changelog

19 April 2022: ‘exercise: RPC failure scenarios’: clarify that we give up if waiting too long

21 April 2022: gRPC examples: make port numbers consistent

last time (1)

redo logging (finish)

- log first, then commit, then do actual operations
- promise: if committed, will do actual operations (redo if needed)
- log operations need to be idempotent (safe to do extra times)

reasons for distribution — social, technical

client/server model

- clients (sometimes on) contact servers (always on)
- sometimes chains of client/server relationships

last time (2)

mailbox model and routing

- names v addresses
- IP address = machine; port number = program on machine

connections $\approx$ two-way pipes built atop mailbox model

- POSIX representation: socket file descriptors

remote procedure calls

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*

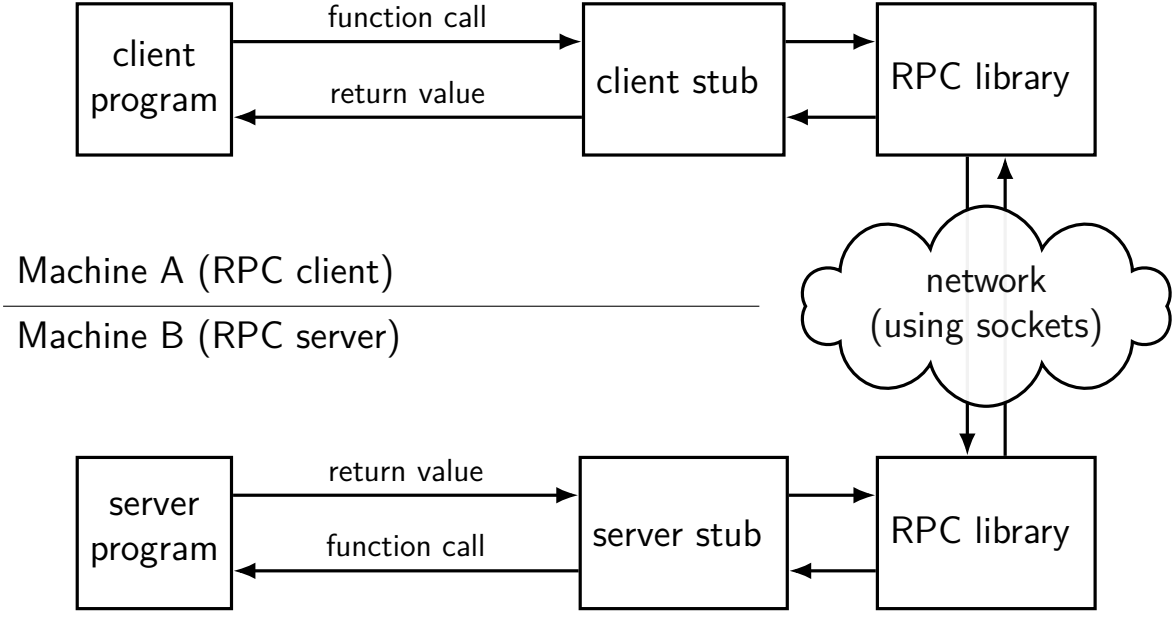
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

