# last time (1)

consistency via careful ordering

- avoid writing pointers to bad data
- can scan entire filesystem for allocated but unused stuff

consistency via redo logging

- write intended operations to log before performing them
- write whether committed or not — uncommitted means nothing done
- on failure, redo operations in log if *committed*"

# last time (2)

handle data loss via redundancy

- mirroring — just make two copies
- *erasure coding* — store extra data that allows recovery if $K$ of $N$ parts lost

multiple versions via copy-on-write snapshots

- filesystem maintains array of versions
- different versions use one copy of common data
- modify one version: copy+modify parts that are changed
- extra indirection to minimize what's needs copying (e.g. split inode array)

# snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around
- accidental deletion? old version stil there
- eventually discard some old versions

can access *snapshot* of files at prior time

# snapshots

filesystem snapshots

idea: filesystem keeps old versions of files around

- accidental deletion? old version stil there
- eventually discard some old versions

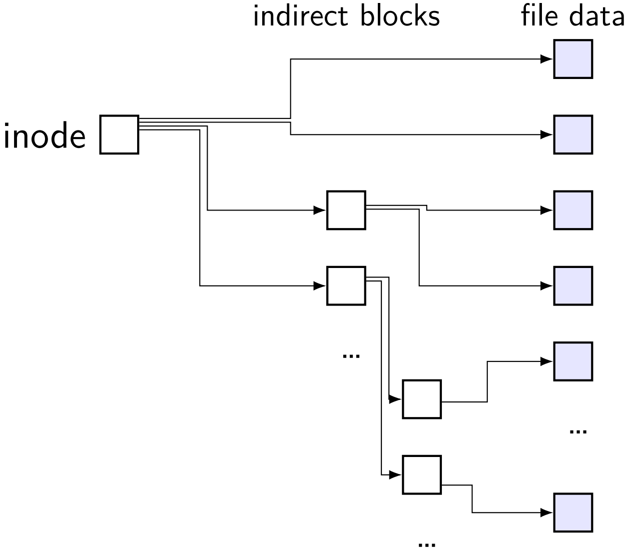
can access *snapshot* of files at prior time

mechanism: copy-on-write

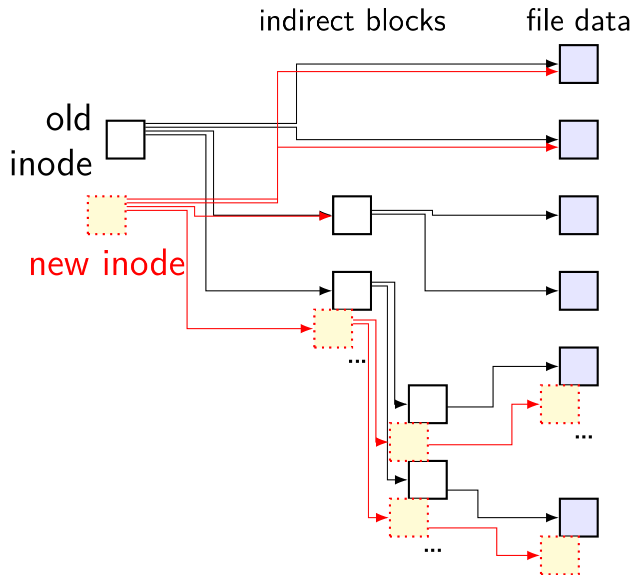
changing file makes new copy of filesystem

common parts shared between versions

# inode and copy-on-write



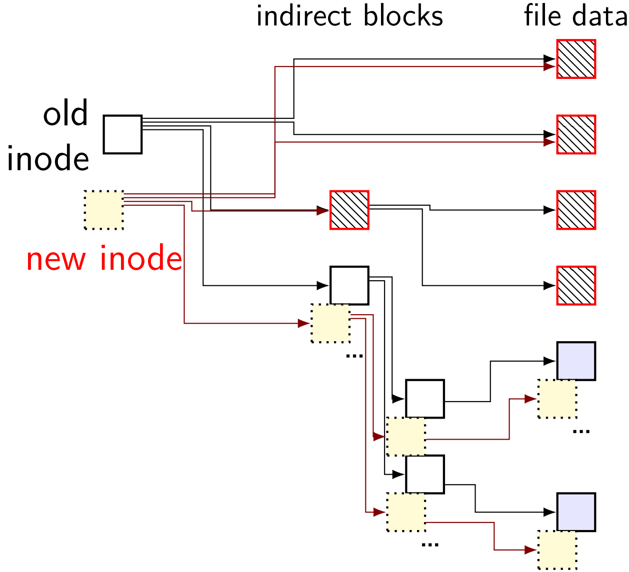
# inode and copy-on-write




update: new data blocks
+ new indirect blocks
+ new inode

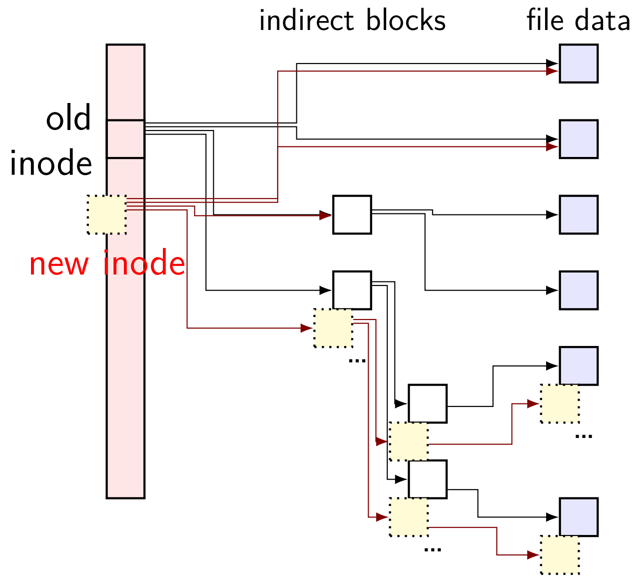
both old+new inode valid

# inode and copy-on-write




unchanged parts of file shared

# inode and copy-on-write

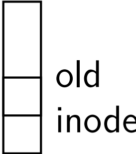



challenge: FFS/xv6/ext2 design has big array of inodes

don't want to write new copy of *entire inode array*

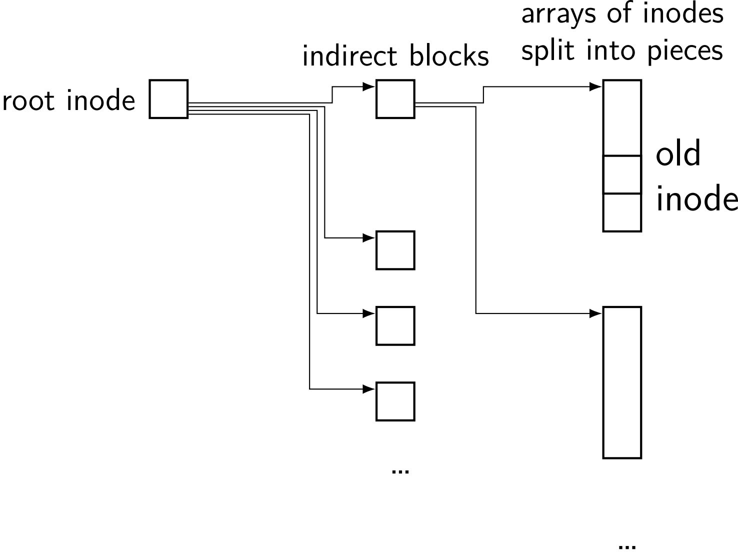
# extra indirection for inode array

arrays of inodes
split into pieces

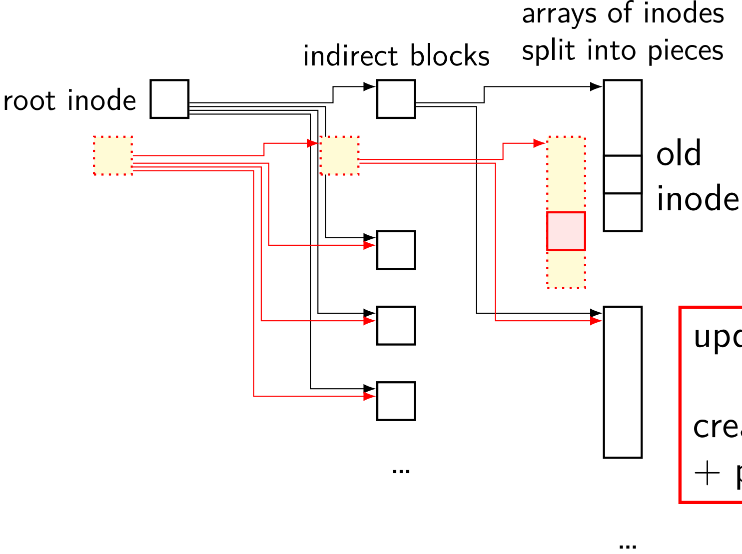


...

