Fault Tolerance in Distributed Systems

Atomic Commitment

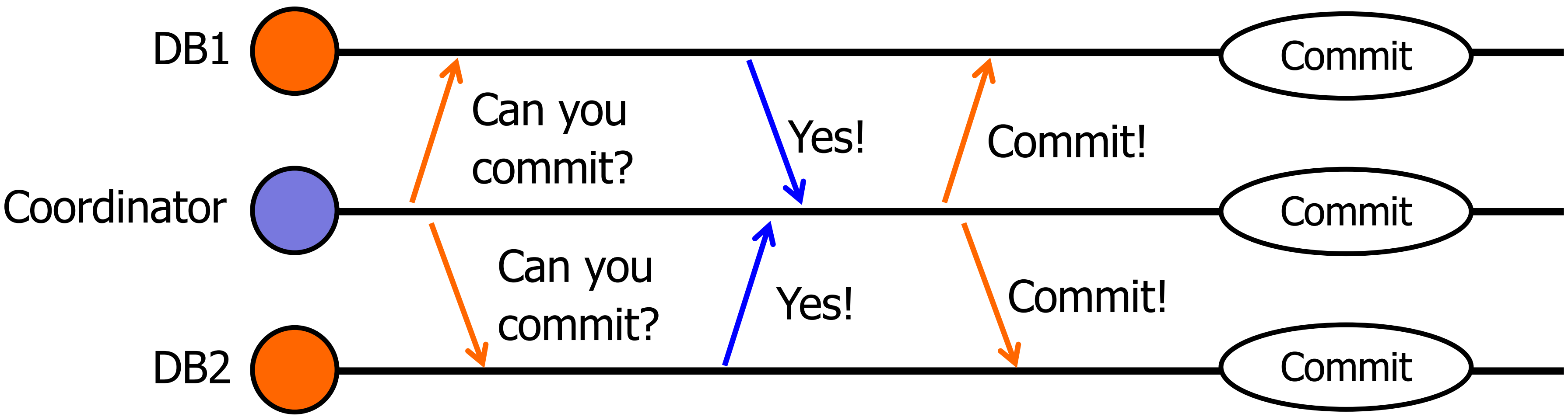
timofei.istomin@unitn.it

Atomic Commitment

- A special case of agreement…
- … particularly important for distributed databases
- What's a distributed database?
  - Two airlines, two separate ticket databases
  - We want to buy either both parts of the route or neither — the databases should either both commit or both abort
- **Distributed transaction**
- Databases have to agree on whether to commit/abort

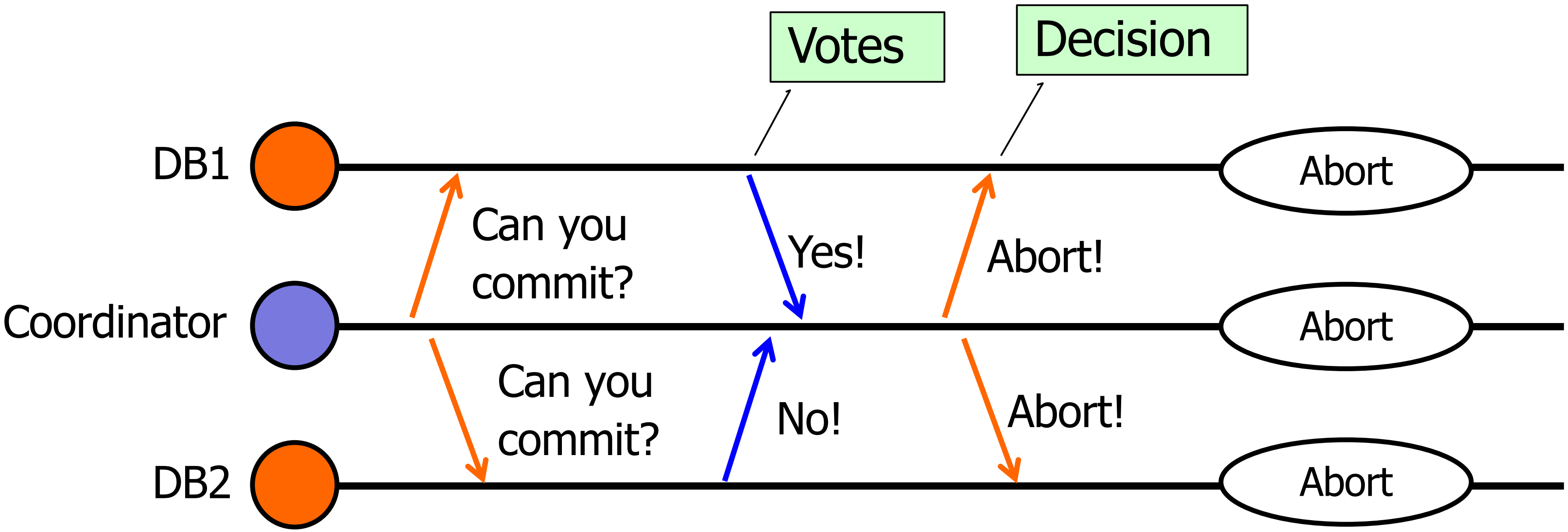
AC without faults is simple

- A dedicated coordinator process asks the DB servers if they are able to commit the transaction
- If all answer Yes, it signals them to commit



AC without faults is simple

- … if someone cannot commit the transaction, all should abort



AC with faults is tricky

- What should the system do if a DB process crashes in the middle?
- What about a coordinator crash?

- What should the coordinator do if a vote does not arrive in time?
- What should a participant do if the decision does not arrive in time?

- We study distributed algorithms that provide answers to these problems

Dealing with crashes

- Processes may store (log) their decisions (and the current step of the protocol) to the stable storage
- If they crash, they can later **recover** by retrieving the state from the log file
- **We assume that storage never fails!**

This converts crash failures into
response omission + performance failures!

Dealing with packet loss

- To deal with lost packets we could, e.g.:
  - Coordinator: re-request periodically the vote from a participant
  - Participant: re-request periodically the decision from the coordinator

This converts response omission into performance failures!

Dealing with slow responses

- We have converted all failures into performance failures!
- But how to deal with them?
- Processes could just wait until the response finally arrives
  - If it's fine for our customers, we are done!
- But they may have to wait for a really long time
  - e.g., when a server must be physically replaced

Blocking

- We say that a process is **blocked** if it has to wait for a reply from another process before it can complete the algorithm
- In other words, it cannot simply timeout in case of no-reply and still guarantee *safety*
  - I.e., in this case, make sure that its decision is coherent with everyone else's

Safety

- Ideally, we want to design a correct **and** non-blocking protocol under all failure modes

Liveness

Impossibility results

- But… it is proven that
  - There are **no** non-blocking AC algorithms in the presence of communication failures
    - Due to the "Coordinated attack" problem
  - There are **no** non-blocking AC algorithms in the presence of total crash failures
    - i.e., at least one non-faulty process is required

- AC is **not** solvable with Byzantine failures:
  - a process might pretend to be correct but decide on the opposite to the global decision in the end

Atomic Commitment

- Properties:
  - **Uniform agreement:** no two processes (correct or not) decide differently (commit vs. abort)

    A process is correct if it never crashed during the whole execution
  - **Validity:** global "commit" decision is possible only if ***all*** processes voted "yes"

    -> If some voted "no", all *must* ABORT
  - **Non-triviality:** global decision ***must*** be "commit" if ***all*** voted "yes" and there are no faults

    -> if there are faults, the global decision *may* be ABORT
  - **Termination:** If there are no faults (or there are, but all get repaired), all processes eventually decide
  - **Decision is final**: once decided, cannot change even with failures
- Goal: design a *correct* atomic commitment protocol that is *unlikely* to block

Two-phase Commit

- It is the most popular algorithm
- Correct even in asynchronous systems with temporary communication failures
  - slow reactions become "performance failures": a fault is caused by a **timeout expiration**
    - a node gives up waiting for a message
  - this relaxes the **non-triviality** and **termination** requirements
    - more situations where the protocol is allowed to ABORT or not terminate…

