two phase commit (con't) / networked FS

last time

RPC — what's in IDLs

RPC calling in gRPC

ways RPC is not transparent

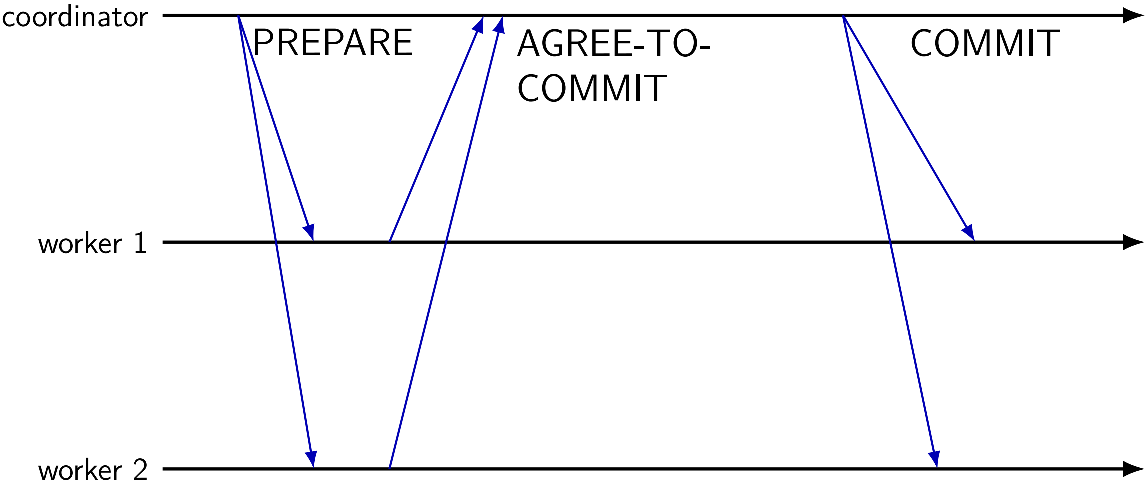
distributed transaction problem

two-phase commit idea

- coordinator + workers all agree to commit: commit
- default outcome: abort
- coordinator collects worker responses, distributes outcome
- logs to recover from anything failing without changing mind

state machines to represent protocol

TPC: normal operation

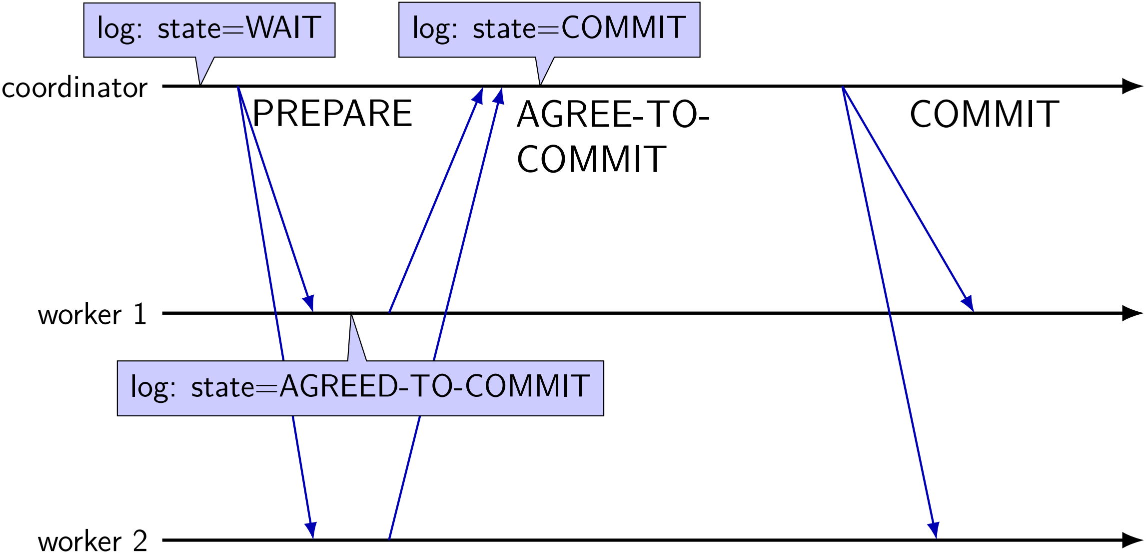


PREPARE

AGREE-TO-COMMIT

COMMIT

TPC: normal operation



log: state=WAIT

log: state=COMMIT

coordinator

PREPARE

AGREE-TO-COMMIT

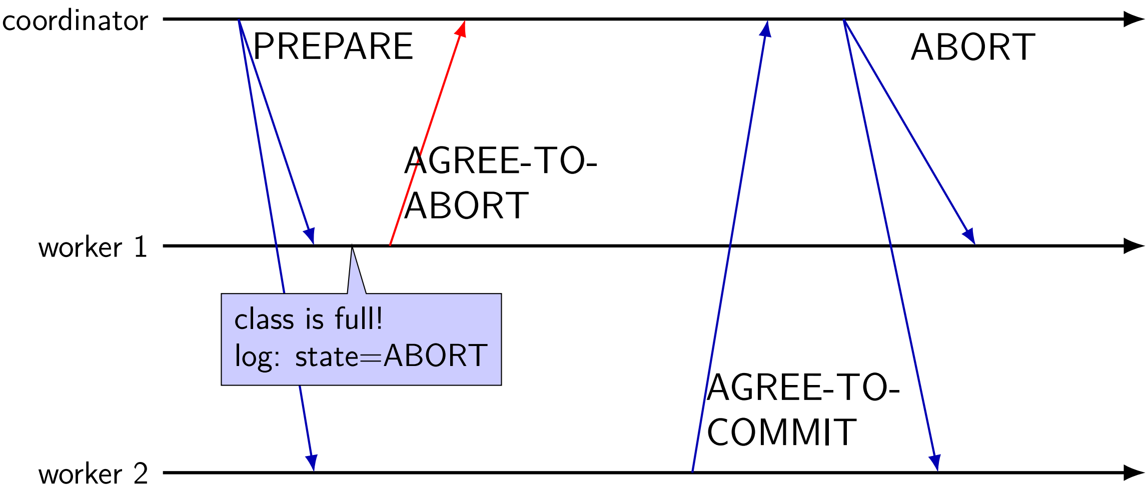
COMMIT

worker 1

log: state=AGREED-TO-COMMIT

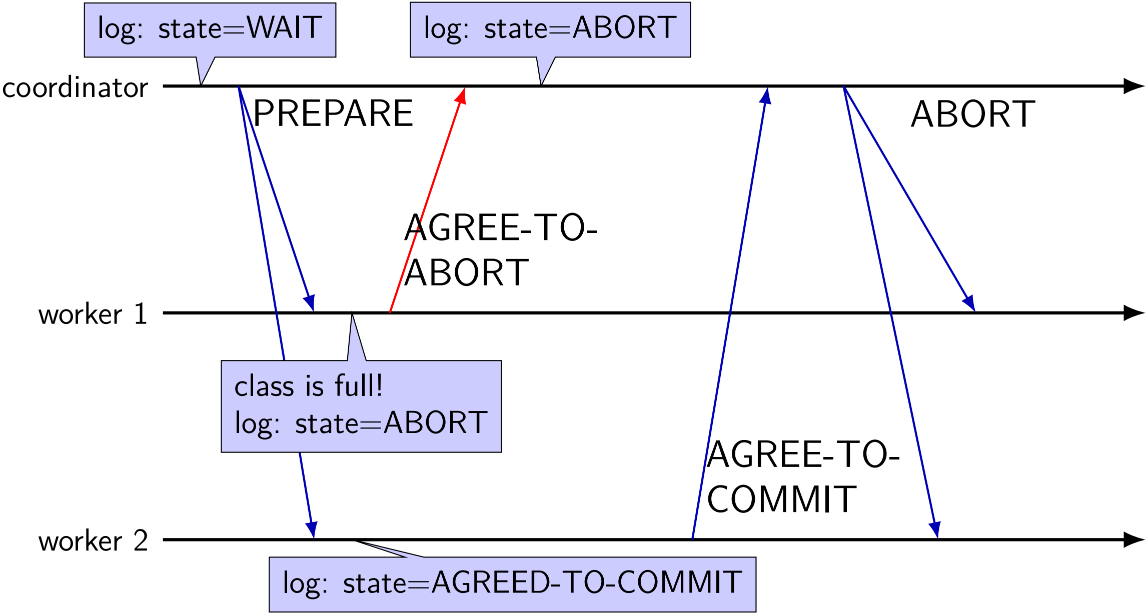
worker 2

TPC: normal operation — conflict



class is full!
log: state=ABORT

TPC: normal operation — conflict

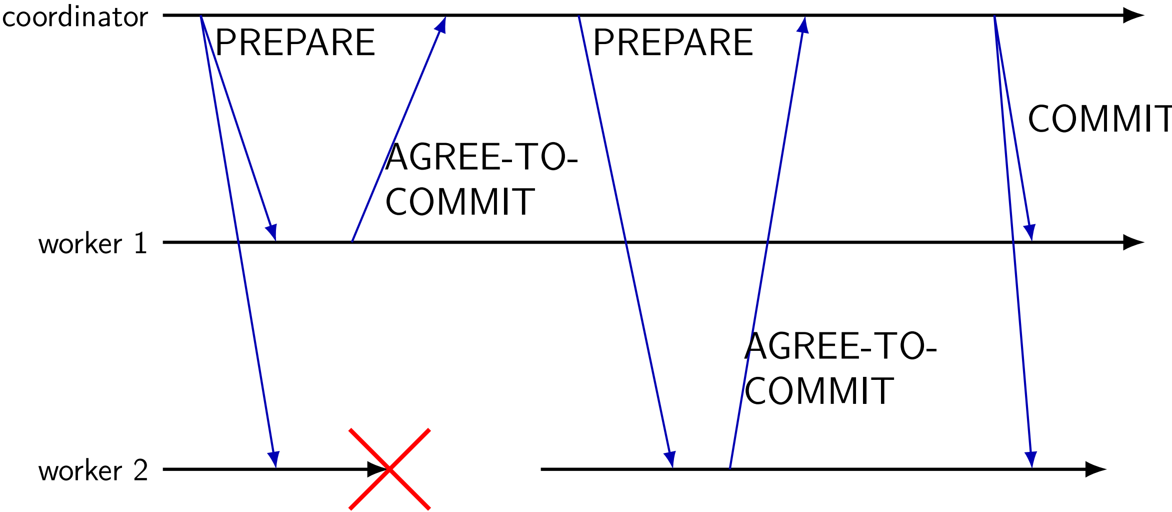


some failure cases

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)



coordinator

worker 1

worker 2

PREPARE

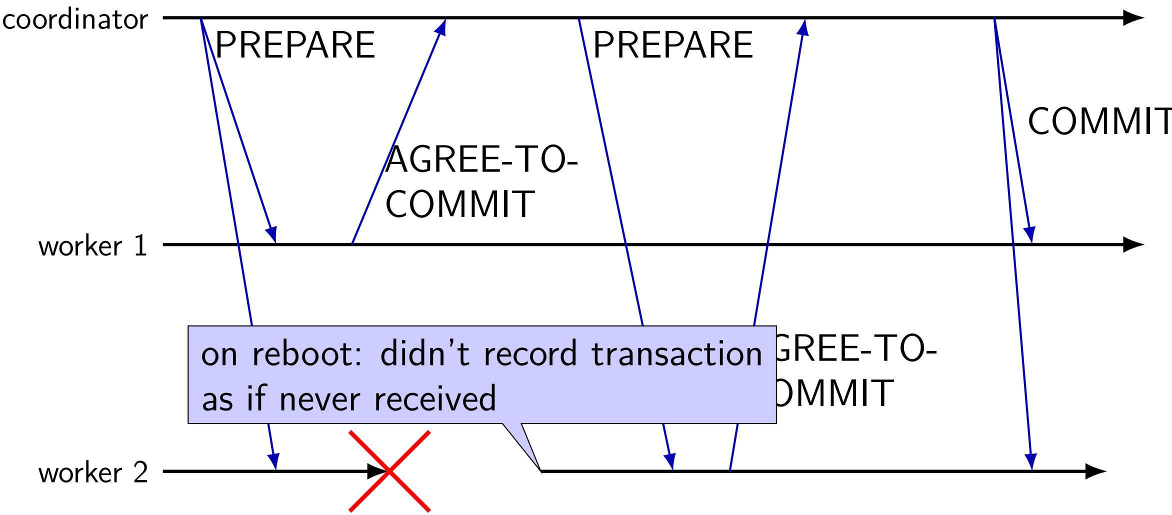
AGREE-TO-COMMIT

PREPARE

AGREE-TO-COMMIT

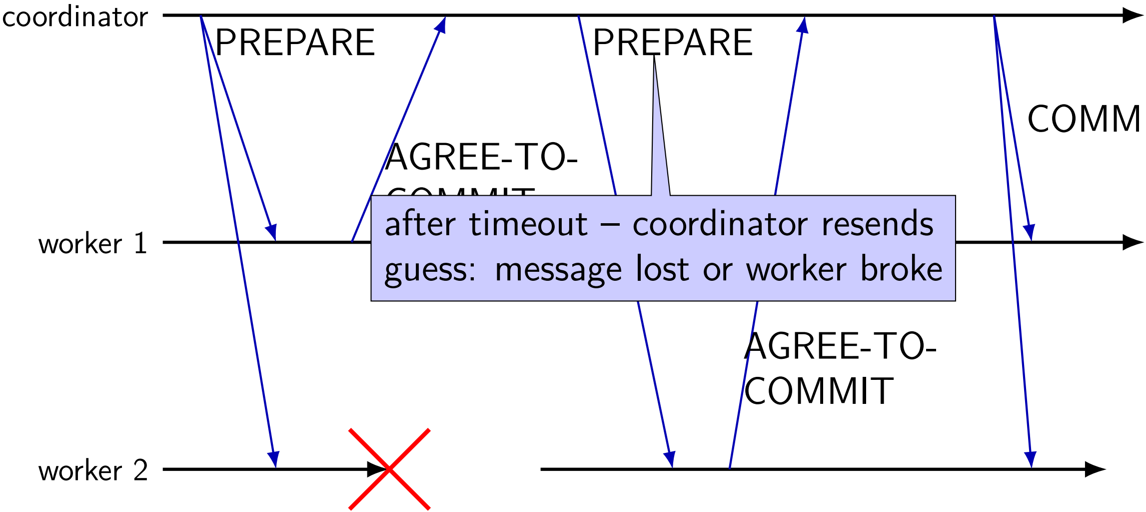
COMMIT

TPC: worker fails after prepare (1)



