two-phase commit / security (start)

Changelog

Changes made in this version not seen in first lecture:

- quorum: add note that part of voting is updating other nodes to latest version

last time (1)

RPC: remote function calls like local

- interface description language compiled into stubs (wrapper functions)
- marshalling (AKA serialization) of arguments/return value into bytes

NFS: file operations into remote procedure calls

NFS is stateless operation

- server uses file IDs — give inode number
- client remembers fd to file ID mapping
- nothing to recover on server failure
- nothing for server to forget on client failure

last time (2)

close-to-open consistency

- check for updates on open, write file on close
- idea: inconsistent behavior if two processes open file at once okay

AFS: callbacks on write rather than proactive checks

- ...but server still needs to know about write to callback

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?

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 machiens 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

distributed transaction problem

distributed transaction

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

even if *a machine fails*

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

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
- why did we split student records across two machines anyways?

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

otherwise, default=abort

two-phase commit: phases

phase 1: *preparing*

each machine states their intention: commit/abort

phase 2: *finishing*

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

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 e

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)

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

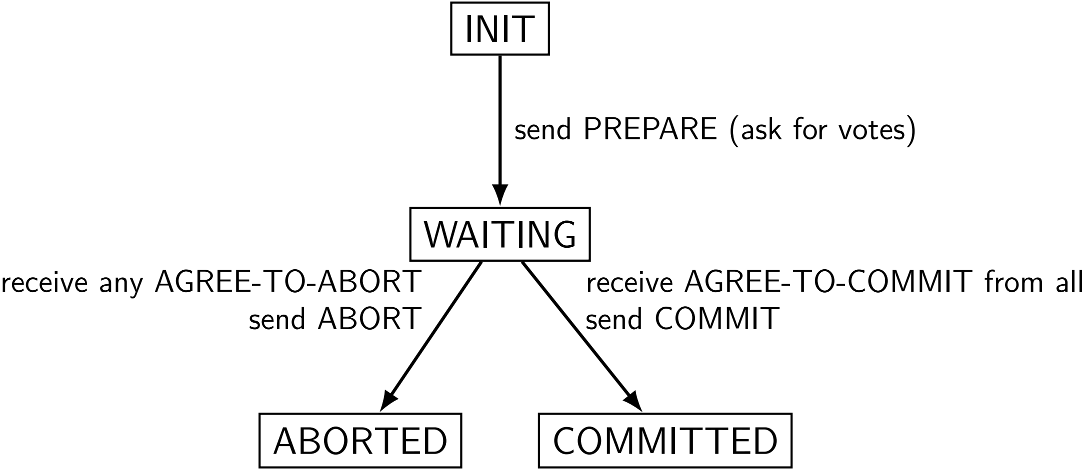
typical tool: state machine

each machine is in some state

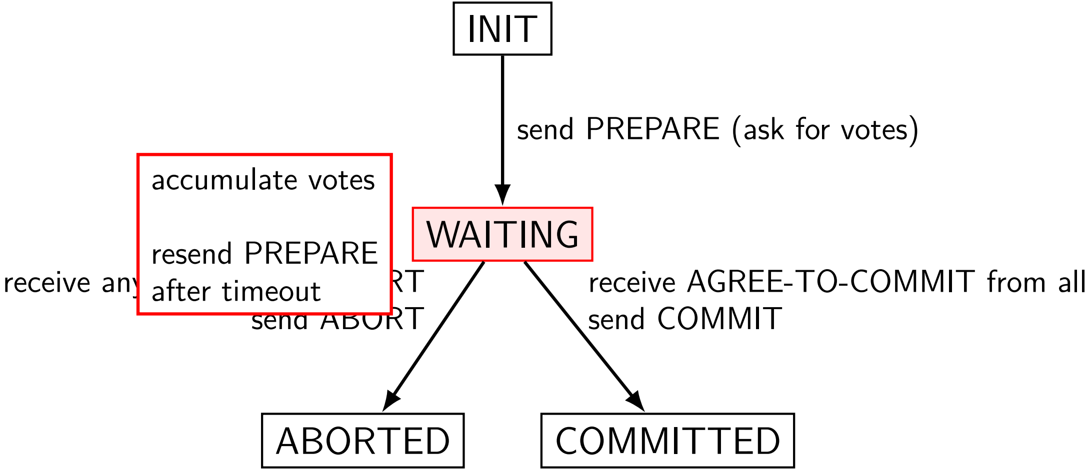
know what every message does in this state

