last time

inode-based filesystem

- inode: all info about files except name
- directory entires specify index in inode array

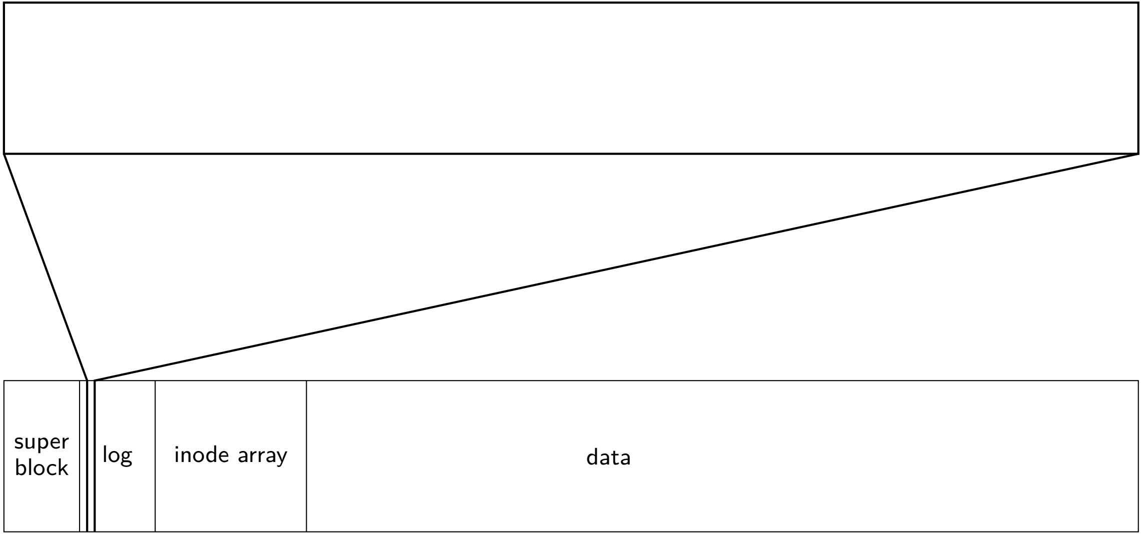
direct and indirect and double-indirect...pointers to data blocks

- small files: only direct pointers in inode array
- indirect pointers to blocks for b

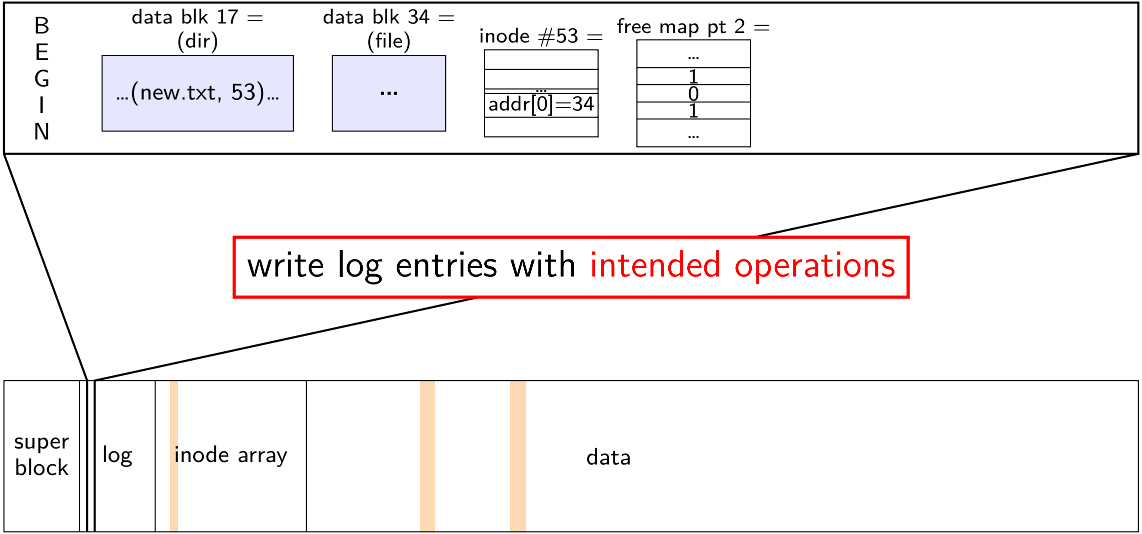
extents

- store (4, 5) instead of 4, 5, 6, 7, 8, 9
- problem: need to allocate contiguous sets of blocks