on reboot: didn't record transaction as if never received

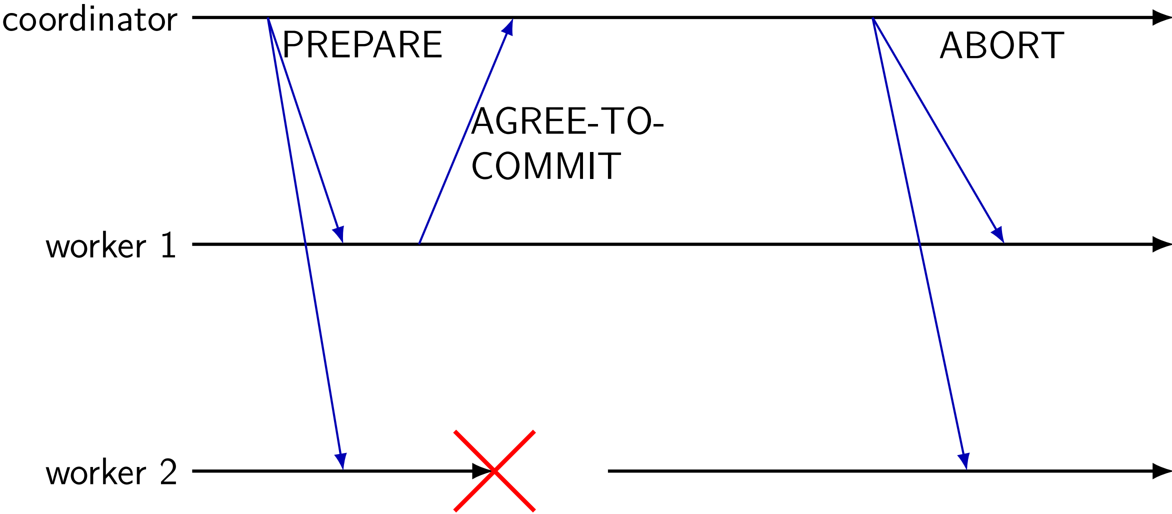
TPC: worker fails after prepare (1)



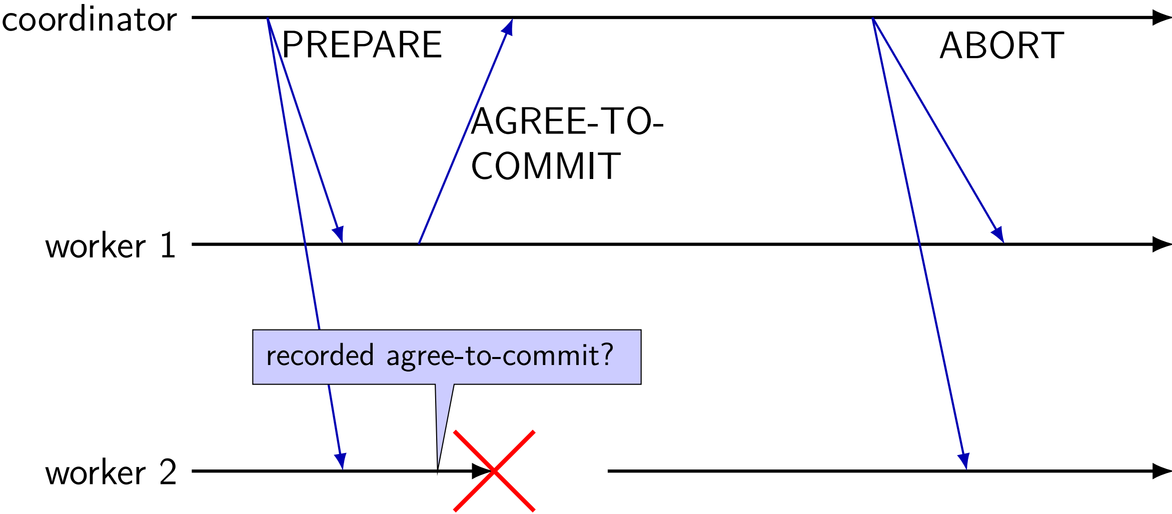
after timeout – coordinator resends

guess: message lost or worker broke

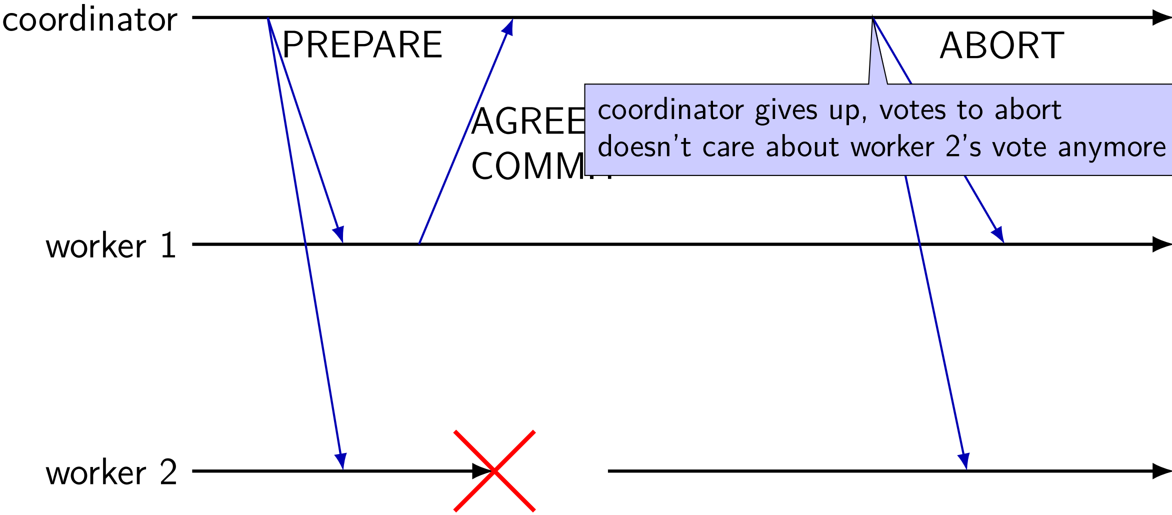
TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)

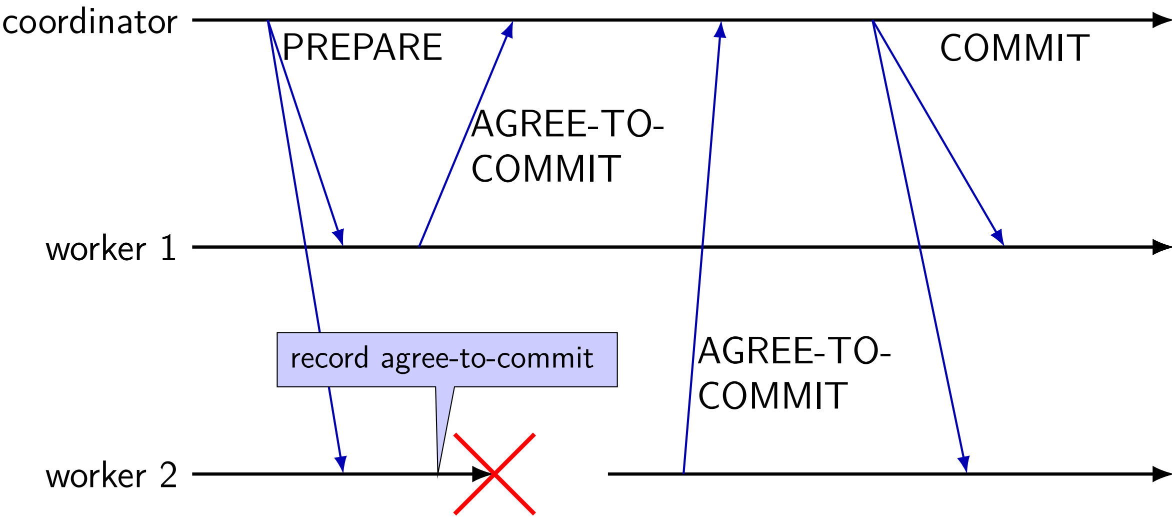


TPC: worker fails after prepare (2)

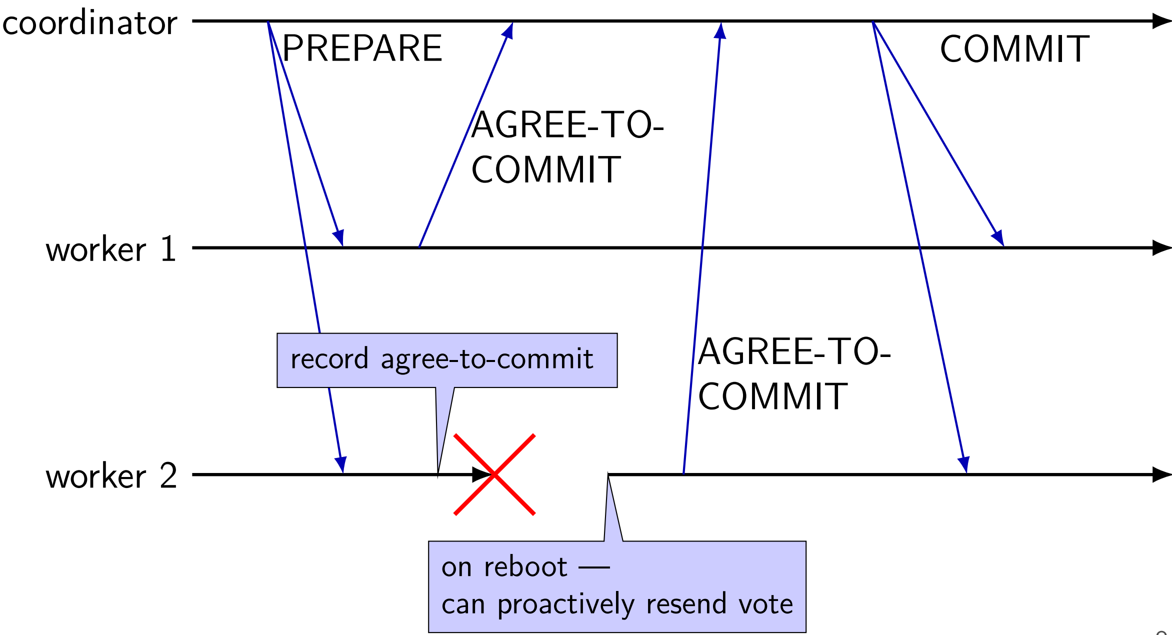


coordinator gives up, votes to abort
doesn't care about worker 2's vote anymore

TPC: worker fails after prepare (3)



TPC: worker fails after prepare (3)



record agree-to-commit

on reboot —
can proactively resend vote

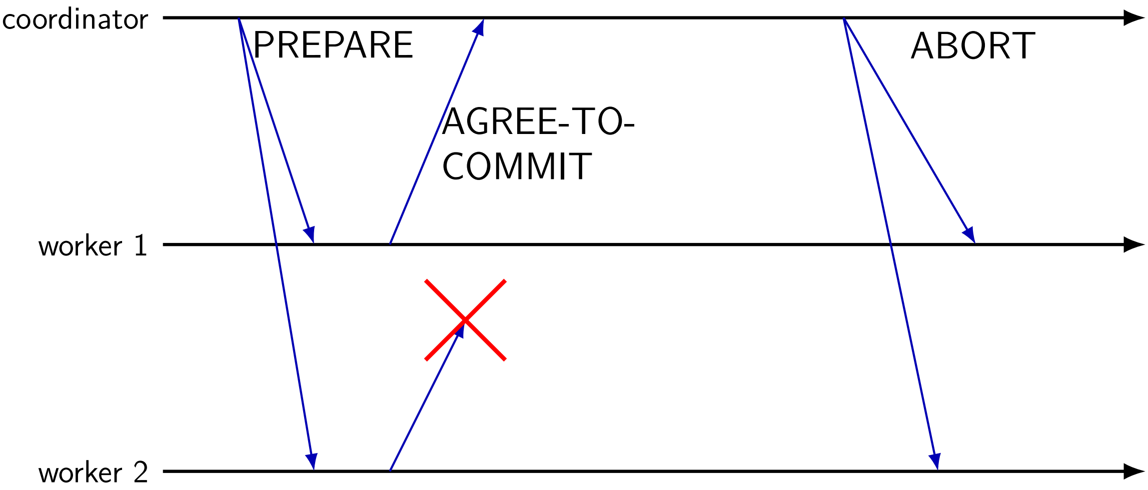
network failure after during voting?

network failure during voting $\approx$ node failure

same options:

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

TPC: network failure (1)



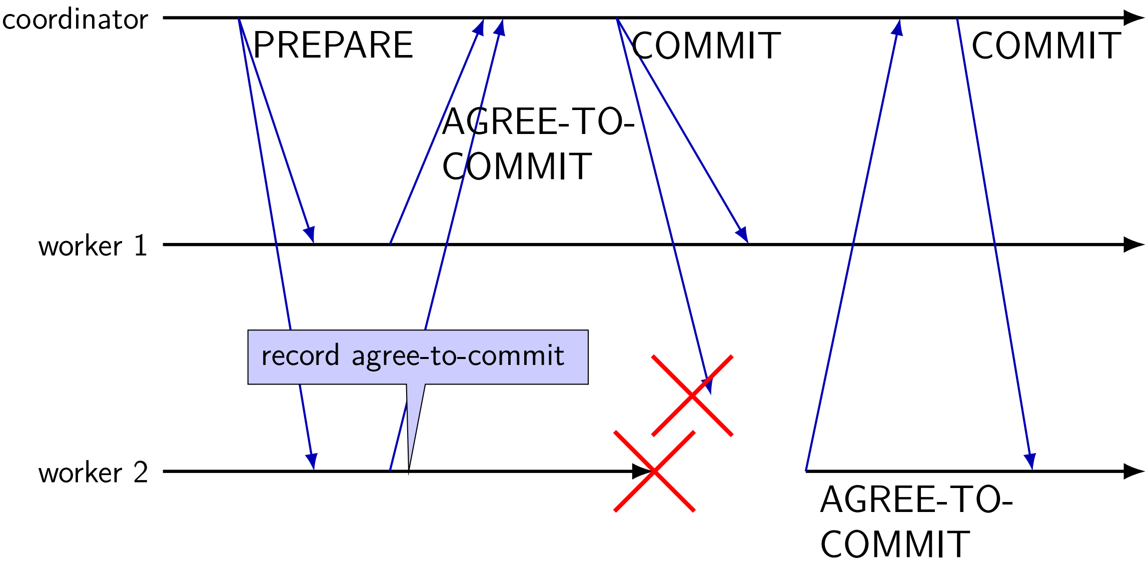
worker failure during commit

worker failure during commit?

