



Distributed Agreement

(with a side discussion on distributed transactions)

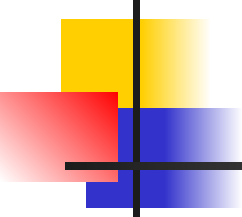


Distributed Agreement

- Two Phase Commit
- Three Phase Commit
- Paxos

Some slides from Roxana Geambasu / Columbia U.

<https://roxanageambasu.github.io/ds-class//assets/lectures/lecture17.pdf>

- 
-
- **Problem:** A set of nodes need to agree on a decision, value of a variable, or order of events.
 - Pervasive in distributed systems

(We've seen instances of the problem already)

- Bank transfer example: both parties to agree that their end of the transaction was successful
- A primary/backup protocol: nodes having to agree on the primary
- A chain replication protocol: nodes having to agree on the structure of the chain
- A quorum protocol: nodes having to agree on the next version ID

DISTRIBUTED AGREEMENT PROBLEM

- You want to go out with 4 friends at 6pm on Tuesday
 - Goal: go out only if ALL friends can make it
- What do you do?
 - Call each of them and ask if can do 6pm on Tuesday (**voting phase**)
 - If all can do Tuesday, call each friend back to ACK (**commit**)
 - If one can't do Tuesday, call other three to cancel (**abort**)
- Critical details:
 - While you are calling everyone to ask, people who've promised they can do 6pm Tuesday **must block that slot**
 - You need to remember the decision (commit/abort) and **tell everyone what the decision was.**



Two 'flavors' for the distributed agreement problem:

Participants need to agree on a value / action ...

- [consensus problem] ... and they are willing and capable to accept any value.
- [atomic commitment problem] ... and they have specific constraints on whether they can accept any particular value.

We'll focus on this first



ACID properties

- **Atomic: All or nothing**
 - State shows either all the effects of a transaction, or none of them:
- **Consistent** (think of it as **C**orrect): **Guarantees basic properties**
 - A transaction moves the database from a state where integrity holds, to another where integrity holds (given constraint, triggers, etc)
- **Isolated: Each transaction runs as if alone**
 - Effect of concurrent transactions is the same as transactions running one after another (ie looks like batch mode)
- **Durable: Cannot be undone**
 - Once a transaction has committed, it can not be undone in spite of failures.

Where would you need distributed agreement?

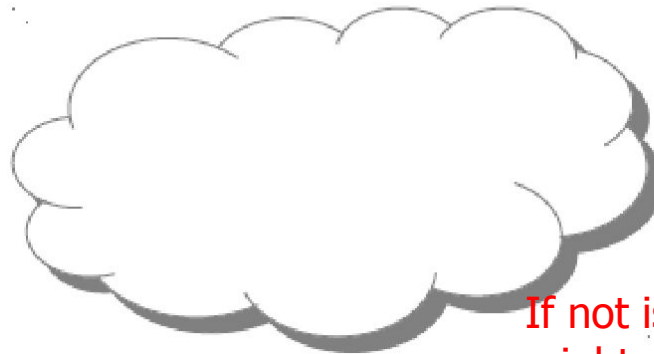
Example: bank transfer between two banks



client

transfer (X@bank A, Y@bank B, \$20)
Suppose initially: X.bal = \$100
Y.bal = \$3

Bank A

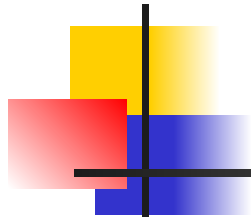


Bank B

If not isolated and atomic:
▪ might overdraw the src account
▪ might "create" or "destroy" money

Clients desire:

- 1. Atomicity: transfer either happens or not at all
- 2. Concurrency control: maintain serializability
(the "I" – Isolated in ACID)
- 3. Durable: once transaction has committed,
it can not be undone in spite of failures.



- A set of processes
 - Transaction Coordinator (TC): initiates an action:
 - Transaction Participants (TP): may vote for/against the action

- Goal: all-or-nothing outcome
 - All will perform the action only, if all vote in favor;
if any votes against (**or does not vote**), all will “abort” (none will take the action)

With: isolation and durability
(all in the presence of faults)



Desired properties of a solution:

■ Safety (correctness)

- All nodes agree on the same value (action)
- The agreed value has been proposed by some node
(the one proposed by the coordinator)
- [when failures] Never forget the decision taken

■ Liveness (availability)

- [if less than some fraction of nodes crash] the correctly operating nodes should reach agreement and the system should operate normally



Strawman solution

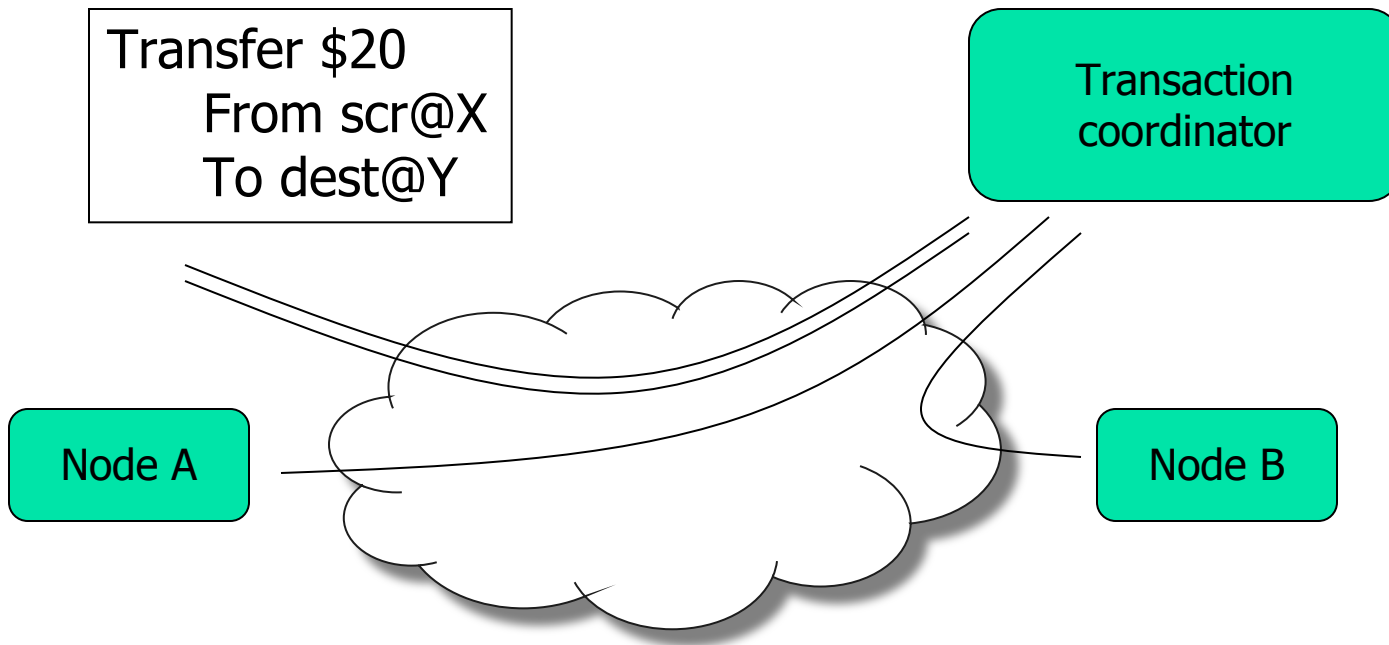


Transfer \$20
From scr@X
To dest@Y

Transaction
coordinator

Node A

Node B

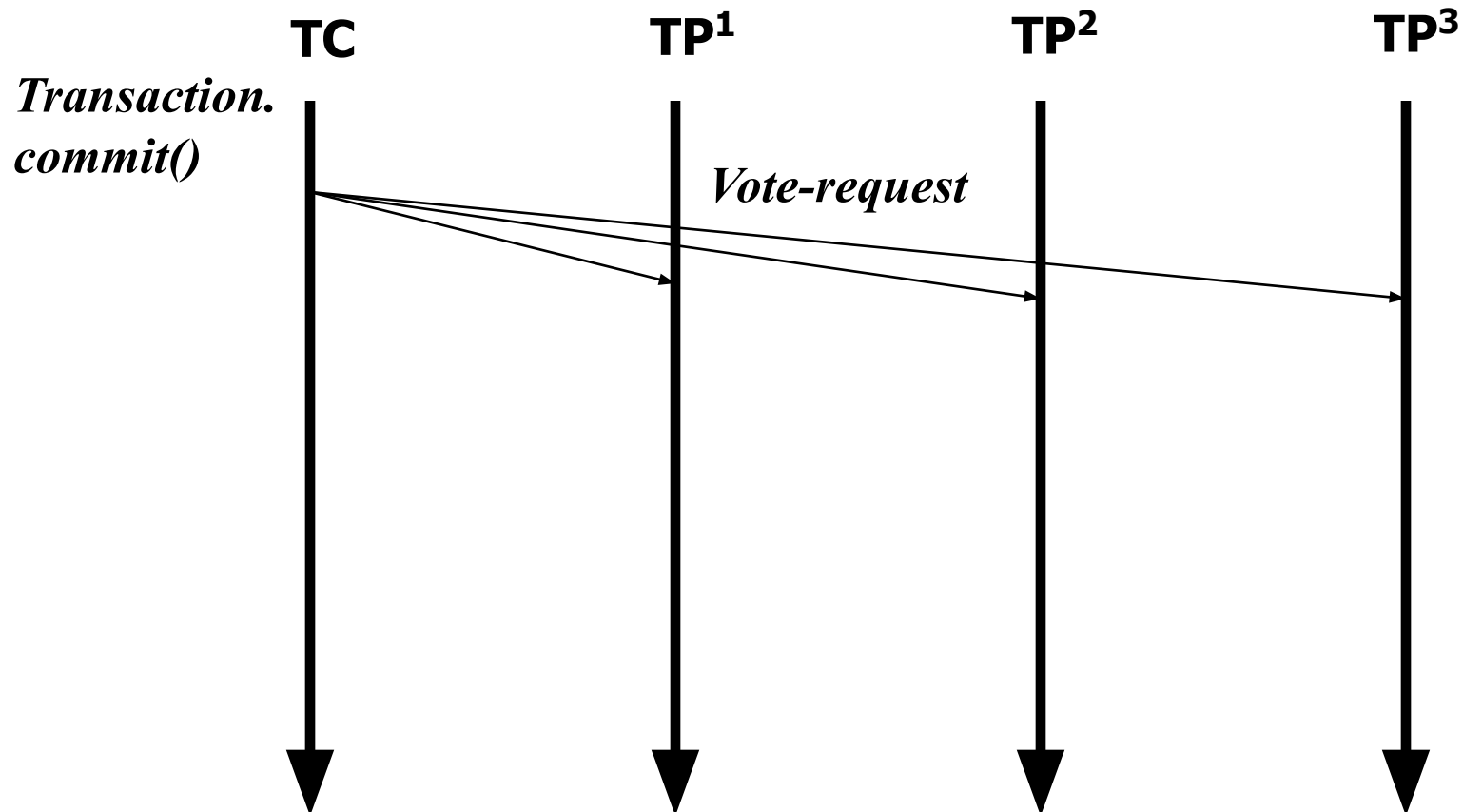




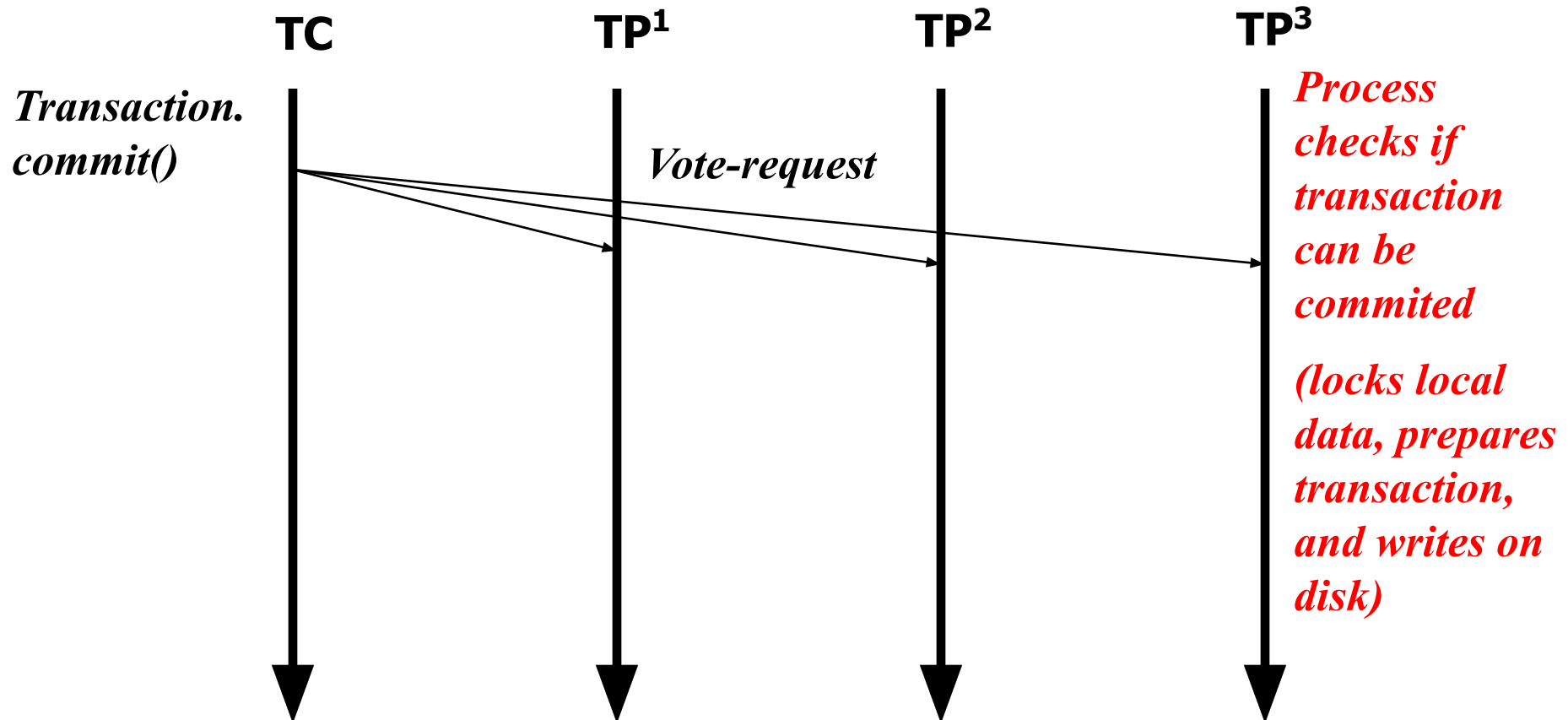
Typical protocol: two-phase commit (2PC)