# extra indirection for inode array



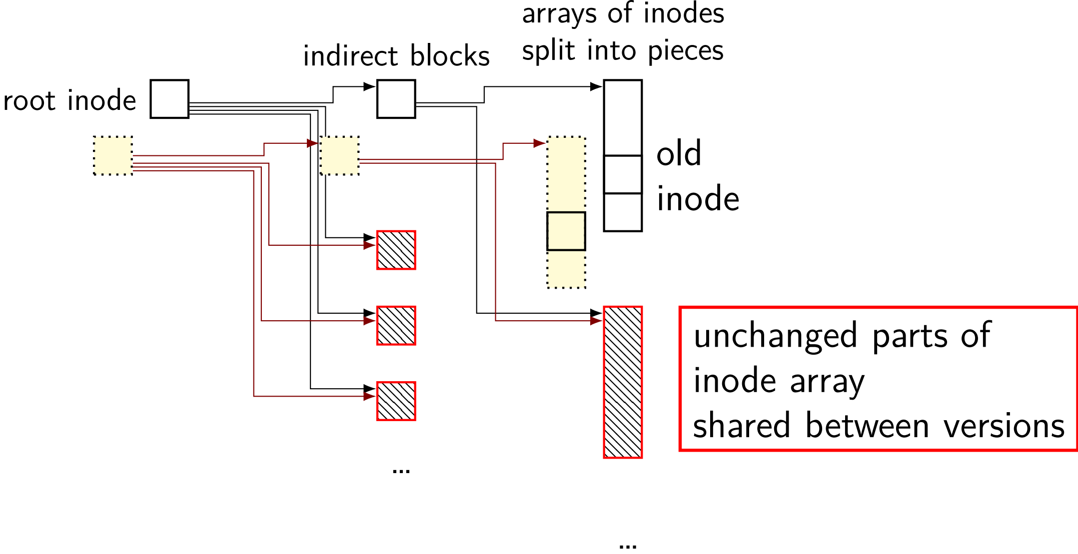
# extra indirection for inode array



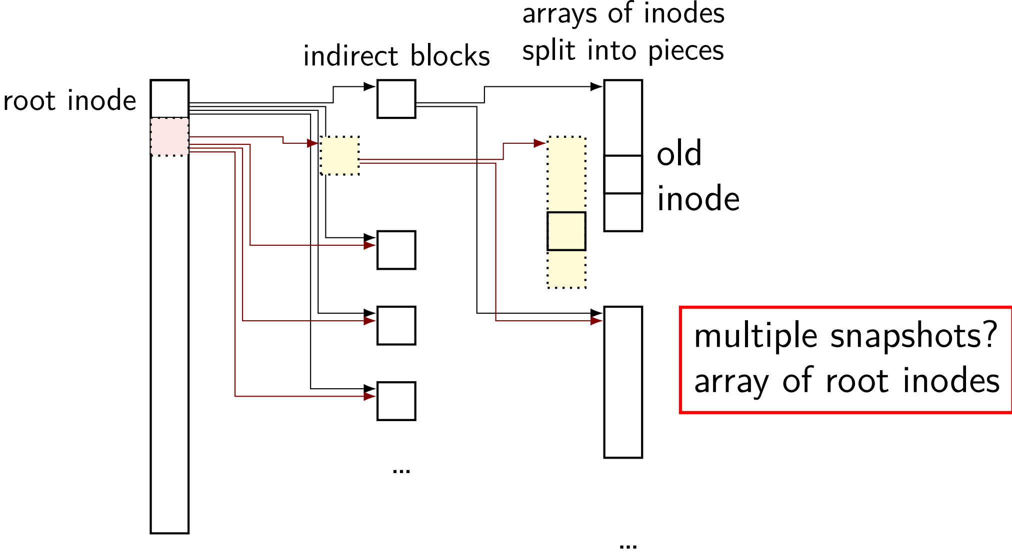

update one inode?

create new root inode
+ pointers

# extra indirection for inode array



# extra indirection for inode array




multiple snapshots?
array of root inodes

# copy-on-write indirection

file update = replace with new version

array of versions of entire filesystem

only copy modified parts

keep reference counts, like for paging assignment

lots of pointers — only change pointers where modifications happen

# snapshots in practice

ZFS supports this (if turned on)

example: `.zfs/snapshots/11.11.18-06` pseudo-directory

contains contents of files at 11 November 2018 6AM

# mounting filesystems

Unix-like system

root filesystem appears as /

other filesystems *appear as directory*

- e.g. lab machines: my home dir is in filesystem at /net/zf15

directories that are filesystems look like normal directories

- `/net/zf15/..` is `/net` (even though in different filesystems)

# mounts on a dept. machine

```
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
...
udev on /dev type devtmpfs (rw,mode=0755)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=0620)
tmpfs on /run type tmpfs (rw,noexec,nosuid,size=10%,mode=0755)
...
/dev/sda3 on /localtmp type ext4 (rw)
...
zfs1:/zf2 on /net/zf2 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                noacl,sloppy,addr=128.143.136.9)
zfs3:/zf19 on /net/zf19 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
zfs4:/sw on /net/sw type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                              noacl,sloppy,addr=128.143.136.9)
zfs3:/zf14 on /net/zf14 type nfs (rw,hard,intr,proto=udp,nfsvers=3,
                                  noacl,sloppy,addr=128.143.67.236)
...
```

# kernel FS abstractions

Linux: *virtual file system* API

object-oriented, based on FFS-style filesystem

to implement a filesystem, create object types for:

- superblock (represents "header")
- inode (represents file)
- dentry (represents cached directory entry)
- file (represents *open file*)

common code handles directory traversal

- and caches directory traversals

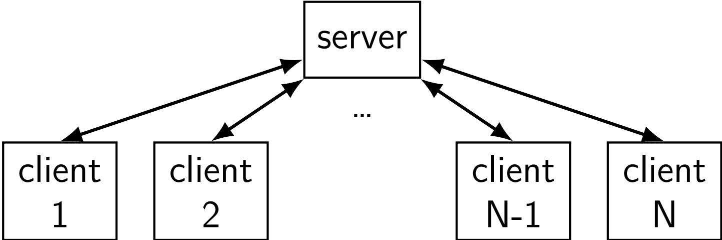
common code handles file descriptors, etc.

# distributed systems

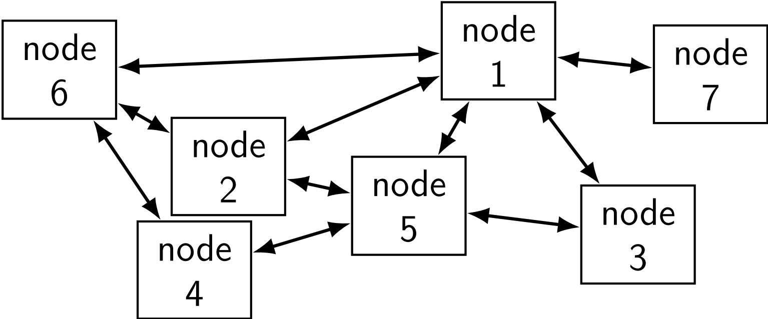
multiple machines working together to perform a single task

called a *distributed system*

# some distibuted systems models

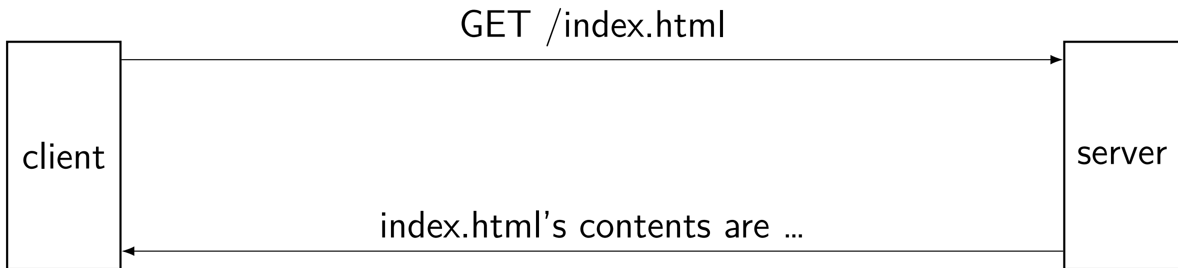



client/server

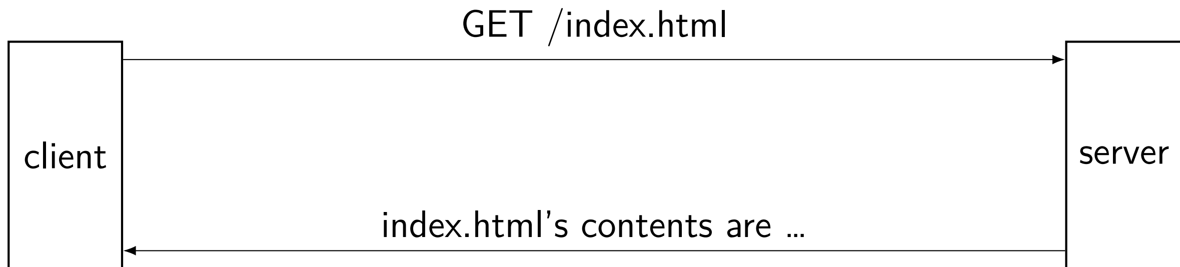



peer-to-peer

# client/server model



# client/server model

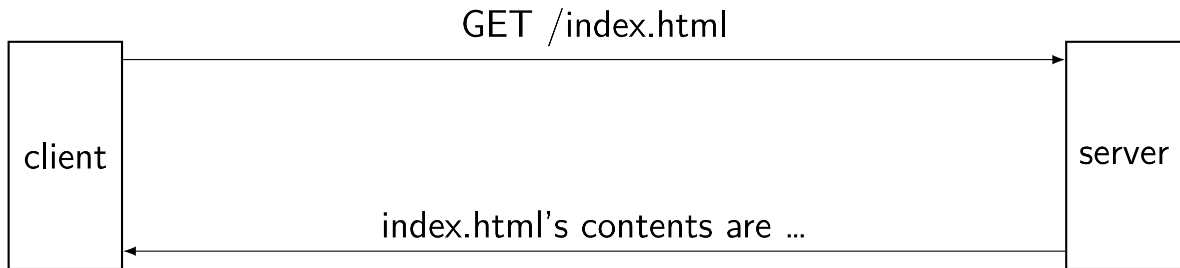



client(s): "sometimes on"

sends requests to server(s)

needs to know
how to contact server

# client/server model




client(s): "sometimes on"

