Distributed 3: Network FS (finish) / Failure

Changelog

Changes made in this version not seen in first lecture:

16 April 2019: move and relocate Coda/disconnected operation slides to better explain connection to last-writer-wins being a problem

last time

transparency

remote procedure calls
- interface description languages
- generic among architectures/languages?

network filesystems via RPCs

stateless servers
- server remembers nothing about client
- server doesn't care if client crashes
- trick: client stores opaque IDs/cookies/etc. for server

NFSv2: stateless servers for filesystem
- file IDs (based on inode number) tracked by clients

things NFSv2 didn't do well

performance — each read goes to server?

- would like to cache things in the clients

performance — each write goes to server?

- observation: usually only one user of file at a time
- would like to usually cache writes at clients
- writeback later

offline operation?

- would be nice to work on laptops where wifi sometimes goes out

statefulness

stateful protocol (example: FTP)

- 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:

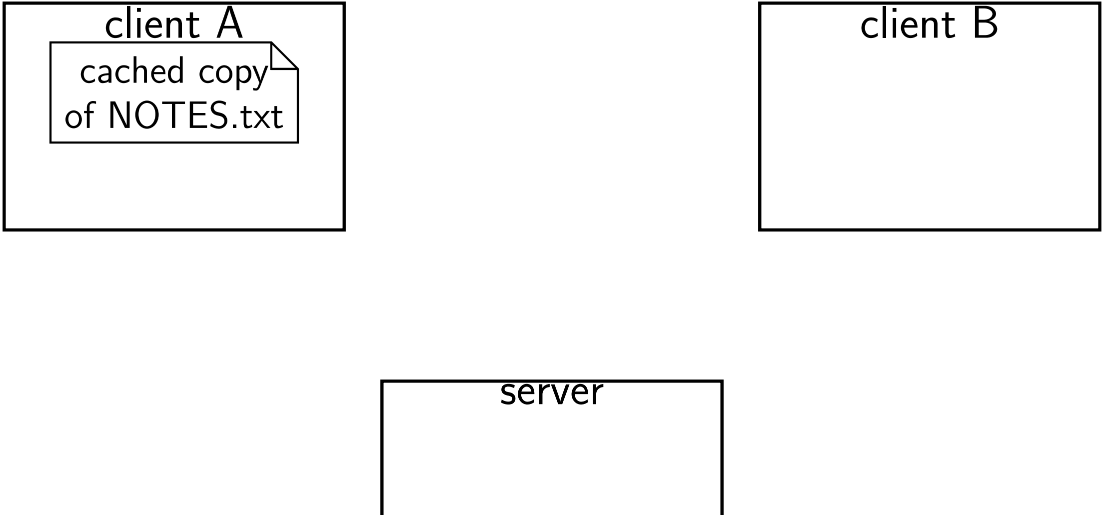
stateless: more work for client, less for server

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

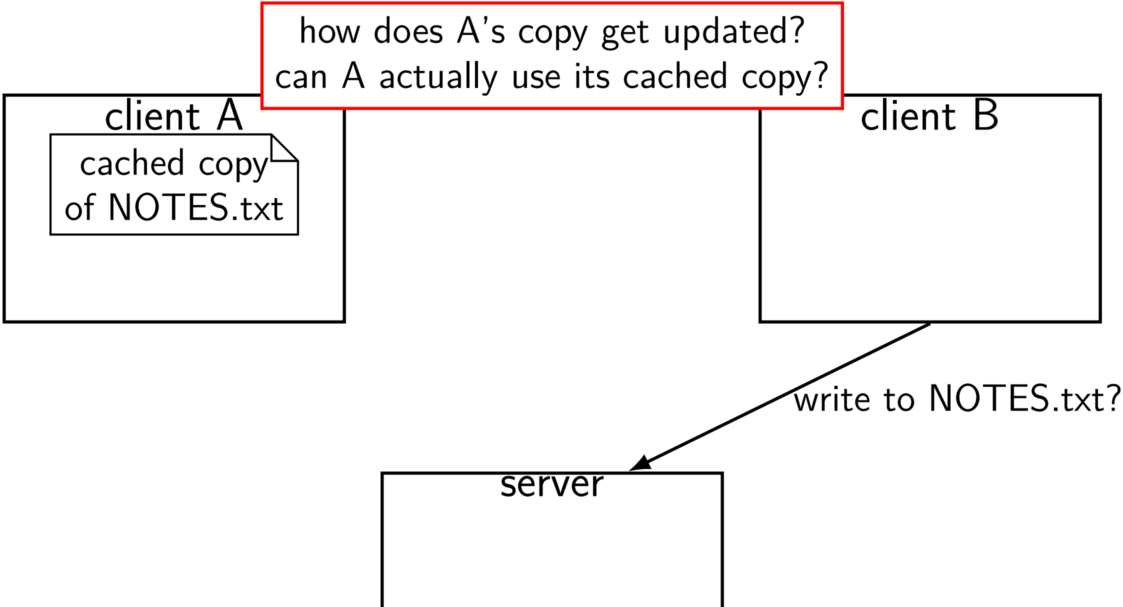
stateful: more work for server, less for client

- 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

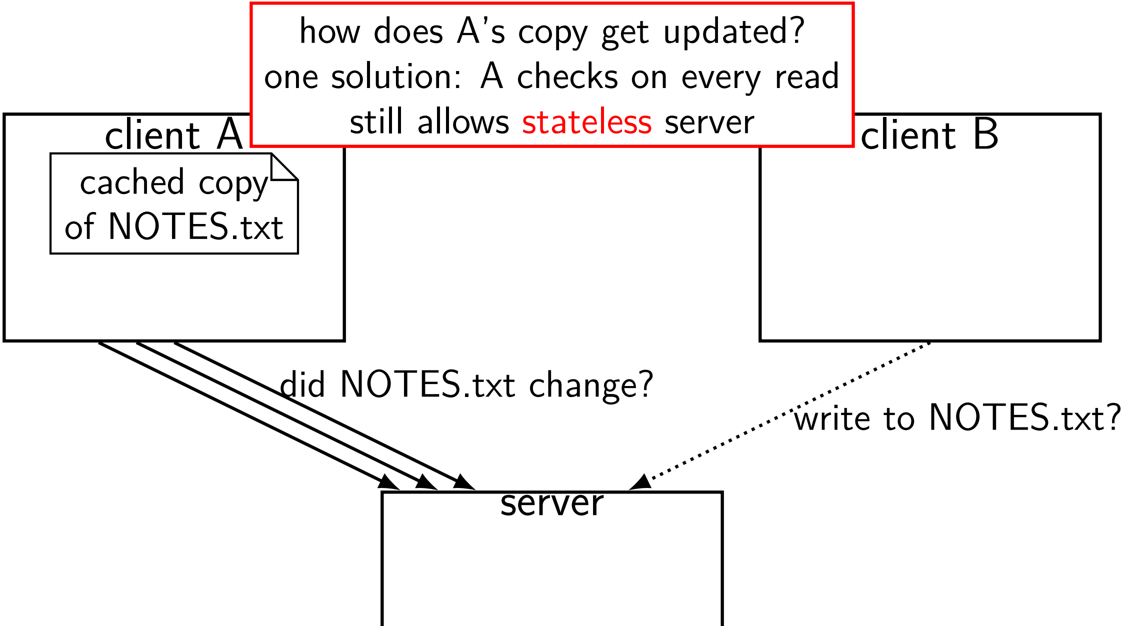
updating cached copies?



