last time

redo logging to ensure consistency after crash

- write intention to a log before performing operation
- "commit" message indicates complete intention
- after commit, do operation eventually
- after operation done, eventually cleanup log
- on crash: do operation for anything committed, ignore anything not

why distributed systems?

- non-technical reasons: cooperation, separate management
- technical reasons: relability, adding capacity incrementally, ...

mailbox model: send 'letters' with address over network

connection model: two way pipe over network

names versus addresses

on difficulty, etc. (1)

on grading:

I'm not going to add additional "extra credit" assignments

- making "too much work" problem worse

the grading policy listed says that the thresholds for a D-/GC are 60% weighted raw score *or lower*

- I'll make this decision based on how the final goes (Spring 2020: 54% was D-/GC threshold)
- department grading guidelines: "D is used for students who demonstrate minimal competence in learning objectives, but not enough to recommend further studies or activities in related areas."

on difficulty, etc. (2)

re: amount of time on assignments

from surveys a few years ago: big variance in self-reported time on assignments

- probably worse with less effective office hours + other changes — but how much?

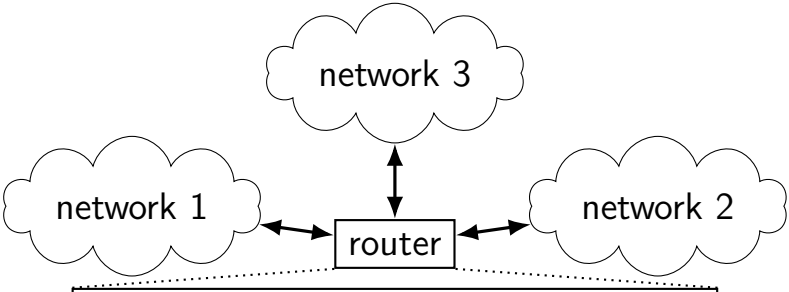
would like to understand/avoid the sort of issues that cause some students to report much higher amount of time than others

- more guidance re: C++ pointer issues?
  - some prior faculty have tried review session at beg. of semester, but seems students who need it most didn't attend
  - could maybe provide some more utility functions
- more guidance re: general code organization
  - (but worried about polluting already long assignment writeups)
- guidance re: debugging?
  - better testing code can help sometimes
  - but it seems providing more testing code often makes students debug worse

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

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

connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

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

syntax (format of messages, etc.)

- e.g. mailbox model: where does address go?
- e.g. connection: where does return address go?

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

- e.g. connection: when to consider connection created?

human protocol: telephone

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

layered protocols

IP: protocol for sending data by IP addresses

- mailbox model
- limited message size

UDP: send *datagrams* built on IP

- still mailbox model, but *with port numbers*

TCP: reliable connections built on IP

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

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

other notable protocols (transport layer)

TLS: Transport Layer Security — built on TCP

like TCP, but adds encryption + authentication

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

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

HTTPS: HTTP, but over TLS instead of TCP

FTP: file transfer protocol

...

sockets

socket: POSIX abstraction of network I/O queue

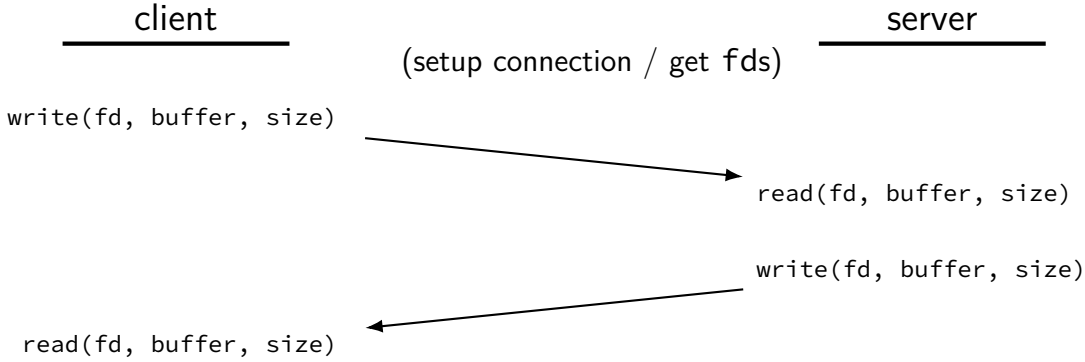
- any kind of network
- can also be used between processes on same machine

