RPC (finish) / two-phase commit

Changelog

Changes made in this version not seen in first lecture:

- 19 November 2019: gRPC IDL example: update to be consistent with version of gRPC syntax used in assignment
- 19 November 2019: gRPC IDL example: add missing Empty message
- 19 November 2019: gRPC client/server examples: use name 'path' instead of 'name' for field from argument messages to be consistent with IDL
- 19 November 2019: gRPC server example: corrected inheritence from DirectoriesService to DirectoriesService*r*
- 19 November 2019: coordinator state machine (less simplified?): adjust failure/timeout action in prepare to be *ABORTing or* resending PREPARE
- 19 November 2019: leaking resources?: remove mention of statefulness which we haven't covered yet

RPC use pseudocode (C-like)

client:

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

---

server:

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

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

RPC use pseudocode (C-like)

client:

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

---

server:

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

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

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

RPC use pseudocode (C-like)

client:

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

---

server:

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

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

transparency failure:
doesn't look like a normal function call anymore
can we do better than this?

RPC use pseudocode (OO-like)

client:

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

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

---

server:

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

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

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? (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?

IDL pseudocode + marshalling example

```
protocol dirprotocol {
    1: int32 mkdir(string);
    2: int32 rmdir(string);
}
```

`mkdir("/directory/name")` returning `0`

client sends: `\x01/directory/name\x00`

server sends: `\x00\x00\x00\x00`

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)

```
channel = grpc.insecure_channel('127.0.0.1:43534')
stub = dirproto_pb2_grpc.DirectoriesStub(channel)
args = dirproto_pb2.MakeDirectoryArgs(path="/directory/name")
try:
  stub.MakeDirectory(args)
except grpc.RpcError as error:
  ... # handle error
```

RPC client implementation (method 2)

```
channel = grpc.insecure_channel('127.0.0.1:43534')
stub = dirproto_pb2_grpc.DirectoriesStub(channel)
args = dirproto_pb2.MakeDirectoryArgs(name="/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

gRPC: returning errors

any RPC can result in an error

- both errors from libraries and from RPCs can use same API

Python client: throws a grpc.RpcError exception

- no support for custom exceptions types (probably because tricky to make language-neutral)

C++ client: method return value is a Status object

- result of method 'returned' by modifying result object passed via pointer (for historical reasons, Google doesn't like C++ exceptions)

some gRPC errors

method not implemented

- e.g. server/client versions disagree
- local procedure calls — linker error

deadline exceeded

- no response from server after a while — is it just slow?

connection broken due to network problem

leaking resources?