- Coordinator asks all processes if they can take the action
- Processes decide if they can, and send back “commit” or “abort”
- Coordinator collects all the answers (or times out)
- Coordinator computes outcome and sends decision back

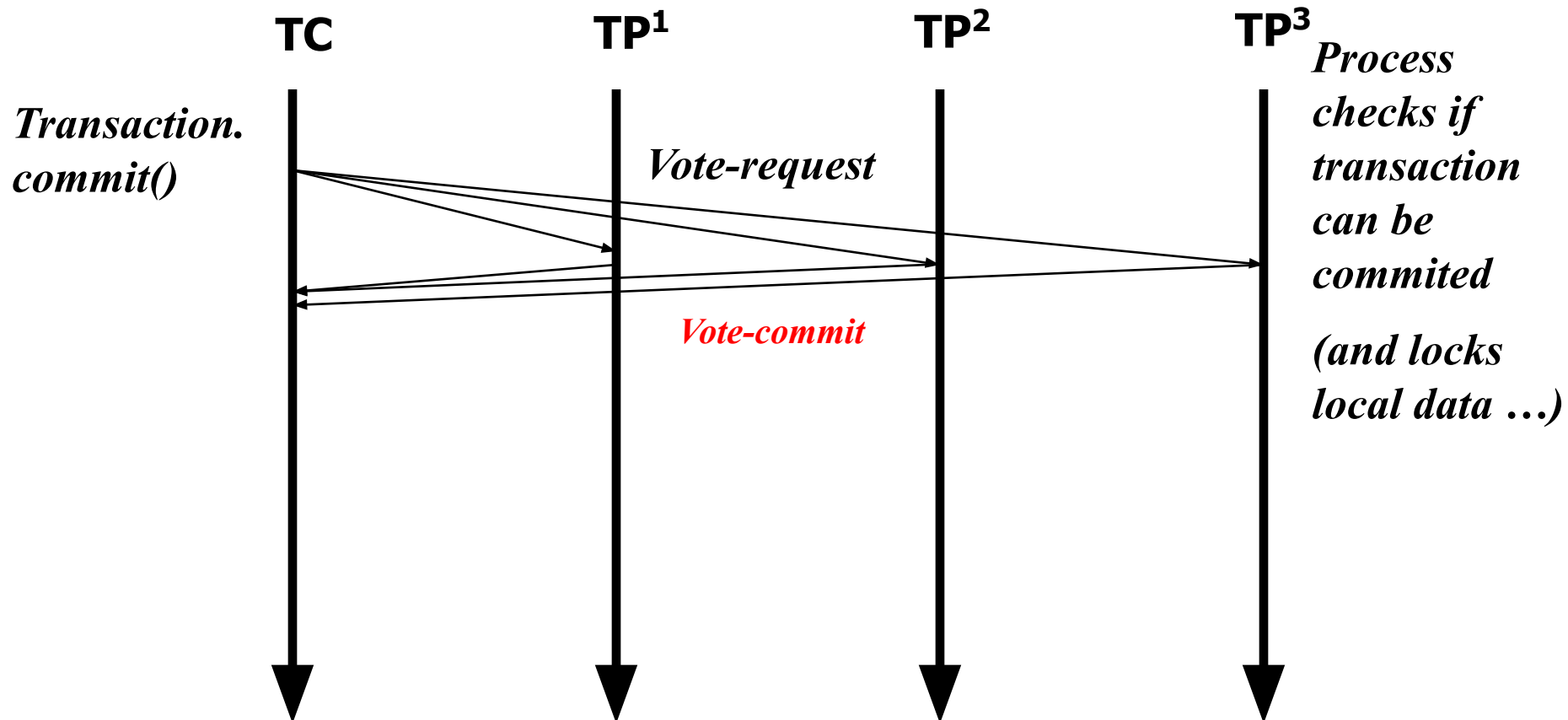
2-Phase Commit Protocol Illustrated



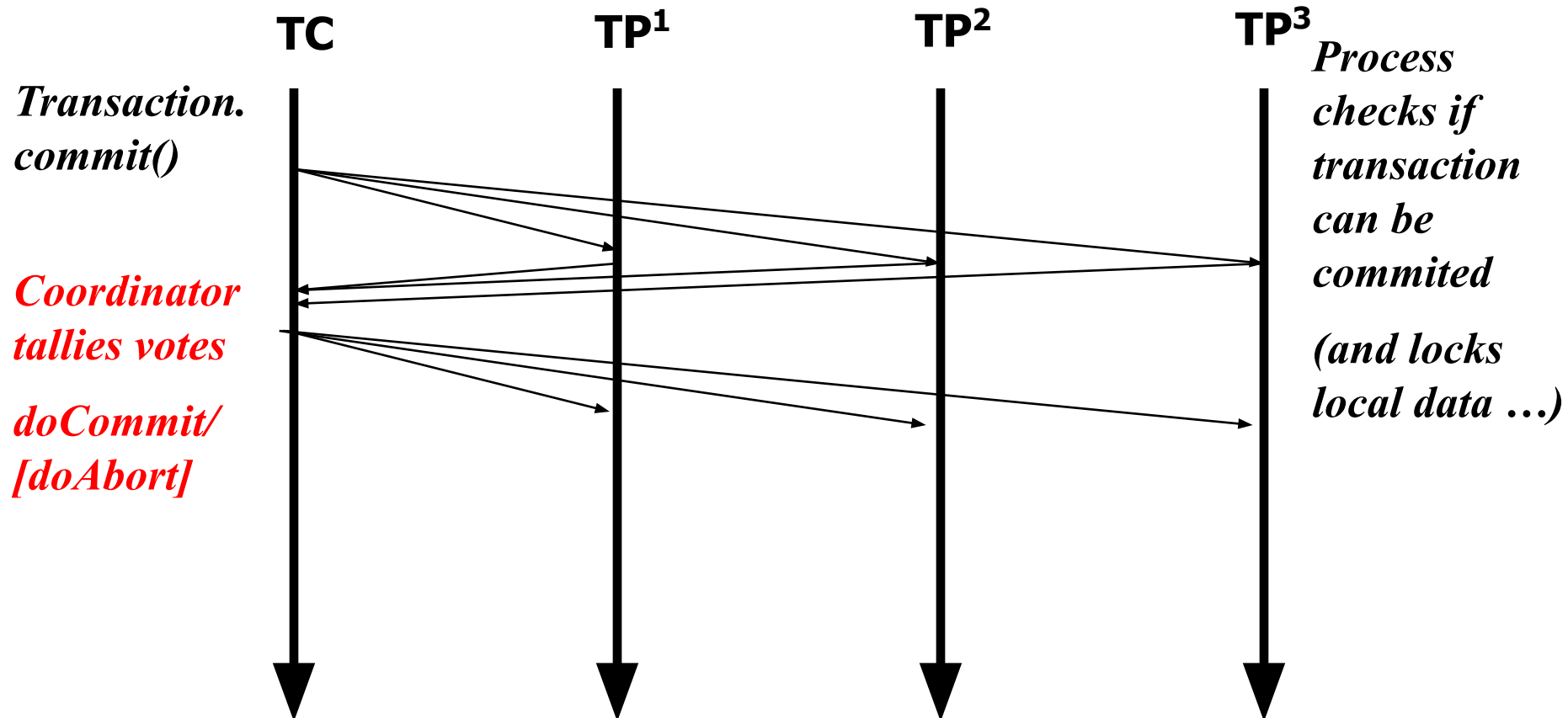
2-Phase Commit Protocol Illustrated



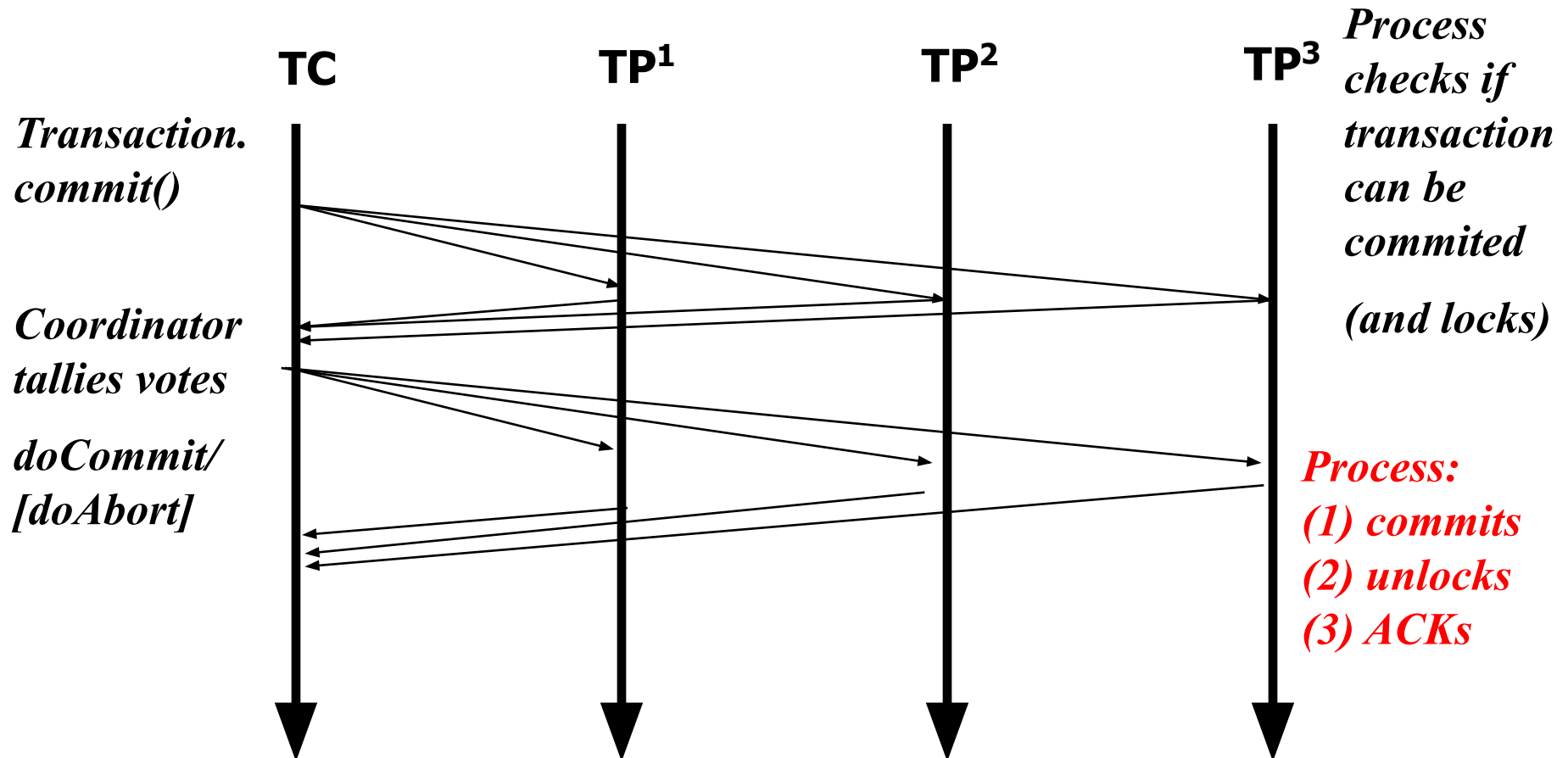
2-Phase Commit Protocol Illustrated



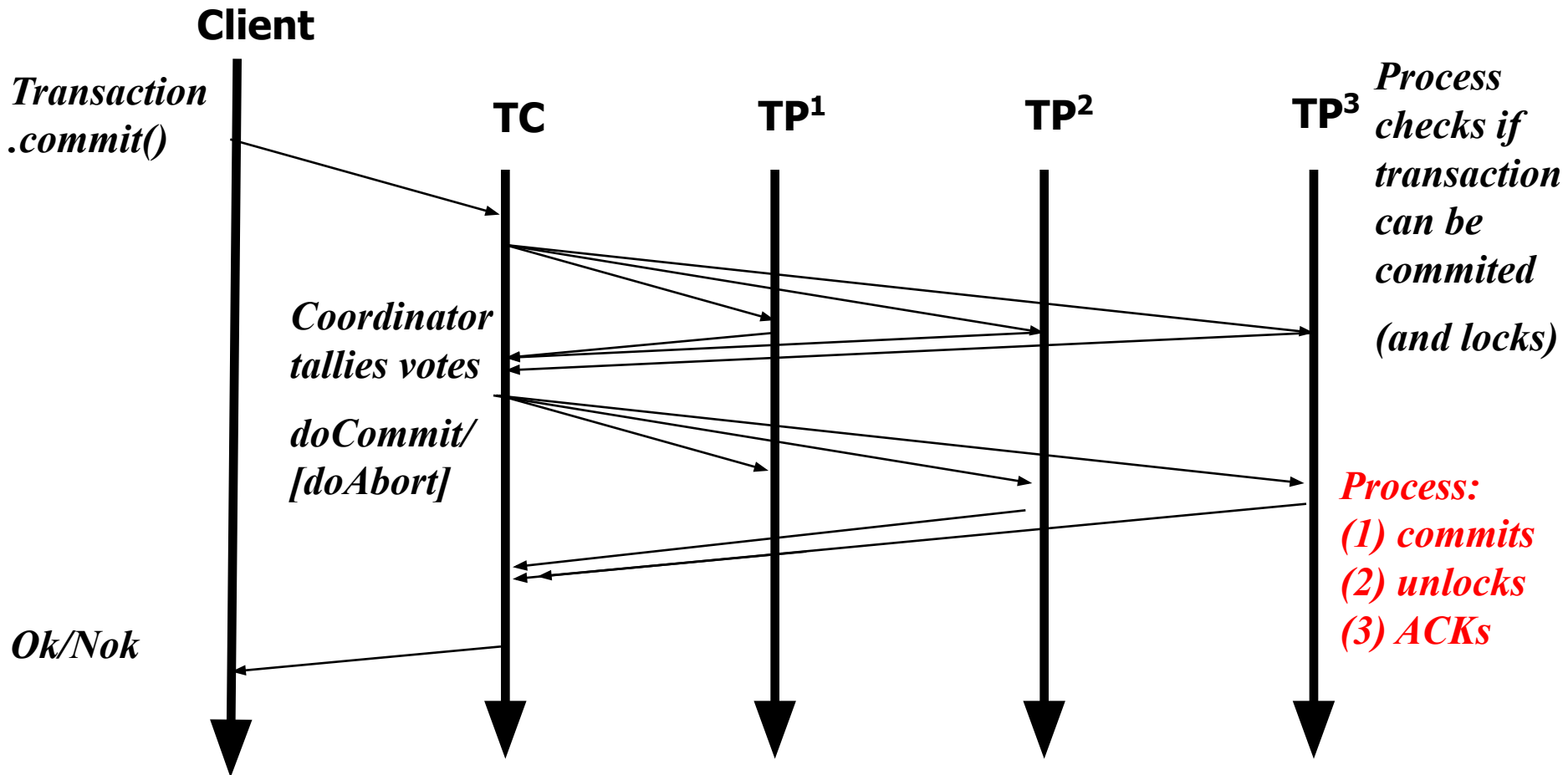
2-Phase Commit Protocol Illustrated

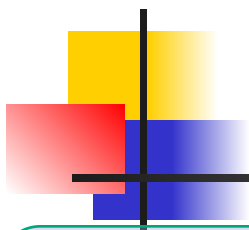


2-Phase Commit Protocol Illustrated

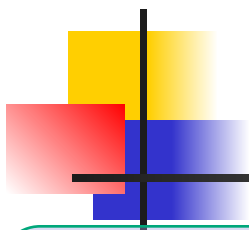


We have skipped the "client"





- **Atomic: All or nothing**
 - State shows either all the effects of a transaction, or none of them:
- **Consistent** (think of it as **C**orrect): **Guarantees basic properties**
 - A transaction moves the database from a state where integrity holds, to another where integrity holds (given constraint, triggers, etc)
- **Isolated: Each transaction runs as if alone**
 - Effect of concurrent transactions is the same as transactions running one after another (ie looks like batch mode)
- **Durable: Cannot be undone**
 - Once a transaction has committed, it can not be undone in spite of failures.



- **Atomic: All or nothing**
 - State shows either all the effects of a transaction, or none of them:
- **Consistent** (think of it as **C**orrect): **Guarantees basic properties**
 - A transaction moves the database from a state where integrity holds, to another where integrity holds (given constraint, triggers, etc)
- **Isolated: Each transaction runs as if alone**
 - Effect of concurrent transactions is the same as transactions running one after another (ie looks like batch mode)
- **Durable: Cannot be undone**
 - Once a transaction has committed, it can not be undone in spite of failures.

Example

- If not isolated and atomic:
- might overdraw the src account
- might "create" or "destroy" money

transfer (X@bank A, Y@bank B, \$20)

Suppose initially: X.bal = \$100

Y.bal = \$3

```
int transfer(src, dst, amt) {  
    transaction = begin();  
    if (src.bal > amt) {  
        src.bal -= amt;  
        dst.bal += amt;  
        return transaction.commit();  
    } else {  
        transaction.abort();  
        return ABORT;  
    }  
}
```

For simplicity, assume the client code looks like this:

```
int transfer(src, dst, amt) {  
    transaction = begin();  
    src.bal -= amt;  
    dst.bal += amt;  
    return transaction.commit();  
}
```

The banks can unilaterally decide to COMMIT or ABORT transaction



Problems to avoid

```
int transfer(src, dst, amt) {  
    transaction = begin();  
    src.bal -= amt;  
    dst.bal += amt;  
    return transaction.commit();  
}
```

- Lost updates
 - Another transaction overwrites your change based on a previous value of some data
- Inconsistent retrievals
 - You read data that can never occur in a consistent state
 - E.g., partial writes by other transactions, writes by a transaction that later aborts