a kind of file descriptor

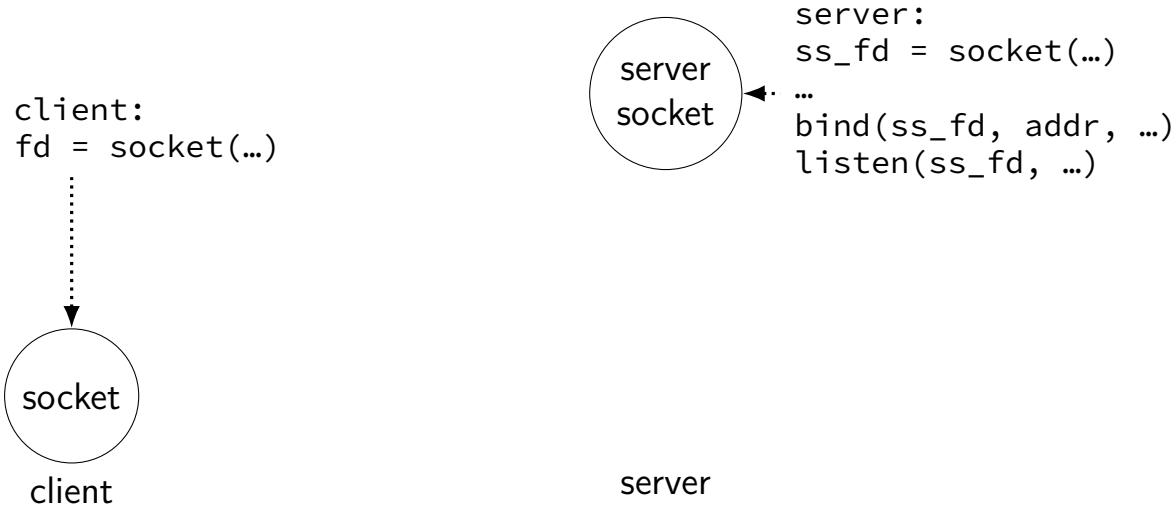
connected sockets

sockets can represent a connection

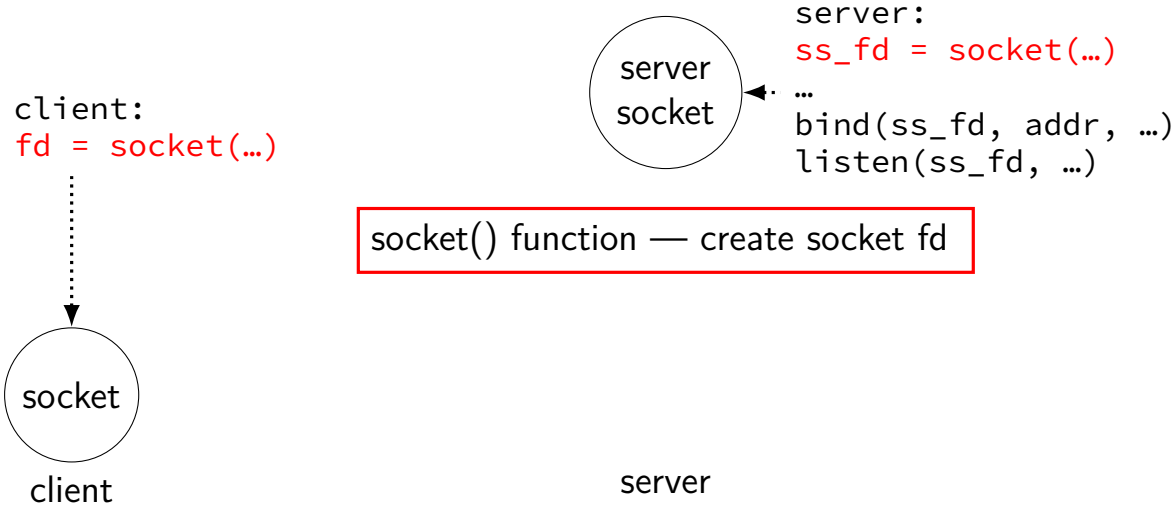
act like bidirectional pipe



sockets and server sockets

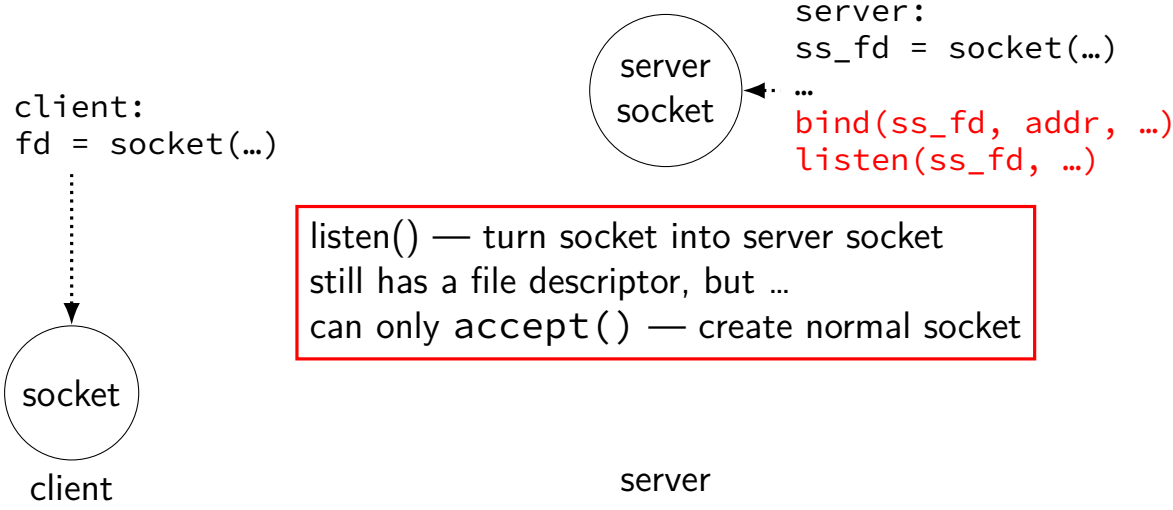


sockets and server sockets



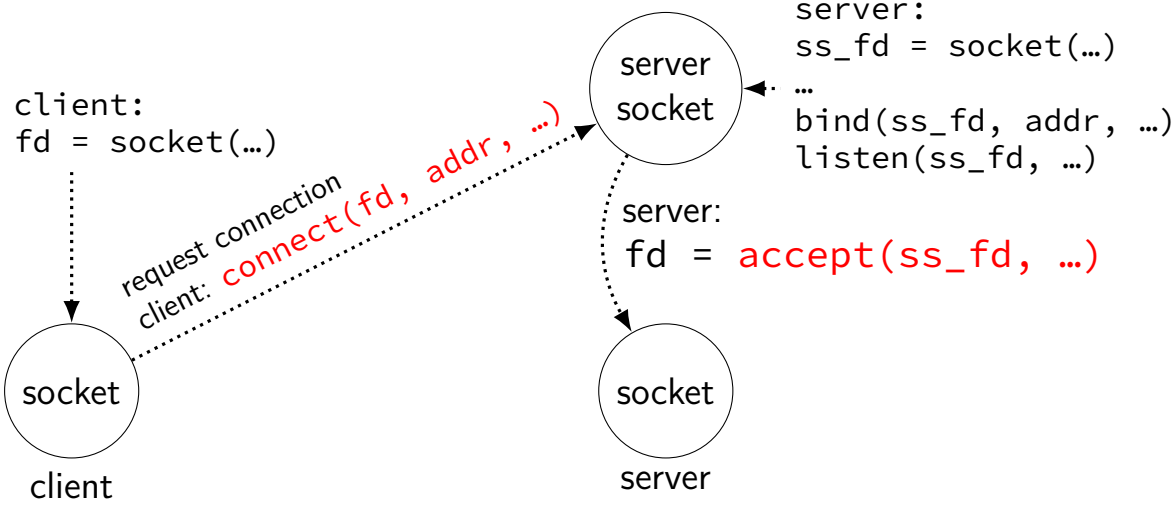
socket() function — create socket fd

sockets and server sockets

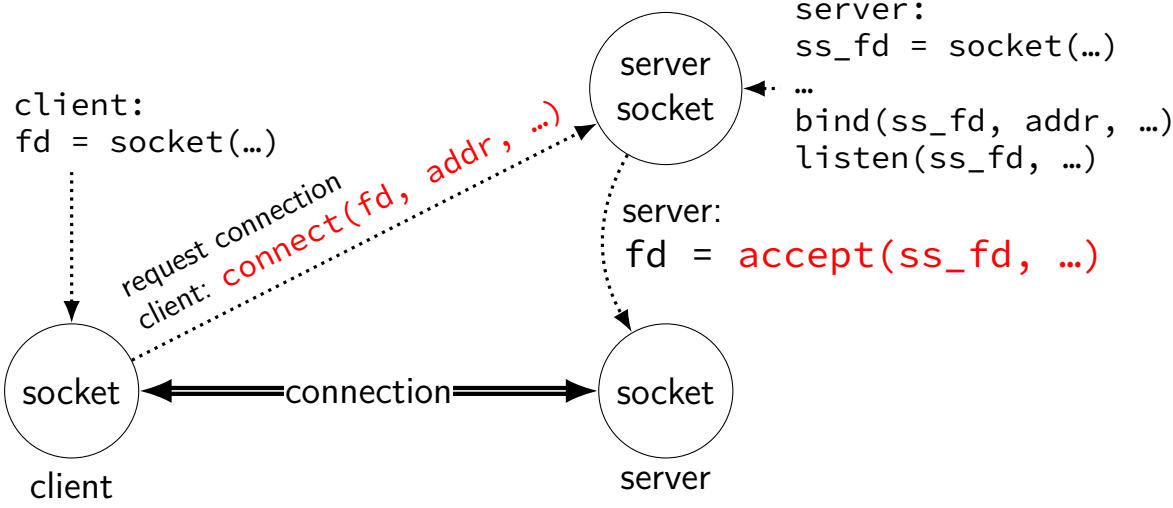


listen() — turn socket into server socket
still has a file descriptor, but ...
can only `accept()` — create normal socket

sockets and server sockets



sockets and server sockets



connections in TCP/IP

on network: connection identified by *5-tuple*

used by OS to lookup "where is the file descriptor?"