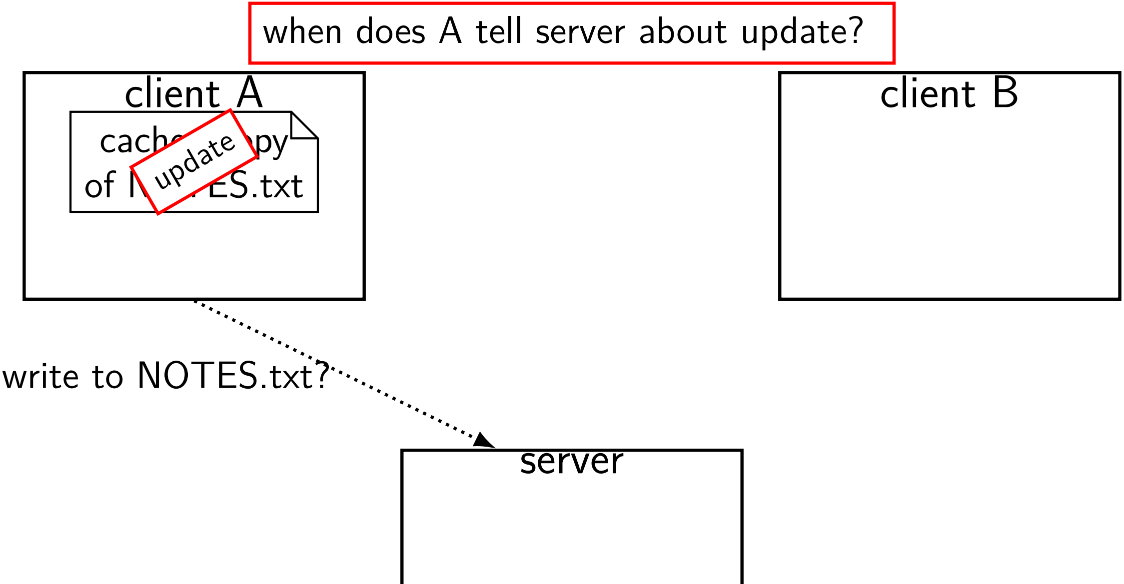
updating cached copies?



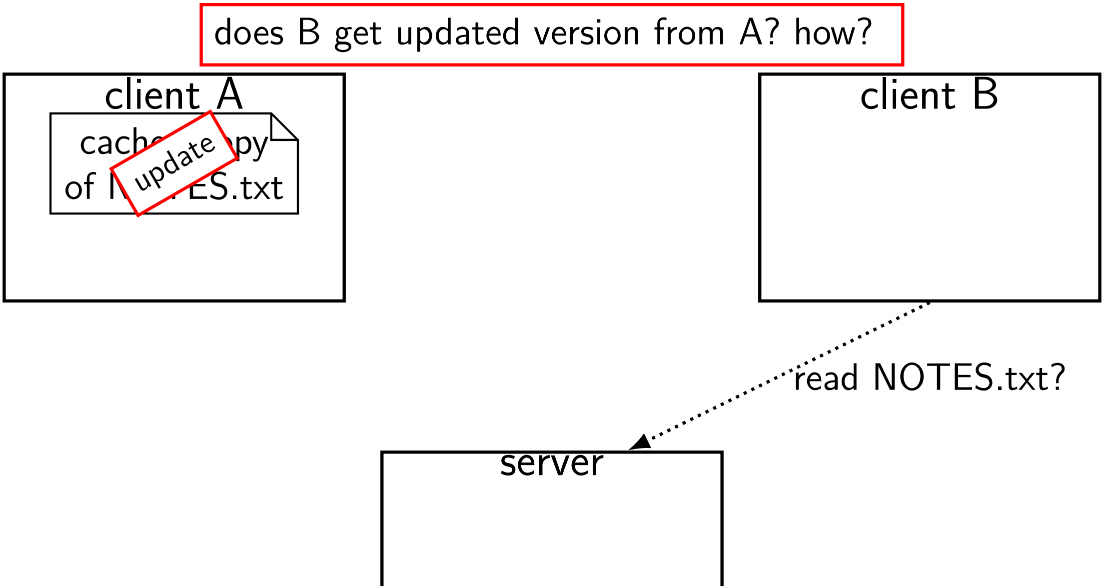
updating cached copies?



updating cached copies?



updating cached copies?



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

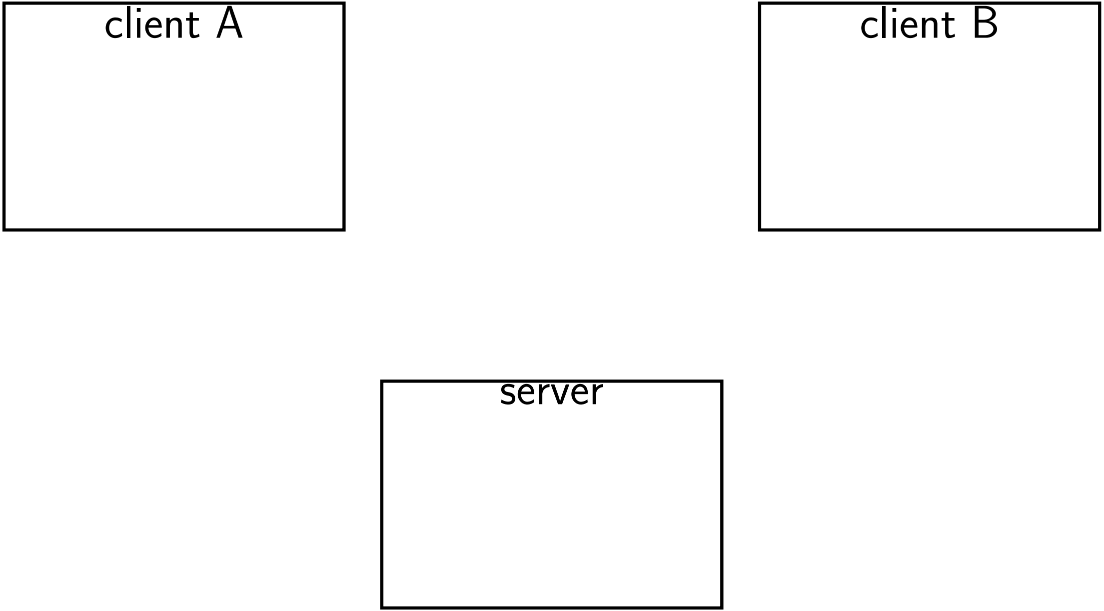
last writer wins

NFS: last writer wins per block

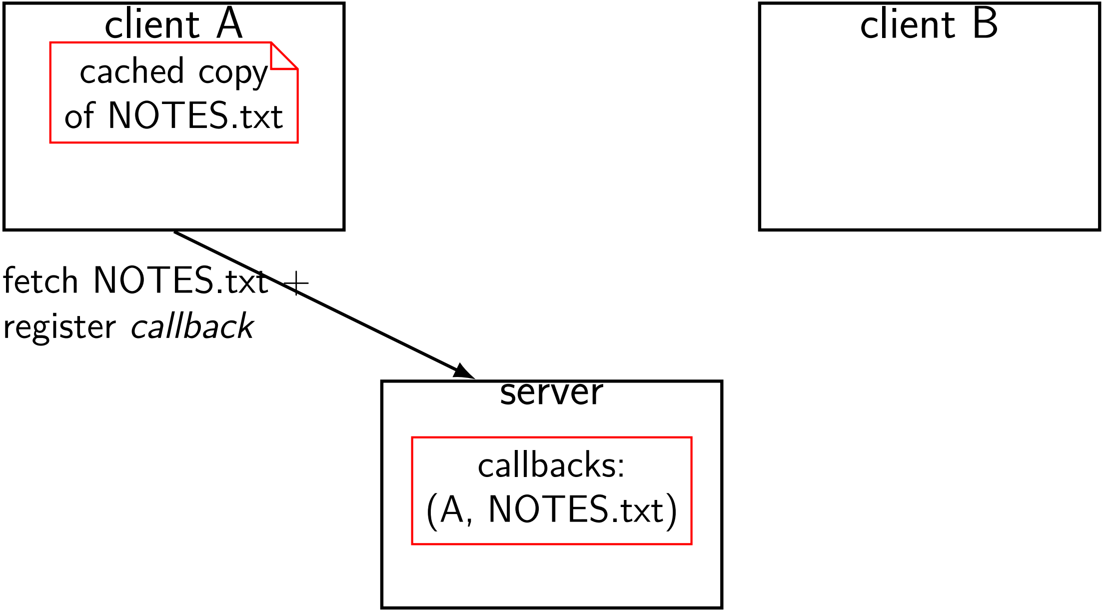
on client A	on client B
open NOTES.txt	
	open NOTES.txt
write to cached NOTES.txt	
	write to cached NOTES.txt
close NOTES.txt	
NFS: write NOTES.txt block 0	
	close NOTES.txt
	NFS: write NOTES.txt block 0
	NFS: write NOTES.txt block 1
NFS: write NOTES.txt block 1	
NFS: write NOTES.txt block 2	
	NFS: write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