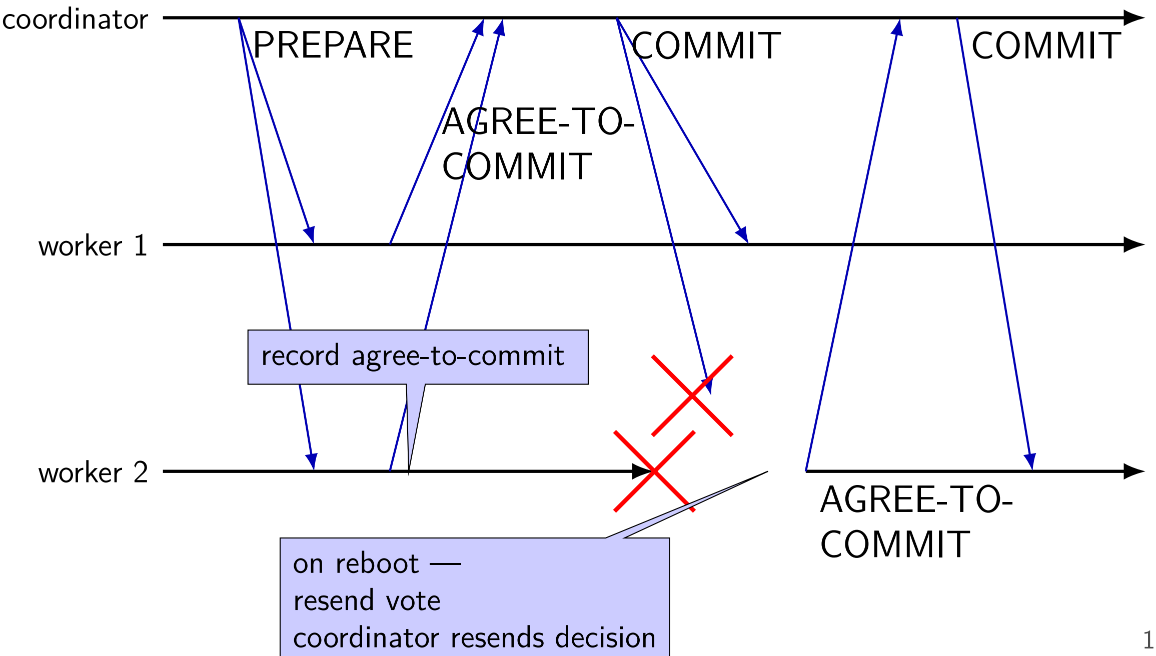
- option 1: worker resends vote (coordinator resends outcome)
- option 2?: coordinator resends outcome somehow? (but how would it know)

NB: coordinator can't give up

TPC: worker failure during commit (1)



TPC: worker failure during commit (1)



coordinator

worker 1

worker 2

PREPARE

AGREE-TO-COMMIT

COMMIT

AGREE-TO-COMMIT

COMMIT

record agree-to-commit

on reboot —
resend vote
coordinator resends decision

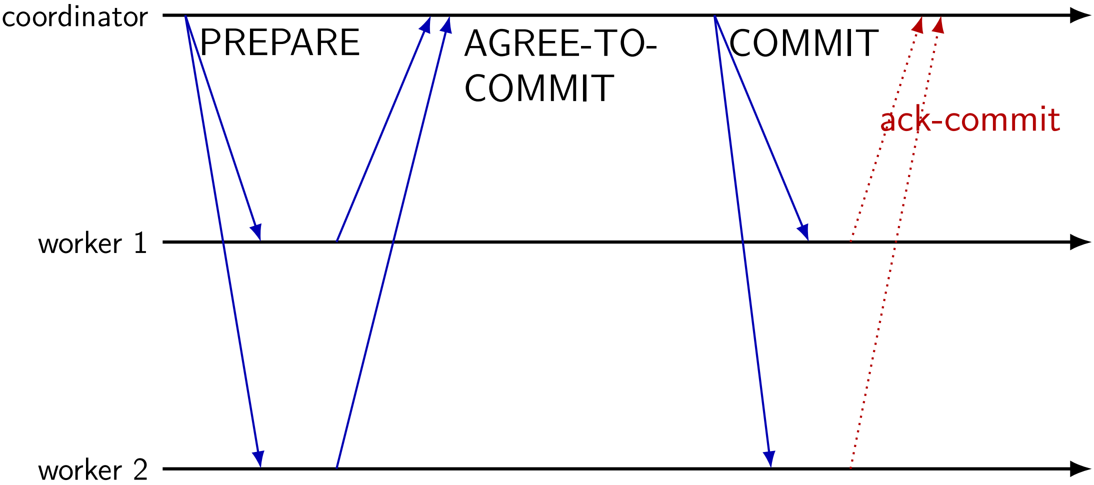
worker failure during commit

worker failure during commit?

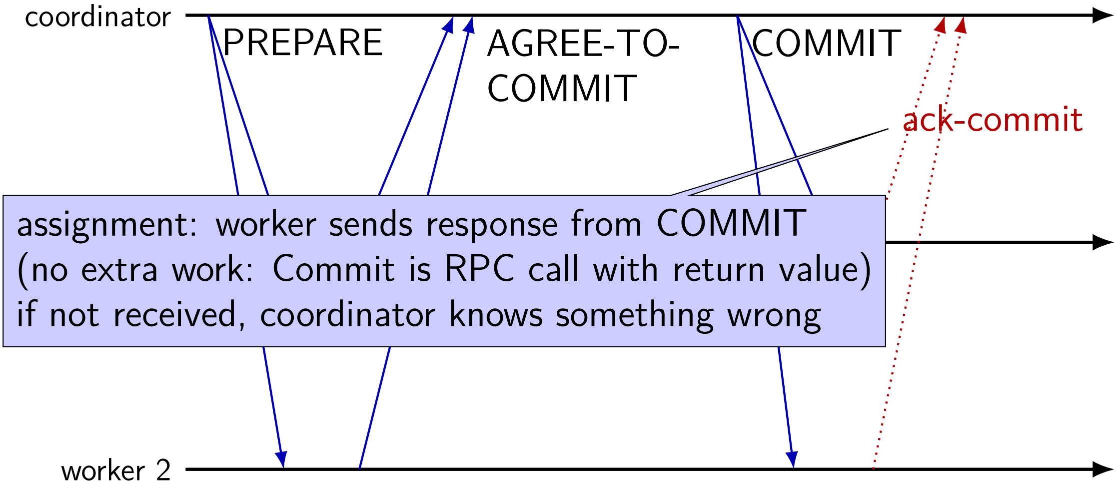
- option 1: worker resends vote (coordinator resends outcome)
- option 2?: coordinator resends outcome somehow? (but how would it know)

NB: coordinator can't give up

alternate approach: worker ACKs

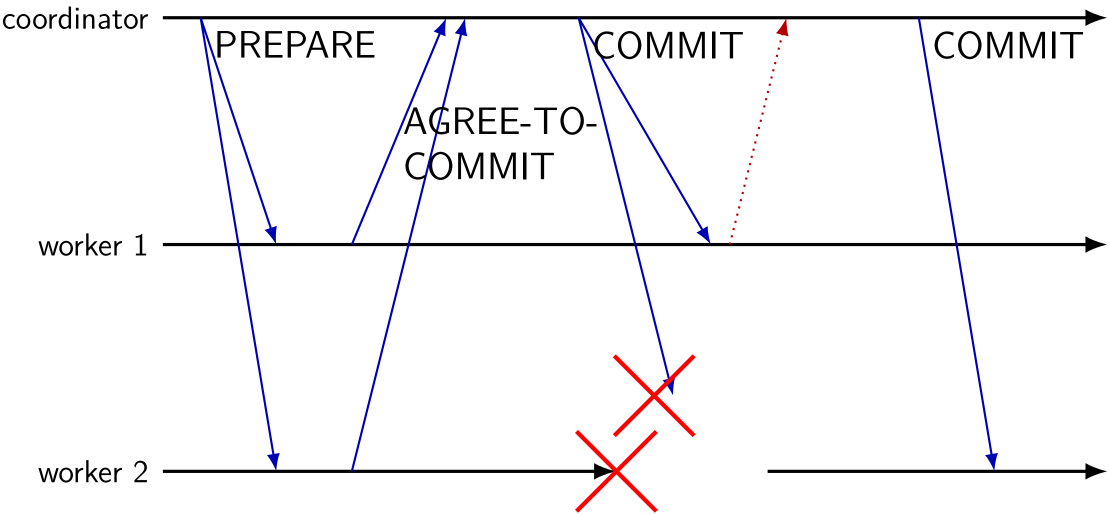


alternate approach: worker ACKs

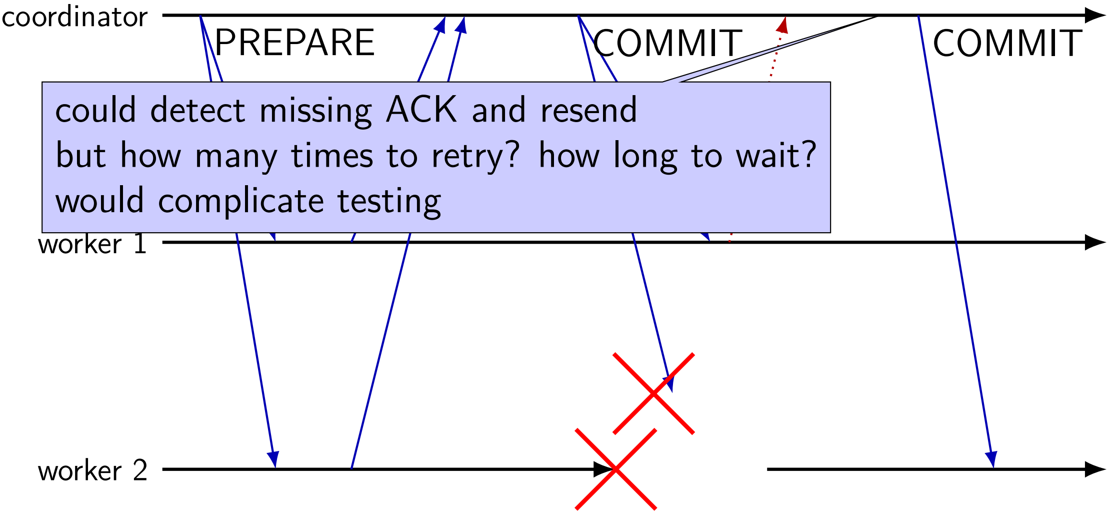


assignment: worker sends response from COMMIT
(no extra work: Commit is RPC call with return value)
if not received, coordinator knows something wrong

coordinator resend automatically



coordinator resend automatically



could detect missing ACK and resend
but how many times to retry? how long to wait?
would complicate testing

two-phase commit assignment

two phase commit assignment

store *single value* across workers

single coordinator sends messages to/from workers to change values

- workers current value can be queried directly

goal: several replicas all have same value *or unavailable*

…even if failures

assignment: RPC

coordinator talks to worker by making RPC calls

workers only talk to coordinator by replying to RPC

- example: make "prepare" call, worker's "agree-to-X" is return value

RPC system detects worker being down, network errors, etc.

- become Python exception in coordinator

coordinator verifies Commit/Abort received instead of worker asking again

- automatic: Commit/Abort message is RPC call; RPC call fails if problem

assignment: failure recovery

to simplify assignment: always return error if you detect failure

assume testing code/user will restart the coordinator+workers

coordinator sends messages to workers on reboot to recover

- resend prepare or commit, abort, etc.

assignment: failure types

send RPC and

- it gets lost
- it gets sent, but acknowledge/reply is lost
- it gets sent, but delayed until after another RPC

assignment: failure types

send RPC and

- it gets lost
- it gets sent, but acknowledge/reply is lost
- it gets sent, but delayed until after another RPC

transaction/sequence numbers

each RPC is a separate connection

potential for RPC sent before restarting coordinator to be received after

you'll need to detect reordering

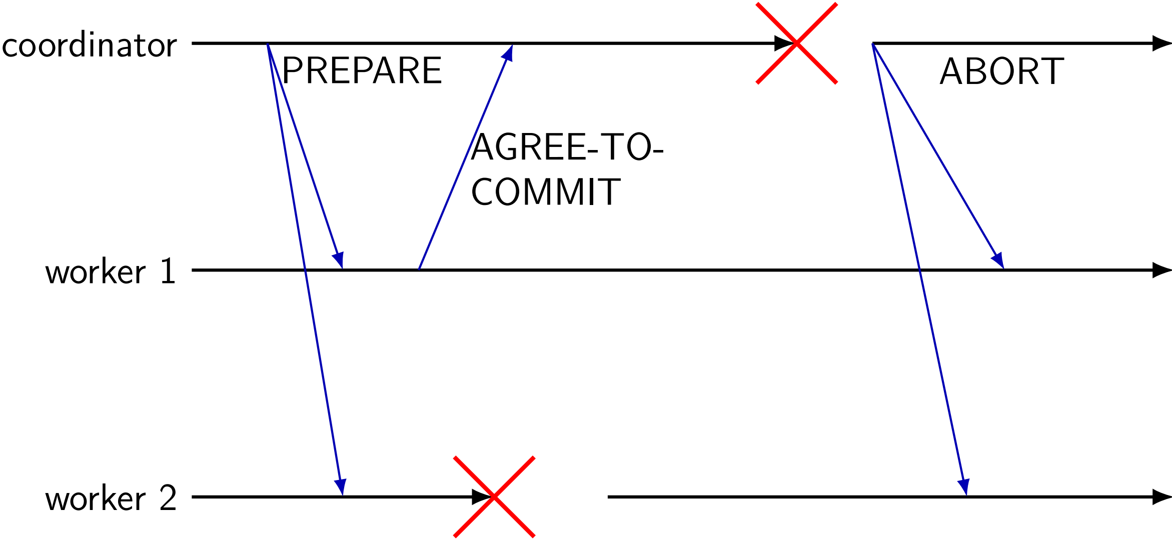
note on skeleton code

I included some extra methods (from my reference impl) in the IDL file in the skeleton earlier