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

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
```

exercise

if I have a server socket and I call accept() on it to create a connection,
we would expect this to send a message to the client machine:

A. immediately after the call to accept()
B. sometime after the client machine calls connect()
C. A and B
D. neither A nor B

for the server to talk to the client that just connected, it should write() to

A. the server socket that it passed to accept()
B. the file descriptor returned from accept()
C. A or B (either will work)
D. neither A nor B

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 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*

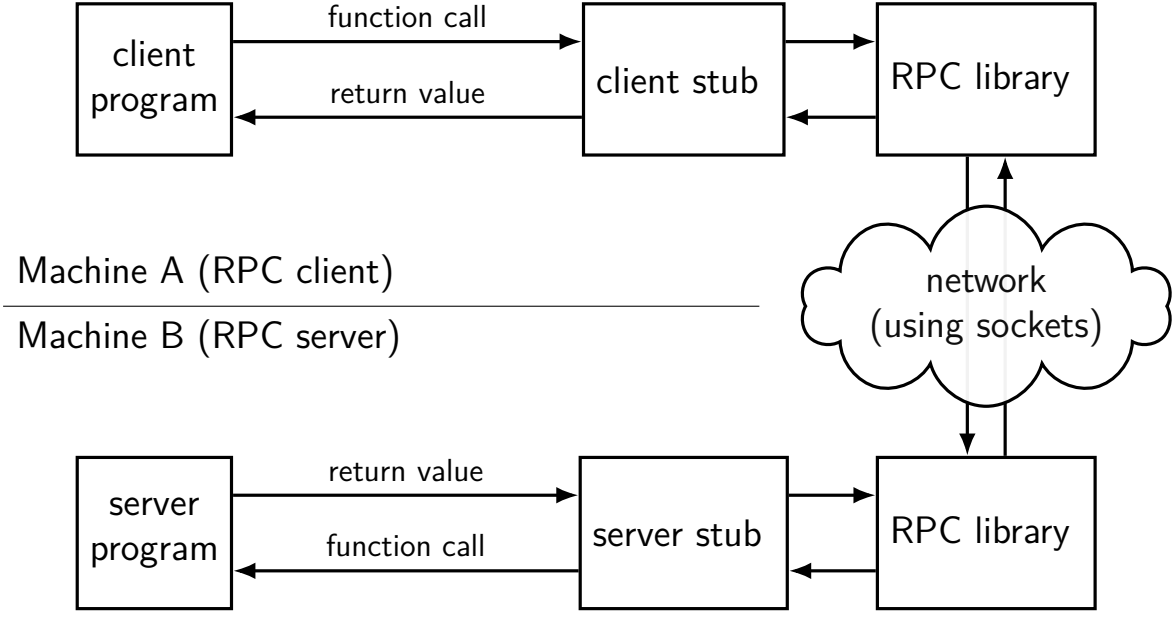
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

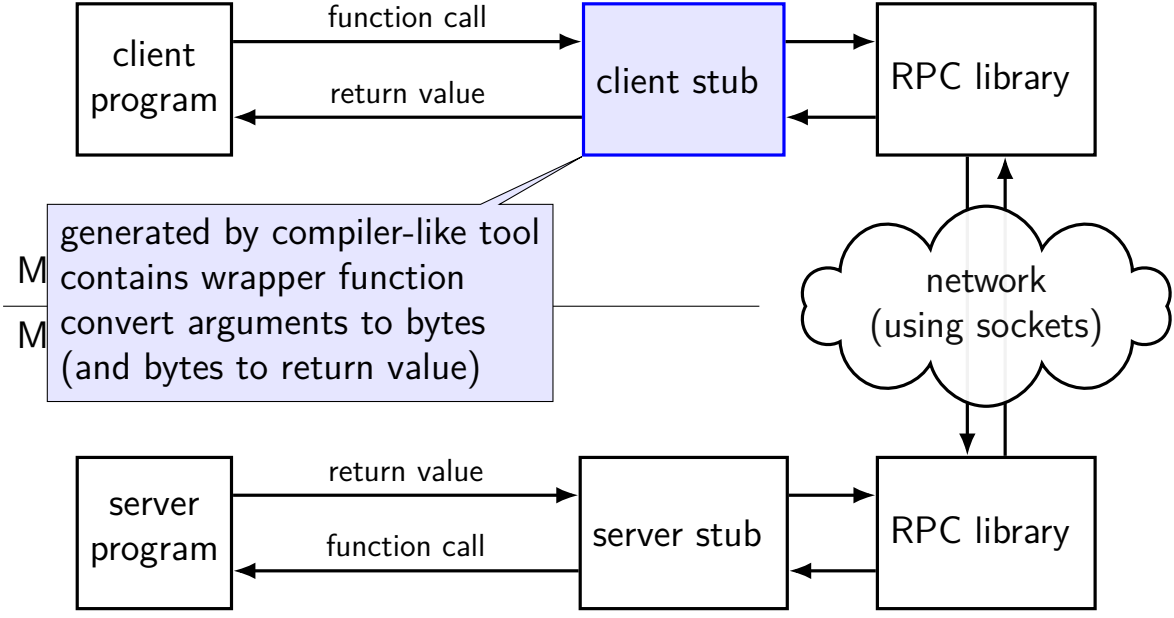
same prototype are remote procedure

implementing remote procedure? a stub function calls you

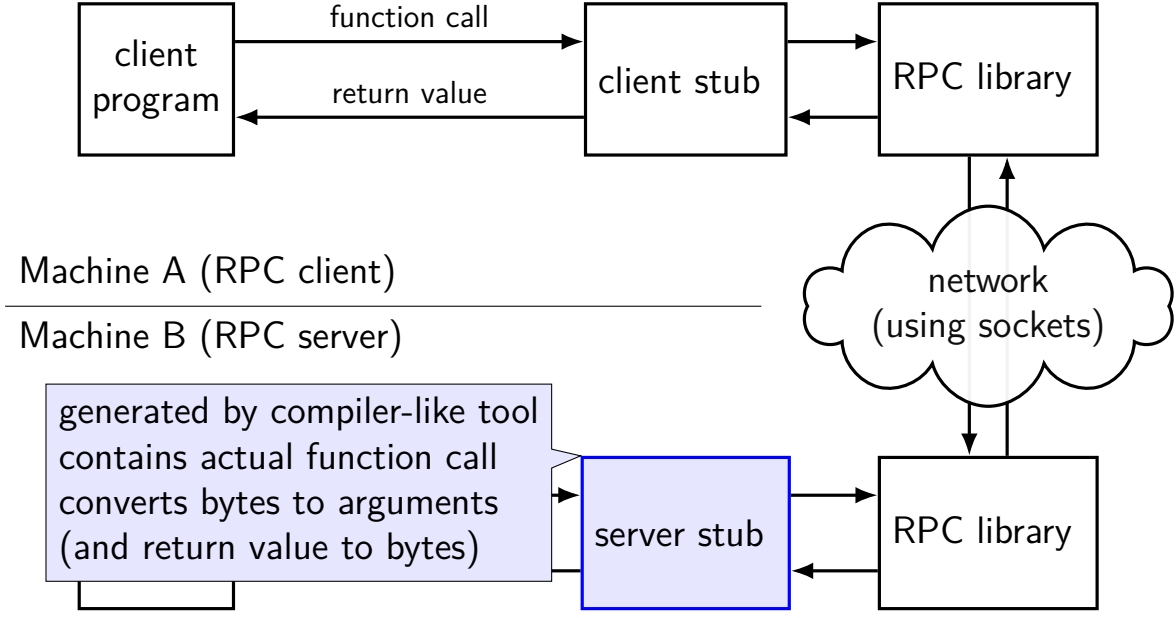
typical RPC data flow



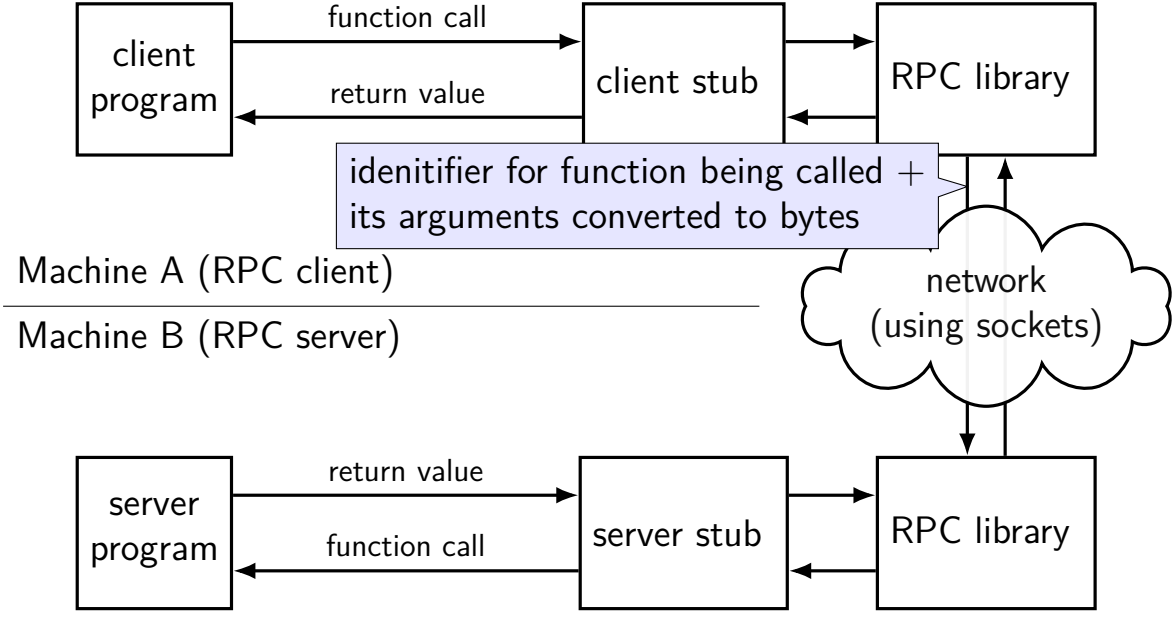
typical RPC data flow



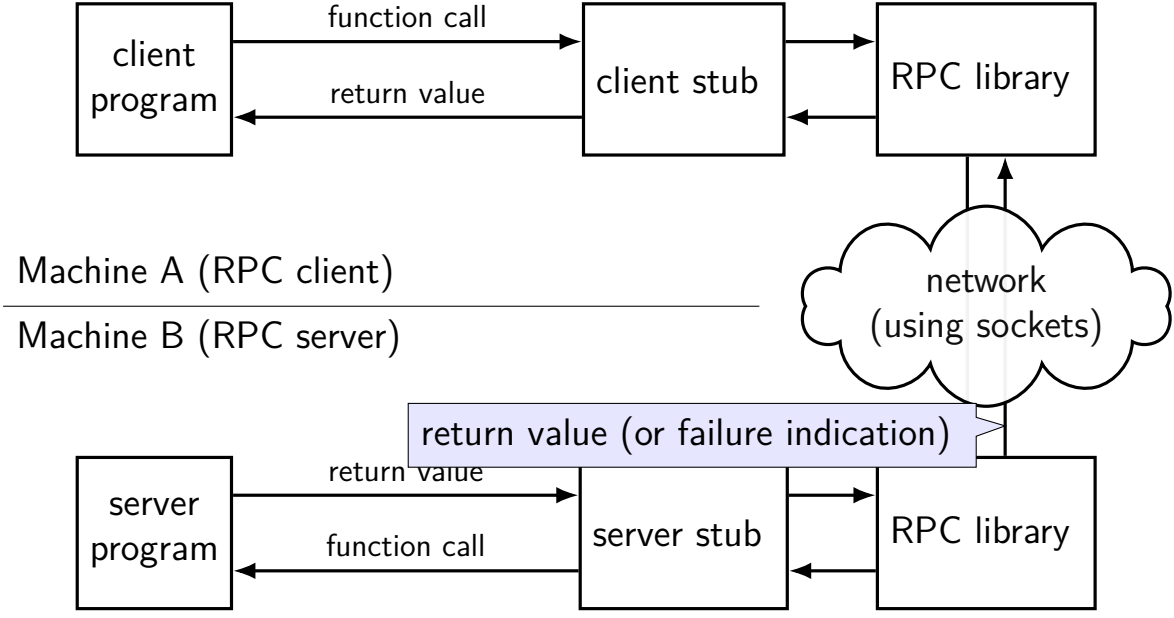
typical RPC data flow



typical RPC data flow



typical RPC data flow



exercise: errors that can occur in RPC?

exercise: ways *remote* procedure calls can fail that local procedure calls probably can't?

(name examples in the chat)

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    exce
      co
    retu
```

client: calls "MakeDirectory" function on server

local-only code would have been:

`MakeDirectory(path="/directory/name")`

gRPC code preview

server: defines "MakeDirectory" function

local-only code would have been:

```
def MakeDirectory(path):
    ...
```

client:

```
stub = ...
try:
  stub.MakeDir                                    ory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return Empty()
```

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except
      conte
    return
```

stub and *context* to pass info about
where the function is actually located (on client)
and how it was called (on server)

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except
      conte
    return
```

gRPC requires exactly one arguments object
to simplify library/cross-language compatability
some other RPC systems are more flexible

)

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    exc
      c
    ret
```

generated code ("server stub") defines base class
server subclass overrides methods to provide remote calls
so it's easy for library to find them

gRPC code preview

client:

```
stub = ...
try:
  stub.MakeDirectory(MakeDirectoryArgs(path="/directory/name"))
except:
  # handle error
```

server:

```
class DirectoriesImpl(DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return Empty()
```

marshalling

RPC system needs to send arguments over the network

- and also return values

called *marshalling* or *serialization*

can't just copy the bytes from arguments

- pointers (e.g. `char*`)
- different architectures (32 versus 64-bit; endianness)

interface description langauge

tool/library needs to know:

- what remote procedures exist
- what types they take

typically specified by RPC server author in *interface description language*

- abbreviation: IDL

compiled into stubs and marshalling/unmarshalling code

why IDL?

could just use a source file, but...

missing info: how should a char be passed?

- string? fixed length array? pointer to single char?
- who allocates the memory?

want to be machine/programming language-neutral

- choose set of types that work in both C, Python

versioning/compatiblity

- what if older server interoperates with newer client?

gRPC IDL example + marshalling

```
message MakeDirArgs { string path = 1; }

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {
}
```

---

example possible format (*not what gRPC actually does*):

MakeDirectory(MakeDirArgs(path="/foo")) becomes:

```
\x0dMakeDirectory\x01\x04/foo
```

`0x0d` = length of 'MakeDirectory'
`0x04` = length of '/foo'

GRPC examples

will show examples for gRPC

RPC system originally developed at Google

what we'll use for upcoming assignment

defines interface description language, message format

uses a protocol on top of HTTP/2

note: gRPC makes some choices other RPC systems don't

GRPC IDL example

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

message DirectoryEntry {
    string name = 1;
    bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

GRPC IDL example

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

message DirectoryEntry {
    string name = 1;
    bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

service D
    rpc M
    rpc L                                                    {}
}
```

messages: turn into C++/Python classes
with accessors + marshalling/demarshalling functions
part of *protocol buffers* (usable without RPC)

GRPC IDL example

```
syntax="proto3";
message MakeDirArgs { string path = 1; }
message ListDirArgs { string path = 1; }

message DirectoryEntry {
    string name = 1;
    bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

service D
    rpc M
    rpc L                                    ) {}
}
```

fields are numbered (can have more than 1 field)
numbers are used in byte-format of messages
allows changing field names, adding new fields, etc.

GRPC IDL example

```
syntax="proto3";
message MakeDirA
message ListDirArgs { string path = 1; }

message DirectoryEntry {
    string name = 1;
    bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

will become method of Python class

GRPC IDL example

```
syntax="pro
message Mak
message ListDirArgs { string path = 1; }

message DirectoryEntry {
    string name = 1;
    bool is_directory = 2;
}

message DirectoryList {
    repeated DirectoryEntry entries = 1;
}

message Empty {}

service Directories {
    rpc MakeDirectory(MakeDirArgs) returns (Empty) {}
    rpc ListDirectory(ListDirArgs) returns (DirectoryList) {}
}
```

rule: arguments/return value always a *message*

RPC server implementation (method 1)

```
import dirproto_pb2
import dirproto_pb2_grpc

class DirectoriesImpl(dirproto_pb2_grpc.DirectoriesServicer):
  ...
  def MakeDirectory(self, request, context):
    print("MakeDirectory called with path=", request.path)
    try:
      os.mkdir(request.path)
    except OSError as e:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return dirproto_pb2.Empty()
```

RPC server implementation (method 2)

```
import dirproto_pb2, dirproto_pb2_grpc
from dirproto_pb2 import DirectoryList, DirectoryEntry

class DirectoriesImpl(dirproto_pb2_grpc.DirectoriesServicer):
  ...
  def ListDirectory(self, request, context):
    try:
      result = DirectoryList()
      for file_name in os.listdir(request.path)
        result.entries.append(DirectoryEntry(name=file_name, ...))
    except OSError as err:
      context.abort(grpc.StatusCode.UNKNOWN,
                    "OS returned error: {}".format(err))
    return result
```

RPC server implementation (starting)

```
# create server that uses thread pool with
# three threads to run procedure calls
server = grpc.server(
    futures.ThreadPoolExecutor(max_workers=3)
)
# DirectoriesImpl() creates instance of implementaiton class
# add_DirectoryServicer_to_server part of generated code
dirproto_pb2_grpc.add_DirectoryServicer_to_server(
    DirectoriesImpl()
)
server.add_insecure_port('127.0.0.1:12345')
server.start()  # runs server in separate thread
```

RPC client implementation (method 1)

```
from dirproto_pb2_grpc import DirectoriesStub
from dirproto_pb2 import MakeDirectoryArgs

channel = grpc.insecure_channel('127.0.0.1:43534')
stub = DirectoriesStub(channel)
args = MakeDirectoryArgs(path="/directory/name")
try:
  stub.MakeDirectory(args)
except grpc.RpcError as error:
  ... # handle error
```

RPC client implementation (method 2)

```
from dirproto_pb2_grpc import DirectoriesStub
from dirproto_pb2 import ListDirectoryArgs