One bad solution

```
int transfer(src, dst, amt) {  
    transaction = begin();  
    src.bal -= amt;  
    dst.bal += amt;  
    return transaction.commit();  
}
```

- A global lock
- Let only one transaction run at a time
 - make changes permanent on commit or undo changes on abort, if necessary



Better ...

```
int transfer(src, dst, amt) {  
    transaction = begin();  
    src.bal -= amt;  
    dst.bal += amt;  
    return transaction.commit();  
}
```

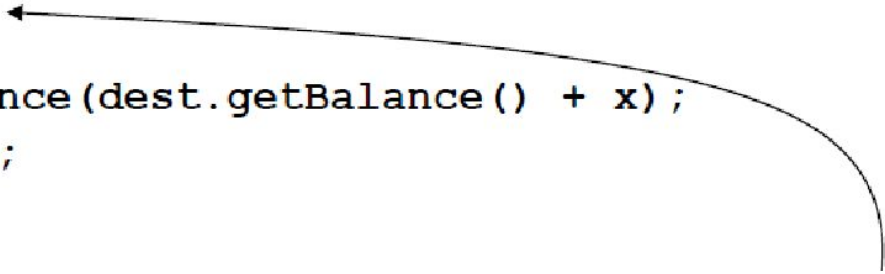
- Lock objects independently
 - E.g., one lock for the src account, one lock for the dest account
 - Other transactions can execute concurrently, as long as they don't read or write the src or dest accounts
- Fairly easy to implement with the tools we have



Locks alone are insufficient

- [... you need to use them correctly 😊]

```
bool xfer(Account src, Account dest, long x) {  
    lock(src);  
    if (src.getBalance() >= x) {  
        src.setBalance(src.getBalance() - x);  
        unlock(src);  
        lock(dest);  
        dest.setBalance(dest.getBalance() + x);  
        unlock(dest);  
        return TRUE;  
    }  
    unlock(src);  
    return FALSE;  
}
```





What's the next problem?

- ...deadlocks!

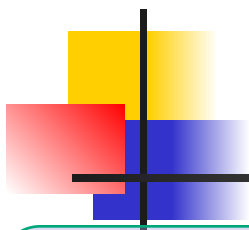
- Preventing deadlocks

- Each transaction gets all locks at once
- Each transaction gets all locks in a predefined order
- Impractical: transactions often do not know in advance what locks they will need



So?

- Detecting deadlock
 - Build a 'wait-for' graph
 - Each vertex is a transaction.
 - $T1 \square T2$ if $T1$ waits for a lock held by $T2$
 - If cycles then you have deadlock
 - Abort one of the transactions in the cycle
- “Ignoring” deadlocks
 - Automatically abort a long transaction
 - Not a bad strategy if you expect transactions to be short
 - (A long-running “short” transaction is likely a deadlock)



- **Atomic: All or nothing**
 - State shows either all the effects of a transaction, or none of them:
- **Consistent** (think of it as **C**orrect): **Guarantees basic properties**
 - A transaction moves the database from a state where integrity holds, to another where integrity holds (given constraint, triggers, etc)
- **Isolated: Each transaction runs as if alone**
 - Effect of concurrent transactions is the same as transactions running one after another (ie looks like batch mode)
- **Durable: Cannot be undone**
 - Once a transaction has committed, it can not be undone in spite of failures.



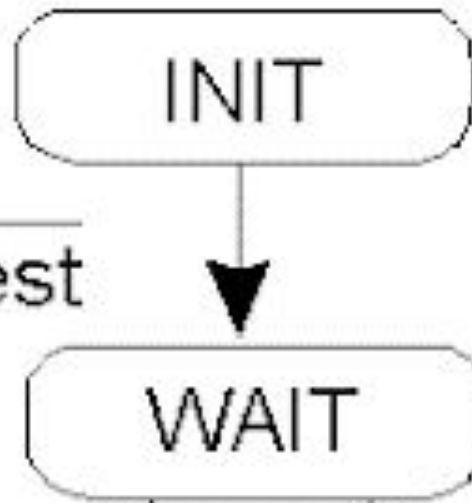
Let's have an eye on durable part:

- key issue: maintain transaction decision (commit / abort) in spite of failures (e.g., timeouts / reboots)
- Worth thinking about: how would a world where transactions / agreement is NOT durable
 - Blockchain world: transactions are not durable – their 'finality' is probabilistic

Finite State Machine: Notations

*Message
received*

Commit
Vote-request



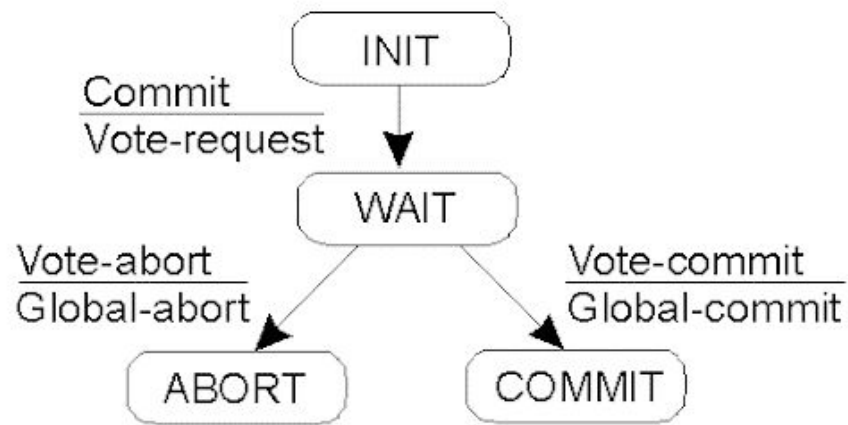
State

Message sent

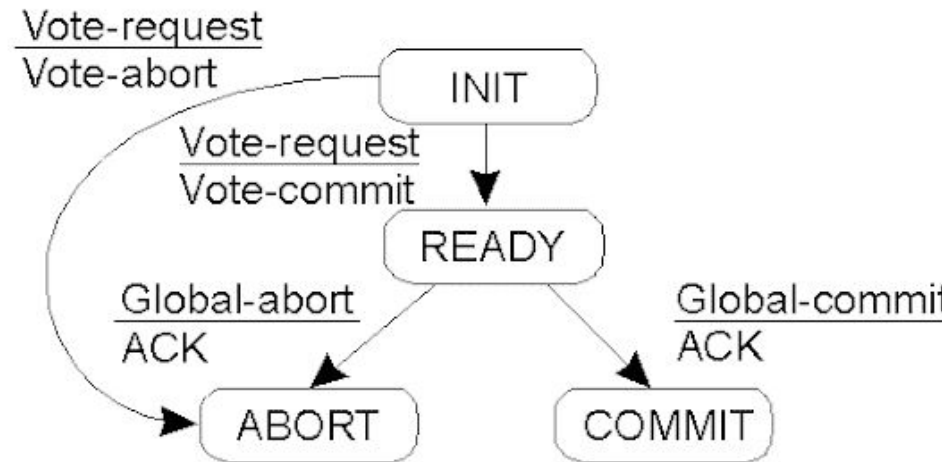


Finite State Machine: Participant?