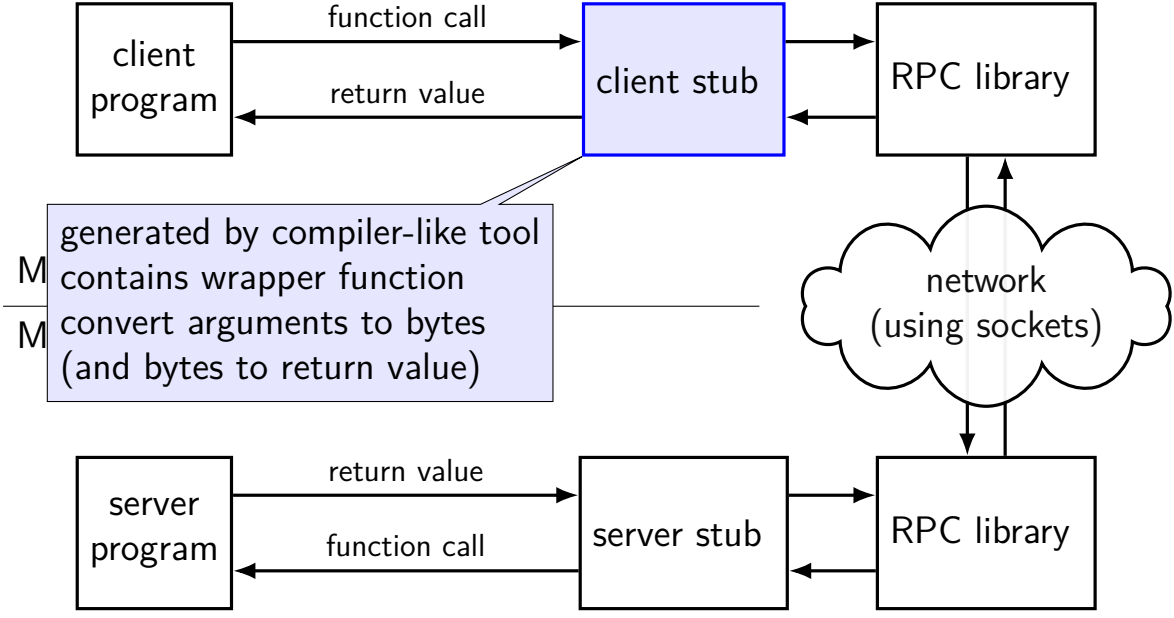
- same prototype are remote procedure

implementing remote procedure? a stub function calls you

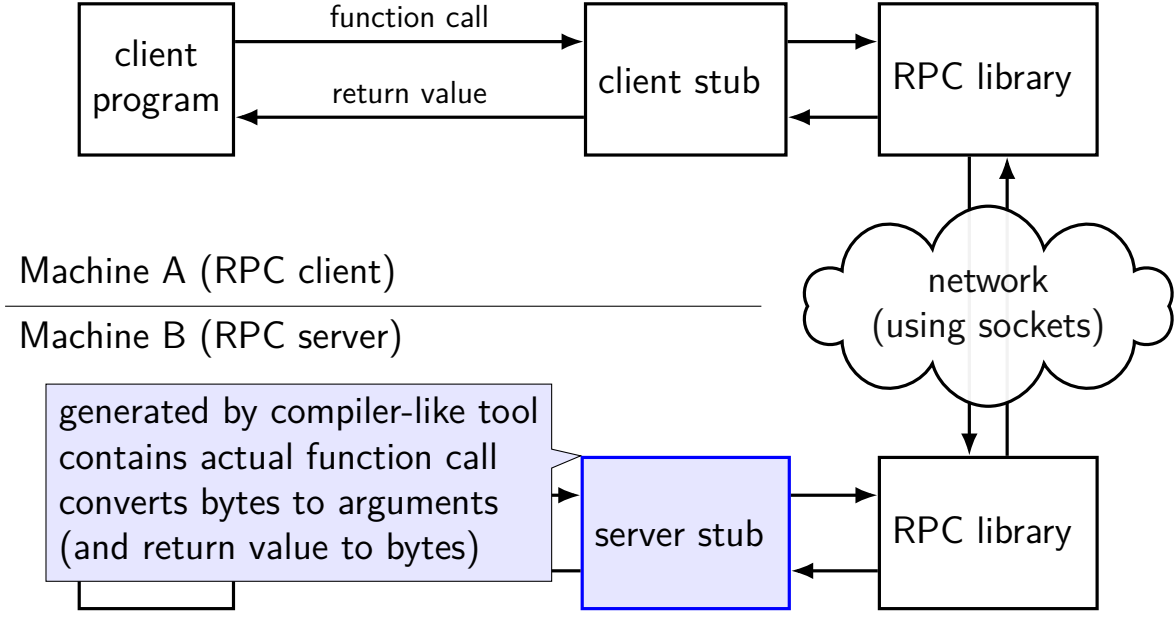
typical RPC data flow