```
stub = ...
remote_file_handle = stub.RemoteOpen(filename)
write_request = RemoteWriteRequest(
    file_handle=remote_file_handle,
    data="Some text.\n"
)
stub.RemotePrint(write_request)
stub.RemoteClose(remote_file_handle)
```

---

what happens if client crashes?

does server still have a file open?

on versioning

normal software: multiple versions of library?

- extra argument for function
- change what function does
- ...

just link against "correct version"

RPC: server gets upgraded out-of-sync with client

want to upgrade functions without breaking old clients

gRPC's versioning

gRPC: messages have field numbers

renaming fields? doesn't matter, just number changes

rules allow adding new (optional) fields

- get message with extra field — ignore it
- get message missing field — default/null value

otherwise, need to make new methods for each change

- ...and keep the old ones working for a while

versioned protocols

alternative approach: version numbers in protocol/messages

server can implement multiple versions

eventually discard old versions:

RPC performance

local procedure call: ~ 1 ns

system call: ~ 100 ns

network part of remote procedure call

(typical network) $> 400\,000$ ns

(super-fast network) $2\,600$ ns

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

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?

looks same as machine failing!

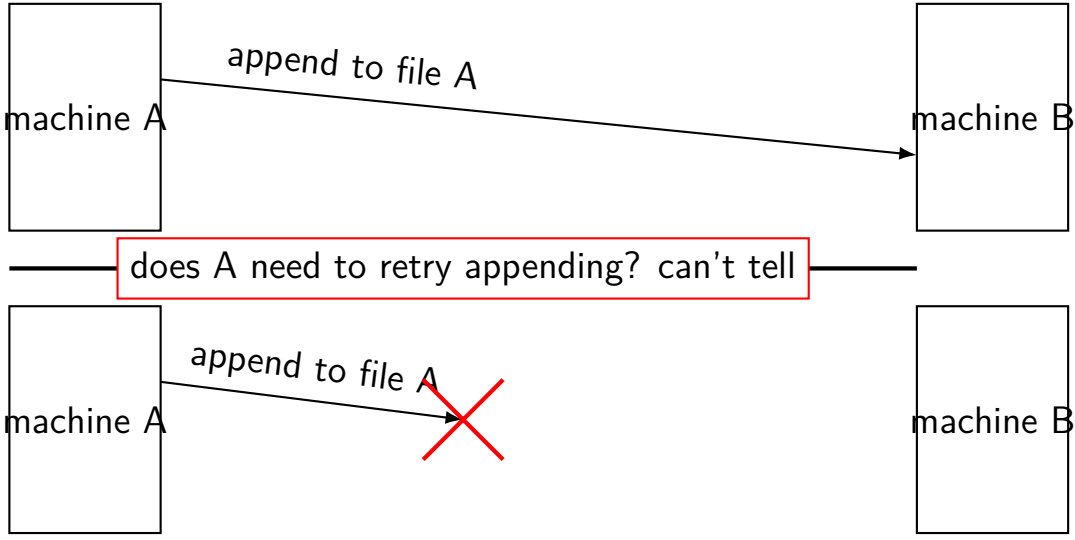
detect with acknowledgements

can recover by retrying

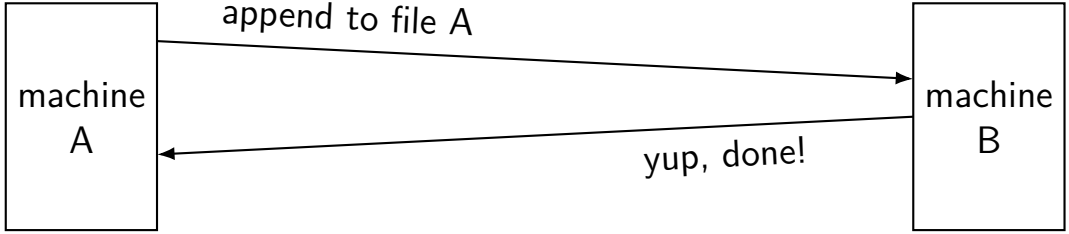
can't distinguish: original message lost or acknowledgment lost

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

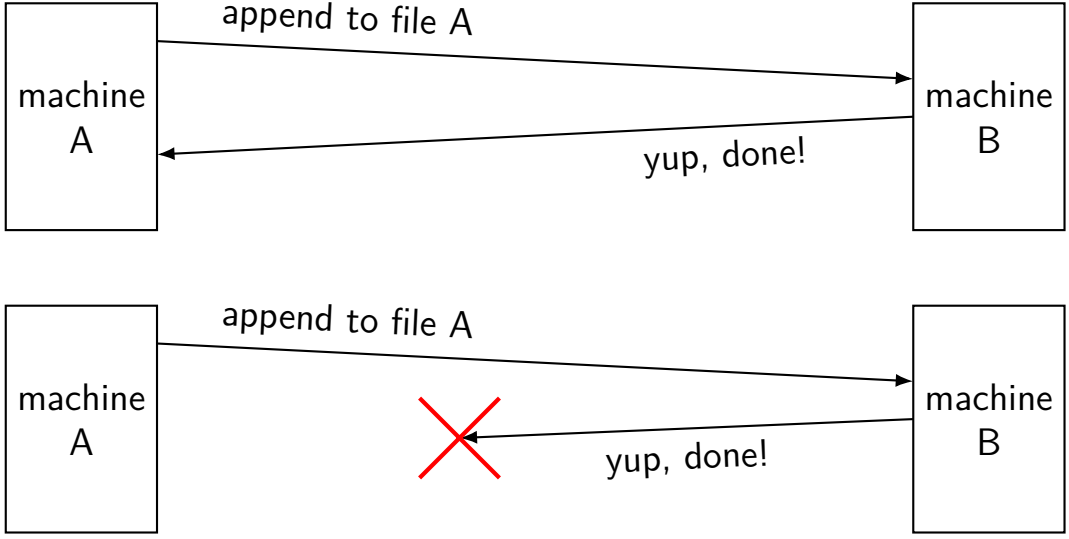
dealing with network message lost



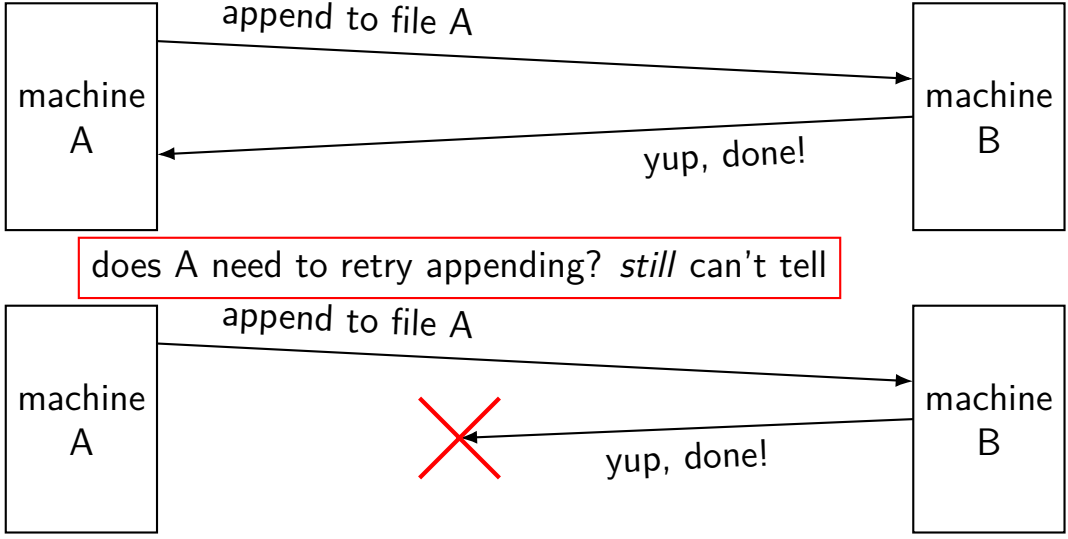
handling failures: try 1



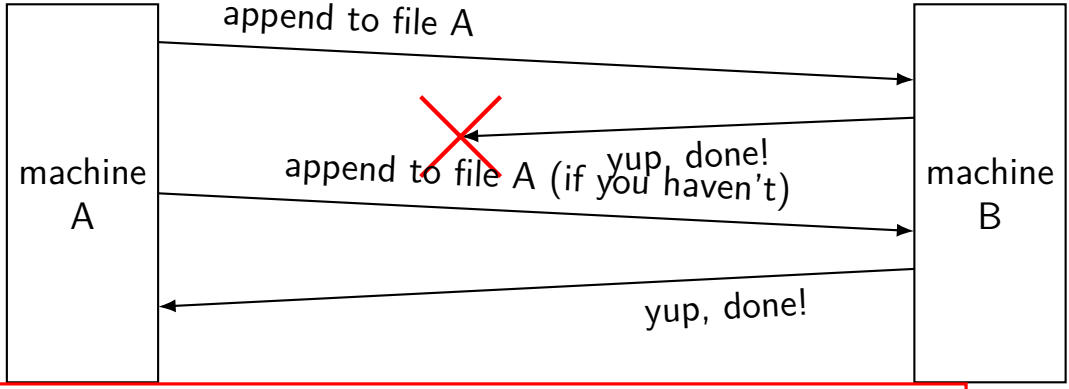
handling failures: try 1



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

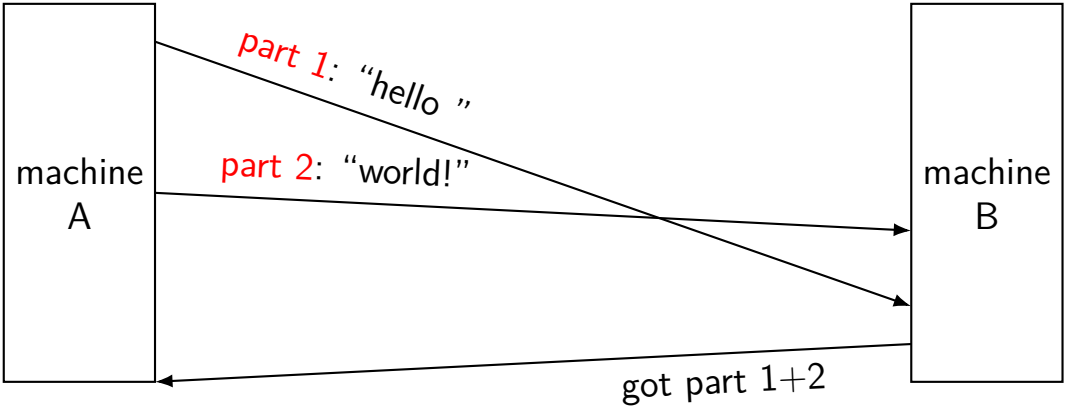
connection protocols do this

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

naive distributed transaction? (1)

machine A and B: student records; machine C: course records

any machine can be queried directly for info (e.g. by SIS web interface)

proposed add student to course procedure:

execute code on A or B where student is stored

tell C: add student to course

wait for response from C (if course full, return error)

locally: add student to course

what inconsistencies can be seen *if no failures*?

what inconsistencies can be seen *if failures*?