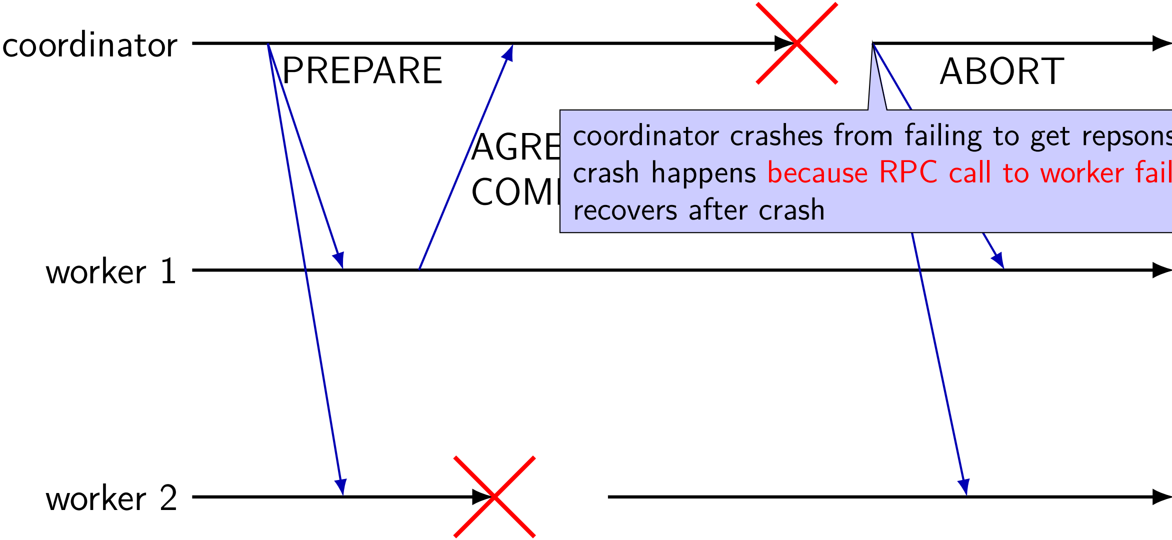
removed from updated version from yesterday

if you started early (while assignment still tentative), might want to ignore and/or use updated skeleton code

assignment: fails during prepare

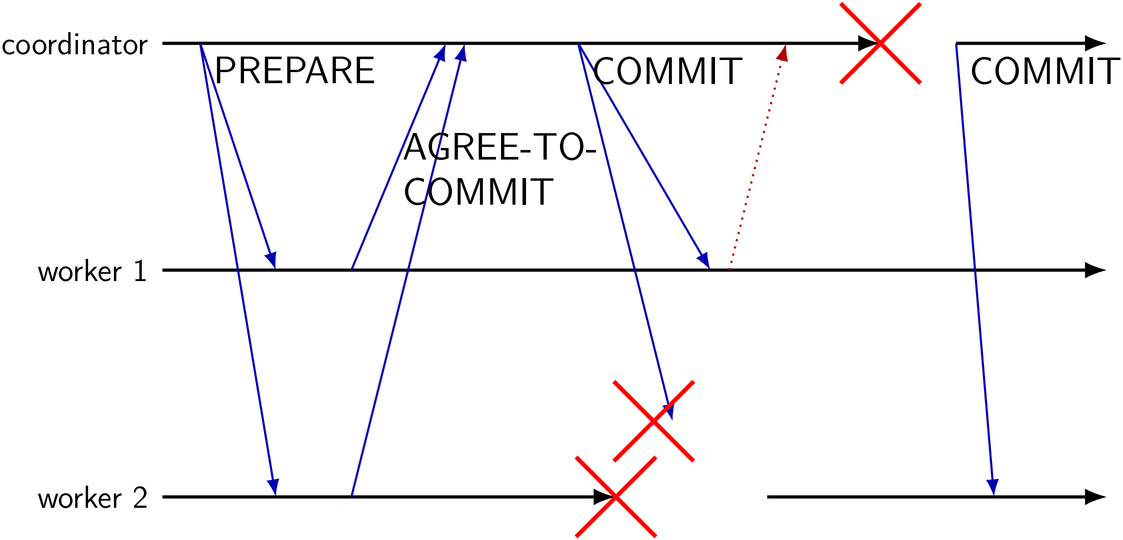


assignment: fails during prepare

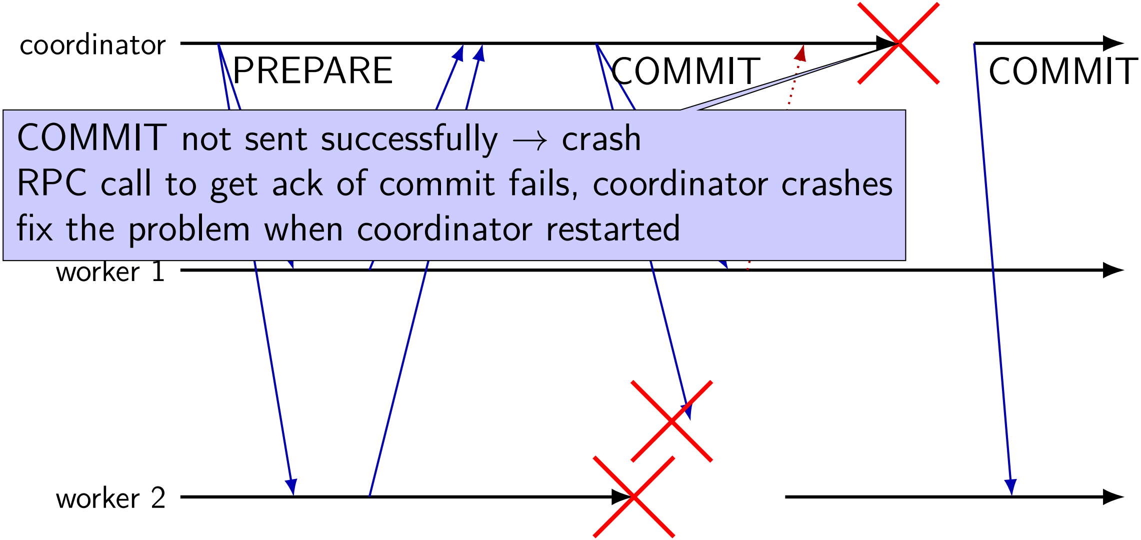


coordinator crashes from failing to get repsons
crash happens because RPC call to worker fail
recovers after crash

assignment: failuring during commit

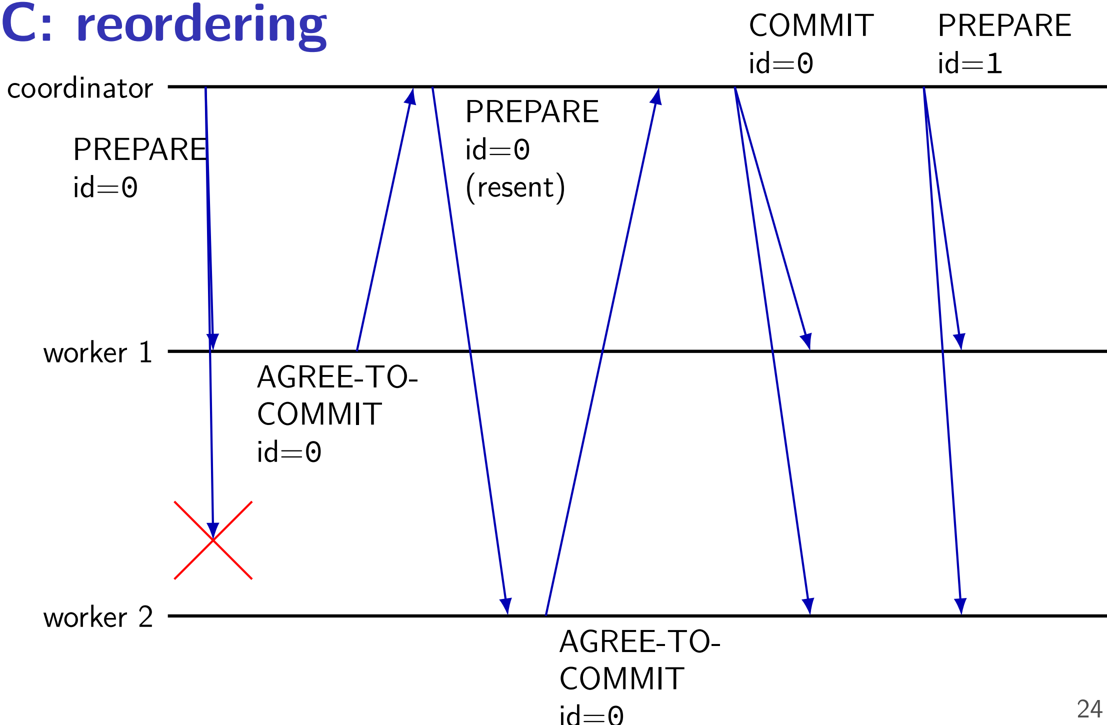


assignment: failuring during commit



COMMIT not sent successfully → crash
RPC call to get ack of commit fails, coordinator crashes
fix the problem when coordinator restarted

TPC: reordering



coordinator

worker 1

worker 2

PREPARE
id=0

AGREE-TO-
COMMIT
id=0

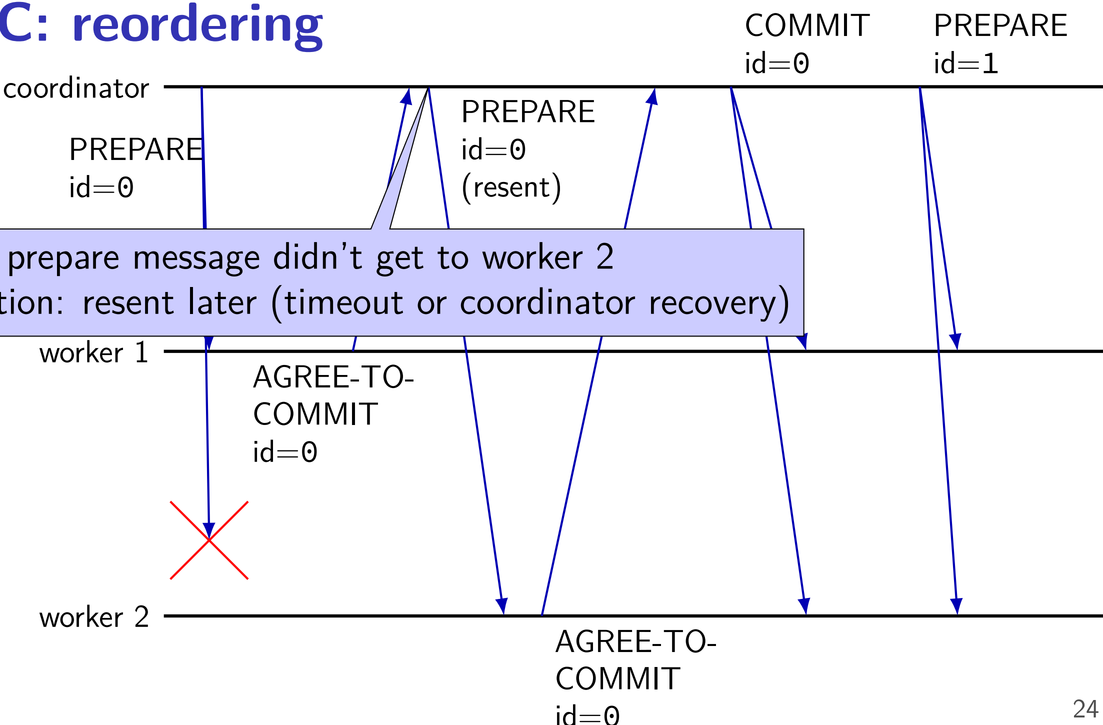
PREPARE
id=0
(resent)

AGREE-TO-
COMMIT
id=0

COMMIT
id=0

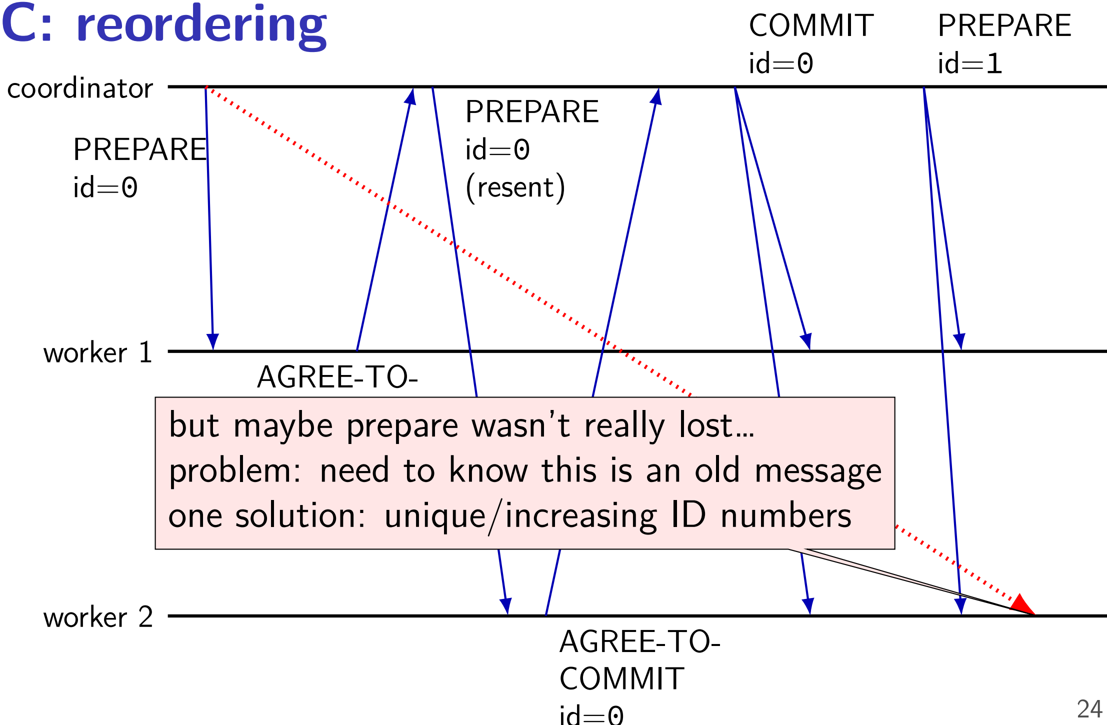
PREPARE
id=1

TPC: reordering



first prepare message didn't get to worker 2
solution: resent later (timeout or coordinator recovery)

