last time

distributed transaction idea:

- update across machines happens together or not
- hide "half-done" state, even if failures

two-phase commit

- coordinator collects *votes* from workers, makes decision
- coordinator only decides to commit if all votes say to commit
- if pending transaction, workers don't allow conflicting operations
- redo logging: write log before sending any message, resend on crash

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

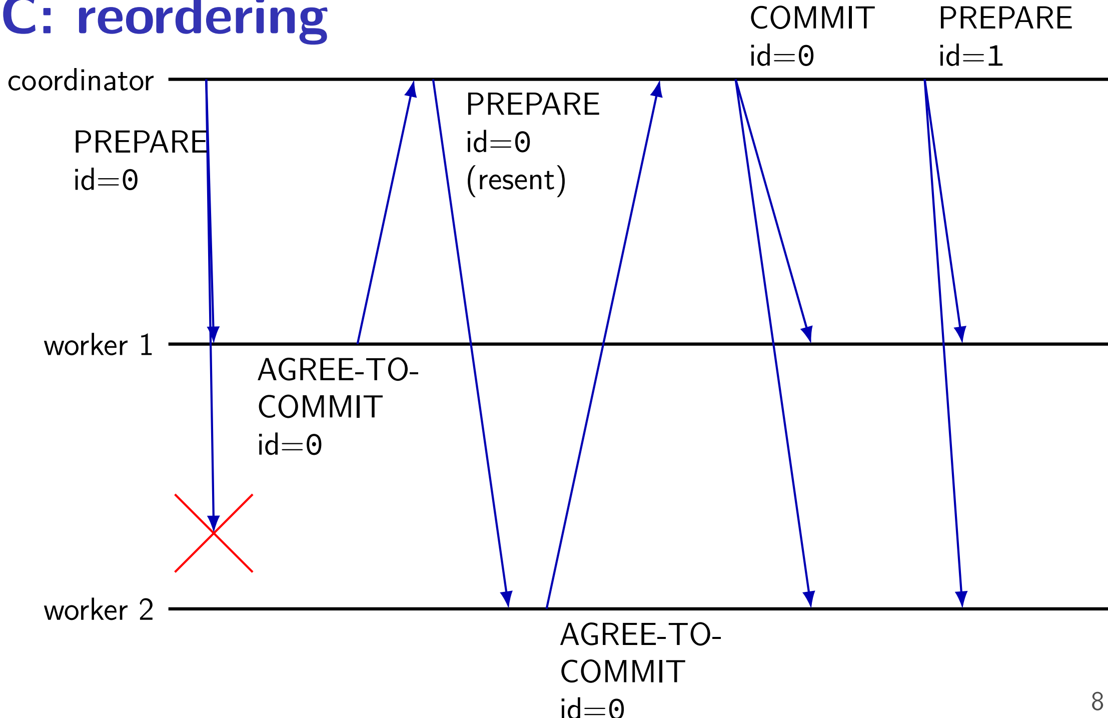
- it gets lost
- it gets sent, but acknowledgment/reply is lost
- it gets sent, but delayed until after another RPC

assignment: failure types

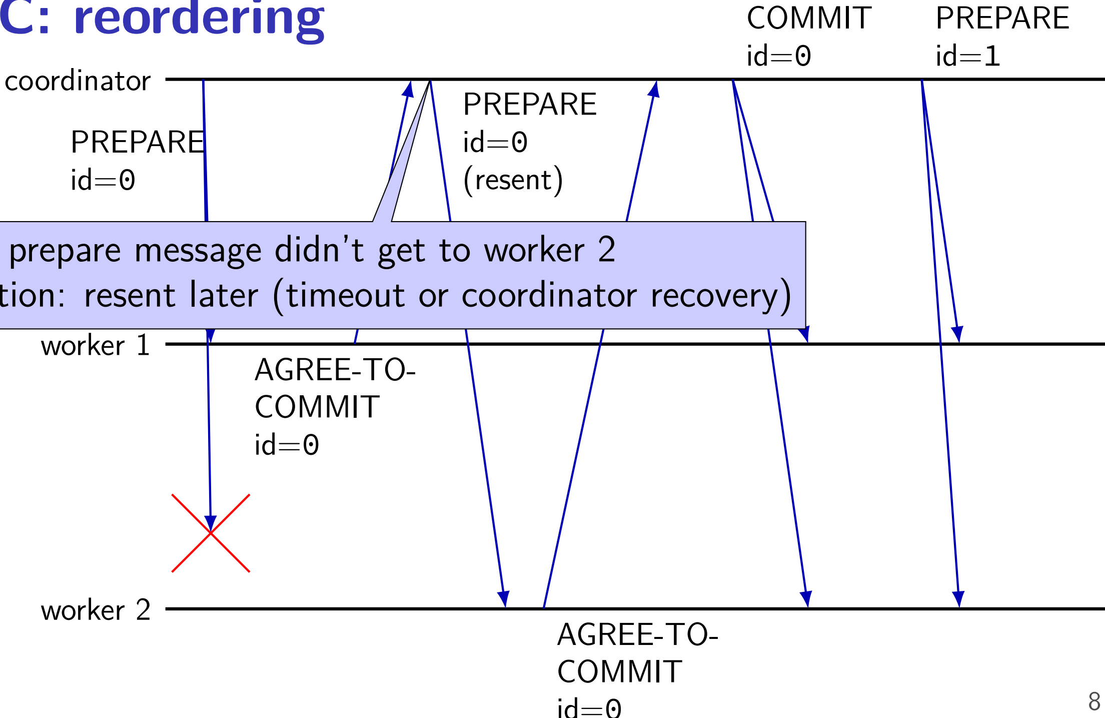
send RPC and

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

TPC: reordering

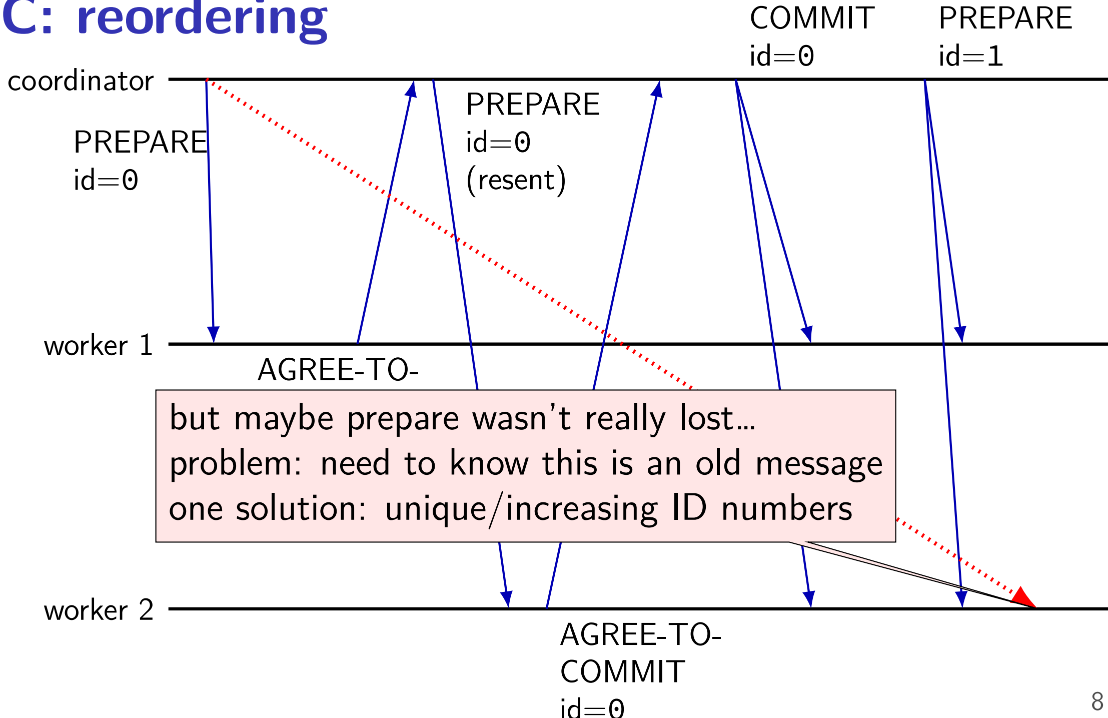


TPC: reordering



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?

