last time

distributed transaction idea

problems with naive solutions

- values being changed — need something like locking
- leave operation half-done on failure

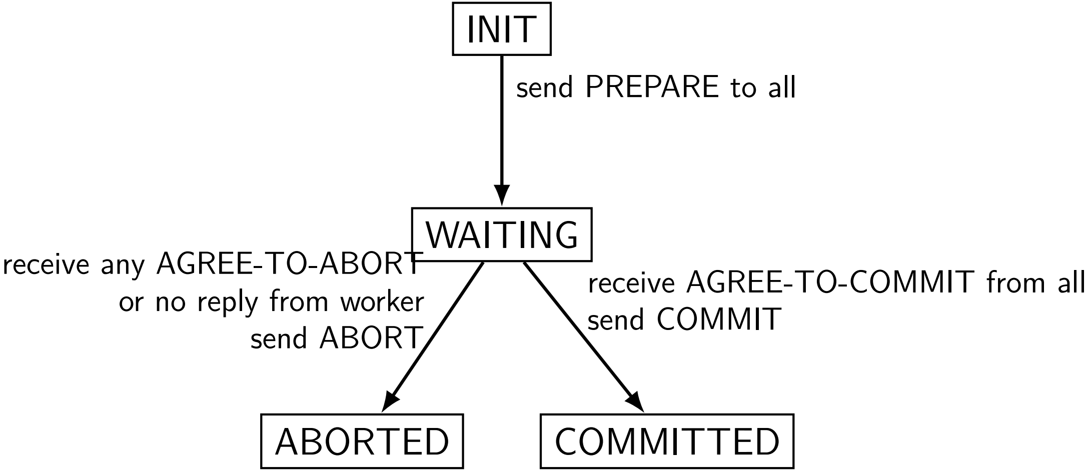
two-phase commit

- prepare: see if everyone can do it
- commit: every does it
- log intended actions before doing it

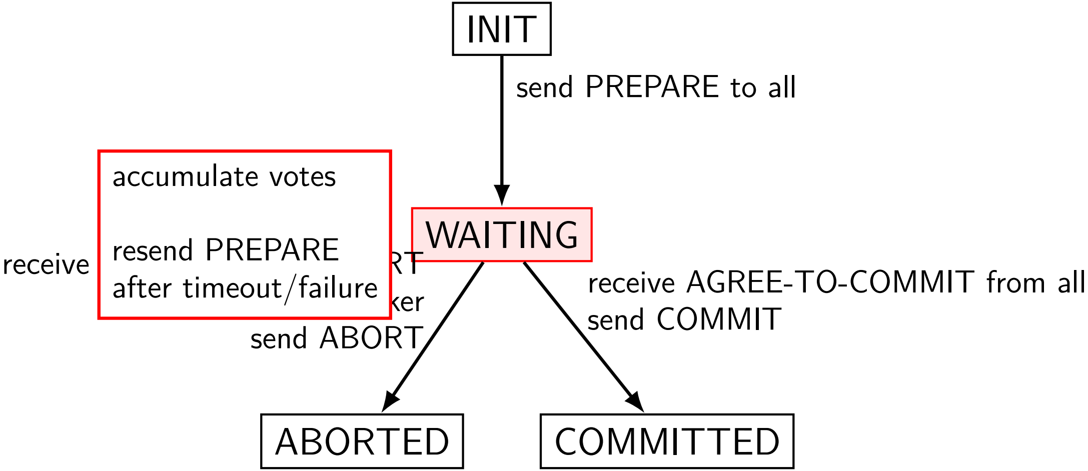
mapping protocol to state machine

- "current state" recorded in log
- redo action to enter state on restart
- identify what to do for each message in each state

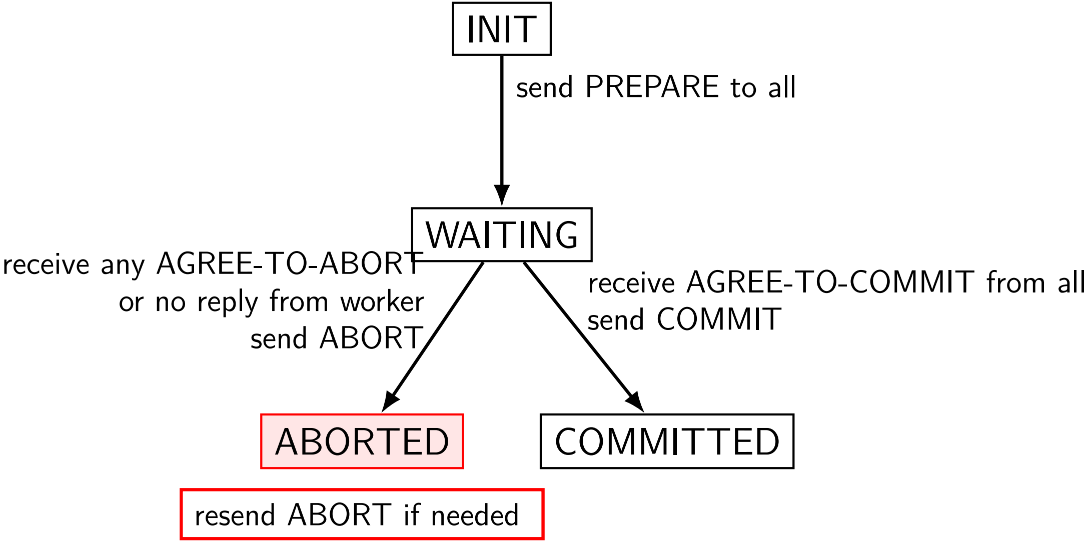
coordinator state machine (simplified?)



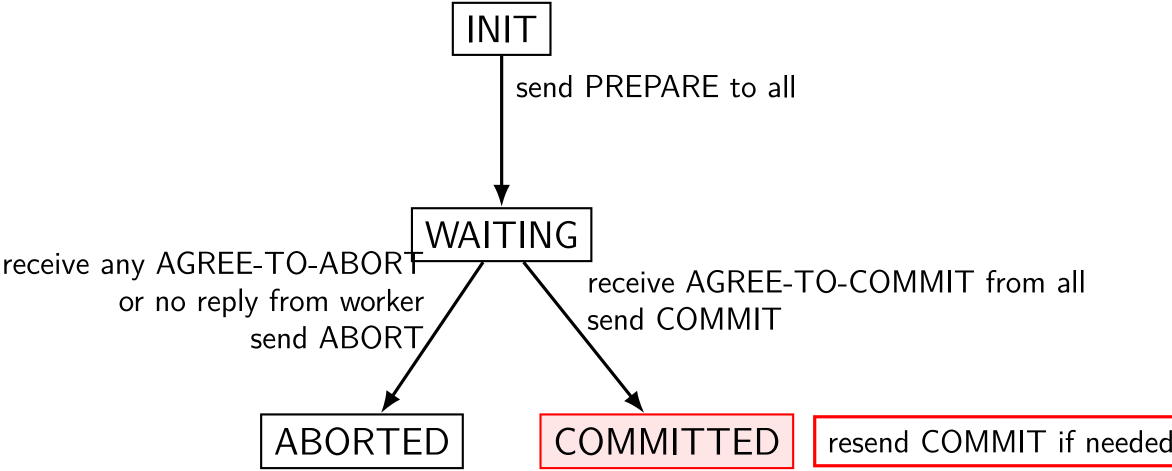
coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator state machine (simplified?)



coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

  or, if allowed, maybe send ABORT

worker doesn't get COMMIT/ABORT?

  in assignment: worker sends acknowledgment; arrange retry if no ack

  other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

$\rightarrow$ resend last message

    or, if allowed, maybe send ABORT

worker doesn't get COMMIT/ABORT?

    in assignment: worker sends acknowledgment; arrange retry if no ack
    other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

workers need to handle duplicate messages!
coordinators need to handle duplicate replies!

in assignment: worker sends acknowledgment, arrange retry if no ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worker case: log written, but message not sent

→ resend last message

or, if allowed, maybe send ABORT

worker do

haven't sent commit? can abort instead (simpler?)

in assi ... ack

other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

worst case: log written, but message not sent

in assignment, errors detected only at coordinator
using gRPC — so have return value from “COMMIT” RPC

worker doesn’t get COMMIT/ABORT?

- in assignment: worker sends acknowledgment; arrange retry if no ack
- other option: worker asks again after timeout

coordinator failure recovery

duplicate messages okay — unique transaction ID!

coordinator crashes? *log* indicating last state

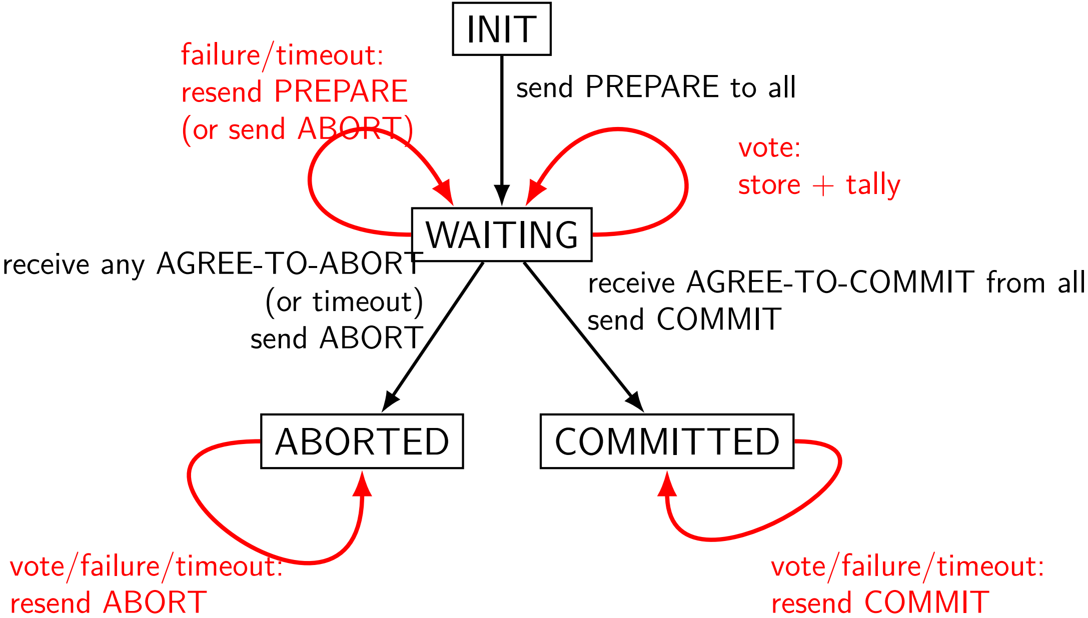
worst case: log written, but message not sent

normal strategy: wait for timeout, then resend
assignment: you throw exception; we'll restart (easier testing)