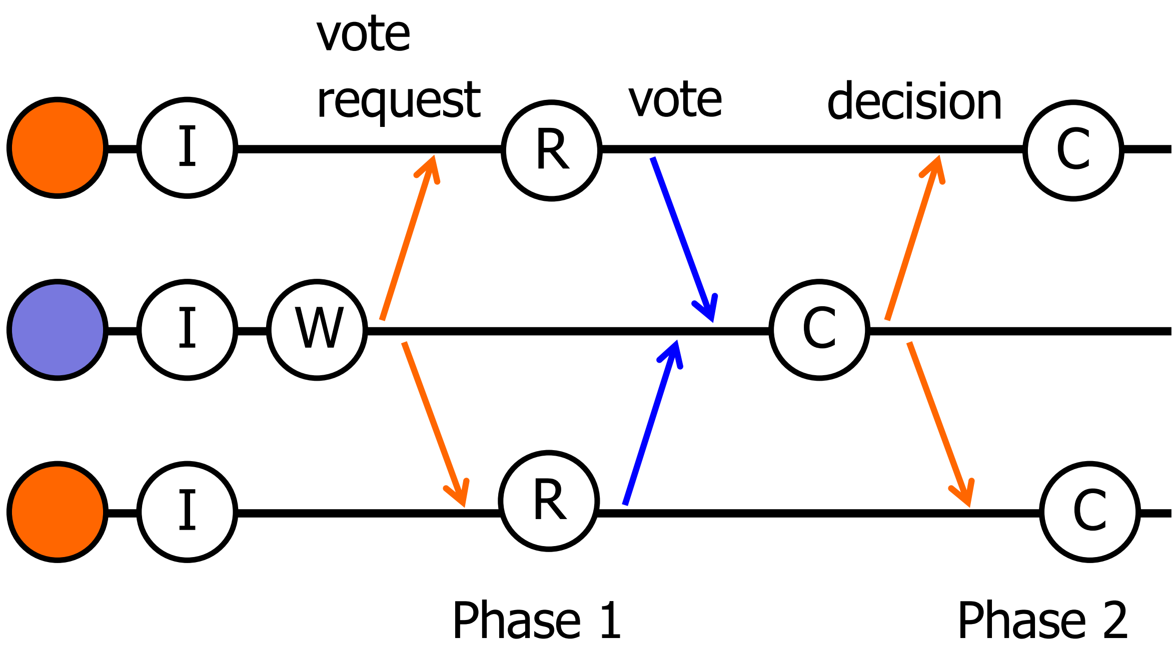
How come? Before we saw that agreement is impossible in async systems with crash failures

The crashes are **masked** using reliable storage!

Two-phase Commit

Round 1: coordinator requests votes, collects responses and decides COMMIT or ABORT

Round 2:
coordinator sends out the decision



States:

INIT, **W**AIT,
READY, **C**OMMIT,
ABORT

Termination

- Question: when does a process decide (and therefore terminate)?
- Anyone voting NO decides ABORT immediately, at the beginning of the transaction
  - Processes may decide to vote NO for applicative reasons, e.g., a leg of a multi-leg flight is unavailable
- Coordinator decides after collecting votes
  - If it sees a NO, decides ABORT;
  - otherwise it decides COMMIT
- All the other nodes decide when they receive the final decision from the coordinator

Coordinator timeout

- This is fine, but what if a *participant* (*cohort*)
  - crashes,
  - slows down, or
  - gets isolated <u>before sending the vote,</u>
  - or the vote gets lost in the network?
- We could wait until the participant recovers, but this is in general impractical…
- To avoid blocking, the coordinator can safely time out, decide ABORT and terminate
  - even if all voted YES, we are allowed to abort in the presence of failures (detected by the timeout)

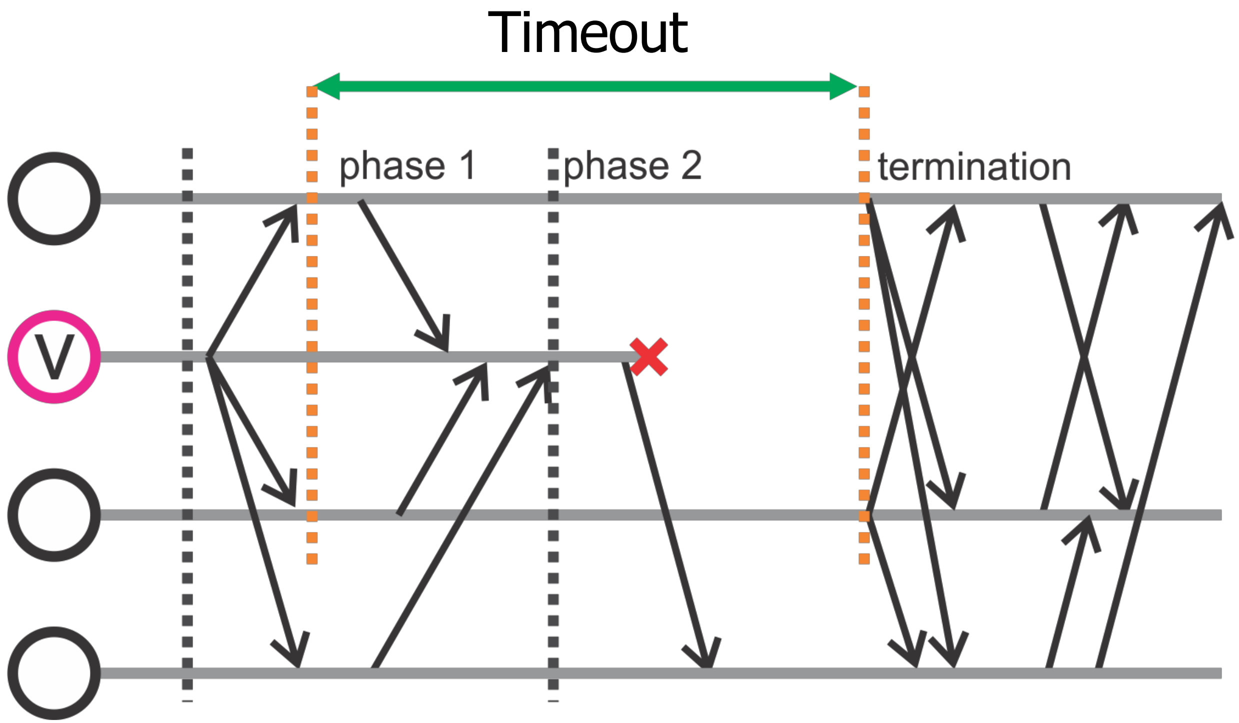
Can participants timeout?

- The *coordinator* can also fail!
- Then participants can timeout in various situations:
  - before sending a vote -> the participant can unilaterally ABORT
  - after sending YES **vote**, while waiting for the global decision -> the participant asks everybody else for the **decision**:
    - if it finds a COMMIT, commits;
    - if it finds an ABORT, aborts;

Termination protocol

2PC Termination protocol

- If the decision is not arriving, cohorts try to contact other processes



- What if nobody knows the decision?

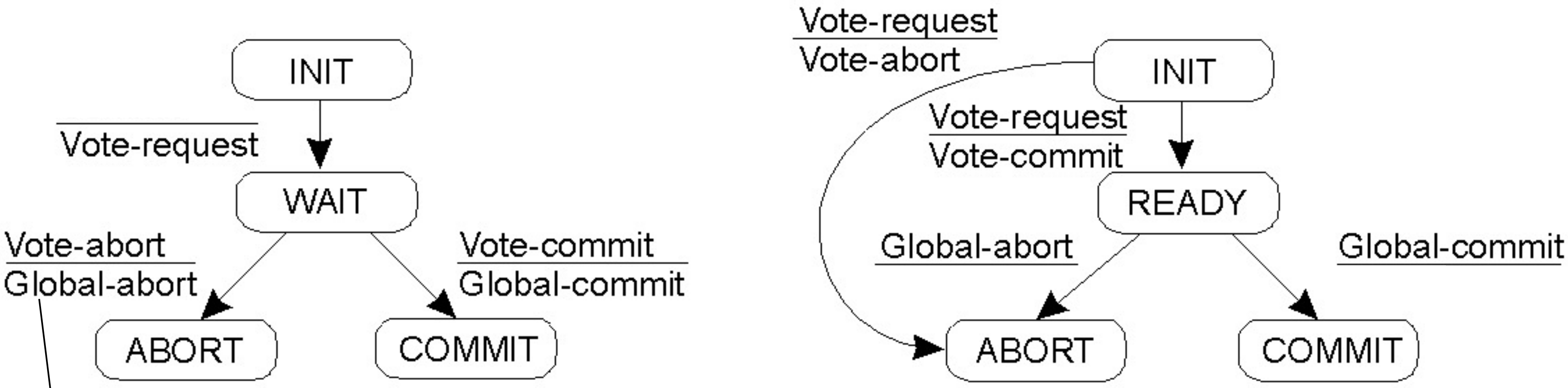
Two-phase Commit

- If no one who's alive has received the decision
  - Can they COMMIT? No!
    - e.g., another cohort may have decided ABORT and crashed before telling this …
    - … or the coordinator may have decided ABORT (e.g. due to timeout) and crashed before sending it
  - Can they ABORT? No!
    - e.g., the coordinator might have decided to COMMIT and crash before sending it
- What can they do, then?
  - Nothing. Must **block** and **wait** until the coordinator recovers

blocking protocol

Two-phase Commit

State diagrams for coordinator and participant



coordinator

cohort = participant

Also in case of timeout

Two-phase Commit

Actions by coordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;

while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
}

if all participants sent VOTE_COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Init