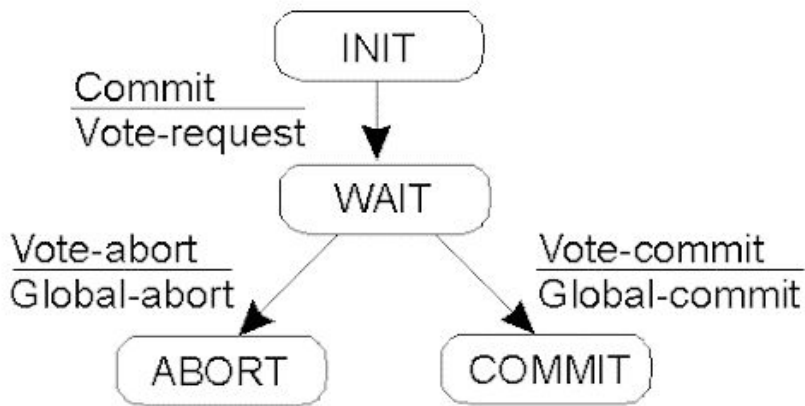
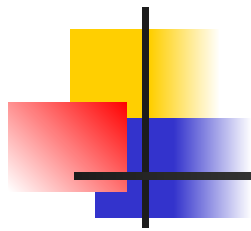
Two-Phase Commit – Finite State Machines



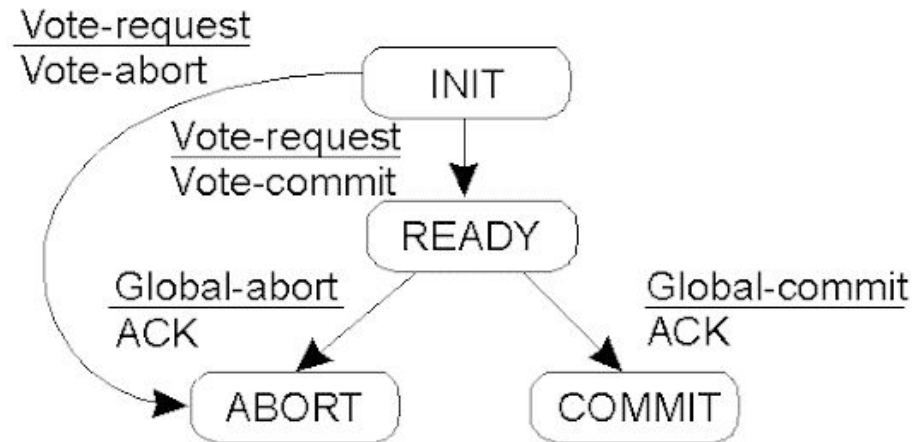
Coordinator (TC)



Participant (TP)



Coordinator (TC)



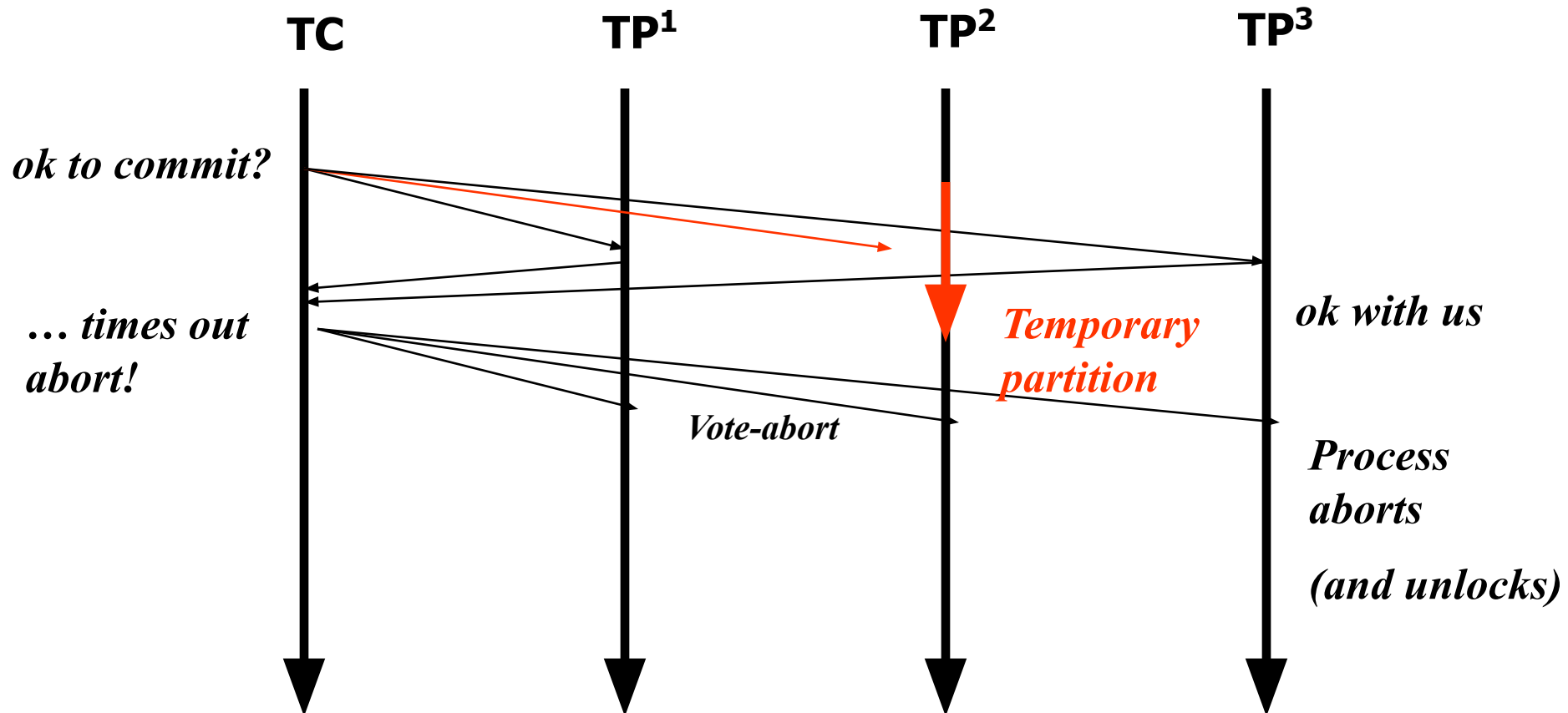
Participant (TP)

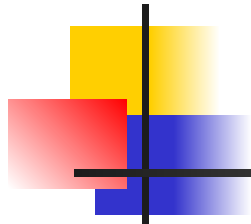
Observation: Blocking protocol

- in WAIT and READY states: progress is made by collaborating with others.
- Potentially slow / What if failures ...?

Goal: continue to operate after network and node failures

Impact of network failure ...





- Goal: **all-or-nothing outcome**
 - All will perform the action only, if all vote in favor;
if any votes against (**or does not vote**), all will “abort” (none will take the action)
 - with: **isolation** and **durability**
(all in the presence of faults)

What’s the fault model?

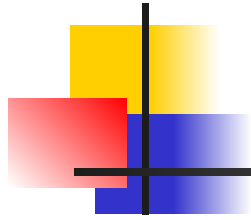
Fault model

Different safety/liveness properties are possible under different failure models!

- (1) Time bounds on communication / progress rate
 - **synchronous systems**: machines and networks can only be delayed by a bounded time
 - i.e., using a sufficiently large timeout, you can tell **with certainty** whether the machine crashed or the network is just slow
 - ***asynchronous systems**: machines and networks can be arbitrarily delayed
 - no way to tell whether a machine has crashed or is just slow
- (2) What happens after a crash? ***Fail-recover** vs. fail-stop
 - **Fail-recover**: Machines can crash ... but will reboot (and will be ok/fixed)
 - **Assumption 1**: disks cannot fail
 - **Assumption 2**: failures are not 'hard' - i.e., reboot from bug, power loss cause reboot
- (3) Trust: Are there 'byzantine' faults?

Makes sense in the context of a "atomic commit" (fail-stop for "consensus"?)

* □ **our failure model in what's next**



We'll assume:
asynchronous
fail-recover
no byzantine

Desired properties of a solution:

■ Safety (correctness)

- All nodes agree on the same value (action)
- The agreed value has been proposed by some node
(the one proposed by the coordinator)
- [when failures] Never forget the decision taken

■ Liveness (availability)

- [If less than some fraction of nodes crash] the correctly operating nodes should reach agreement and the system should operate normally

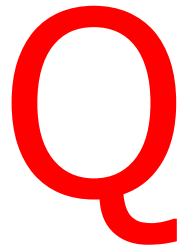
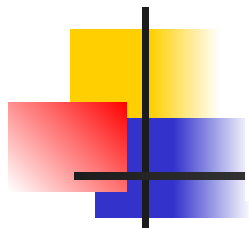
Implementation mechanisms at two levels:

- **Local system:** we assume write-ahead logging.
 - A node persists: transaction workspace + state machine change
- **Distributed system:** Coordination between nodes (our focus)



Failure Recovery in 2PC

- Two cases:
 - Recovery after crashes/reboots
 - Recovery after timeouts
 - Network (or nodes!) can be very slow
 - TC times-out waiting for participants' votes
 - TP times-out waiting for decision by TC



All nodes must log protocol progress
What and when does TC log to disk?
What and when does TP log to disk?

- Nodes cannot back out if commit was decided

- Examples

- TC crashes just after
 - Cannot forget about
- TP crashes after send
 - Cannot forget about

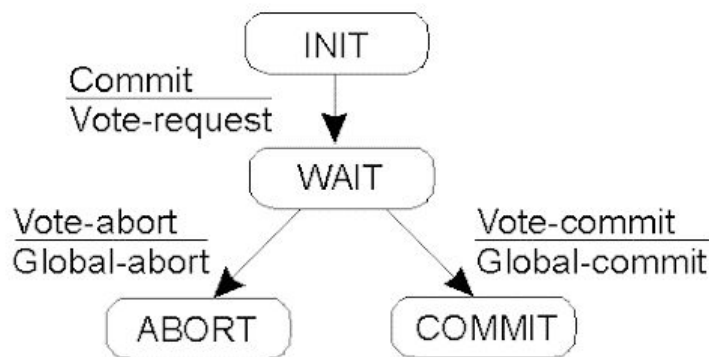
Log the state-changes (new state)

Coordinator: INIT, WAIT, COMMIT/ABORT

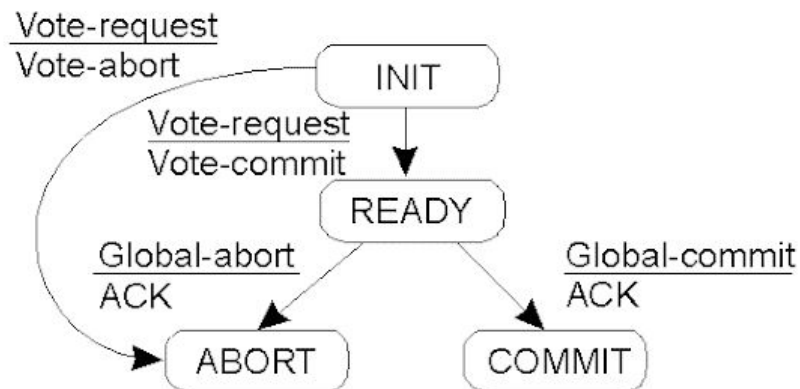
Participants: INIT, READY, COMMIT/ABORT

Make sure messages are idempotent!