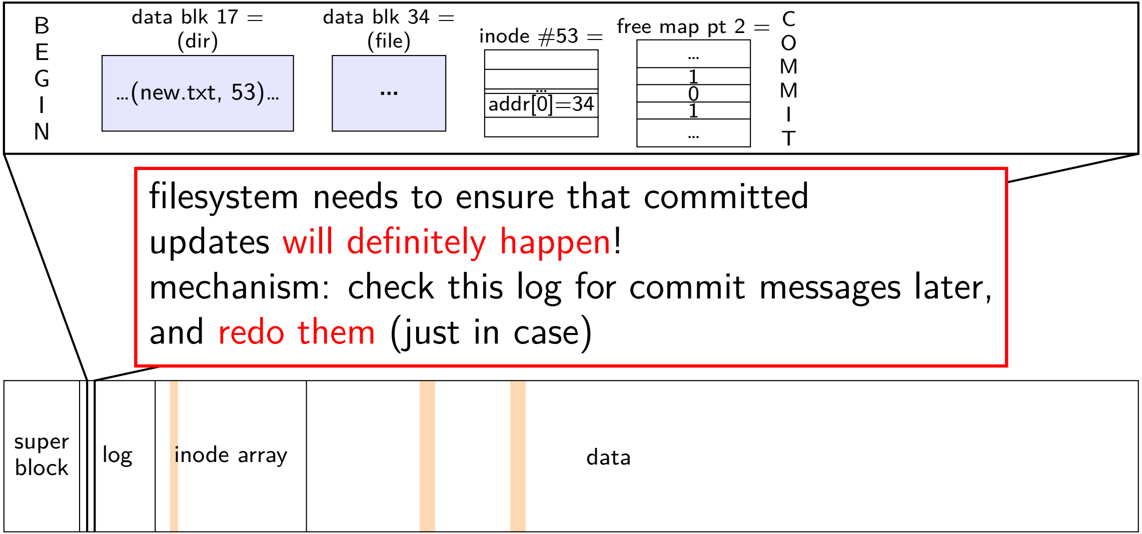
redo logging: file creation



redo logging: file creation

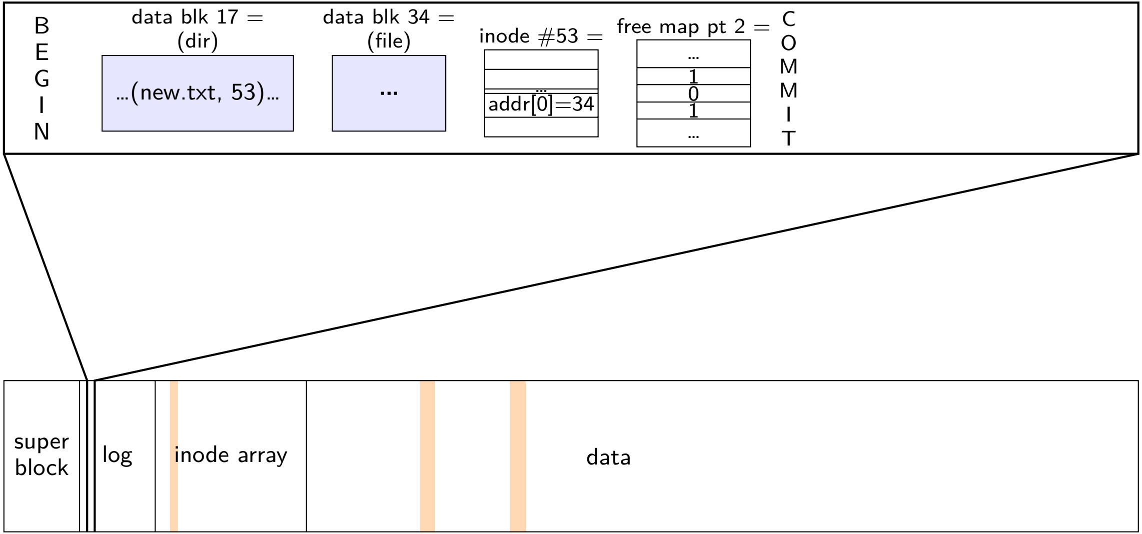


redo logging: file creation

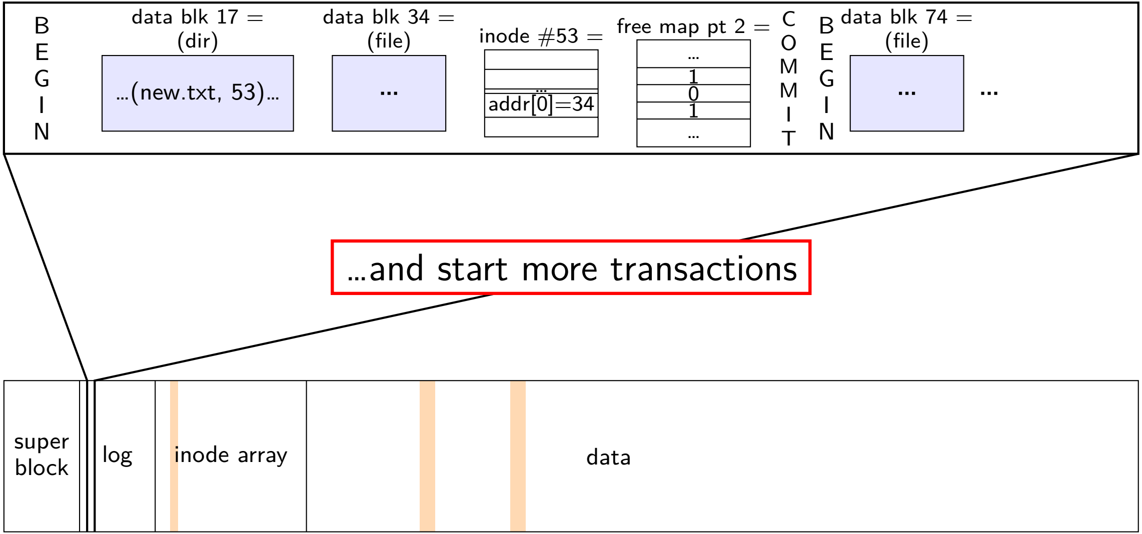


filesystem needs to ensure that committed updates will definitely happen!
mechanism: check this log for commit messages later, and redo them (just in case)

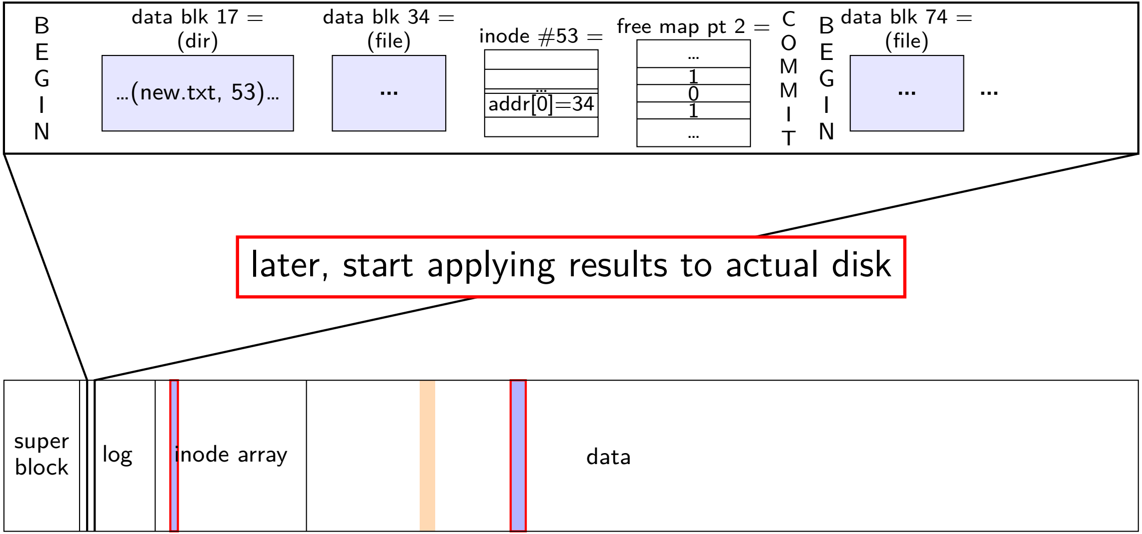
redo logging: file creation



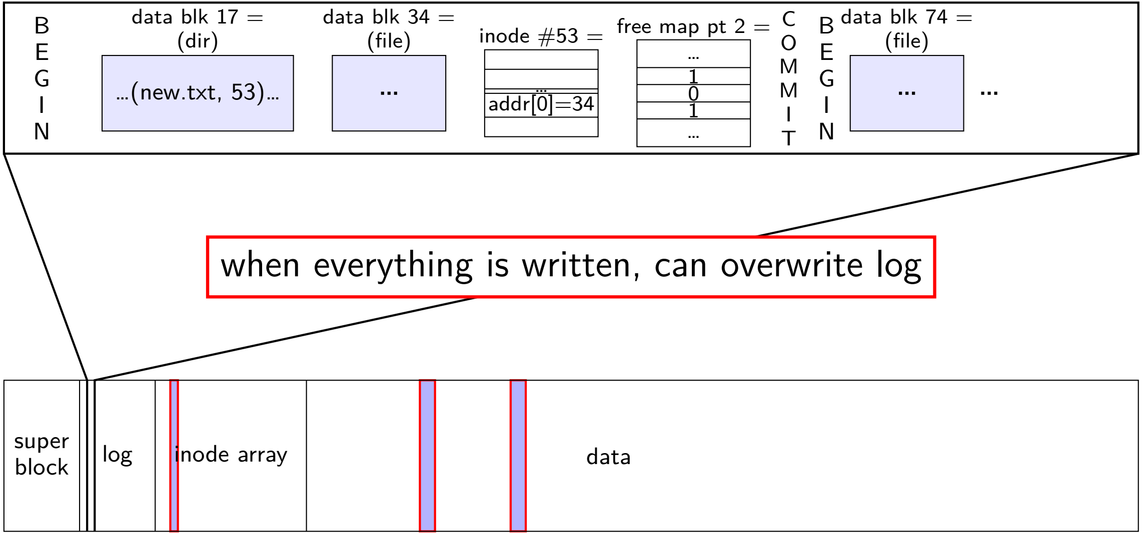
redo logging: file creation



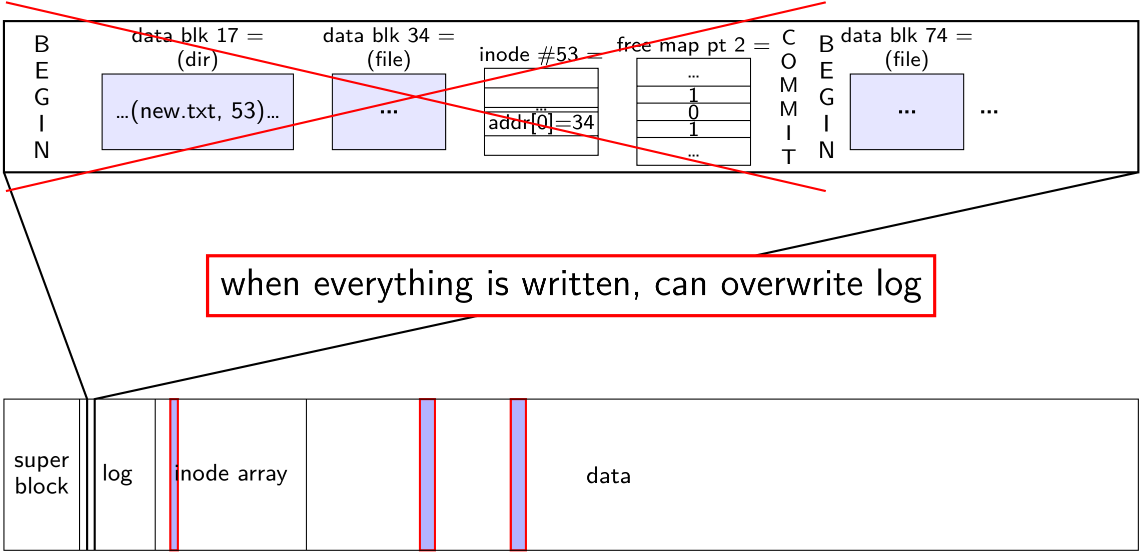
redo logging: file creation



redo logging: file creation



redo logging: file creation



redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

redo logging: file creation

normal operation

write to log transaction steps:

data blocks to create
direcotry entry, inode to write
directory inode (size, time)
update

