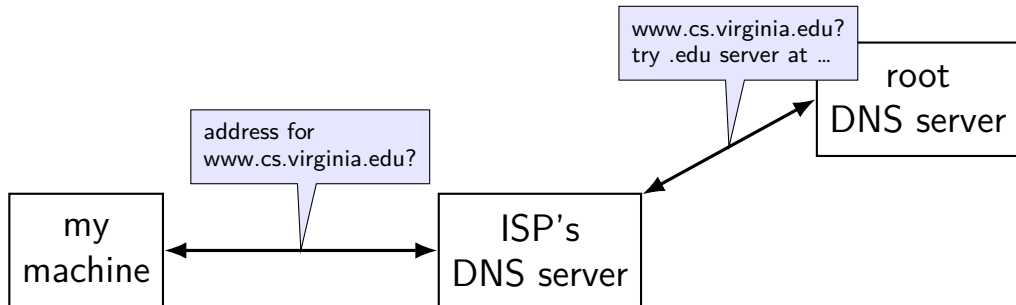
DNS: distributed database



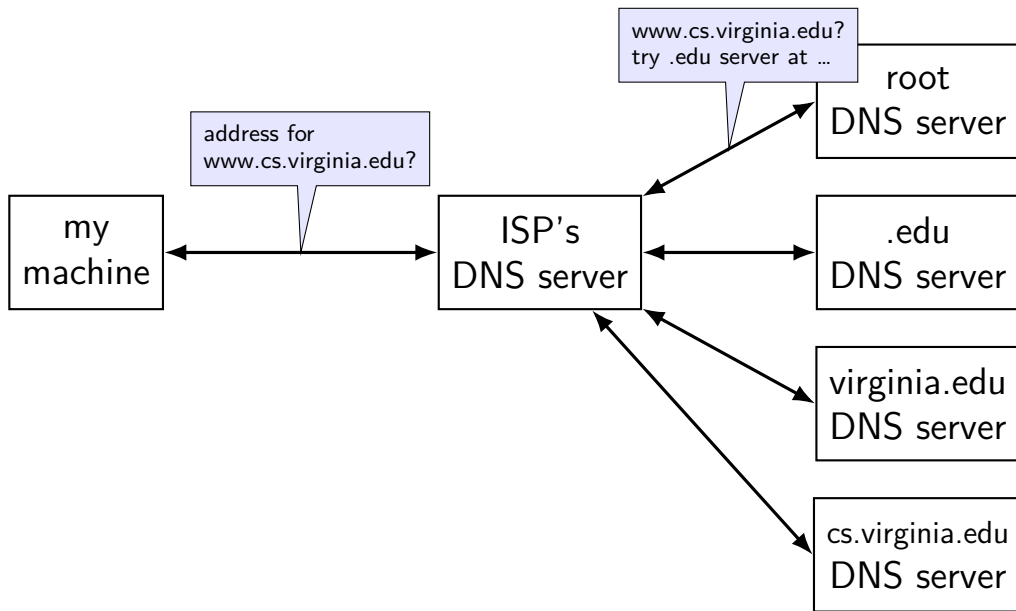
DNS: distributed database



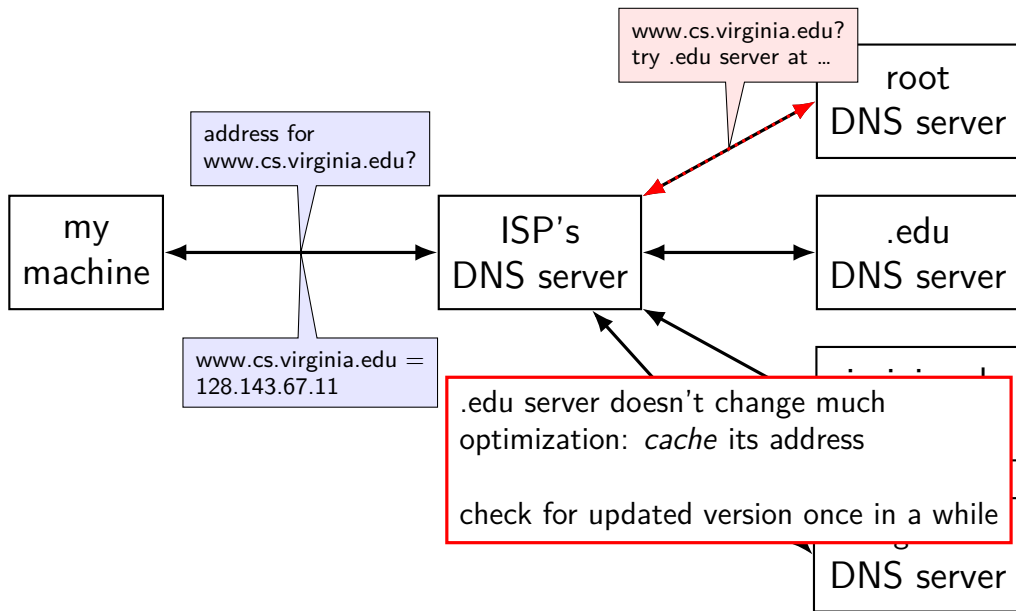
DNS: distributed database



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

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 addresses: 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
...
```

network address to link-layer address + interface

only tracks things directly connected to my local network

how is that table made?

ask all 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

Uniform Resource 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/

authority = 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

firewalls

don't want to expose network service to everyone?

solutions:

- service picky about who it accepts connections from
- filters in OS on machine with services
- filters on router

later two called “firewalls”

firewall rules examples?

ALLOW tcp port 443 (https) FROM everyone

ALLOW tcp port 22 (ssh) FROM my desktop's IP address

BLOCK tcp port 22 (ssh) FROM everyone else

ALLOW from address X to address Y

...

spoofing

if I only allow connections from my desktop's IP addresses,
how would you attack this?

hint: how do we know what address messages come from?

backup slides

link layer quality of service

if frame gets...

event	on Ethernet	on WiFi
collides with another	detected + may resend	resend
not received	lose silently	resent
header corrupted	usually discard silently	usually resend
data corrupted	usually discard silently	usually resend
too long	not allowed to send	not allowed to send
reordered (v. other messages)	received out of order	received out of order
destination unknown	lose silently	usually resend??
too much being sent	discard excess?	discard excess?

network layer quality of service

if packet ...

event

on IPv4/v6

collides with another

out of scope — handled by link layer

not received

lost silently

header corrupted

usually discarded silently

data corrupted

received corrupted

too long

dropped with notice or “fragmented” + recombined

reordered (v. other messages)

received out of order

destination unknown

usually dropped with notice

too much being sent

discard excess

network layer quality of service

if packet ...

event

on IPv4/v6

collides with another

out of scope — handled by link layer

not received

lost silently

header corrupted

usually discarded silently

data corrupted

received corrupted

too long

dropped with notice or “fragmented” + recombined

reordered (v. other messages)

received out of order

destination unknown

usually dropped with notice

too much being sent

discard excess

includes dropped by link layer
(e.g. if detected corrupted there)

'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));
...
```

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.sin_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
/* or: addr.sin_addr.in_addr = htonl(...); */
addr.sin_port = htons(9999); /* port number 9999 */
```

```
if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
```

```
listen
```

INADDR_ANY: accept connections for any address I can!

alternative: specify specific address

```
int s
```

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.sin_addr.in_addr = INADDR_LOOPBACK (127.0.0.1) */
/* or: addr.sin_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, 10);
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(server_socket_fd, 10); /* choose the number of unaccepted connections */
...
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 */
    // ...
}