In recovery, resend whatever message was next



Coordinator



Participant

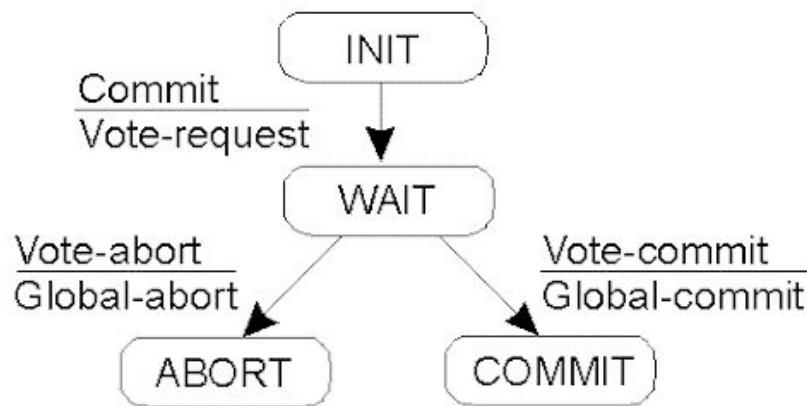
Recovery Upon Reboot

TC: If TC finds "COMMIT", then resend 'global-commit',

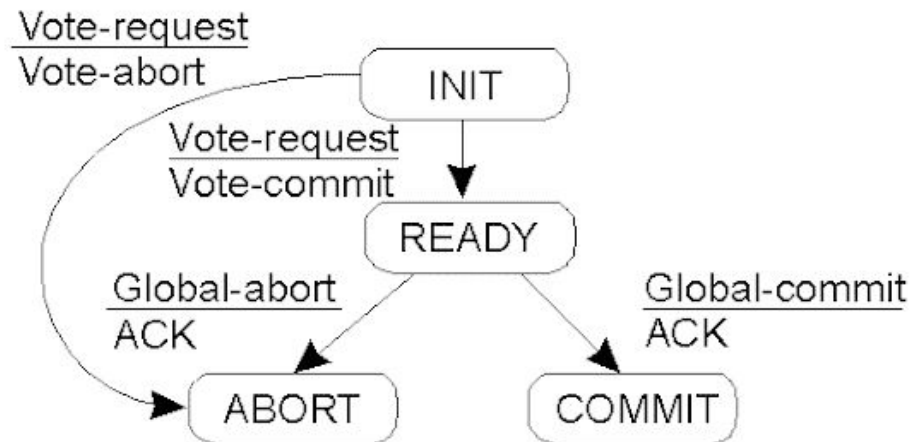
else abort

TP: If TP finds 'INIT' on disk, then abort

If TP finds 'READY', then run *termination protocol*



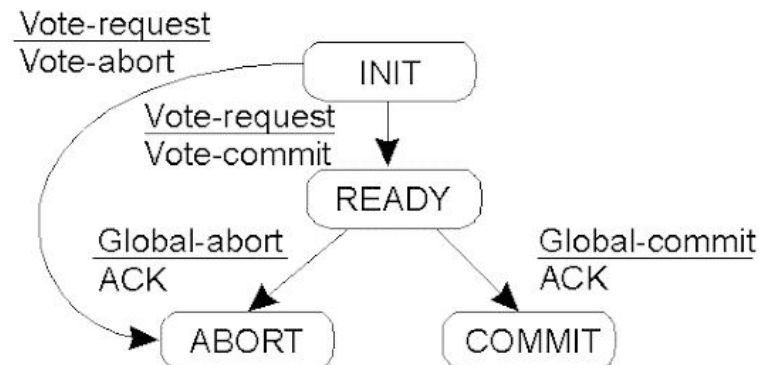
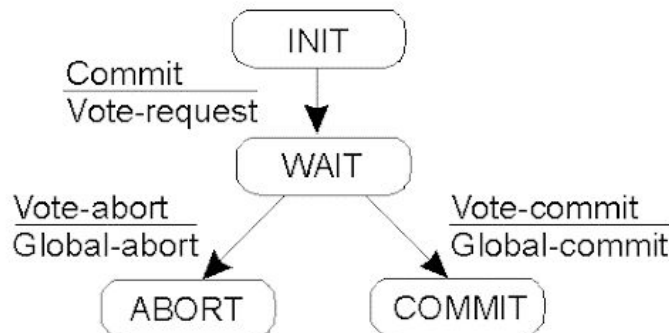
Coordinator



Participant

Handling Timeouts

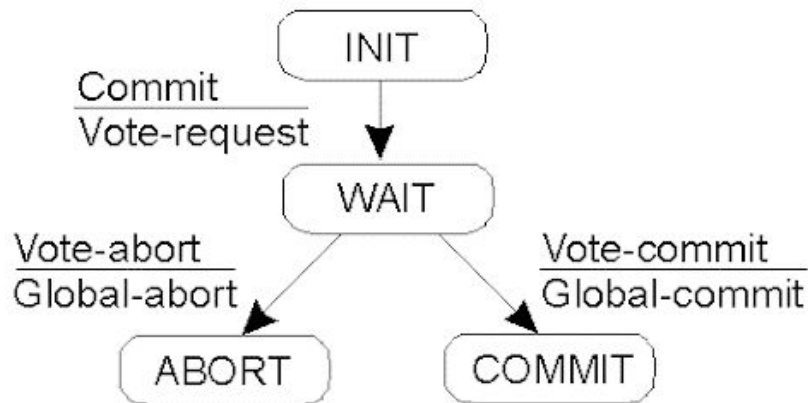
- Two blocking states that may lead to timeouts:
 - TC in **WAIT**: times out waiting for a TP response
 - TP in **READY**: times out waiting for TC's decision
- Note: timeouts aren't necessarily due to network problems
 - They could be due to slow, overloaded hosts



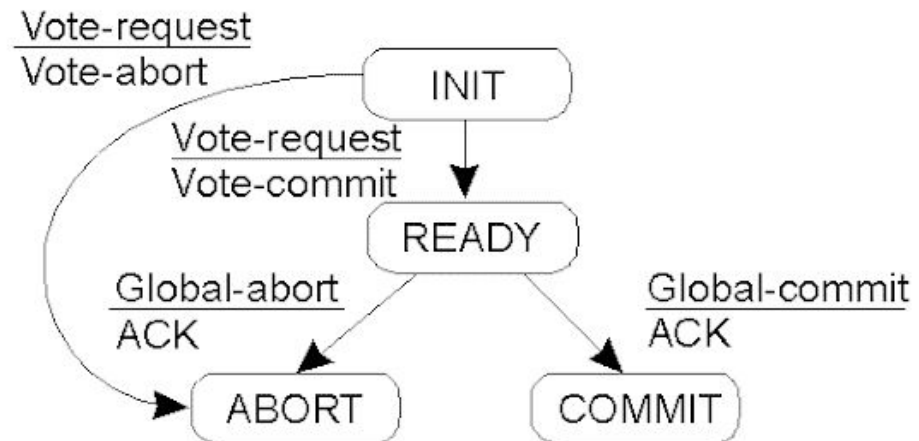
Handling Timeouts @ Coordinator?

- What can TC do if timeout?
 - (the only waiting TC state is 'WAIT')

ABORT!



Coordinator

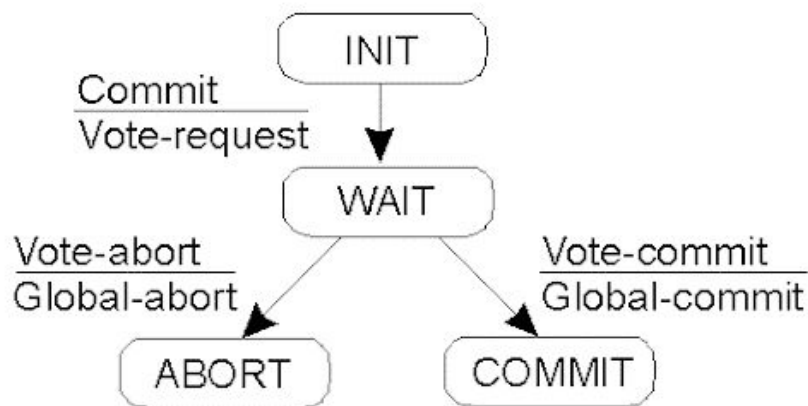


Participant

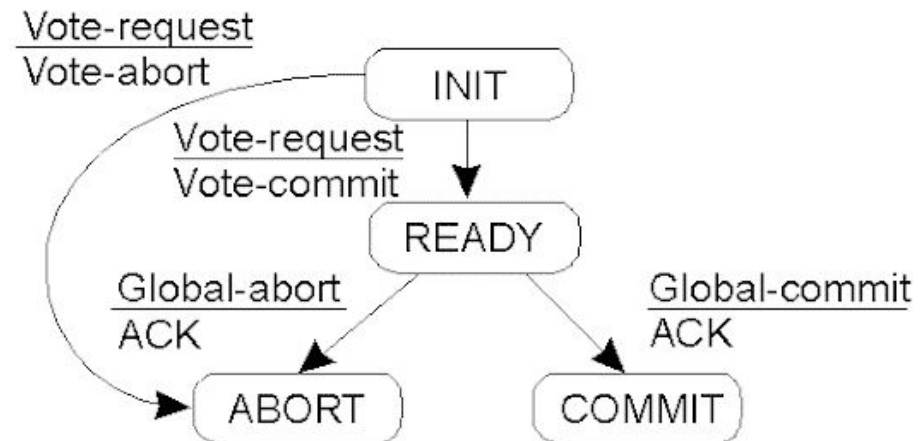
What if additional info is available?

- TP² has received global-commit/abort
- TP² responded with ABORT
- TP² has not responded yet
- TP² responded with COMMIT

- TP¹ times out in READY state waiting for TC “Global-commit / Global-abort” response
 - Can TP¹ unilaterally decide to commit?
 - Can TP¹ unilaterally decide to abort?