TPC: reordering



but maybe prepare wasn't really lost...

problem: need to know this is an old message

one solution: unique/increasing ID numbers

message reordering and assignment

assignment: you need to worry about reordering

- connections prevent reordering, but...
- RPC system doesn't prevent it: can use multiple connections

problem: old request *seems to fail*, but is actually slow

you repeat old request again

later on slow old request reaches machine → must be ignored!

solution: sequence numbers or transactions ID and/or timestamps

- some way to tell "this is old"

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

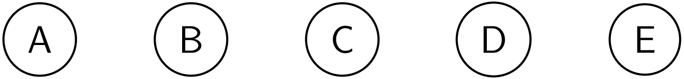
goal: work (including updates!) despite a few failing nodes

just require “enough” nodes to be working

for now — assume fail-stop

nodes don’t respond or tell you if broken

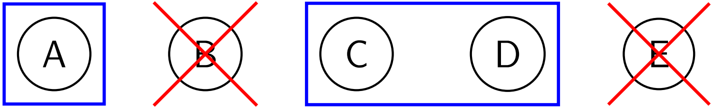
quorums (1)



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

any quorum enough — okay if some nodes fail

quorums (1)



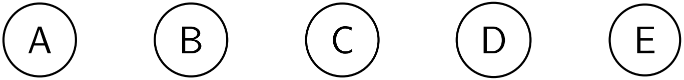
perform read/write with vote of any *quorum* of nodes

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

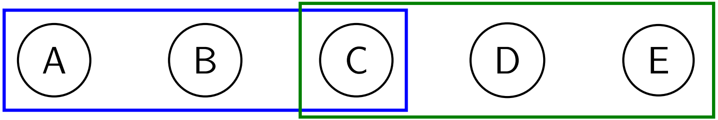
e.g. every operation requires majority of nodes

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

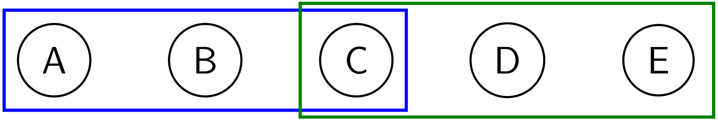
- e.g. every operation requires majority of nodes

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

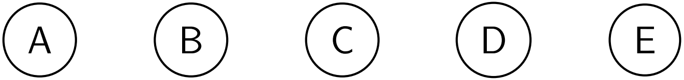
e.g. every operation requires majority of nodes

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

quorums (3)



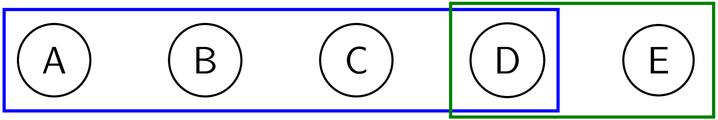
sometimes vary quorum based on operation type

example: update quorum = 4 of 5; read quorum = 2 of 5

requirement: read *overlaps* with last update

compromise: better performance sometimes, but tolerate less failures

quorums (3)



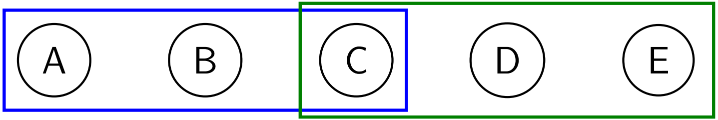
sometimes vary quorum based on operation type

example: **update** quorum = 4 of 5; **read** quorum = 2 of 5

requirement: read *overlaps* with last update

compromise: better performance sometimes, but tolerate less failures

quorums



details very tricky

- what about coordinator failures?
- how does recovery happen?
- what information needs to be logged?
- “catching up” nodes that aren’t part of several updates

full details: lookup Raft or Paxos

Raft sketch

Raft: quorum consensus algorithm

leader election: agree on leader (≈ coordinator)

- elect new leader on leader failure
- constraint: can't be leader if not up-to-date with quorum
- enforcement: quorum must elect each leader
- nodes only believe in in latest (highest numbered) leader

leader uses other machines (followers) as remote logs

leader ensures quorum logs operations (≈ commits them)

lots of tricky details around failures

- e.g. leader starts sending transaction to log + fails

quorums for Byzantine failures

just overlap not enough

problem: node can give inconsistent votes

tell A “I agree to commit”, tell B “I do not”

need to confirm consistency of votes with other notes

need *supermajority*-type quorums

f failures — $3f + 1$ nodes

full details: lookup PBFT

network filesystems

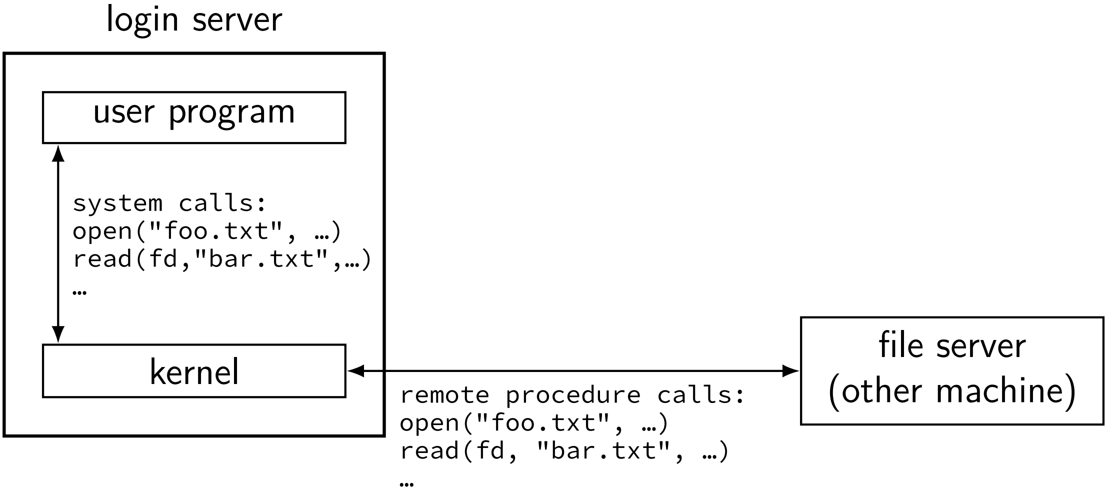
department machines — your files always there

- even though several machines to log into

how? there's a *network file server*

filesystem is backed by a remote machine

simple network filesystem



login server

user program

```
system calls:
open("foo.txt", …)
read(fd,"bar.txt",…)
…
```

kernel

```
remote procedure calls:
open("foo.txt", …)
read(fd, "bar.txt", …)
…
```

file server
(other machine)

system calls to RPC calls?

just turn system calls into RPC calls?

(or calls to the kernel's internal fileystem abstraction, e.g. Linux's Virtual File System layer)

has some problems:

what state does the server need to store?

what if a client machine crashes?

what if the server crashes?

how fast is this?

state for server to store?

open file descriptors?

- what file
- offset in file

current working directory?

gets pretty expensive across N clients, each with many processes

if a client crashes?

well, it hasn't responded in N minutes, so

can the server delete its open file information yet?

what if its cable is plugged back in and it works again?

if the server crashes?

well, first we restart the server/start a new one…

then, what do clients do?

probably need to restart to?

can we do better?

NFSv2

NFS (Network File System) version 2

standardized in RFC 1094 (1989)

based on RPC calls

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

file ID: opaque data (support multiple implementations)
example implementation: device+inode number+“generation number”

NFSv2 client versus server

clients: file descriptor →server name, file ID, offset

client machine crashes? mapping automatically deleted

- “fate sharing”

server: convert file IDs to files on disk

- typically find unique number for each file
- usually by inode number

server doesn’t get notified unless client is using the file

file IDs

device + inode + "*generation number*"?

generation number: incremented every time inode reused

file IDs

device + inode + “*generation number*”?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file
- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

file IDs

device + inode + “*generation number*”?

generation number: incremented every time inode reused

problem: file removed while client has it open

later client tries to access the file
- maybe inode number is valid *but for different file*
- inode was deallocated, then reused for new file

Linux filesystems store a “generation number” in the inode
- basically just to help implement things like NFS

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) → file ID

GETATTR(file ID) → (file size, owner, ...)

READ(file ID, offset, length) → data

WRITE(file ID, data, offset) → success/failure

CREATE(dir file ID, filename, metadata) → file ID

REMOVE(dir file ID, filename) → success/failure

SETATTR(file ID, size, owner, ...) → success/failure

“stateless protocol” — no open/close/etc.
each operation stands alone

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset “cookie”) → (names and file IDs, next offset “cookie”)

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset “cookie”) →
(names and file IDs, next offset “cookie”)

pattern: client storing opaque tokens

- for client: remember this, don’t worry about what it means

tokens represent something the server can easily lookup

- file IDs: inode, etc.
- directory offset cookies: byte offset in directory, etc.

strategy for making stateful service stateless

statefulness

stateful protocol (example: FTP, two-phase commit)

- previous things in connection matter
- e.g. logged in user
- e.g. current working directory
- e.g. where to send data connection

stateless protocol (example: HTTP, NFSv2)

- each request stands alone
- servers remember nothing about clients between messages
- e.g. file IDs for each operation instead of file descriptor

stateful versus stateless

in client/server protocols:

stateless: more work for client, less for server

- client needs to remember/forward any information
- can run multiple copies of server without syncing them
- can reboot server without restoring any client state

stateful: more work for server, less for client

- client sets things at server, doesn't change anymore
- hard to scale server to many clients (store info for each client
- rebooting server likely to break active connections

performance

before: reading/writing files/directories goes to local memory

- lots of work to use memory to cache, read-ahead

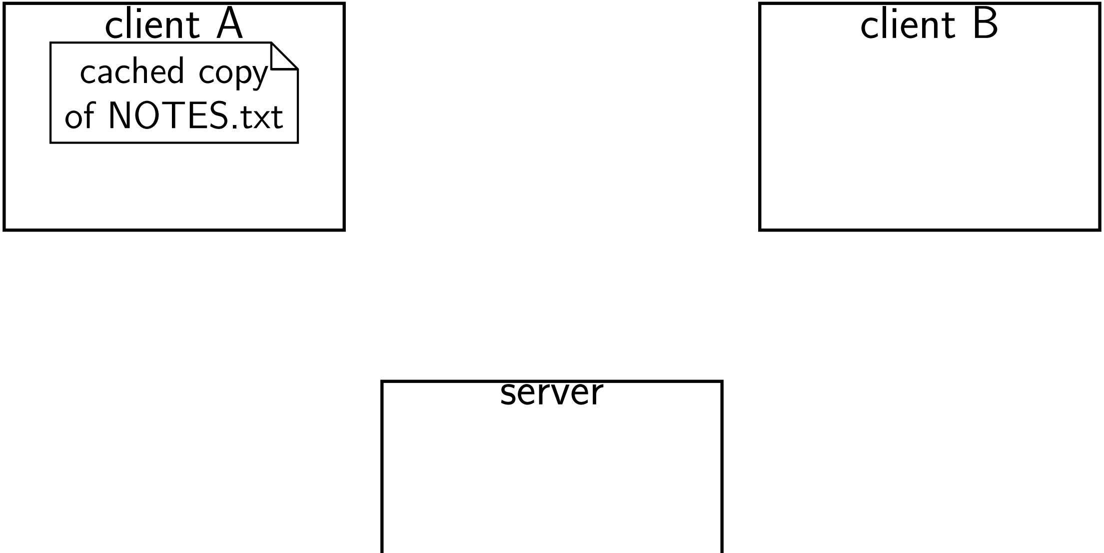
so open/read/write/close/rename/readdir/etc. take microseconds

- open that file? yes, I have the direntry cached
- read from that file? already in my memory

now: take milliseconds+

- open that file? let's ask the server if that's okay
- read from that file? let's copy it from the server
- etc.

updating cached copies?