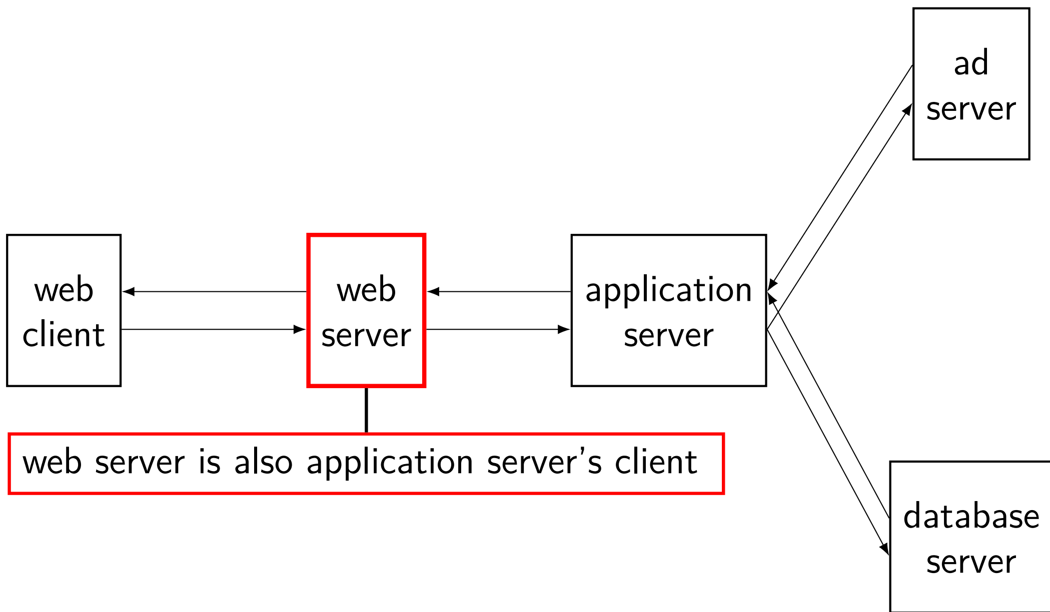
sends requests to server(s)

needs to know
how to contact server

server(s): "always on"

responds to client requests
never initiaties contact
with a client

# layers of servers?



# example: Wikipedia architecture

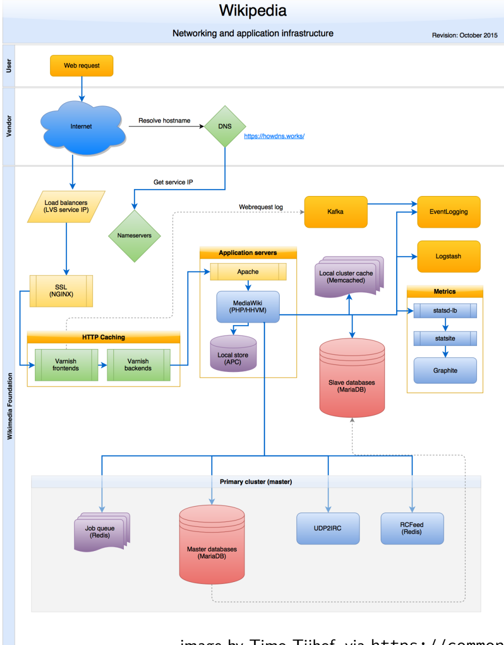



image by Timo Tijhof, via https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png

# example: Wikipedia architecture (zoom)

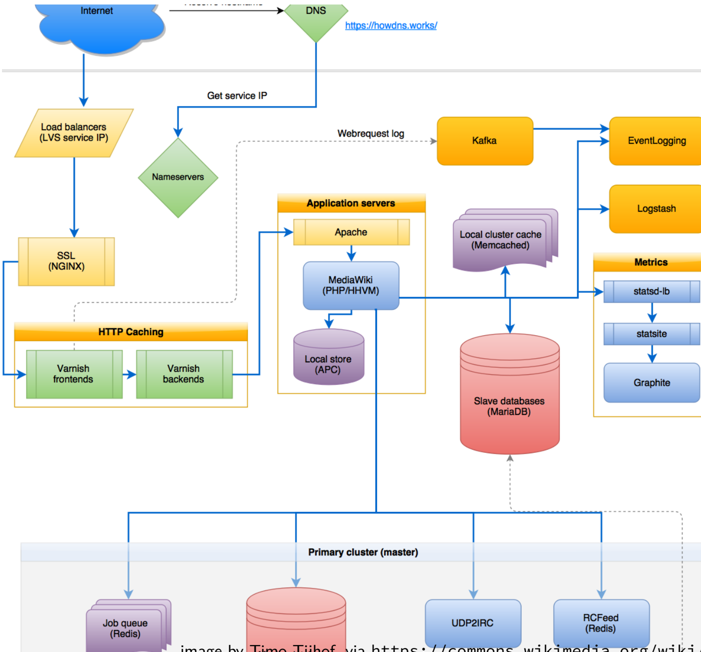



image by Timo Tijhof, via https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png

# peer-to-peer

no always-on server everyone knows about

- hopefully, no one bottleneck — “scalability”

any machine can contact any other machine

- every machine plays an approx. equal role?

set of machines may change over time

# why distributed?

multiple machine owners collaborating

delegation of responsiblity to other entity

put (part of) service “in the cloud”

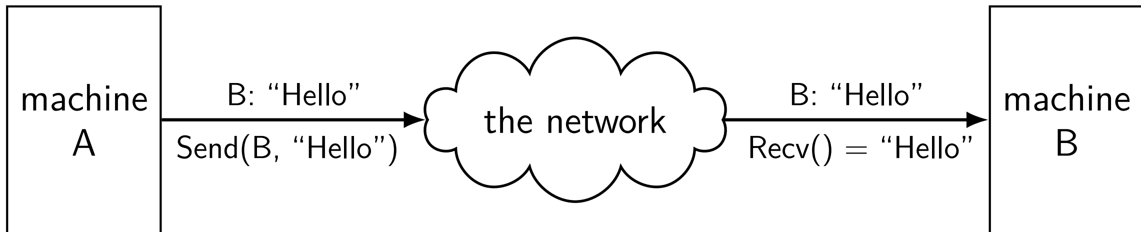
combine many cheap machines to replace expensive machine

easier to add incrementally

redundancy — one machine can fail and *system* still works?

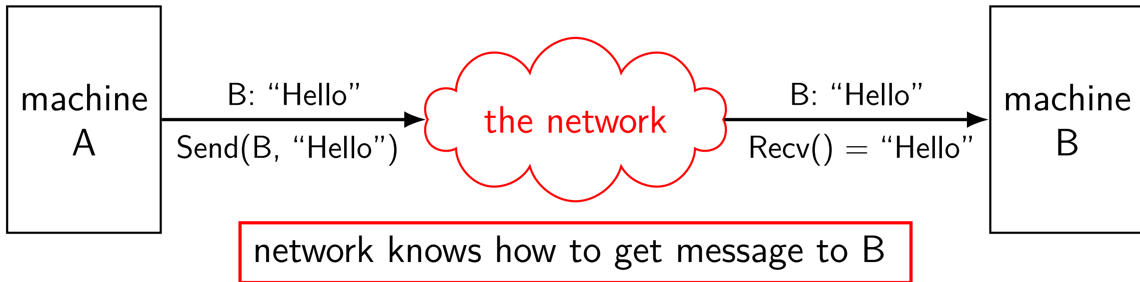
# mailbox model

*mailbox* abstraction: send/receive messages



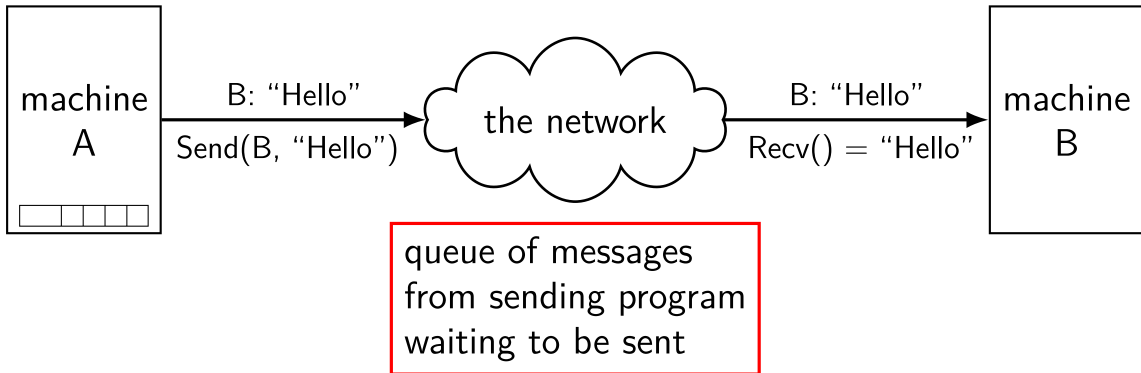
# mailbox model

*mailbox* abstraction: send/receive messages



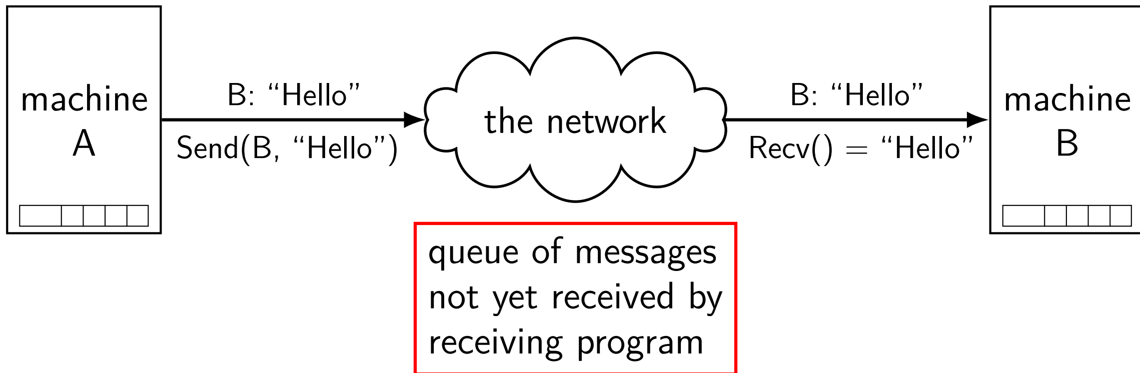
# mailbox model

*mailbox* abstraction: send/receive messages



# mailbox model

*mailbox* abstraction: send/receive messages



# what about servers?

client/server model: server wants to reply to clients

might want to send/receive multiple messages

# what about servers?

client/server model: server wants to reply to clients

might want to send/receive multiple messages

can build this with mailbox idea

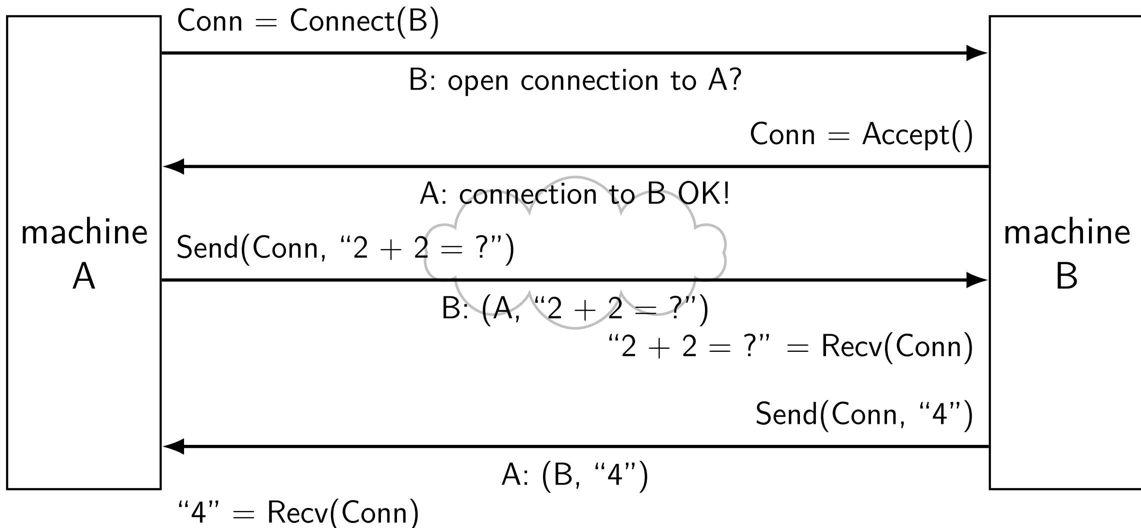
- send a 'return address'
- need to track related messages

common abstraction that does this: the connection

# extension: conections

*connections*: two-way channel for messages

extra operations: connect, accept



# connections versus pipes

connections look kinda like two-direction pipes

in fact, in POSIX will have the same API:

each end gets file descriptor representing connection

can use read() and write()