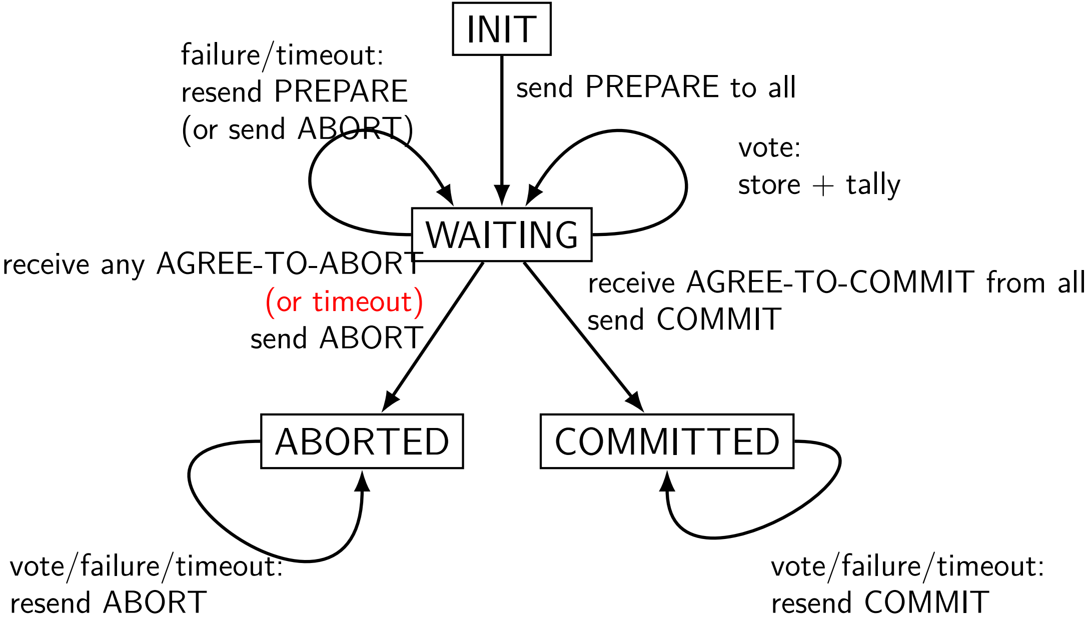
worker doesn't get COMMIT/ABORT?

- in assignment: worker sends acknowledgment; arrange retry if no ack
- other option: worker asks again after timeout

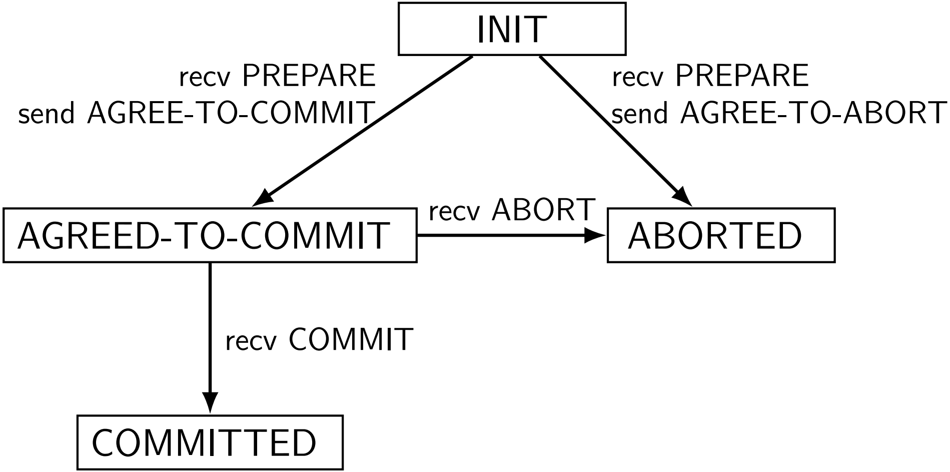
coordinator state machine (less simplified?)



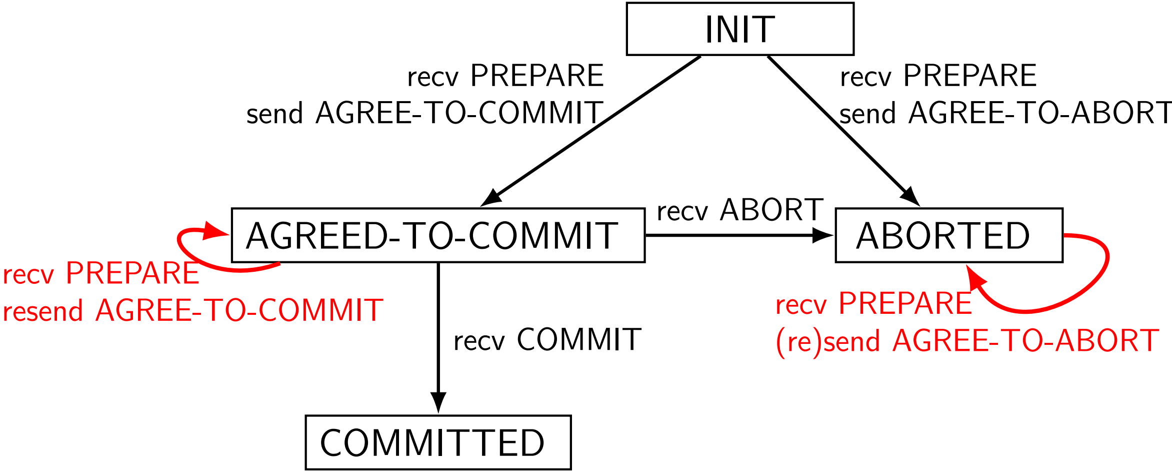
coordinator state machine (less simplified?)



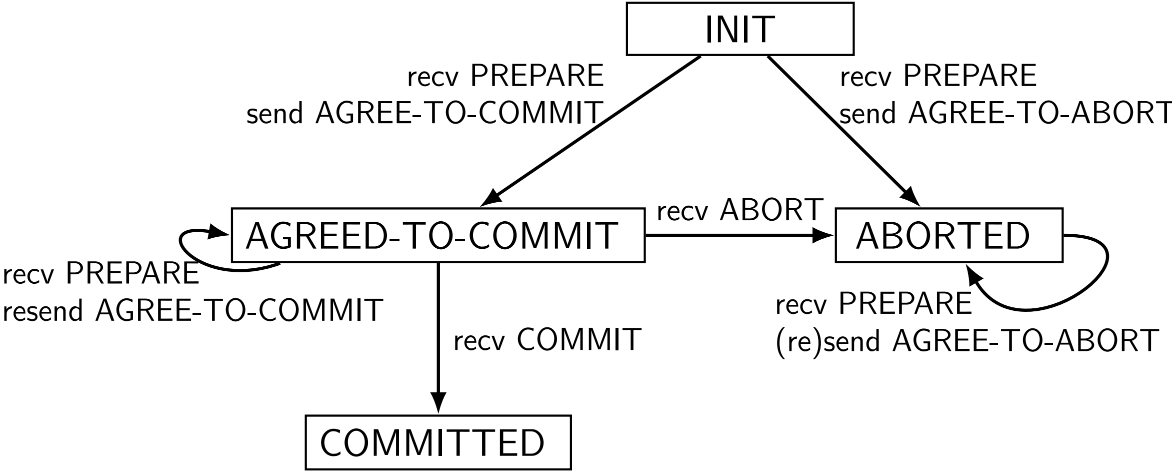
worker state machine (simplified)



worker state machine (less simplified?)



worker state machine (less simplified?)



worker failure recovery

worker crashes? *log* indicating last state

log written before acting on that state

if INIT:
wait for PREPARE (resent)?

if AGREE-TO-COMMIT or ABORTED:
resend AGREE-TO-COMMIT/ABORT

if COMMITTED:
redo operation (just like redo logging)

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?
- ...

everything specified: machine checkable?

want to discard finished transactions eventually

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 with return value; RPC call fails if problem getting return value

workers might never agree-to-abort (and that's okay)

- no conflicting operations: only crash or agree-to-commit

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

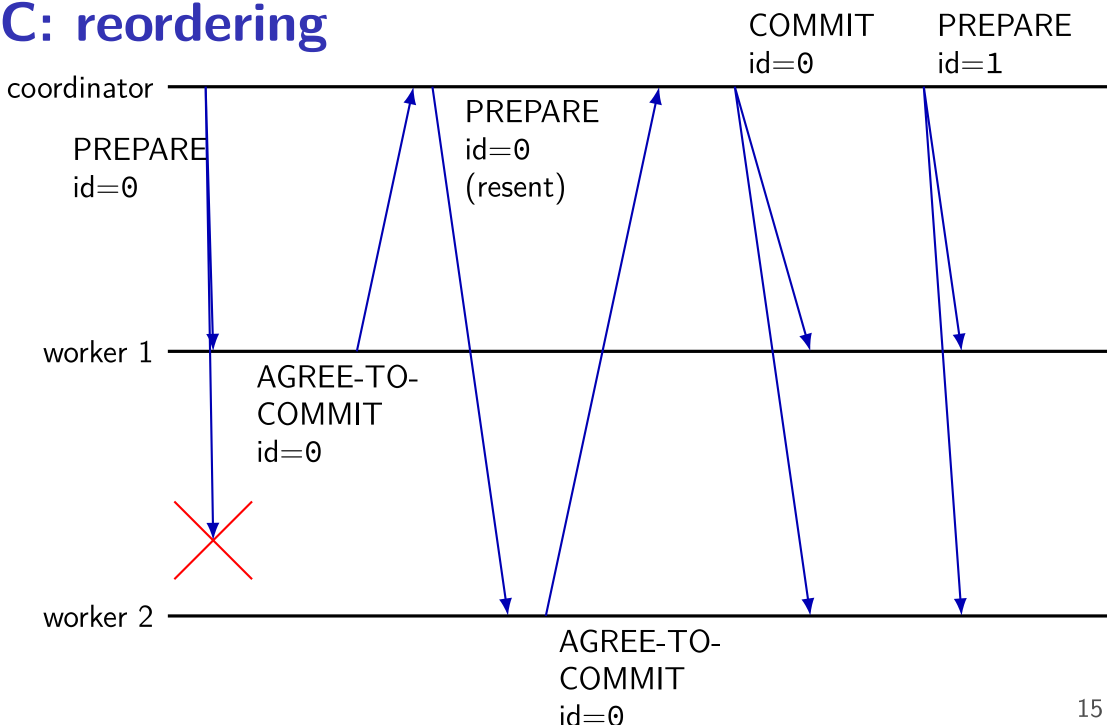
- it gets lost
- it gets sent, but acknowledgment/reply is lost
- it gets sent, but delayed until after another RPC