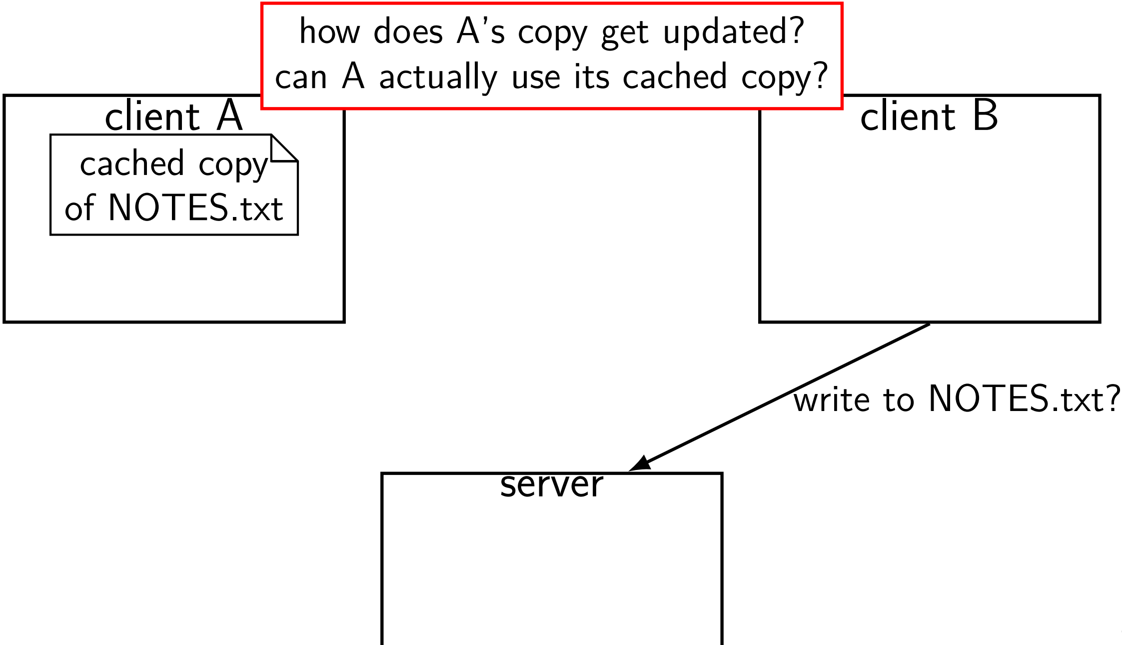
client A

cached copy of NOTES.txt

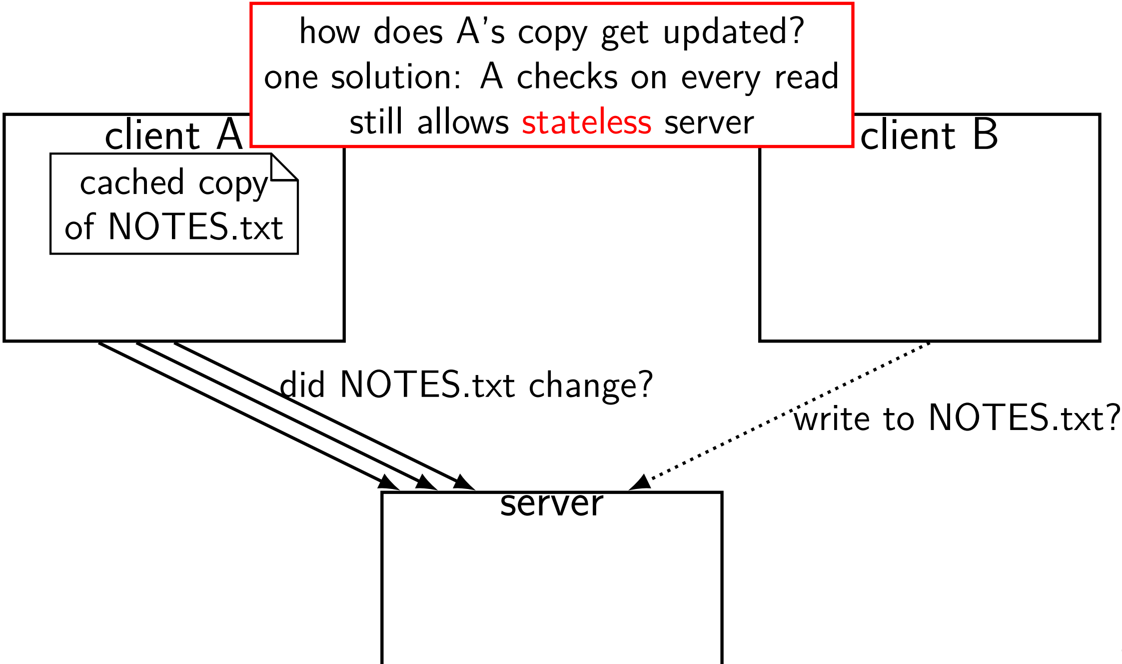
client B

server

updating cached copies?

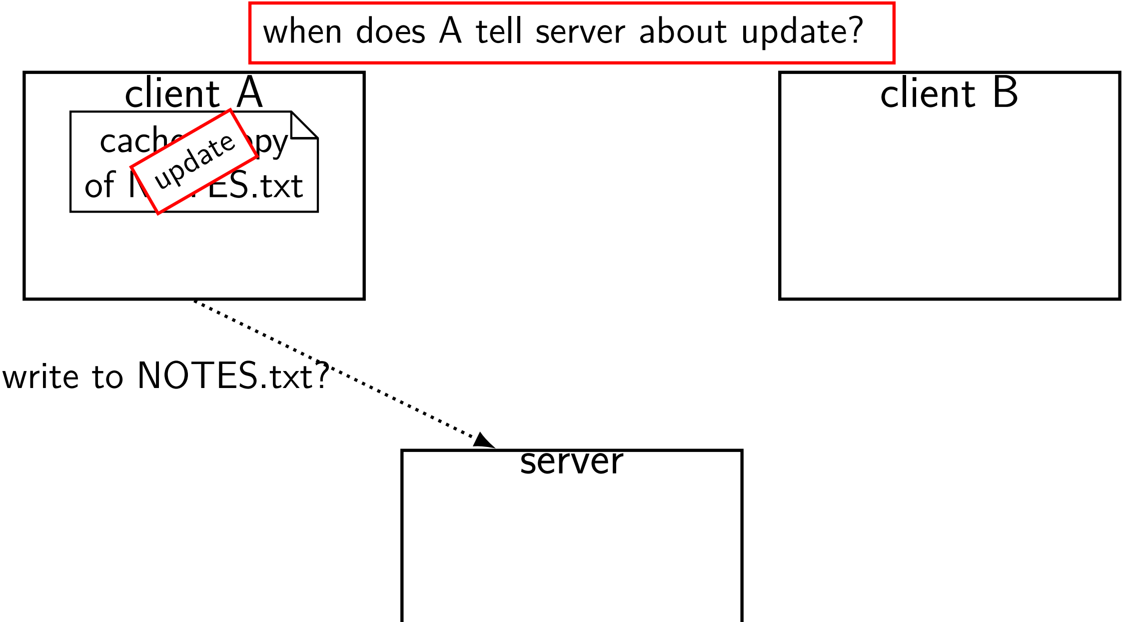

how does A's copy get updated?
can A actually use its cached copy?

updating cached copies?



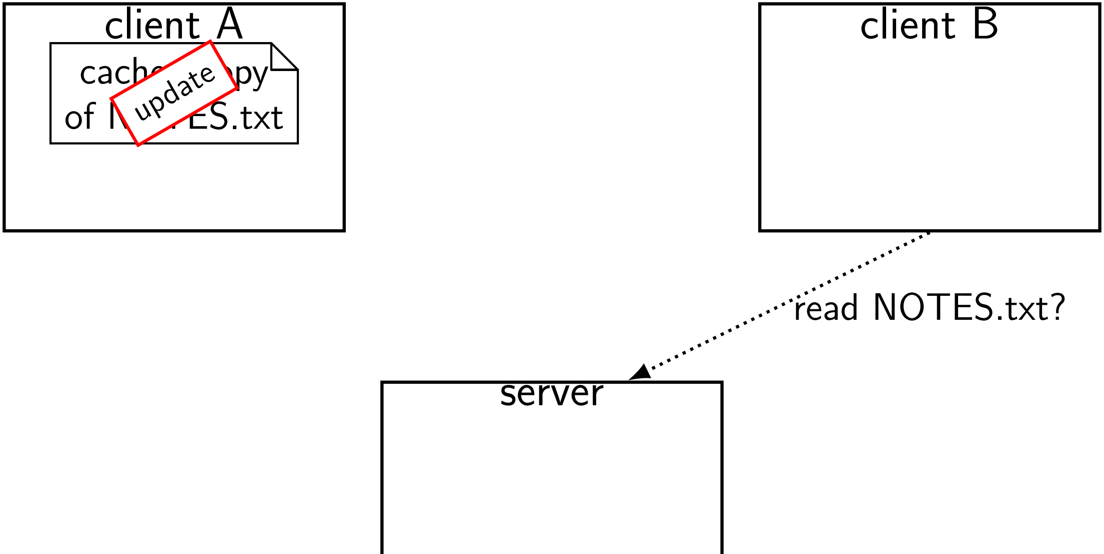
how does A's copy get updated?
one solution: A checks on every read
still allows stateless server

updating cached copies?



updating cached copies?

does B get updated version from A? how?



consistency with stateless server

always check server before using cached version

write through *all* updates to server

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

...but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

consistency with stateless server

always check server before using cached version

write through *all* updates to server

allows server to not remember clients

- no extra code for server/client failures, etc.

...but kinda destroys benefit of caching

- many milliseconds to contact server, even if not transferring data

NFSv3's solution: allow inconsistency

typical text editor/word processor

typical word processor:

opening a file:

open file, read it, load into memory, close it

saving a file:

open file, write it from memory, close it

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

two people saving a file?

have a word processor document on shared filesystem

Q: if you open the file while someone else is saving, what do you expect?

Q: if you save the file while someone else is saving, what do you expect?

observation: not things we really expect to work anyways

most applications don't care about accessing file while someone has it open

open to close consistency

a compromise:

opening a file checks for updated version

- otherwise, use latest cache version

closing a file writes updates from the cache

- otherwise, may not be immediately written

open to close consistency

a compromise:

opening a file checks for updated version
- otherwise, use latest cache version

closing a file writes updates from the cache
- otherwise, may not be immediately written

idea: as long as one user loads/saves file at a time, great!

an alternate compromise

application opens a file, read it a day later, result?

- day-old version of file

modification 1: check server/write to server after an amount of time

doesn't need to be much time to be useful

- word processor: typically load/save file in $<$ second

AFSv2

Andrew File System version 2

uses a stateful server

also works file at a time — not parts of file

- i.e. read/write entire files

but still chooses consistency compromise

- still won't support simulatenous read+write from diff. machines well

stateful: avoids repeated 'is my file okay?' queries

NFS versus AFS reading/writing

NFS reading: read/write block at a time

AFS reading: always read/write *entire file*

exercise: pros/cons?

- efficient use of network?
- what kinds of inconsistency happen?
- does it depend on workload?

AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: write whole file

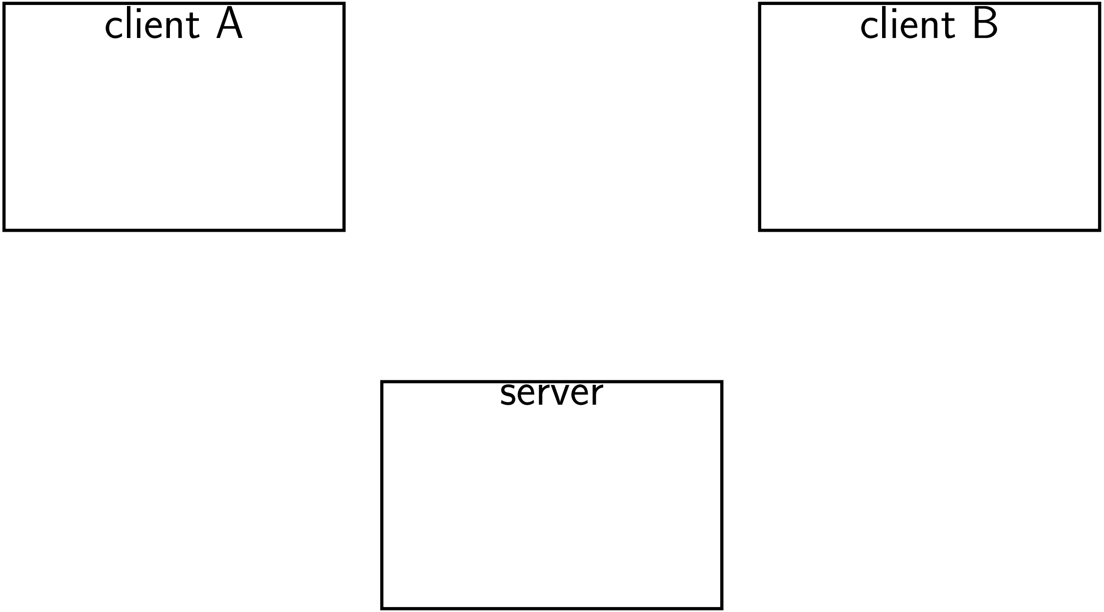
last writer wins

NFS: last writer wins per block

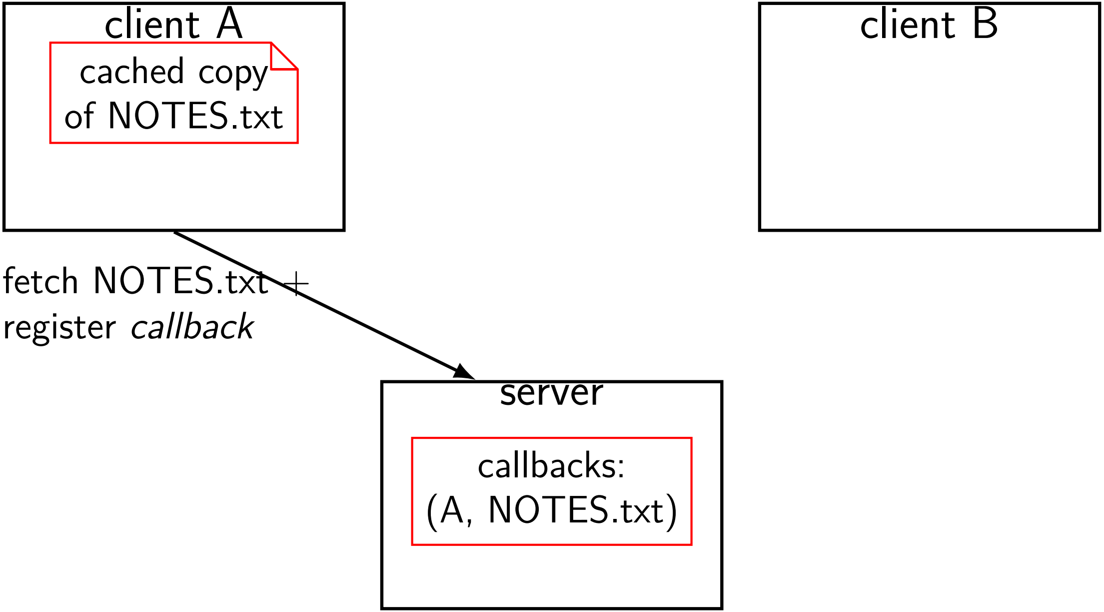
on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
NFS: write NOTES.txt block 0	
	close NOTES.txt
	NFS: write NOTES.txt block 0
	NFS: write NOTES.txt block 1
NFS: write NOTES.txt block 1	
NFS: write NOTES.txt block 2	
	NFS: write NOTES.txt block 2