# connections over mailboxes

real Internet: mailbox-style communication

- send packets to particular mailboxes
- no gaurentee on order, when received
- no relationship between

connections implemented on top of this

full details: take networking (CS/ECE 4457)

quick summary — next slide

# connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

# names and addresses

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

# hostnames

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

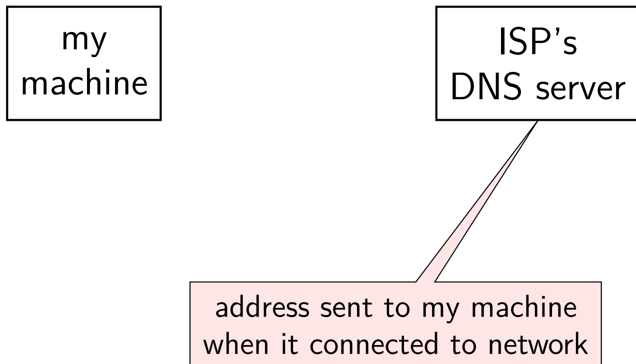
maps logical names like `www.virginia.edu`

- chosen for humans
- hierarchy of names

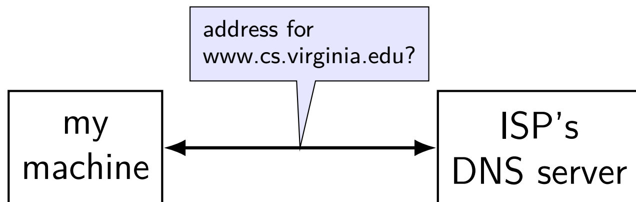
...to *addresses* the network can use to move messages

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

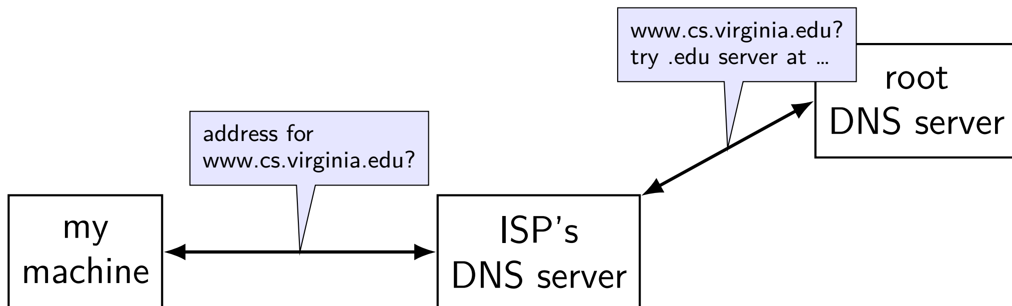
# DNS: distributed database



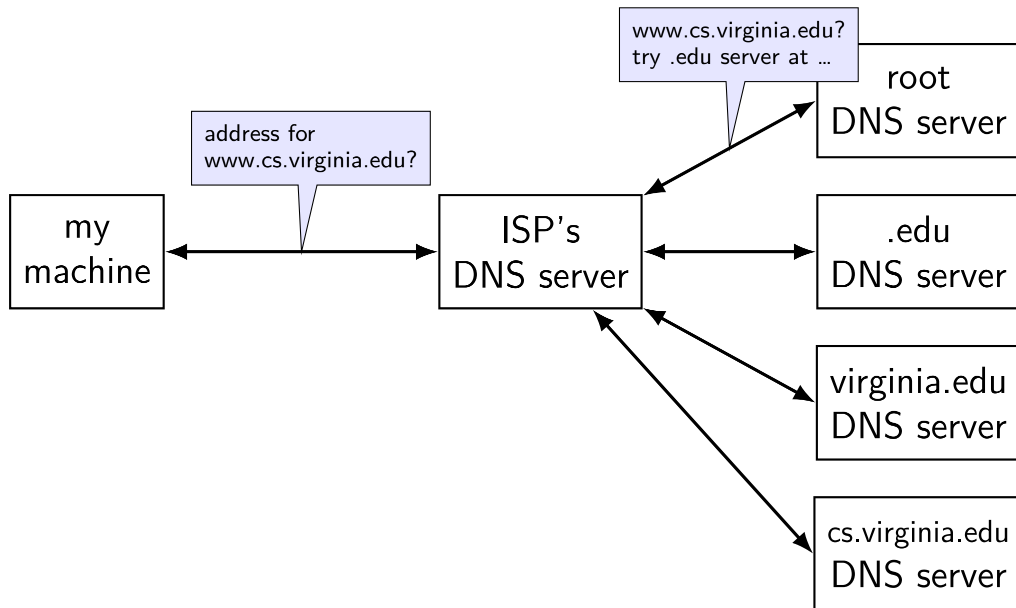
# DNS: distributed database



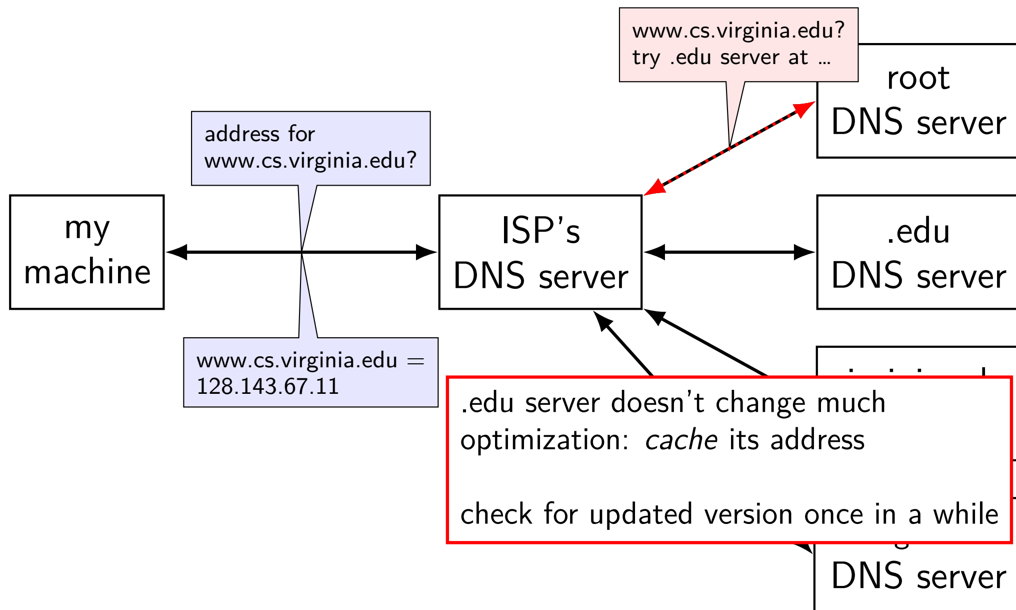
# DNS: distributed database



# DNS: distributed database



# DNS: distributed database




.edu server doesn't change much
optimization: *cache* its address

check for updated version once in a while

# IPv4 addresses

32-bit numbers

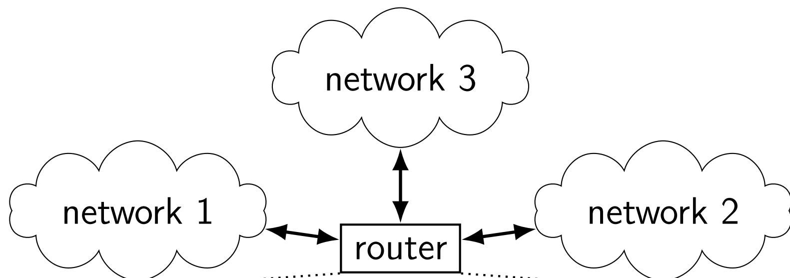
typically written like 128.143.67.11

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

organizations get blocks of IPs

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

# IPv4 addresses and routing tables




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

# selected special IPv4 addresses

127.0.0.0 — 127.255.255.255 — localhost

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

192.168.0.0–192.168.255.255 and
10.0.0.0–10.255.255.255 and
172.16.0.0–172.31.255.255

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

169.254.0.0-169.254.255.255

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

# network address translation

IPv4 addresses are kinda scarce

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

locally: use private IP addresses for machines

outside: private IP addresses become a single public one

commonly how home networks work (and some ISPs)

# IPv6 addresses

IPv6 like IPv4, but with 128-bit numbers

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

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

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

no need for address translation?

`2607:f8b0:400d:c00::6a` =

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

`2607f8b0400d0c00000000000000006a`$_{\text{SIXTEEN}}$

# selected special IPv6 addresses

`::1` = localhost

anything starting with `fe80` = link-local addresses

never forwarded by routers

# port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

# port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

# port numbers

we run multiple programs on a machine

- IP addresses identifying machine — not enough

so, add 16-bit *port numbers*

- think: multiple PO boxes at address

0–49151: typically assigned for particular services

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

49152–65535: allocated on demand

- default "return address" for client connecting to server

# protocols

protocol = agreement on how to comunicate

sytnax (format of messages, etc.)

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

# human protocol: telephone

| caller | callee |
|---|---|
| caller: pick up phone<br>caller: check for service<br>caller: dial<br>caller: wait for ringing | |
| | callee: "Hello?" |
| caller: "Hi, it's Casey..." | |
| | callee: "Hi, so how about ..." |
| caller: "Sure, ..." | |
| ... | ... |
| | callee: "Bye!" |
| caller: "Bye!" | |
| hang up | hang up |

# layered protocols

IP: protocol for sending data by IP addresses

- mailbox model
- limited message size

UDP: send *datagrams* built on IP

- still mailbox model, but *with port numbers*

TCP: reliable connections built on IP

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

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

# other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

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

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

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

...