channel = grpc.insecure_channel('127.0.0.1:43534')
stub = DirectoriesStub(channel)
args = ListDirectoryArgs(path="/directory/name")
try:
  result = stub.ListDirectory(args)
  for entry in result.entries:
    print(entry.name)
except grpc.RpcError as error:
  ... # handle error
```

RPC non-transparency

setup is not transparent — what server/port/etc.

- ideal: system just knows where to contact?

errors might happen

- what if connection fails?

server and client versions out-of-sync

- can't upgrade at the same time — different machines

performance is very different from local

RPC locally

not uncommon to use RPC on one machine

more convenient alternative to pipes?

allows shared memory implementation

- mmap one common file
- use mutexes+condition variables+etc. inside that memory

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

network failures: two kinds

messages lost

messages delayed/reordered

network failures: message lost?

detect with acknowledgements ("yes I got it")

can recover by retrying

can't distinguish: original message lost or acknowledgment lost

can't distinguish: machine crashed or network down/slow for a while

failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

exercise: RPC failure scenarios

RPC with MakeDirectory("foo")

option A: client stub returns when sent to server

option B: client stub waits for server to return OK

for now, assume only *network* failures

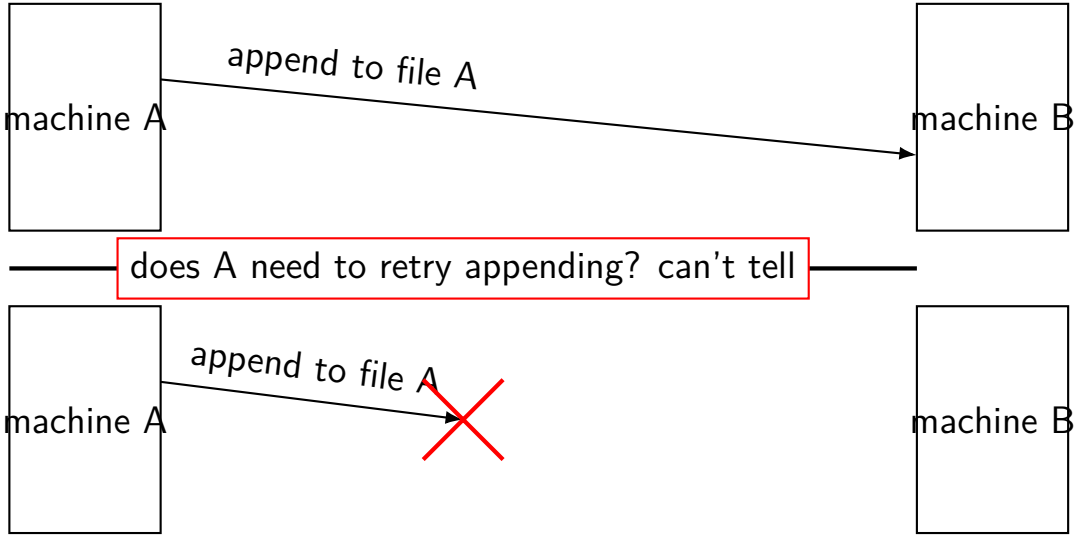
I call MakeDirectory("foo") and it throws an exception:

- with Option A: could directory have been created?
- with Option B: could directory have been created?

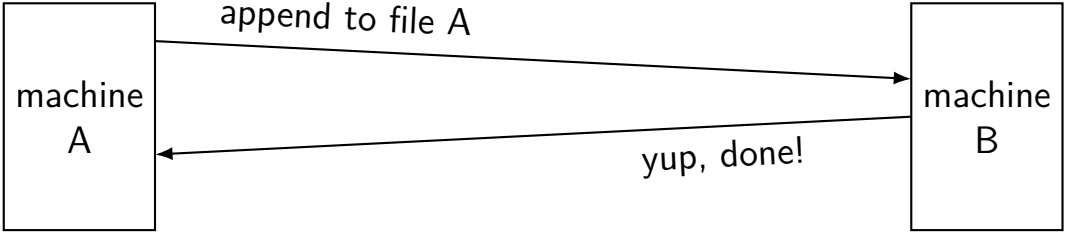
I call MakeDirectory("foo") and it throws no exception:

- with Option A: could directory have NOT been created?
- with Option B: could directory have NOT been created?

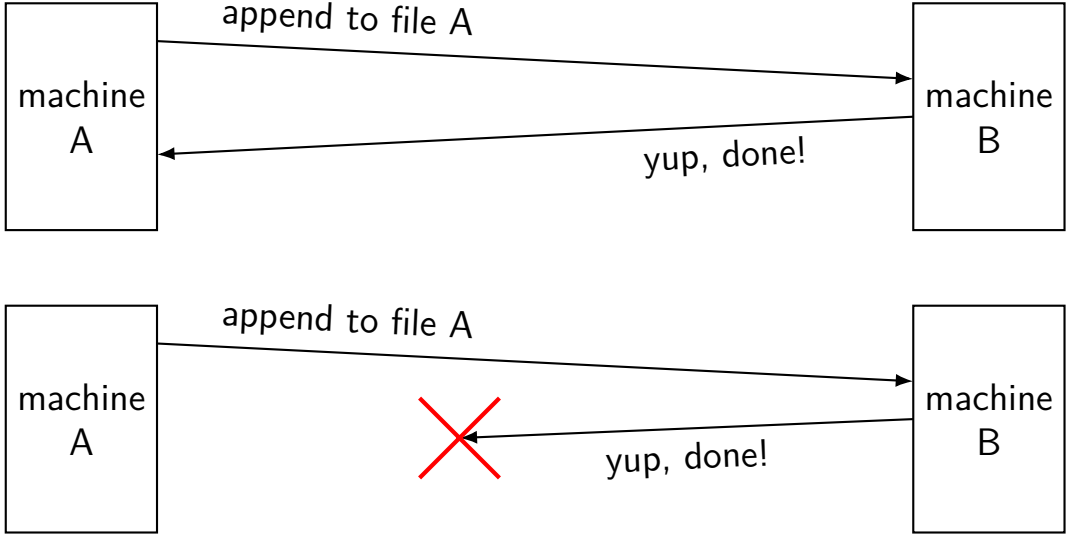
dealing with network message lost



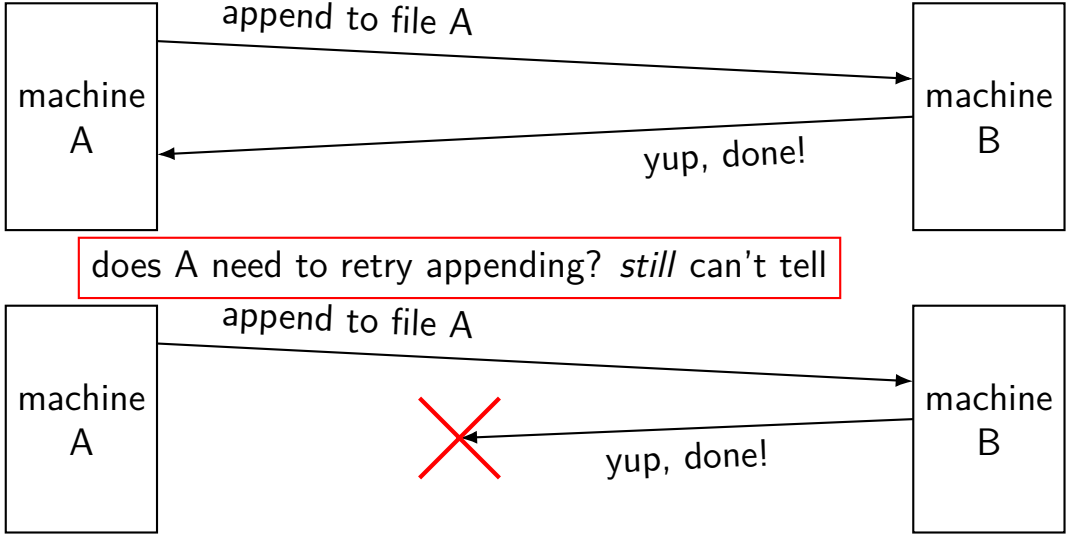
handling failures: try 1