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

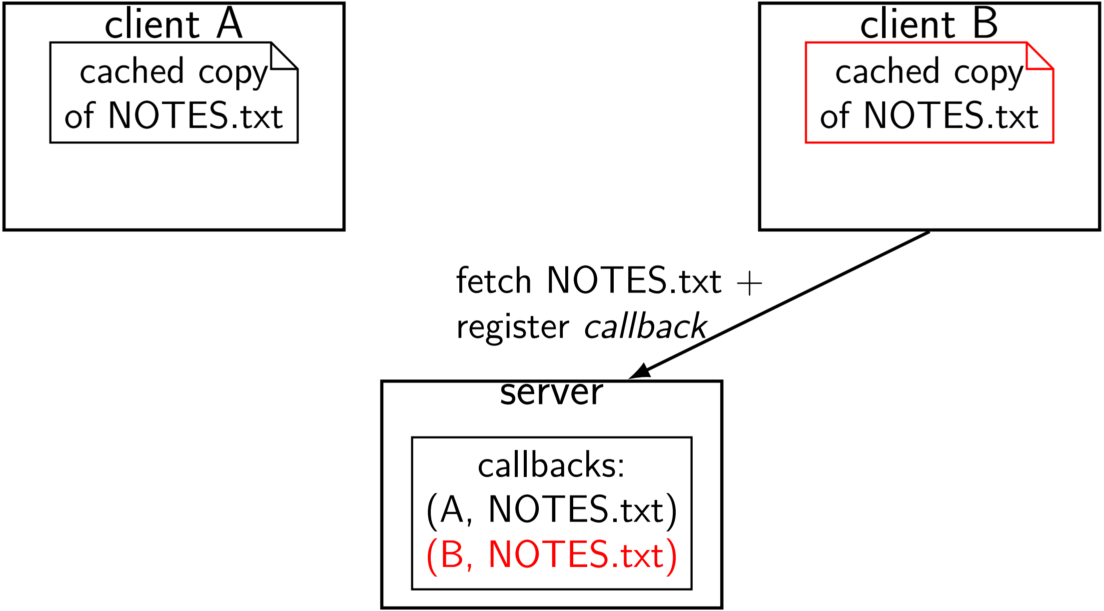
AFS caching



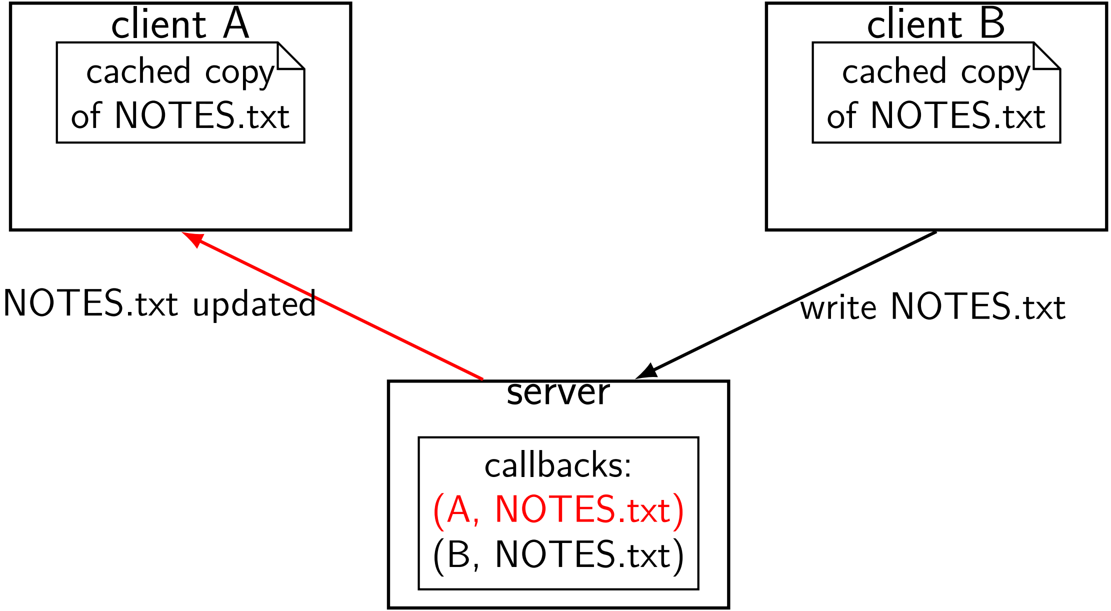
AFS caching


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	
	(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)

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)

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

on client A	on client B
open NOTE	
(AFS: NOTE	
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)

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

on connections and how they fail

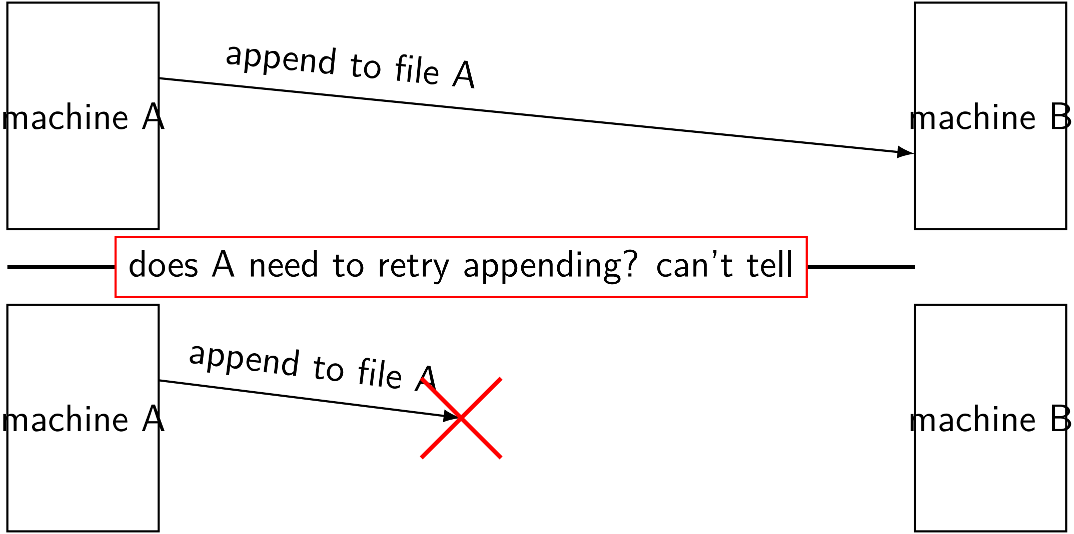
for the most part: don't look at details of connection implementation

...but will do so to explain how things fail

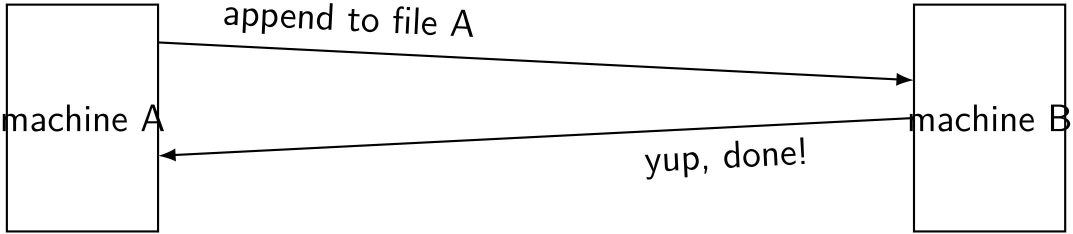
why? important for designing protocols that change things

how do I know if any action took place?