// specify IPv4 instead of IPv6 or local-only sockets
// specify TCP (byte-oriented) instead of UDP ('datagram' oriented)
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 */
sock_fd = socket(htonl/s = host-to-network long/short
                 AF_INET, /* network byte order = big endian */
                 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 = / struct representing IPv4 address + port number  
sock_fd = declared in <netinet/in.h>  
          AF_INET see man 7 ip on Linux for docs  
          SOCK_STREAM  
          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);
```

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, MAX_SIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);
        if (read_count != write_count) {...error?...}
    }
}
```

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, MAX_SIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);
        if (read_count != write_count) {...error?...}
    }
}
```

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, MAX_SIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);
        if (read_count != write_count) {...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 = 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 = AI_PASSIVE; /* hostname could also be NULL
                               means "use all possible addresses"
                               only makes sense for servers */
rv = getaddrinfo(hostname, portname, &hints, &server);
if (rv != 0) {
```

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 = 0;

rv = getaddrinfo(hostname, portname, &hints, NULL);
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 *hostname = "127.0.0.1";
...
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, addrinfo

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

int server_socket_fd = socket(
    server->ai_family,
    server->ai_socktype,
    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, 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 = 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,  
    // addrinfo contains all information needed to setup socket  
    // set by getaddrinfo function (next slide)  
);  
if (sock_fd < 0) {  
    if (errno == EAFNOSUPPORT) {  
        /* handles IPv4 and IPv6 */  
        /* handles DNS names, service names */  
    }  
    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 = 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-addrinfo;  
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 = 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
```

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

```
    // ai_family = AF_INET (IPv4) or AF_INET6 (IPv6) or ...
    server->ai_socktype,
    // ai_socktype = SOCK_STREAM (bytes) or ...
    server->ai_protocol
    // 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 = AF_UNSPEC; /* for IPv4 OR IPv6 */
// hints.ai_socktype = AF_INET; /* for TCP or UDP */
NB: pass pointer to pointer to addrinfo to fill in
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
...
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) */
```

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, current->ai_protocol);
    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, current->ai_protocol);
    if (sock_fd < 0) continue;
    if (connect(sock_fd, current->ai_addr, current->ai_addrlen) == 0)
        break;
}
close(sock_fd);
}
```

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));
...
```


aside: on server port numbers

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

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

so, for testing: probably ports > 1023