~~write to log "commit transaction"~~

in any order:

update file data blocks
update directory entry
update file inode
update directory inode

reclaim space in log

"garbage collection"

crash before *commit*?
file not created
no partial operation to real data

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

crash after *commit*?

file created

promise: will perform logged updates

(after system reboots/recovers)

redo logging: file creation

normal operation

write to log transaction steps:

- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log "commit transaction"

in any order:

- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log

- "garbage collection"

redo logging: file creation

normal operation

write to log transaction steps:
- data blocks to create
- direcotry entry, inode to write
- directory inode (size, time) update

write to log “commit transaction”

in any order:
- update file data blocks
- update directory entry
- update file inode
- update directory inode

reclaim space in log
- “garbage collection”

recovery

read log and...

ignore any operation with no “commit”

redo any operation with “commit”
- already done? — okay, setting inode twice

reclaim space in log

idempotency

logged operations should be *okay to do twice* = *idempotent*

good example: set inode link count to 4

bad example: increment inode link count

good example: overwrite inode number X with new value

as long as last committed inode value in log is right...

bad example: allocate new inode with particular contents

good example: overwrite data block with new value

bad example: append data to last used block of file

redo logging summary

write intended operation to the log

- before ever touching 'real' data
- in format that's safe to do twice

write marker to commit to the log

- if exists, the operation *will be done eventually*

actually update the real data

redo logging and filesystems

filesystems that do redo logging are called *journalling filesystems*

exercise (1)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging, ext2-like filesystem with 1KB blocks, 4B block pointers

part 1: what's modified?

[A] free block map
[B] data blocks for file
[C] indirect blocks for file
[D] data blocks for directory
[E] inode for file
[F] inode for directory
[G] the log

exercise (2)

suppose OS performing operation of appending 100KB to a 100KB file X in directory Y and uses redo logging

part 2: crash happens after writing:

- log entries for entire operation
- free block map changes
- indirect blocks for file

...what is written after restart as part of this operation?

- [A] free block map
- [B] data blocks for file
- [C] indirect blocks for file
- [D] data blocks for directory
- [E] inode for file
- [F] inode for directory
- [G] the log

degrees of consistency

not all journalling filesystem use redo logging for everything

some use it *only for metadata operations*

some use it *for both metadata and user data*

only metadata: avoids lots of duplicate writing

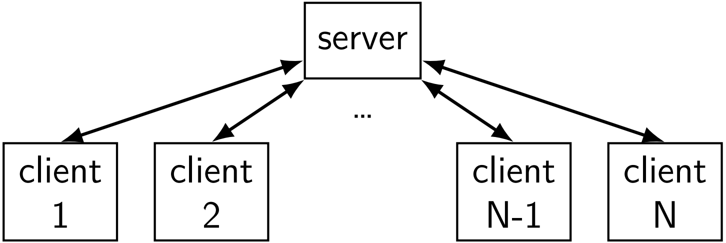
metadata+user data: integrity of user data guaranteed

distributed systems

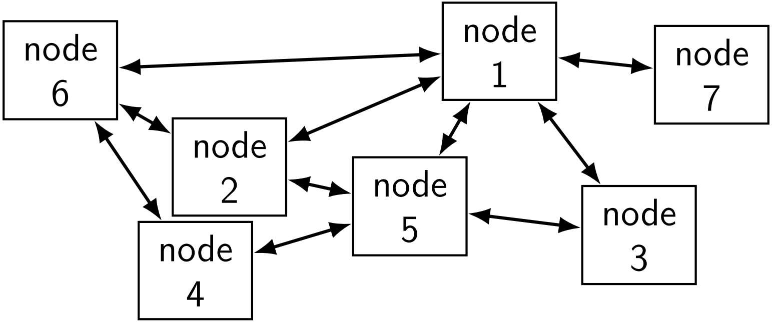
multiple machines working together to perform a single task

called a *distributed system*

some distibuted systems models

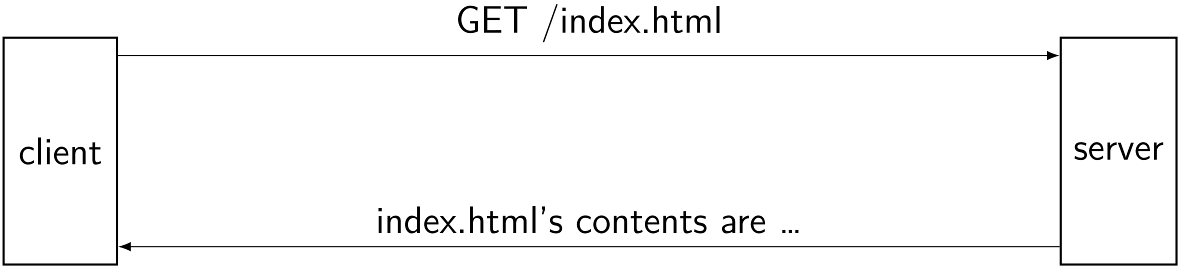


client/server

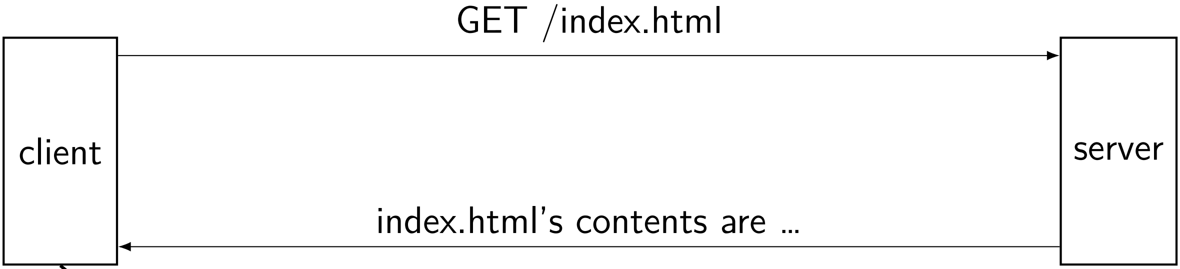


peer-to-peer

client/server model



client/server model

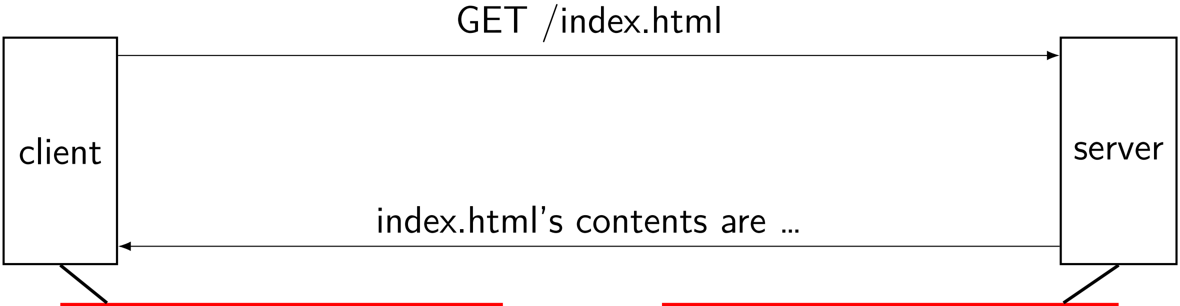


client(s): "sometimes on"

sends requests to server(s)

needs to know
how to contact server

client/server model



client(s): "sometimes on"