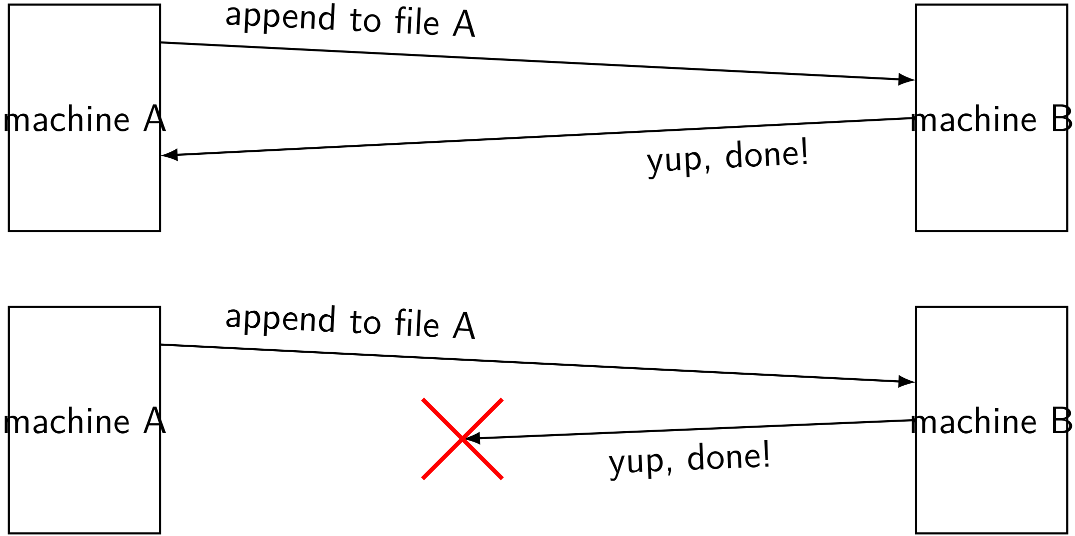
dealing with network failures



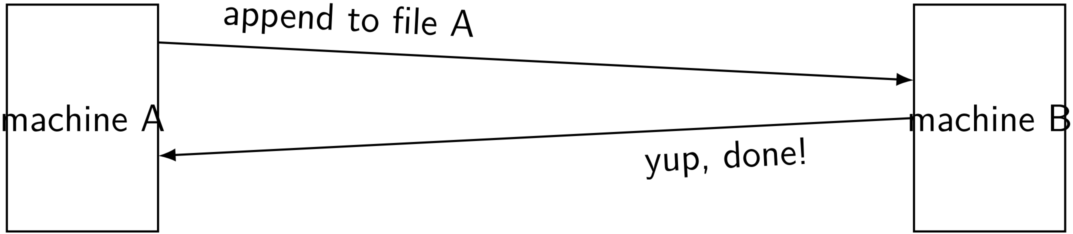
handling failures: try 1



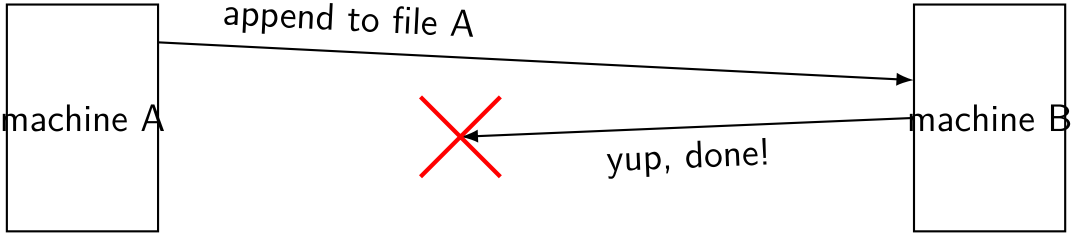
handling failures: try 1



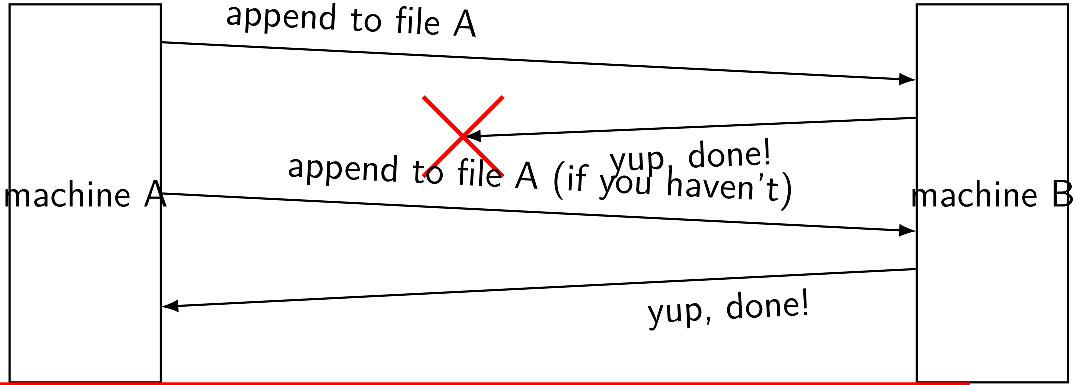
handling failures: try 1



does A need to retry appending? *still* can't tell



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?

dealing with failures

real connections: acknowledgements + retrying

but have to give up eventually

means on failure — can't always know what happened remotely!

- maybe remote end received data
- maybe it didn't
- maybe it crashed
- maybe it's running, but it's network connection is down
- maybe our network connection is down

also, connection knows *whether program received data*

- not whether program did whatever commands it contained

failure models

how do machines fail?...

well, lots of ways

two models of machine failure

fail-stop

failing machines stop responding

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

- requires 1 extra machine for fail-stop
- requires $3F + 1$ 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

- requires 1 extra machine for fail-stop
- requires $3F+1$ 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

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

...even if one machine fails

no confusion about student is in course

"consistency"

the centralized solution

one solution: a new machine D decides what to do

for machines A-C which store records

machine D maintains a redo log for all machines

treats them as just data storage

the centralized solution

one solution: a new machine D decides what to do
- for machines A-C which store records

machine D maintains a redo log for all machines

treats them as just data storage

problem: we'd like machines to work indepdently
- not really taking advantage of distributed system
- why did we split student records across two machines anyways?

decentralized solution sketch

want each machine to be responsible just for their own data

decentralized solution sketch

want each machine to be responsible just for their own data

only coordinate when transaction crosses machine

e.g. changing course + student records

only coordinate with involved machines

decentralized solution sketch

want each machine to be responsible just for their own data

only coordinate when transaction crosses machine

- e.g. changing course + student records

only coordinate with involved machines

hopefully, scales to tens or hundreds of machines

- typical transaction would involve 1 to 3 machines?

distributed transactions and failures

extra tool: persistent log

idea: machine remembers what happen on failure

same idea as redo log: record what to do in log

- preview: whether trying to do/not do action

…but need to handle if machine stopped while writing log

two-phase commit: setup

every machine *votes* on transaction

commit — do the operation (add student A to class)

abort — don't do it (something went wrong)

require unanimity to commit

default=abort

two-phase commit: phases

phase 1: *preparing*

each machine states their intention: agree to commit/abort

phase 2: *finishing*

gather intentions, figure out whether to do/not do it

single global decision

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

nev

to keep promise: can't allow interfering operations
e.g. agree to add student to class → reserve seat in class
(even though student might not be added b/c of other machines)

finishing

learn all machines agree to commit → commit transaction

- actually apply transaction (e.g. record student is in class)
- record decision in local log

learn any machine agreed to abort → abort transaction

- don't ever try to apply transaction
- record decision in local log

finishing

learn all machines agree to commit → commit transaction

- actually apply transaction (e.g. record student is in class)
- record decision in local log

learn any machine agreed to abort → abort transaction

- don't ever try to apply transaction
- record decision in local log

unsure which? just ask everyone what they agreed to do

- they can't change their mind once they tell you

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?

machine goes away, two-phase commit state is uncertain

never resolve what happens

solution in practice: manual intervention

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: VOTE-COMMIT or VOTE-ABORT

- I agree to commit/abort transaction
- worker records decision in log, returns same result each time

coordinator → worker: GLOBAL-COMMMIT or GLOBAL-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