avoids common problem: don't know what message does

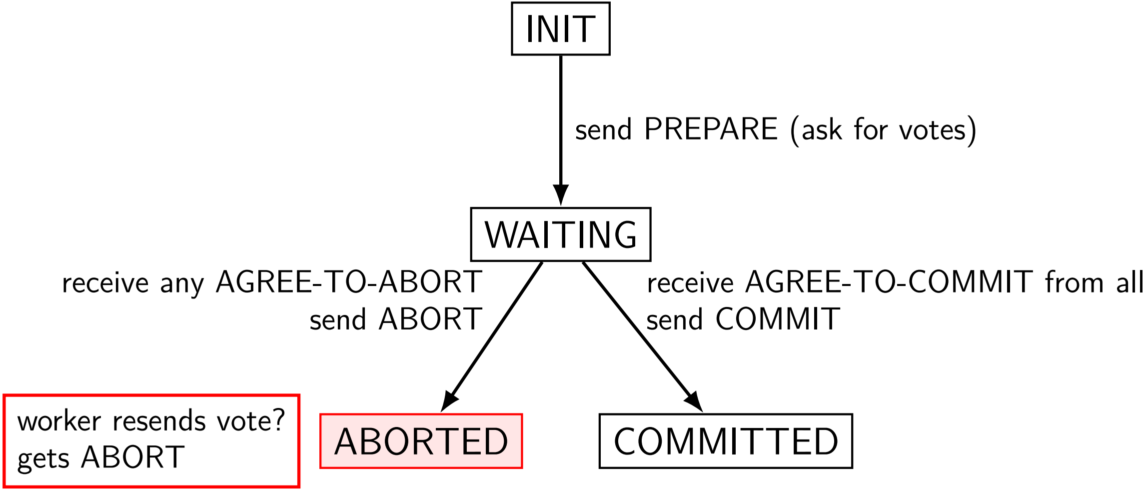
coordinator state machine (simplified)



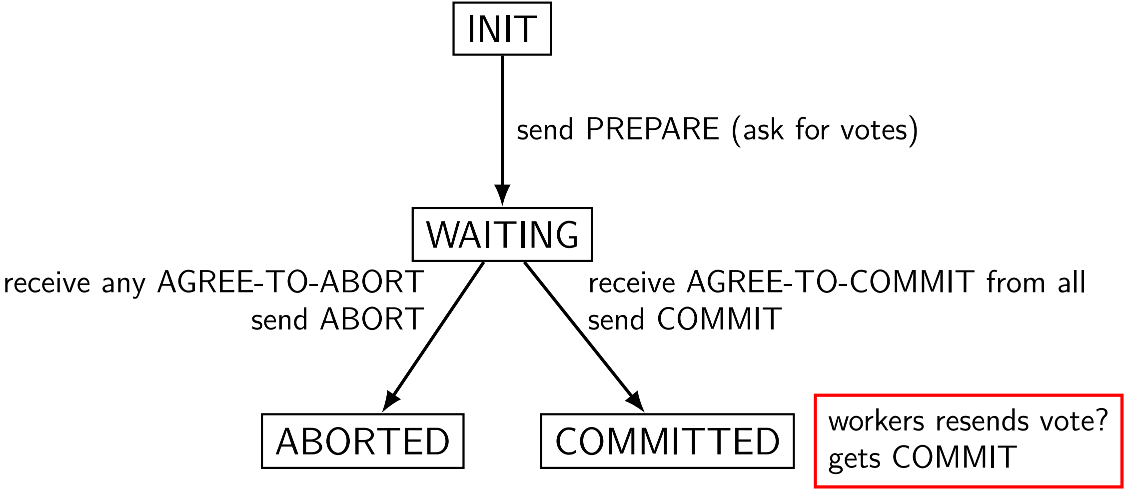
coordinator state machine (simplified)



coordinator state machine (simplified)



coordinator state machine (simplified)



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!

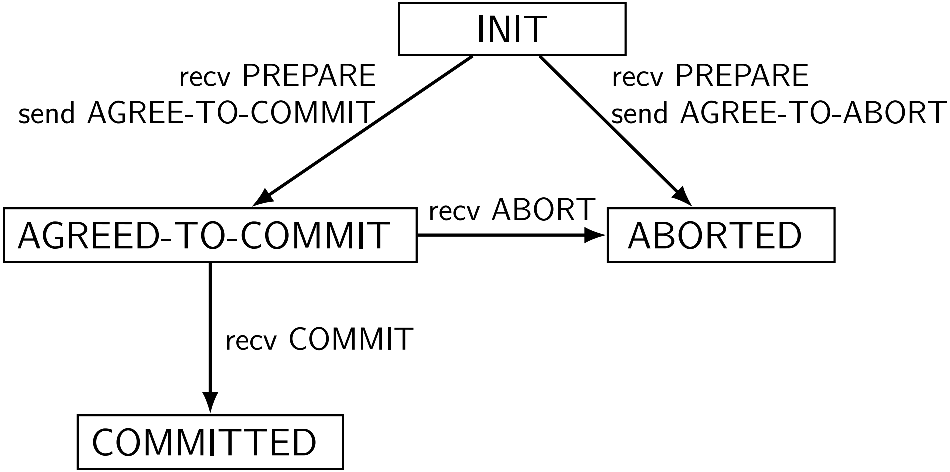
coordinator crashes? *log* indicating last state