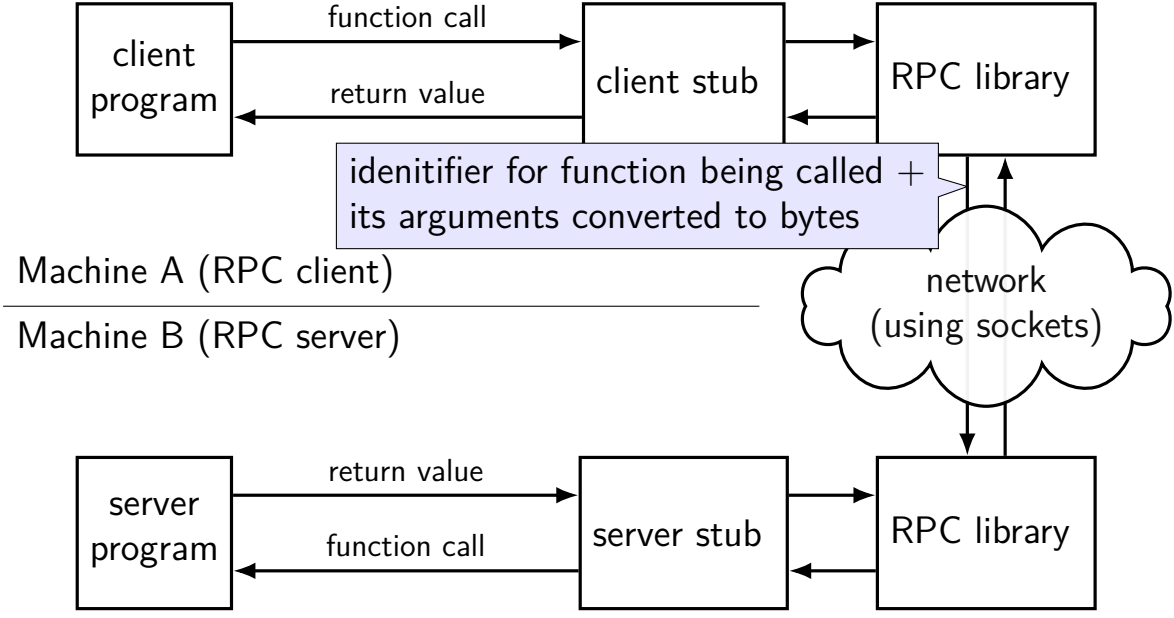
typical RPC data flow



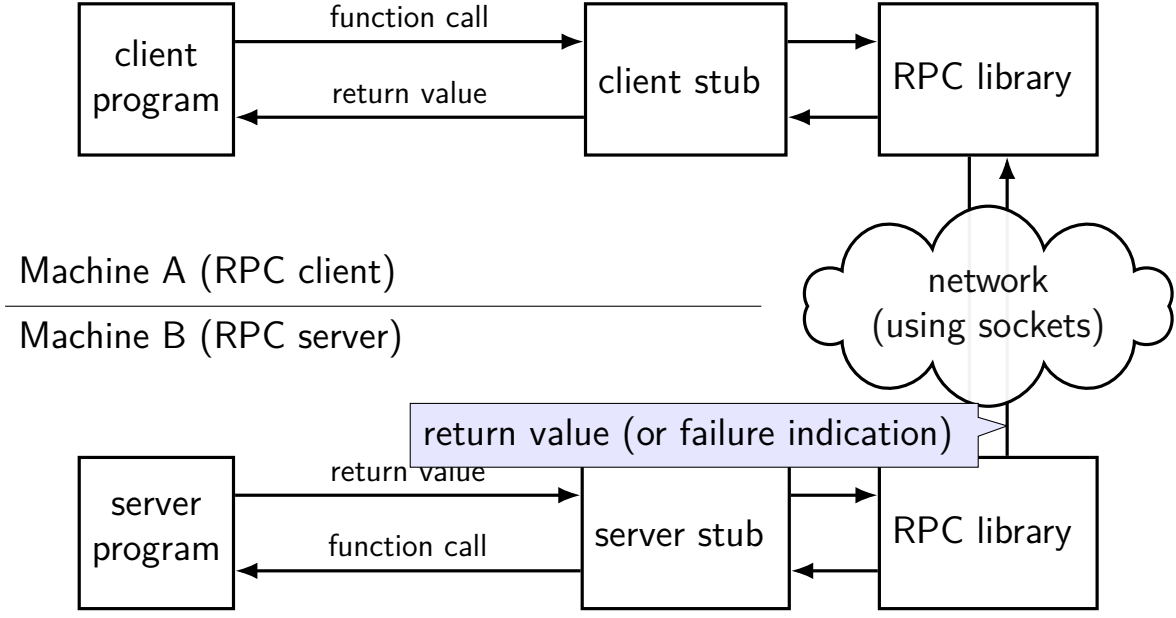
typical RPC data flow



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?