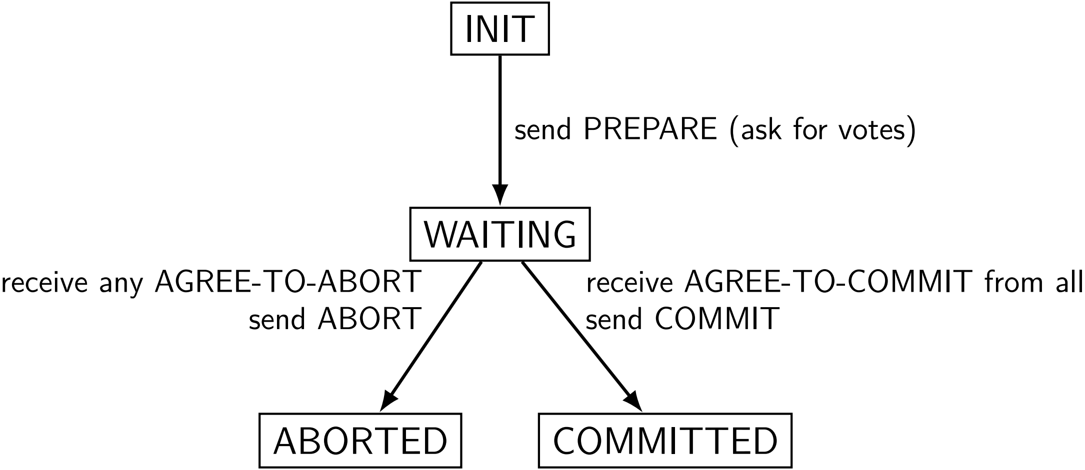
typical tool: state machine

each machine is in some state

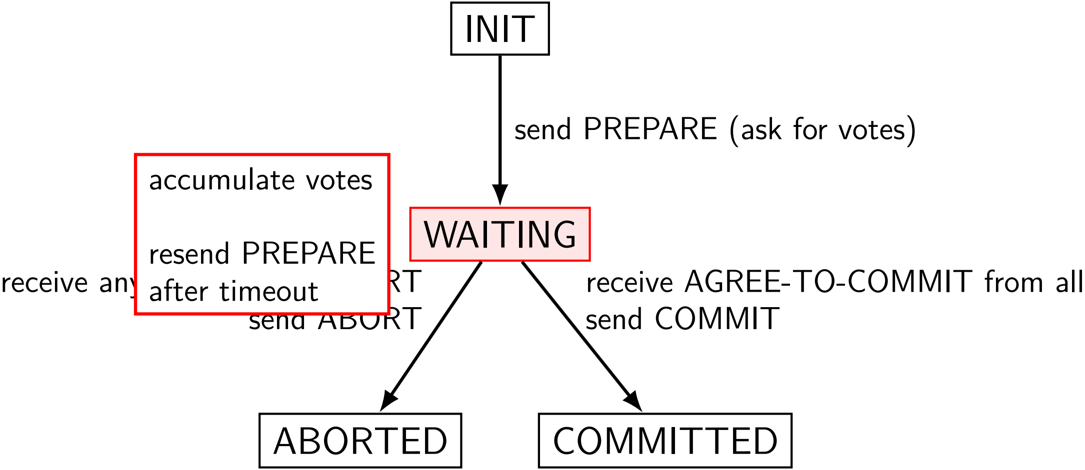
know what every message does in this state

avoids common problem: don't know what message does

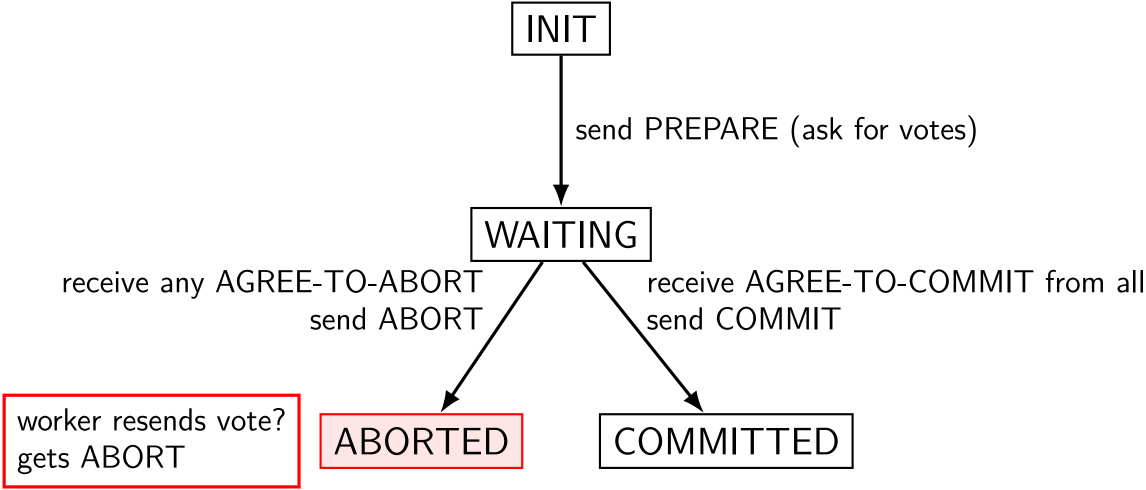
coordinator state machine (simplified)



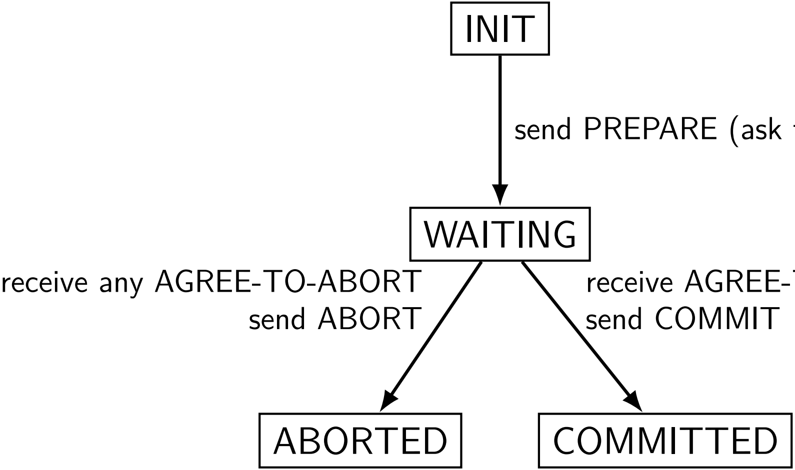
coordinator state machine (simplified)



coordinator state machine (simplified)



coordinator state machine (simplified)



workers resends vote?
gets COMMIT

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: send ABORT to all (dups okay!)
- if COMMITTED: resend COMMIT to all (dups okay!)

message doesn't make it to worker?

- coordinator can resend PREPARE after timeout (or just ABORT)
- worker can resend vote to coordinator to get extra reply

coordinator failure recovery

duplicate messages okay — unique transaction ID!

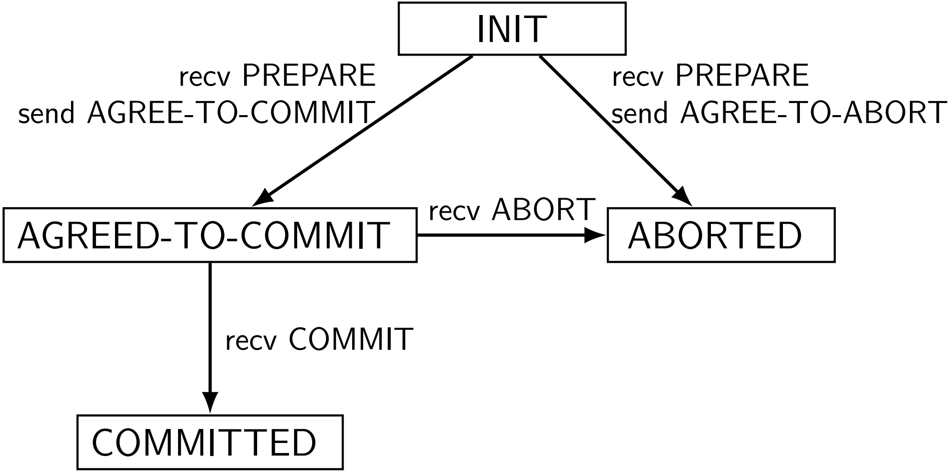
coordinator crashes? *log* indicating last state

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

message doesn't make it to worker?

- coordinator can resend PREPARE after timeout (or just ABORT)
- worker can resend vote to coordinator to get extra reply

worker state machine (simplified)



worker failure recovery

duplicate messages okay — unqiue transaction ID!