- 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

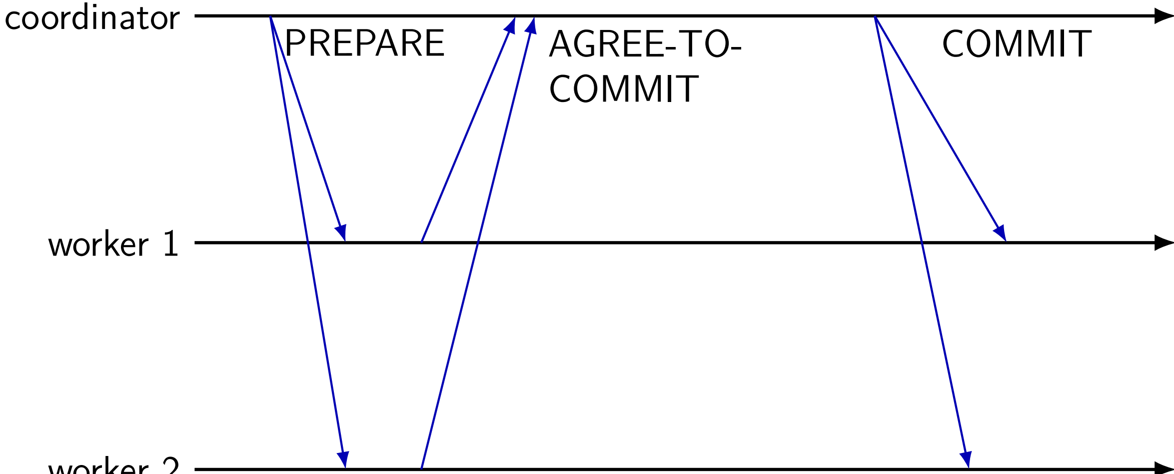
state machine missing details

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

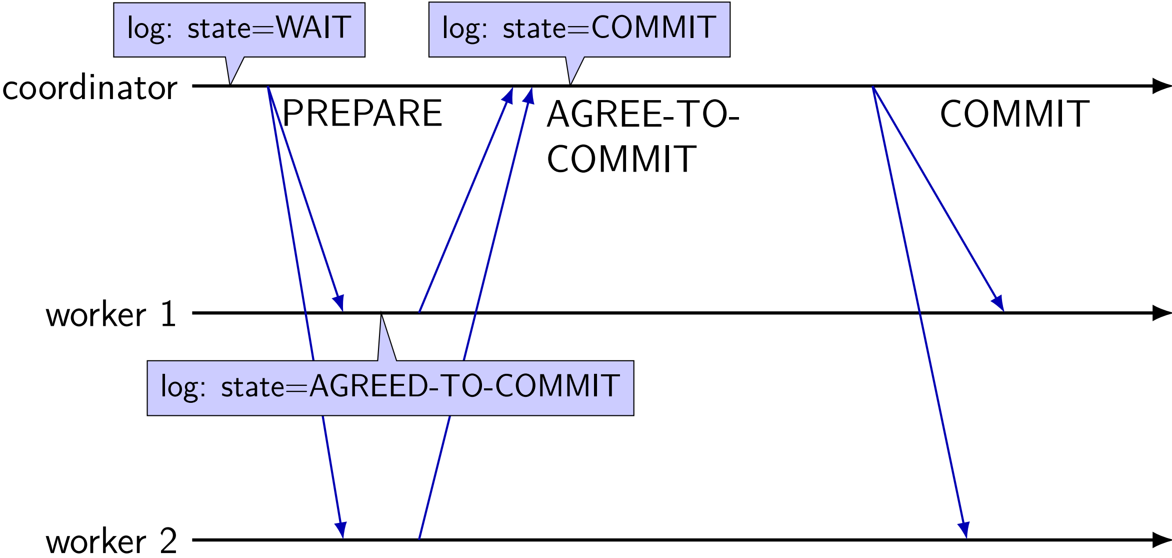
allows verifying properties of state machine