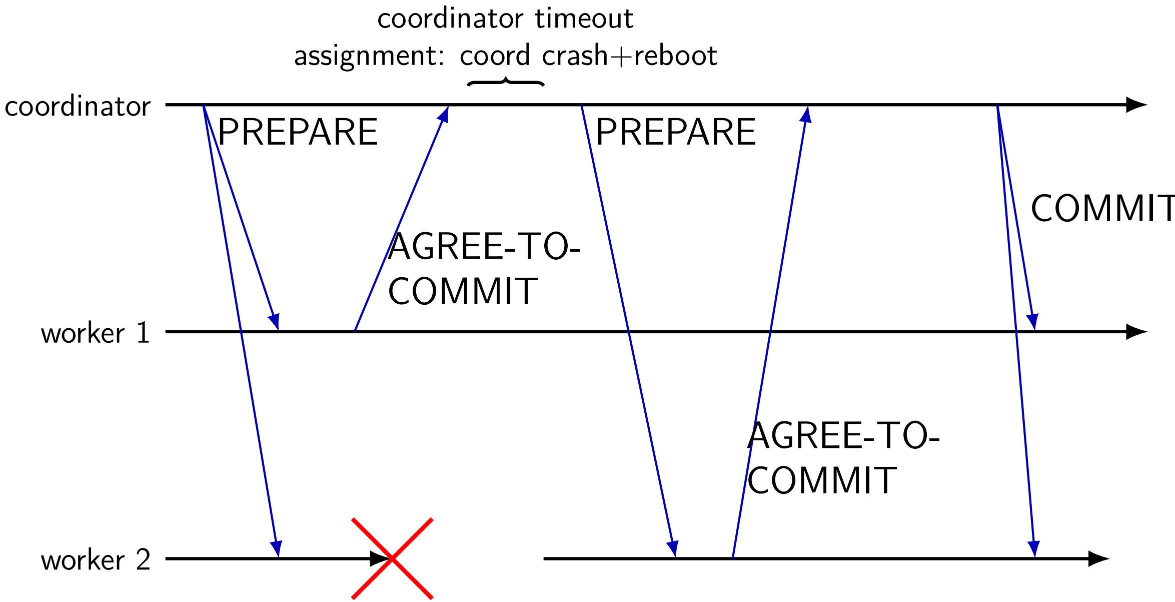
- option 1: coordinator retries prepare
- option 2: coordinator gives up, sends abort
- option 3: worker resends vote proactively

worker failure during prepare

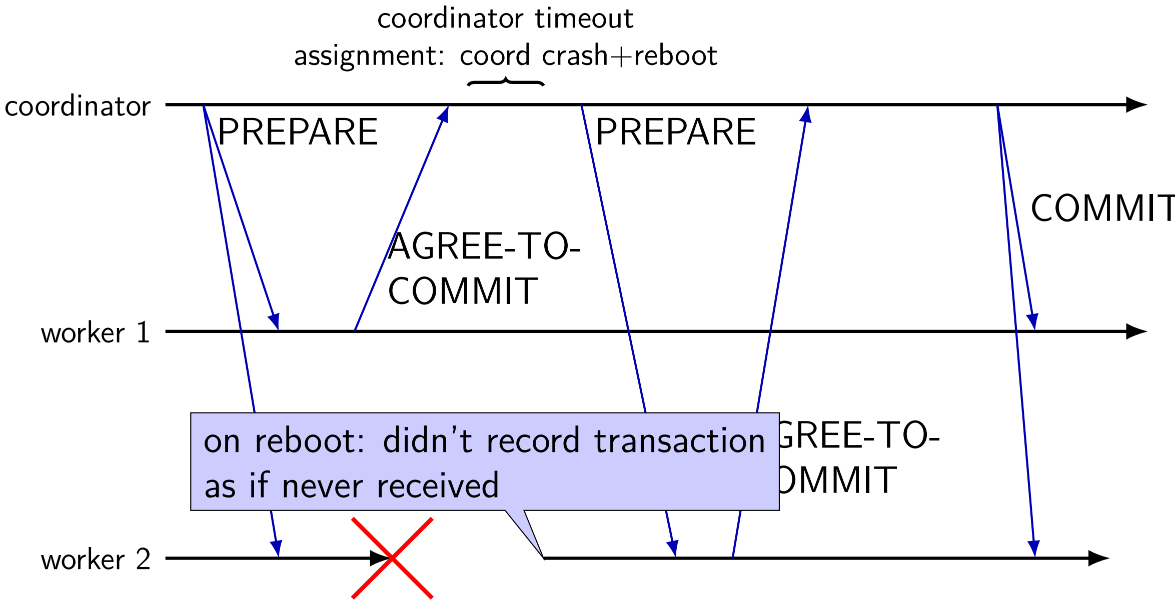
worker failure after prepare without sending vote?

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

TPC: worker fails after prepare (1a)

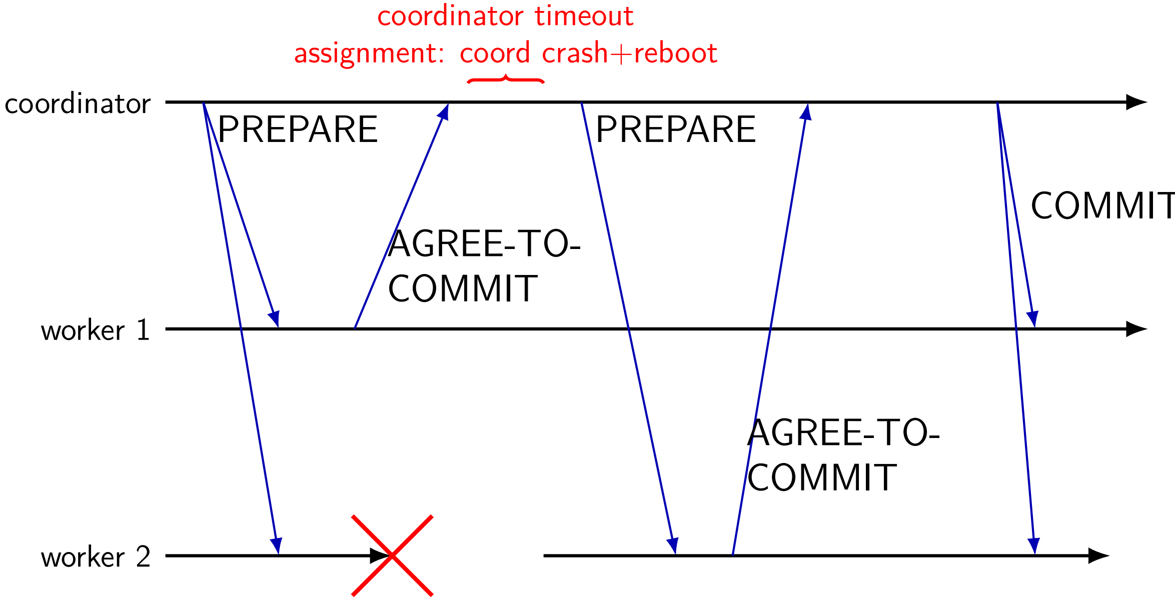


TPC: worker fails after prepare (1a)

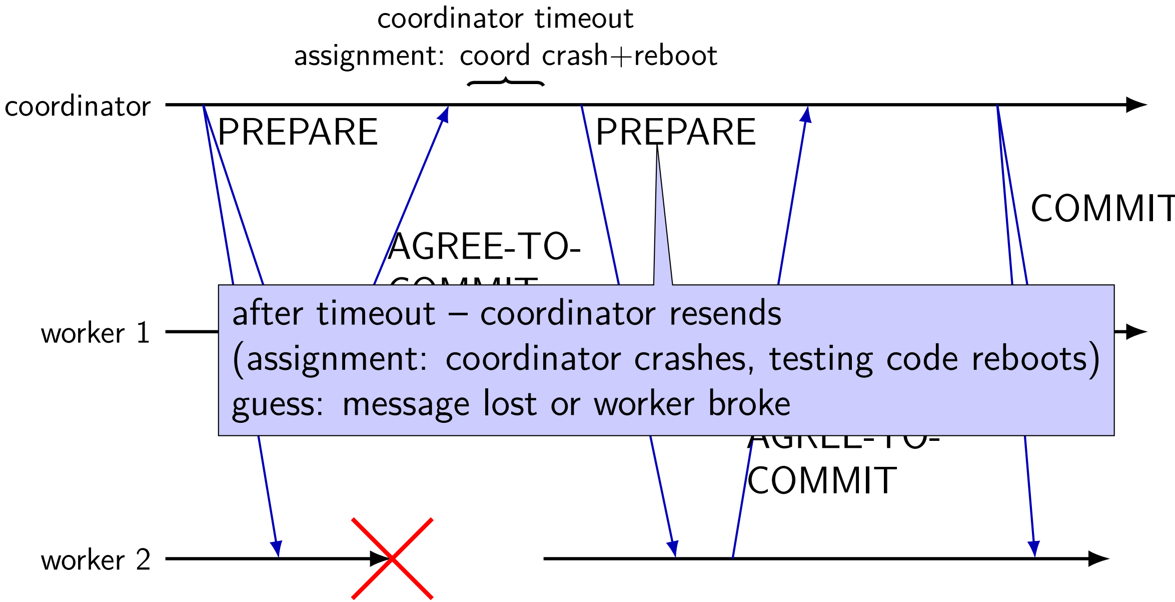


on reboot: didn't record transaction as if never received

TPC: worker fails after prepare (1a)