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 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()
```

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('127.0.0.1:12345')
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('127.0.0.1:12345')
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
- gives up if waiting too long

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?

throws an exception

Option A (returns when sent)

problem sending request
→ probably not created

Option B (waits for OK)

problem sending request?
request sent, but problem receiving reply?
→ could have been created

throws no exception

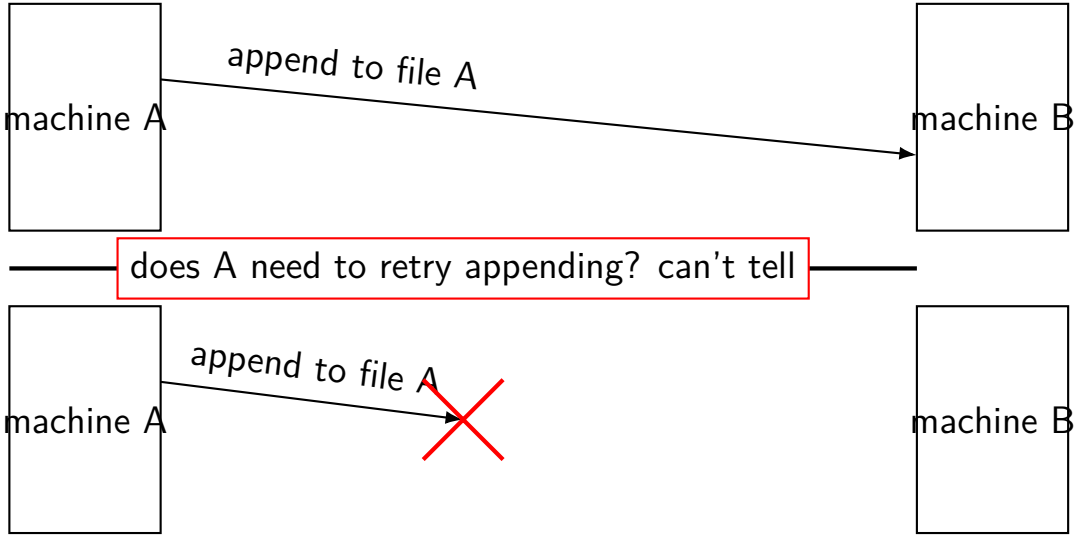
Option A (returns when sent)