assignment: failure types

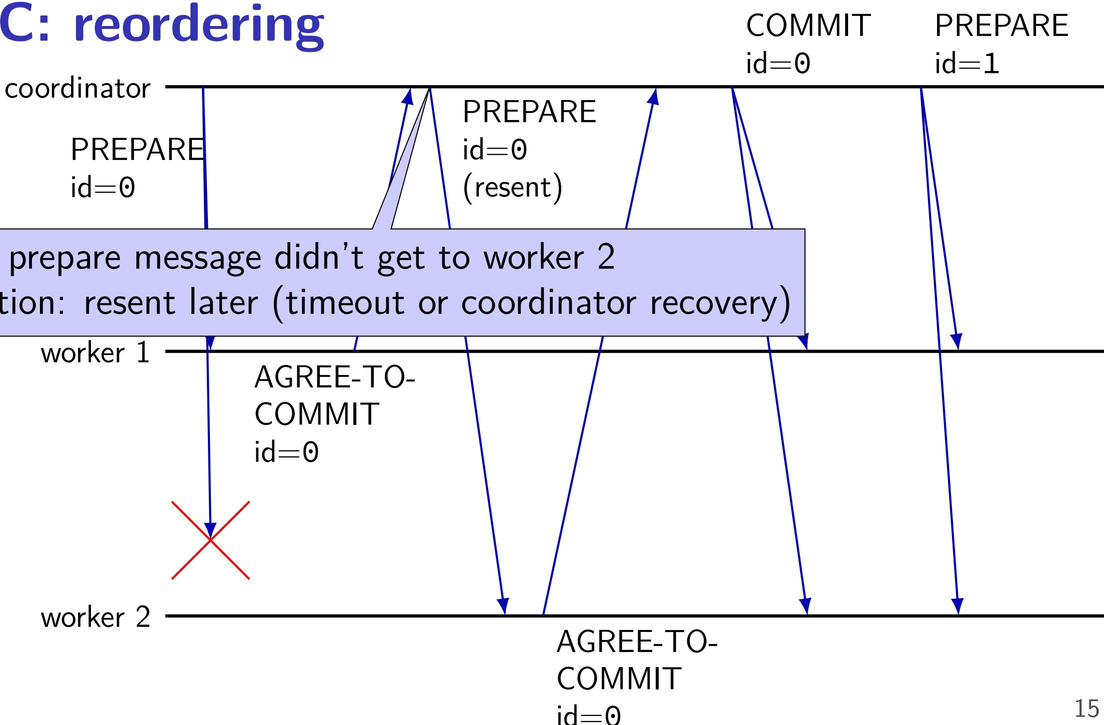
send RPC and

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

TPC: reordering

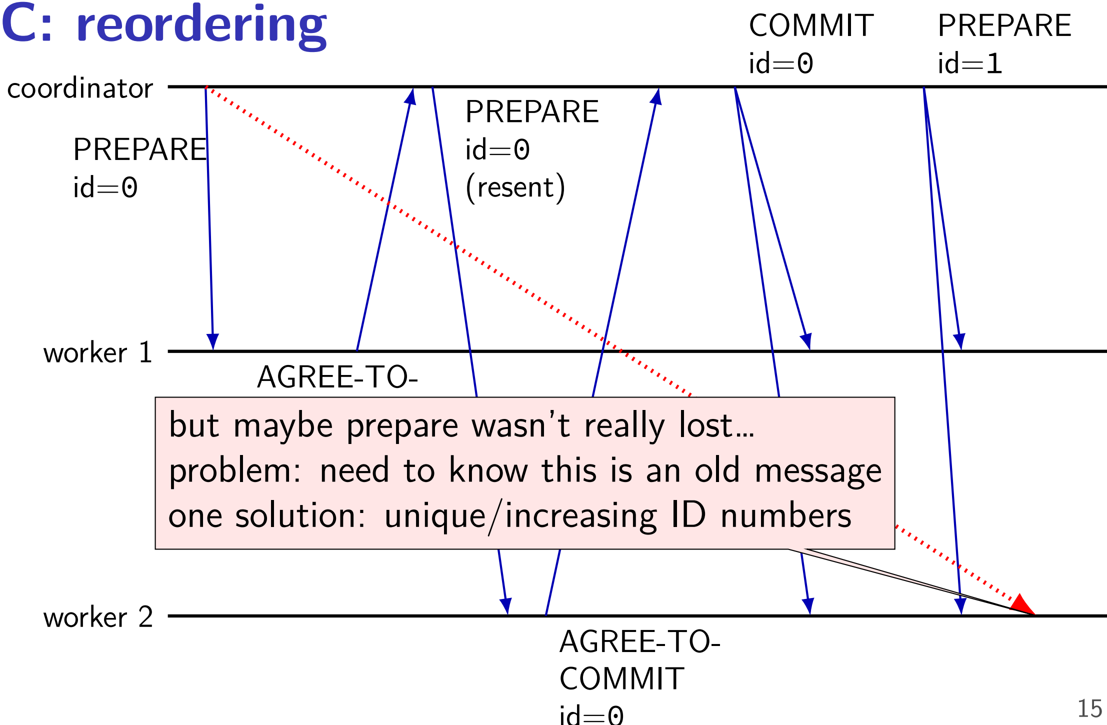


TPC: reordering



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 $\rightarrow$ must be ignored!

solution: sequence numbers or transactions ID and/or timestamps

some way to tell "this is old"

worker failure during prepare

worker failure after prepare without sending vote?

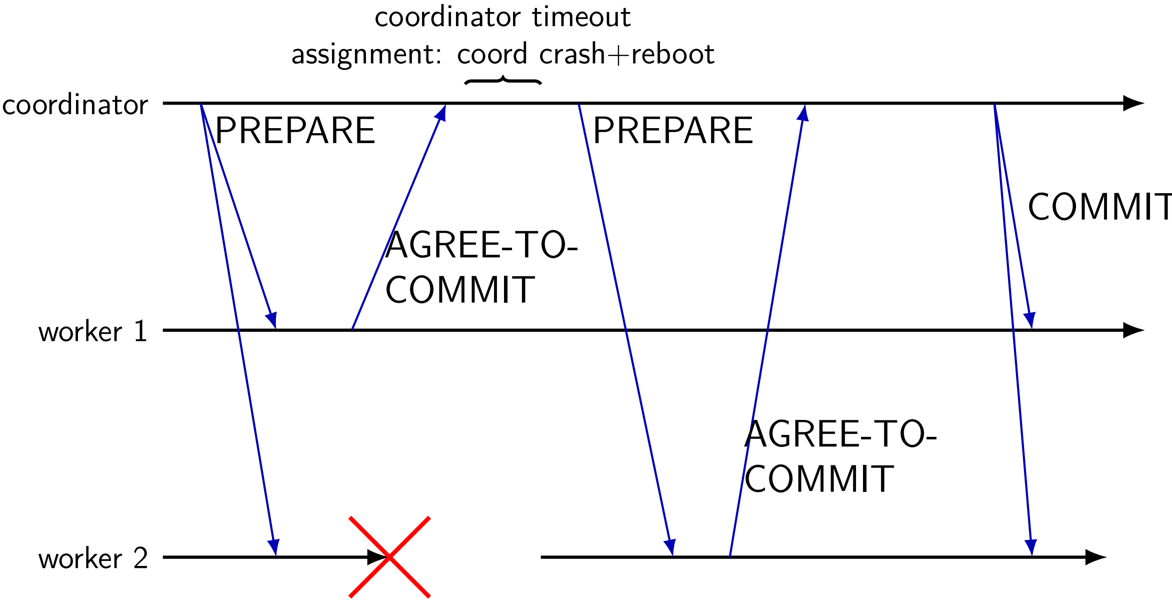
- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

worker failure during prepare

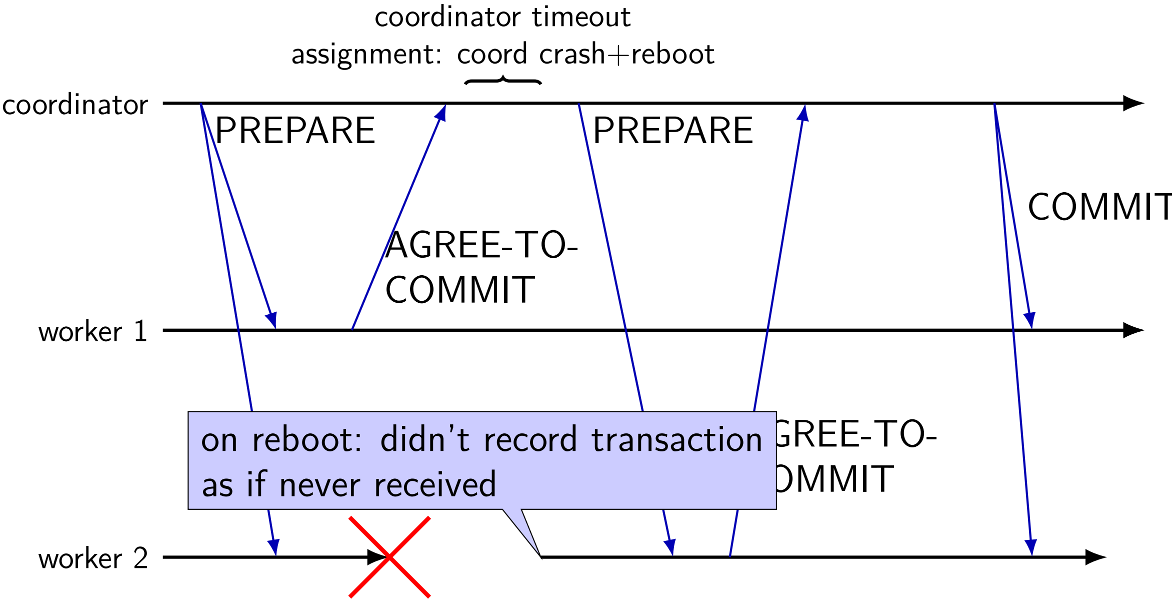
worker failure after prepare without sending vote?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