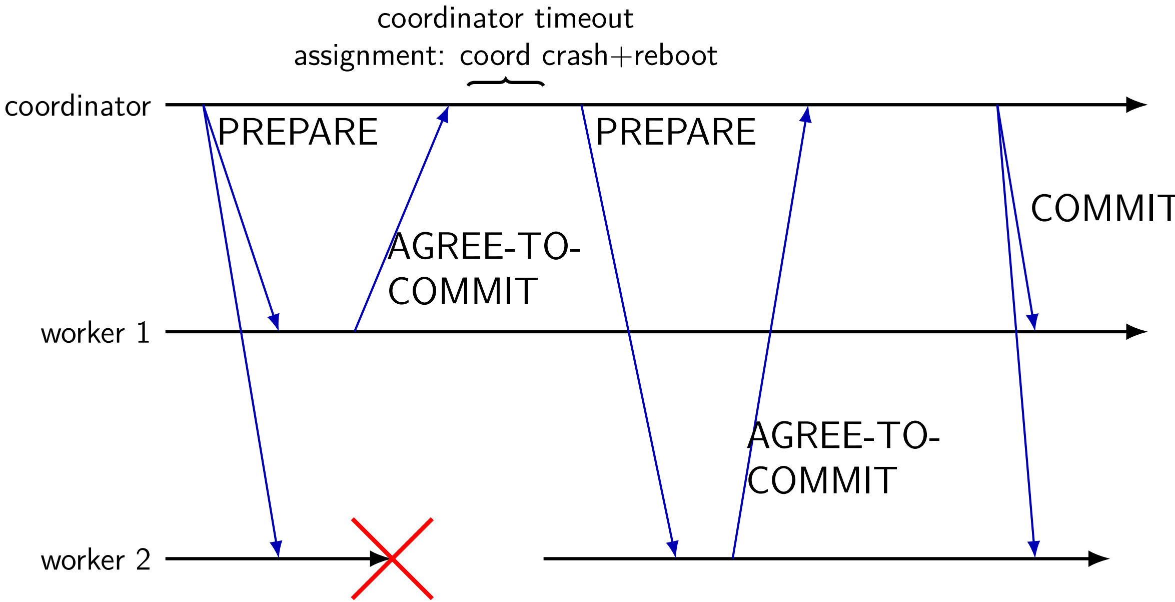


TPC: worker fails after prepare (1a)

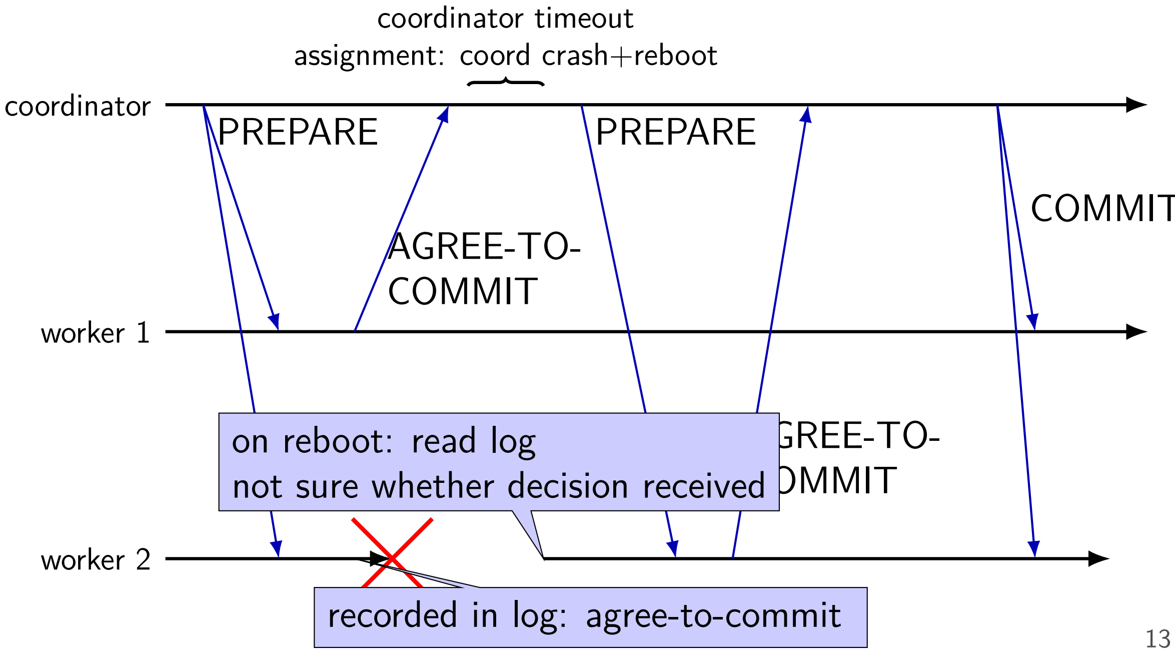


after timeout – coordinator resends
(assignment: coordinator crashes, testing code reboots)
guess: message lost or worker broke

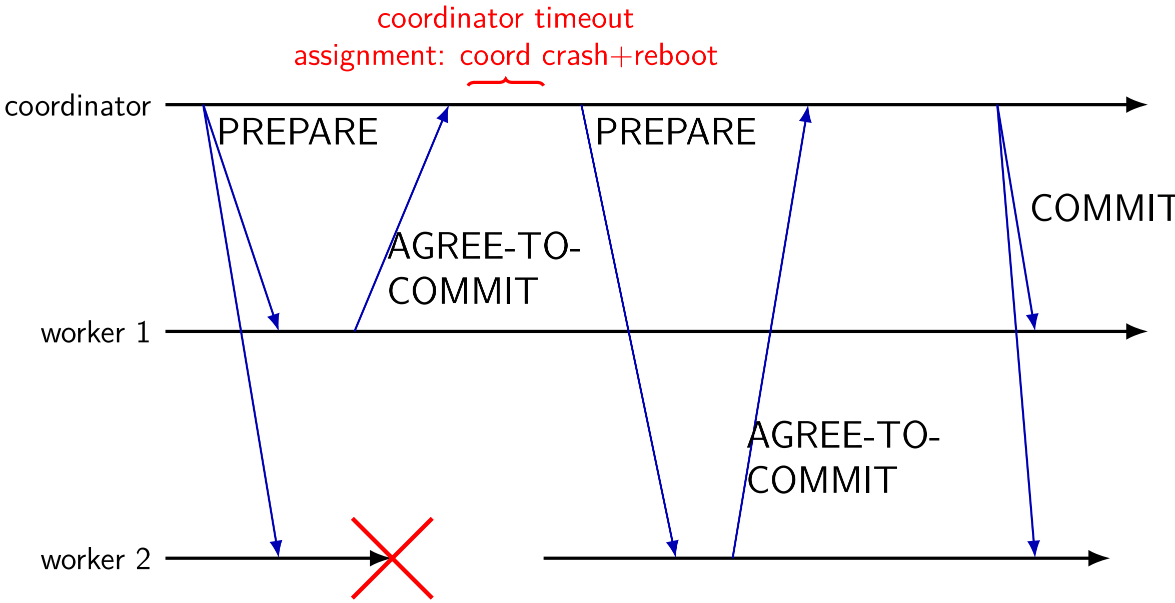
TPC: worker fails after prepare (1b)