- successfully sent
- did server recieve, process?
- $\rightarrow$ don't know!

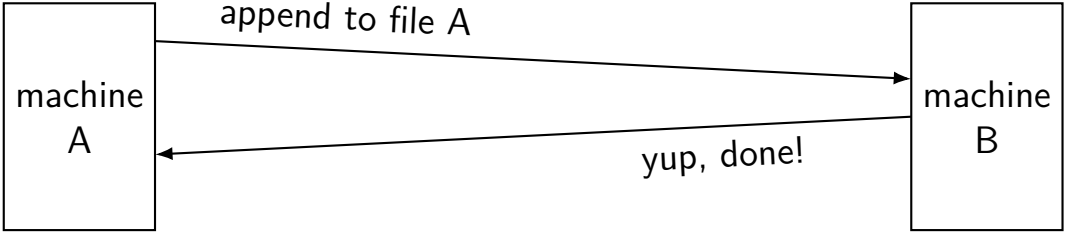
Option B (waits for OK)

- successfully sent AND
- successfully received reply
- $\rightarrow$ server created directory

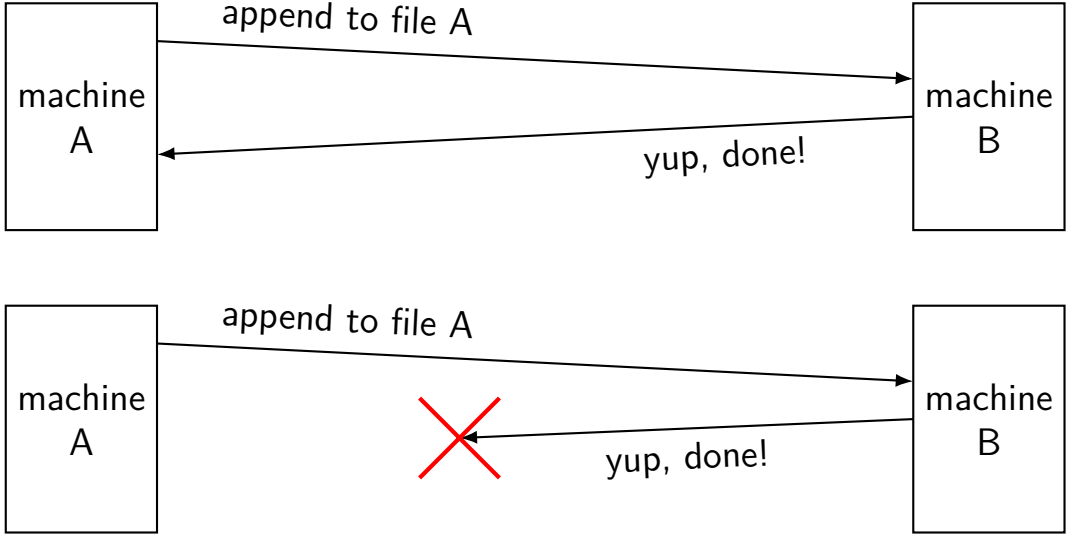
dealing with network message lost



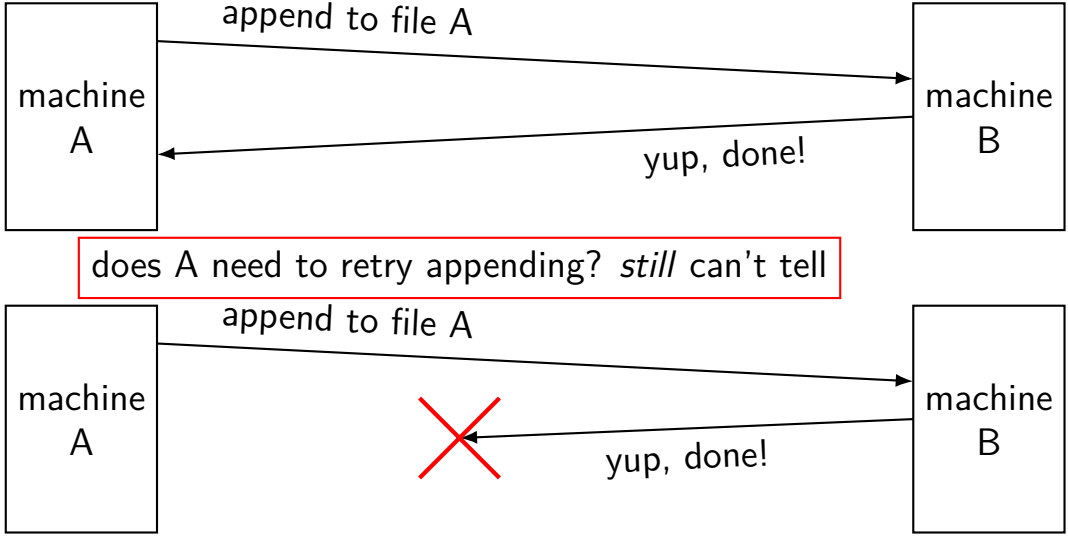
handling failures: try 1



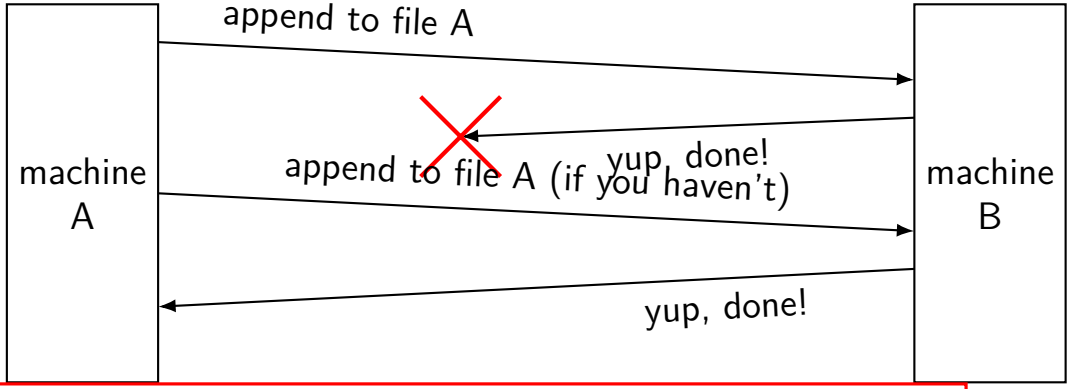
handling failures: try 1



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

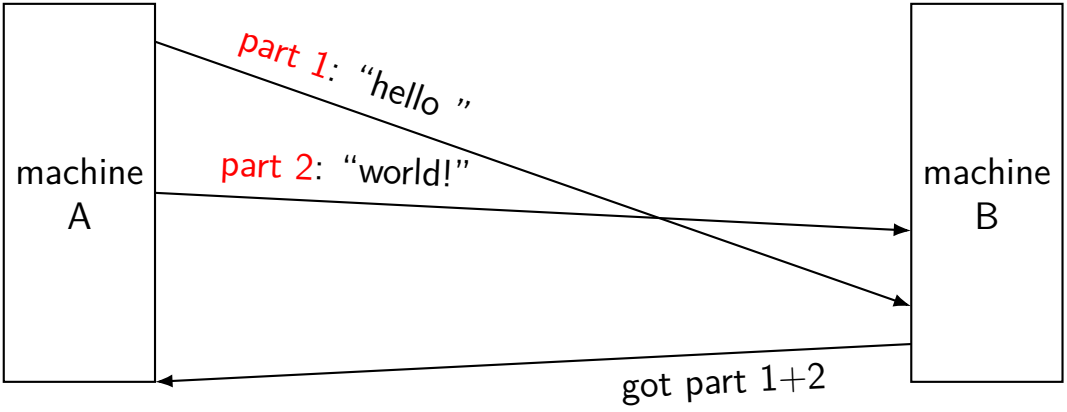
connection protocols do this

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

distributed transaction problem

distributed transaction

two machines both agree to do something *or not do something*