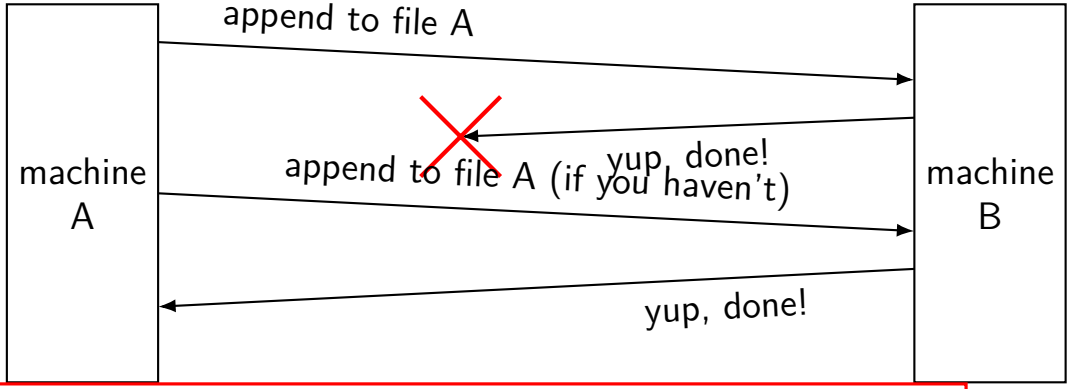
handling failures: try 1



handling failures: try 1



handling failures: try 2



retry (in an idempotent way) until we get an acknowledgement
basically the best we can do, but when to give up?

network failures: message reordered?

can detect with sequence numbers

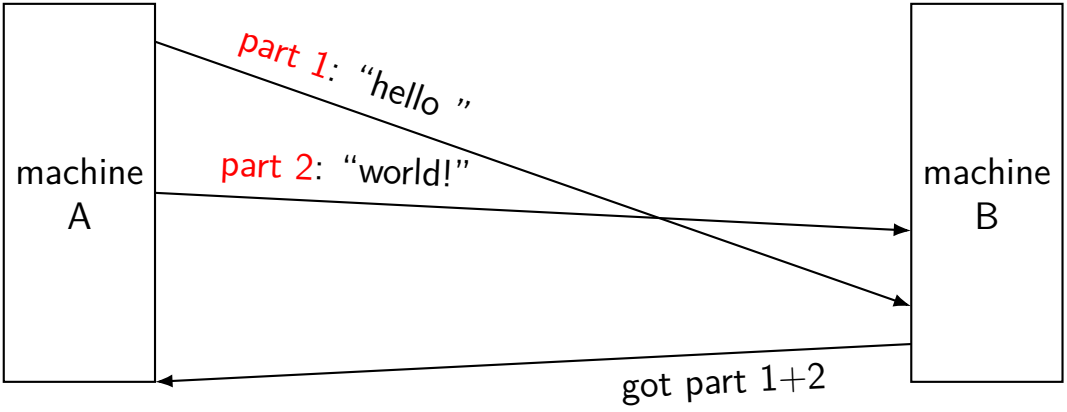
connection protocols do this

RPC abstraction — generally doesn't

potentially receive 'stale' RPC call

can't distinguish: message lost or just delayed and not received yet

handling reordering



failure models

how do networks ‘fail’?...

how do machines ‘fail’?...

well, lots of ways

two models of machine failure

fail-stop

failing machines stop responding/don't get messages

or one always detects they're broken and can ignore them

Byzantine failures

failing machines do the worst possible thing

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F + 1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

dealing with machine failure

recover when machine comes back up

- does not work for Byzantine failures

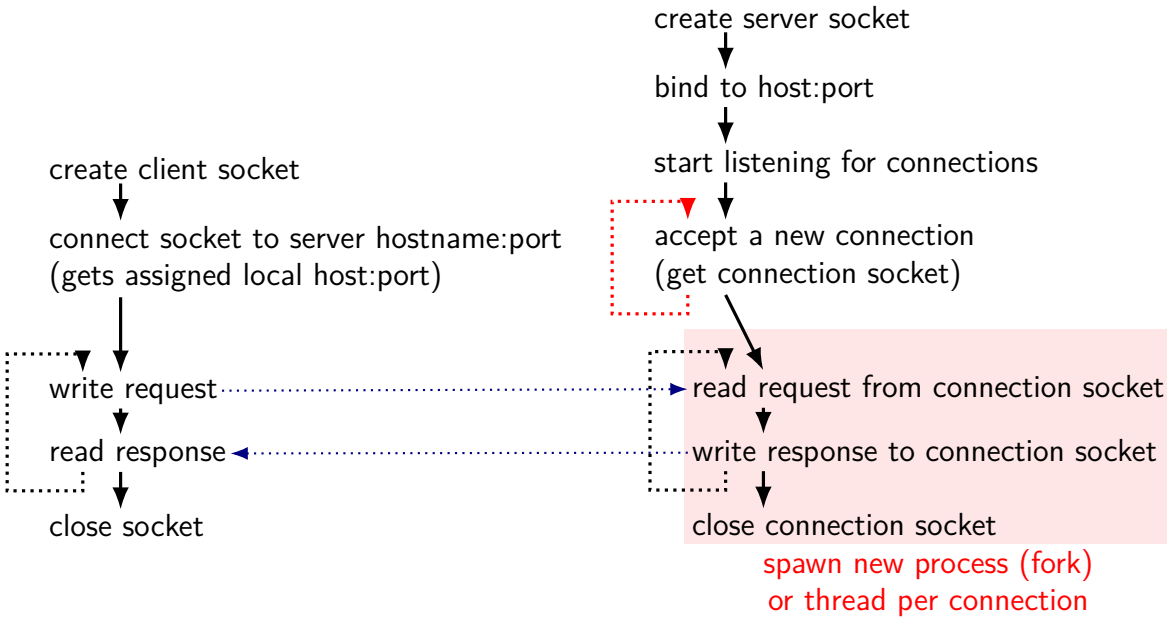
rely on a *quorum* of machines working

- minimum 1 extra machine for fail-stop
- minimum $3F + 1$ to handle F failures with Byzantine failures

can replace failed machine(s) if they never come back

backup slides

client/server flow (multiple connections)



RPC locally

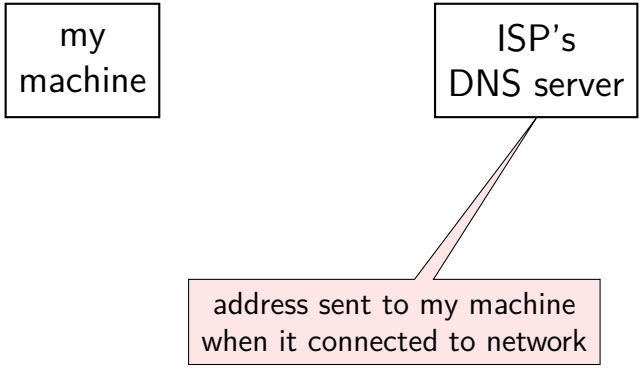
not uncommon to use RPC on one machine

more convenient alternative to pipes?

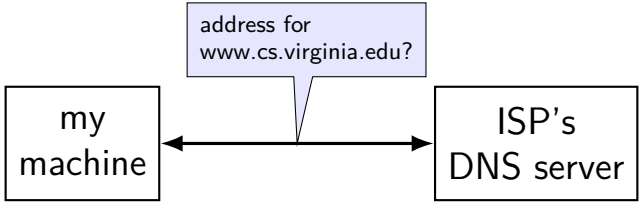
allows shared memory implementation

- mmap one common file
- use mutexes+condition variables+etc. inside that memory

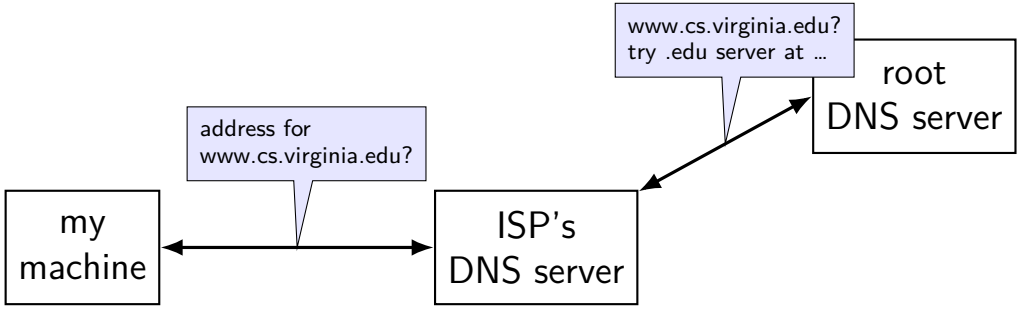
DNS: distributed database



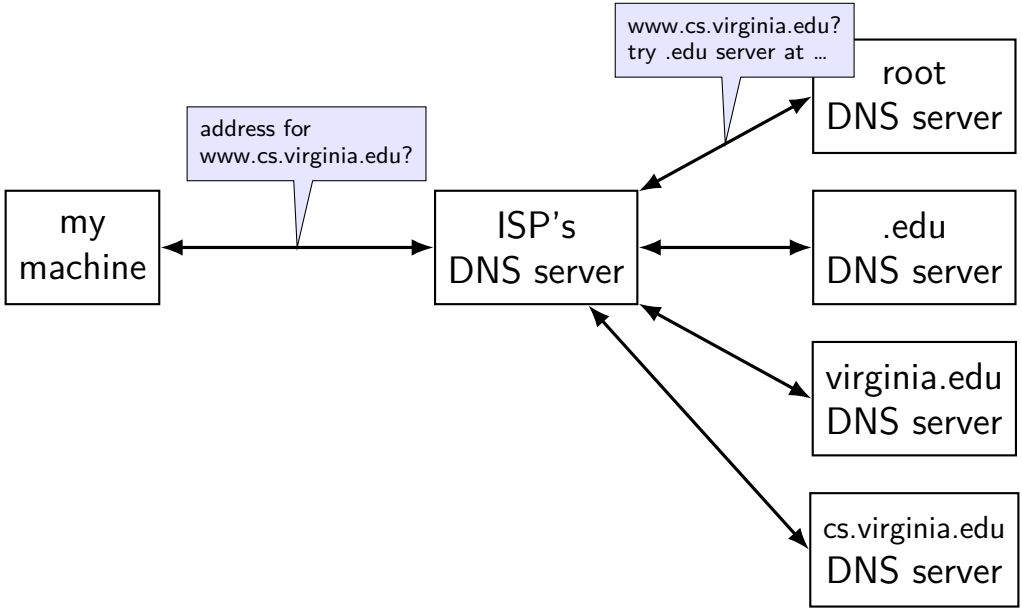
DNS: distributed database



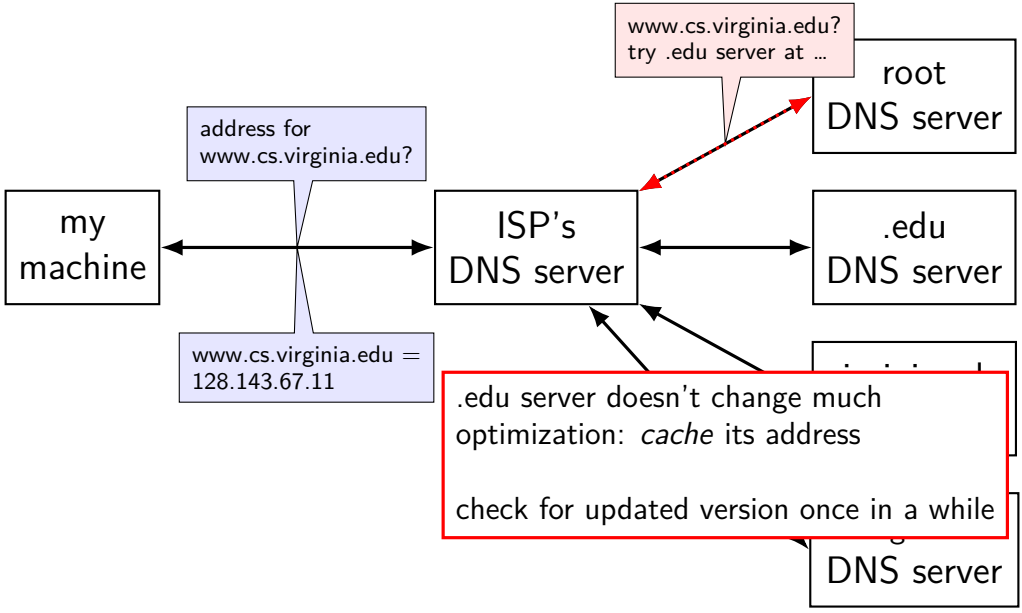
DNS: distributed database



DNS: distributed database



DNS: distributed database



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
```