the centralized solution

one solution: a new machine D decides what to do

for machines A-C just which store records

machine D maintains a redo log for all machines

write to machine D's log

tell machine A-C to do operation

treats them as just data storage

problems with centralized solution

limited scaling — log-machine only so big/fast

combined responsibility — all data put together

- maybe reason for different machines was to separate data by type
- example: different organizations manage each type of data
- example: different regulatory requirements for each type of data

decentralized solution properties

each machine handles only its own data

- no sending machine to central place

machines involved in transaction if and only if have relevant data

- change only to courses? don't tell student machines
- change to course + student A? don't tell machine with student B

make progress as long as relevant machines don't fail

- losing one of K student machines? still runs for 1 of K students

decentralized solution properties

each machine handles only its own data

- no sending machine to central place

machines involved in transaction if and only if have relevant data

- change only to courses? don't tell student machines
- change to course + student A? don't tell machine with student B

make progress as long as relevant machines don't fail

- losing one of K student machines? still runs for 1 of K students

hope: scales to tens/hundreds of machines

- typical transaction: 1 to 3 machines?

two-phase commit

will look at solution that satisfies these propties

known as two-phase commit

name from two steps: figure out what to do, then do it

persisting past failures

will still use presistent log on each machine

idea: machine remembers what it was doing on failure

doesn't store data of other machines

...just some identifier/contact info for the transaction

two-phase commit: roles

elect one machine to be *coordinator*

other machines are *workers*

common implementation: one physical machine runs both coordinator+one of the workers

abort if anyone decides to abort

coordinator collects workers' vote: will they abort?

coordinator makes final decision

two-phase commit: no take-backs

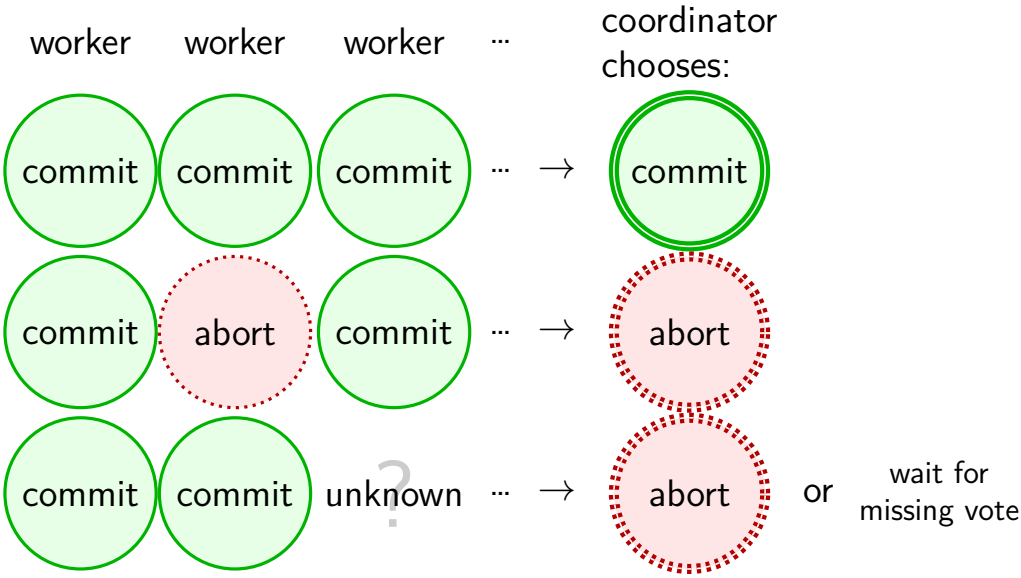
once worker agrees not to abort, they can't change their mind

once coordinator makes decision, it is final

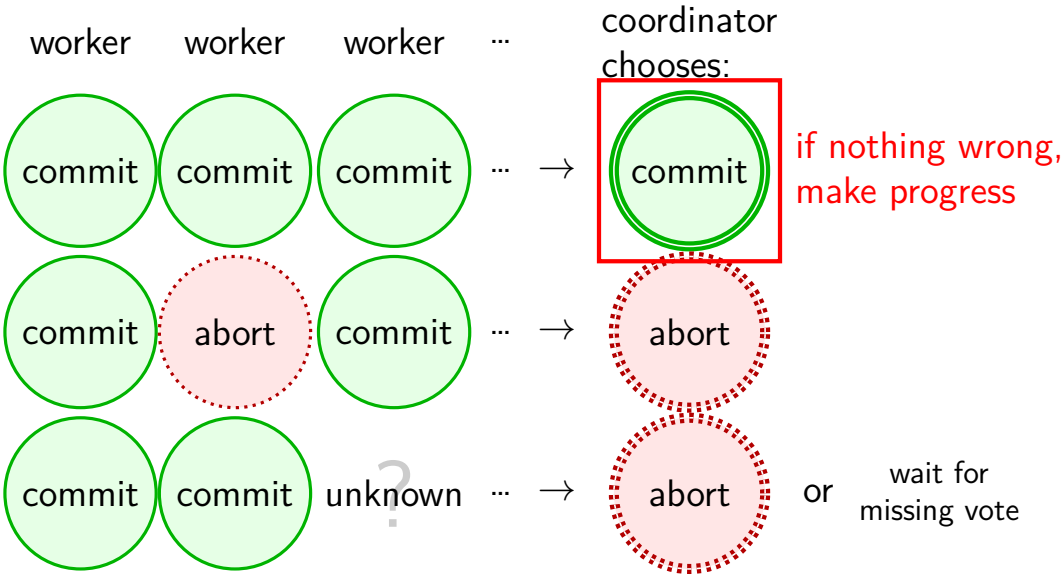
both cases: need to remember decision in log

- fail-stop $\rightarrow$ assume log will be there