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

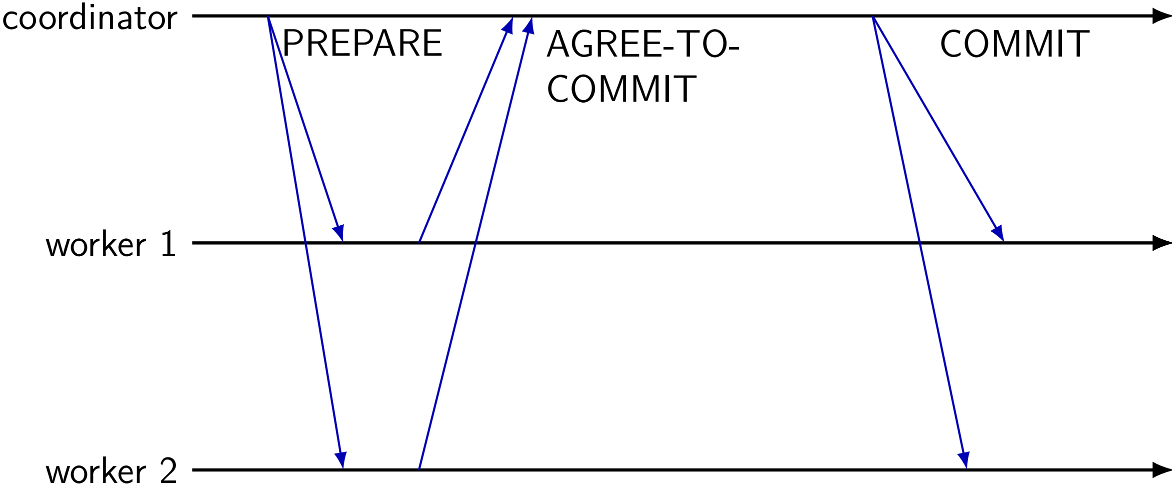
state machine missing details

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

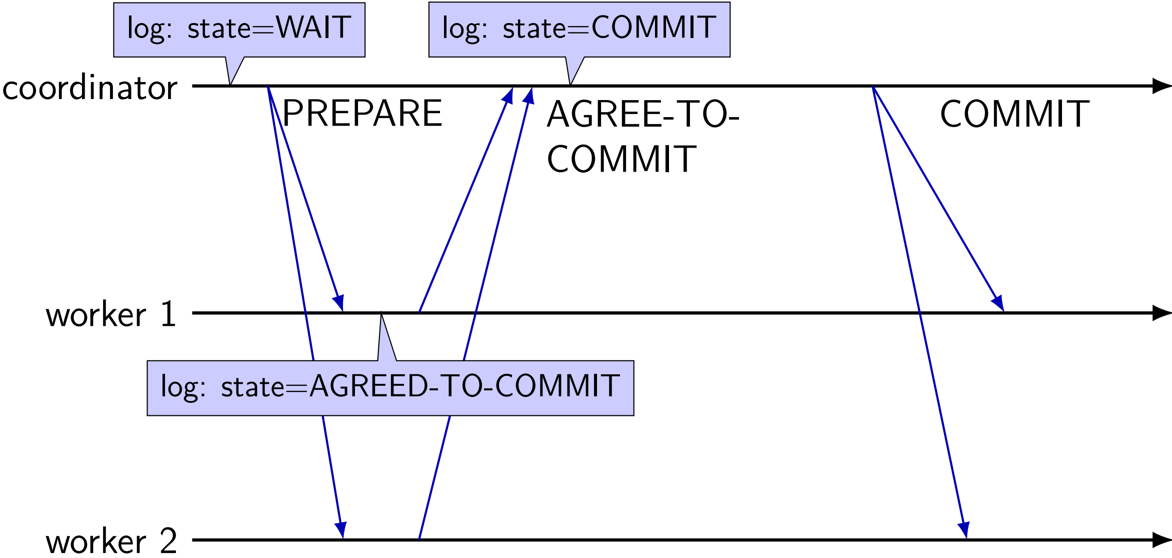
allows verifying properties of state machine

- what happens if machine fails at each possible time?
- what happens if possible message is lost?
- …