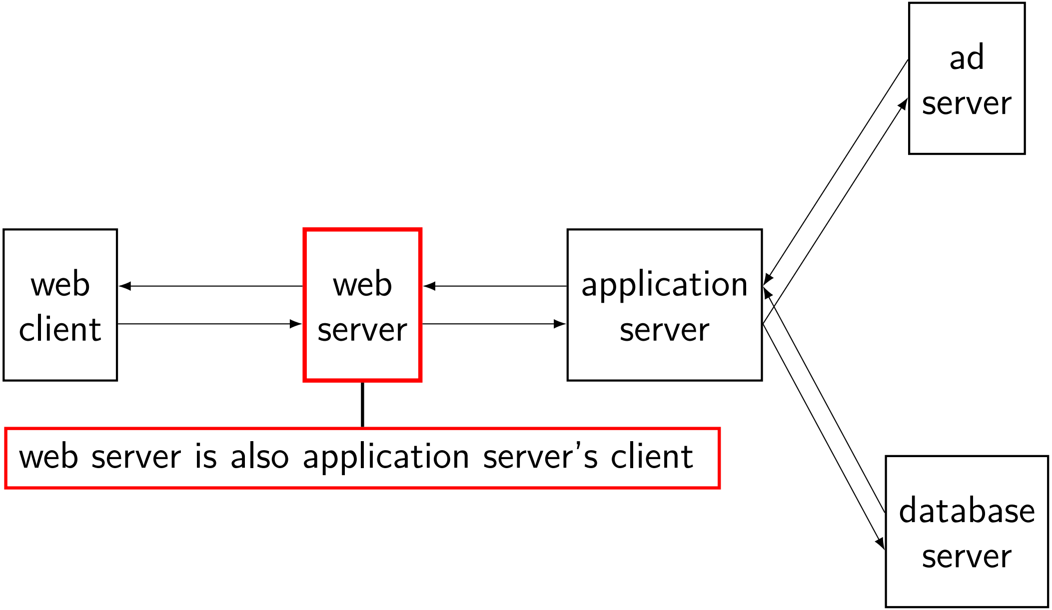
sends requests to server(s)

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

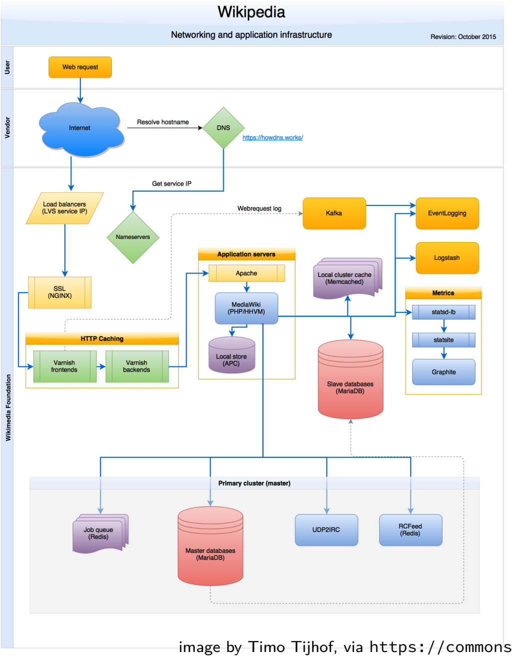


image by Timo Tijhof, via `https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png`

example: Wikipedia architecture (zoom)

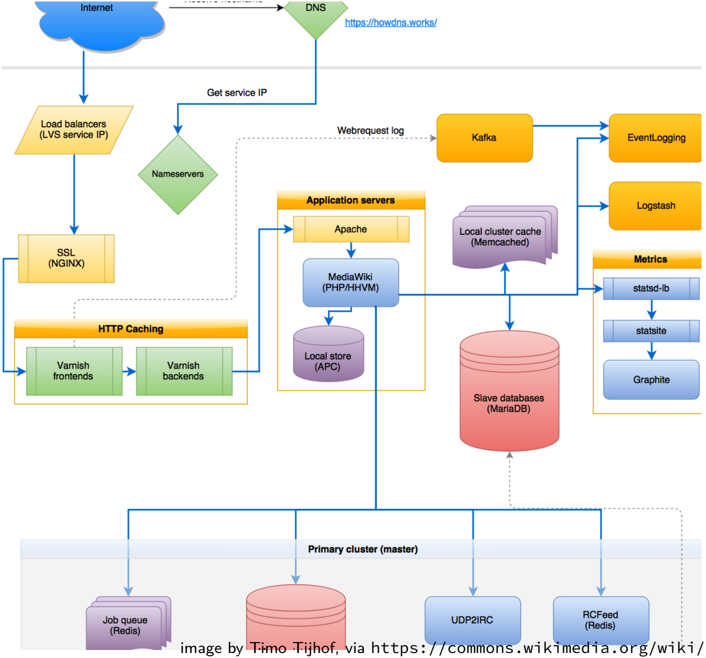


image by Timo Tijhof, via https://commons.wikimedia.org/wiki/File:Wikipedia_webrequest_flow_2015-10.png

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"

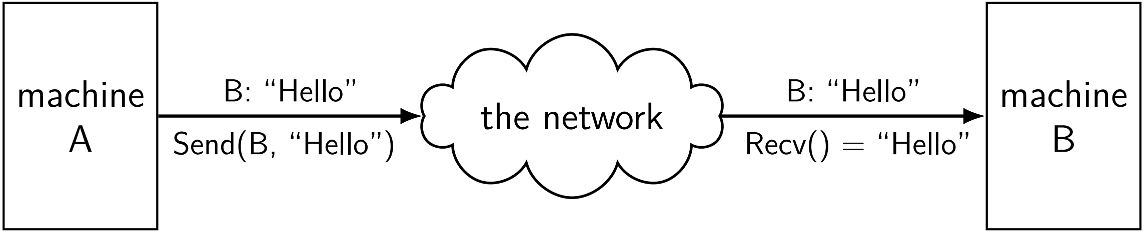
combine many cheap machines to replace expensive machine

easier to add incrementally

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

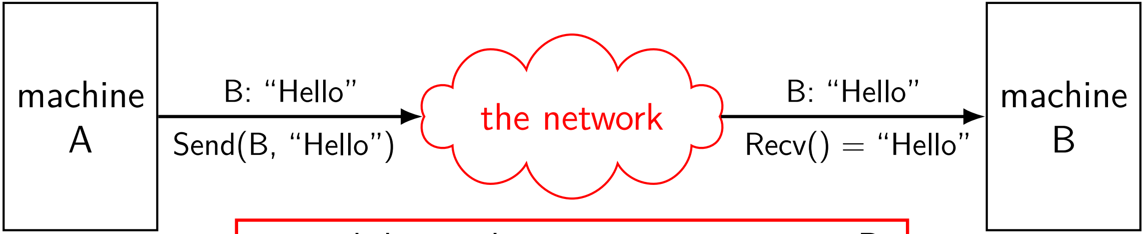
mailbox model

mailbox abstraction: send/receive messages



mailbox model

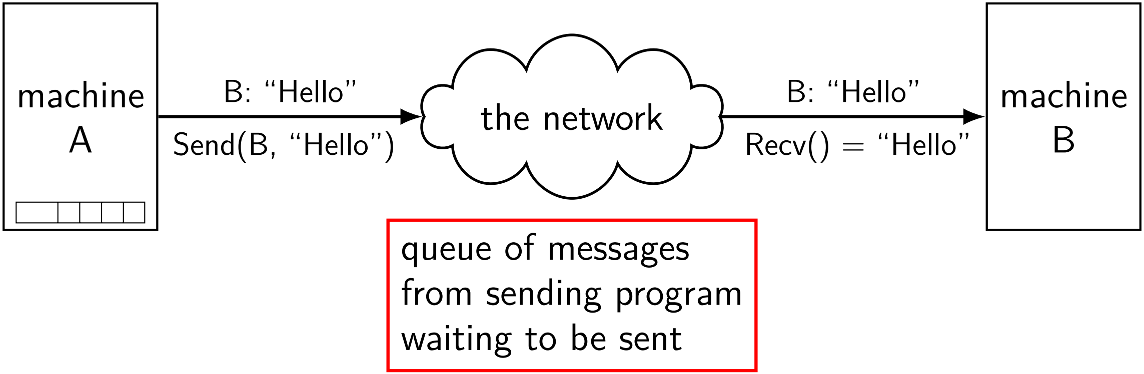
mailbox abstraction: send/receive messages