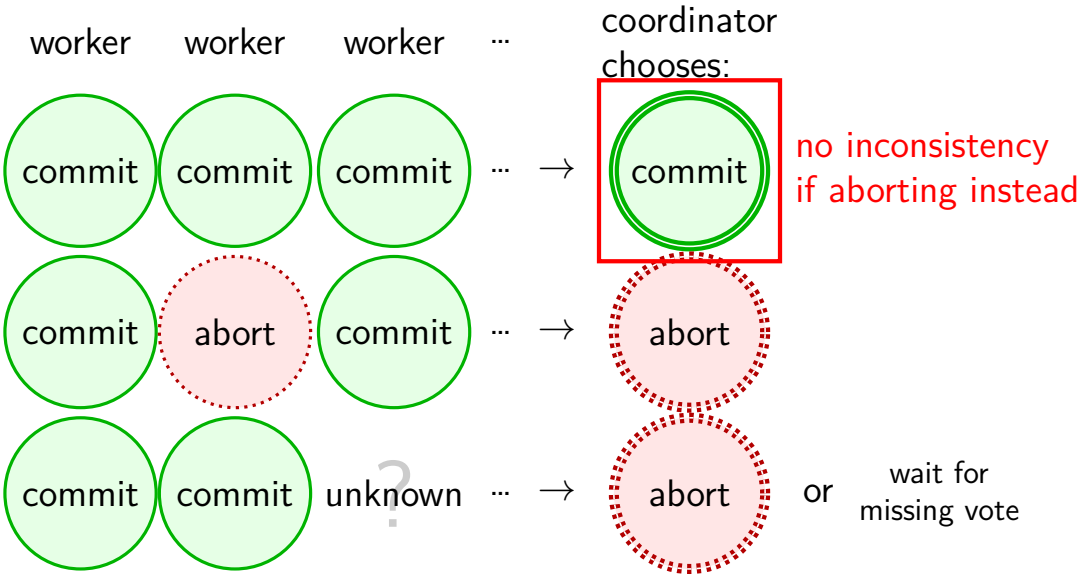
two-phase commit: voting



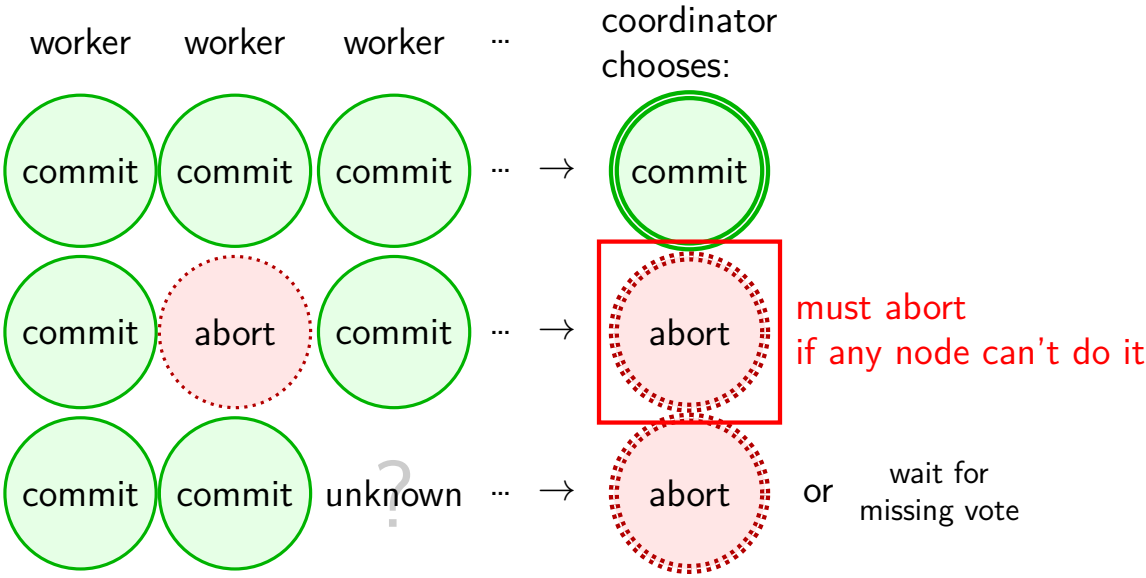
two-phase commit: voting



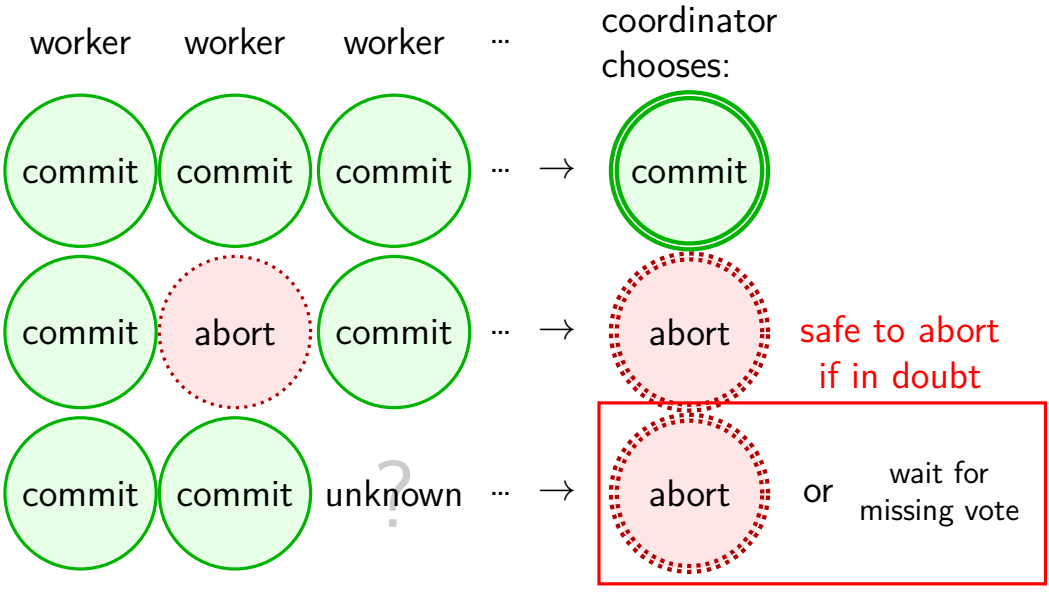
two-phase commit: voting



two-phase commit: voting



two-phase commit: voting



two-phase commit: phases

phase 1: *preparing*

workers tell coordinator their votes: agree to commit/abort

phase 2: *finishing*

coordinator gathers votes, decides and tells everyone the outcome

preparing

agree to commit

promise: "I will accept this transaction"
promise recorded in the machine log in case it crashes

agree to abort

promise: "I will **not** accept this transaction"
promise recorded in the machine log in case it crashes

never ever take back agreement!

preparing

agree to commit

promise: "I will accept this transaction"

promise recorded in the machine log in case it crashes

agree to abort

promise: "I will **not** accept this transaction"

promise recorded in the machine log in case it crashes

never ever take back agreement!

to keep promise: can't allow interfering operations

e.g. agree to add student to class $\rightarrow$ reserve seat in class

(even though student might not be added b/c of other machines)

coordinator decision

coordinator can't take back global decision

must record in presistent log to ensure not forgotten

coordinator decision

coordinator can't take back global decision

must record in presistent log to ensure not forgotten

coordinator fails without logged decision? collect votes again

finishing

coordinator says commit → commit transaction

- worker applies transcation (e.g. record student is in class)

coordinator (or anyone) says abort → abort transaction

- worker never ever applies transaction
- still want to do operation? make a new transaction

finishing

coordinator says commit → commit transaction

- worker applies transcation (e.g. record student is in class)

coordinator (or anyone) says abort → abort transaction

- worker never ever applies transaction
- still want to do operation? make a new transaction

unsure which? option 1: ask coordinator

- e.g. worker policy: keep asking if no outcome

unsure which? option 2: make sure coordinator resends outcome

- e.g. coordinator keeps sending outcome until it gets "yes, I got it" reply

two-phase commit: blocking

agree to commit "add student to class"?

can't allow conflicting actions...

...until know transaction *globally* committed/aborted

two-phase commit: blocking

agree to commit "add student to class"?