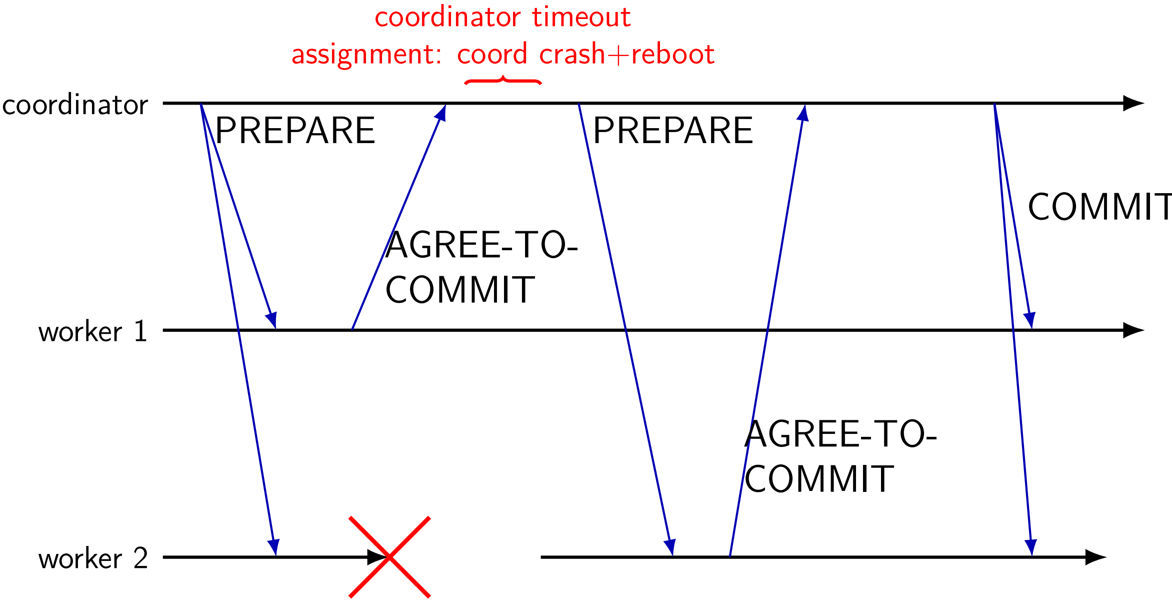
TPC: worker fails after prepare (1a)



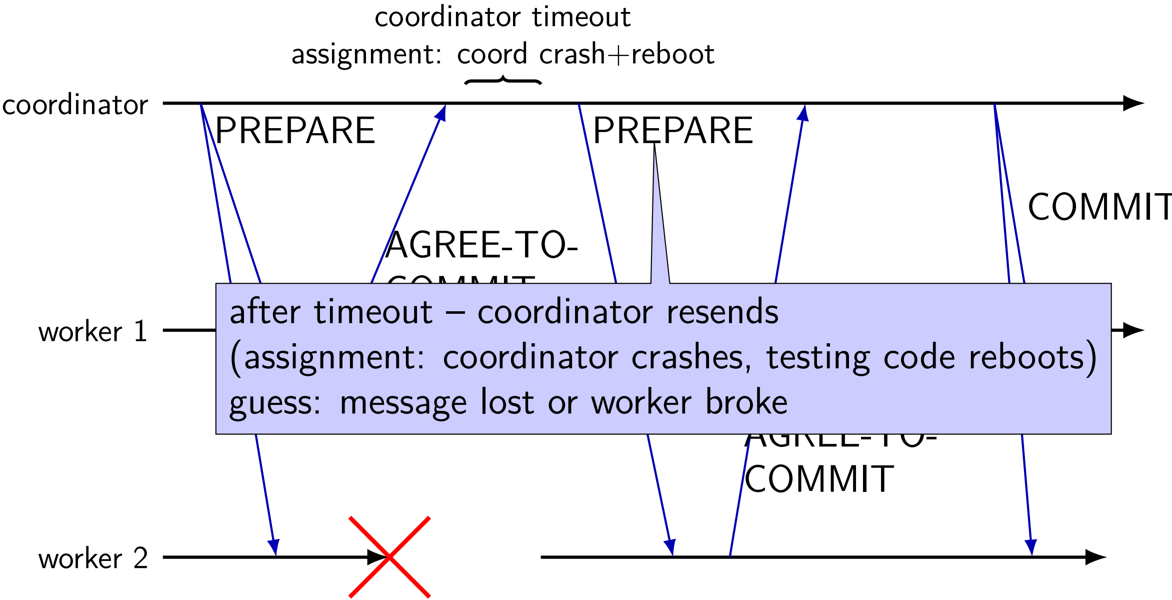
TPC: worker fails after prepare (1a)



TPC: worker fails after prepare (1a)

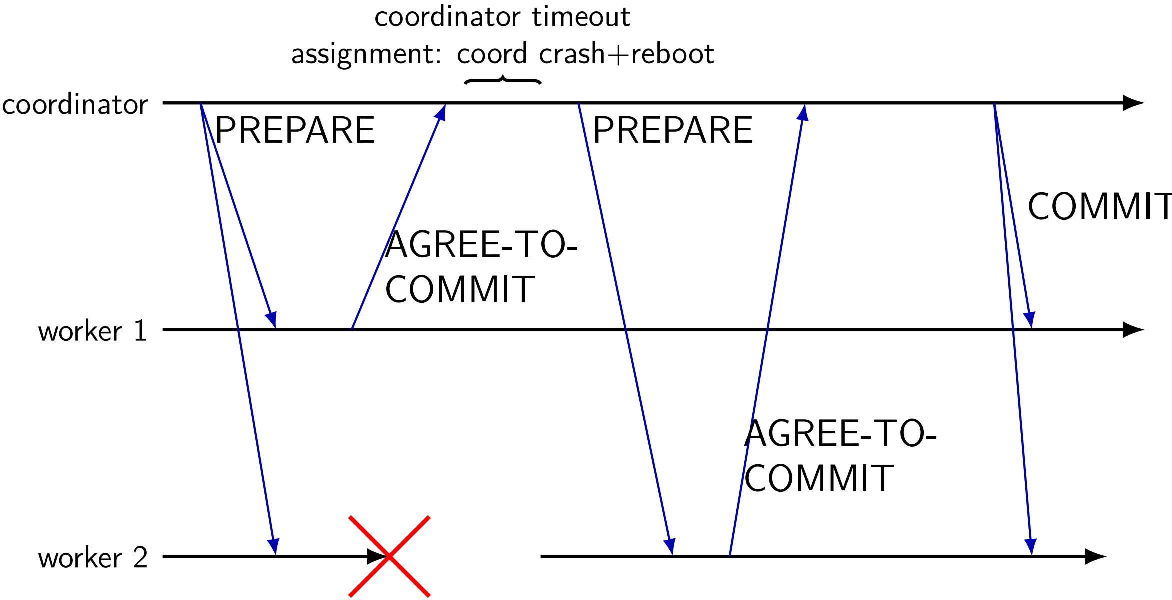


TPC: worker fails after prepare (1a)

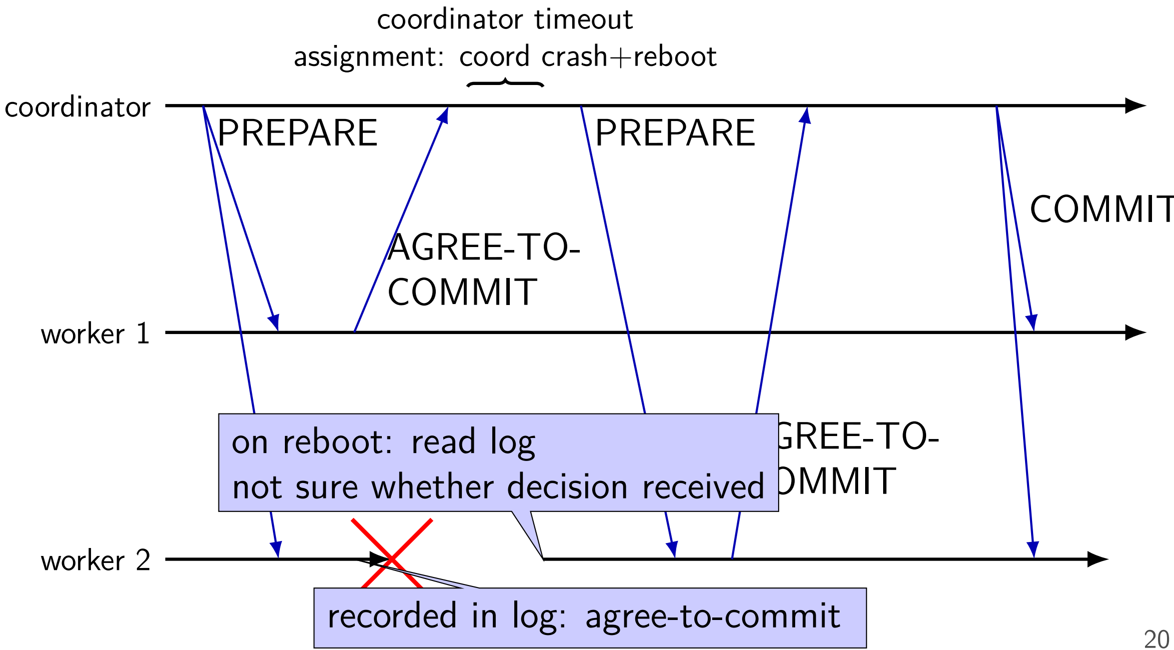


after timeout – coordinator resends
(assignment: coordinator crashes, testing code reboots)
guess: message lost or worker broke

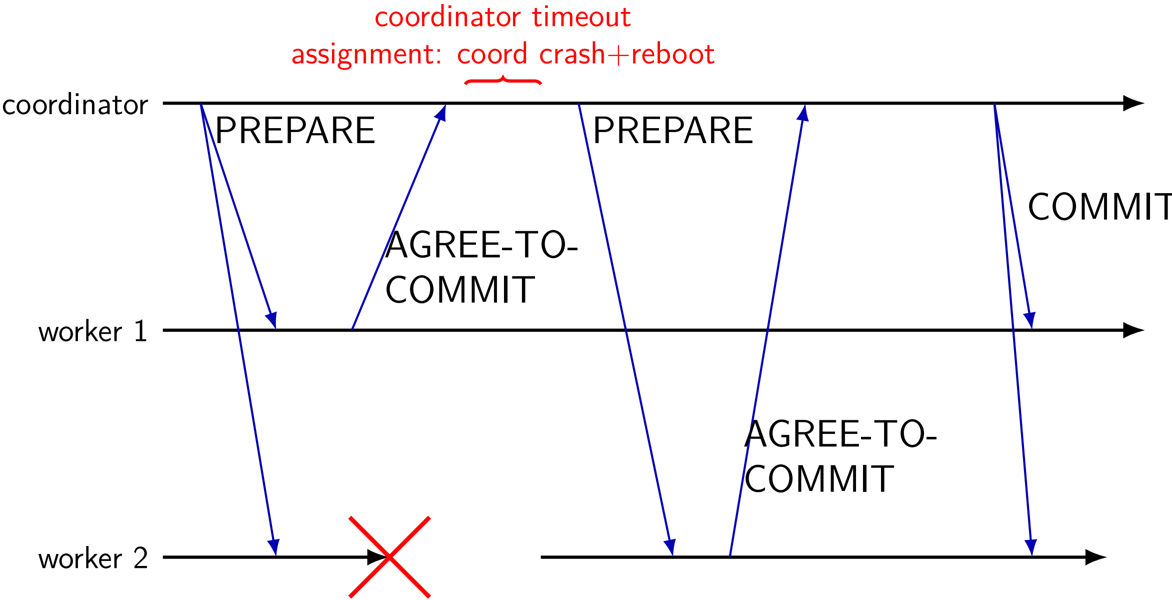
TPC: worker fails after prepare (1b)