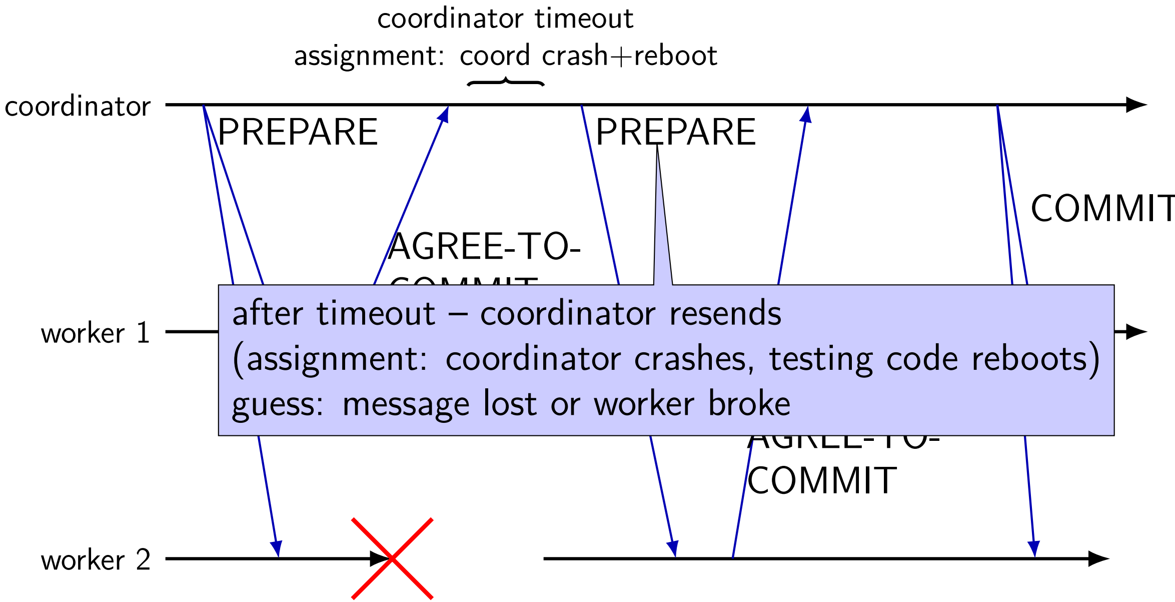
TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)



TPC: worker fails after prepare (1b)



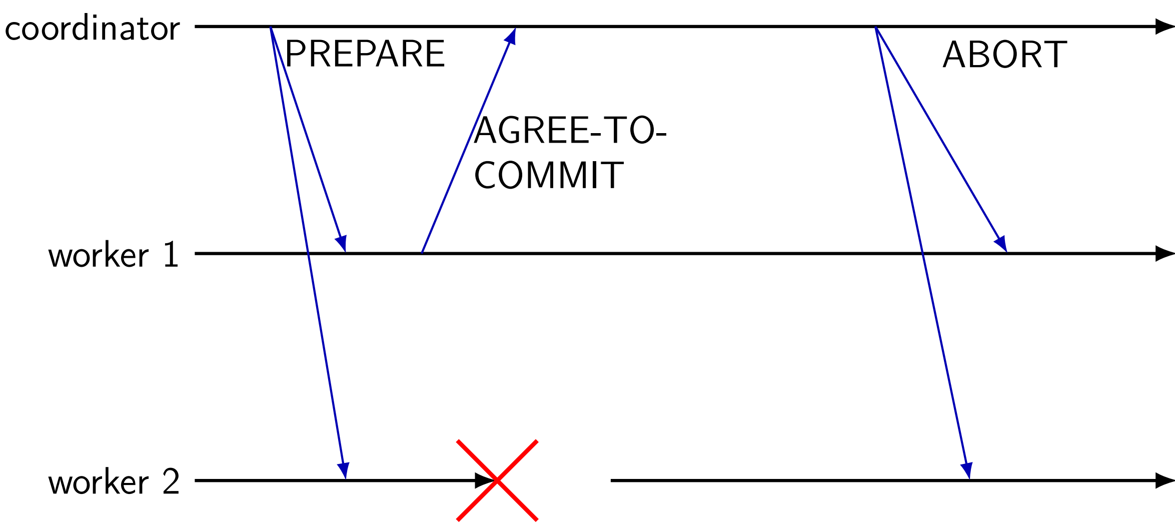
after timeout – coordinator resends
(assignment: coordinator crashes, testing code reboots)
guess: message lost or worker broke

worker failure during prepare

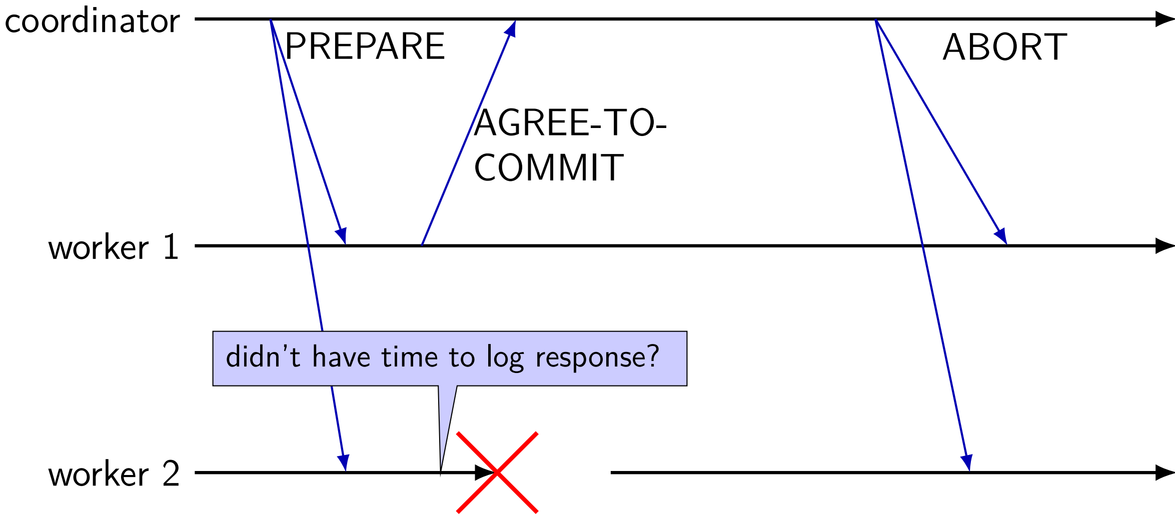
worker failure after prepare without sending vote?

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

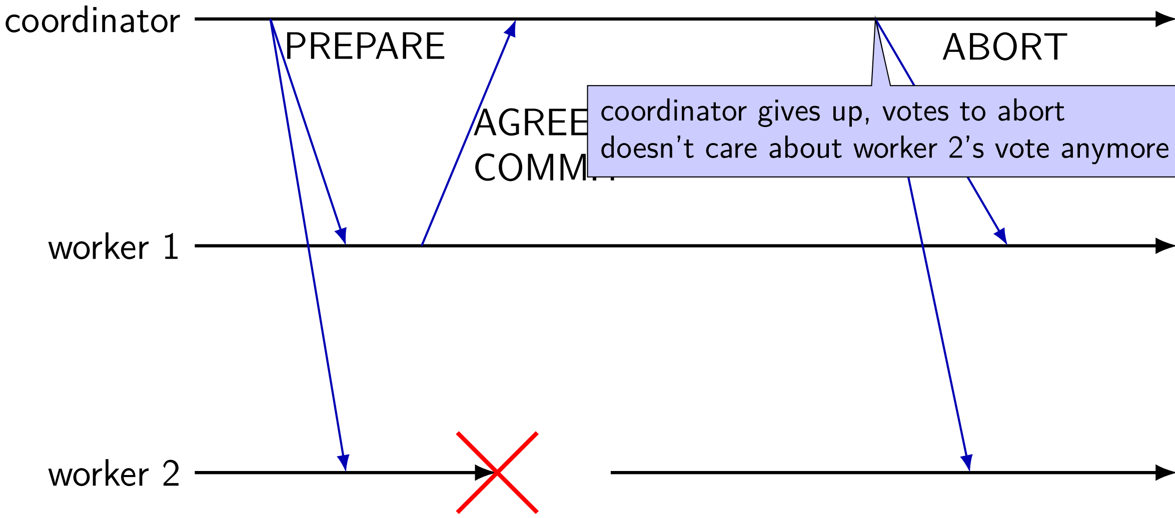
TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)



TPC: worker fails after prepare (2)