Wait

Commit

Abort

Two-phase Commit

Actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write GLOBAL_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; //remain blocked
    }
    write DECISION to local log;
} else {
    write GLOBAL_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
```

Init

Ready

Commit or Abort

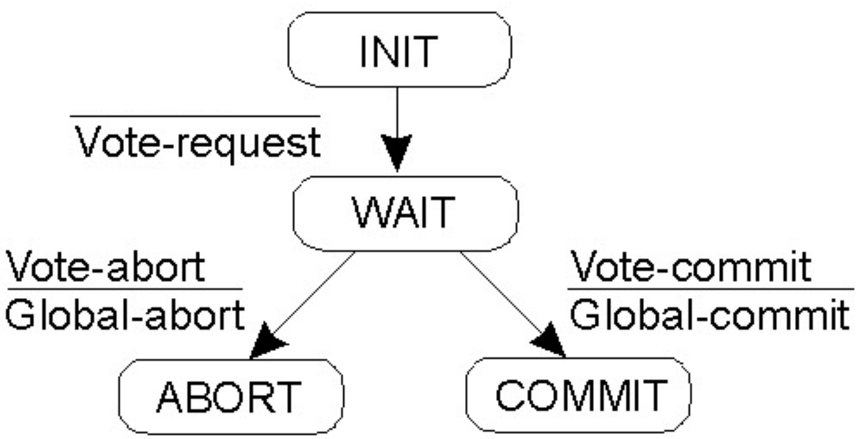
Abort

2PC: recovery actions

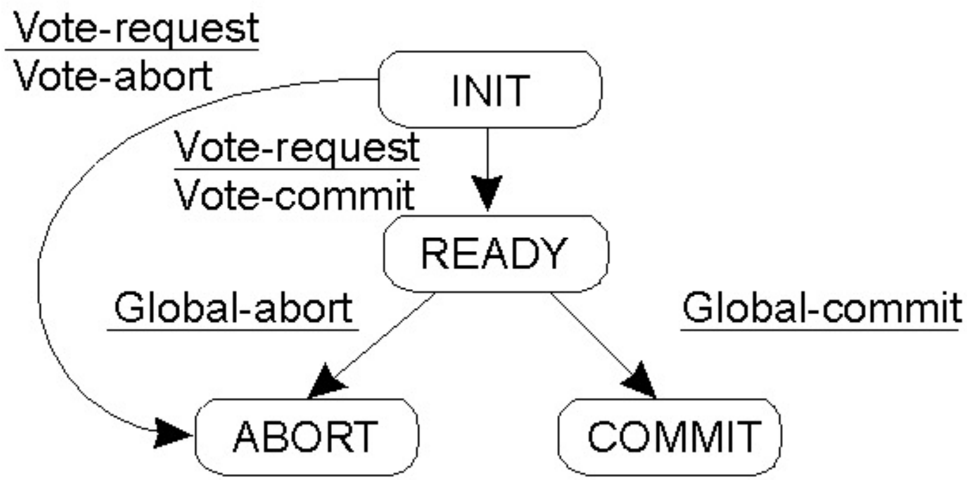
- If crashed, the process is restarted and executes a recovery action which depends on the logged state:
  - **Cohort**:
    - not even voted -> ABORT
    - decided (COMMIT/ABORT) before crashing -> do nothing
    - READY -> perform the termination protocol
  - **Coordinator**:
    - not decided -> ABORT (why is it safe?)
    - decided -> send the decision to participants

Two-phase Commit

- 2PC leads to blocking because a process can go from an **uncertainty state** directly to **commit** or **abort**

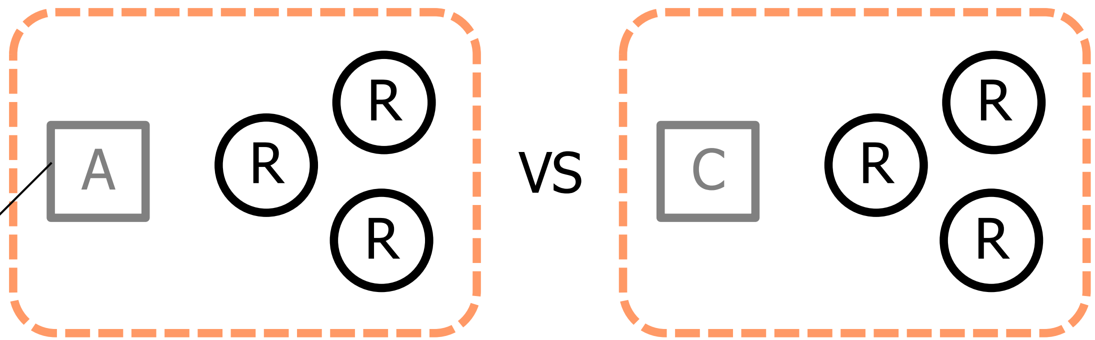


Coordinator



Cohort

Crashed coordinator



"R"s voted yes but do not know in which state are the crashed ones

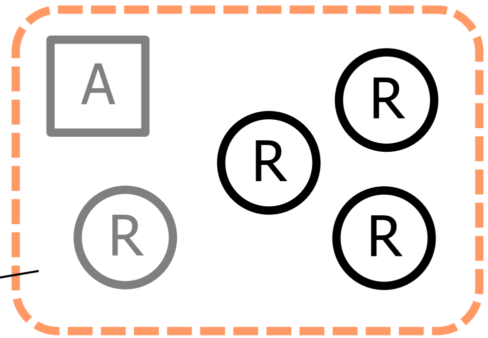
Towards three-phase commit

- 2PC leads to blocking because a process can go from an **uncertainty state** directly to **commit** or **abort**

- What if we could guarantee the following property:

  P1: If any *operational* process is uncertain (R), then no process (operational or failed) can have decided to commit (C)

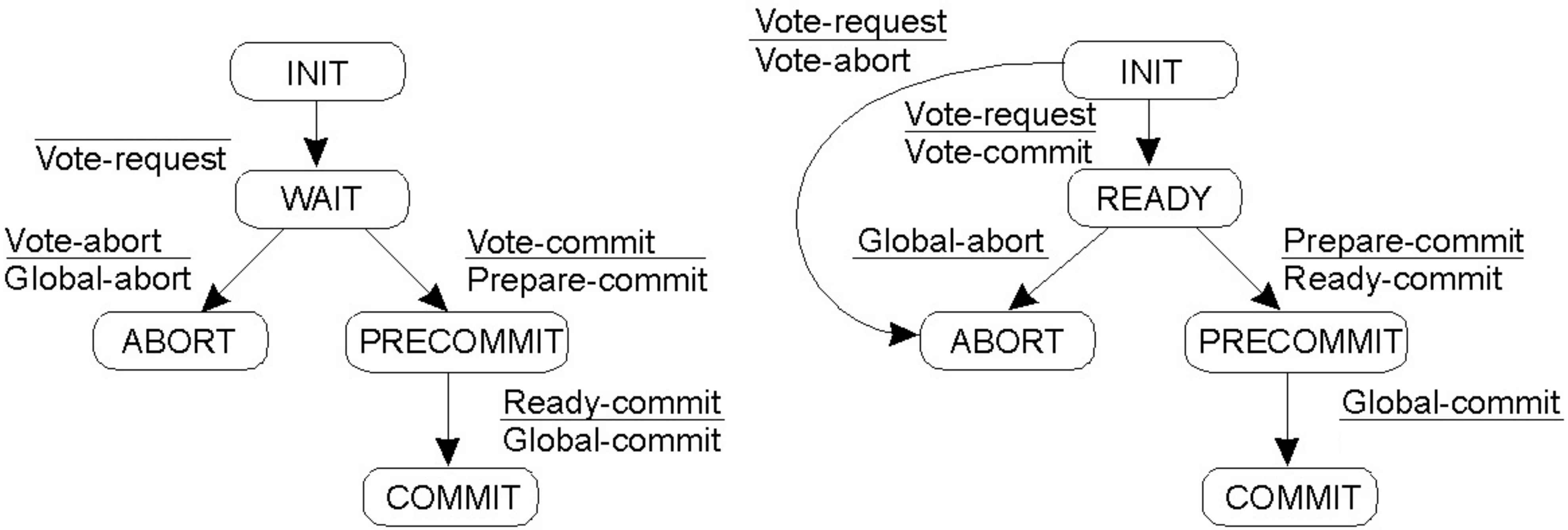
- That would solve our problems. Why?



Nobody could have committed, the survivors may **abort** without blocking