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

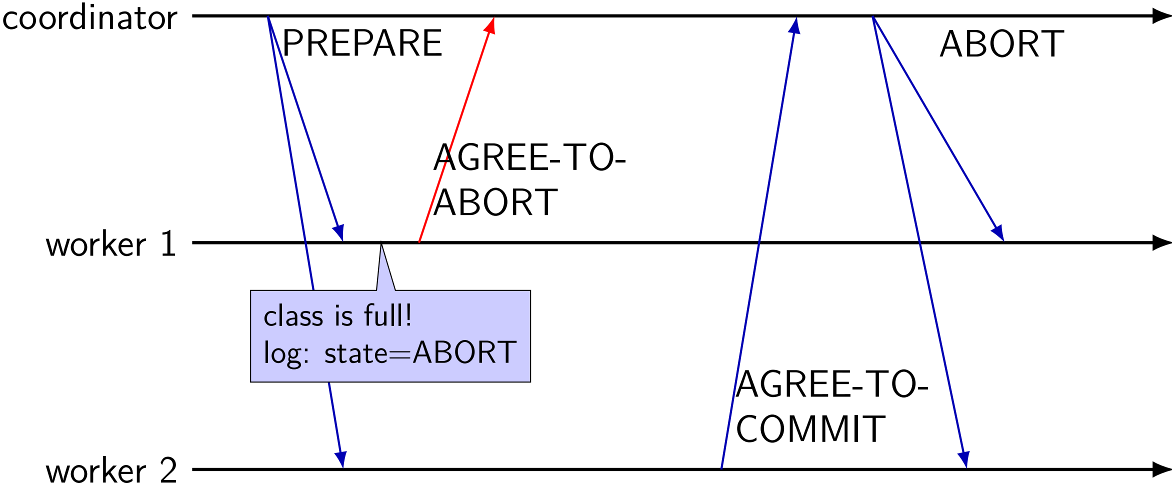
TPC: normal operation



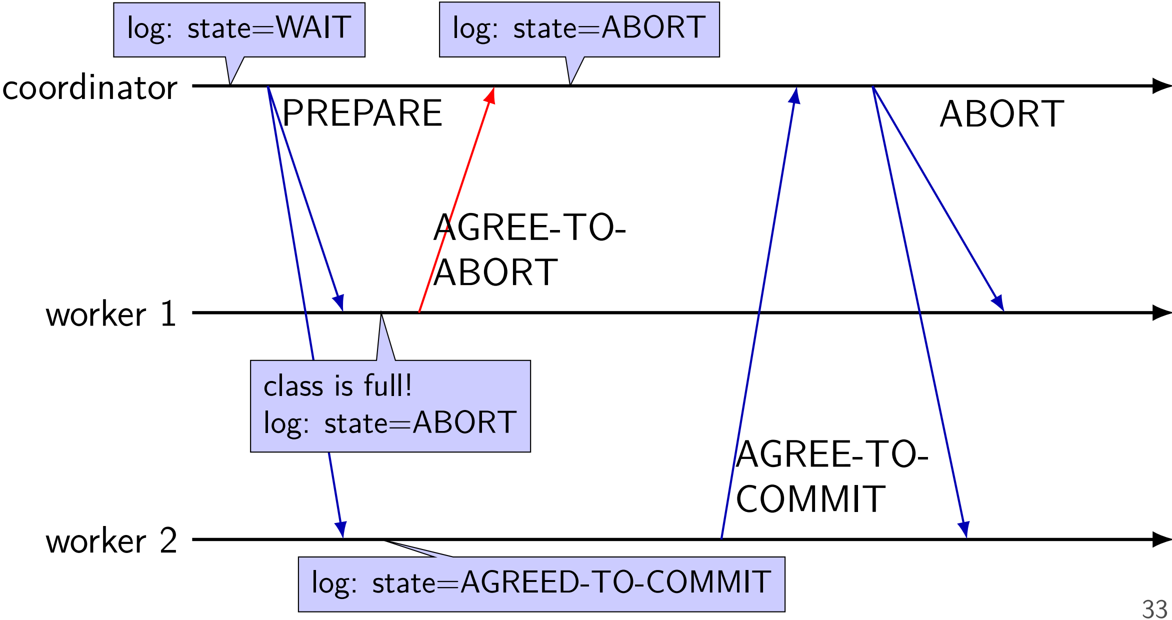
TPC: normal operation