# other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

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

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

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

...

# sockets

socket: POSIX abstraction of network I/O queue

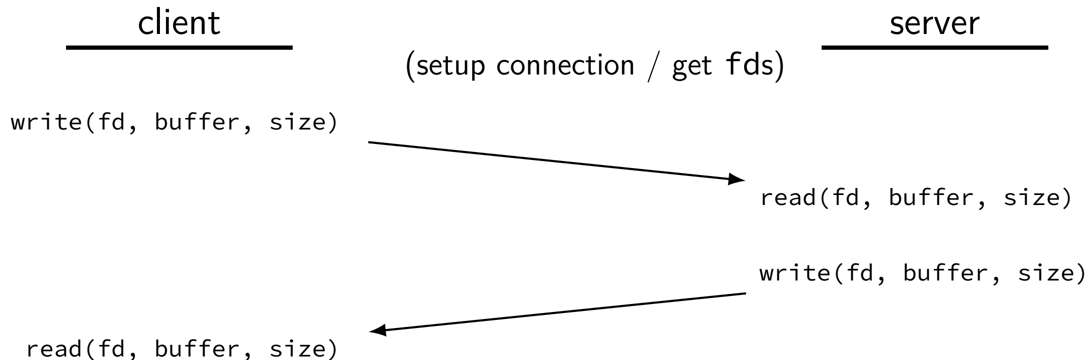
- any kind of network
- can also be used between processes on same machine

a kind of file descriptor

# connected sockets

sockets can represent a connection

act like bidirectional pipe



# echo client/server

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

---

```
void server_for_connection(int socket_fd) {
    int read_count, write_count; char request_buf[MAX_SIZE];
    while (1) {
        read_count = read(socket_fd, request_buf, 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?...}
    }
}
```

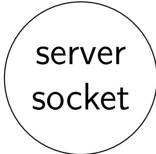
# echo client/server

```
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);
    }
}
```

---

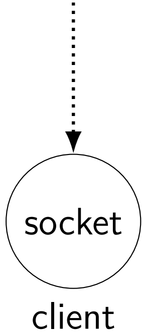
```
void server_for_connection(int socket_fd) {
    int read_count, write_count; char request_buf[MAX_SIZE];
    while (1) {
        read_count = read(socket_fd, request_buf, MAX_SIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);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

# sockets and server sockets




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

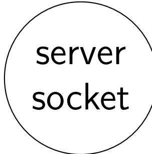
```
client:
fd = socket(…)
```




client

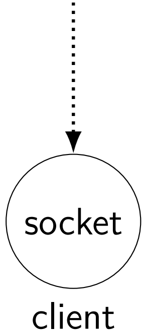
server

# sockets and server sockets




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

client:
`fd = socket(…)`

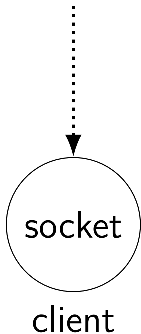



socket() function — create socket fd

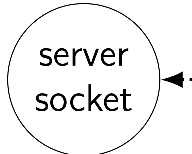
server

# sockets and server sockets

```
client:
fd = socket(…)
```




client

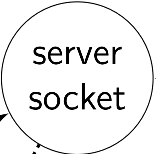


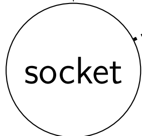