Three-phase Commit

- 3PC does exactly that, by introducing a PRECOMMIT (P) state (it is not uncertain)

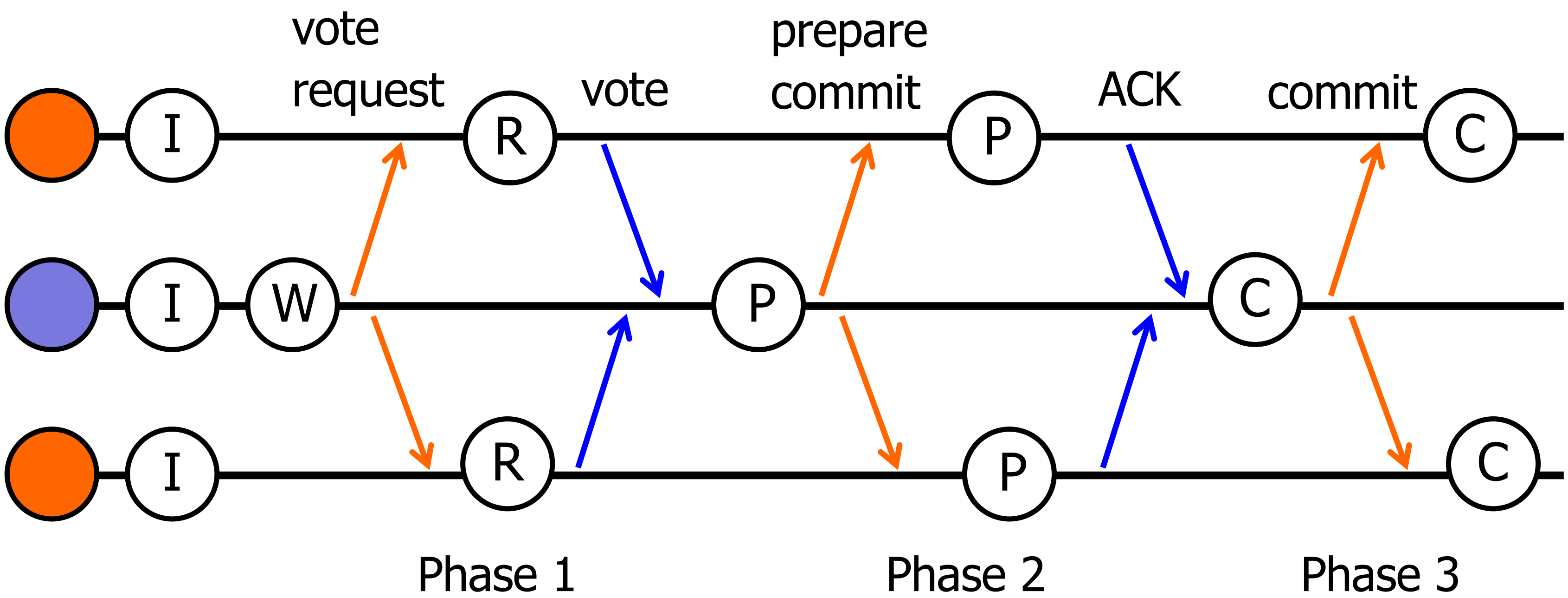


Coordinator

Cohort

- Coordinator first tells everyone that all cohorts voted YES, and only later signals COMMIT

Three-phase Commit



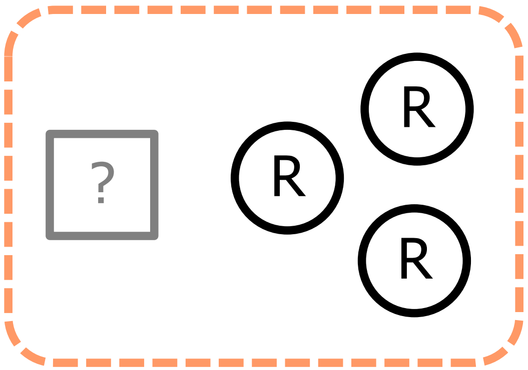
Three-phase Commit

Let’s assume that we have a synchronous system with reliable network

- We’ll relax this requirement later

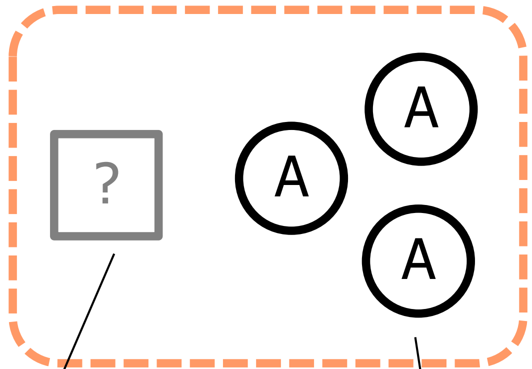
Three-phase Commit

- If all survivors are uncertain (in READY state)



this caused
blocking in 2PC

Now they can safely
ABORT without waiting

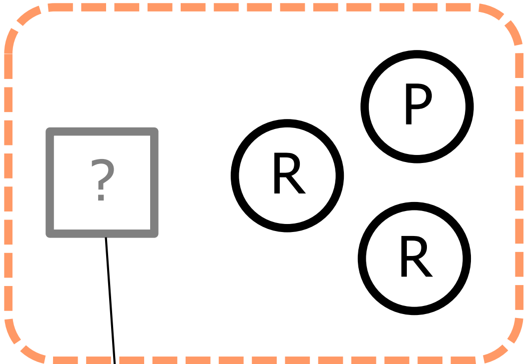


It can be either uncertain, in
PRECOMMIT or decided to ABORT

To decide collectively but without the
coordinator they use a termination
protocol which we'll see later

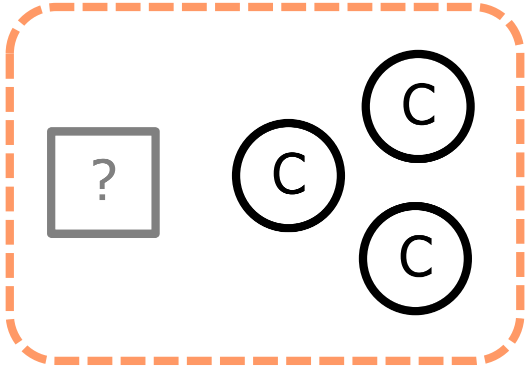
Three-phase Commit

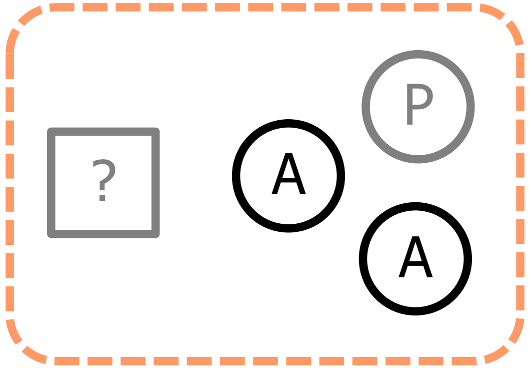
- If **some** are in PRECOMMIT (P)
  - It is known that all voted YES



If "P" survives: COMMIT

If "P" crashes: ABORT





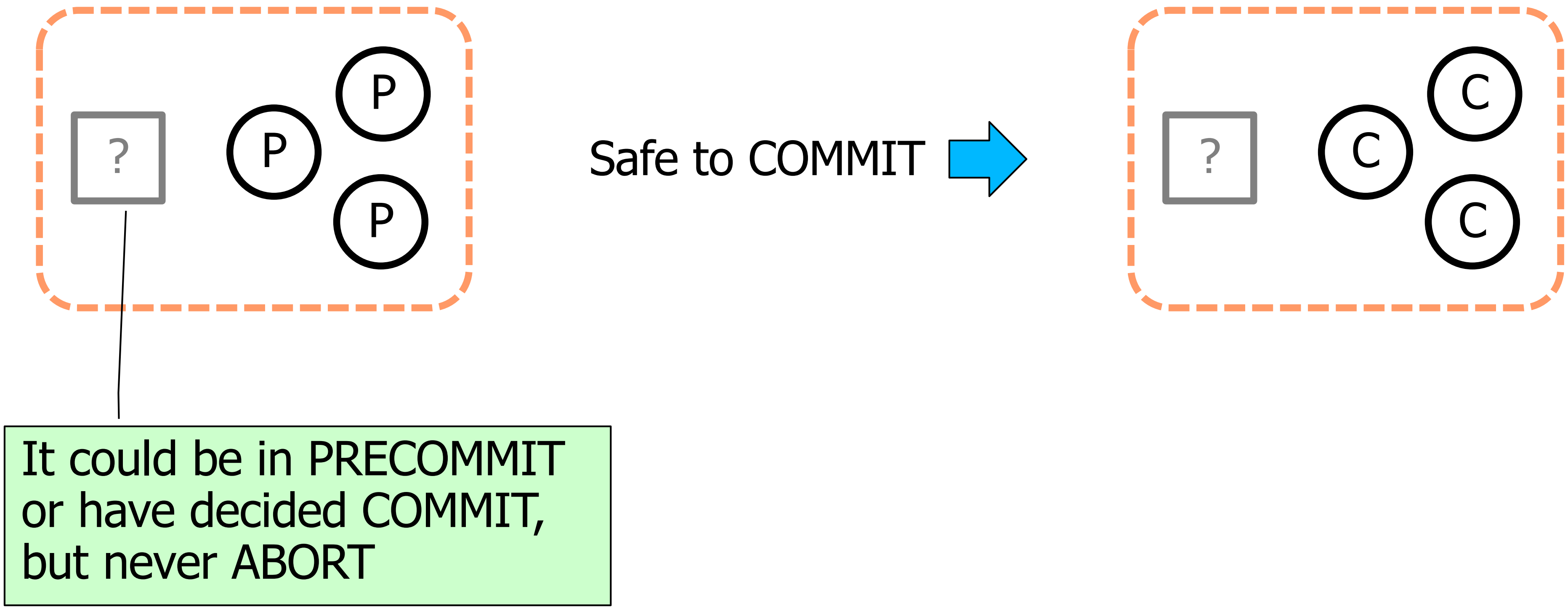
Cannot have decided COMMIT, because some cohorts are still in READY

Neither can have decided ABORT because there are cohorts in P

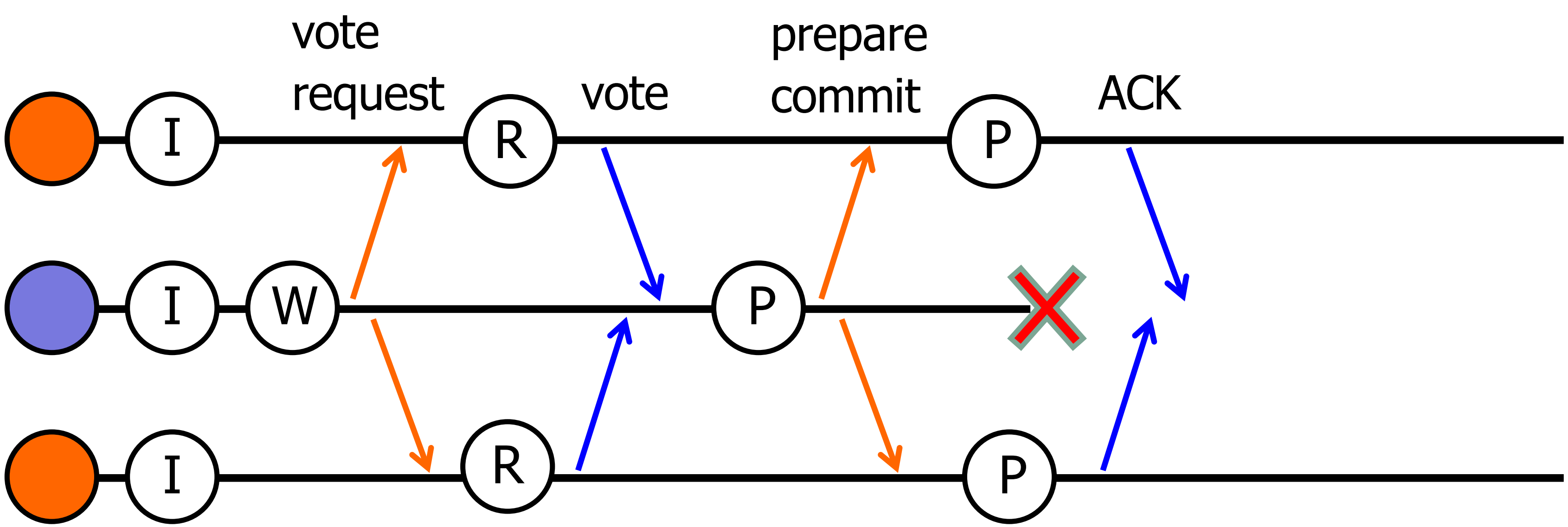
When the crashed ones recover they will ask the decision from others

Three-phase Commit

- If **all** are in PRECOMMIT

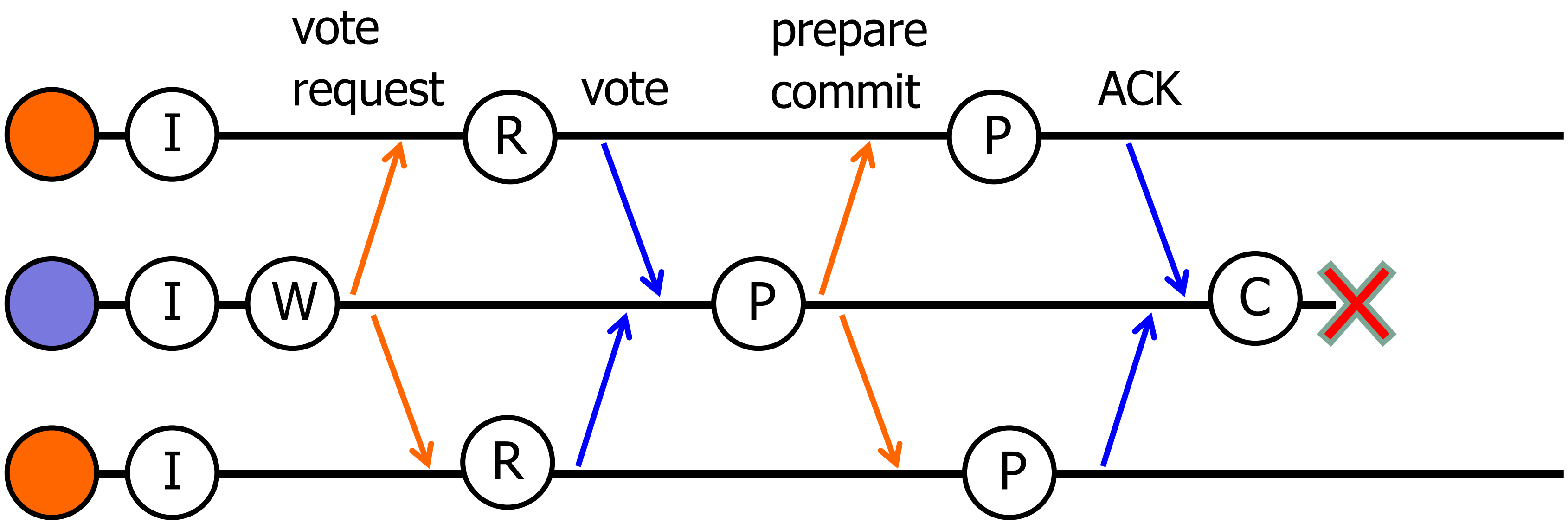


If all are in PRECOMMIT (1)



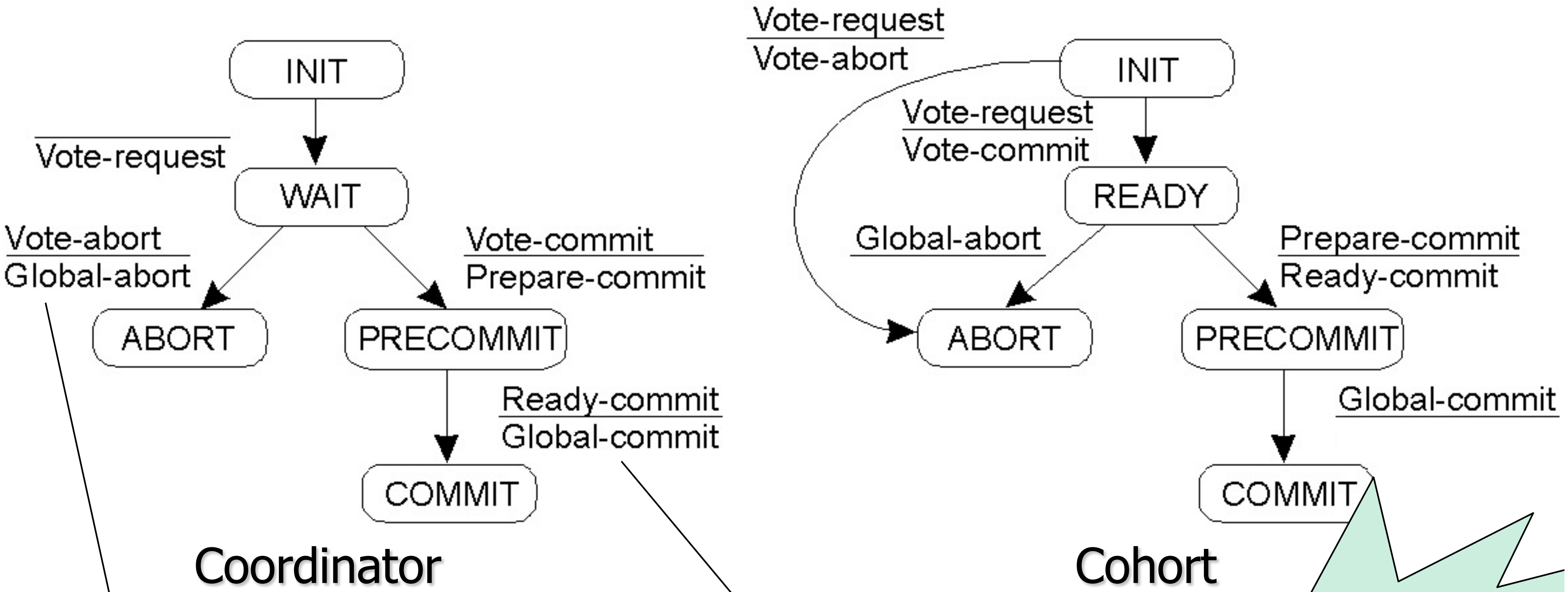
The coordinator will recover in P,
it could not have aborted

If all are in PRECOMMIT (2)