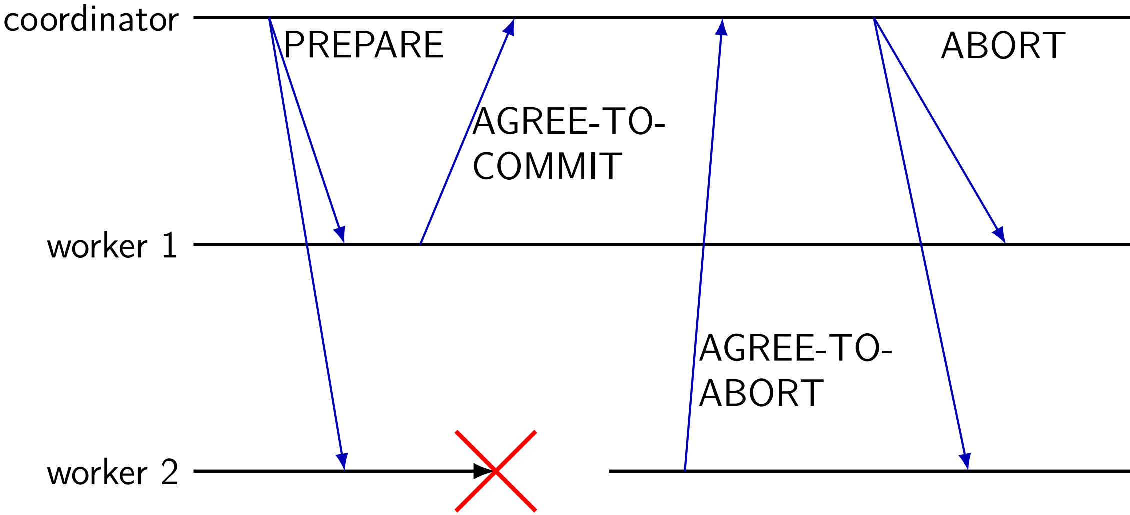
TPC: normal operation — conflict



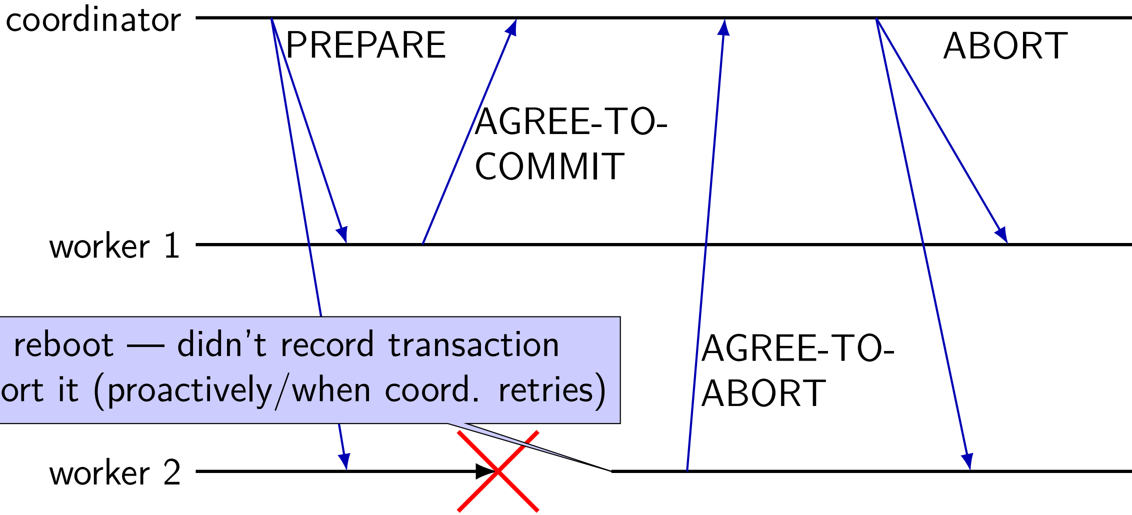
TPC: normal operation — conflict



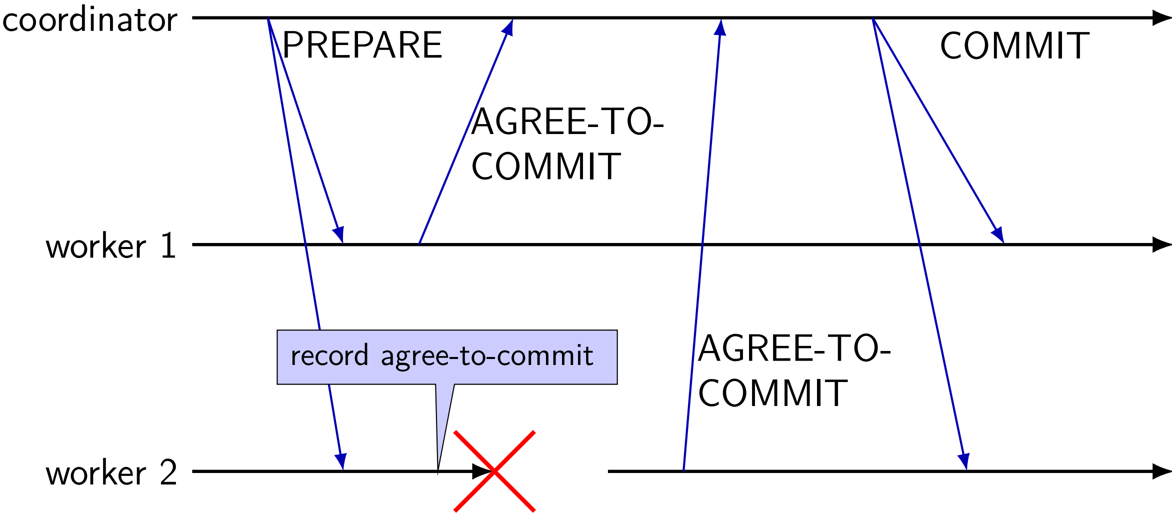
TPC: worker failure (1)