can't allow conflicting actions...

- adding student to conflicting class?
- removing student from the class?
- not leaving seat in class?

...until know transaction *globally* committed/aborted

waiting forever?

if machine goes away at wrong time, might *never* decide what happens

solution in practice: manual intervention

waiting forever?

if machine goes away at wrong time, might *never* decide what happens

solution in practice: manual intervention

mitigation (1): coordinator aborts if still possible

- requires coordinator not to go away
- handles *workers* failing before decision made

mitigation (2): workers share outcomes without coordinator

- possibly handles coordinator failing (if all workers still working fine)
- other worker can say "coordinator said ABORT/COMMIT" (even if coordinator now down)
- if any worker agreed to abort, don't need coordinator

two-phase commit: roles

typical two-phase commit implementation

several *workers*

one *coordinator*

- might be same machine as a worker

two-phase-commit messages

coordiantor → worker: PREPARE

> "will you agree to do this action?"
> on failure: can ask multiple times!

worker → coordinator:
AGREE-TO-COMMIT or AGREE-TO-ABORT

> worker records decision in log (before sending)

coordinator → worker: COMMIT or ABORT

> I counted the votes and the result is commit/abort
> only commit if all votes were commit

reasoning about protocols: state machines

very hard to reason about dist. protocol correctness

typical tool: state machine

each machine is in some state

know what every message does in this state

reasoning about protocols: state machines

very hard to reason about dist. protocol correctness

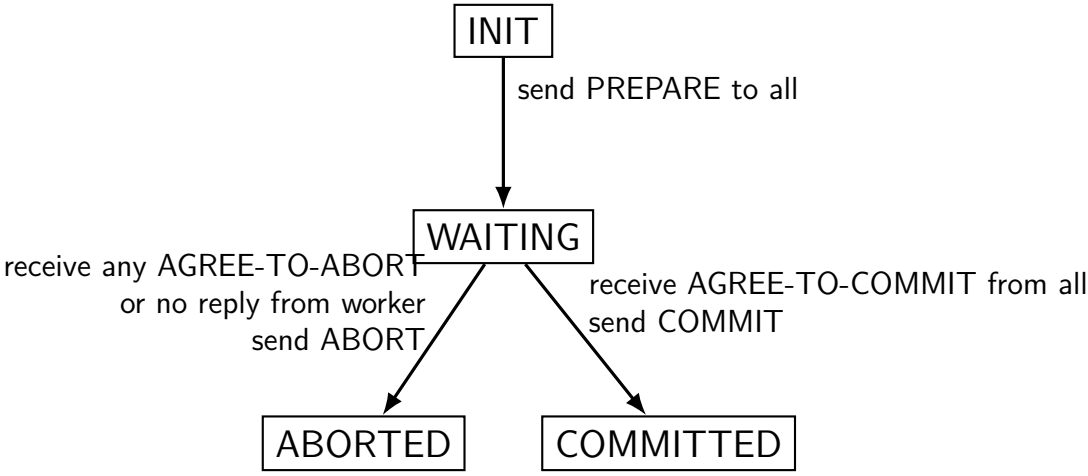
typical tool: state machine

each machine is in some state

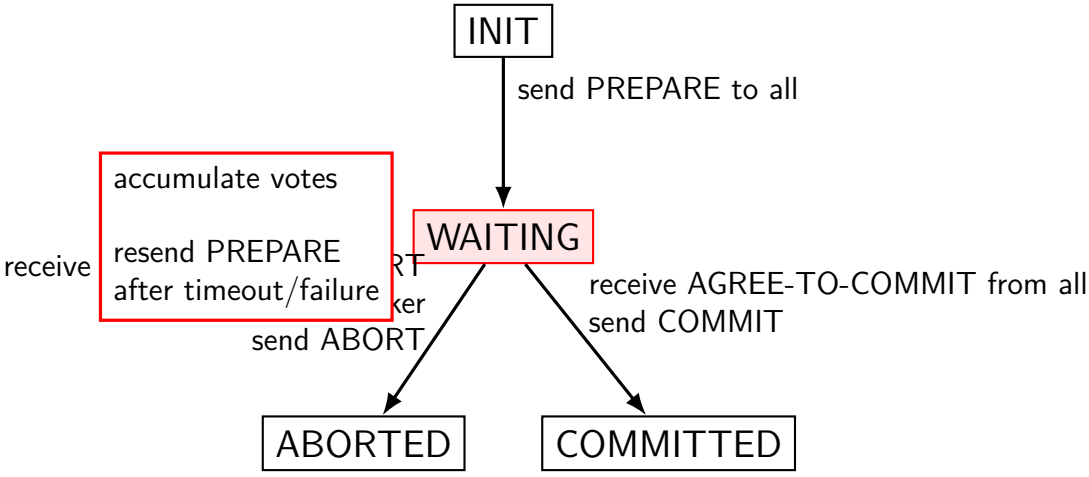
know what every message does in this state

avoids common problem: don't know what message does

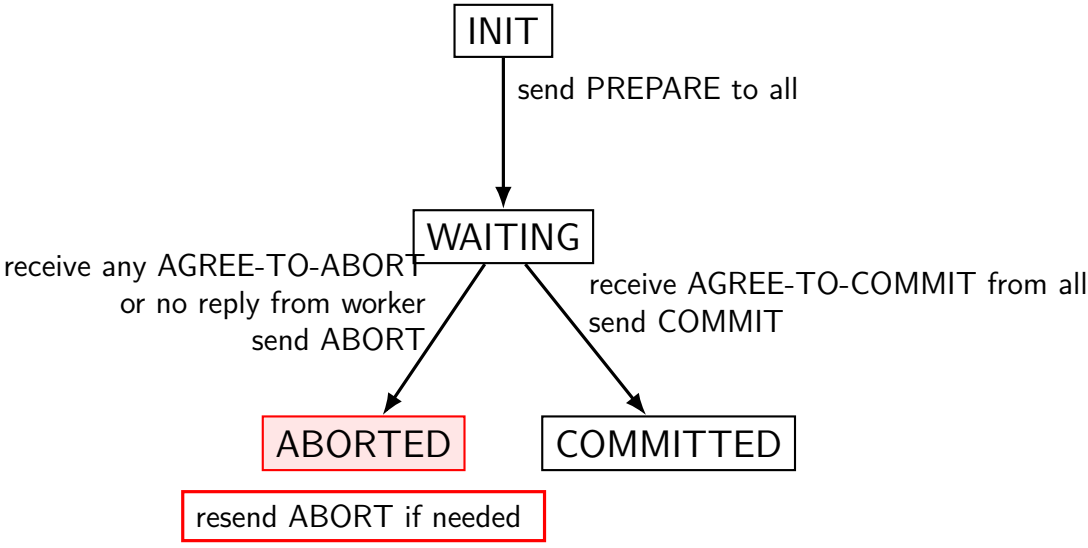
coordinator state machine (simplified?)



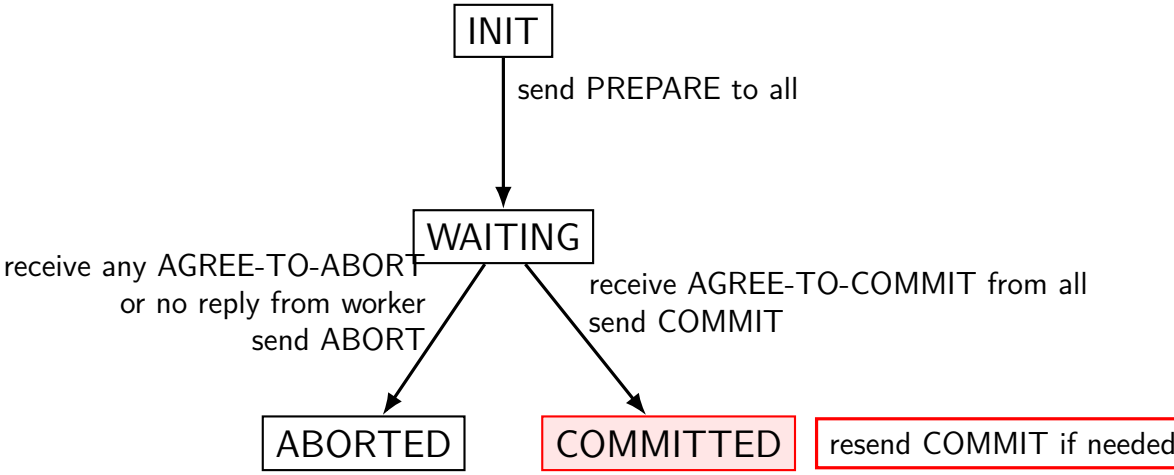
coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

- log written *before* sending any messages
- if INIT: resend PREPARE,
  if WAIT/ABORTED: (re)send ABORT to all
- if COMMITTED: (re)send COMMIT to all

no vote from worker?

- ABORT or resend after timeout

COMMIT/ABORT doesn't make it to worker

- worker can ask to resend after timeout, or
  coordinator can ask workers for acknowledgment, resend if none

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