The coordinator has committed

Three-phase Commit



Also in case of timeout

Even if timed out: some (or all) cohorts crashed, but it does not break **P1**

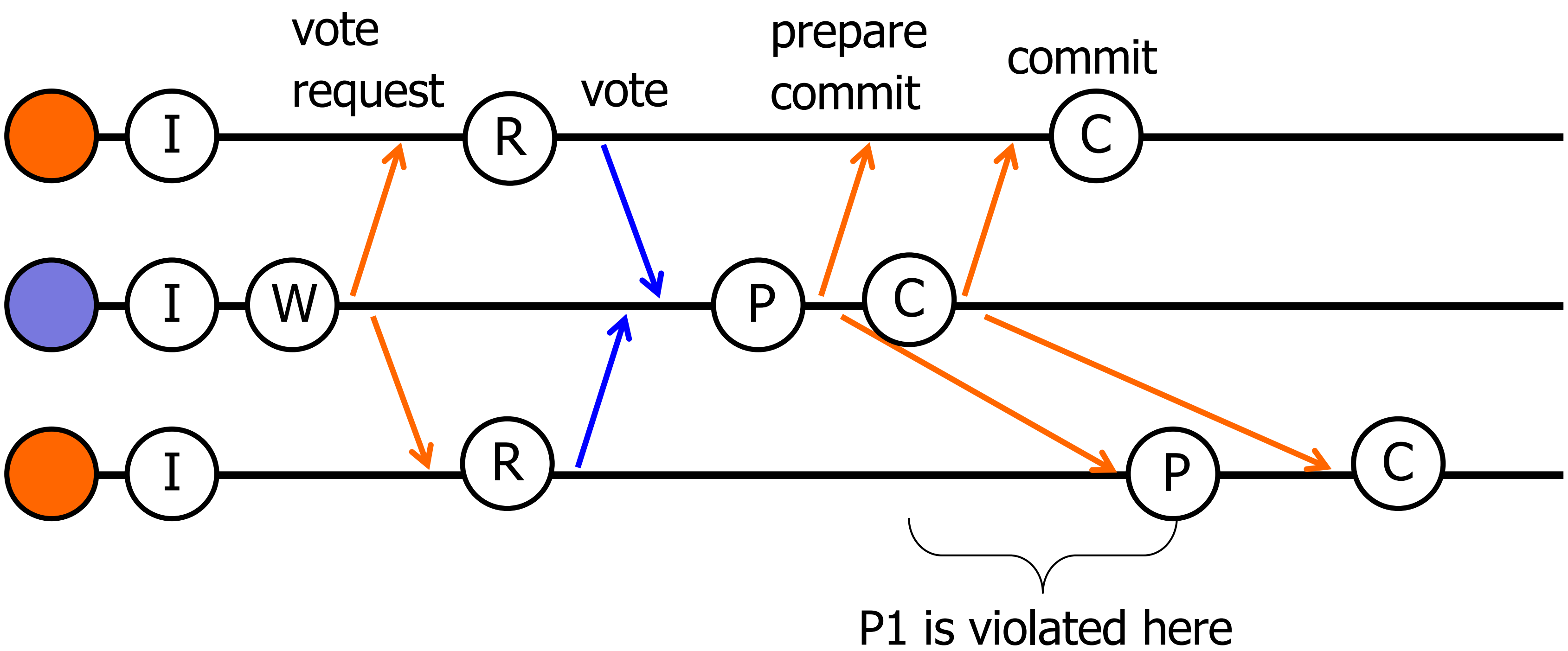
Will not work with network or performance failures

Why acknowledge?

- We wanted to design a non-blocking algorithm, so the coordinator should not wait in PRECOMMIT more than the fixed timeout
  - It cannot decide ABORT (would be like 2PC)
  - So it decides COMMIT
- Why do we need the pre-commit acknowledgements at all then?
- We don't need them for *safety*, but they improve *liveness*
  - Can send commit without waiting after having received ACKs from everyone

Why wait?

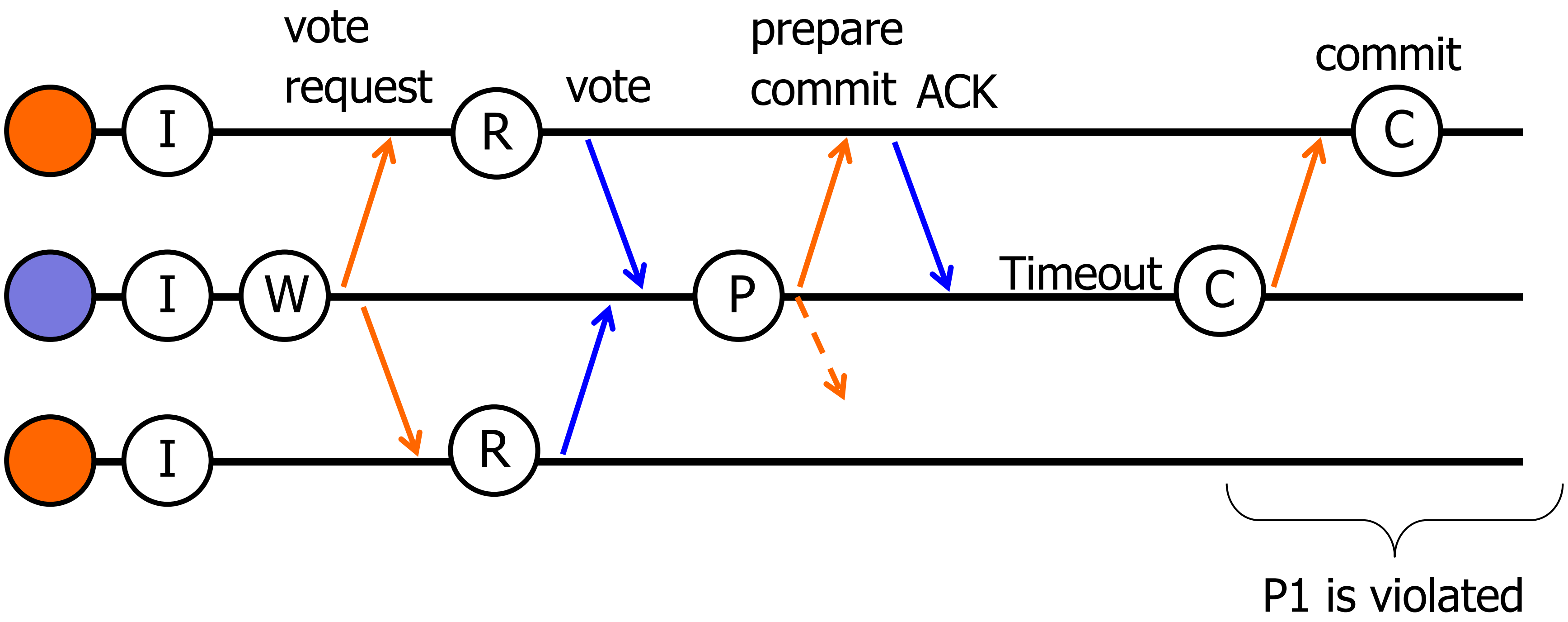
Why the delay (timeout) is at all needed in PRECOMMIT? Why not send COMMIT right away?



The timeout is needed to guarantee that a process has either received the PRECOMMIT message or crashed

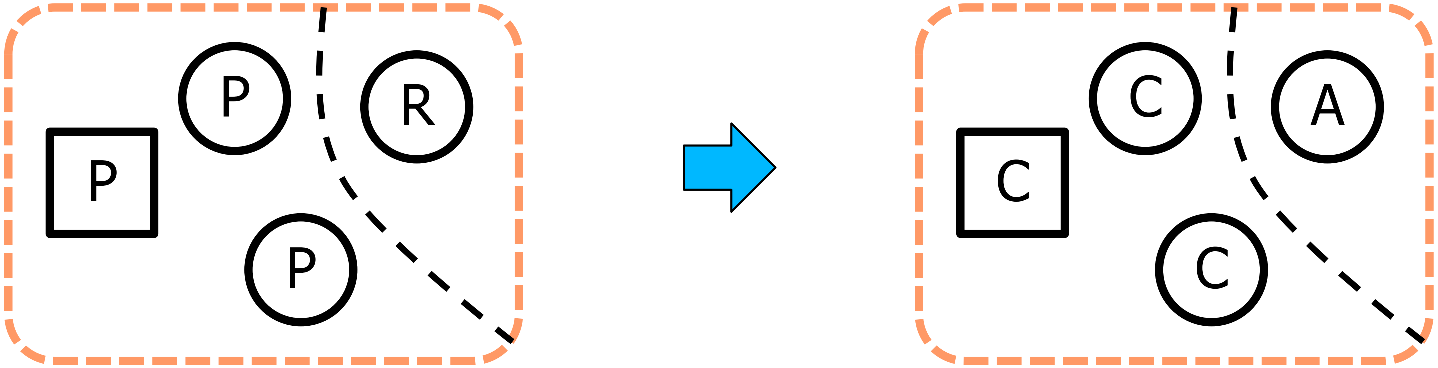
Network faults break it

If a cohort or the network are slow, or messages get lost, the coordinator might decide COMMIT even when the cohort is still in READY



The problem

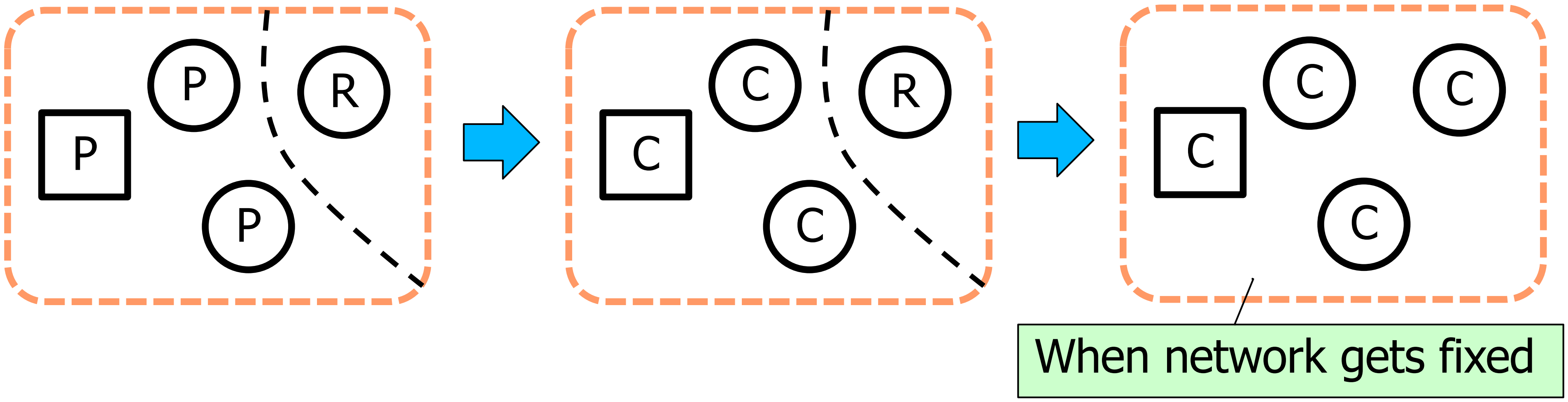
- It's dangerous to violate P1:
  - if the cohort gets isolated in READY and times out, it decides to ABORT (since it does not find anybody in PRECOMMIT or COMMIT around)



- This algorithm tolerates **only** crash failures!
  - it is **incorrect** in the presence of communication or performance failures

Solution is blocking

- The problem can be solved if we add a **majority rule**:
  - Can decide only if can contact a majority of nodes