network knows how to get message to B

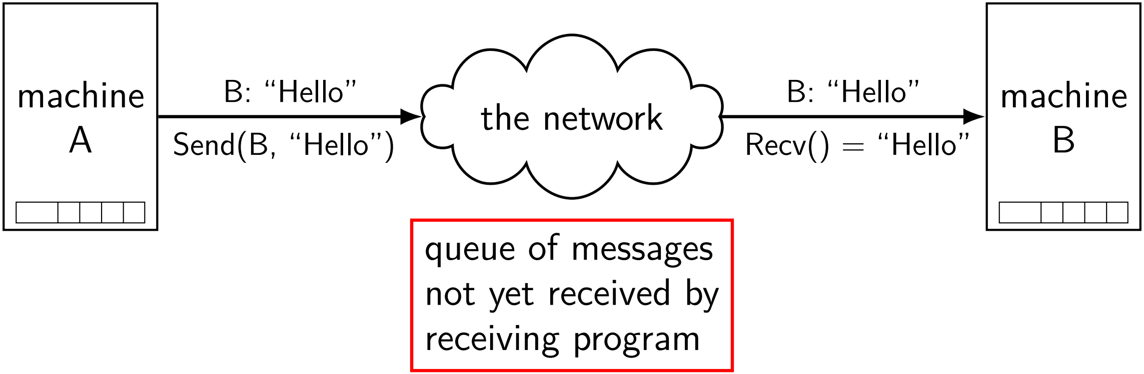
mailbox model

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

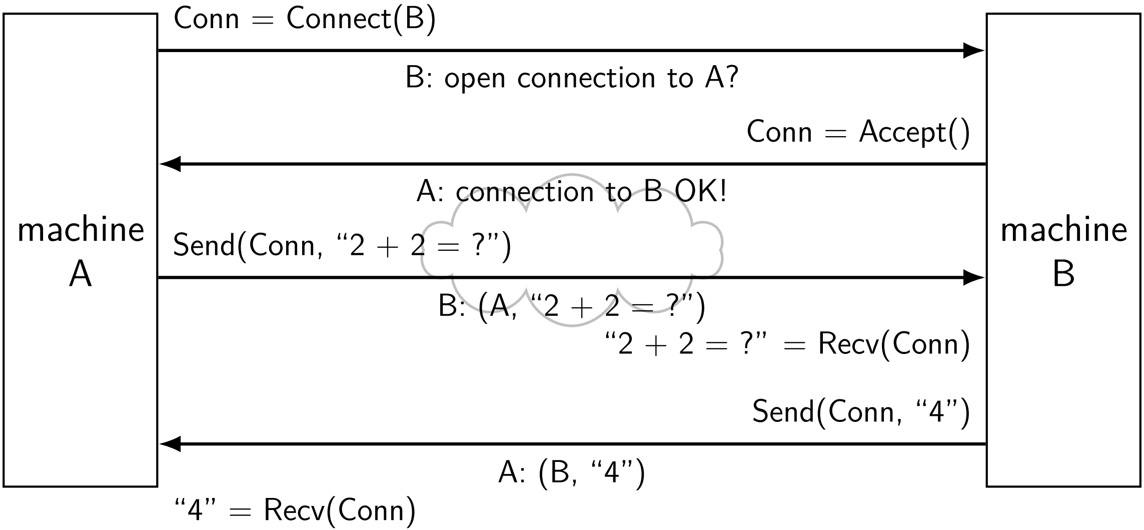
- send a 'return address'
- need to track related messages

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)

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`

connection missing pieces?

how to specify the machine?

multiple programs on one machine? who gets the message?

IPv4 addresses

32-bit numbers

typically written like 128.143.67.11

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

organizations get blocks of IPs

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

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

IPv6 addresses

IPv6 like IPv4, but with 128-bit numbers

written in hex, 16-bit parts, 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`

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

selected special IPv6 addresses

`::1` = localhost

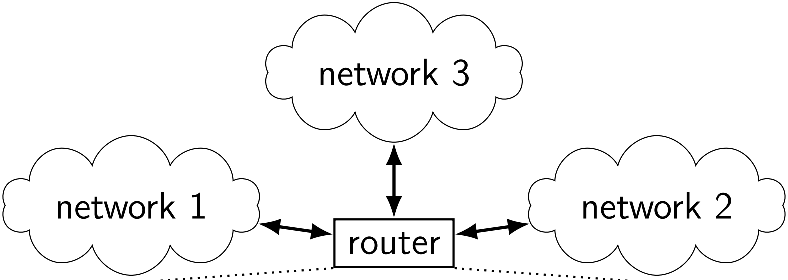
anything starting with `fe80` = link-local addresses

never forwarded by routers

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

sockets

socket: POSIX abstraction of network I/O queue

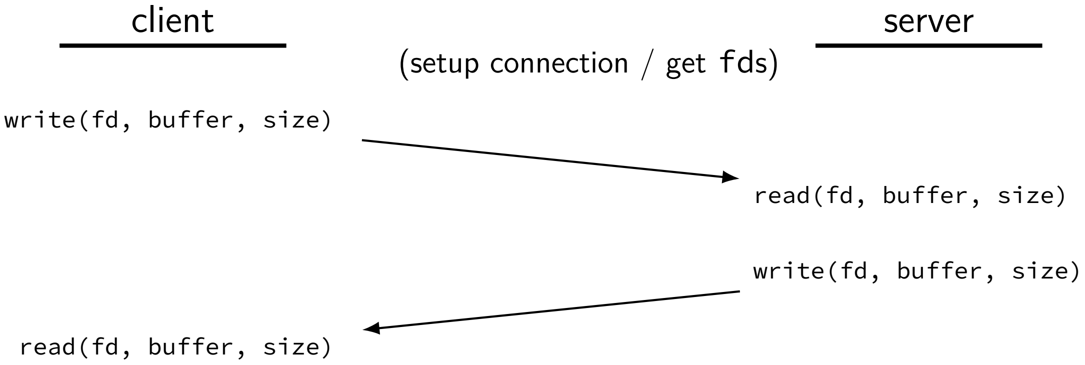
- any kind of network
- can also be used between processes on same machine