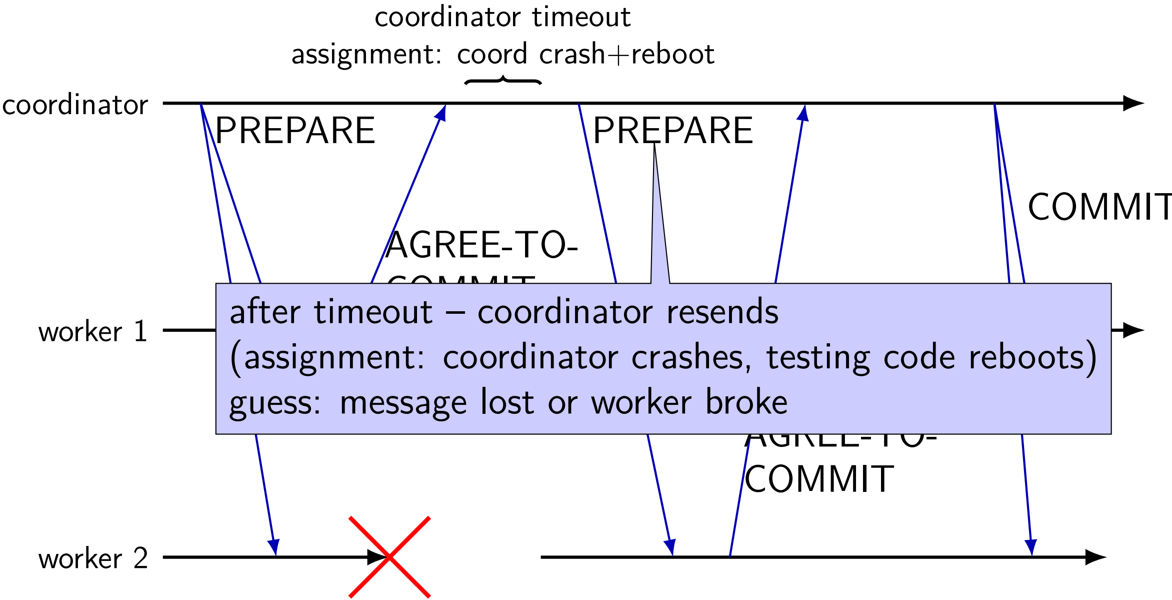
TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)



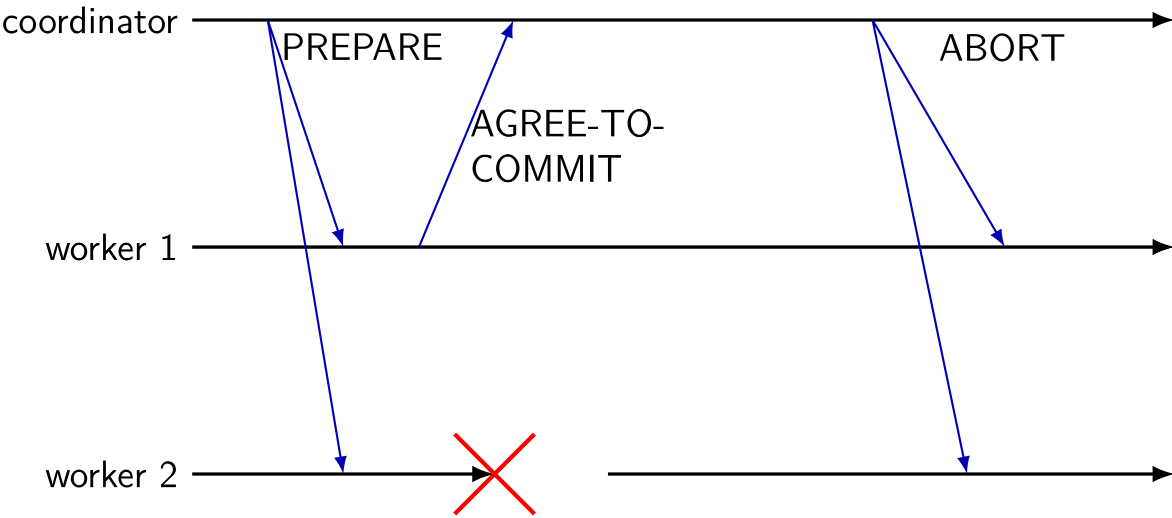
after timeout – coordinator resends
(assignment: coordinator crashes, testing code reboots)
guess: message lost or worker broke

worker failure during prepare

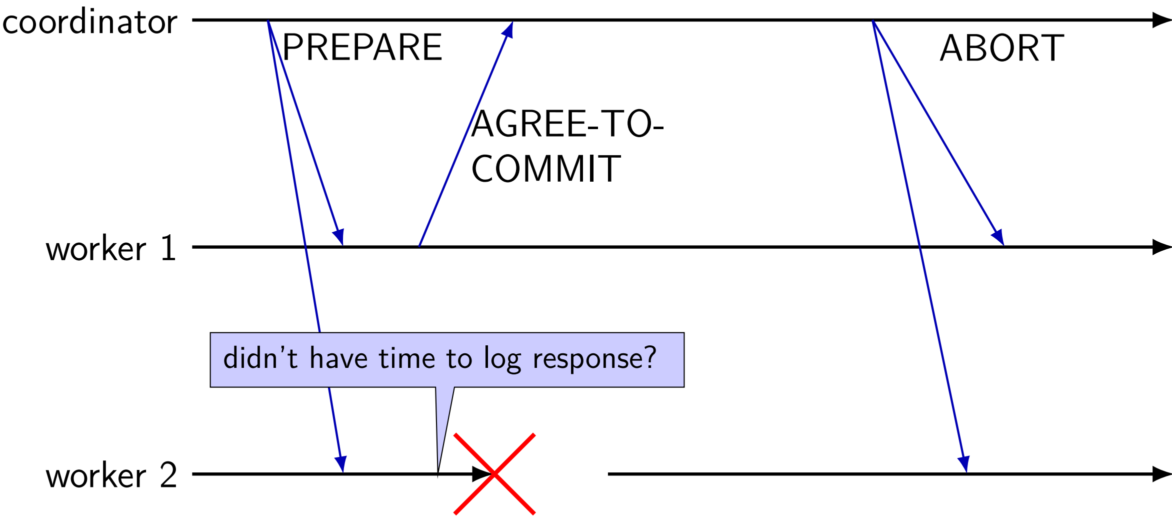
worker failure after prepare without sending vote?

- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

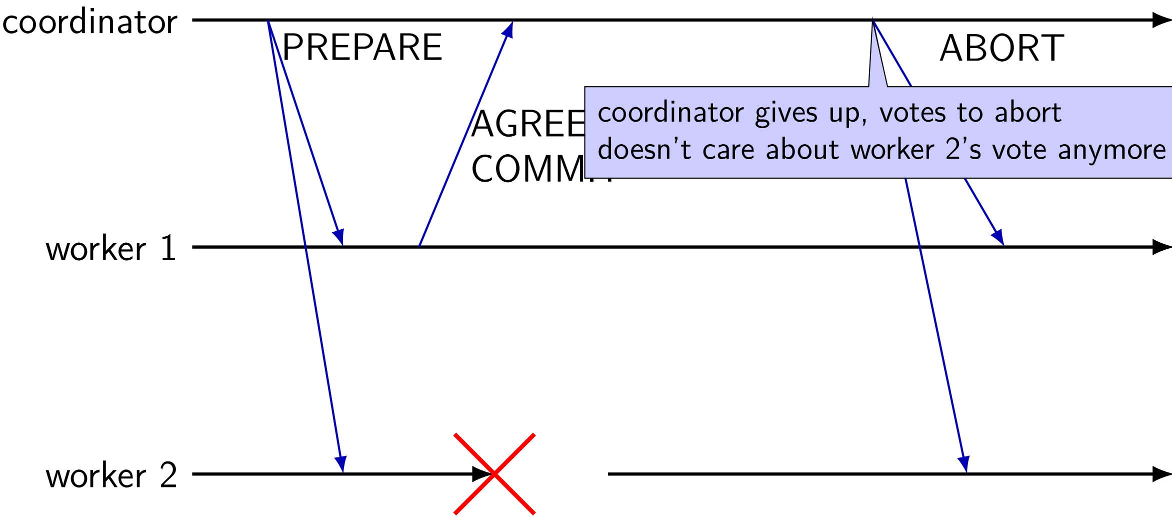
TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)



worker failure during prepare

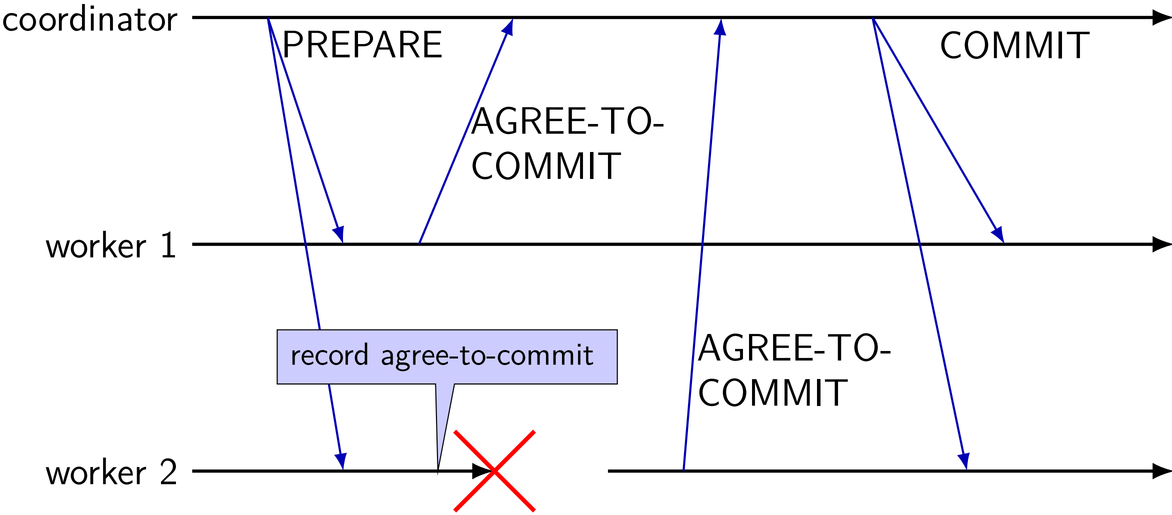
worker failure after prepare without sending vote?