NOTES.txt: 0 from B, 1 from A, 2 from B

AFS caching



AFS caching



client A

cached copy of NOTES.txt

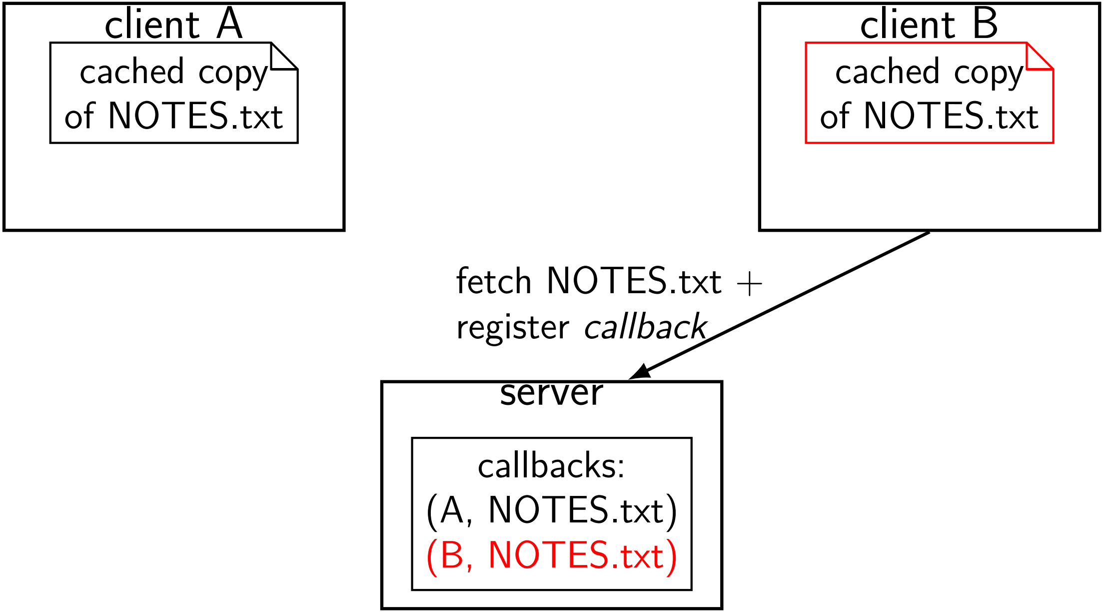
client B

fetch NOTES.txt + register *callback*

server

callbacks:
(A, NOTES.txt)

AFS caching



client A

cached copy of NOTES.txt

client B

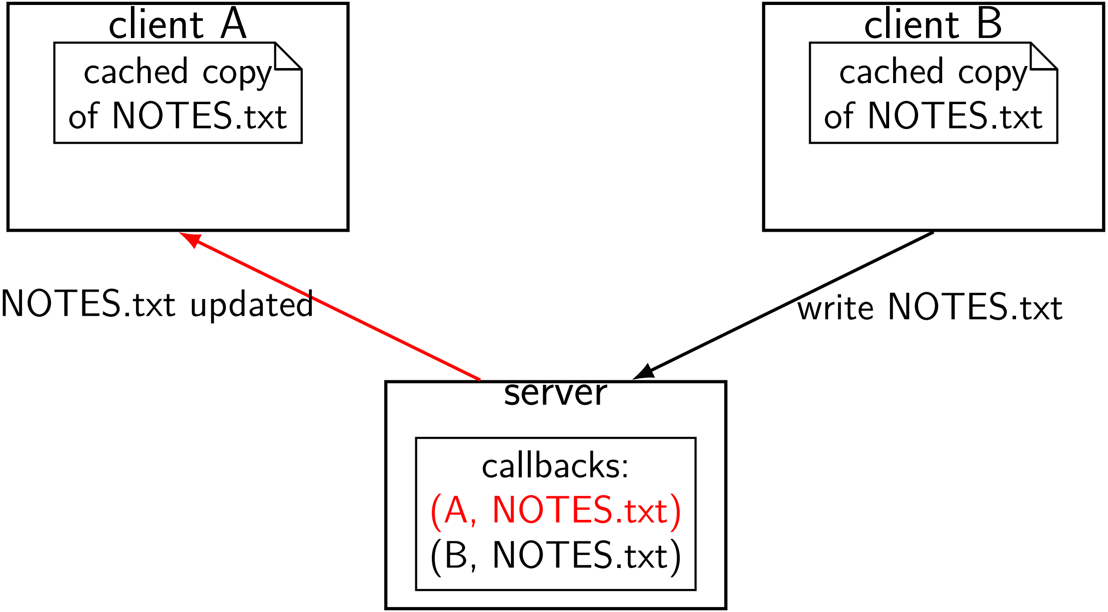
cached copy of NOTES.txt

fetch NOTES.txt + register *callback*

server

callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

AFS caching


client A
cached copy of NOTES.txt
client B
cached copy of NOTES.txt
NOTES.txt updated
write NOTES.txt
server
callbacks:
(A, NOTES.txt)
(B, NOTES.txt)

callback inconsistency (1)

on client A	on client B
open NOTES.txt	
(AFS: NOTES.txt fetched)	
read from cached NOTES.txt	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

callback inconsistency (1)

on client A	on client B
open NOTE	
(AFS: NOTE	
read from ca	
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

problem with close-to-open consistency
same issue w/NFS: B can't know about write
because server doesn't
(could fix by notifying server earlier)

callback inconsistency (1)

on client A	on client B
open NOTE	
(AFS: NOTE	
read from ca	
	open NOTES.txt
	(NOTES.txt fetched)
	read from NOTES.txt
write to cached NOTES.txt	
	read from NOTES.txt
write to cached NOTES.txt	
close NOTES.txt	
(write to server)	
	(AFS: callback: NOTES.txt changed)

close-to-open consistency assumption:
are not accessing file from two places at once

protection/security

protection: mechanisms for controlling access to resources

- page tables, preemptive scheduling, encryption, …

security: *using protection* to prevent misuse

- misuse represented by **policy**
- e.g. “don’t expose sensitive info to bad people”

this class: about mechanisms more than policies

goal: provide enough flexibility for many policies

adversaries

security is about **adversaries**

do the worst possible thing

challenge: adversary can be clever…

authorization v authentication

authentication — who is who

authorization v authentication

authentication — who is who

authorization — who can do what

probably need authentication first...

authentication

password

hardware token

…

authentication

password

hardware token

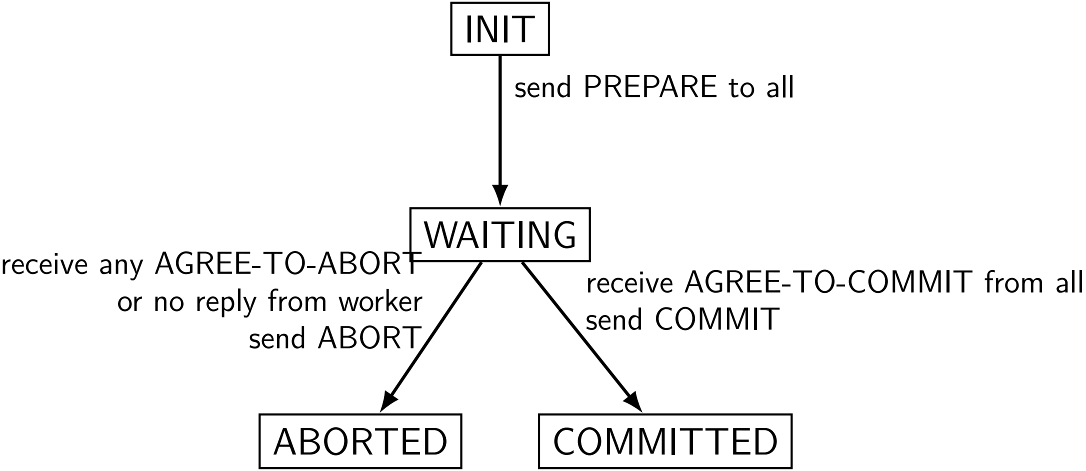
…

this class: mostly won't deal with how

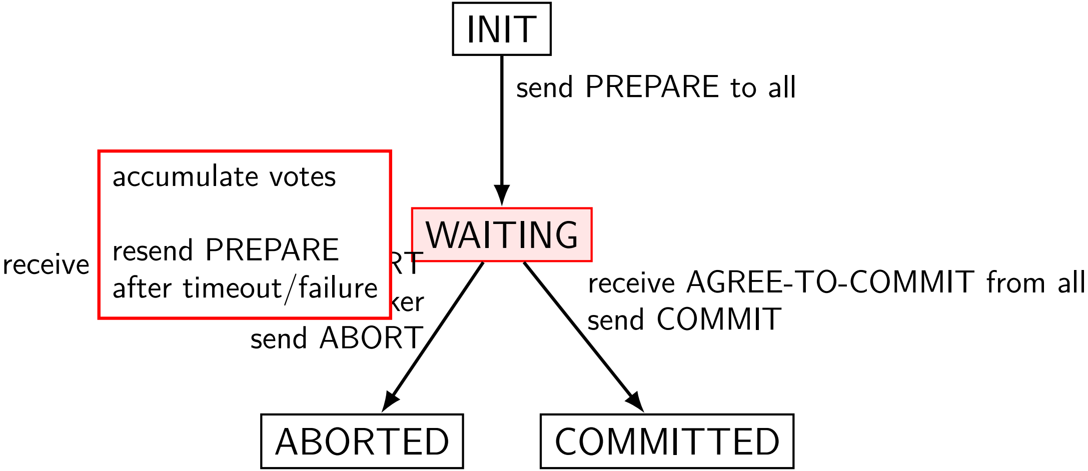
just tracking afterwards

backup slides

coordinator state machine (simplified?)



coordinator state machine (simplified?)



INIT

send PREPARE to all

accumulate votes

resend PREPARE
after timeout/failure

receive

RT

ker

send ABORT

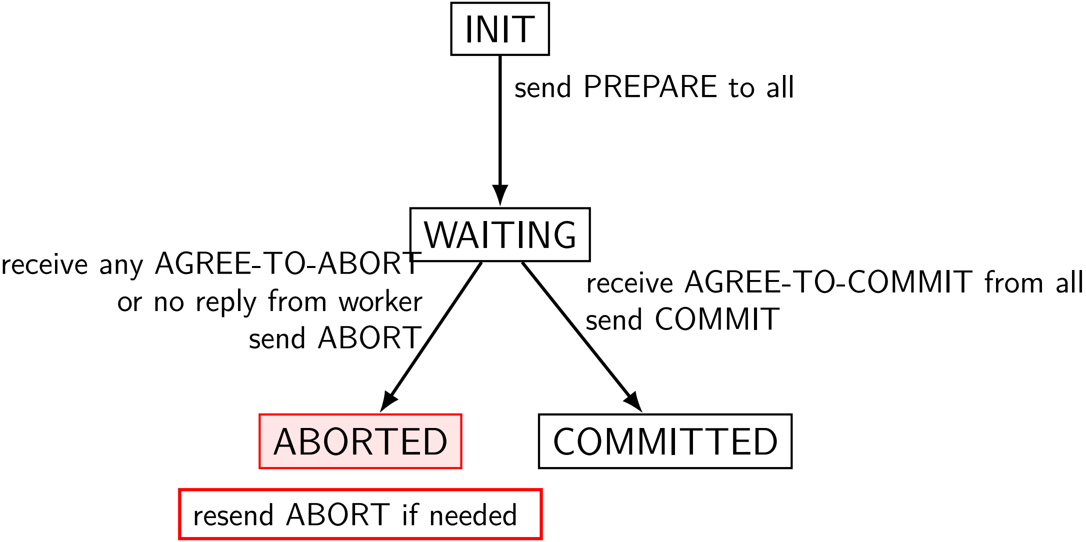
WAITING

receive AGREE-TO-COMMIT from all
send COMMIT

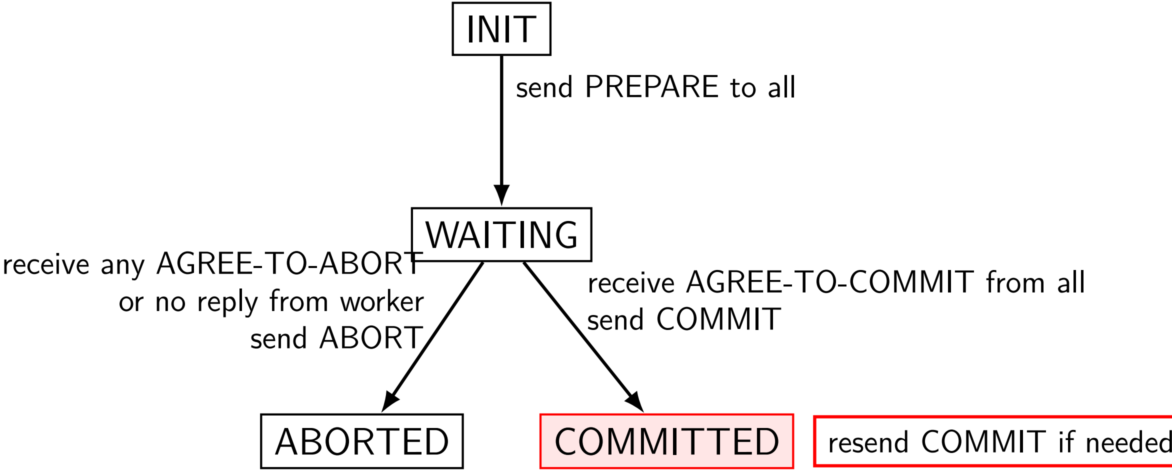
ABORTED

COMMITTED

coordinator state machine (simplified?)



coordinator state machine (simplified?)