a kind of file descriptor

connected sockets

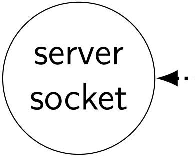
sockets can represent a connection

act like bidirectional pipe

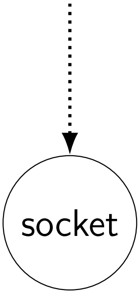


sockets and server sockets

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



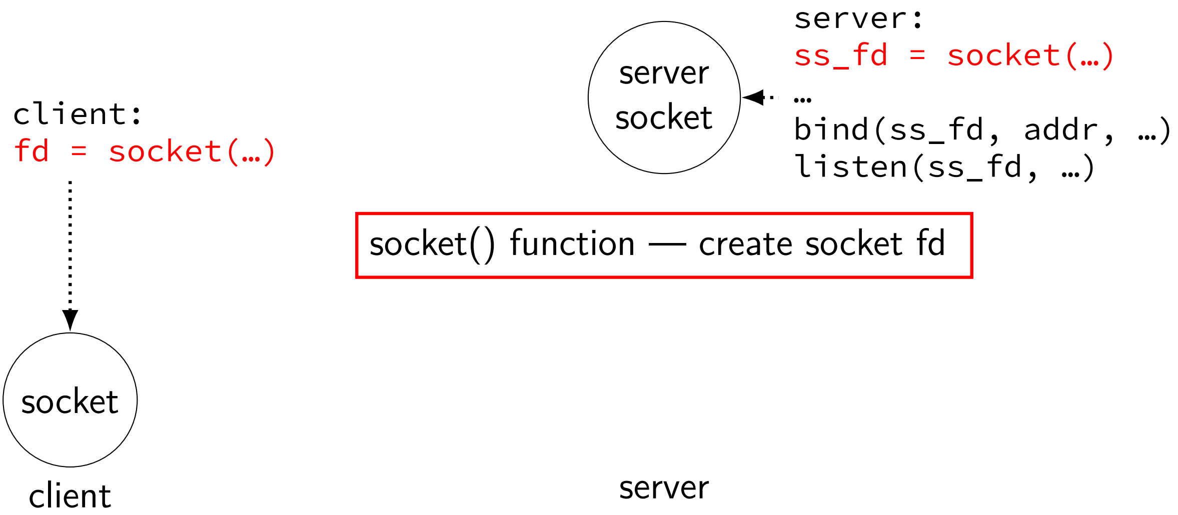

```
client:
fd = socket(…)
```



client

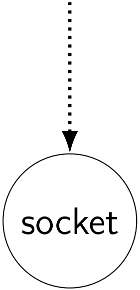
server

sockets and server sockets

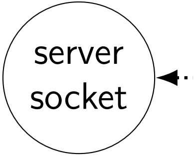


sockets and server sockets

```
client:
fd = socket(…)
```



client

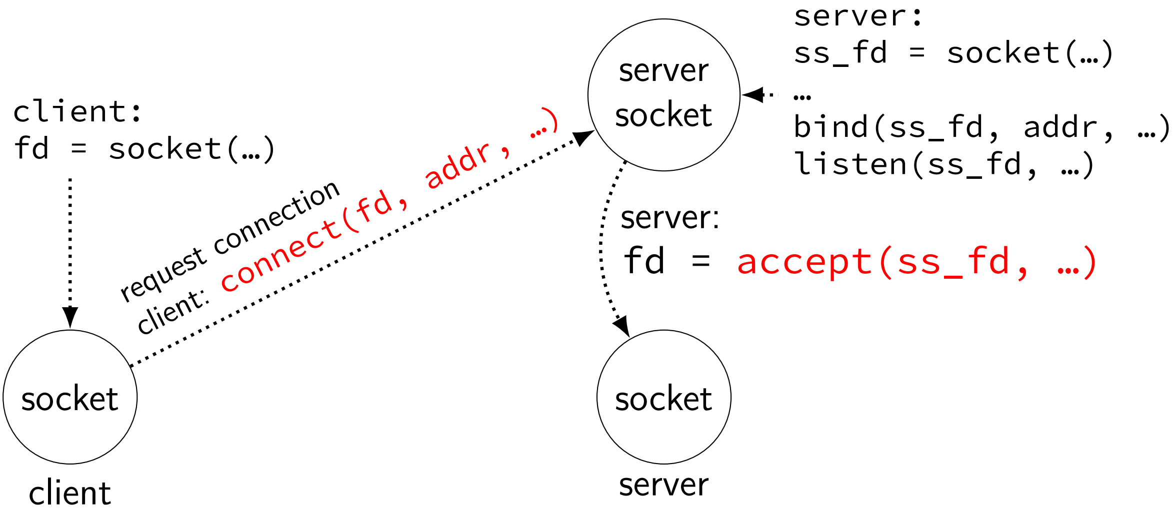



```
server:
ss_fd = socket(…)
…
bind(ss_fd, addr, …)
listen(ss_fd, …)
```

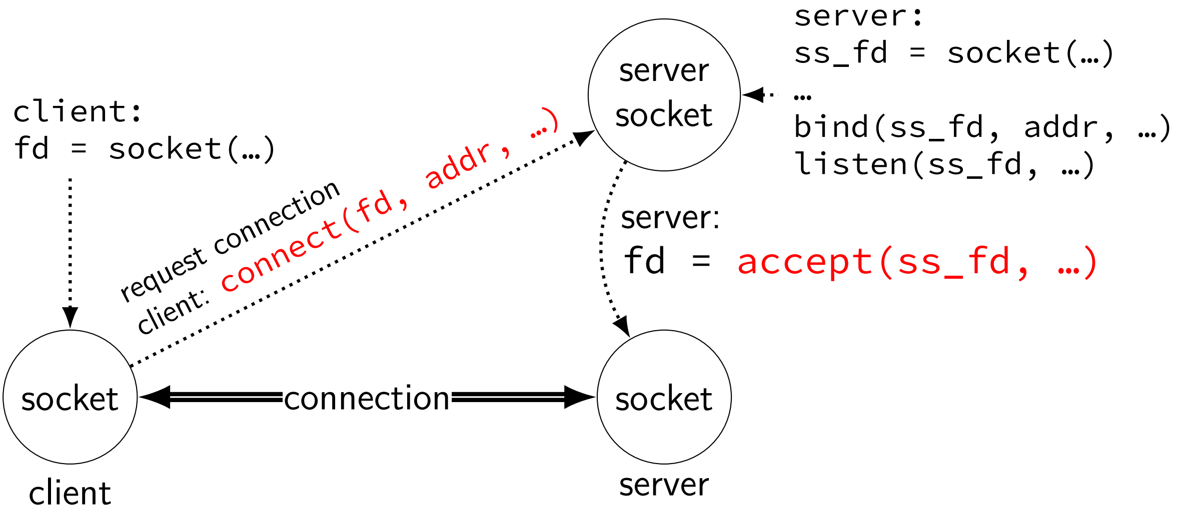
listen() — turn socket into server socket
still has a file descriptor, but …
can only `accept()` — create normal socket

server

sockets and server sockets



sockets and server sockets



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

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*

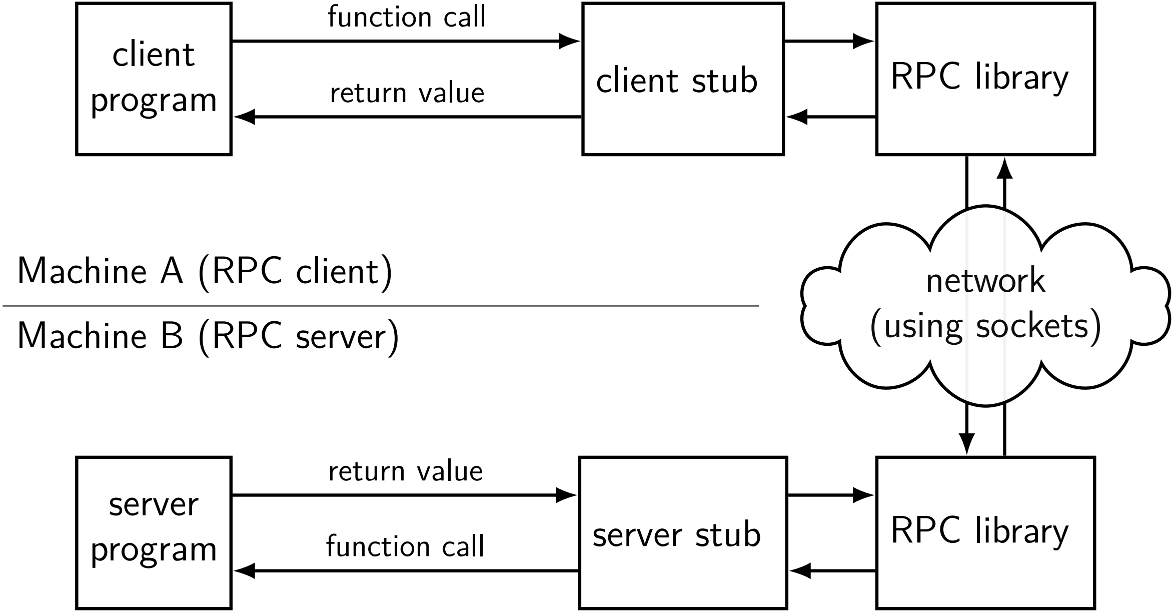
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