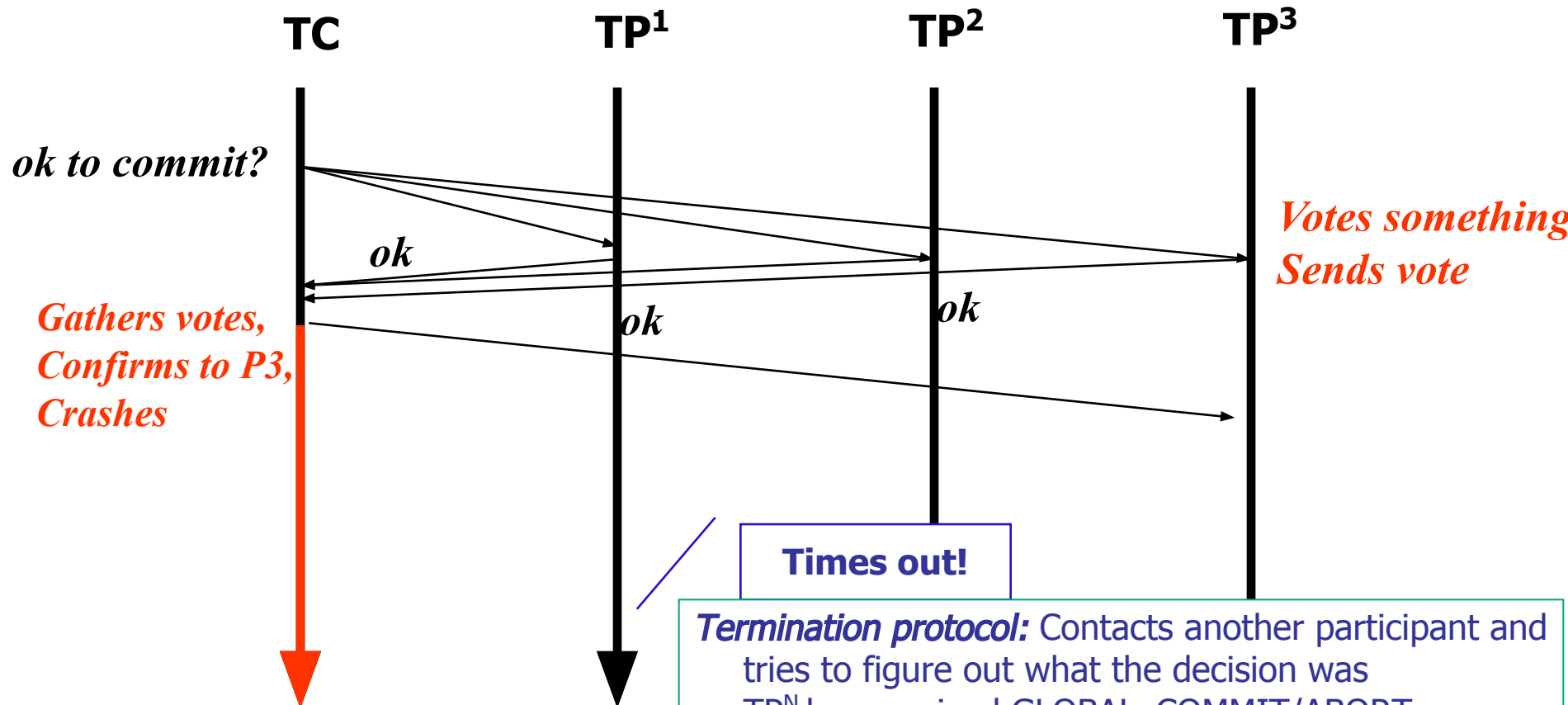


Coordinator



Participant

Example of a hard scenario

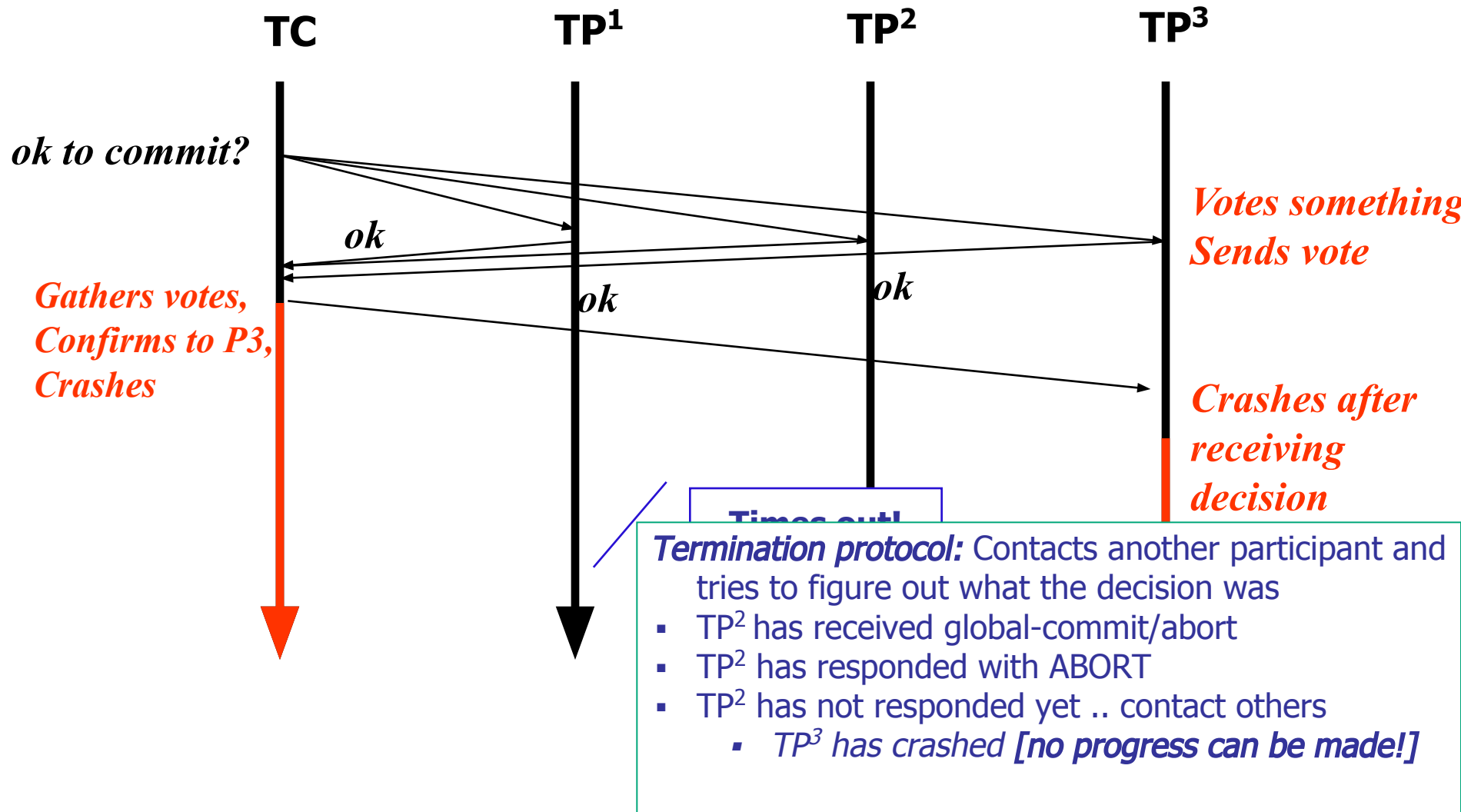


Times out!

Termination protocol: Contacts another participant and tries to figure out what the decision was

- TP^N has received GLOBAL_COMMIT/ABORT
- TP^N has responded with ABORT
- TP^N has not responded yet
- TP^N has responded with COMMIT [no progress can be made! Let's contact others]

Example of a hard scenario





Possible termination protocol

- Execute termination protocol if TP^1 times out on TC and has voted “yes” (and timed-out or recovered after failure)
 - TP^1 sends “status” message to TP^2
 - If TP^2 has received “commit”/“abort” from TC ...
 - If TP^2 has responded to TC with “no”, ...
 - If TP^2 has not responded to TC, ...
 - If TP^2 has responded with “yes”, ...
- Resolves most failure cases **except** sometimes when TC + one TP fail
 - If TC is also participant, as it typically is, then this protocol is vulnerable [it blocks] to a single-node failure (the TC’s failure)!



Summary so far.

Desired properties of a solution:

- **Safety** (correctness)

- All nodes agree on the same value
- The agreed value X has been proposed by some node

- **Liveness** (availability)

- [If less than some fraction of nodes fail] the correctly operating nodes reach agreement and the system operates normally

2PC: can block even when one (or a few) machines fail (i.e., the whole system can not make progress during the failure).

- **Safety** (correctness) – YES;
- **Liveness** (availability) – NO, in some failure cases

3PC: liveness but at the cost of safety (issues only with network partitions)

Paxos: fixes the problem in most cases in you'll see in practice



Summary so far



Two 'flavors' for the distributed agreement problem:

Participants need to agree on a value / action ...

- [consensus problem] ... and they are willing and capable to accept any value.
- [atomic commitment problem] ... and they have specific constraints on whether they can accept any particular value

We have focused on this so far

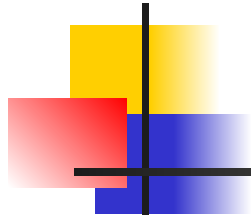


Fault model

- (1) Time bounds on communication / progress rate
 - **synchronous systems**: machines and networks can only be delayed by a bounded time
 - Consequence: using a sufficiently large timeout, you can tell **with certainty** whether the machine crashed or the network is just slow
 - ***asynchronous systems**: machines and networks can be arbitrarily delayed
 - no way to tell whether a machine has crashed or is just slow
- (2) What happens after a crash? ***Fail-recover** vs. fail-stop
 - **Fail-recover**: Machines can crash ... but will reboot (and will be ok/fixed)
 - **Assumption 1**: disks cannot fail
 - **Assumption 2**: failures are not 'hard' - i.e., reboot from bug, power loss cause reboot
- (3) Trust: Are there 'byzantine' faults?

Makes sense in the context
of a "atomic commit"
(fail-stop for "consensus"?)

*** our failure model in what's next**



We assumed:
asynchronous
fail-recover
no byzantine

Desired properties of a solution:

■ Safety (correctness)

- All nodes agree on the same value (action)
- The agreed value has been proposed by some node
(the one proposed by the coordinator)
- [when failures] Never forget the decision taken

■ Liveness (availability)

- [If less than some fraction of nodes crash] the correctly operating nodes should reach agreement and the system should continue to operate normally

Different safety/liveness properties are possible under different failure models!

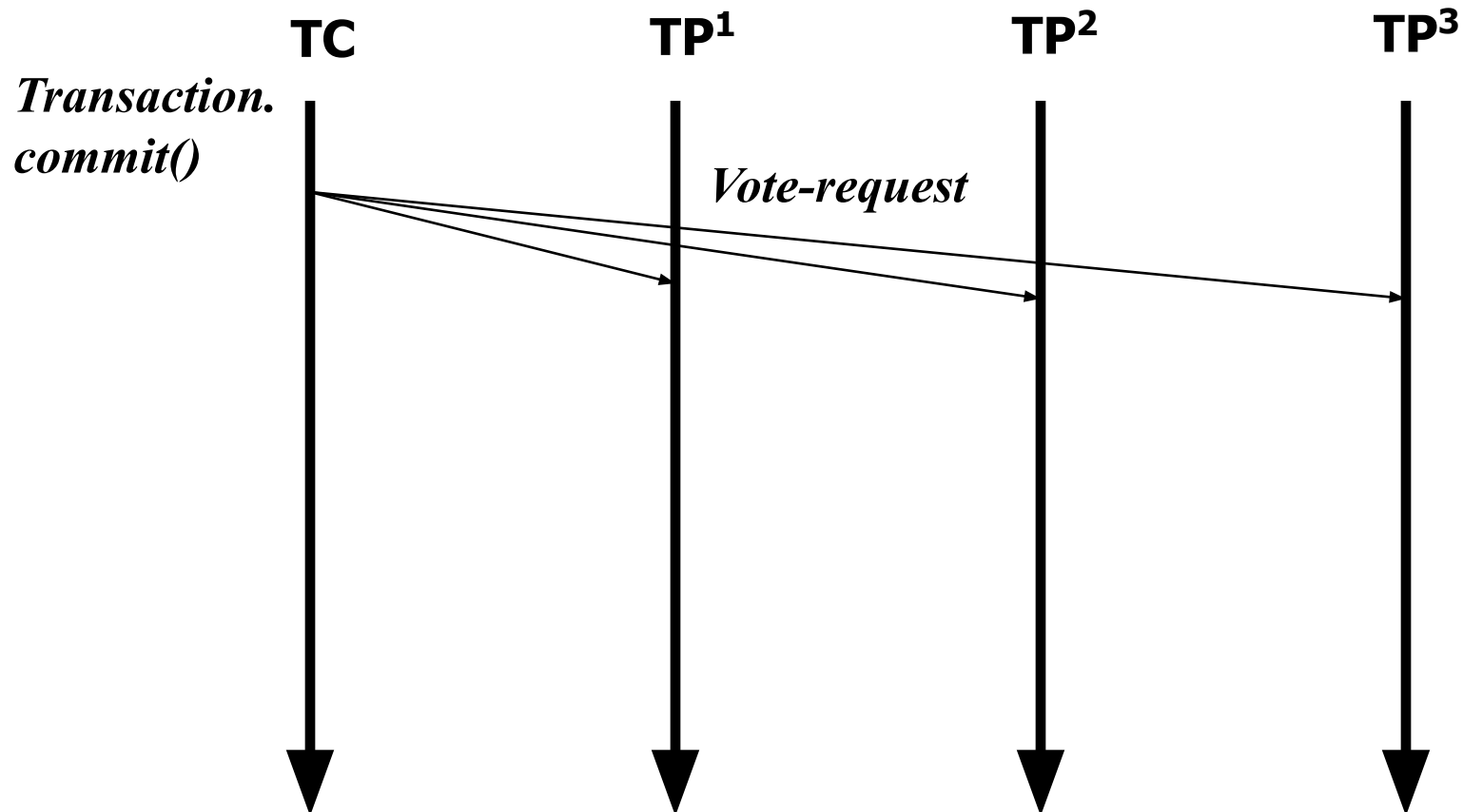


Typical protocol: two-phase commit (2PC)

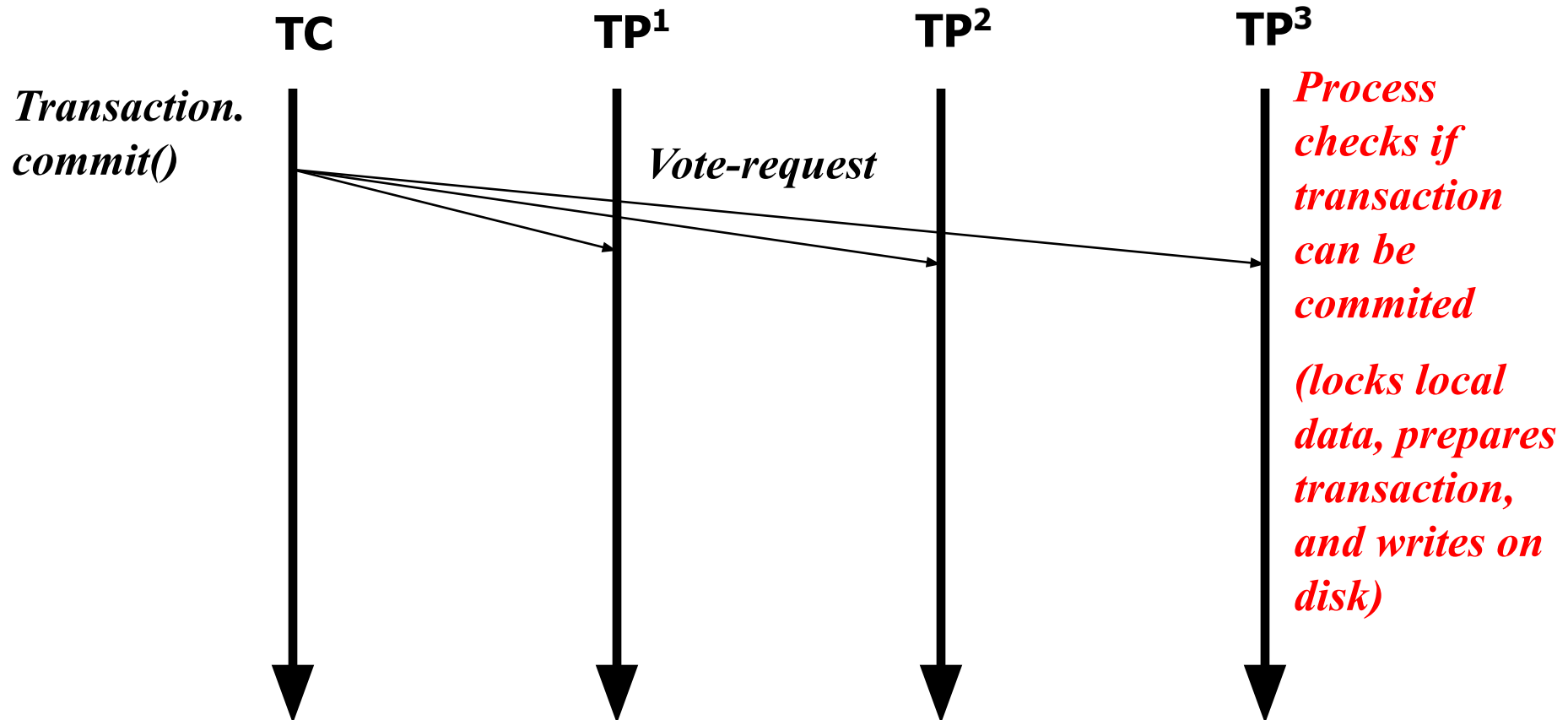
Implementation mechanisms at two levels:

- **Local system:** write-ahead logging.
 - A node persists: transaction workspace + state machine transition
- **Distributed system:** coordination between nodes

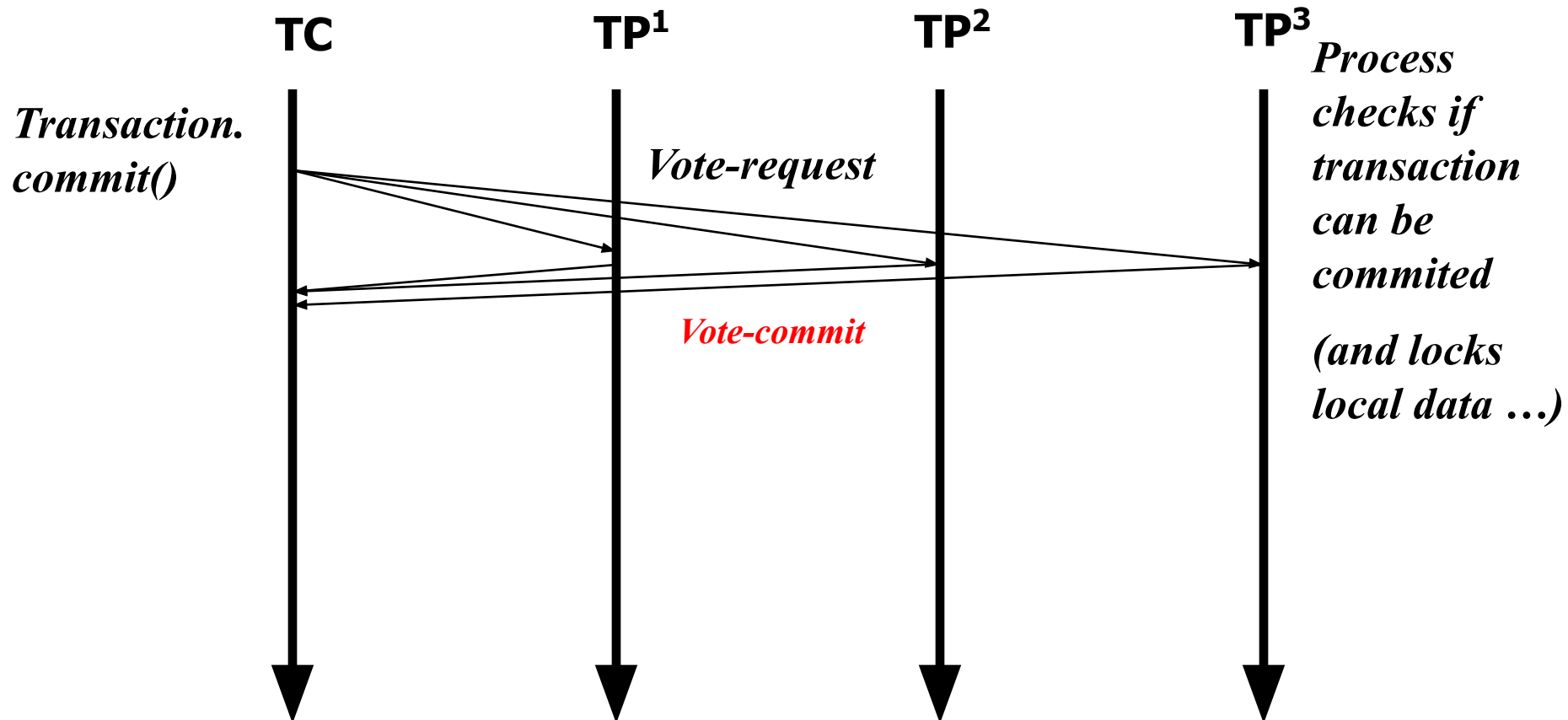
2-Phase Commit Protocol Illustrated



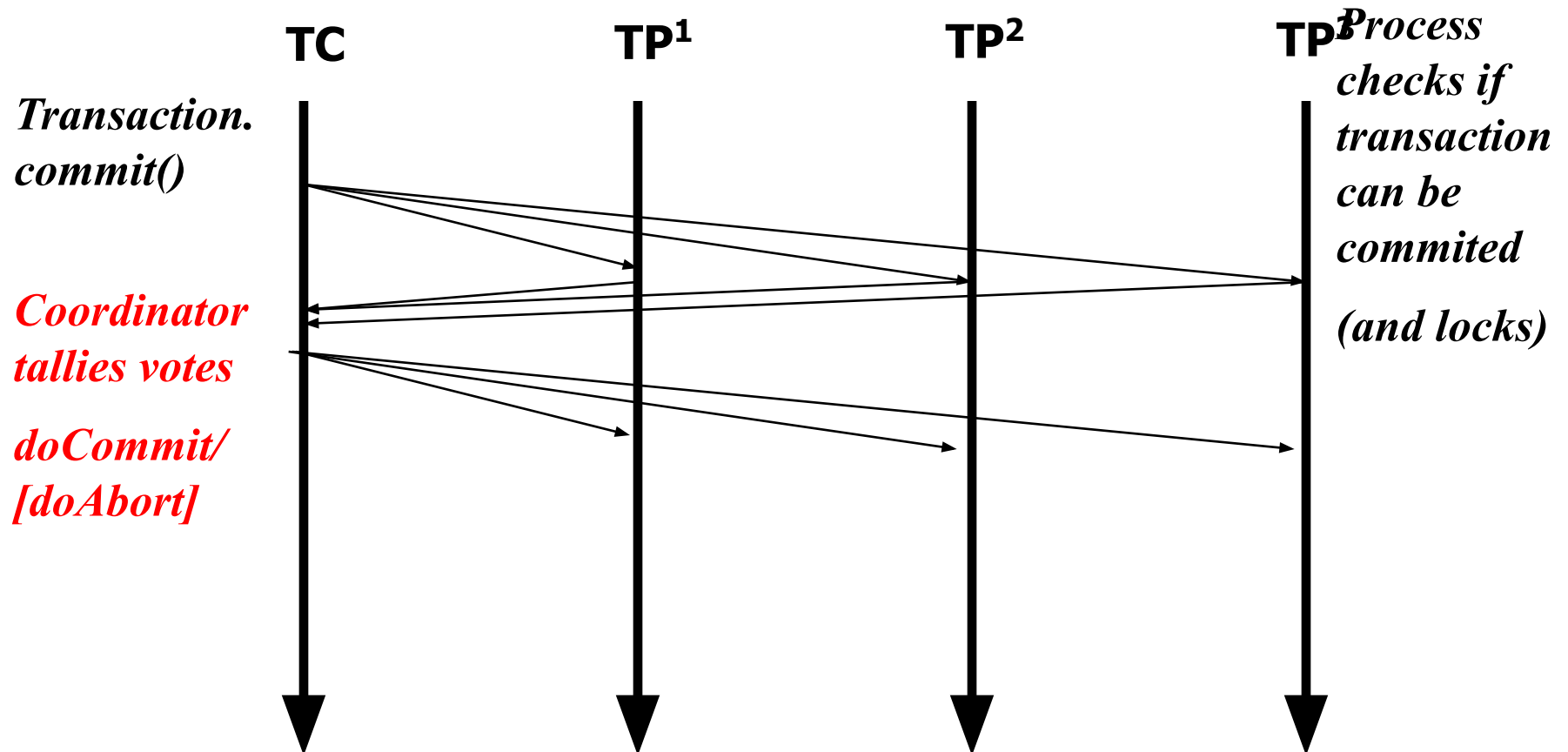
2-Phase Commit Protocol Illustrated



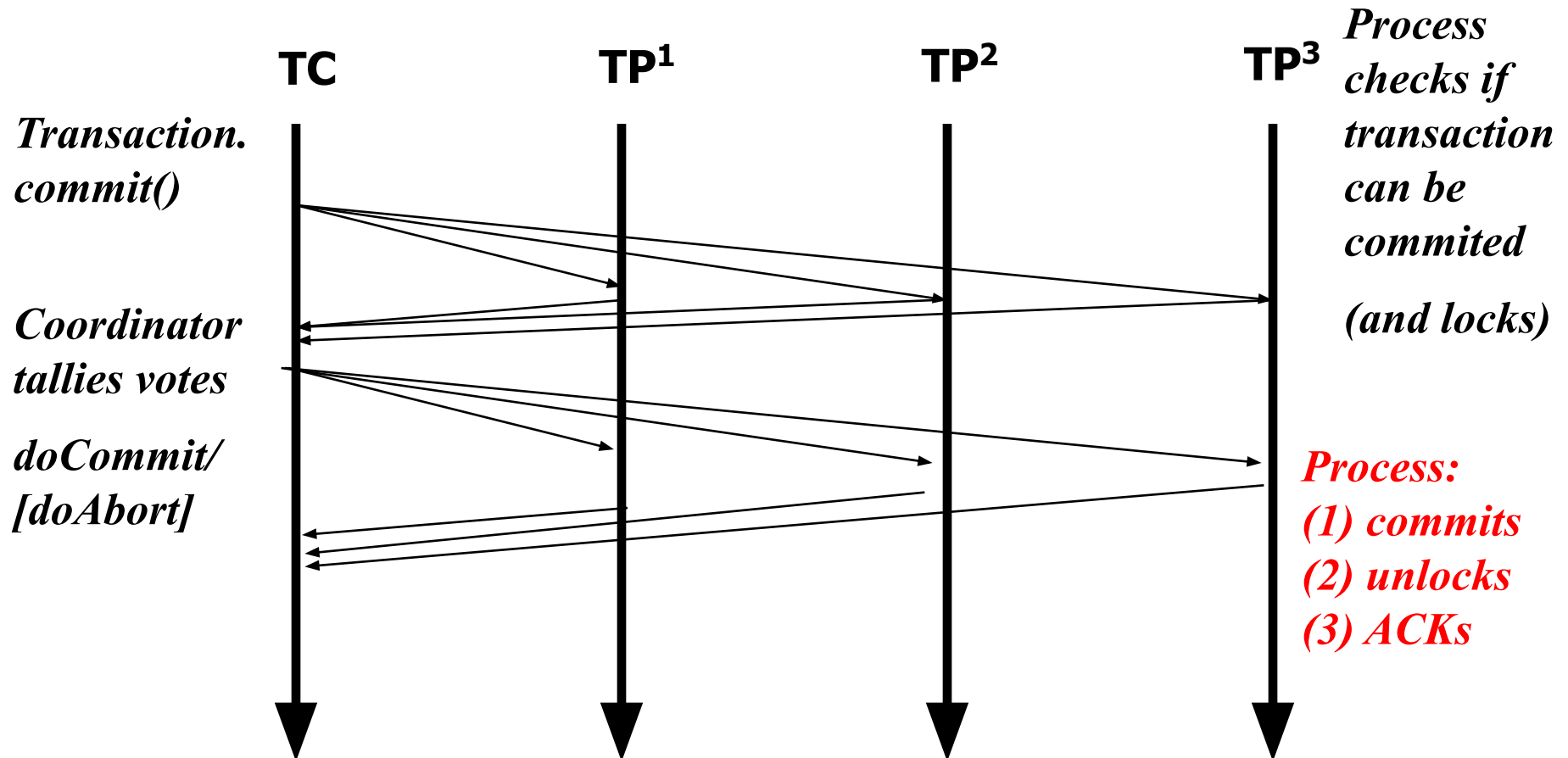
2-Phase Commit Protocol Illustrated



2-Phase Commit Protocol Illustrated



2-Phase Commit Protocol Illustrated



We have skipped the "client"