    - This holds also for the partition with the coordinator!
  - But the isolated minority will have to **block** until the network is fixed



3PC: recovery actions

- On restart (after crash):
  - Any process:
    - decided (COMMIT/ABORT) before crashing -> do nothing
  - Cohort:
    - not voted -> ABORT
    - voted, but not decided -> ask the others
  - Coordinator:
    - started, but not decided -> ask the others

They might have decided without waiting for the coordinator's approval, so it has to ask

3PC: coordinator timeout

- If a *cohort* crashes:
  - and coordinator is waiting for the vote
    -> coordinator should abort
  - and coordinator is waiting in PRE-COMMIT
    -> it can safely commit (no risk that the failed cohorts had aborted since they voted “yes”)
    - some *failed* cohorts might still be in READY, but it does not violate **P1**
    - they will for sure commit after recovery

Actually, it can **only** commit. Cannot go to ABORT from PRECOMMIT (unless it reboots after a crash and finds others in ABORT).
And cannot wait for cohort to recover — would be a blocking protocol

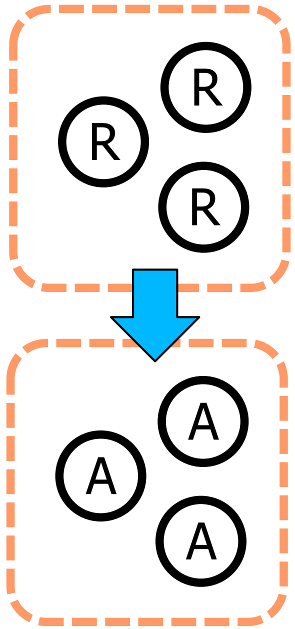
3PC: cohort timeout

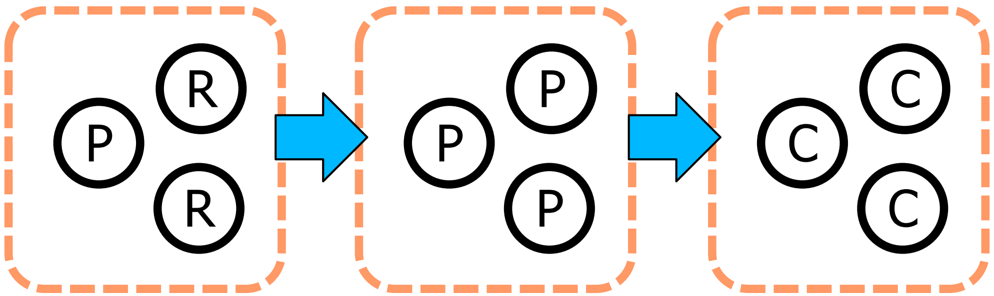
- If the *coordinator* crashes, cohorts (failed or not) can be mixed in the following combinations of states:
  - INIT/READY/ABORT — coordinator failed during vote request
  - READY/PRECOMMIT — coordinator failed during PRECOMMIT
  - PRECOMMIT/COMMIT — coord. failed during GLOBAL_COMMIT

- If a cohort times out in INIT, it can unilaterally abort: nobody can have committed so it's safe