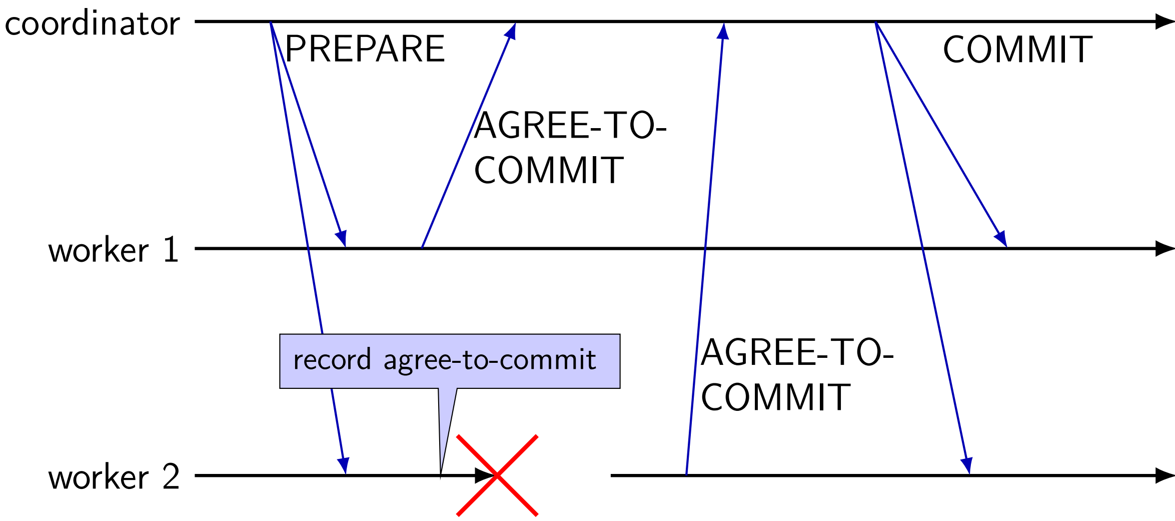

coordinator
worker 1
worker 2
PREPARE
AGREE
COMMIT
ABORT
coordinator gives up, votes to abort
doesn't care about worker 2's vote anymore

worker failure during prepare

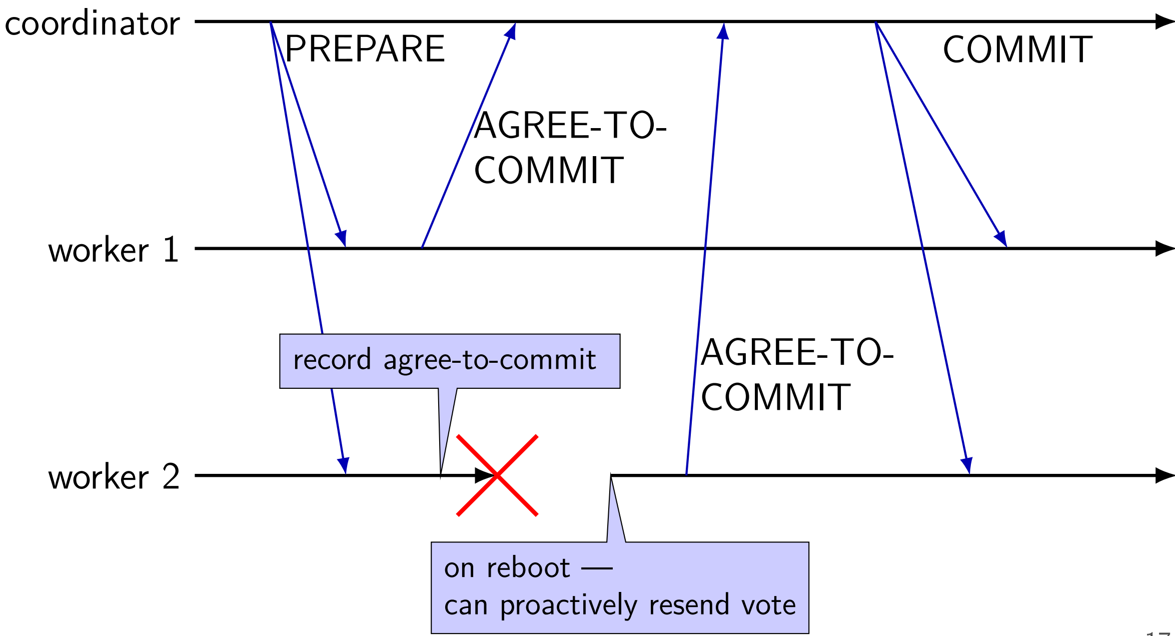
worker failure after prepare without sending vote?

- 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)



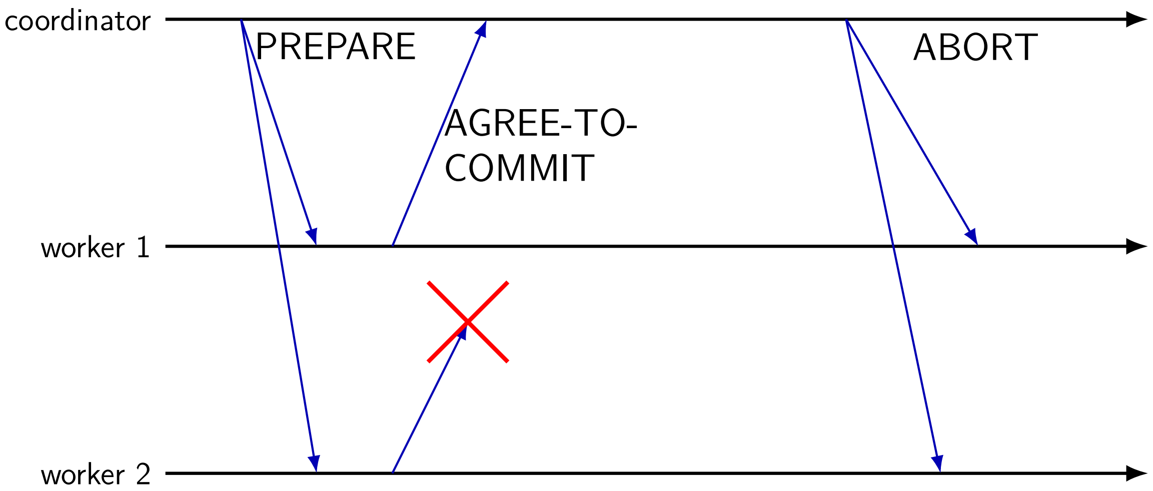
network failure after during voting?

network failure during voting ≈ 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

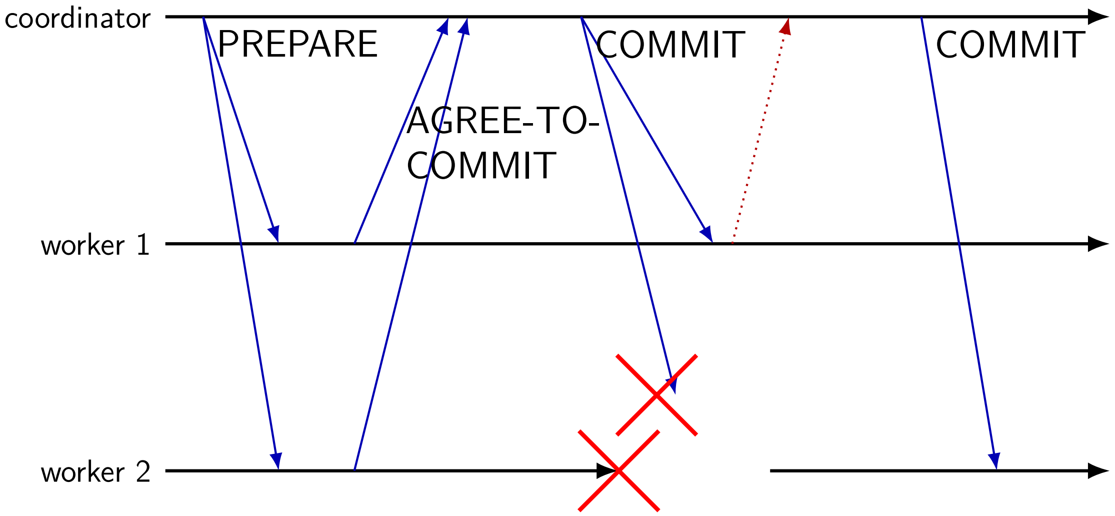
worker failure during commit

worker failure during commit?