INIT

send PREPARE to all

WAITING

receive any AGREE-TO-ABORT
or no reply from worker
send ABORT

receive AGREE-TO-COMMIT from all
send COMMIT

ABORTED

COMMITTED

resend COMMIT if needed

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

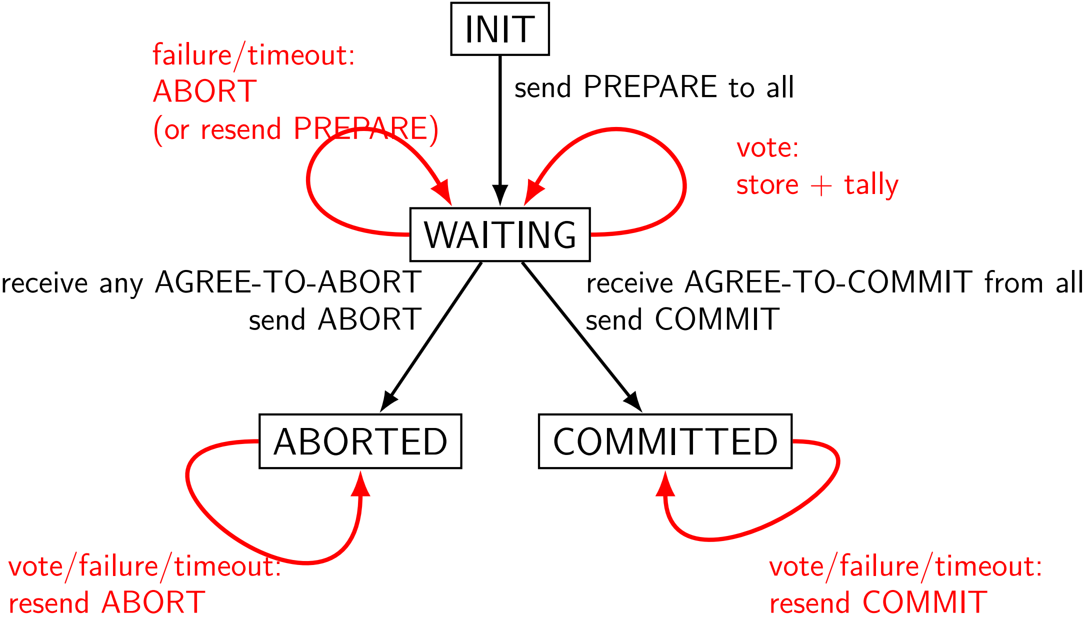
no vote from worker?

- 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?)



INIT

send PREPARE to all

failure/timeout:
ABORT
(or resend PREPARE)

vote:
store + tally

WAITING

receive any AGREE-TO-ABORT
send ABORT

receive AGREE-TO-COMMIT from all
send COMMIT

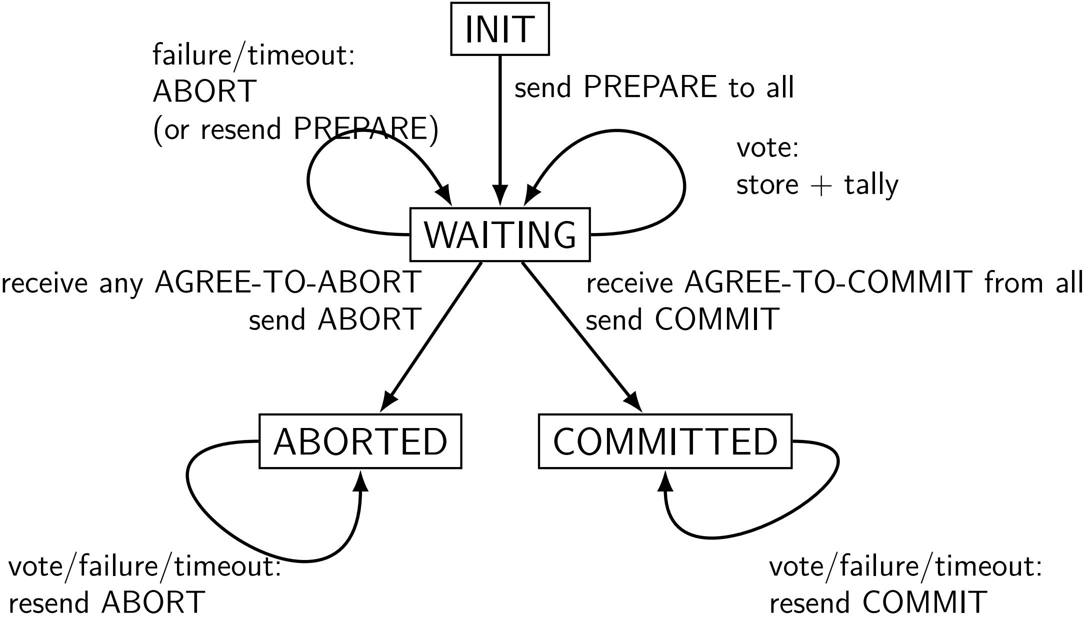
ABORTED

COMMITTED

vote/failure/timeout:
resend ABORT

vote/failure/timeout:
resend COMMIT

coordinator state machine (less simplified?)



INIT

send PREPARE to all

failure/timeout:
ABORT
(or resend PREPARE)

vote:
store + tally

WAITING

receive any AGREE-TO-ABORT
send ABORT

receive AGREE-TO-COMMIT from all
send COMMIT

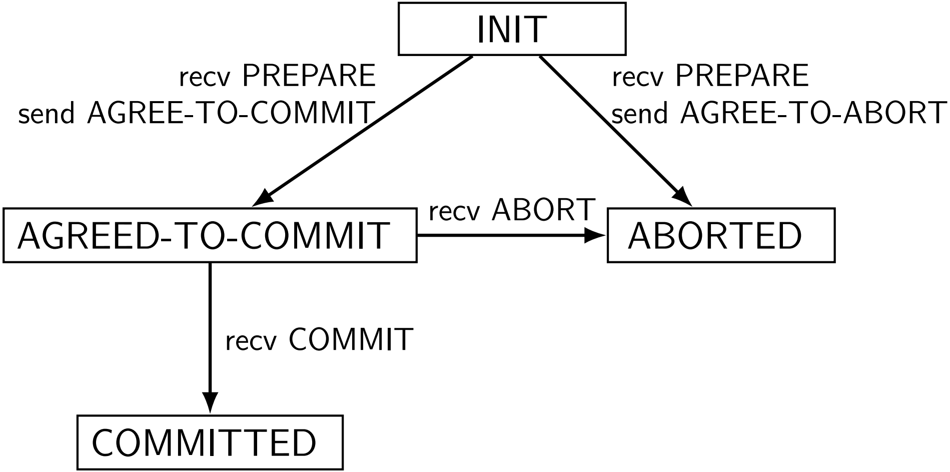
ABORTED

COMMITTED

vote/failure/timeout:
resend ABORT

vote/failure/timeout:
resend COMMIT

worker state machine (simplified)



INIT

recv PREPARE
send AGREE-TO-COMMIT

recv PREPARE
send AGREE-TO-ABORT

AGREED-TO-COMMIT

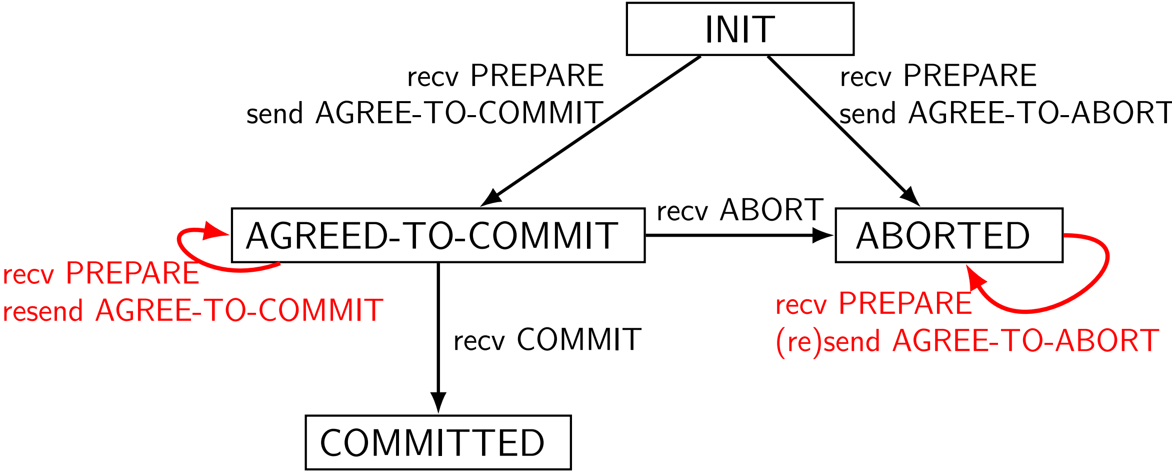
recv ABORT

ABORTED

recv COMMIT

COMMITTED

worker state machine (less simplified?)



INIT

recv PREPARE
send AGREE-TO-COMMIT

recv PREPARE
send AGREE-TO-ABORT

AGREED-TO-COMMIT

recv ABORT

ABORTED

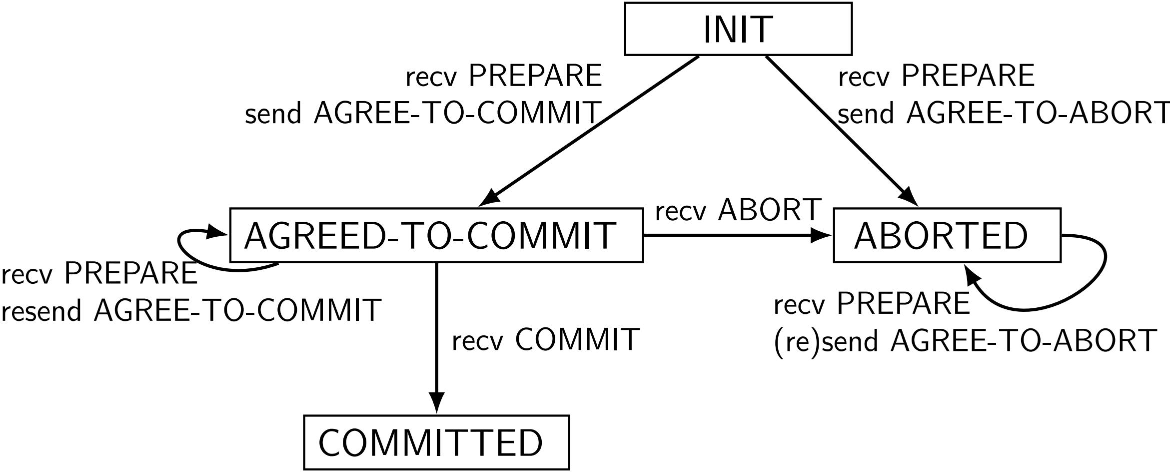
recv PREPARE
resend AGREE-TO-COMMIT

recv PREPARE
(re)send AGREE-TO-ABORT

recv COMMIT

COMMITTED

worker state machine (less simplified?)



INIT

recv PREPARE
send AGREE-TO-COMMIT

recv PREPARE
send AGREE-TO-ABORT

AGREED-TO-COMMIT

recv ABORT

ABORTED

recv PREPARE
resend AGREE-TO-COMMIT

recv COMMIT

recv PREPARE
(re)send AGREE-TO-ABORT

COMMITTED

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?
- …

supporting offline operation

so far: assuming constant contact with server

someone else writes file: we find out

we finish editing file: can tell server right away

good for an office

- my work desktop can almost always talk to server

not so great for mobile cases

- spotty airport/café wifi, no cell reception, …

basic offline operation idea

when offline: work on cached data only

writeback whole file only

problem: more opportunity for overlapping accesses to same file

recall: AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: (over)write whole file

probably losing data!

usually wanted to merge two versions

recall: AFS: last writer wins

on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
AFS: write whole file	
	close NOTES.txt
	AFS: (over)write whole file

probably losing data!

usually wanted to merge two versions

worse problem with delayed writes for disconnected operation

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

- avoid problem of last writer wins

Coda FS: conflict resolution

Coda: distributed FS based on AFSv2 (c. 1987)

supports offline operation with conflict resolution

while offline: clients remember *previous version ID of file*

clients include version ID info with file updates

allows detection of conflicting updates

- avoid problem of last writer wins

and then…ask user? regenerate file? …?

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

- uses version IDs to decide what to update

Coda FS: what to cache

idea: user specifies list of files to keep loaded

when online: client synchronizes with server

- uses version IDs to decide what to update

DropBox, etc. probably similar idea?

version ID?

not a version number?

actually a *version vector*

version number for each machine that modified file

- number for each server, client

allows use of multiple servers

- if servers get desync'd, use version vector to detect
- then do, uh, something to fix any conflicting writes

file locking

so, your program doesn't like conflicting writes

what can you do?

if offline operation, probably not much…

otherwise file locking

except it often doesn't work on NFS, etc.

advisory file locking with fcntl

```
int fd = open(...);
struct flock lock_info = {
    .l_type = F_WRLCK,  // write lock; RDLOCK also available
    // range of bytes to lock:
    .l_whence = SEEK_SET, l_start = 0, l_len = ...
};
/* set lock, waiting if needed */
int rv = fcntl(fd, F_SETLKW, &lock_info);
if (rv == −1) { /* handle error */ }
/* now have a lock on the file */

/* unlock --- could also close() */
lock_info.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &lock_info);
```

advisory locks

fcntl is an *advisory* lock

doesn't stop others from accessing the file…

unless they always try to get a lock first

POSIX file locks are horrible

actually two locking APIs: fcntl() and flock()

fcntl: *not* inherited by fork

fcntl: closing any fd for file release lock

    even if you dup2'd it!

fcntl: maybe sometimes works over NFS?

flock: less likely to work over NFS, etc.

fcntl and NFS

seems to require extra state at the server

typical implementation: separate *lock server*

not a stateless protocol

lockfiles

use a separate *lockfile* instead of “real” locks

- e.g. convention: use `NOTES.txt.lock` as lock file

lock: create a *lockfile* with `link()` or `open()` with `O_EXCL`

- can’t lock: link()/open() will fail “file already exists”
- for current NFSv3: should be single RPC calls that always contact server
- some (old, I hope?) systems: link() atomic, open() O_EXCL not

unlock: remove the lockfile

- annoyance: what if program crashes, file not removed?