lots of writing?

entire log can be written sequentially

- ideal for hard disk performance
- also pretty good for SSDs

no waiting for 'real' updates

- application can proceed while updates are happening
- files will be updated even if system crashes

often better for performance!

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);
    }
}
```

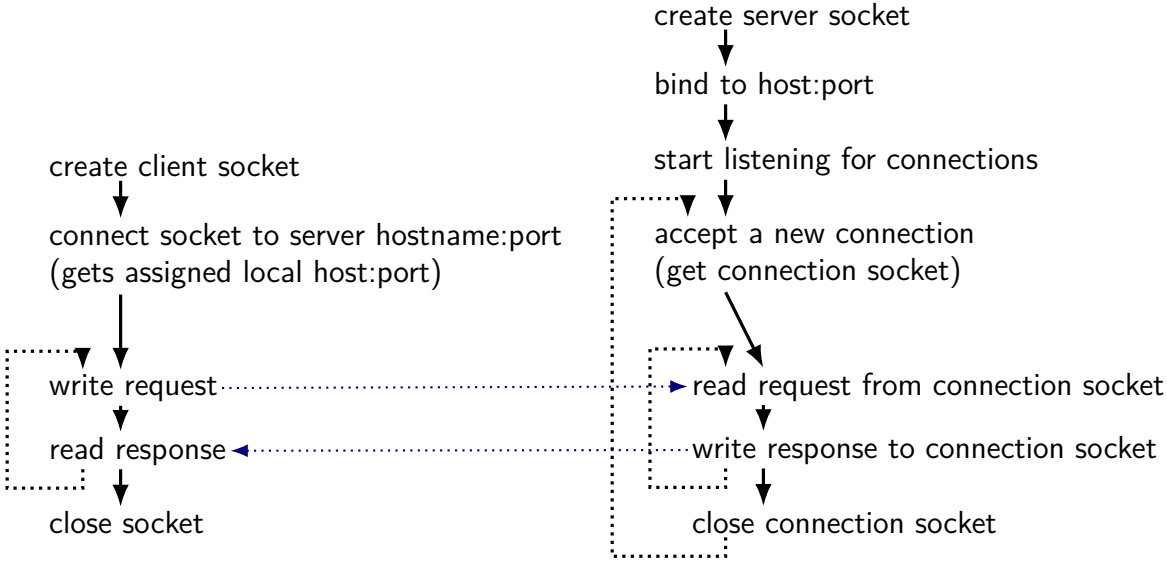
```
void server_for_connection(int socket_fd) {
    int read_count, write_count; char request_buf[MAX_SIZE];
    while (1) {
        read_count = read(socket_fd, request_buf, MAX_SIZE);
        if (read_count <= 0) return; // error or EOF
        write_count = write(socket_fd, request_buf, read_count);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