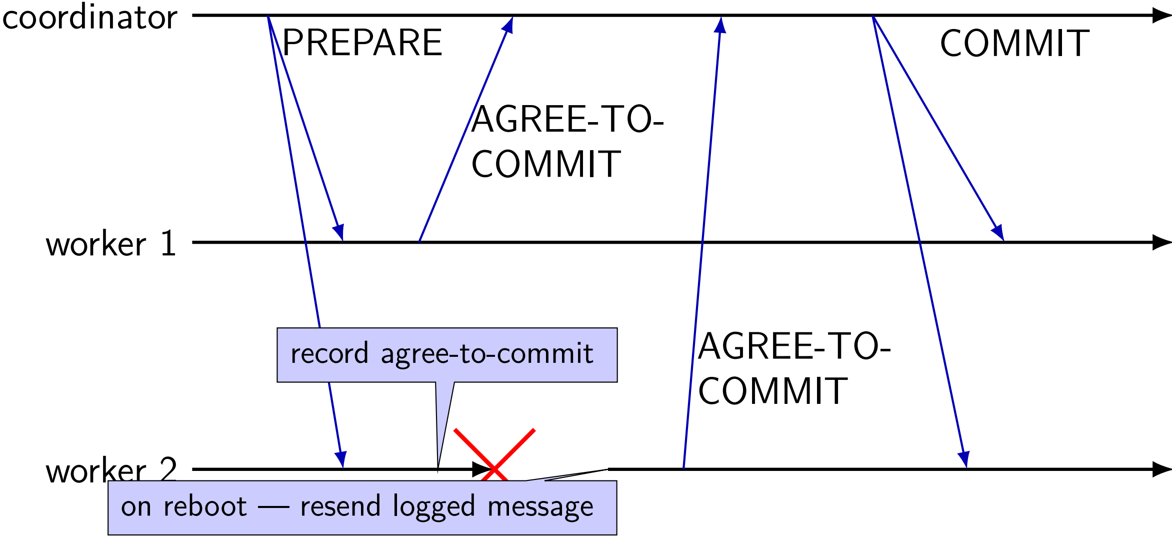
TPC: worker failure (1)



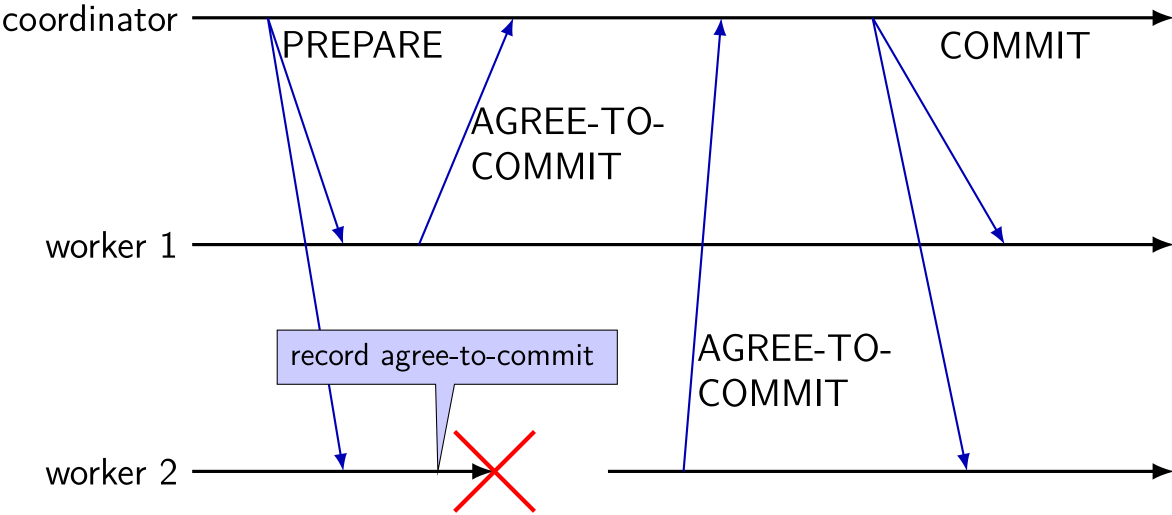
TPC: worker failure (2)