even if *a machine fails*

primary goal: *consistent* state

secondary goal: do it if nothing breaks

distributed transaction example

course database across many machines

machine A and B: student records

machine C: course records

want to make sure machines agree to add students to course

no confusion about student is in course even if failures

"consistency"

okay to say "no" — if possible, can retry later

backup slides

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

other model: every node has a copy of data

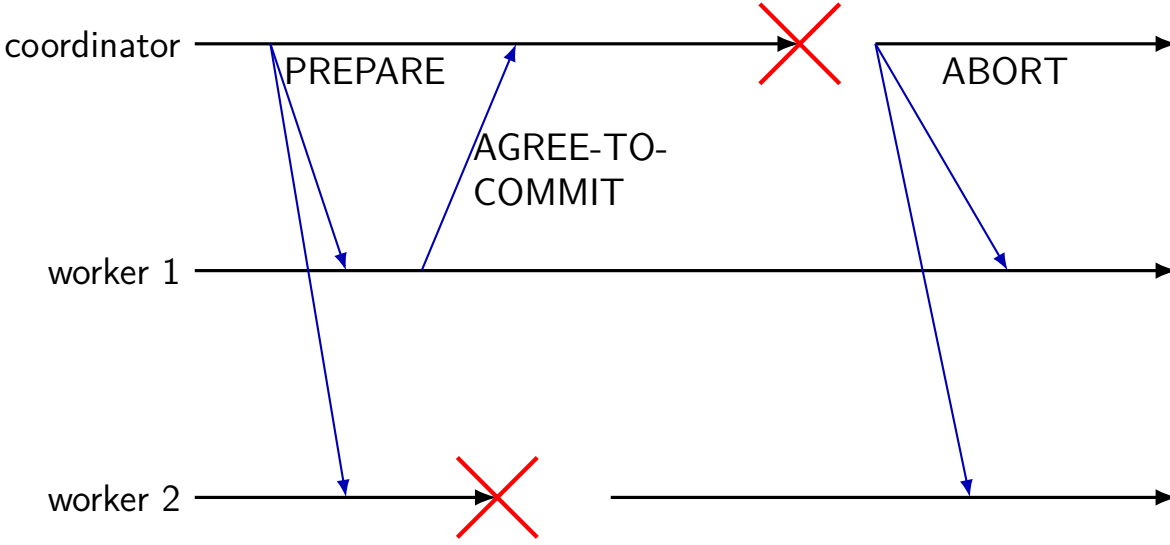
goal: work (including updates!) despite a few failing nodes

just require "enough" nodes to be working