- log written *before* sending any messages
- if INIT: resend PREPARE,
  if WAIT/ABORTED: (re)send ABORT to all
- if COMMITTED: (re)send COMMIT to all

no vote from worker?

- ABORT or resend after timeout

COMMIT/ABORT doesn't make it to worker

- worker can ask to resend after timeout, or
  coordinator can ask workers for acknowledgment, resend if none

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

- log written *before* sending any messages
- if INIT: resend PREPARE,
- if WAIT/ABORTED: (re)send ABORT to all
  - if WAIT, could also resend PREPARE (try to get votes again)
- if COMMITTED: (re)send COMMIT to all

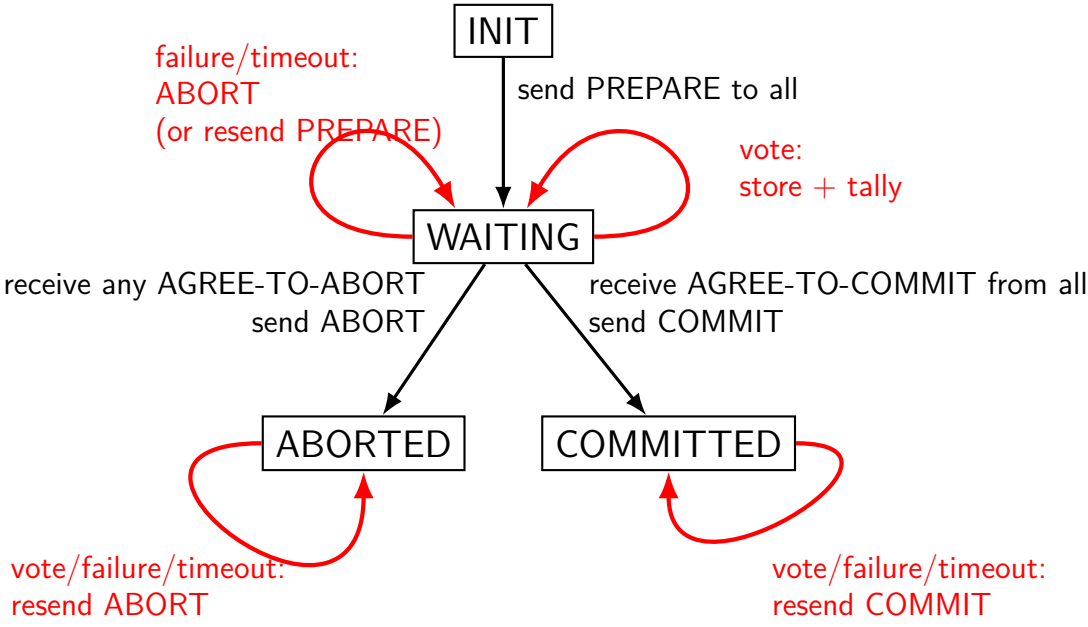
no vote from worker?

- ABORT or resend after timeout

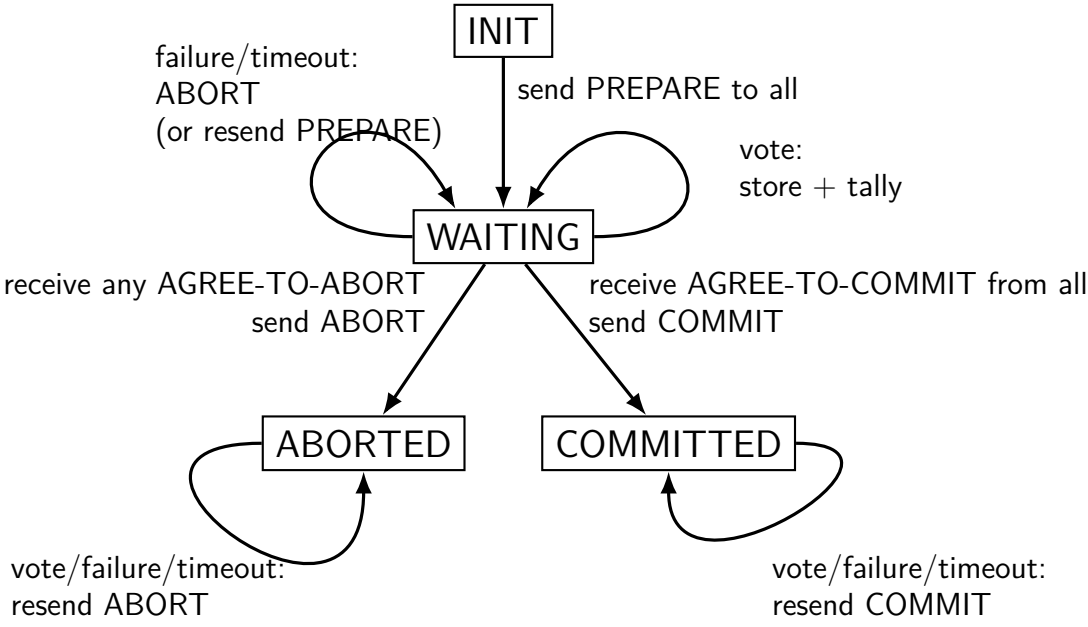
COMMIT/ABORT doesn't make it to worker

- worker can ask to resend after timeout, or
- coordinator can ask workers for acknowledgment, resend if none

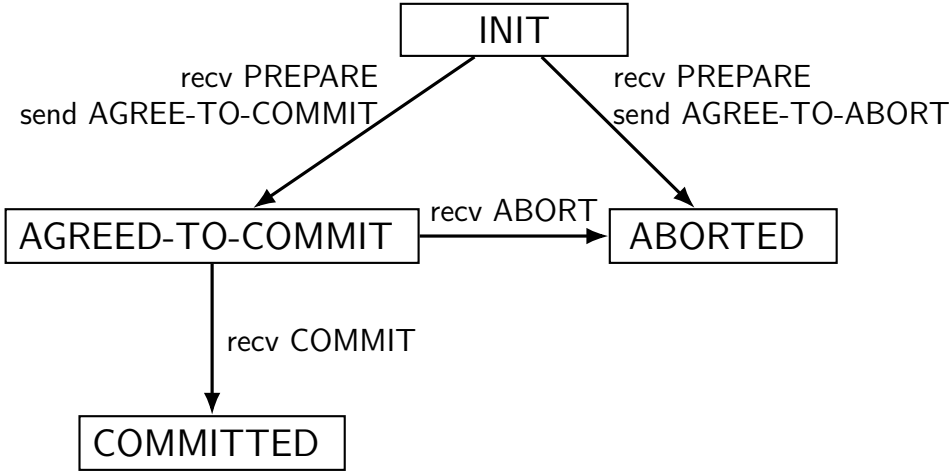
coordinator state machine (less simplified?)



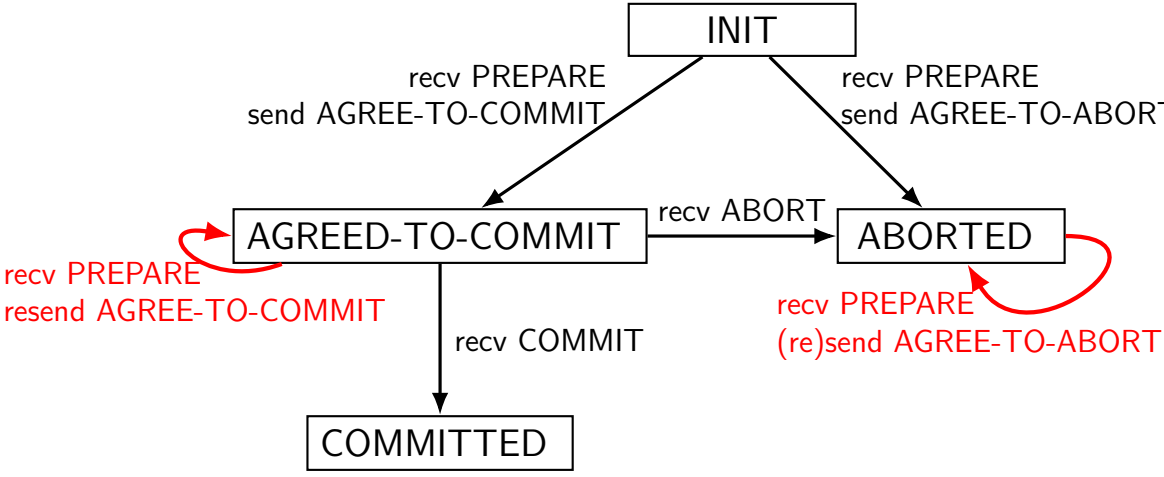
coordinator state machine (less simplified?)



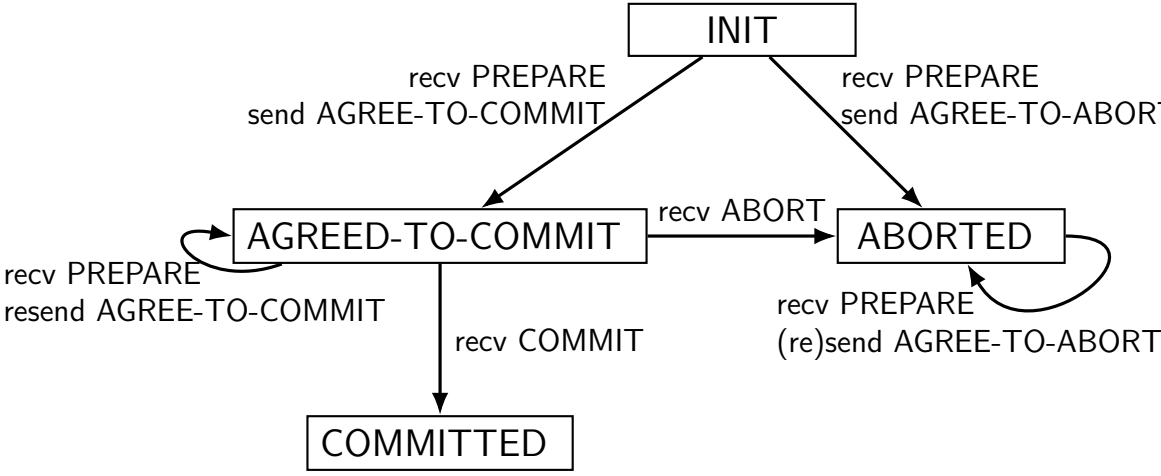
worker state machine (simplified)



worker state machine (less simplified?)



worker state machine (less simplified?)



worker failure recovery

worker crashes? *log* indicating last state

- if INIT: wait for PREPARE (resent)?
- if AGREE-TO-COMMIT or ABORTED: resend AGREE-TO-COMMIT/ABORT
- if COMMITTED: redo operation

message doesn't make it to coordinator

- resend after timeout or during reboot on recovery

state machine missing details

really want to specify *result of/action for every message!*

- worker recv ABORT in ABORTED: do nothing
- worker recv ABORT in INIT: go to ABORTED
- worker recv PREPARE in COMMITTED: ignore?
- ...

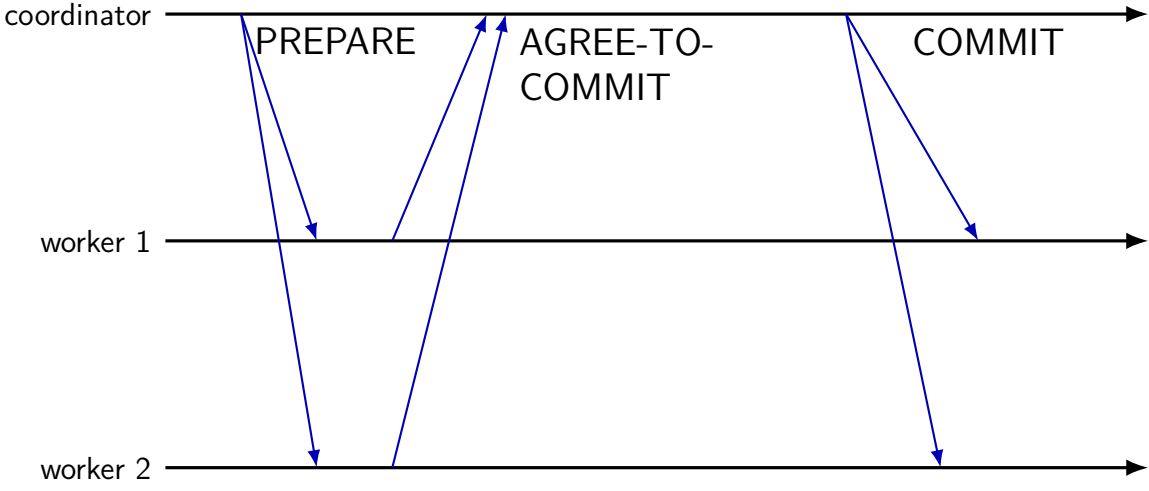
want to discard finished transactions eventually

- ...need to not get confused by delayed messages

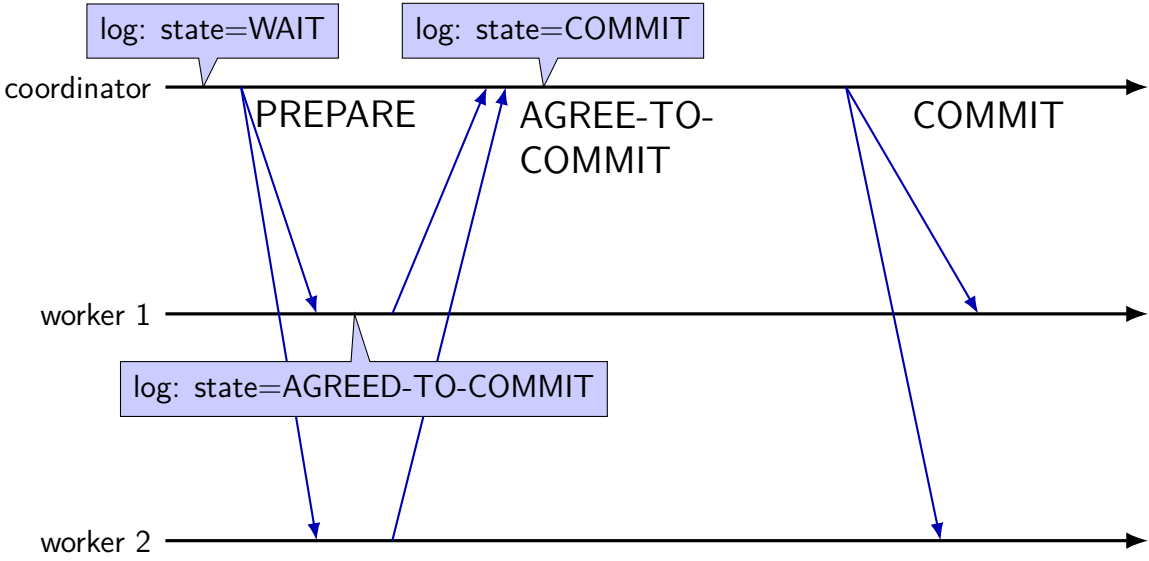
allows *programmatic* verifying properties of state machine

- what happens if machine fails at each possible time?
- what happens if each subset of messages is lost?
- ...

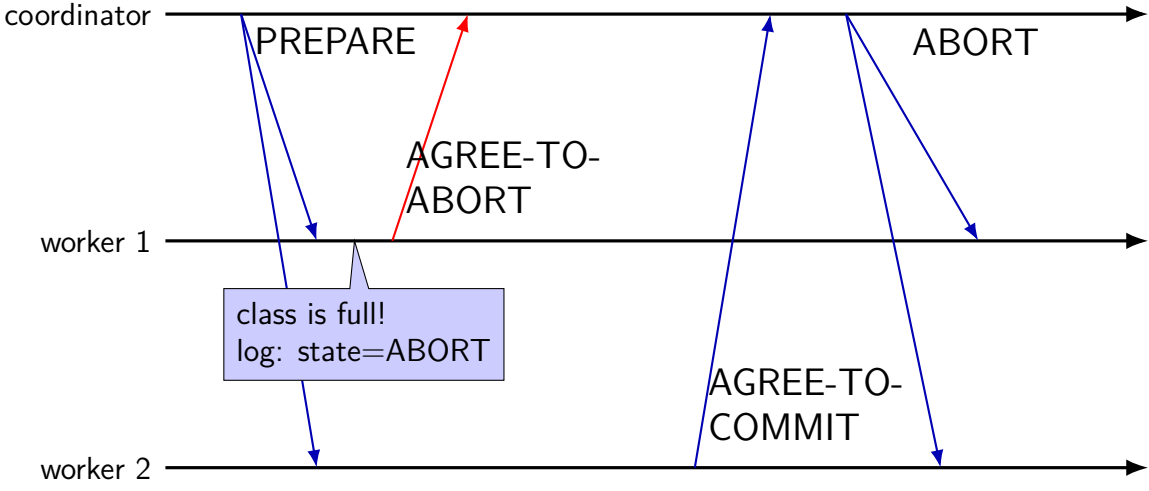
TPC: normal operation



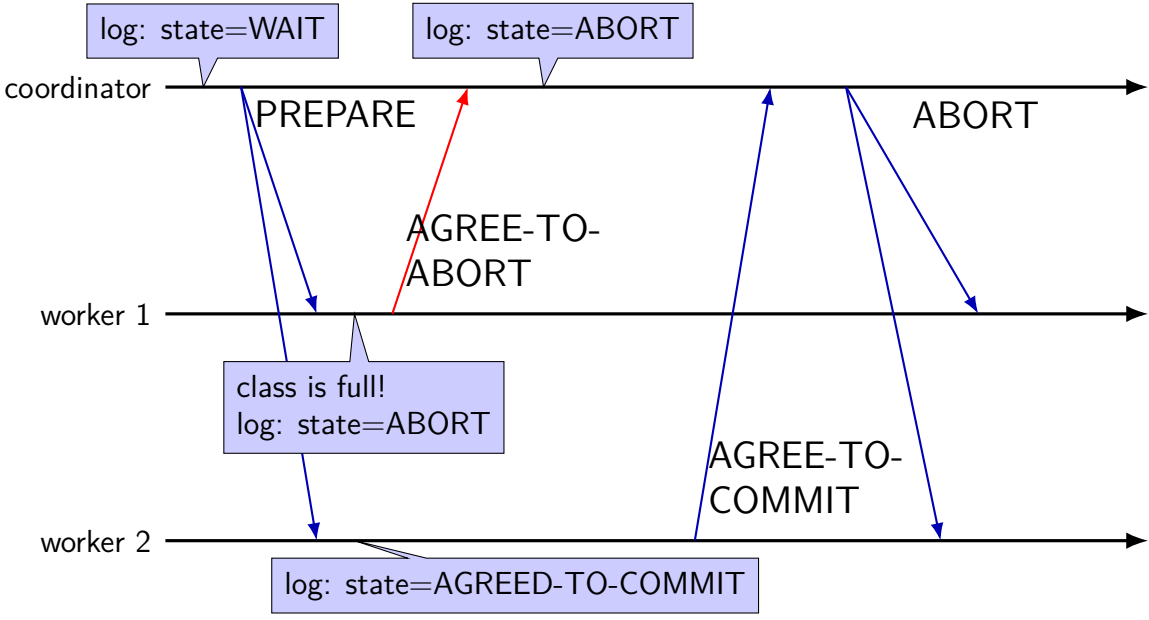
TPC: normal operation



TPC: normal operation — conflict



TPC: normal operation — conflict

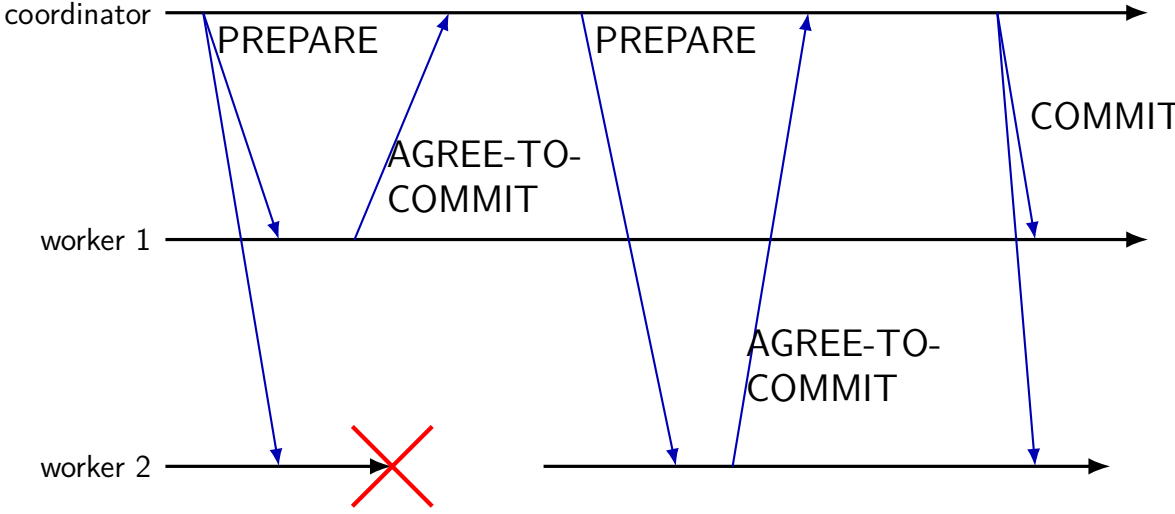


some failure cases

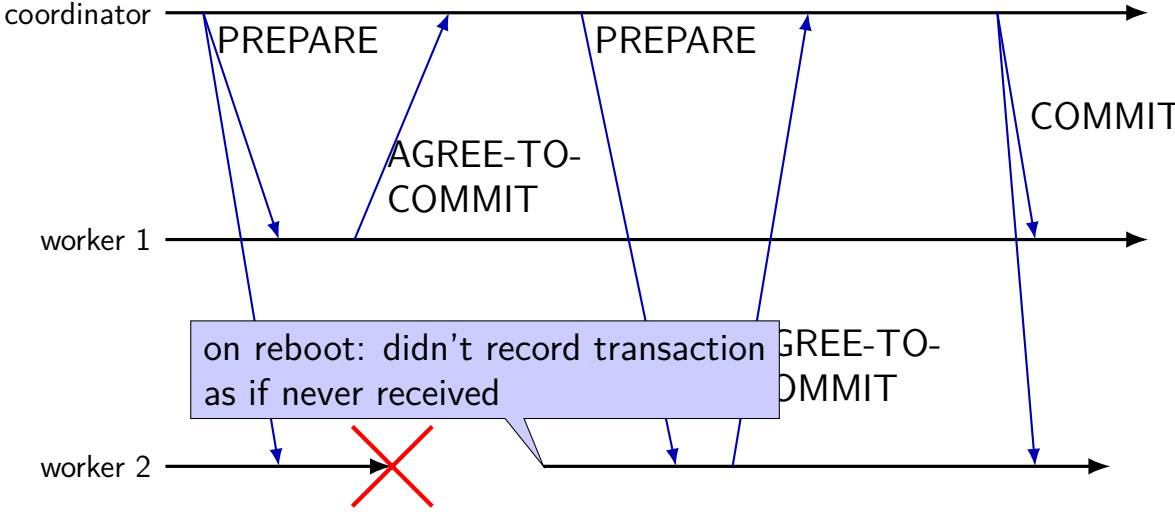
worker failure after prepare?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote (must have recorded prepare)