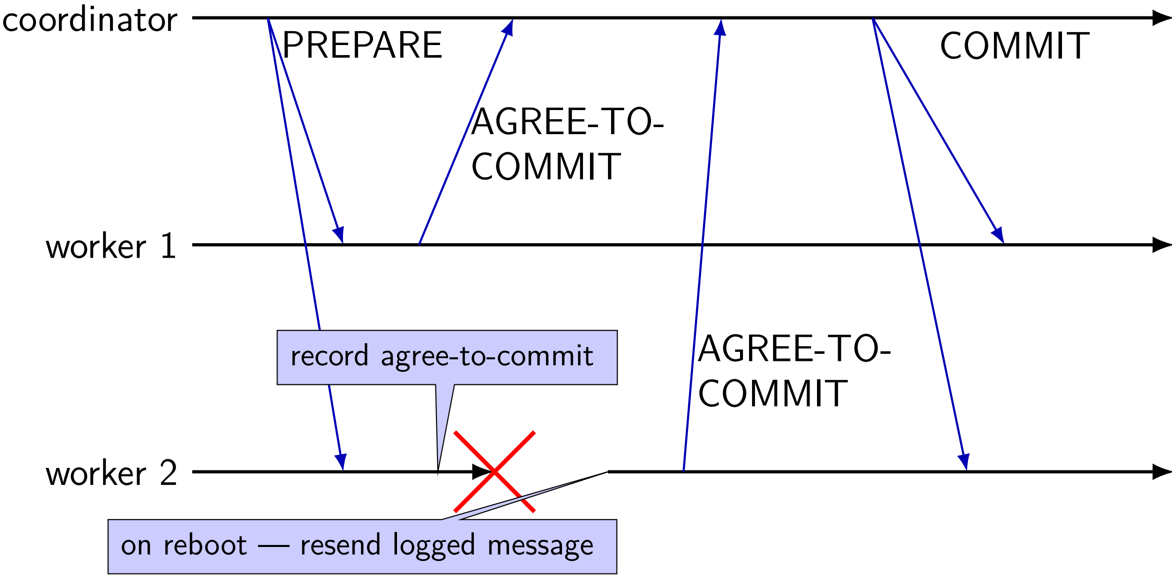
TPC: worker failure (2)



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

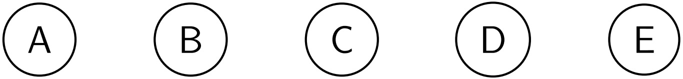
goal: work despite a few failing nodes

just require "enough" nodes to be working

for now — assume fail-stop

nodes don't respond or tell you if broken

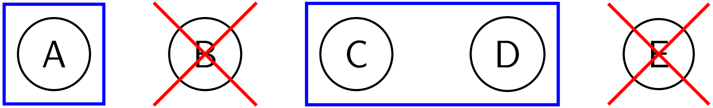
quorums (1)



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

any quorum enough — okay if some nodes fail

quorums (1)



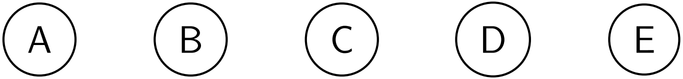
perform read/write with vote of any *quorum* of nodes

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

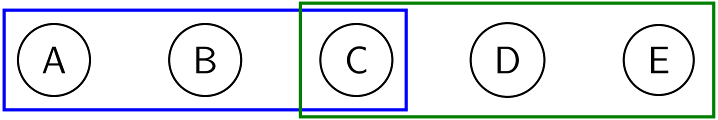
e.g. every operation requires majority of nodes

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

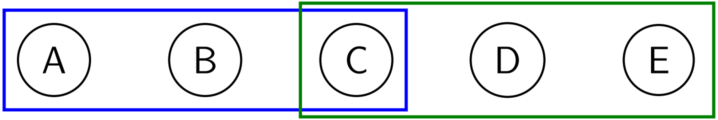
e.g. every operation requires majority of nodes

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

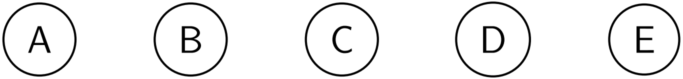
e.g. every operation requires majority of nodes

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)



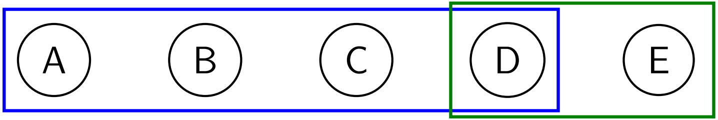
sometimes vary quorum based on operation type

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)



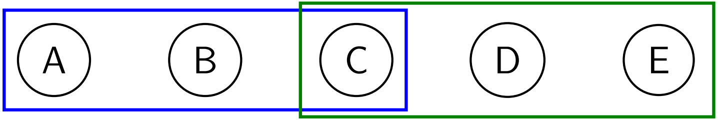
sometimes vary quorum based on operation type

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 Paxis

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

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

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

access control matrix: who does what?

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs
to 1+ *protection domains*:
"user cr4bd"
"group csfaculty"
...

access control matrix: who does what?

objects (whatever type) with restrictions

	file 1	file 2	process 1
domain 1	read/write		
domain 2	read	write	wakeup
domain 3	read	write	kill

each process belongs
to 1+ *protection domains*:
"user cr4bd"
"group csfaculty"
...

representing access

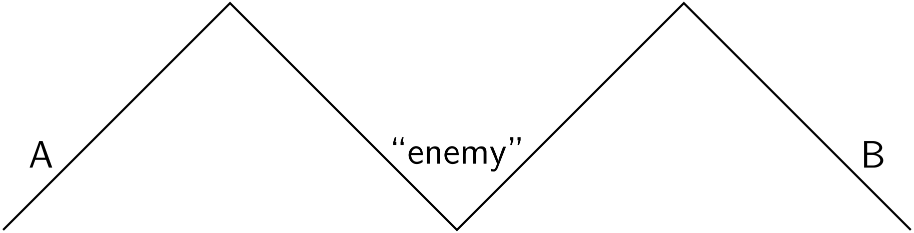
with objects (files, etc.): *access control list*