option 1: coordinator retries prepare

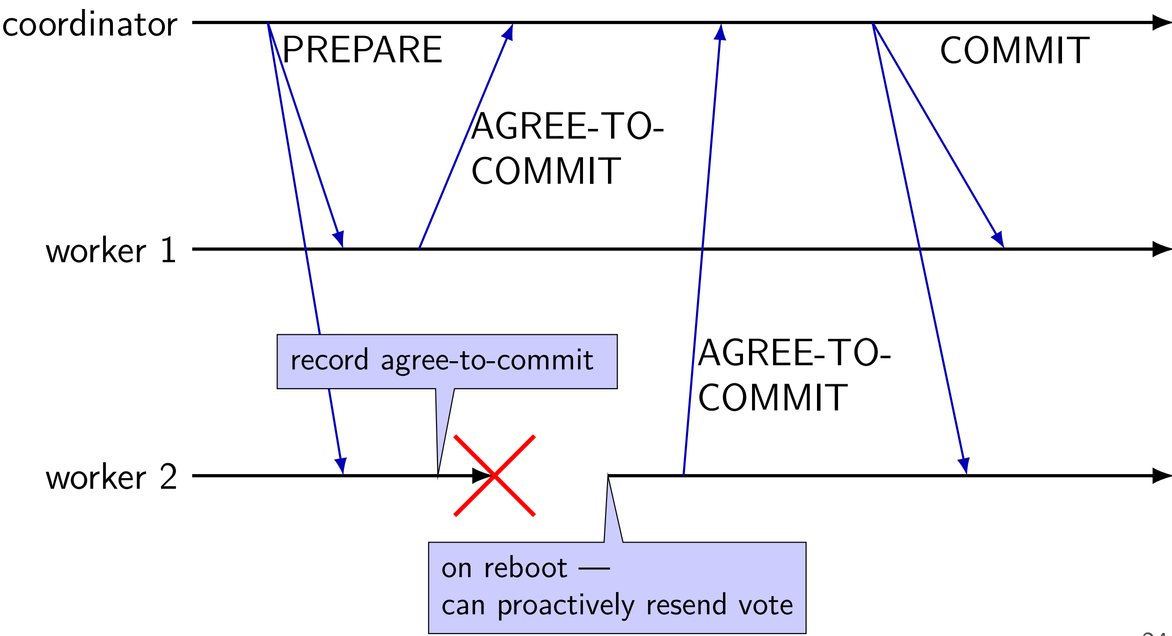
option 2: coordinator gives up, sends abort

option 3: worker resends vote proactively

TPC: worker fails after prepare (3)



TPC: worker fails after prepare (3)



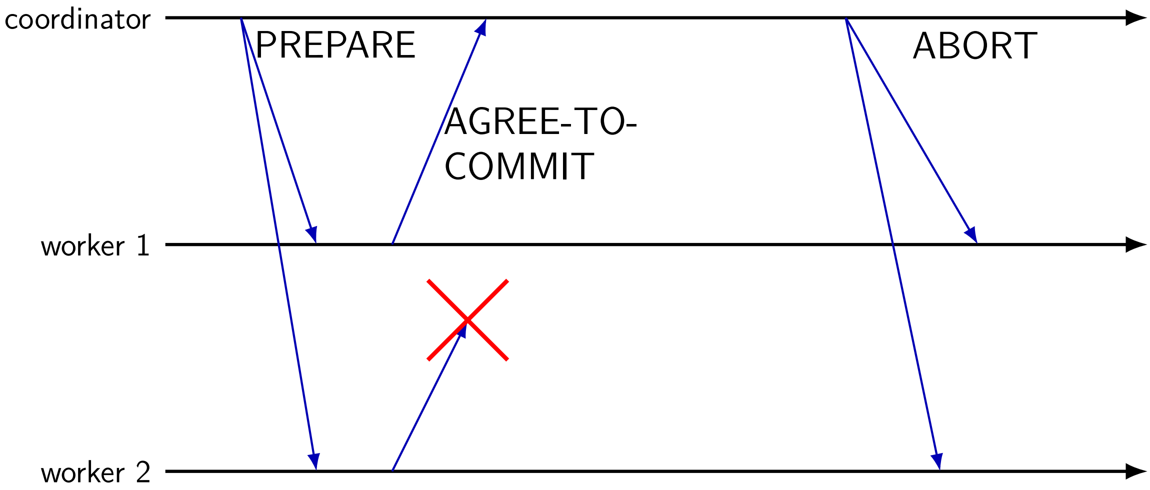
network failure after during voting?

network failure during voting $\approx$ node failure

same options:

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

TPC: network failure (1)



worker failure during commit

worker failure during commit?

- option 1: coordinator resends outcome somehow?
  - requires acknowledgements from worker
  - required for assignment
- option 2: worker resends vote (coordinator resends outcome)

NB: coordinator cannot give up

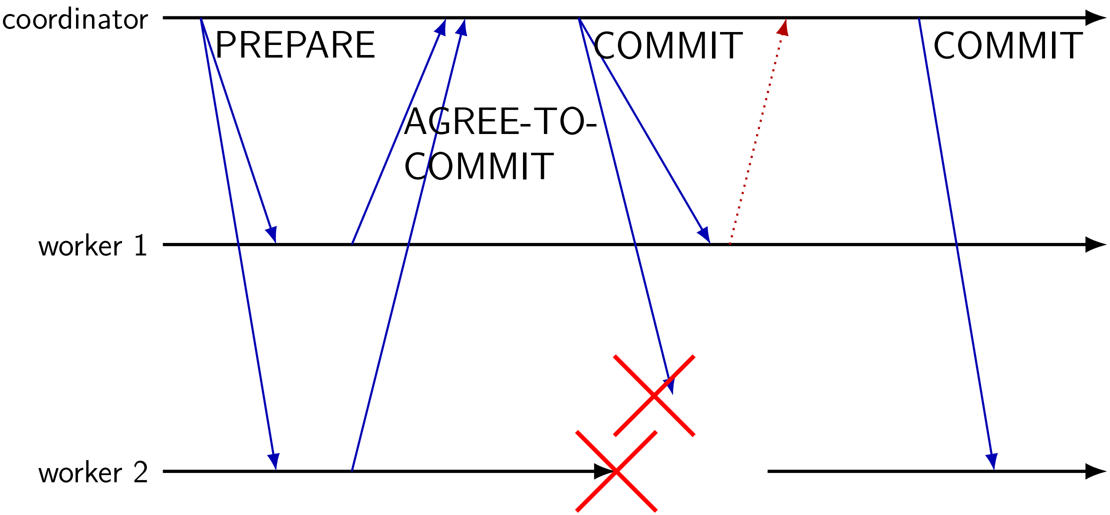
worker failure during commit

worker failure during commit?

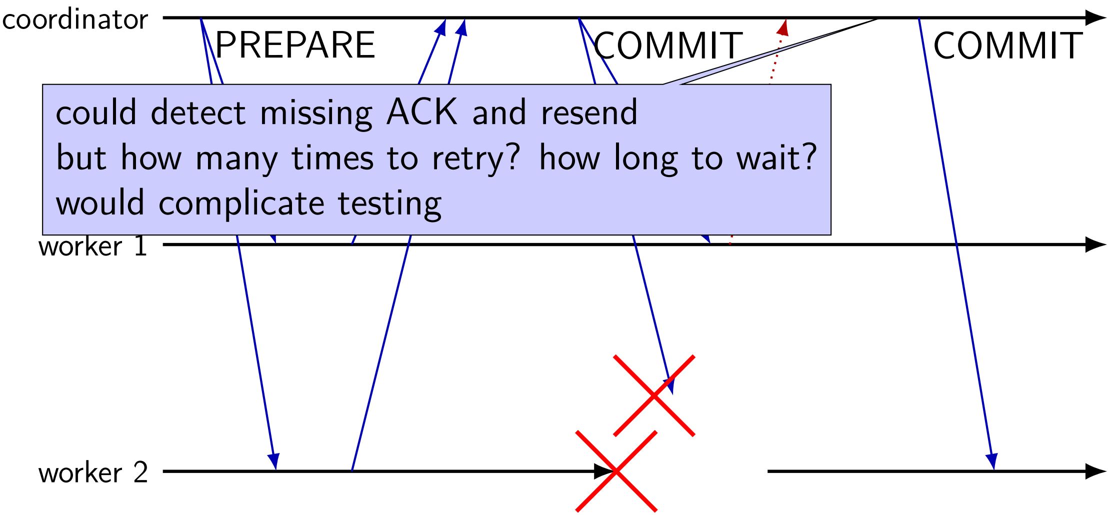
- option 1: coordinator resends outcome somehow?
  - requires acknowledgements from worker
  - required for assignment
- option 2: worker resends vote (coordinator resends outcome)

NB: coordinator cannot give up

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

twophase assignment recovery

on failure: we'll restart everything that failed

"crash-oriented computing": simplifies implementation

- you need to handle everything crashing anyways...
- so just make that the only way you handle errors

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"
...

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"
...

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"
...

user IDs

most common way OSes identify what *domain* process belongs to:

(unspecified for now) procedure sets user IDs

every process has a user ID

user ID used to decide what process is authorized to do

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

POSIX user IDs

```
uid_t geteuid(); // get current process's "effective" user ID
```

process's user identified with unique number

kernel typically only knows about number

effective user ID is used for all permission checks

also some other user IDs — we'll talk later

standard programs/library maintain number to name mapping

- `/etc/passwd` on typical single-user systems
- network database on department machines

POSIX groups

```
gid_t getegid(void);
    // process's"effective" group ID

int getgroups(int size, gid_t list[]);
    // process's extra group IDs
```

POSIX also has *group IDs*

like user IDs: kernel only knows numbers

- standard library+databases for mapping to names

also process has some other group IDs — we'll talk later

id

```
cr4bd@power4
: /net/zf14/cr4bd ; id
uid=858182(cr4bd) gid=21(csfaculty)
        groups=21(csfaculty),325(instructors),90027(cs4414)
```

id command displays uid, gid, group list

names looked up in database
- kernel doesn't know about this database
- code in the C standard library

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

groups that don't correspond to users

example: video group for access to monitor

put process in video group when logged in directly

don't do it when SSH'd in

...but: user can keep program running with video group
in the background after logout?

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 control matrix

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

POSIX file permissions

POSIX files have a very restricted access control list

one user ID + read/write/execute bits for user

"owner" — also can change permissions

one group ID + read/write/execute bits for group

default setting — read/write/execute

(see docs for chmod command)

POSIX/NTFS ACLs

more flexible access control lists

list of (user or group, read or write or execute or …)

supported by NTFS (Windows)

a version standardized by POSIX, but usually not supported

POSIX ACL syntax

```
# group students have read+execute permissions
group:students:r-x
# group faculty has read/write/execute permissions
group:faculty:rwx
# user mst3k has read/write/execute permissions
user:mst3k:rwx
# user tj1a has no permissions
user:tj1a:---

# POSIX acl rule:
    # user take precedence over group entries
```

authorization checking on Unix

checked on system call entry

- no relying on libraries, etc. to do checks

files (open, rename, …) — file/directory permissions

processes (kill, …) — process UID = user UID

…

keeping permissions?

which of the following would still be secure?