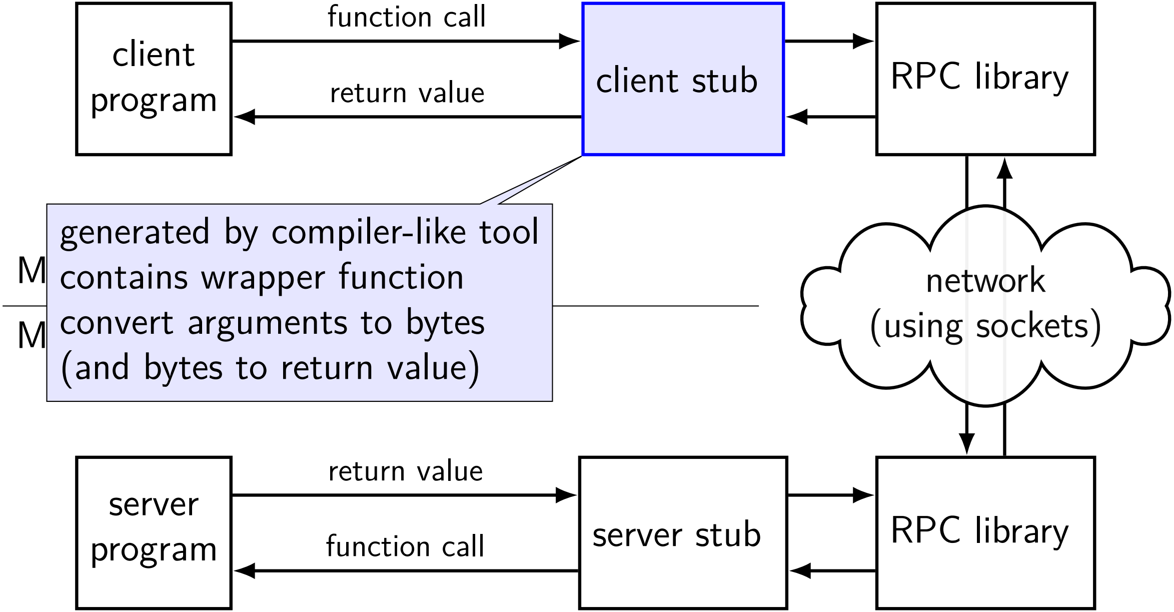
- same prototype are remote procedure

implementing remote procedure? a stub function calls you

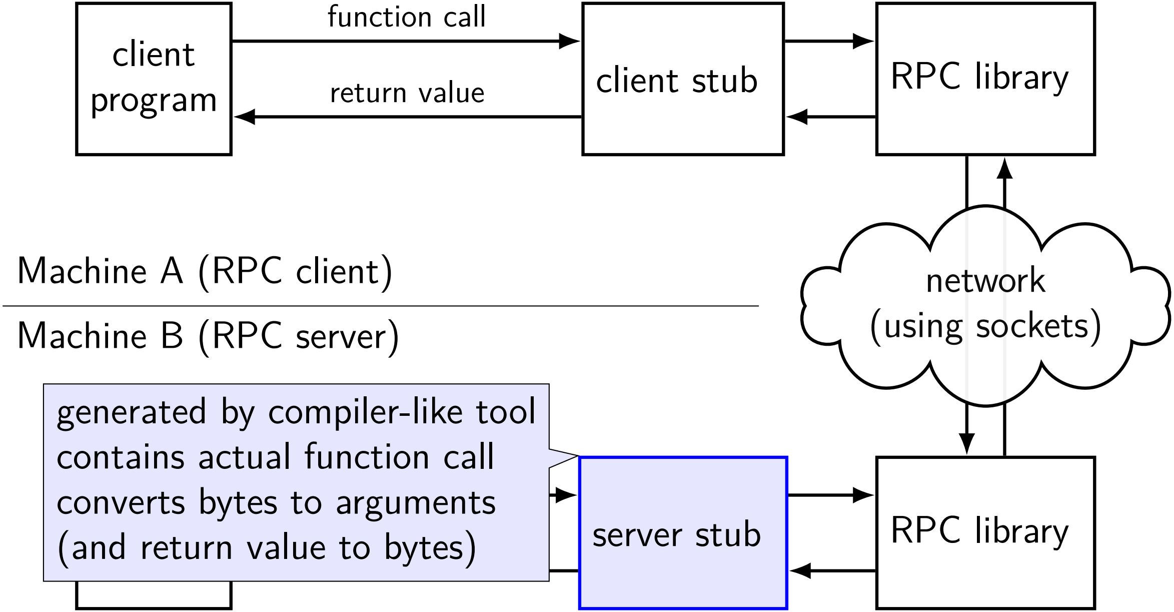
typical RPC data flow



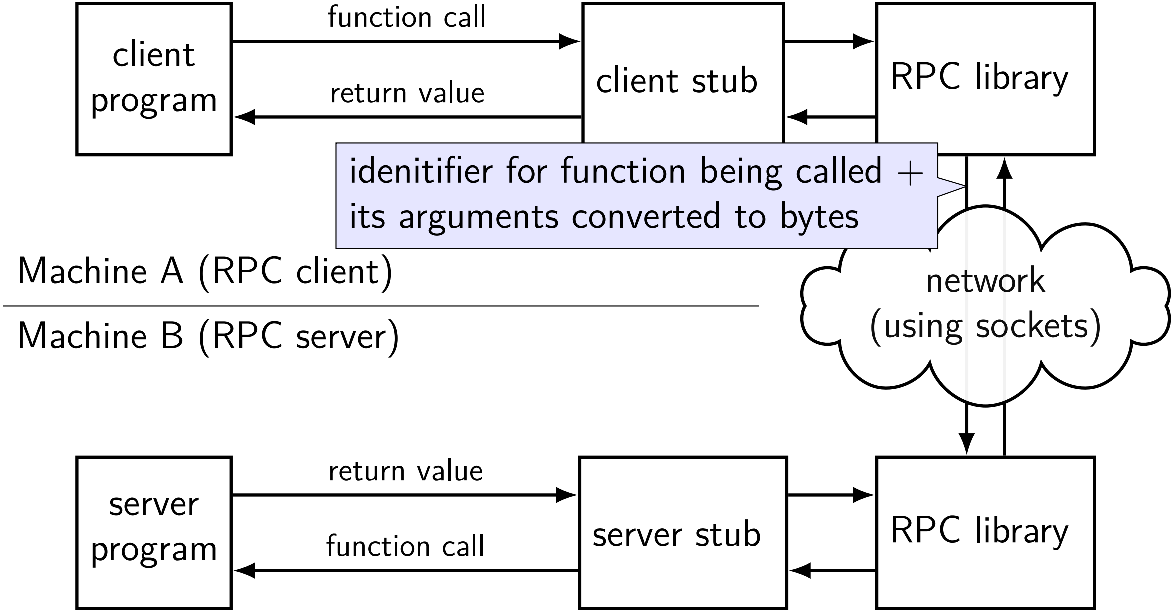
typical RPC data flow



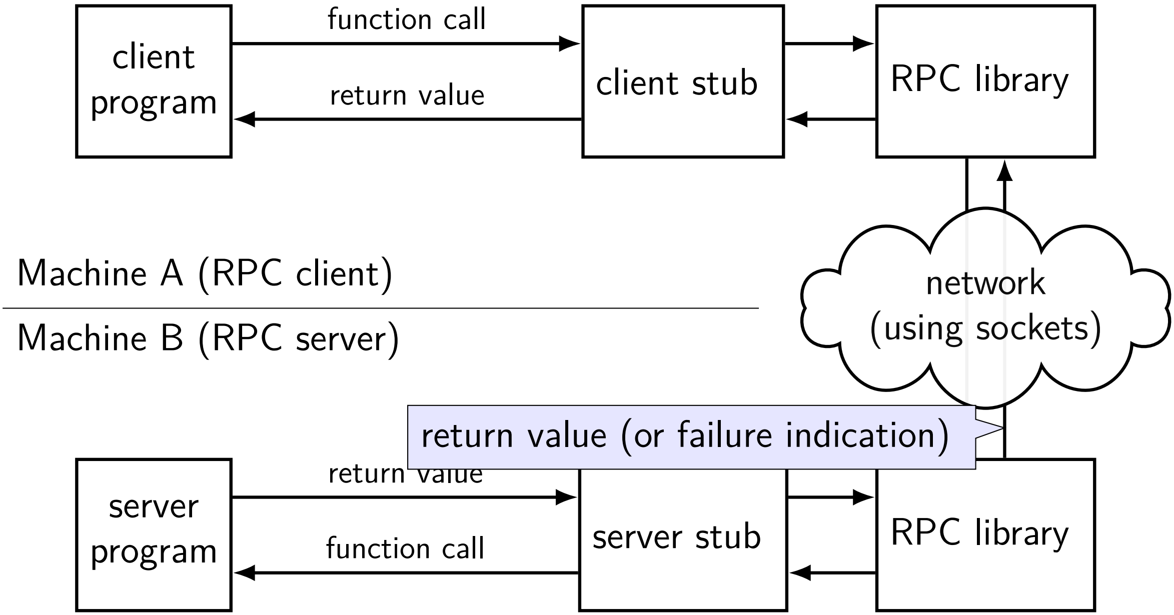
typical RPC data flow



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

client:

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

server: defines "MakeDirectory" function

local-only code would have been:

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

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.mk
    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 [
    rpc [
    rpc [                                        ) {}
}
```

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 Di
    rpc Ma
    rpc Li                                                    ) {}
}
```

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

rule: arguments/return value always a *message*

```
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) {}
}
```

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

backup slides

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"