- list of protection domains (users, groups, processes, etc.) allowed to use each item

list of (domain, object, permissions) stored "on the side"

- example: AppArmor on Linux
- configuration file with list of program + what it is allowed to access
- prevent, e.g., print server from writing files it shouldn't

two general's problem (setup)



general A and B want to agree on time to attack enemy (center)

only attack if they know the other will

- attack together: victory
- attack separately: defeat

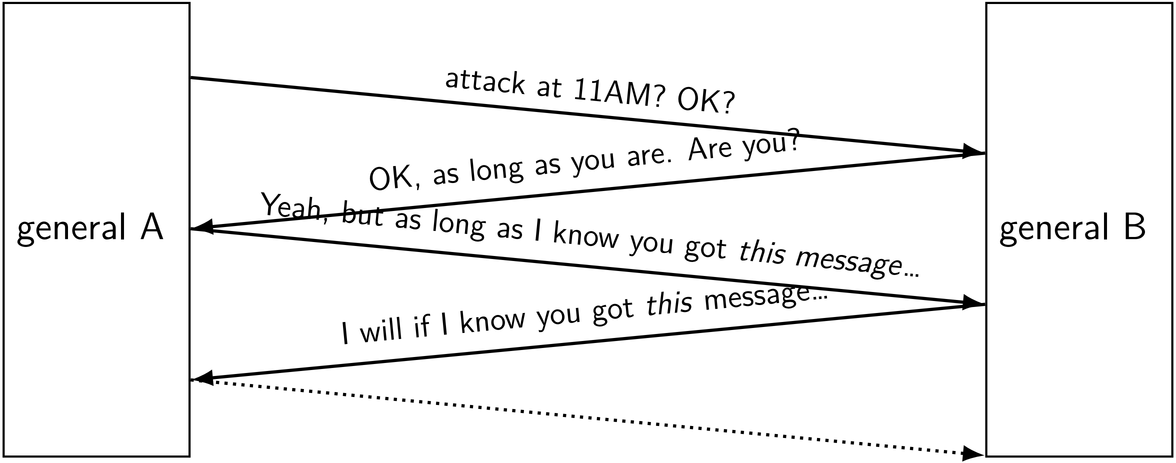
communication mecahnism: unreliable messengers

- could be captured by enemy — message lost

two general's problem

recall: *both* agree to attack at same time

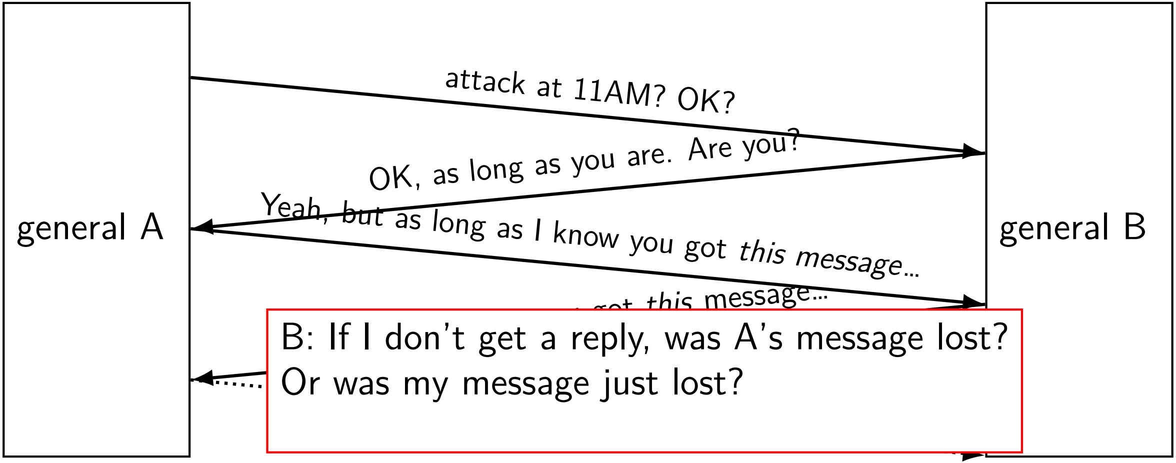
(otherwise don't attack — sure defeat)



two general's problem

recall: *both* agree to attack at same time

(otherwise don't attack — sure defeat)



impossibility

can't gaurentee that both parties will attack

...even if no messages are lost

proof sketch:

some message flips A's state from "attacking" to "not attacking"
...but what if that message is lost — contradiction

relaxing assumptions

can't get gaurentee of receiving message

in practice: best approximation

wait for acknowledgement

retry on timeout

lots of timeouts — look like machine failure