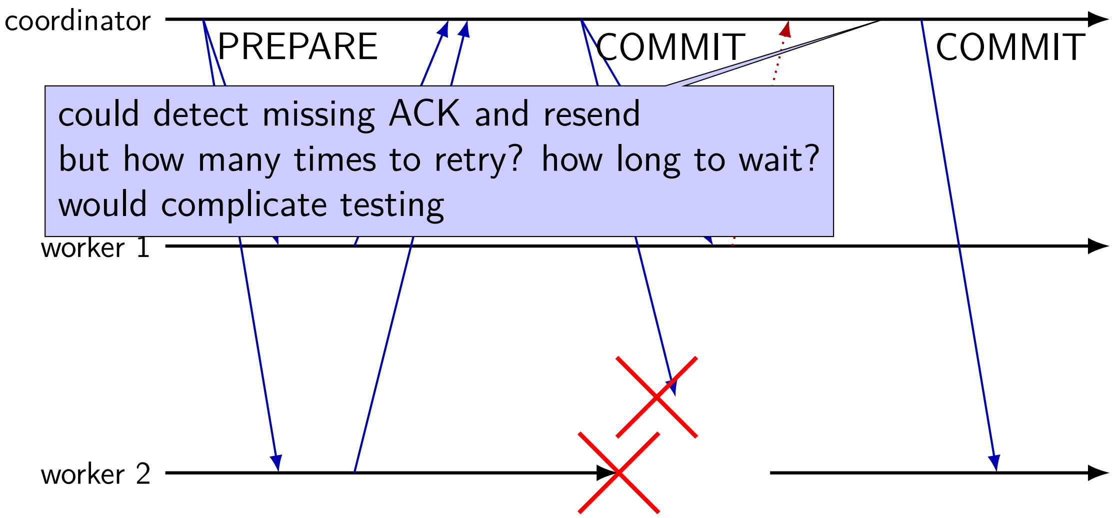
- option 1: coordinator resends outcome somehow?
  - requires acknowledgements from worker
  - required for assignment
- option 2: worker resends vote (coordinator resends outcome)

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

logistical note: due date

due Thurs 6 May, not Weds

twophase Q and A

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/...

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

one (bad?) idea: setuid program to read grade for assignment

```
./print_grade assignment
```

outputs grade from `all-grades/assignment/USER.txt`

a very broken setuid program

print_grade.c:

```
int main(int argc, char **argv) {
    char filename[500];
    sprintf(filename, "all-grades/%s/%s.txt",
            argv[1], getenv("USER"));
    int fd = open(filename, O_RDWR);
    char buffer[1024];
    read(fd, buffer, 1024);
    printf("%s: %s\n", argv[1], buffer);
}
```

HUGE amount of stuff can go wrong

examples?

set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

…

a delegation problem

consider printing program marked setuid to access printer

- decision: no accessing printer directly
- printing program enforces page limits, etc.

command line: file to print

can printing program just call open()?

a broken solution

```
if (original user can read file from argument) {
    open(file from argument);
    read contents of file;
    write contents of file to printer
    close(file from argument);
}
```

hope: this prevents users from printing files than can't read

problem: race condition!

a broken solution / why

setuid program	other user program
	create normal file `toprint.txt`
check: can user access? (yes)	—
	`unlink("toprint.txt")`
	`link("/secret", "toprint.txt"`
`open("toprint.txt")`	—
read ...	—

link: create new directory entry for file

- another option: rename, symlink ("symbolic link" — alias for file/directory)
- another possibility: run a program that creates secret file (e.g. temporary file used by password-changing program)

time-to-check-to-time-of-use vulnerability

TOCTTOU solution

temporarily ‘become’ original user

then open

then turn back into set-uid user

this is why POSIX processes have multiple user IDs

can swap out effective user ID temporarily

practical TOCTTOU races?

can use symlinks *maze* to make check slower

> symlink `toprint.txt` → `a/b/c/d/e/f/g/normal.txt`
> symlink `a/b` → `../a`
> symlink `a/c` → `../a`
> …

lots of time spent following symbolic links when program opening toprint.txt

gives more time to sneak in unlink/link or (more likely) rename

exercise

which (if any) of the following would fix for a TOCTTOU vulnerability in our setuid printing application? (assume the Unix-permissions without ACLs are in use)

[A] **both before and after** opening the path passed in for reading, check that the path is accessible to the user who ran our application

[B] after opening the path passed in for reading, using `fstat` with the file descriptor opened to check the permissions on the file

[C] before opening the path, verify that the user controls the file referred to by the path **and** the directory containing it

some security tasks (1)

helping students collaborate in ad-hoc small groups on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (2)

letting students assignment files to faculty on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (3)

running untrusted game program from Internet?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

ambient authority

POSIX permissions based on user/group IDs process has

- correct user/group ID — can read file
- correct user ID — can kill process

permission information "on the side"

- separate from how to identify file/process

sometimes called *ambient authority*

"there's authorization in the air..."

alternate approach: ability to address = permission to access

capabilities

token to identify = permission to access

(typically *opaque* token)

capabilities

token to identify = permission to access

(typically *opaque* token)

pro: “what object is this token” check = “can access” check:

- simpler?

some capability list examples

file descriptors

- list of open files process has access to

page table (sort of?)

- list of physical pages process is allowed to access

some capability list examples

file descriptors

- list of open files process has access to

page table (sort of?)

- list of physical pages process is allowed to access

list of what process can access *stored with process*

handle to access object = key in permitted object table

- impossible to skip permission check!

sharing capabilities

some ways of sharing capabilities:

inherited by spawned programs

- file descriptors/page tables do this

send over local socket or pipe

- Unix: usually supported for file descriptors!
- (look up `SCM_RIGHTS` — slightly different for Linux v. OS X v. FreeBSD v. …)

Capsicum: practical capabilities for UNIX (1)

Capsicum: research project from Cambridge

adds capabilities to FreeBSD by extending file descriptors

opt-in: can set process to require capabilities to access objects

- instead of absolute path, process ID, etc.

capabilities = fds for each directory/file/process/etc.

more permissions on fds than read/write

- execute
- open files in (for fd representing directory)
- kill (for fd reporesenting process)
- ...

Capsicum: practical capabilities for UNIX (2)

capabilities = no global names

no filenames, instead fds for directories

- new syscall: `openat(directory_fd, "path/in/directory")`
- new syscall: `fexecv(file_fd, argv)`

no pids, instead fds for processes

- new syscall: `pdfork()`

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

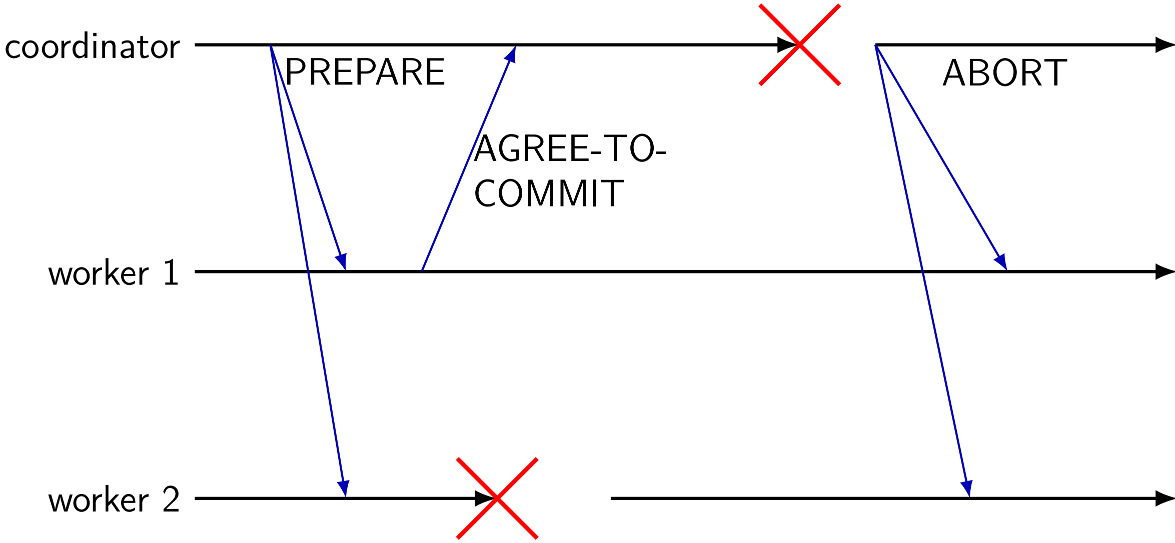
nodes don’t respond or tell you if broken