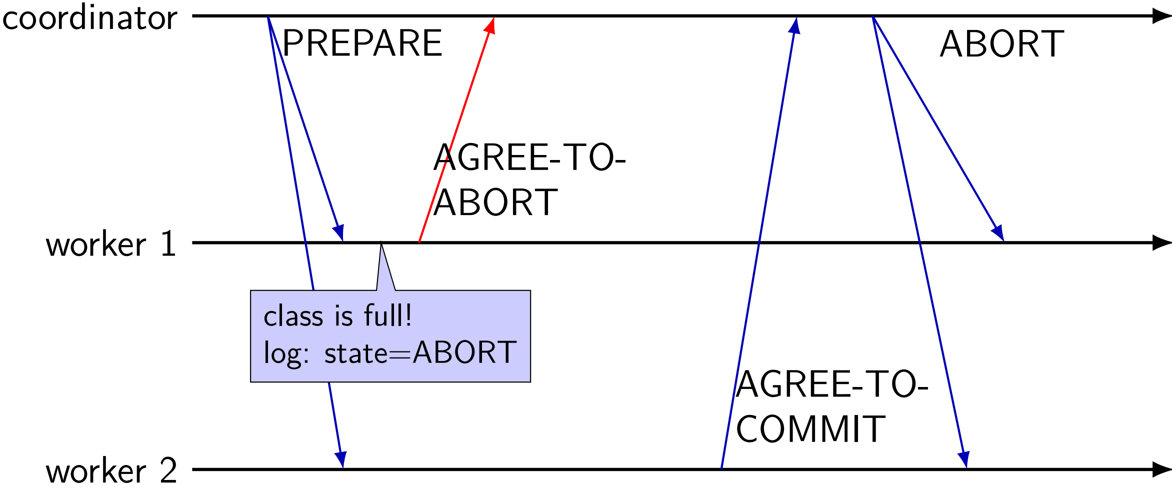
TPC: normal operation



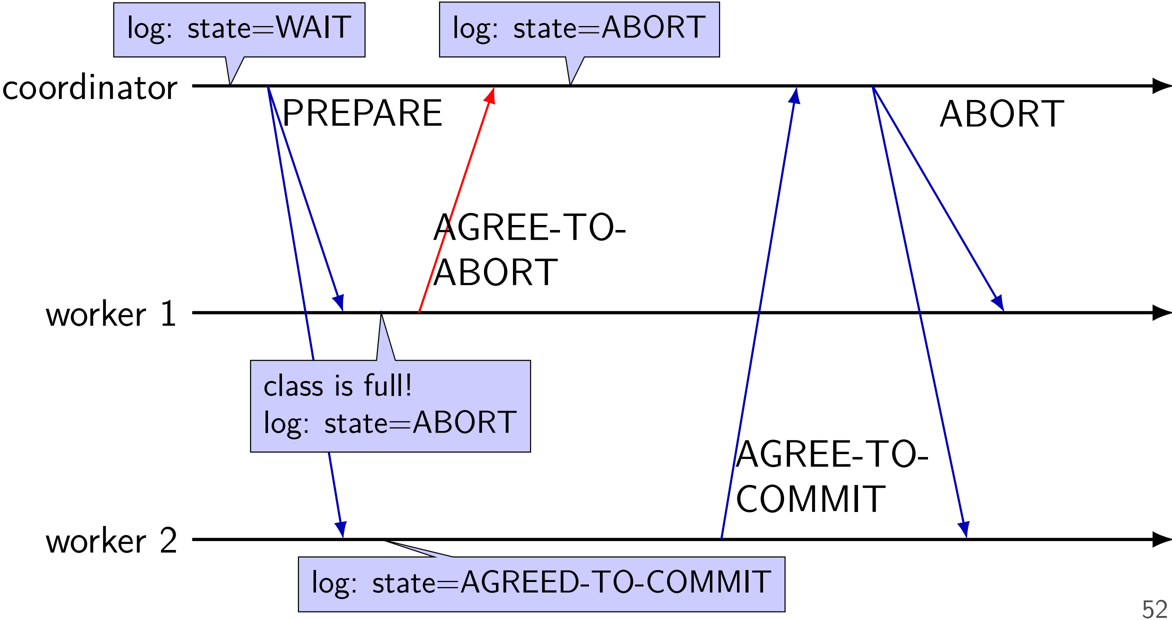
TPC: normal operation



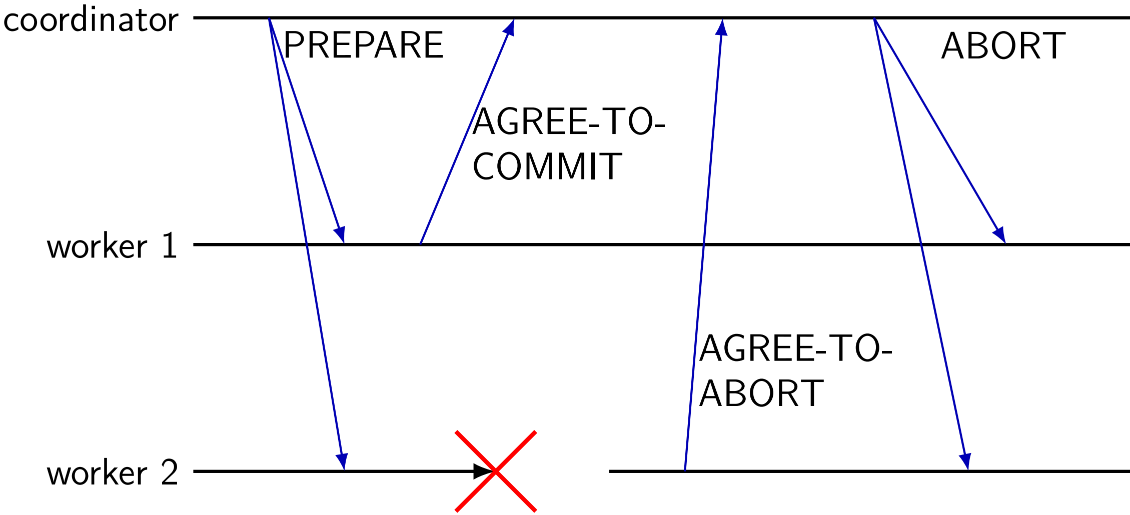
TPC: normal operation — conflict



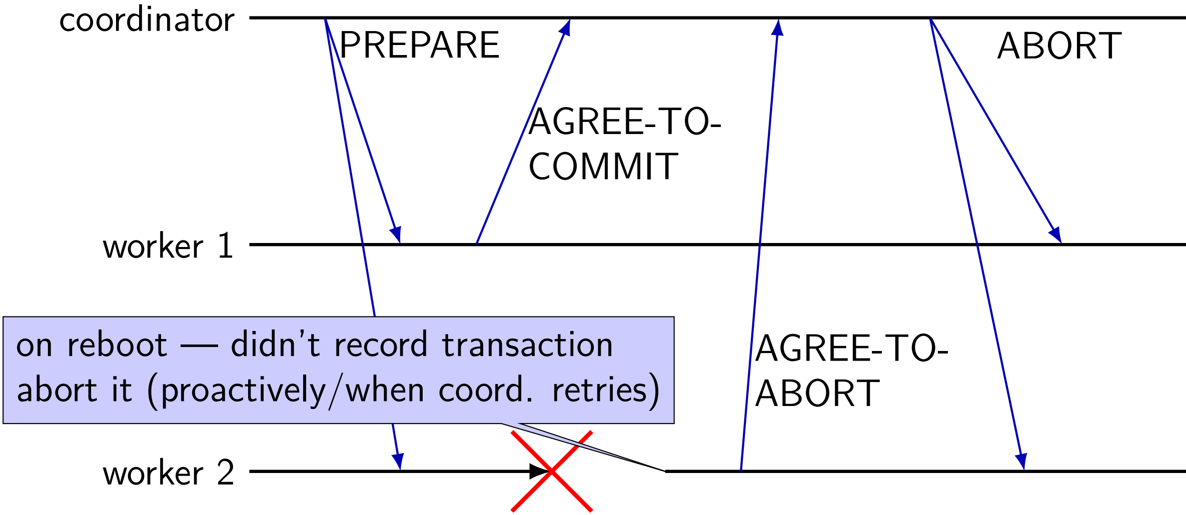
TPC: normal operation — conflict



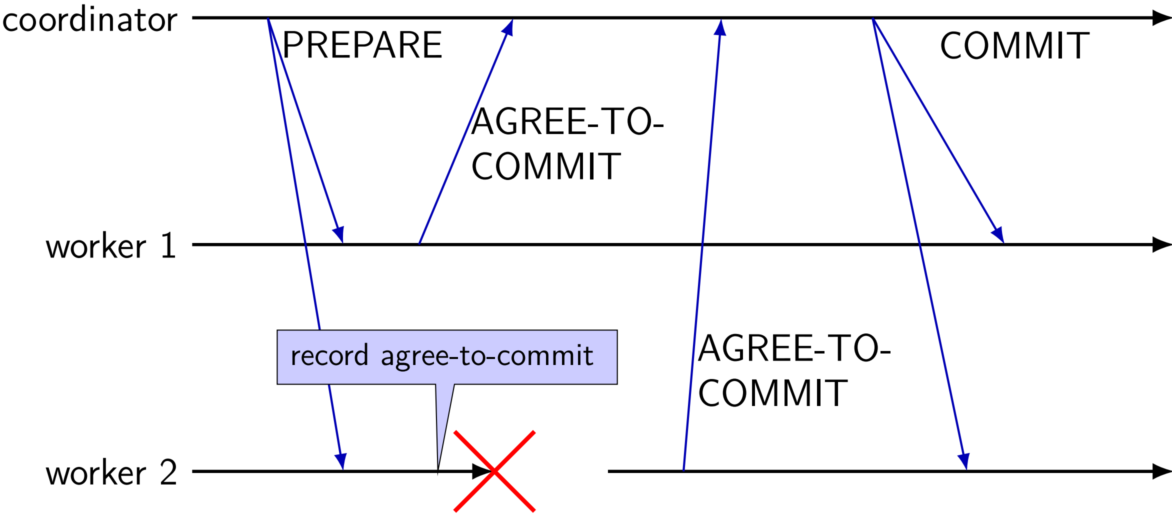
TPC: worker failure (1)



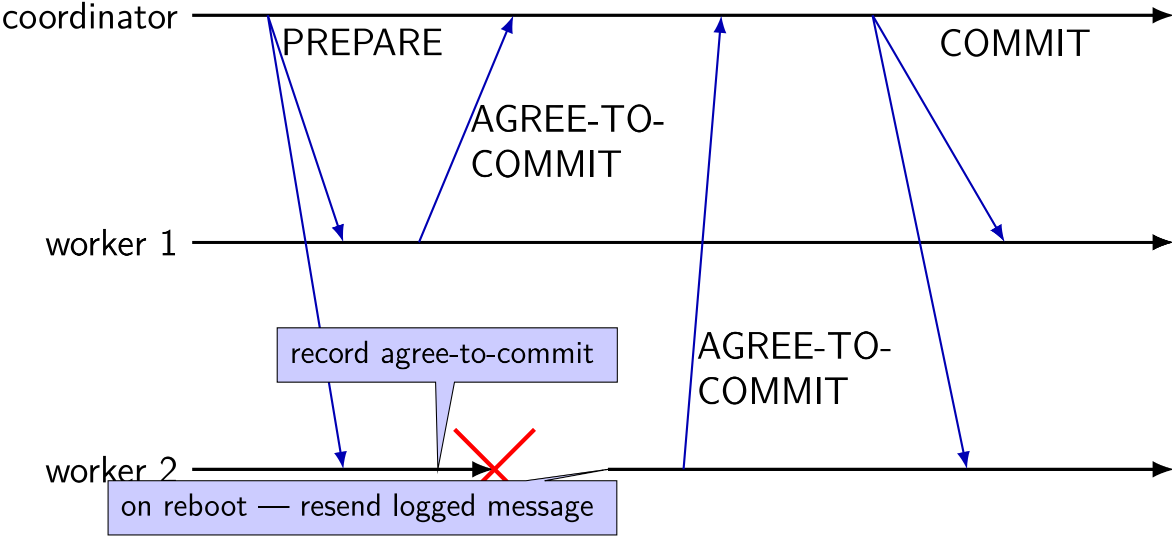
TPC: worker failure (1)