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

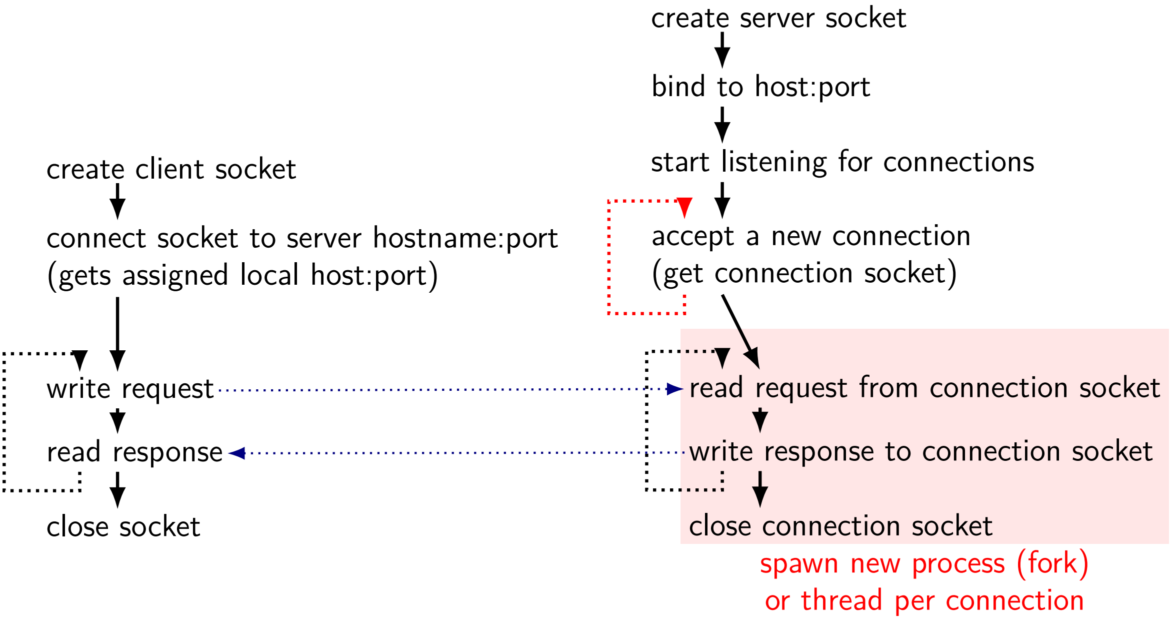
SCP/SFTP: secure copy/secure file transfer — built on SSH

HTTPS: HTTP, but over TLS instead of TCP

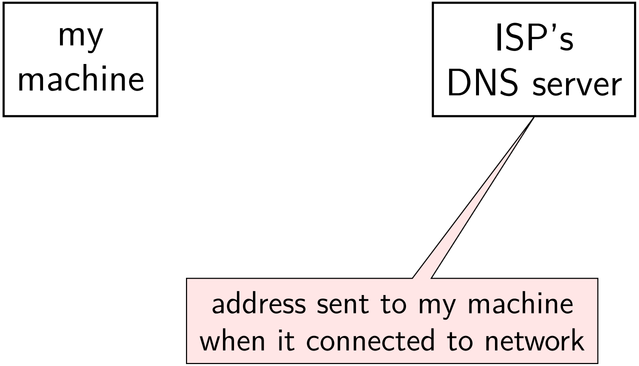
FTP: file transfer protocol

…

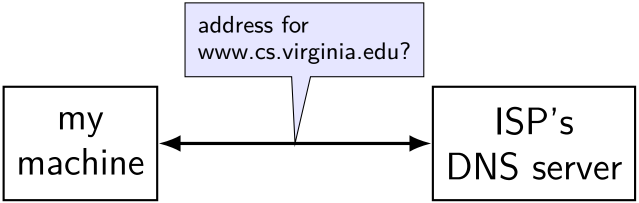
client/server flow (multiple connections)



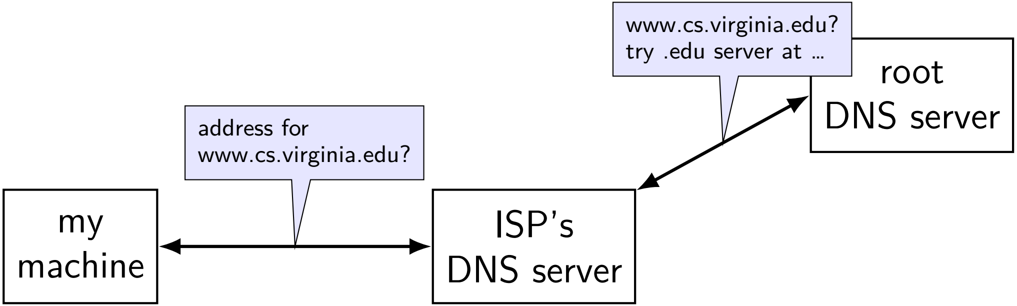
DNS: distributed database



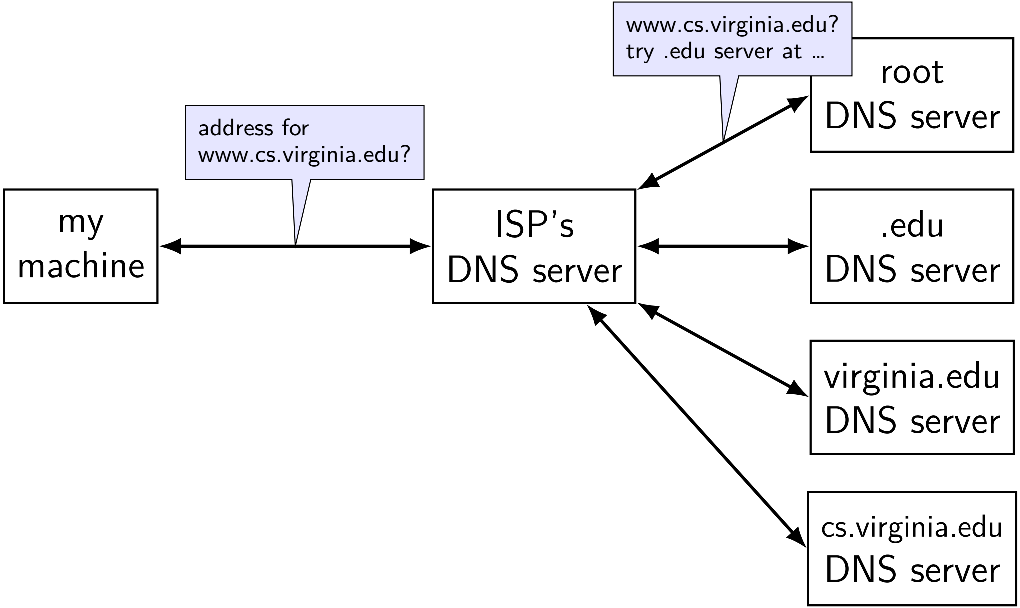
DNS: distributed database



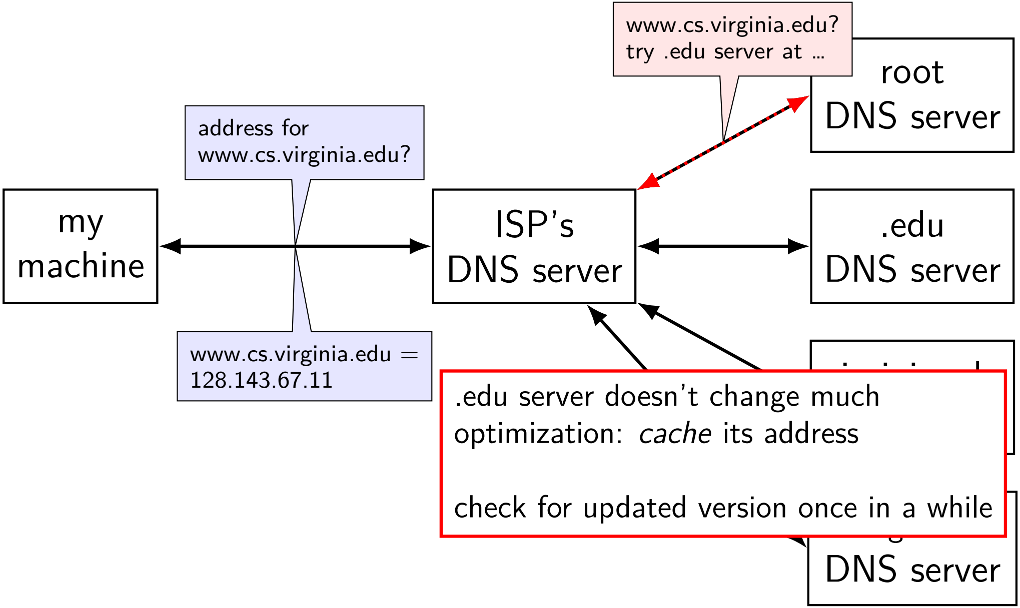
DNS: distributed database



DNS: distributed database



DNS: distributed database



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

check for updated version once in a while

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
```

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?