assignment: failure types

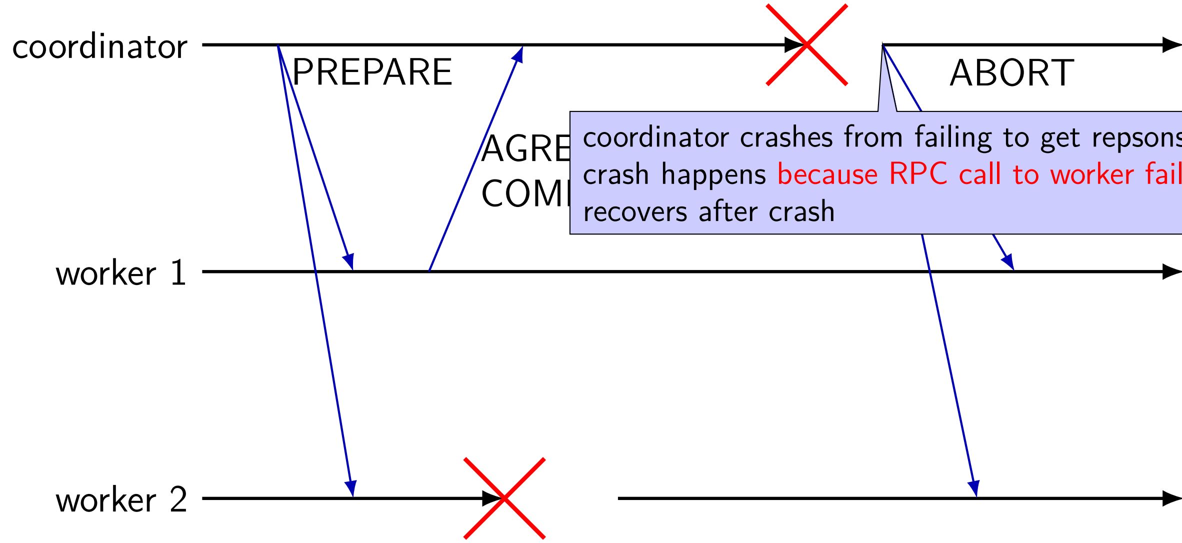
send RPC and

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

assignment: fails during prepare

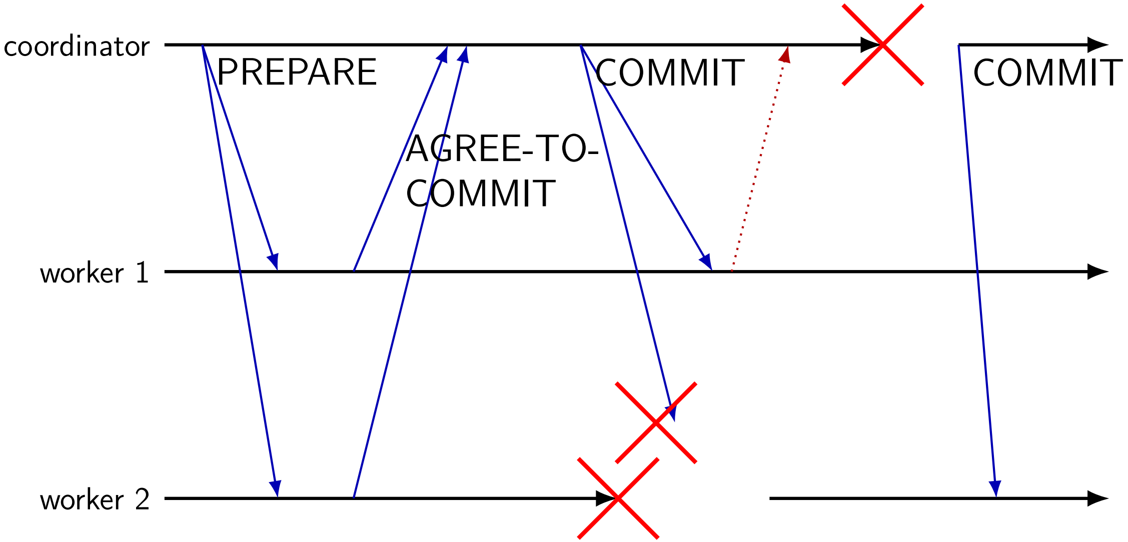


assignment: fails during prepare

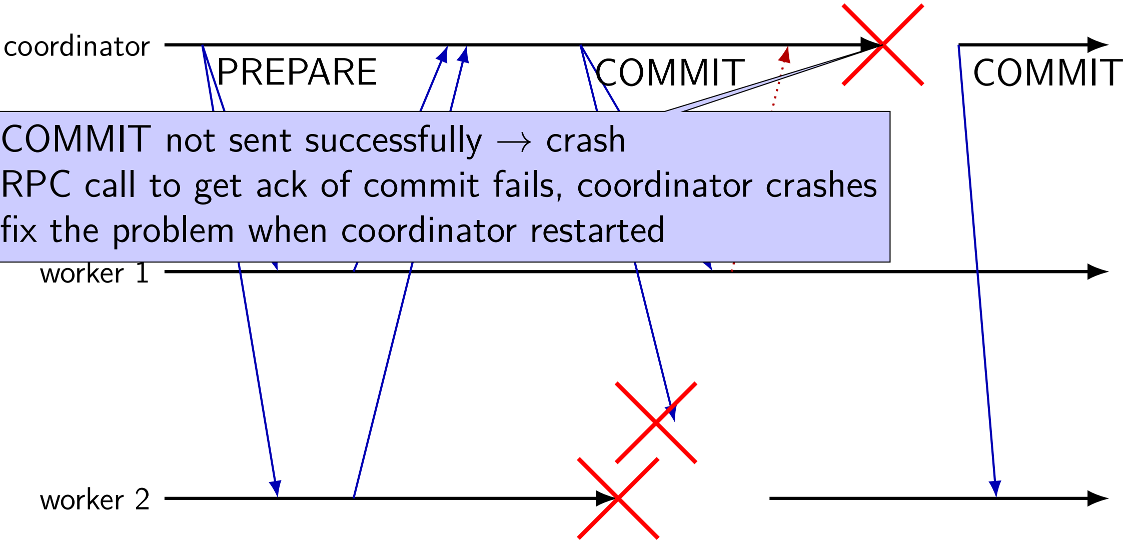


coordinator crashes from failing to get repsons
crash happens because RPC call to worker fail
recovers after crash