```
server:
ss_fd = socket(…)
…
listen(ss_fd, …)
```

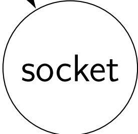
listen() — turn socket into server socket
still has a file descriptor, but …
can only `accept()` — create normal socket

server

# sockets and server sockets



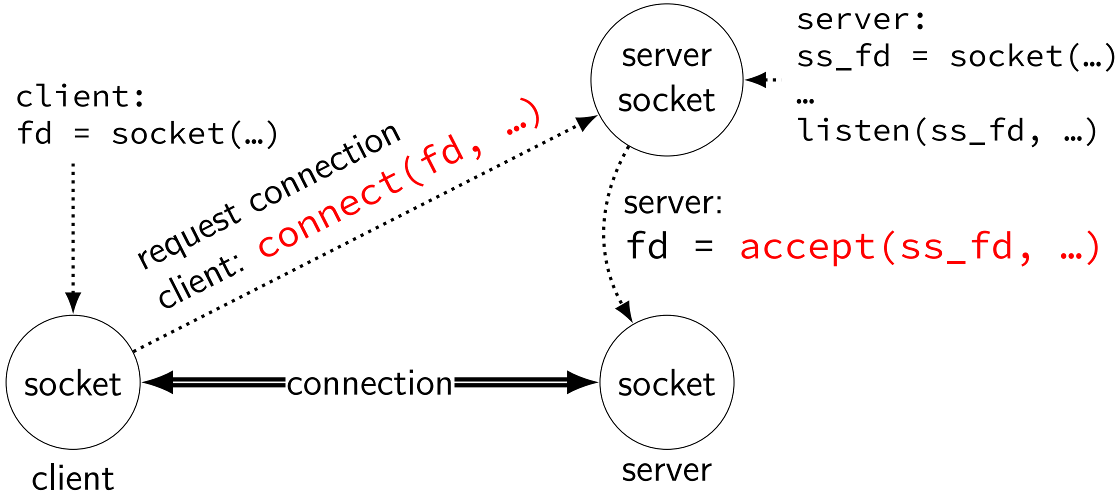

client:
fd = socket(…)

request connection
client: connect(fd, …)

server:
ss_fd = socket(…)
…
listen(ss_fd, …)

server:
fd = accept(ss_fd, …)

# sockets and server sockets



# connections in TCP/IP

connection identified by *5-tuple*

- used to mark messages sent on network
- used by OS to lookup "where is the file descriptor?"

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

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

both ends always have an address+port

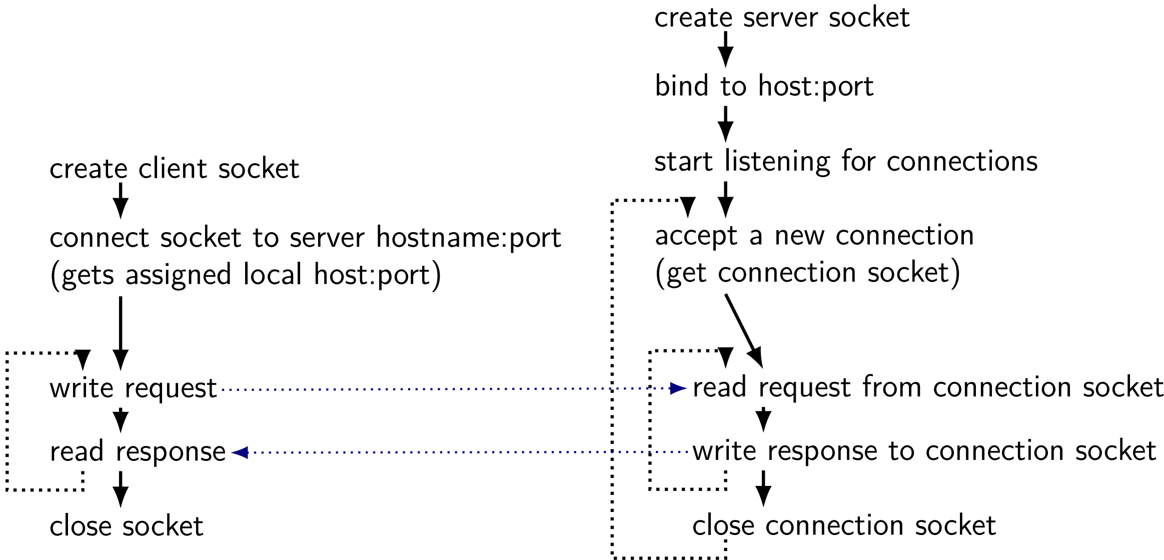
what is the IP address, port number? set with `bind()` function

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

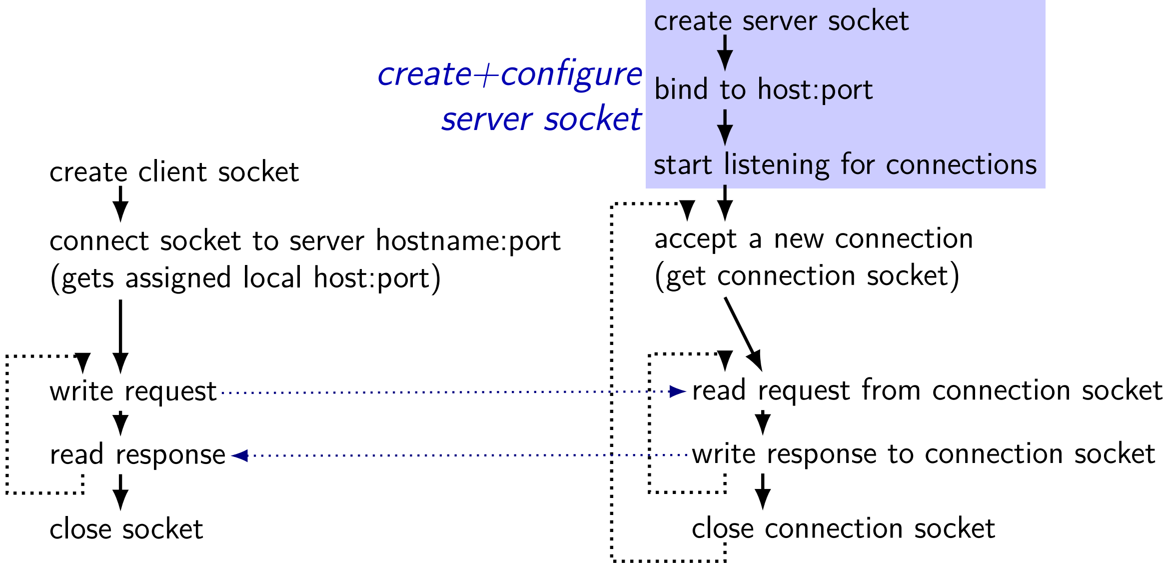
# connections on my desktop

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

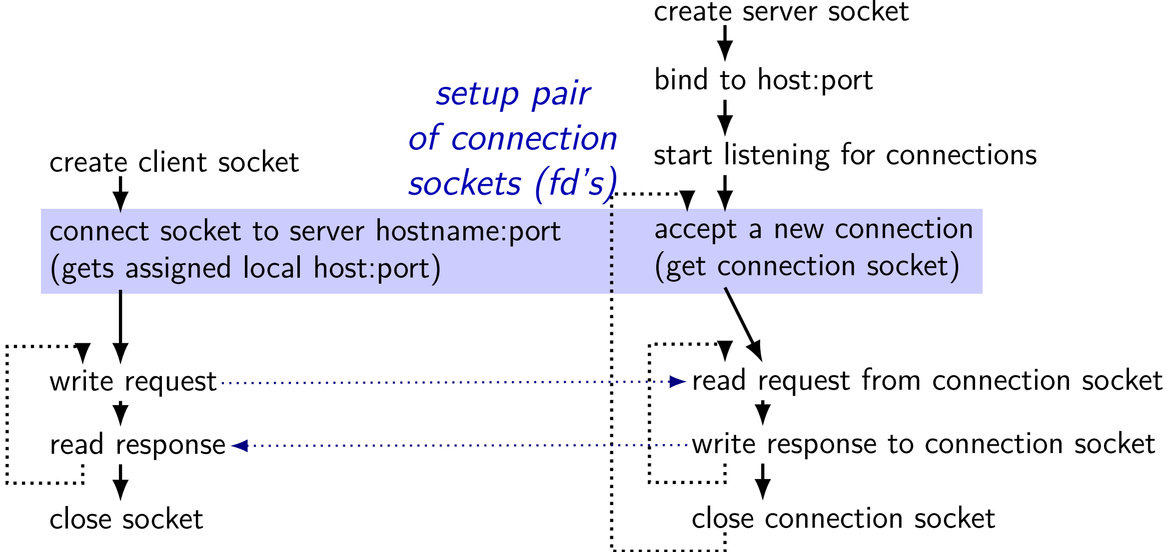
# client/server flow (one connection at a time)



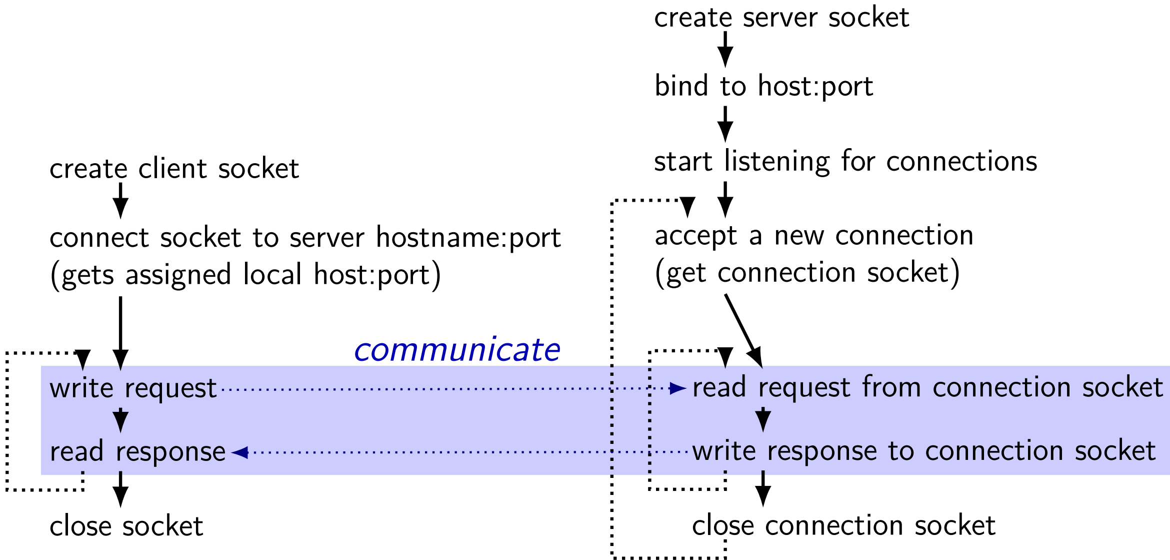
# client/server flow (one connection at a time)



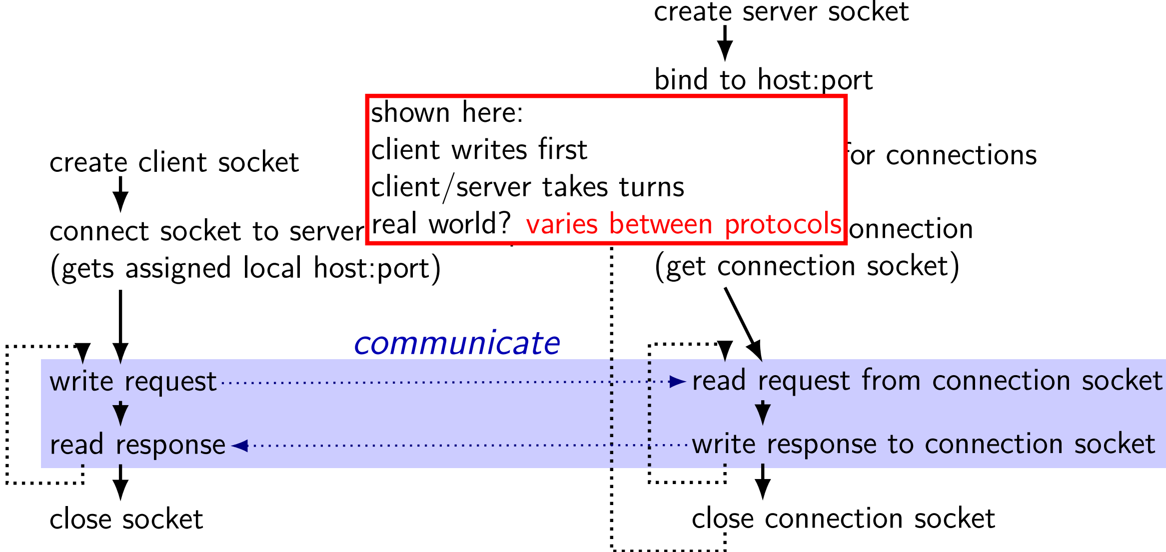
# client/server flow (one connection at a time)



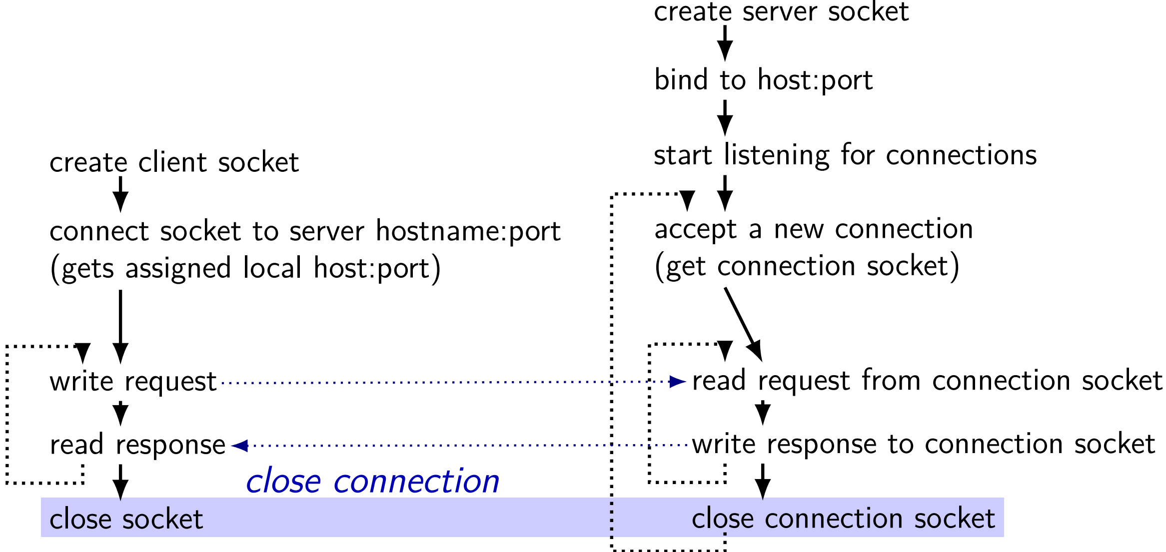
# client/server flow (one connection at a time)



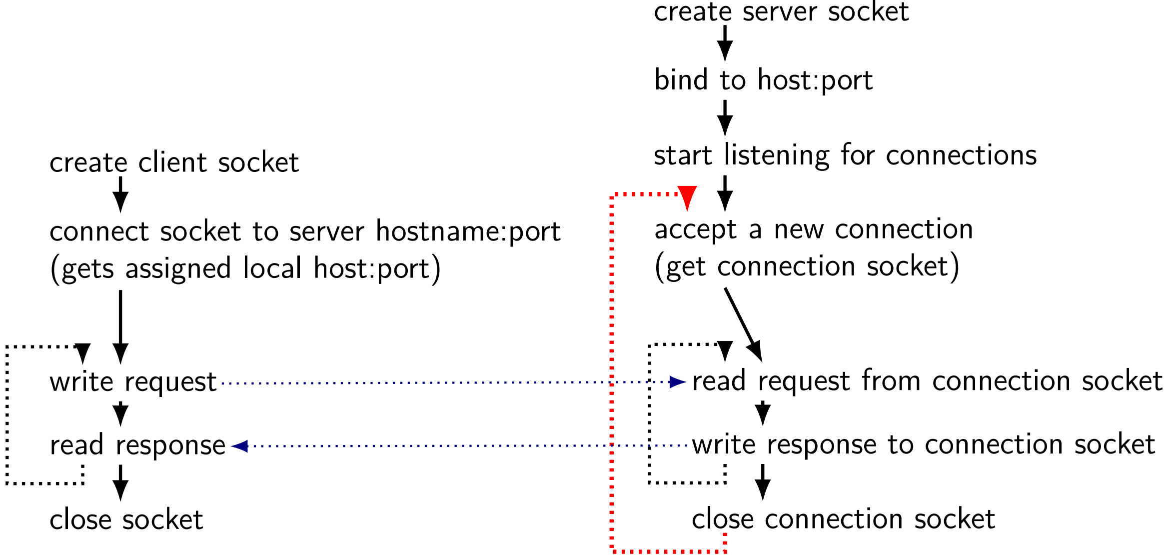
# client/server flow (one connection at a time)



# client/server flow (one connection at a time)



# client/server flow (one connection at a time)



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

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

# connection setup: client, using addrinfo

```
int sock_fd;
str
soc
    // 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

```c
/* 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 c
...
struct
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) */
```

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(                                    ver);
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 = getaddrinf                                              r);
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 *hos
...
struct addrinfo *server;
struct addrinfo hints;
int rv;

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

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

AI_PASSIVE: “I’m going to use bind”