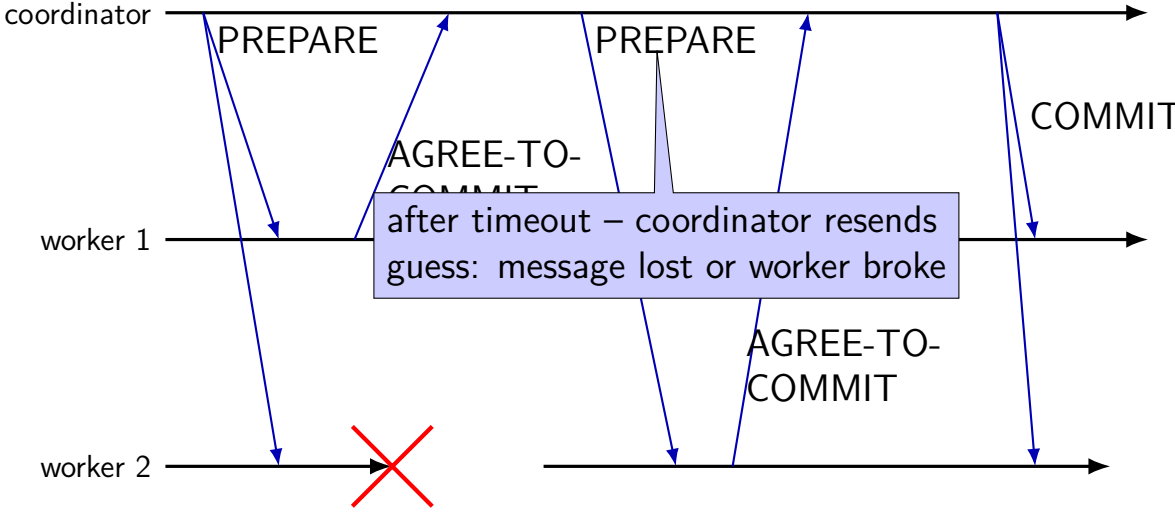
TPC: worker fails after prepare (1)



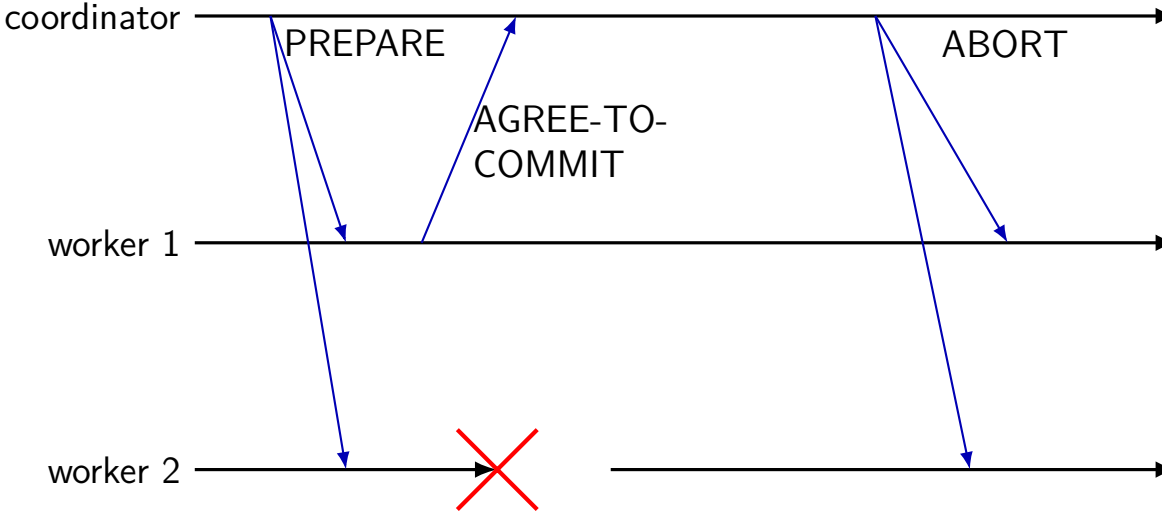
TPC: worker fails after prepare (1)



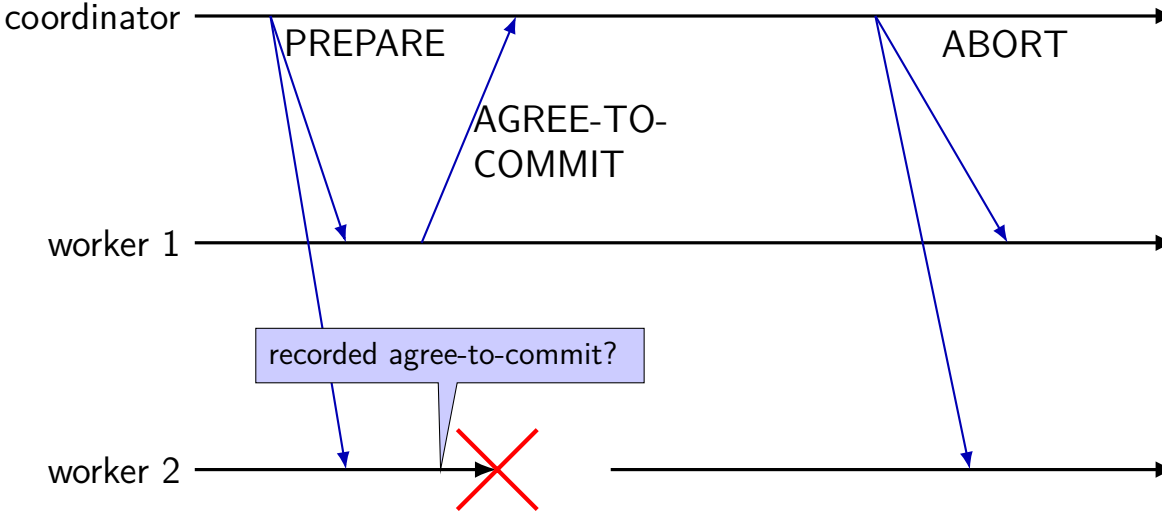
TPC: worker fails after prepare (1)



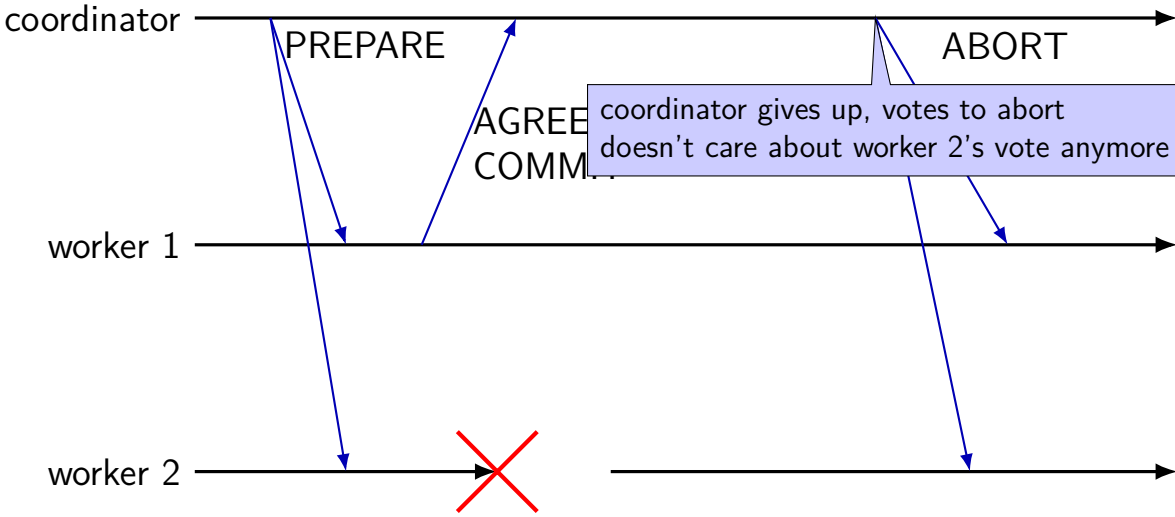
TPC: worker fails after prepare (2)



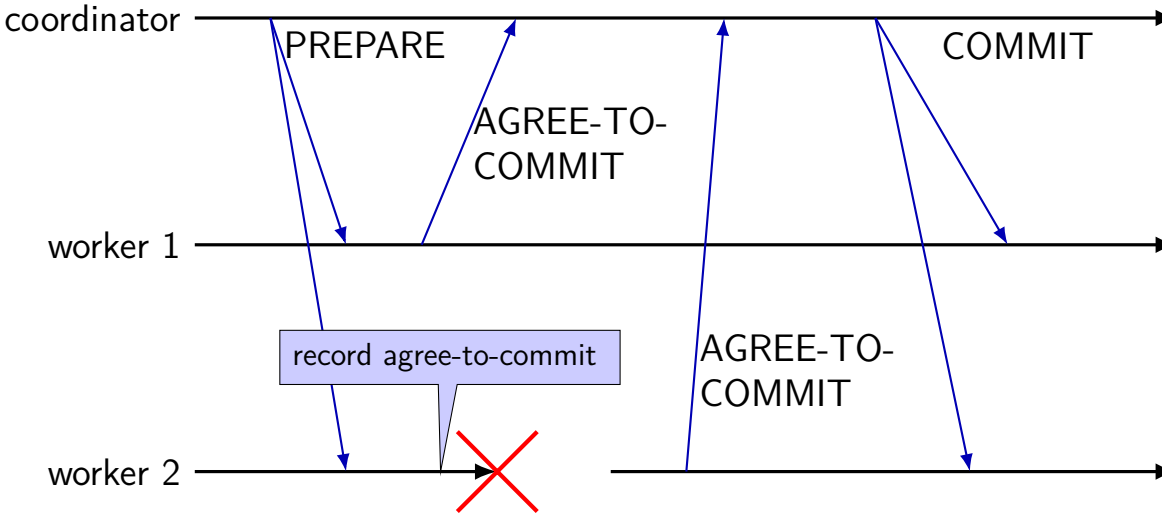
TPC: worker fails after prepare (2)



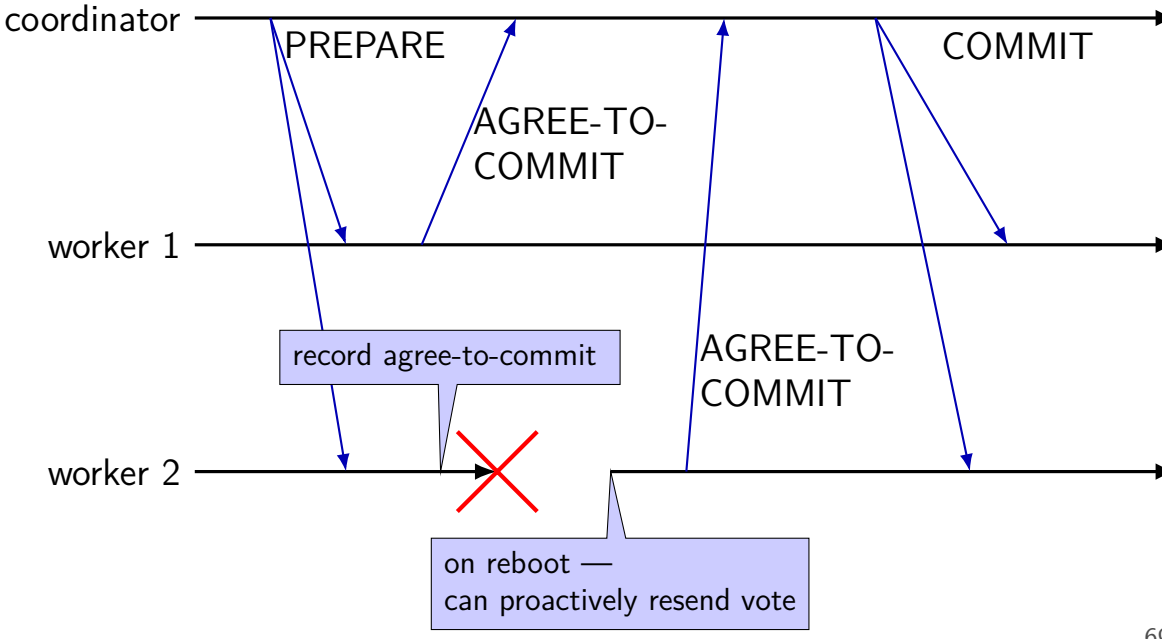
TPC: worker fails after prepare (2)



TPC: worker fails after prepare (3)



TPC: worker fails after prepare (3)



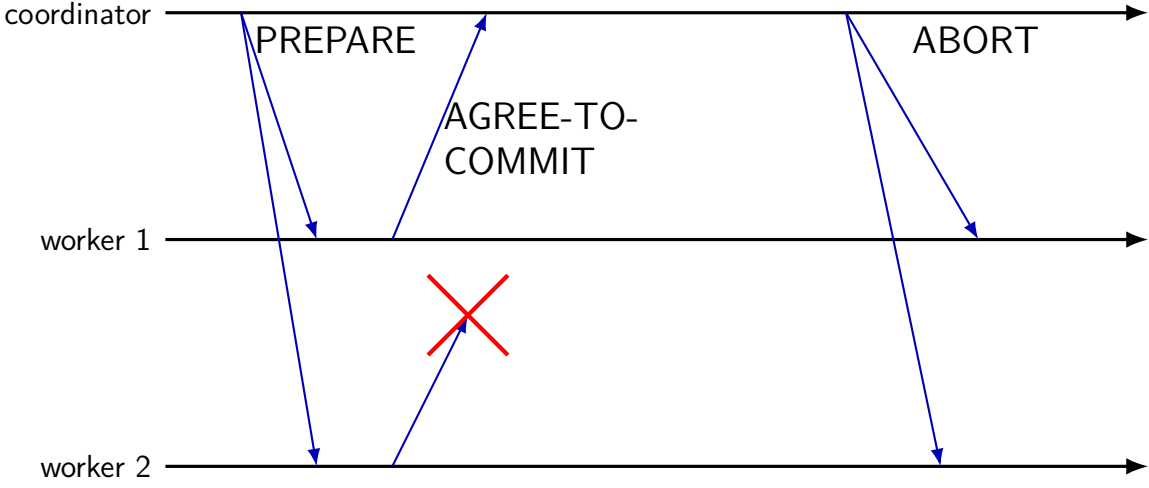
network failure after during voting?

network failure during voting $\approx$ node failure

same options:

- coordinator resends PREPARE
- coordinator gives up
- worker resends vote

TPC: network failure (1)



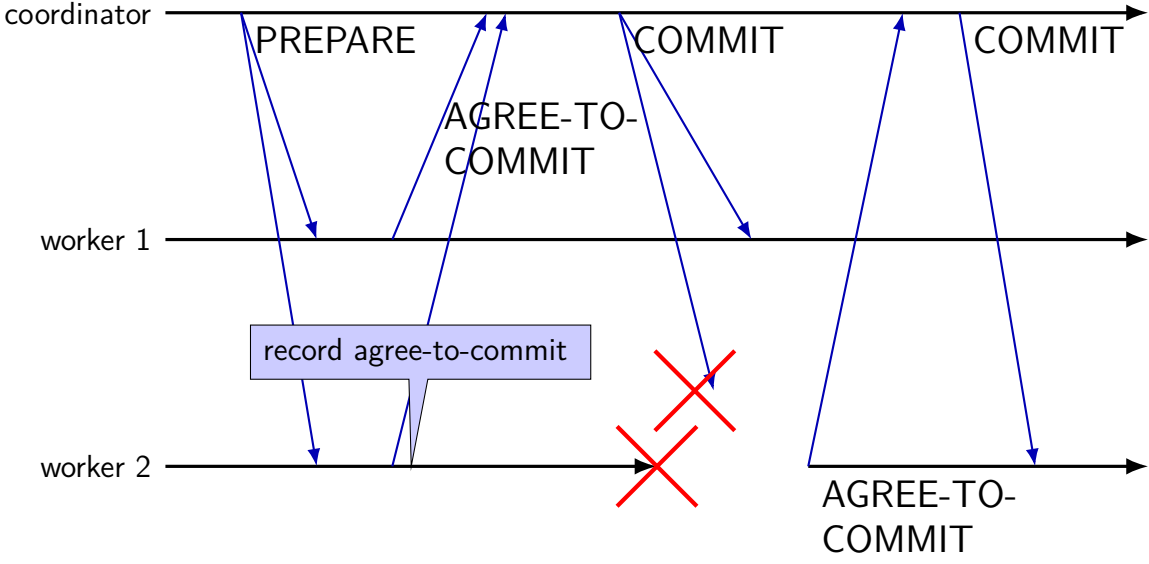
worker failure during commit

worker failure during commit?

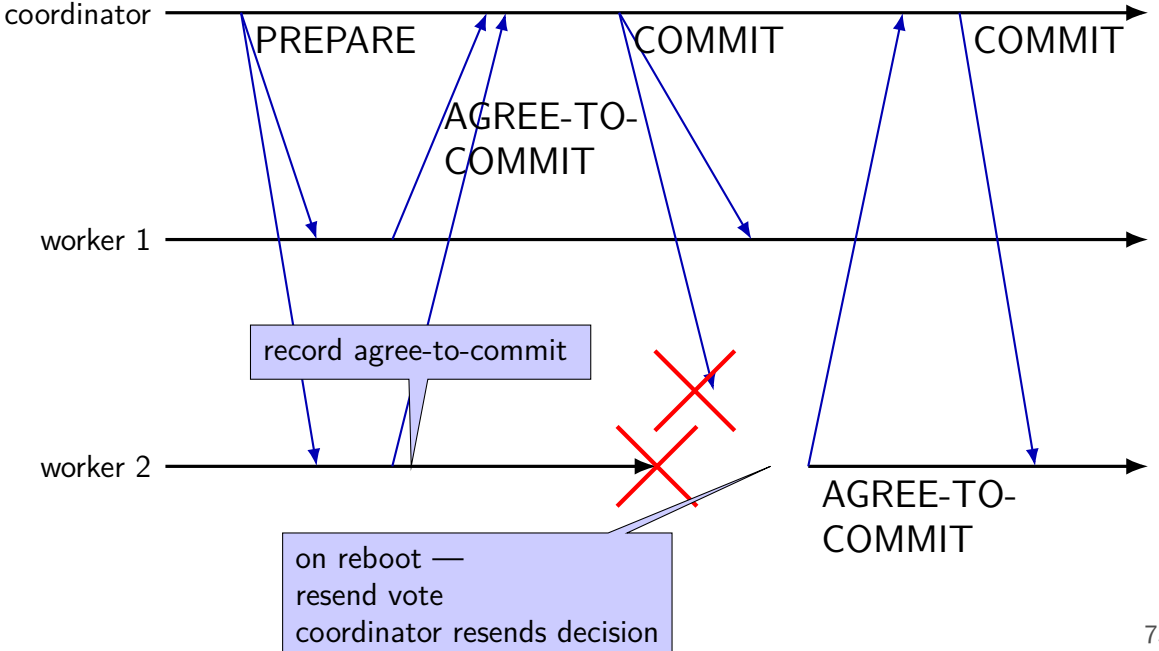
- option 1: worker resends vote (coordinator resends outcome)
- option 2?: coordinator resends outcome somehow? (but how would it know)

NB: coordinator can't give up

TPC: worker failure during commit (1)



TPC: worker failure during commit (1)



backup slides

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*

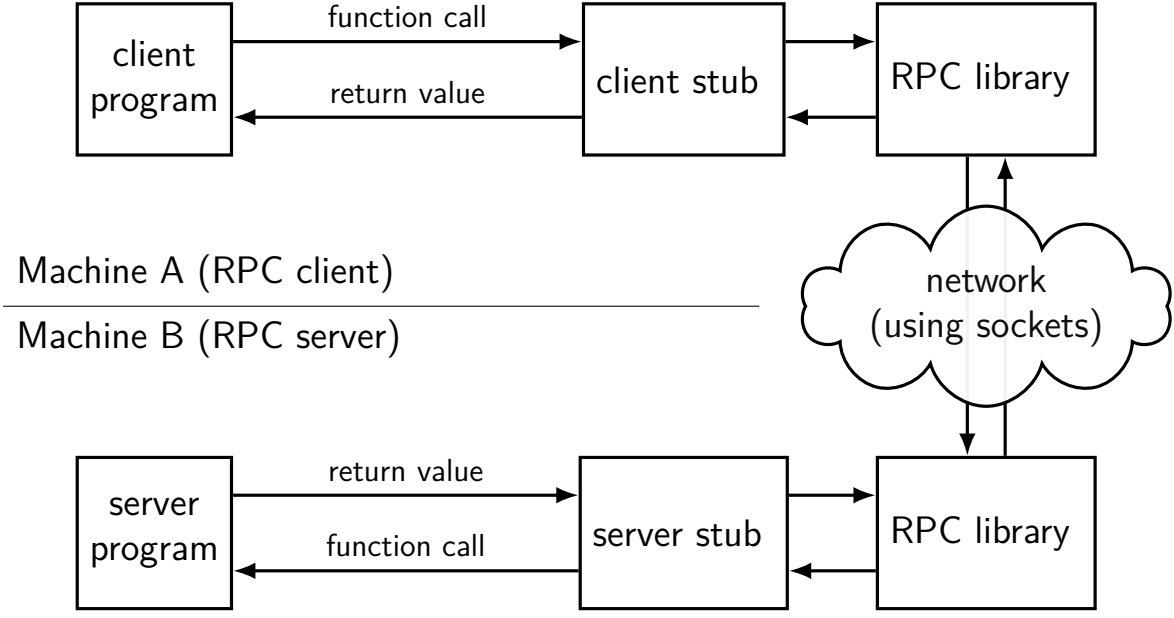
stubs = wrapper functions that stand in for other machine

calling remote procedure? call the stub

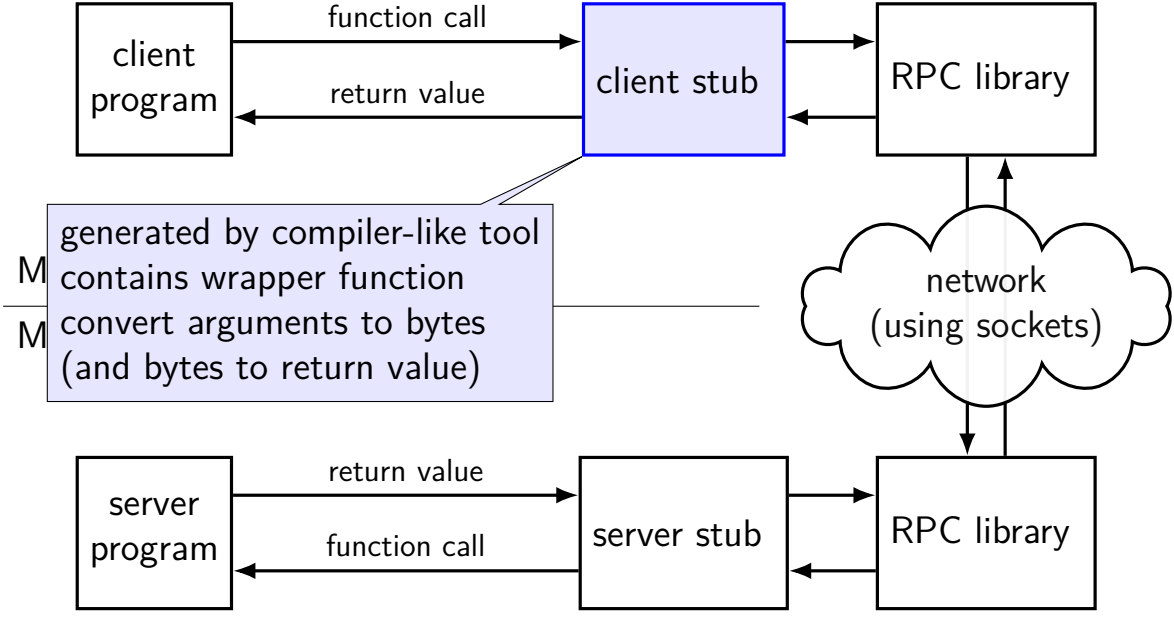
- same prototype are remote procedure

implementing remote procedure? a stub function calls you

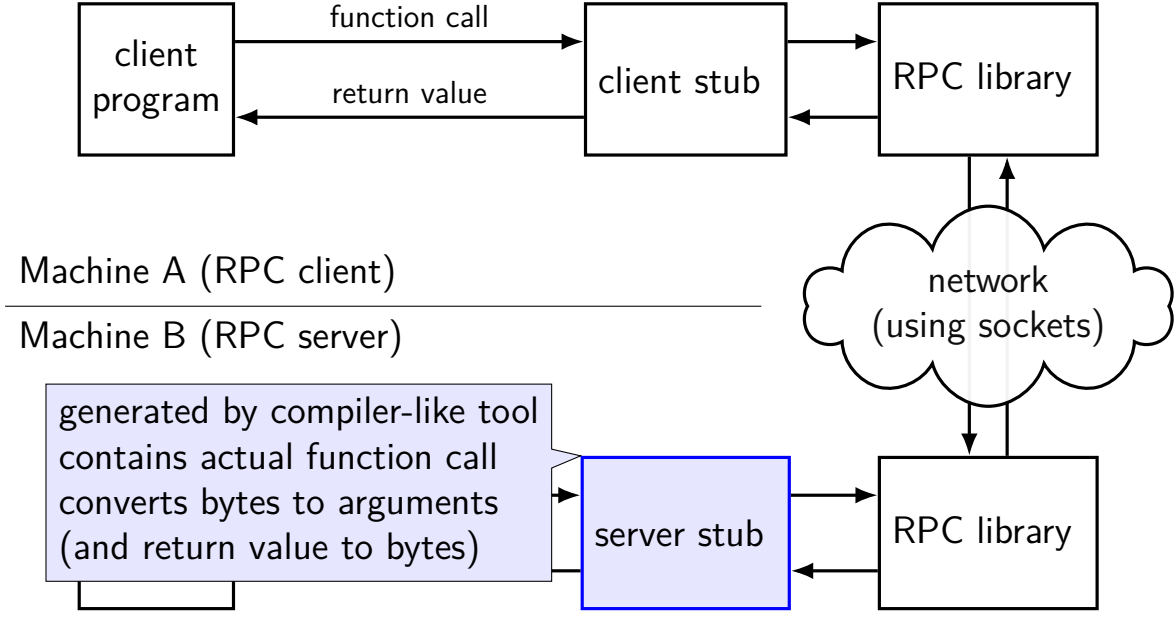
typical RPC data flow



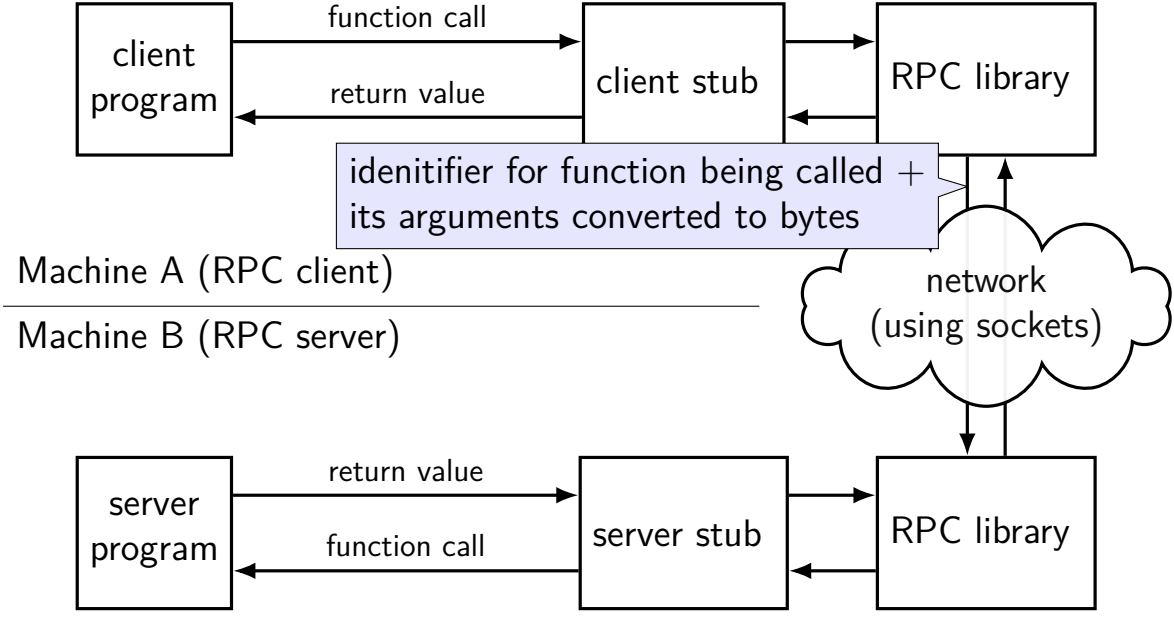
typical RPC data flow



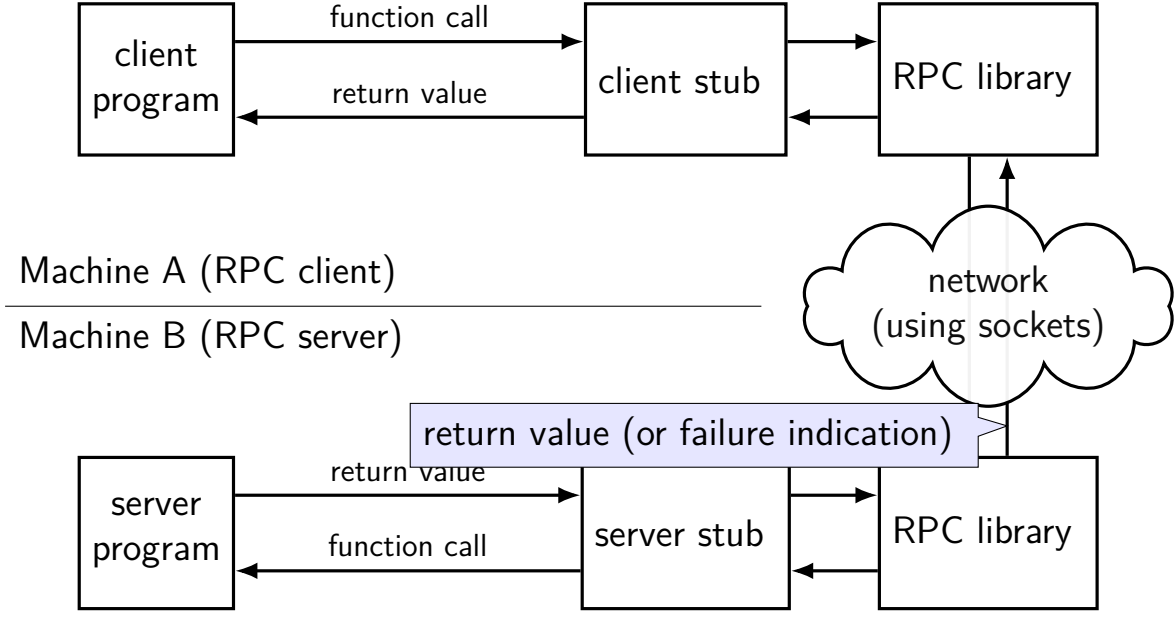
typical RPC data flow



typical RPC data flow



typical RPC data flow



RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->path() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->path() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->path() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 1)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status MakeDirectory(ServerContext *context,
                         const MakeDirArgs* args,
                         Empty *result) {
        std::cout << "MakeDirectory(" << args->path() << ")\n";
        if (-1 == mkdir(args->path().c_str()) {
            return Status(StatusCode::UNKNOWN, strerror(errno));
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (method 2)