- In *all other cases* a **termination protocol** is needed, which is more complex than in 2PC

3PC: termination protocol

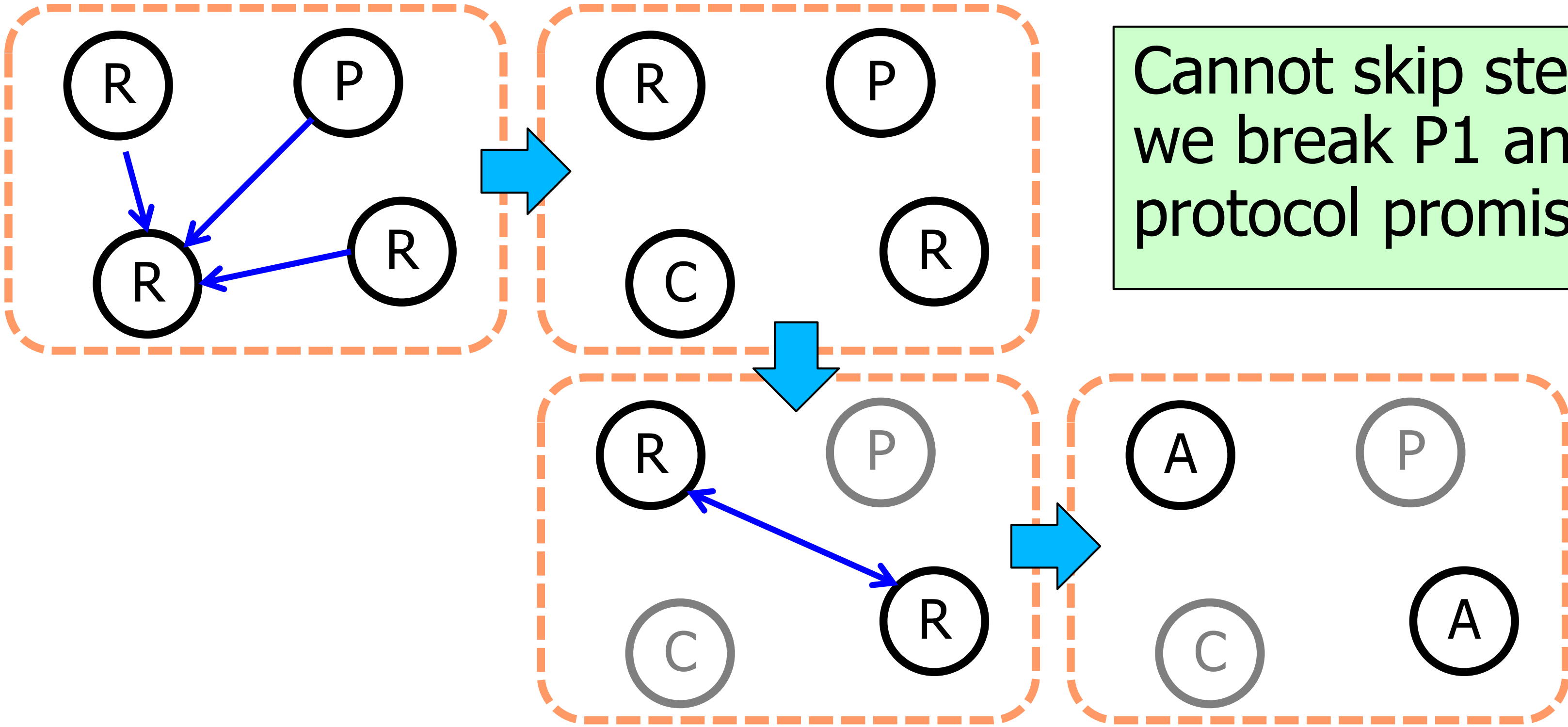
- The survivors *elect a new coordinator*
- It collects states of the survivors and completes the 3PC protocol:
  - if all in READY: aborts — safe (P1): nobody committed (why can't they commit?)
  - if any in COMMIT: commits everybody
  - if READY/PRECOMMIT: **first moves all uncertain participants to PRECOMMIT, then commits**





3PC: termination protocol

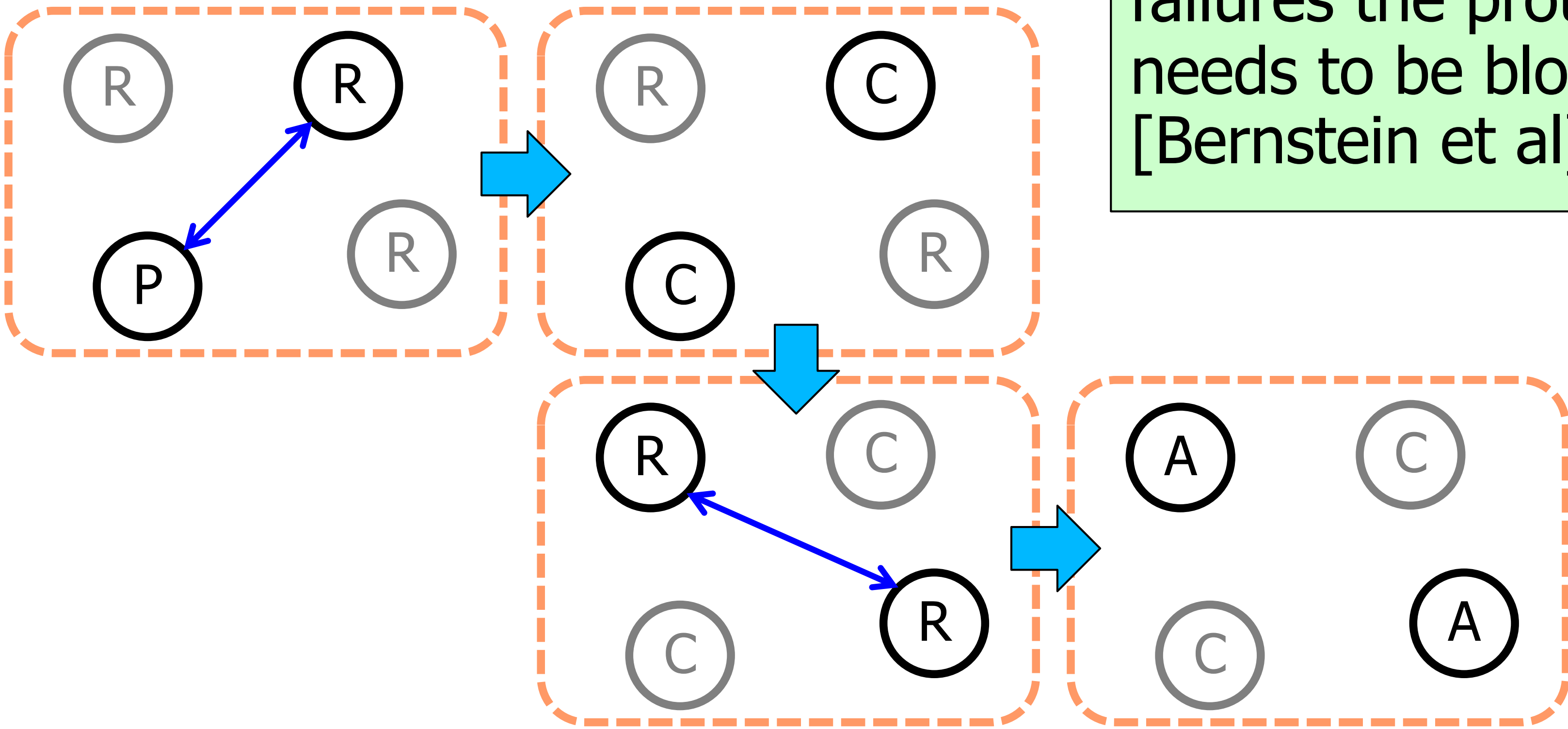
- Why do we need the new coordinator?
  - why can't a cohort just ask around and decide to commit if it finds another cohort in PRECOMMIT? Isn't the decision already taken?



Cannot skip steps or we break P1 and the protocol promises!

3PC: total failures

- The solution as presented goes well with *partial crash failures* (when at least one process is non-faulty)
- *total failures* are more tricky:



To tolerate total failures the protocol needs to be blocking [Bernstein et al]

AC: summary

- 2PC
  - Tolerates crash (including total), network, performance failures, but is **blocking** even with only crash failures
- 3PC
  - **Non-blocking** if only partial crash failures are expected
    - **incorrect**, if network/performance failures happen
  - **Blocking** if modified to tolerate network/performance failures (with the majority rule)
  - **Blocking** if modified to tolerate total crash failures

Agreement: reading

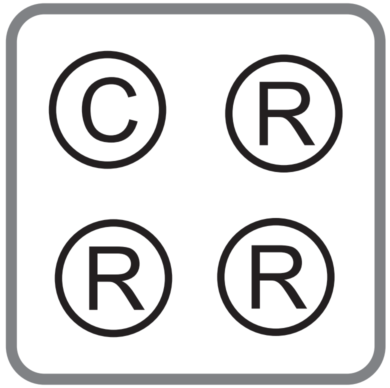
- Commit protocols (2PC/3PC):