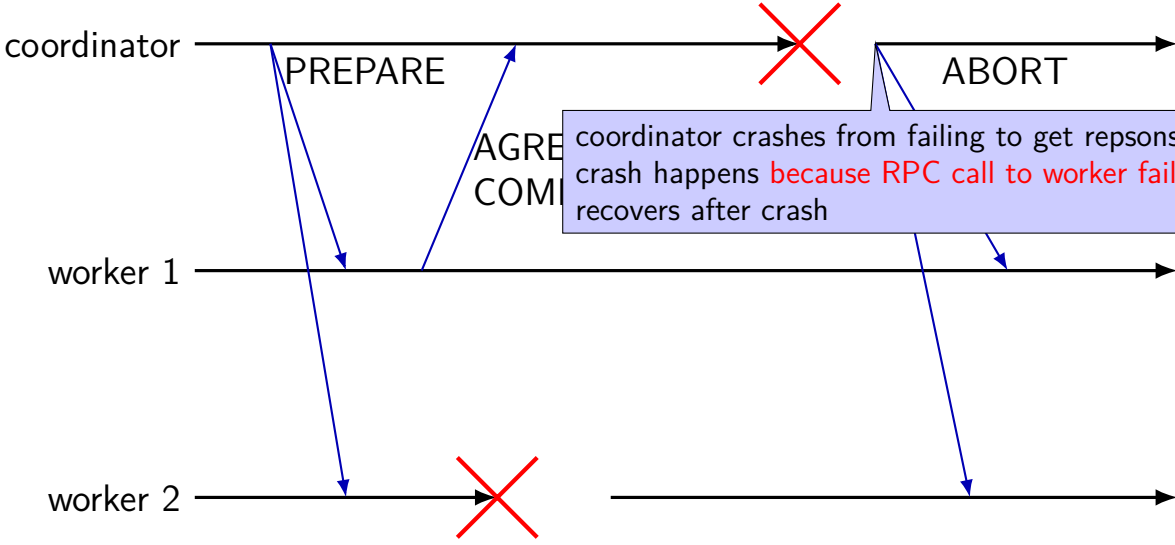
for now — assume fail-stop

- nodes don't respond or tell you if broken

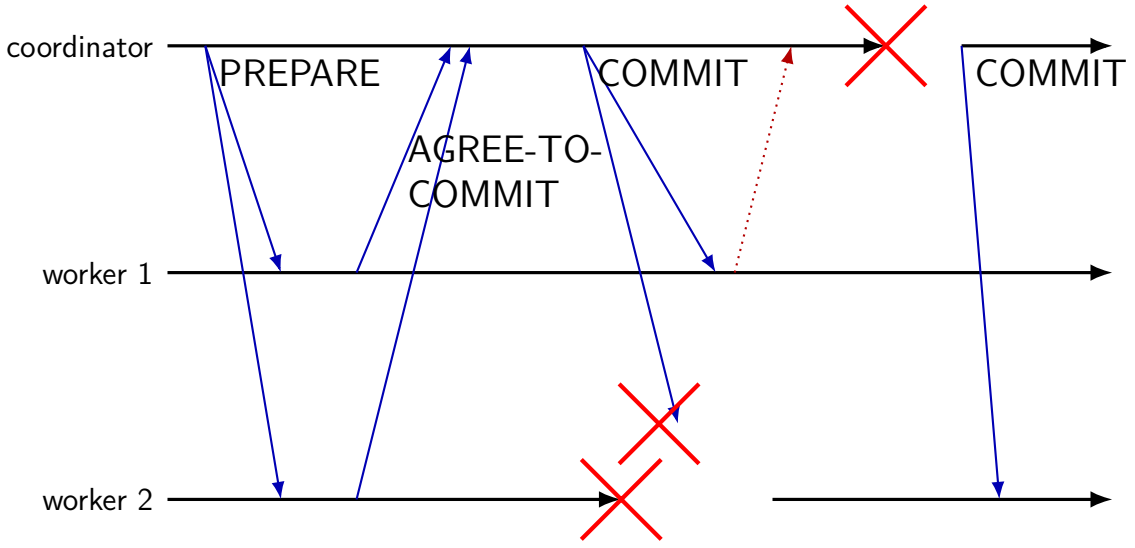
assignment: fails during prepare



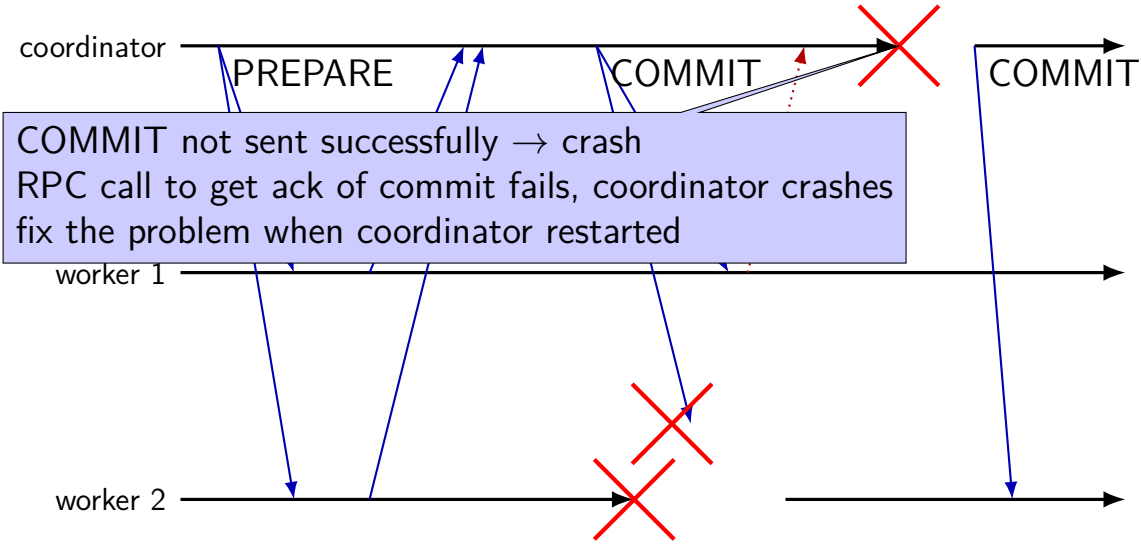
assignment: fails during prepare



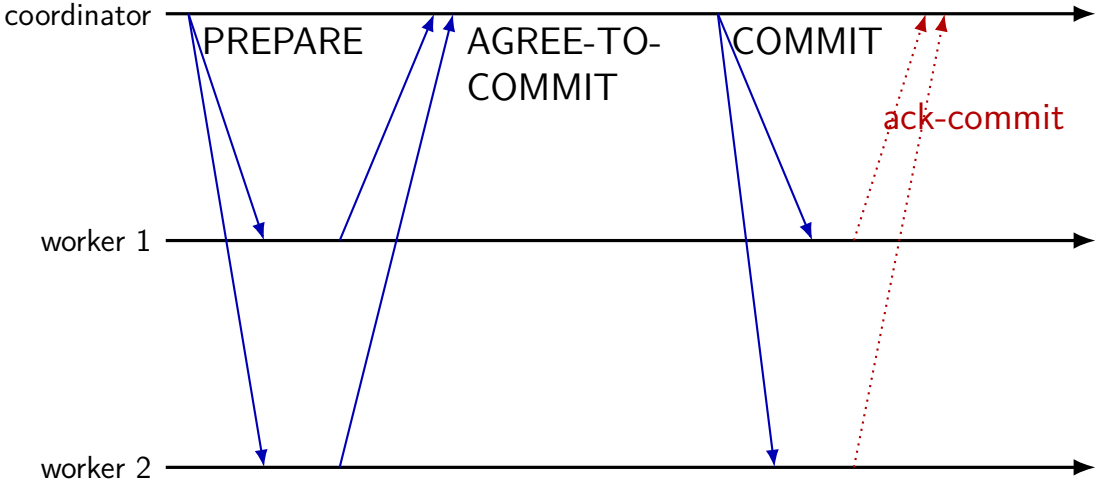
assignment: failuring during commit



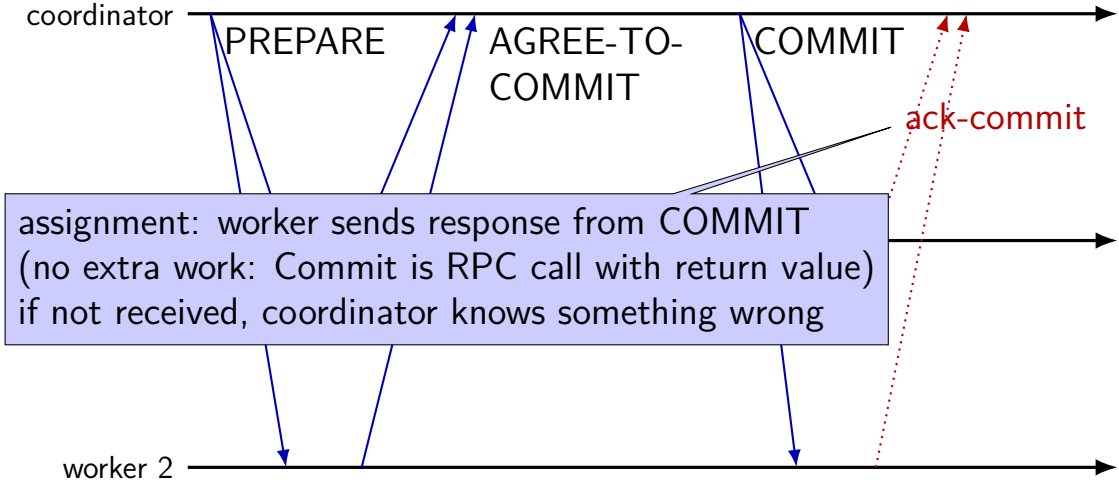
assignment: failuring during commit



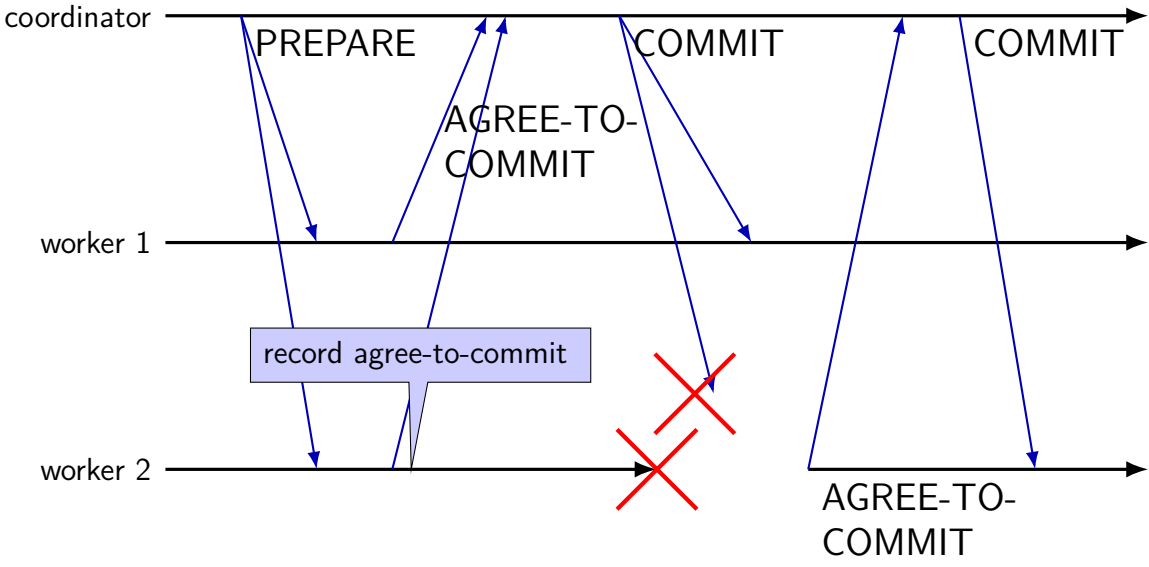
aside: worker ACKs



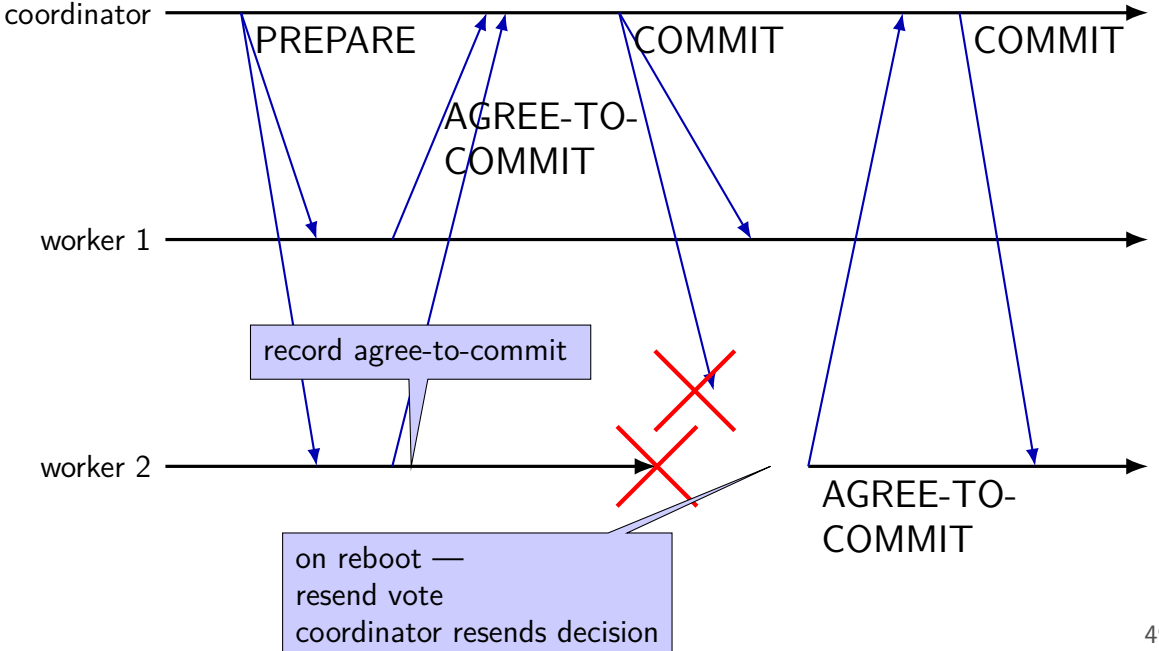
aside: worker ACKs



TPC: worker revoting



TPC: worker revoting

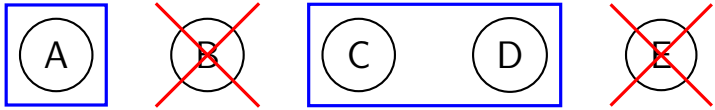


quorums (1)

perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

quorums (1)



perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

if A, C, D agree: that's enough

B, E will figure out what happened when they come back up

quorums (2)

requirement: quorums overlap

overlap = *someone in quorum* knows about every update

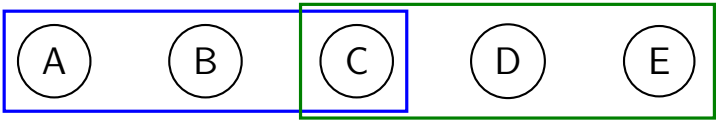
- e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

- make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

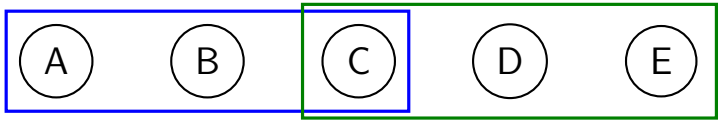
e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

e.g. every operation requires majority of nodes

part of voting — provide other voting nodes with 'missing' updates

make sure updates survive later on

cannot get a quorum to agree on anything conflicting with past updates