```
class DirectoriesImpl : public Directories::Service {
public:
    Status ListDirectory(ServerContext *context,
                         const ListDirArgs* args,
                         DirectoryList *result) {
        ...
        for (...) {
            result->add_entry(...);
        }
        return Status::OK;
    }
    ...
};
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
            grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC server implementation (starting)

```
DirectoriesImpl service;
ServerBuilder builder;
builder.AddListeningPort("127.0.0.1:43534",
                         grpc::InsecureServerCredentials());
builder.RegisterService(&service);
unique_ptr<Server> server = builder.BuildAndStart();
server->Wait();
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 1)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; MakeDirectoryArgs args; Empty empty;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &empty);
if (!status.ok()) { /* handle error */ }
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```

RPC client implementation (method 2)

```
unique_ptr<Channel> channel(
    grpc::CreateChannel("127.0.0.1:43534"),
    grpc::InsecureChannelCredentials()));
unique_ptr<Directories::Stub> stub(Directories::NewStub(channel));
ClientContext context; ListDirectoryArgs args; DirectoryList list;
args.set_name("/directory/name");
Status status = stub->MakeDirectory(&context, args, &list);
if (!status.ok()) { /* handle error */ }
for (int i = 0; i < list.entries_size(); ++i) {
    cout << list.entries(i).name() << endl;
}
```