assignment: failuring during commit

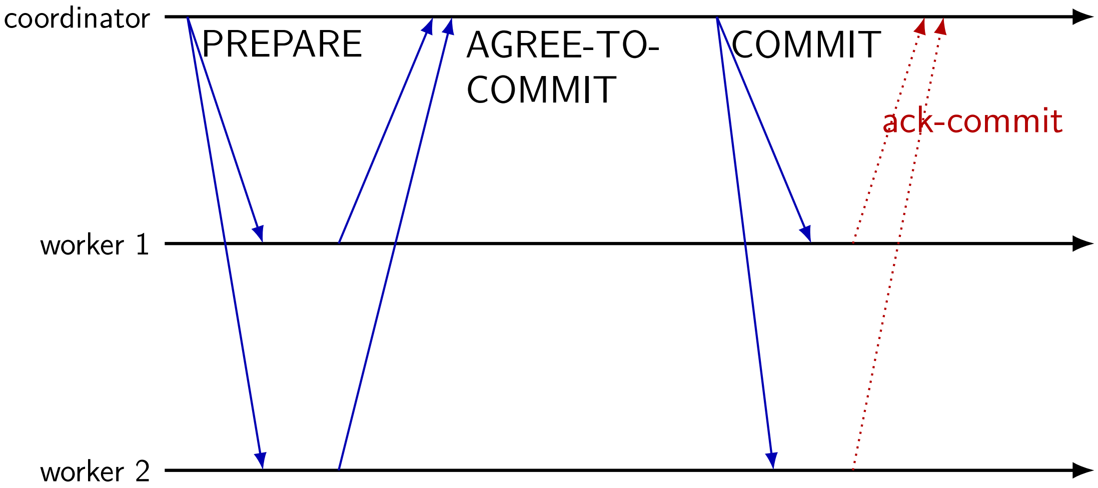


assignment: failuring during commit

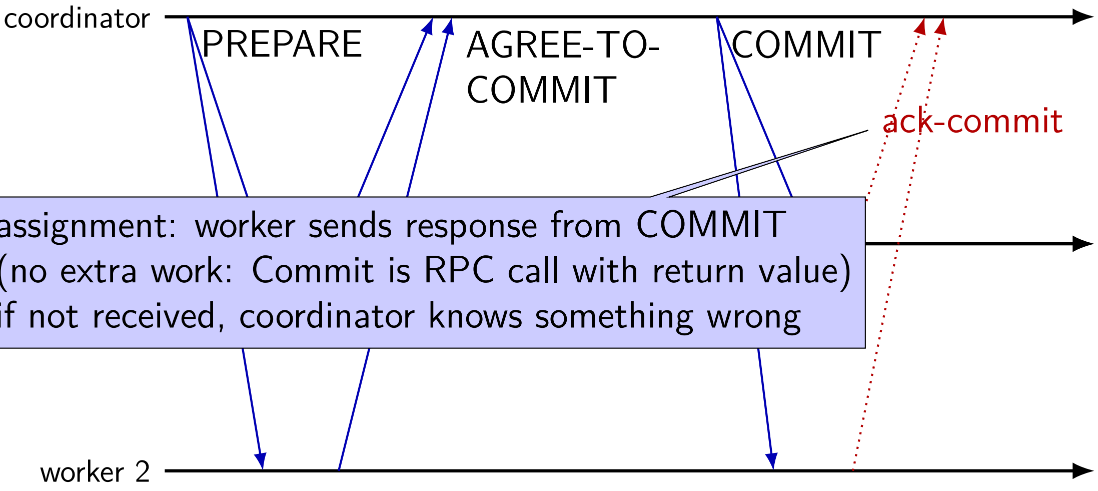


COMMIT not sent successfully → crash
RPC call to get ack of commit fails, coordinator crashes
fix the problem when coordinator restarted

aside: worker ACKs

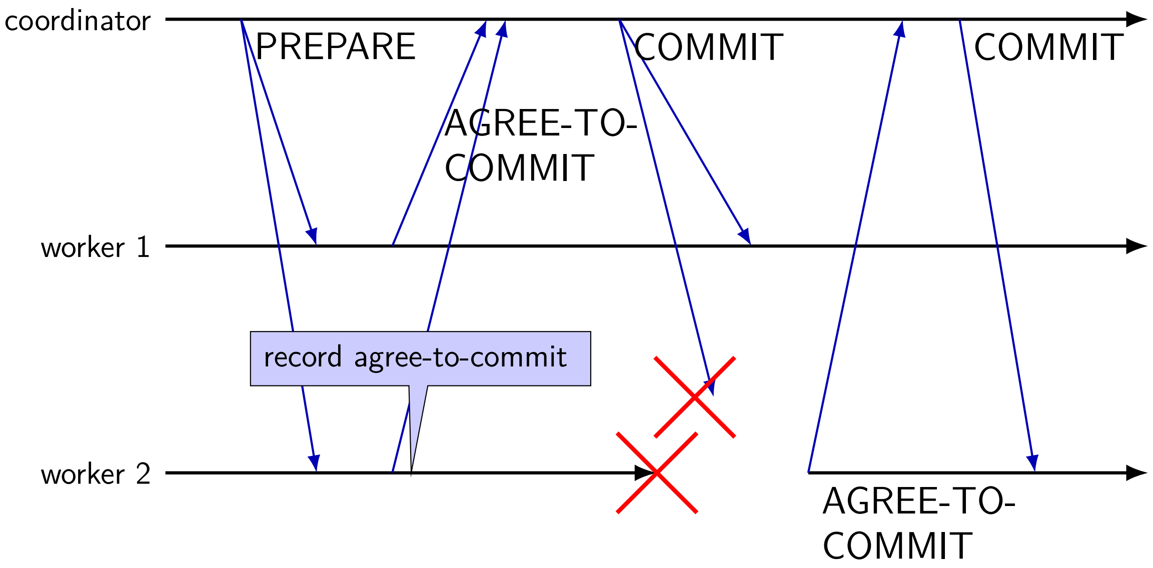


aside: worker ACKs

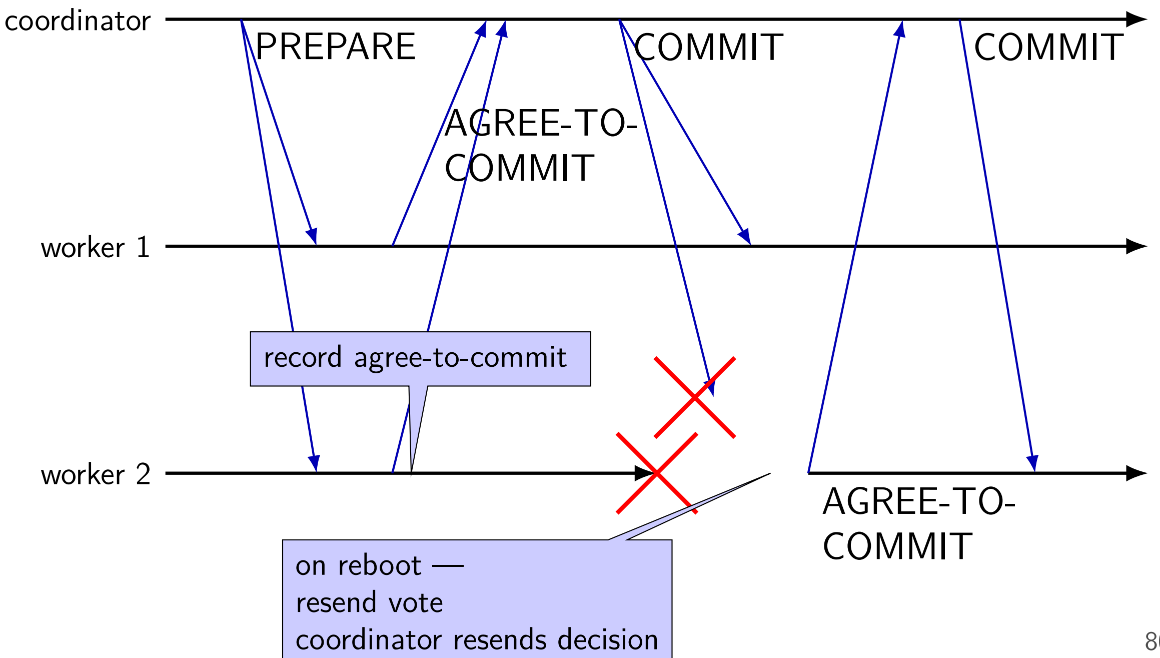


assignment: worker sends response from COMMIT
(no extra work: Commit is RPC call with return value)
if not received, coordinator knows something wrong

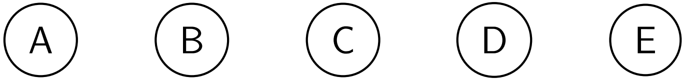
TPC: worker revoting



TPC: worker revoting



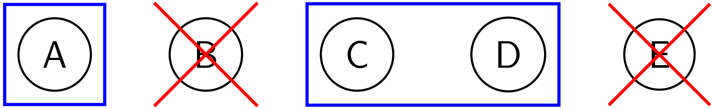
quorums (1)



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

any quorum enough — okay if some nodes fail

quorums (1)



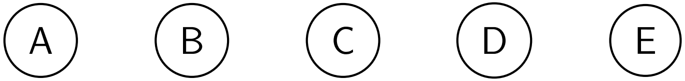
perform read/write with vote of any *quorum* of nodes

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

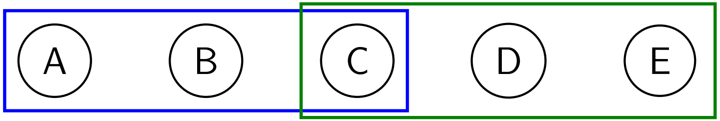
e.g. every operation requires majority of nodes

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

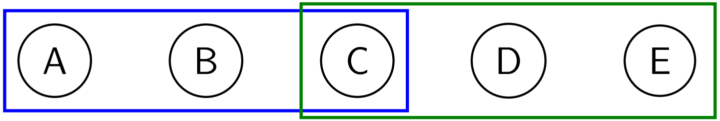
e.g. every operation requires majority of nodes

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