  - Chap. 7: Bernstein, Hadzilacos and Goodman. Concurrency Control and Recovery in Database Systems. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

Exercise

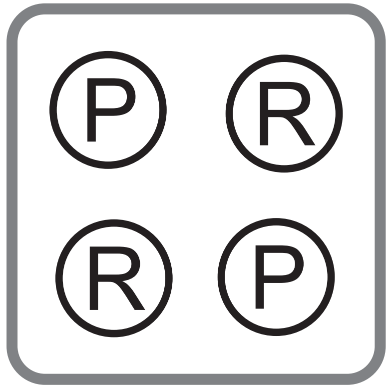
- In the context of the non-blocking *three-phase commit* protocol seen in class answer the following questions.
  1. Which failure modes the protocol is designed to tolerate?
  2. What does it mean for a protocol to be non-blocking?
  3. Consider the following example execution in a distributed system of 5 processes. The coordinator P0 receives YES votes from all the participants, then, before it sends the PRE-COMMIT messages, a network failure splits the processes in two isolated groups A={P0, P1} and B={P2, P3, P4} so that no process of one group can communicate with any of the processes of the other. Describe how the protocol will proceed, assuming that the network remains partitioned long enough for all the processes to time out and that no other faults occur. Characterise the outcome of the protocol. Does it violate any safety properties?

Exercise

- The figures depict the survivor sets right after the coordinator crash for the two atomic commitment protocols (2PC and 3PC) seen in class. Using time diagrams and/or text, show how recovery would unfold assuming no further faults. In particular, show: *i)* the messages exchanged among the processes, and their role/contents; *ii)* the outcome of the transactions. If there is the need to elect a new coordinator, simply point out who the new coordinator is without detailing the election protocol. Consider the base version of 3PC designed to tolerate only partial crash failures.



(a) 2PC



(b) 3PC