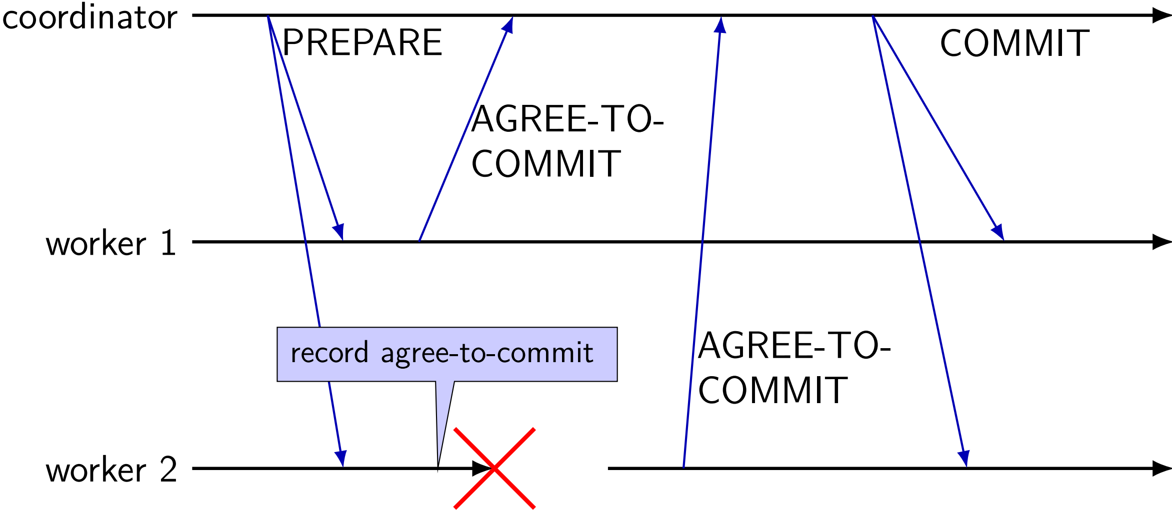
TPC: worker failure (2)



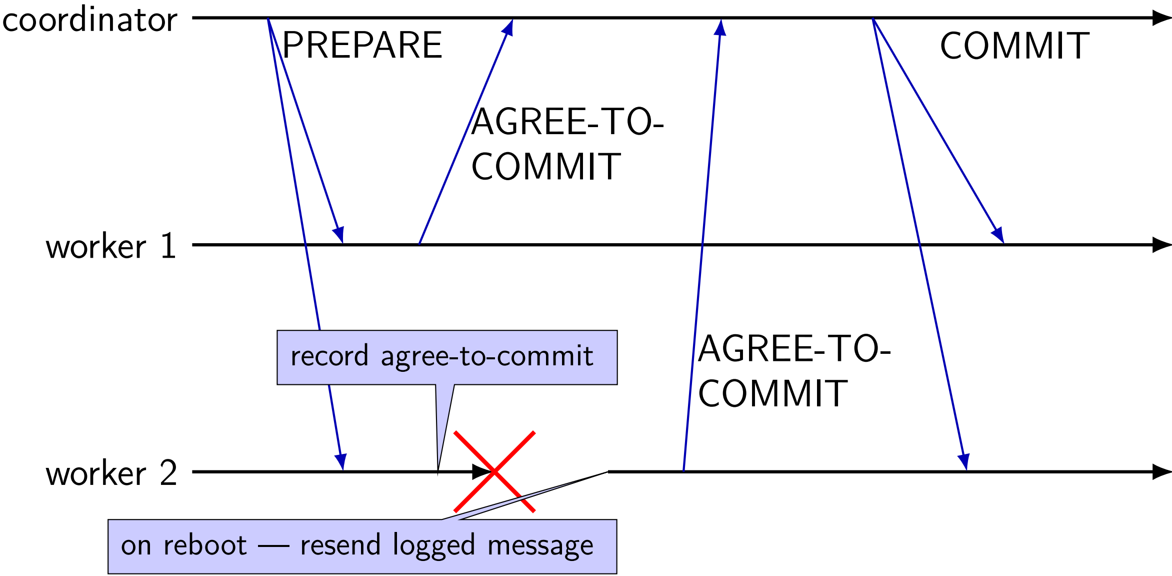
TPC: worker failure (2)



TPC: worker failure (3)



TPC: worker failure (3)



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

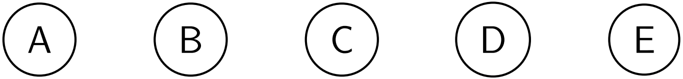
goal: work despite a few failing nodes

just require "enough" nodes to be working

for now — assume fail-stop

nodes don't respond or tell you if broken

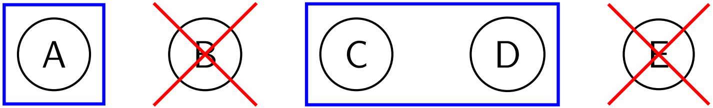
quorums (1)



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

any quorum enough — okay if some nodes fail

quorums (1)



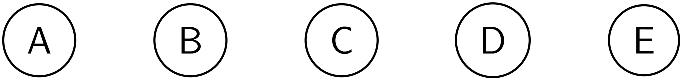
perform read/write with vote of any *quorum* of nodes

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

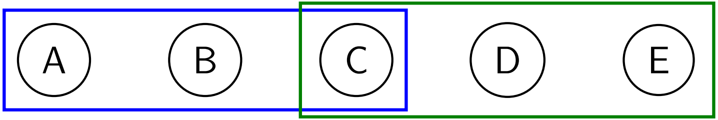
e.g. every operation requires majority of nodes

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

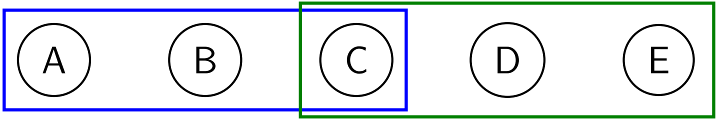
e.g. every operation requires majority of nodes

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

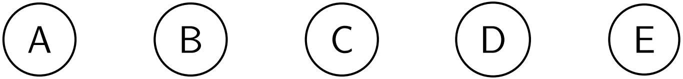
e.g. every operation requires majority of nodes

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)



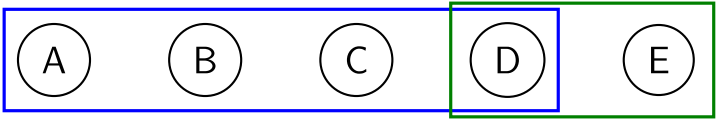
sometimes vary quorum based on operation type

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)



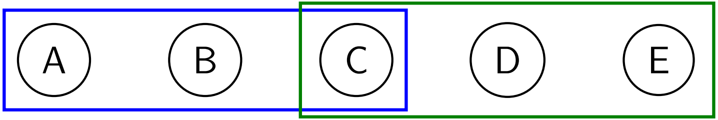
sometimes vary quorum based on operation type

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

backup slides

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) $\rightarrow$ file ID

GETATTR(file ID) $\rightarrow$ (file size, owner, ...)

READ(file ID, offset, length) $\rightarrow$ data

WRITE(file ID, data, offset) $\rightarrow$ success/failure

CREATE(dir file ID, filename, metadata) $\rightarrow$ file ID

REMOVE(dir file ID, filename) $\rightarrow$ success/failure

SETATTR(file ID, size, owner, ...) $\rightarrow$ success/failure

NFSv2 RPC calls (subset)

LOOKUP(dir file ID, filename) $\rightarrow$ file ID

GETATTR(file ID) $\rightarrow$ (file size, owner, ...)

READ(file ID, offset, length) $\rightarrow$ data

WRITE(file ID, data, offset) $\rightarrow$ success/failure

CREATE(dir file ID, filename, metadata) $\rightarrow$ file ID

REMOVE(dir file ID, filename) $\rightarrow$ success/failure

S

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

NFSv2 client versus server

clients: file descriptor $\rightarrow$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) $\rightarrow$ file ID

GETATTR(file ID) $\rightarrow$ (file size, owner, ...)

READ(file ID, offset, length) $\rightarrow$ data

WRITE(file ID, data, offset) $\rightarrow$ success/failure

CREATE(dir file ID, filename, metadata) $\rightarrow$ file ID

REMOVE(dir file ID, filename) $\rightarrow$ success/failure

SETATTR(file

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

NFSv2 RPC (more operations)

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

NFSv2 RPC (more operations)

READDIR(dir file ID, count, optional offset “cookie”) $\rightarrow$
(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

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?