echo client/server

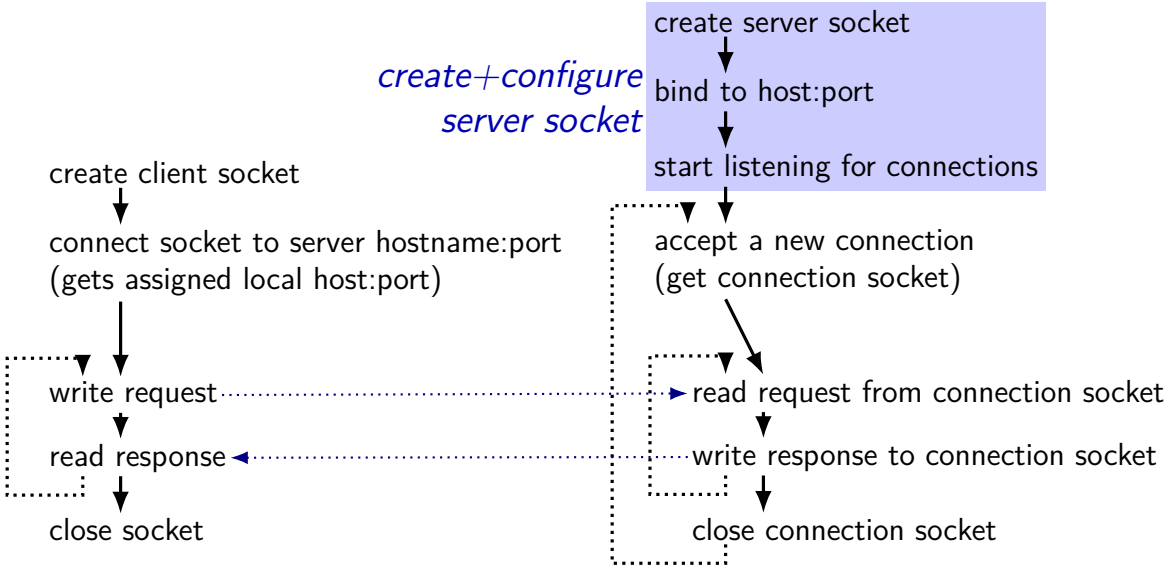
```
void client_for_connection(int socket_fd) {
    int n; char send_buf[MAX_SIZE]; char recv_buf[MAX_SIZE];
    while (prompt_for_input(send_buf, MAX_SIZE)) {
        n = write(socket_fd, send_buf, strlen(send_buf));
        if (n != strlen(send_buf)) {...error?...}
        n = read(socket_fd, recv_buf,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

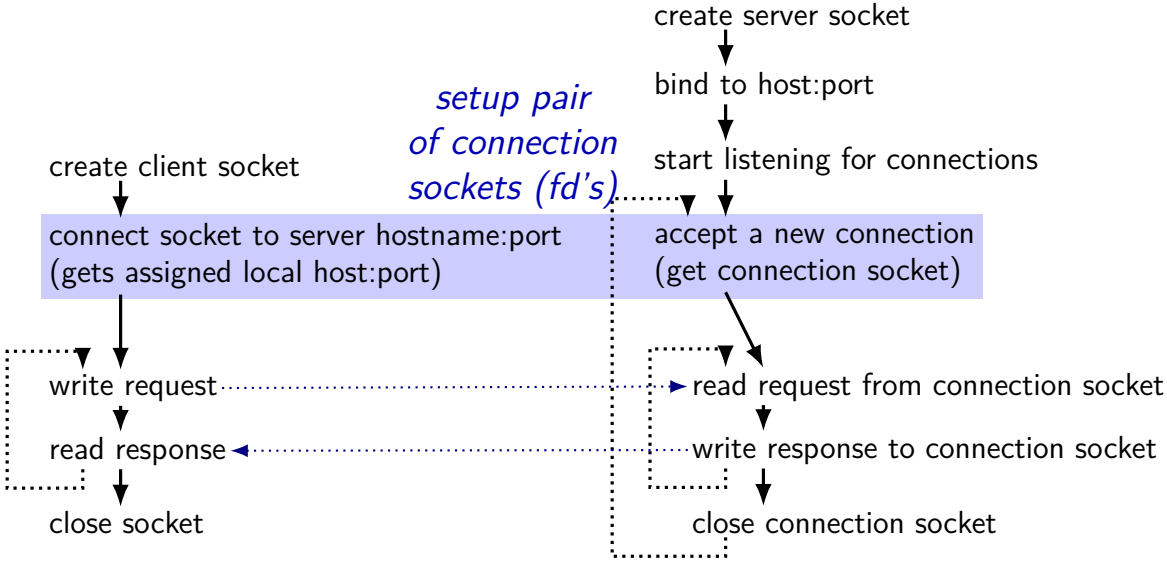
client/server flow (one connection at a time)



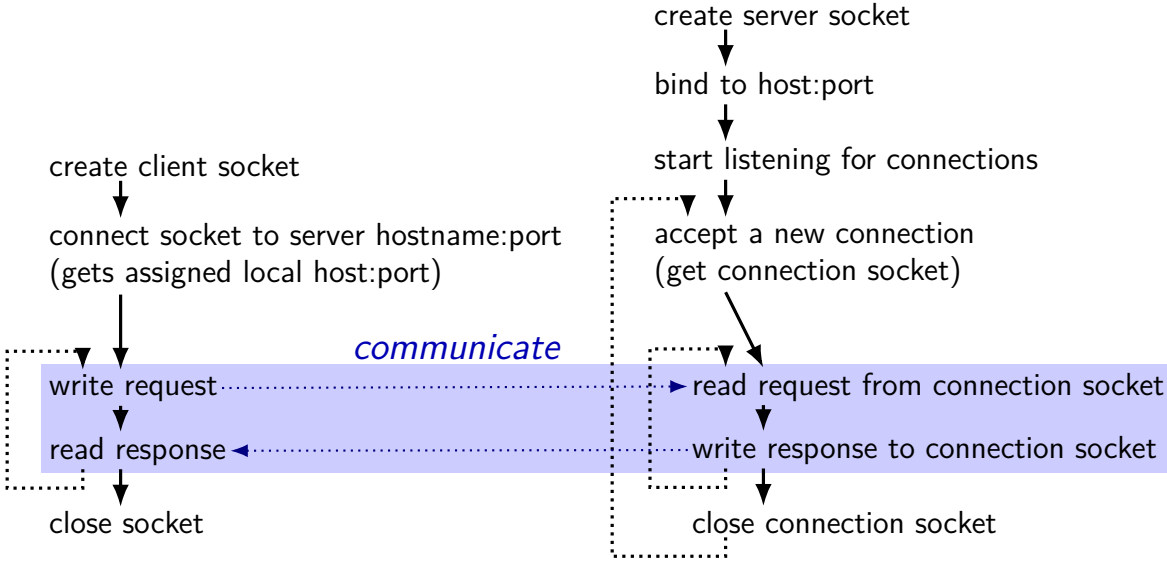
client/server flow (one connection at a time)



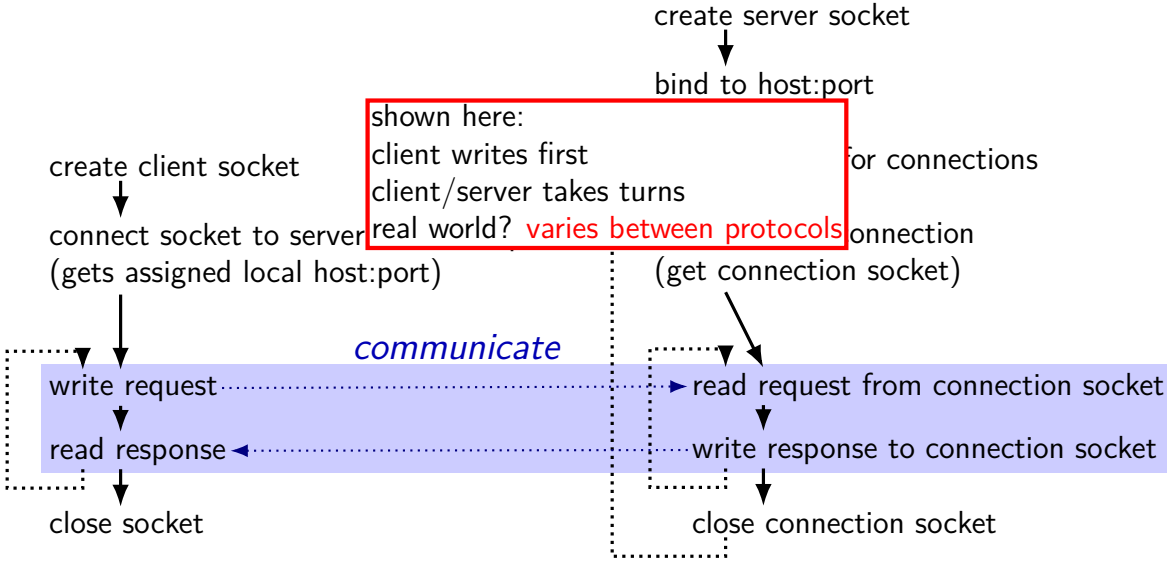
client/server flow (one connection at a time)



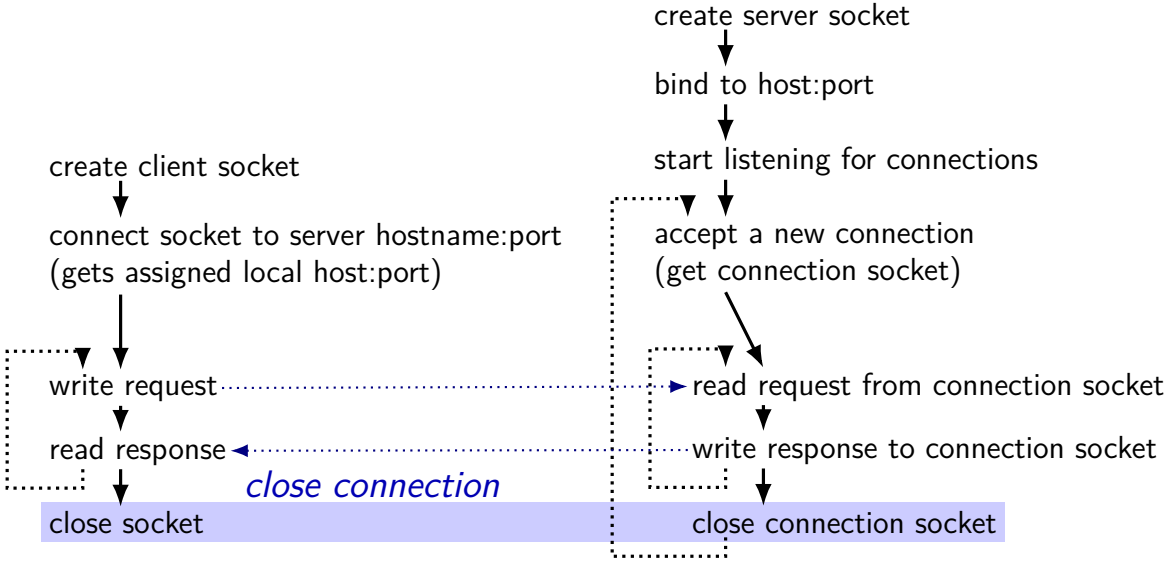
client/server flow (one connection at a time)



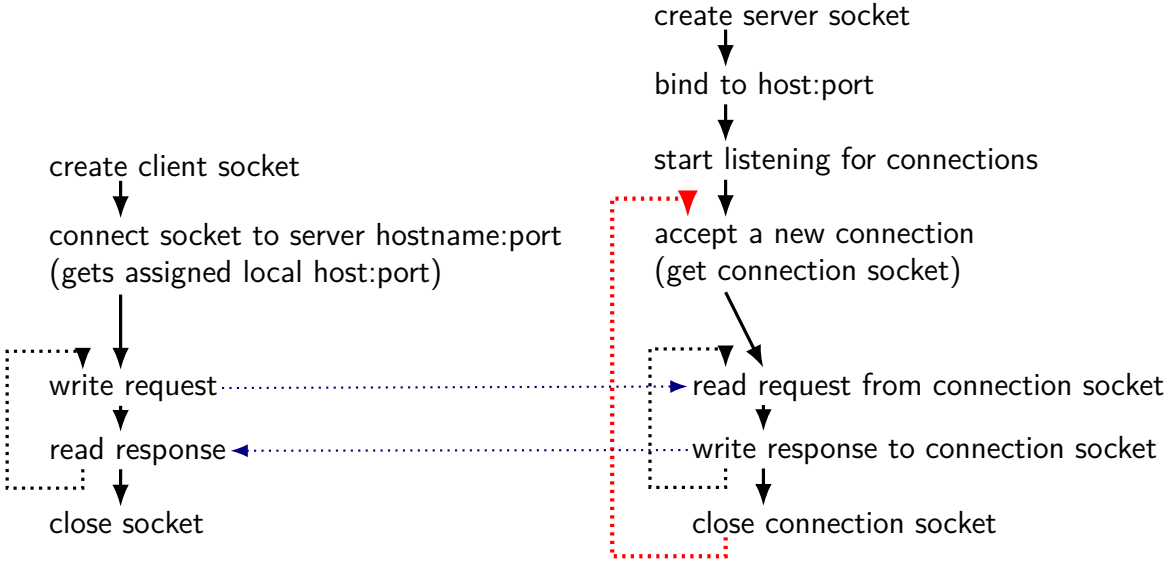
client/server flow (one connection at a time)



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
     // ai_protocol = 0 (default)
);
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_prototocol
     // 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_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);
```

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

connection setup: lookup address

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

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

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

connection setup: lookup address

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

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)

why IDL? (1)

why don't most tools use the normal source code?

alternate model: just give it a header file

why IDL? (1)

why don't most tools use the normal source code?

alternate model: just give it a header file

missing information (sometimes)

- is `char` array nul-terminated or not?
- where is the size of the array the `int*` points to stored?
- is the `List*` argument being used to modify a list or just read it?
- how should memory be allocated/deallocated?
- how should argument/function name be sent over the network?

why IDL? (2)

why don't most tools use the normal source code?

alternate model: just give it a header file

machine-neutrality and language-neutrality

- common goal: call server from any language, any type of machine
- how big should `long` be?
- how to pass string from C to Python server?

why IDL? (2)

why don't most tools use the normal source code?

alternate model: just give it a header file

machine-neutrality and language-neutrality

- common goal: call server from any language, any type of machine
- how big should `long` be?
- how to pass string from C to Python server?

versioning/compatibility

- what should happen if server has newer/older prototypes than client?