# connection setup: server, addrinfo

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

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

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

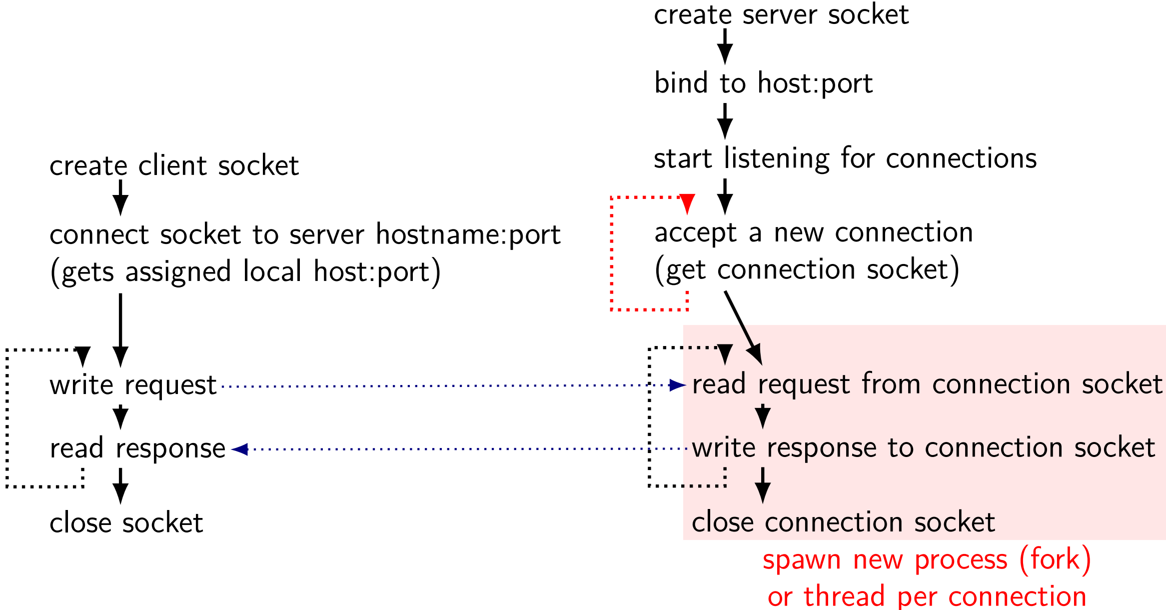
# aside: on server port numbers

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

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

so, for testing: probably ports $> 1023$

# client/server flow (multiple connections)



# reading and writing at once

so far assumption: alternate between reading+writing

- sufficient for FTP assignment
- how many protocols work

"half-duplex"

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

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

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

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

# local/Unix domain sockets

POSIX defines sockets that only work on local machine

example use: apps talking to display manager program

- want to display window? connect to special socket file
- probably don't want this to happen from remote machines

equivalent of name+port: socket file

- appears as a special file on disk

we will use this in assignment

- but you won't directly write code that uses POSIX API

# Unix-domain sockets: client example

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, "/path/to/server.socket");
int fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (connect(fd, &server_addr, sizeof(server_addr)) < 0)
    handleError();
... // use 'fd' here
```

# Unix-domain sockets: client example

```
struct sockaddr_un server_addr;
server_addr.sun_family = AF_UNIX;
strcpy(server_addr.sun_path, "/path/to/server.socket");
int fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (connect(fd, &server_addr, sizeof(server_addr)) < 0)
    handleError();
... // use 'fd' here
```

# Unix-domain sockets on my laptop

```
cr4bd@reiss-lenovo:~$ netstat --unix -a
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  2      [ ]         DGRAM                    40077    /run/user/1000/syst
unix  2      [ ACC ]     SEQPACKET  LISTENING     844      /run/udev/control
unix  2      [ ACC ]     STREAM     LISTENING     40080    /run/user/1000/syst
unix  2      [ ACC ]     STREAM     LISTENING     40084    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37867    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37868    /run/user/1000/bus
unix  2      [ ACC ]     STREAM     LISTENING     37869    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37870    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     60556115 /var/run/cups/cups.
unix  2      [ ACC ]     STREAM     LISTENING     37871    /run/user/1000/gnup
unix  2      [ ACC ]     STREAM     LISTENING     37874    /run/user/1000/keyr
unix  2      [ ACC ]     STREAM     LISTENING     49772163 /run/user/1000/puls
unix  2      [ ACC ]     STREAM     LISTENING     49772158 /run/user/1000/puls
unix  2      [ ACC ]     STREAM     LISTENING     59062776 /run/user/1000/spee
unix  2      [ ACC ]     STREAM     LISTENING     32980    @/tmp/.X11-unix/X0
unix  2      [ ACC ]     STREAM     LISTENING     60557382 /run/cups/cups.sock
...
```

# remote procedure calls

goal: I write a bunch of functions

can call them from another machine

some tool + library handles all the details

called *remote procedure calls* (RPCs)

# transparency

common hope of distributed systems is *transparency*

transparent = can “see through” system being distributed

for RPC: no difference between remote/local calls

(a nice goal, but...we’ll see)

# stubs

typical RPC implementation: generates *stubs*

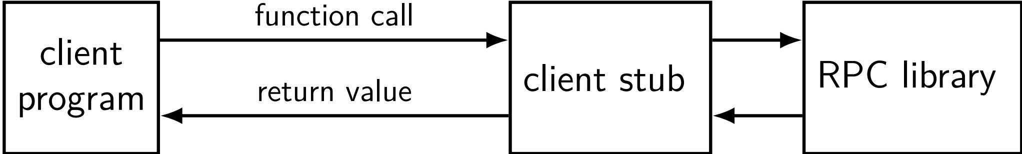
*stubs* = wrapper functions that stand in for other machine

calling remote procedure? call the stub

- same prototype are remote procedure

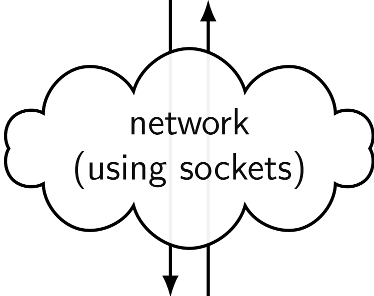
implementing remote procedure? a stub function calls you

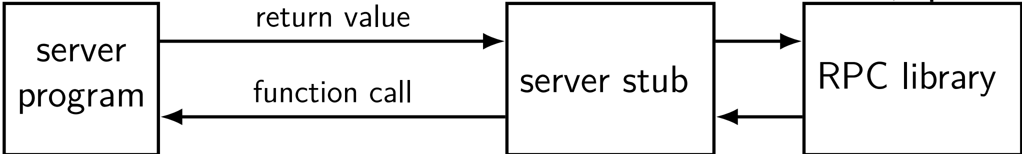
# typical RPC data flow



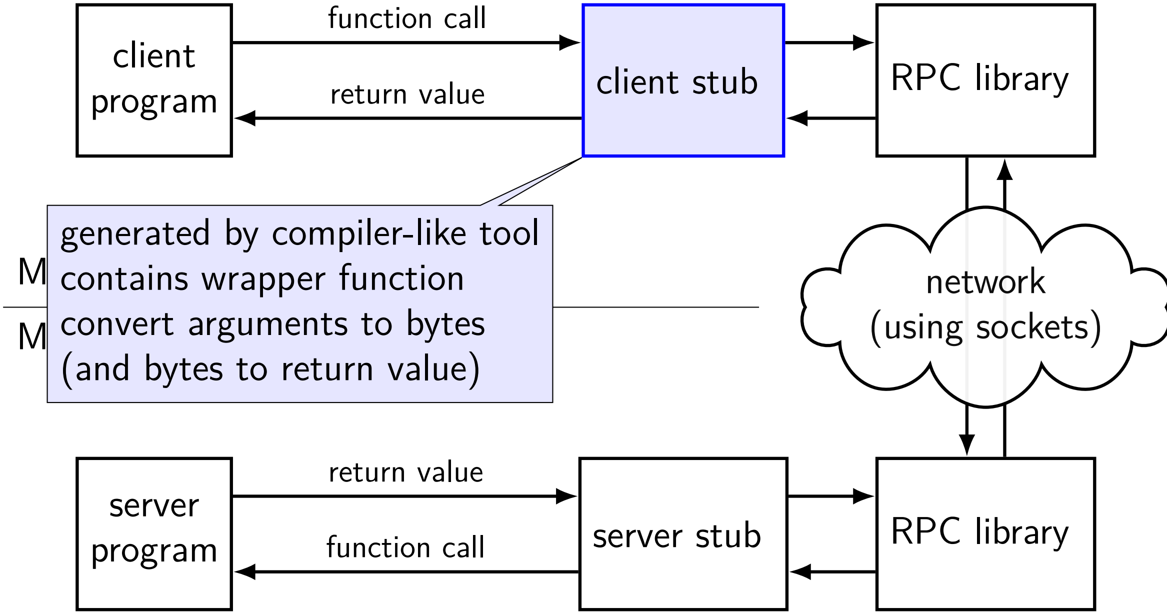

Machine A (RPC client)

Machine B (RPC server)

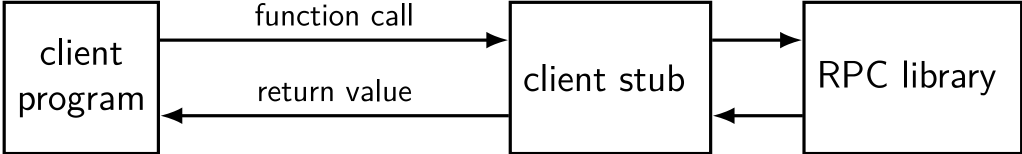


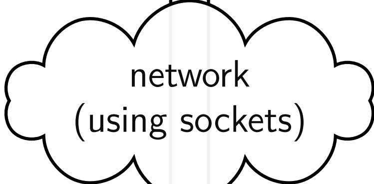



# typical RPC data flow



# typical RPC data flow

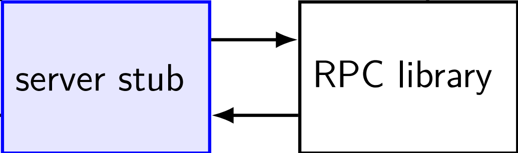


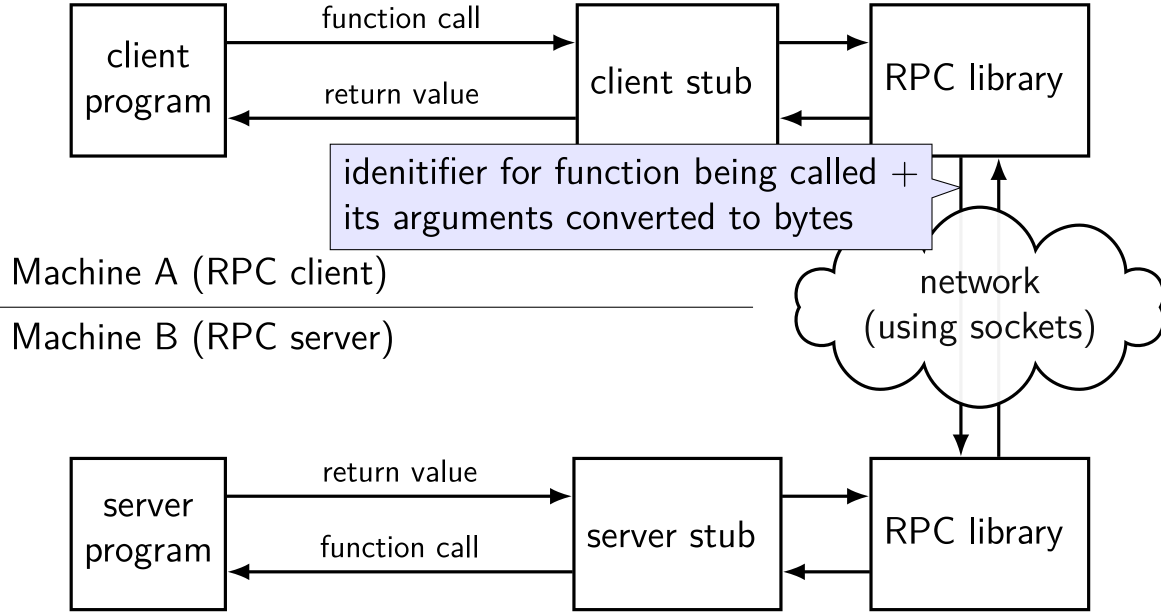

Machine A (RPC client)

Machine B (RPC server)

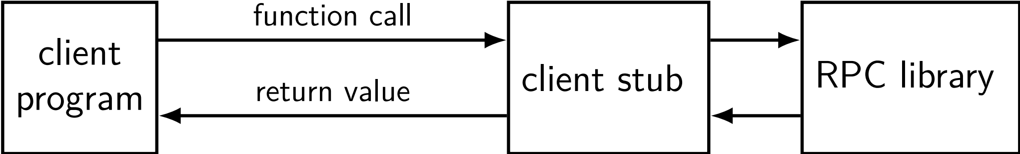
generated by compiler-like tool
contains actual function call
converts bytes to arguments
(and return value to bytes)



# typical RPC data flow

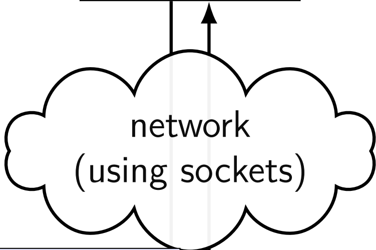


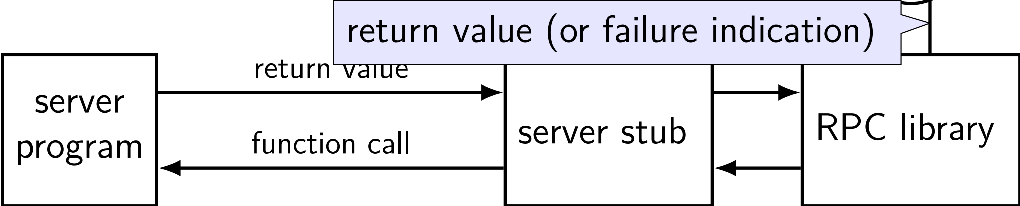
# typical RPC data flow




Machine A (RPC client)

Machine B (RPC server)



# RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

---

server:

```
main() {
  dirprotocol_RunServer();
}

// called by server stub
int real_dirprotocol_mkdir(RPCLibraryContext context, char *name) {
   ...
}
```

# RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

server:

```
main() {
  dirprotocol_RunServer();
}

// called by server stub
int real_dirpro                                    t, char *name) {
  ...
}
```

*context* to specify and pass info about where the function is actually located

# RPC use pseudocode (C-like)

client:

```
    RPCContext context = RPC_GetContext("server name");
    ...
    // dirprotocol_mkdir is the client stub
    result = dirprotocol_mkdir(context, "/directory/name");
```

---

server:

```
main() {
  dirprotocol_RunServer();
}

// called b
int real_di                                           har *name) {
  ...
}
```

transparency failure:

doesn't look like a normal function call anymore

can we do better than this?

# RPC use pseudocode (OO-like)

client:

```
    DirProtocol* remote = DirProtocol::connect("server name");

    // mkdir() is the client stub
    result = remote->mkdir("/directory/name");
```

---

server:

```
main() {
    DirProtocol::RunServer(new RealDirProtocol, PORT_NUMBER);
}

class RealDirProtocol : public DirProtocol { public:
  int mkdir(char *name) {
    ...
  }
};
```

# backup slides

# sockaddr_in

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

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

# sockaddr_in

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

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

# sockaddr_in

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

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

trick: multiple versions of address struct
each have "type" information in same spot
OS/library checks before using

# sockaddr_in6

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

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

# sockaddr_in6

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

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

# connection setup: client — 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 */ }

str
add
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 = /* code
sock_fd = socket
     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 = s
    AF_INET
    SOCK_ST
    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: old lookup function

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

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

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

# connection setup: server, manual

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

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

# connection setup: server, manual

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

if (bind(server_socket_fd, &addr, sizeof(addr)) < 0) {
    /* handle error */
}
listen
...
int 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 */
}
list
...
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

# aside: send/recv

sockets have some alternate read/write-like functions:

- recv, recvfrom, recvmsg
- send, sendmsg

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

# incomplete writes

`write` might write less than requested

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

`read` might read less than requested

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

# handling incomplete writes

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

# on filling buffers

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

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

# reading lines

(note: code below is not tested)

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

# aside: getting addresses

on a socket fd: `getsockname` = local addresss

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

on a socket fd: `getpeername` = remote address

# addresses to string

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

another option: `getnameinfo`

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

# example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

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

set the SO_REUSEADDR "socket option"

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

client handles reading until a newline

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

# example echo client/server

handle reporting errors from incomplete writes

handle avoiding SIGPIPE

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

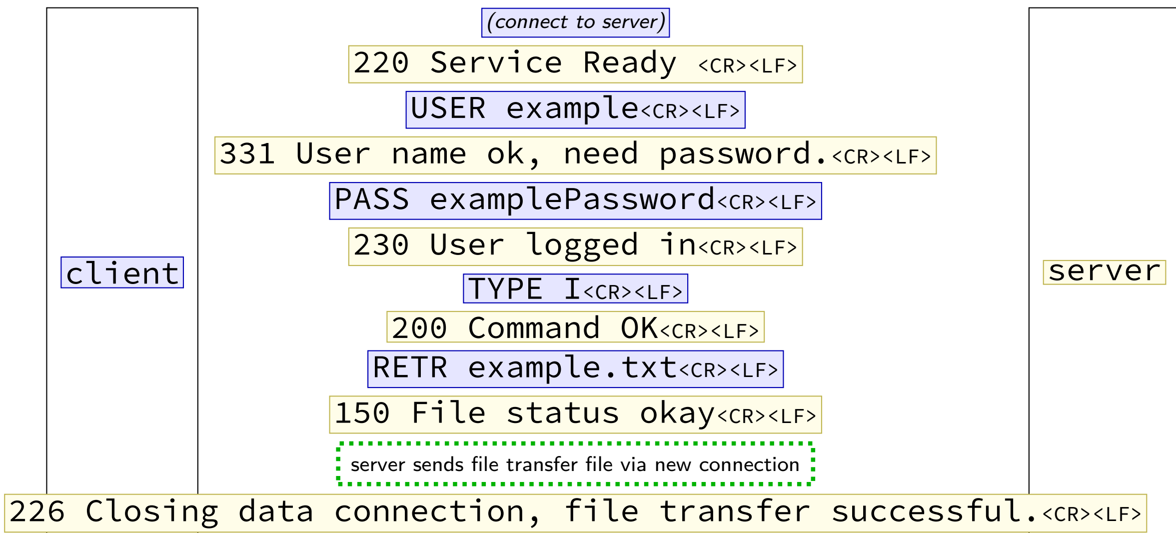
set the SO_REUSEADDR "socket option"

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

client handles reading until a newline

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

# FTP protocol (simplified)



# notable things about FTP

FTP is stateful — previous commands change future ones

- logging in for whole connection
- change current directory
- set image file type (binary, not text)

FTP uses separate connections for transferring data

- PASV: client connects separately to server
- PORT: client specifies where server connects
- (+ very rarely used default: connect back to port 20)

status codes for every command

# kernel FS abstractions

Linux: *virtual file system* API

object-oriented, based on FFS-style filesystem

to implement a filesystem, create object types for:
- superblock (represents "header")
- inode (represents file)
- dentry (represents cached directory entry)
- file (represents *open file*)

common code handles directory traversal
- and caches directory traversals

common code handles file descriptors, etc.

# linux VFS operations

superblock: write_inodez, sync_fs, ...

inode: create, link, unlink, mkdir, open ...
- most just for inodes which are directories

dentry: compare, delete ...
- more commonly argument to inode operation
- can be created for non-yet-existing files

file: read, write, ...

# linux VFS operations example

```
struct inode_operations {
    struct dentry * (*lookup) (struct inode *,struct dentry *, unsig
    ...
    int (*create) (struct inode *,struct dentry *, umode_t, bool);
    int (*link) (struct dentry *,struct inode *,struct dentry *);
    int (*unlink) (struct inode *,struct dentry *);
    int (*symlink) (struct inode *,struct dentry *,const char *);
    int (*mkdir) (struct inode *,struct dentry *,umode_t);
    int (*rmdir) (struct inode *,struct dentry *);
    int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
    int (*rename) (struct inode *, struct dentry *,
                   struct inode *, struct dentry *, unsigned int);
    ...
    int (*update_time)(struct inode *, struct timespec64 *, int);
    int (*atomic_open)(struct inode *, struct dentry *,
                       struct file *, unsigned open_flag,
                       umode_t create_mode);
    ..
}
```

# FS abstractions and awkward FSes

example: inode object for FAT?

- fake it: point to directory entry?