A. setting up a read-only page table entry that allows a process to directly access its user ID from its process control block in user mode

B. performing authorization checks in the standard library in addition to system call handlers

C. performing authorization checks in the standard library instead of system call handlers

D. making the user ID a system call argument rather than storing it in the process control block

superuser

user ID `0` is special

superuser or *root*

some system calls: only work for uid `0`

- shutdown, mount new file systems, etc.

automatically passes all (or almost all) permission checks

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

Unix password storage

typical single-user system: `/etc/shadow`

- only readable by root/superuser

department machines: network service

- Kerberos / Active Directory:
- server takes (encrypted) passwords
- server gives tokens: “yes, really this user”
- can cryptographically verify tokens come from server

aside: beyond passwords

/bin/login entirely user-space code

only thing special about it: when it's run

could use any criteria to decide, not just passwords

- physical tokens
- biometrics
- …

how does login work?

```
somemachine login: jo
password: ********

jo@somemachine$ ls
...
```

this is a program which...

checks if the password is correct, and

changes user IDs, and

runs a shell

changing user IDs

```
int setuid(uid_t uid);
```

if superuser: sets effective user ID to arbitrary value

- and a "real user ID" and a "saved set-user-ID" (we'll talk later)

system starts in/login programs run as superuser

- voluntarily restrict own access before running shell, etc.

sudo

```
tj1a@somemachine$ sudo restart
Password: *********
```

sudo: run command with superuser permissions

started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec()` syscall changes effective user ID to owner's ID

`sudo` program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`

set-user ID gates

set-user ID program: gate to higher privilege

controlled access to extra functionality

make authorization/authentication decisions *outside the kernel*

way to allow normal users to do *one thing that needs privileges*

- write program that does that one thing — nothing else!
- make it owned by user that can do it (e.g. root)
- mark it set-user-ID

want to allow only some user to do the thing

- make program check which user ran it

uses for setuid programs

mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

write to secure log file

- setuid program ensures that log is append-only for normal users

bind to a particular port number < 1024

- setuid program creates socket, then becomes not root

set-user-ID program v syscalls

hardware decision: some things only for kernel

system calls: *controlled* access to things kernel can do

decision about how can do it: in the kernel

kernel decision: some things only for root (or other user)

set-user-ID programs: controlled access to things root/... can do

decision about how can do it: made by root/...

backup slides

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

extending voting

two-phase commit: unanimous vote to commit

assumption: data split across nodes, every must cooperate

other model: every node has a copy of data

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

just require "enough" nodes to be working

for now — assume fail-stop

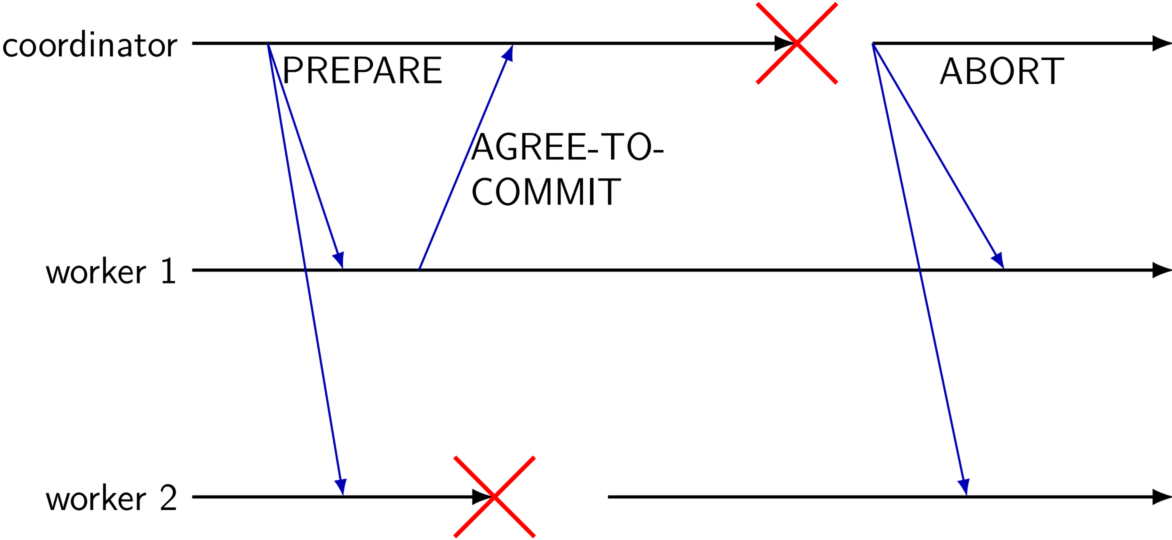
nodes don't respond or tell you if broken

assignment: failure types

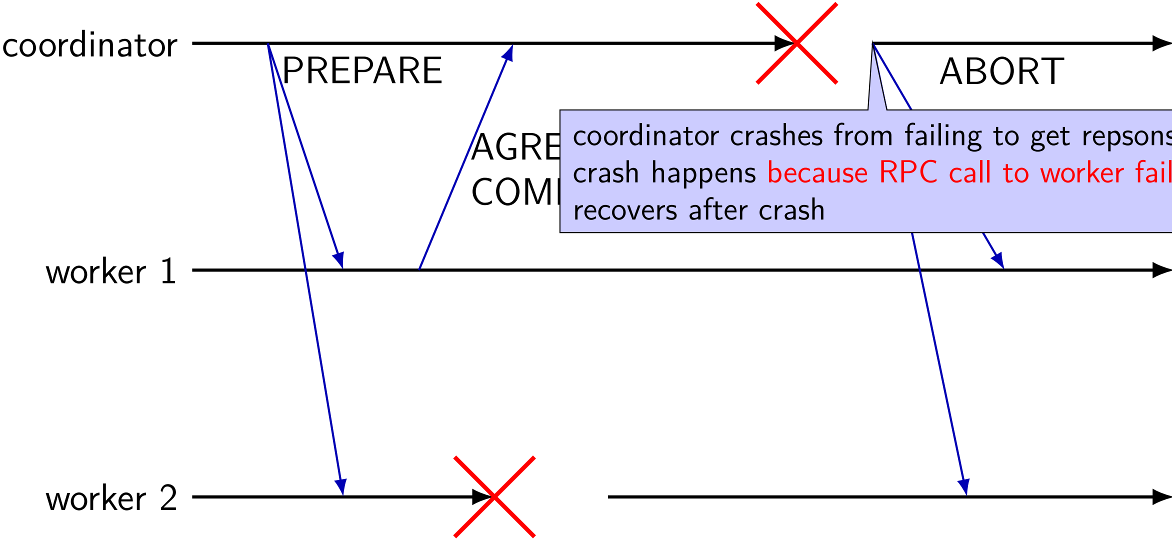
send RPC and

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

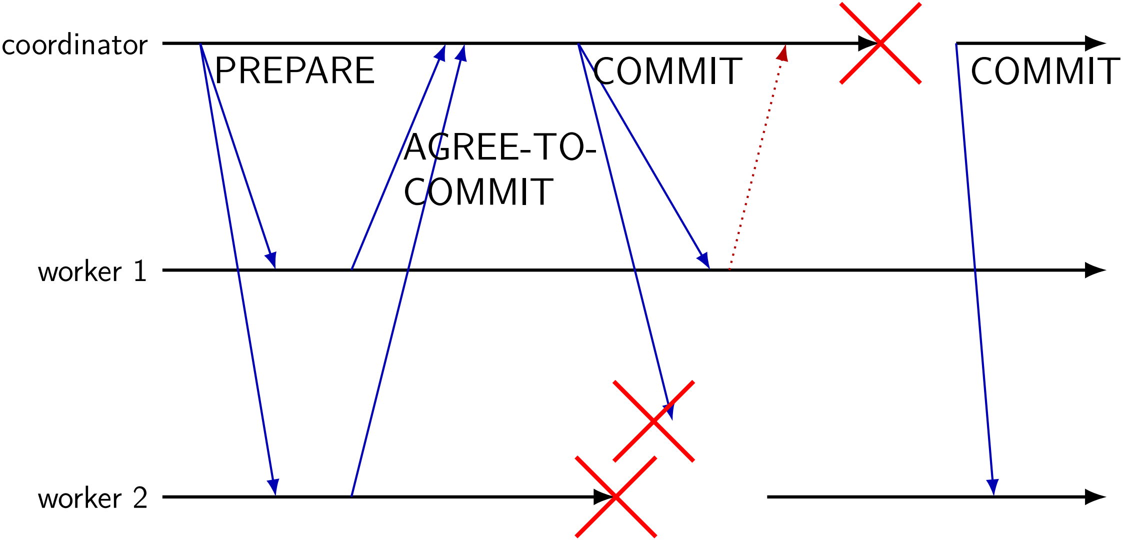
assignment: fails during prepare



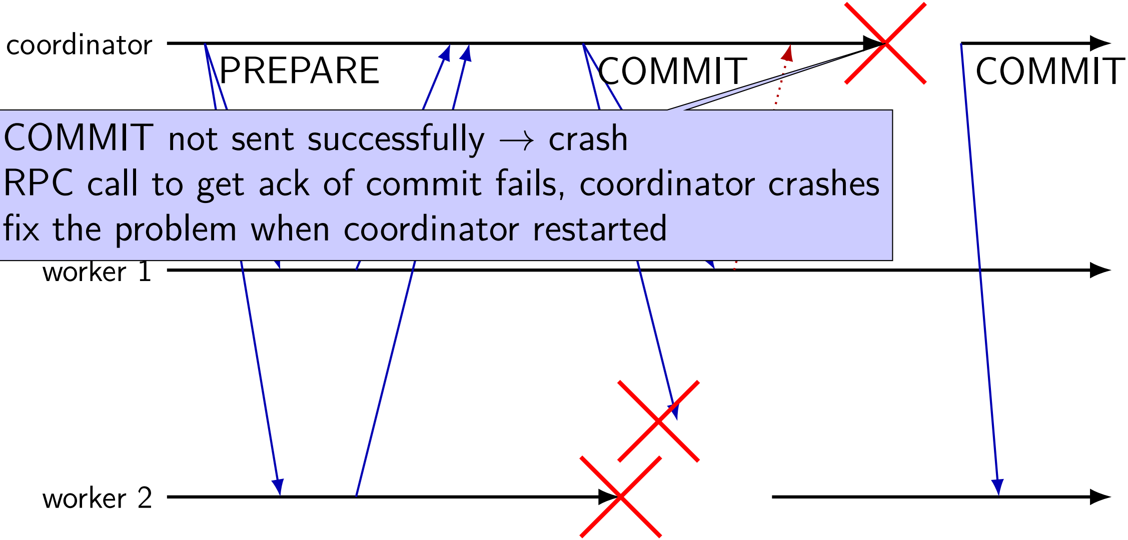
assignment: fails during prepare



assignment: failuring during commit

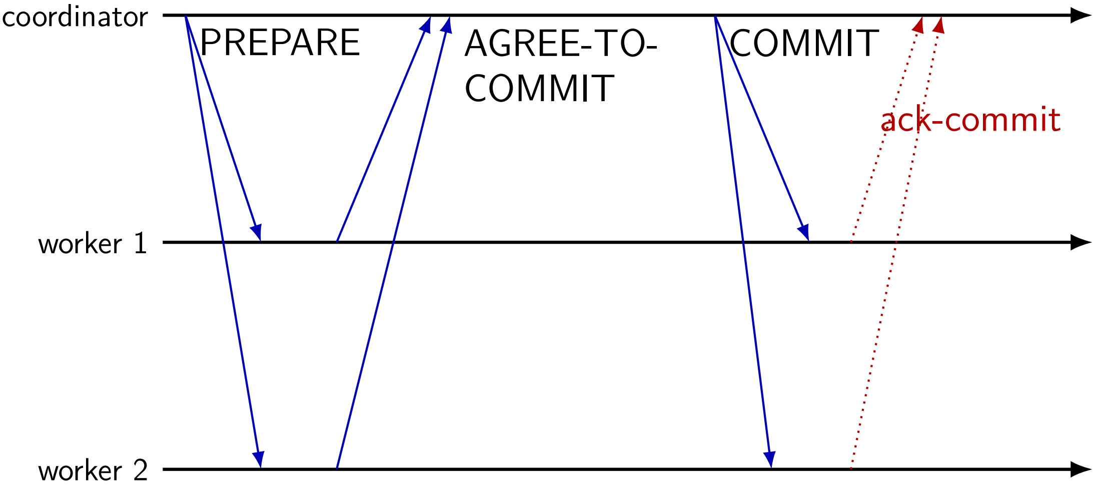


assignment: failuring during commit

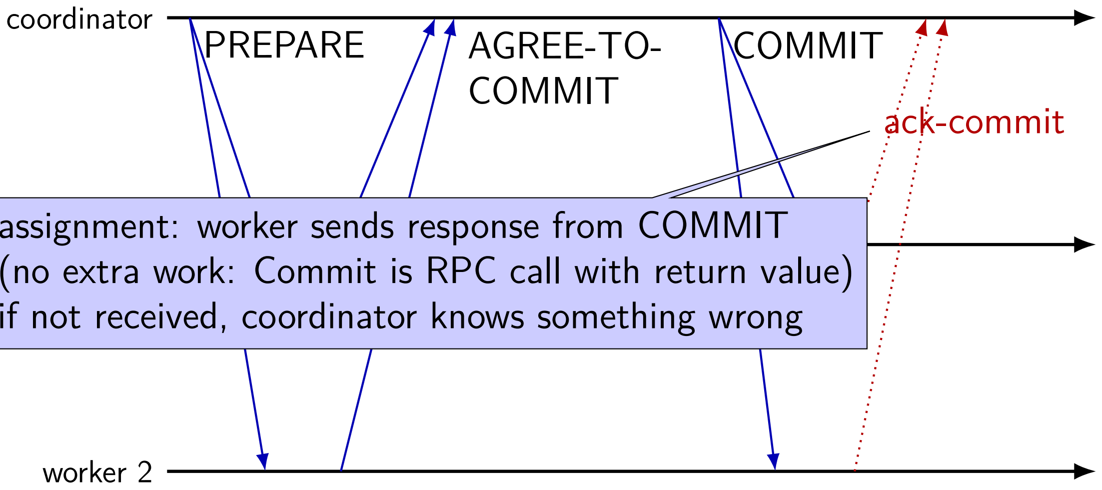


COMMIT not sent successfully → crash
RPC call to get ack of commit fails, coordinator crashes
fix the problem when coordinator restarted

aside: worker ACKs

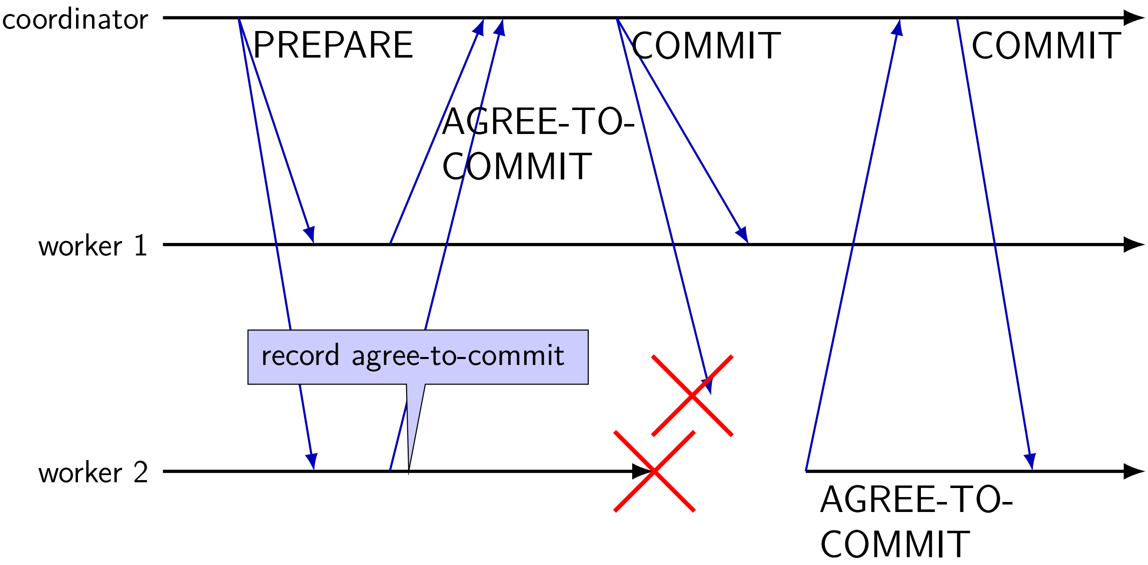


aside: worker ACKs

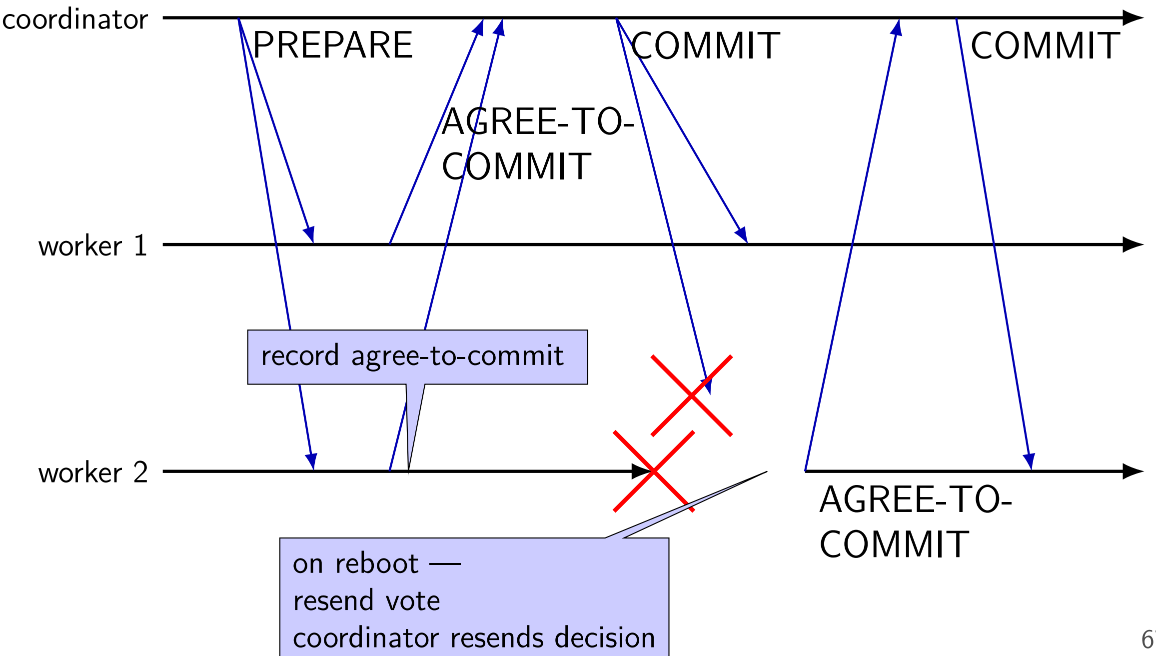


assignment: worker sends response from COMMIT
(no extra work: Commit is RPC call with return value)
if not received, coordinator knows something wrong

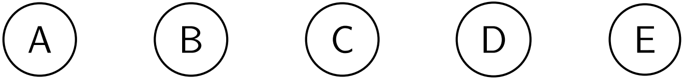
TPC: worker revoting



TPC: worker revoting



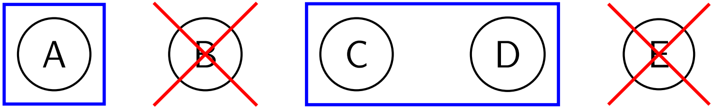
quorums (1)



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

any quorum enough — okay if some nodes fail

quorums (1)



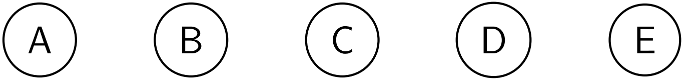
perform read/write with vote of any *quorum* of nodes

any quorum enough — okay if some nodes fail

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

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

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

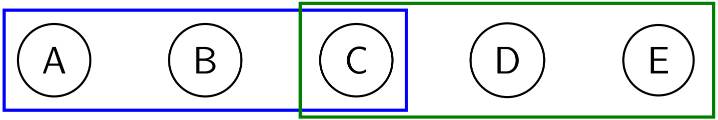
e.g. every operation requires majority of nodes

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

make sure updates survive later on

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

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

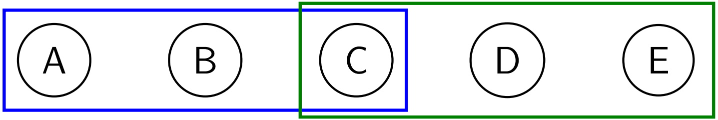
e.g. every operation requires majority of nodes

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

make sure updates survive later on

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

quorums (2)



requirement: quorums overlap

overlap = *someone in quorum* knows about every update

e.g. every operation requires majority of nodes

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

make sure updates survive later on

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