Client

*Transaction
.commit()*

TC

TP¹

TP²

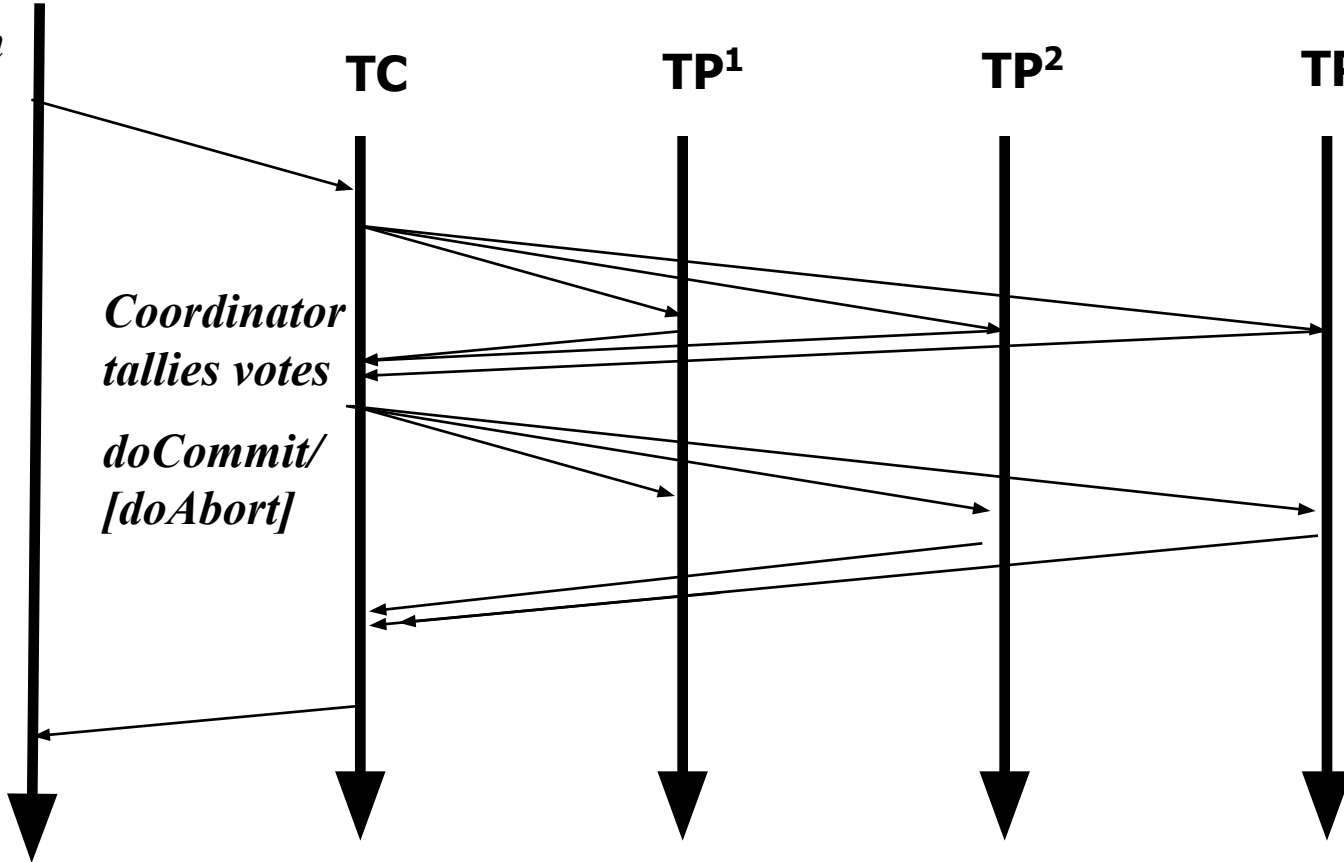
TP³

*Process
checks if
transaction
can be
committed
(and locks)*

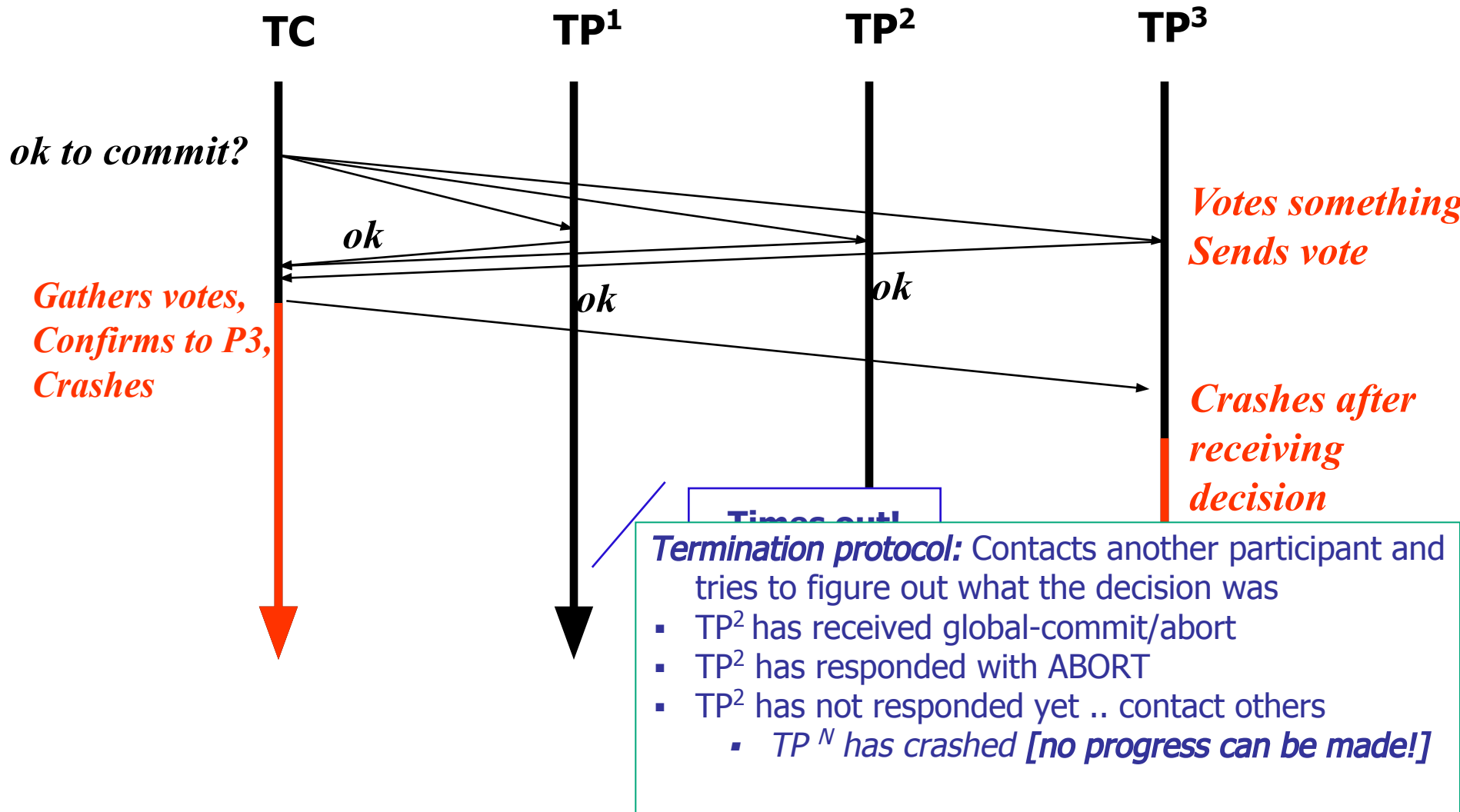
*Coordinator
tallies votes
doCommit/
[doAbort]*

***Process:**
(1) commits
(2) unlocks
(3) ACKs*

Ok/Nok



Example of a hard scenario





So what we've got

Desired properties of a solution:

- **Safety** (correctness)

- All nodes agree on the same value
- The agreed value X has been proposed by some node

- **Liveness** (availability)

- [If less than some fraction of nodes fail] the correctly operating nodes reach agreement and the system operates normally

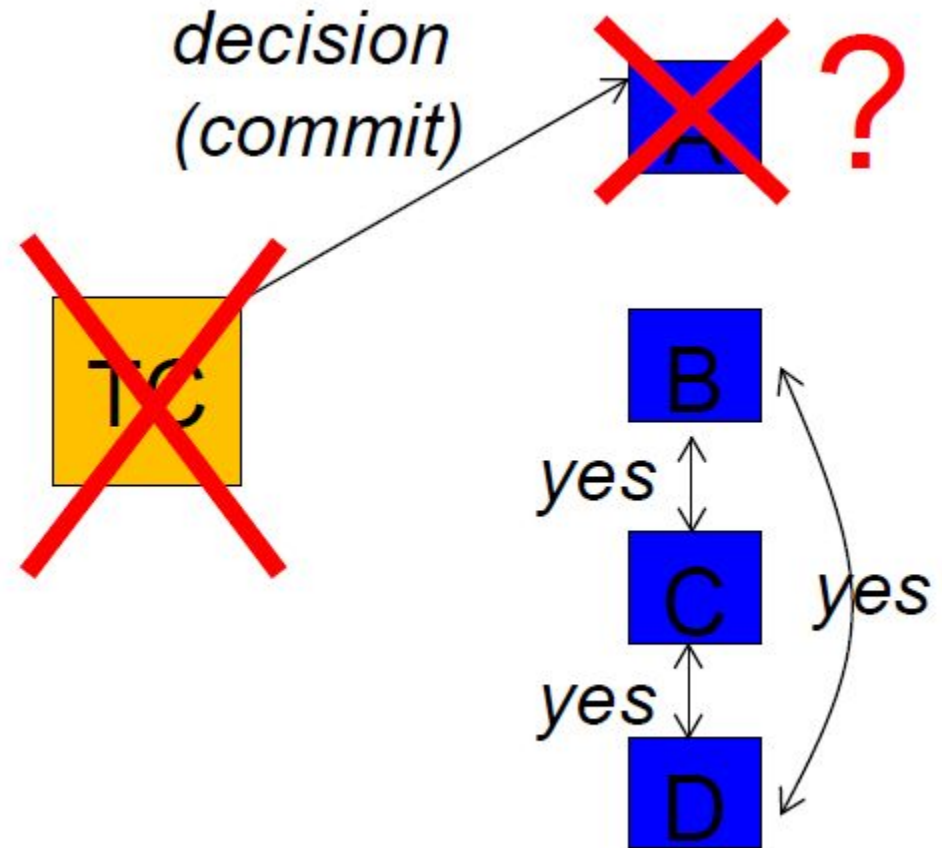
2PC: may block even when one (or a few) machines fail
(i.e., the whole system can not make progress during the failure).

- **Safety** (correctness) – YES;
- **Liveness** (availability) – NO (in some failure cases)

2PC: Safe but at the cost of liveness

- Does the fault model matter?
 - Async vs. sync
 - Fail-recover vs. fail-stop

Assuming fail-stop AND sync
then the protocol has no
liveness issues!



2PC: can block even when one (or a few) machines fail

(i.e., the whole system blocked -- can not make progress -- during the failure)

- **Safety** (correctness) – YES;
- **Liveness** (availability) – NO – in some failure cases

3PC: one attempt to fix liveness (but at the cost of safety)
(issues only with network partitions)

(we keep the initial fault model:

asynchronous

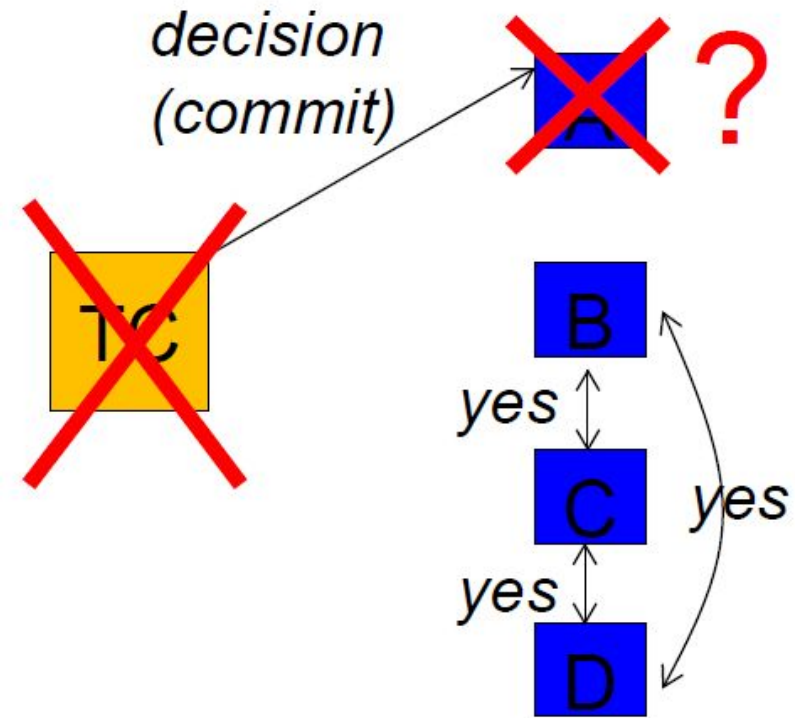
fail-recover

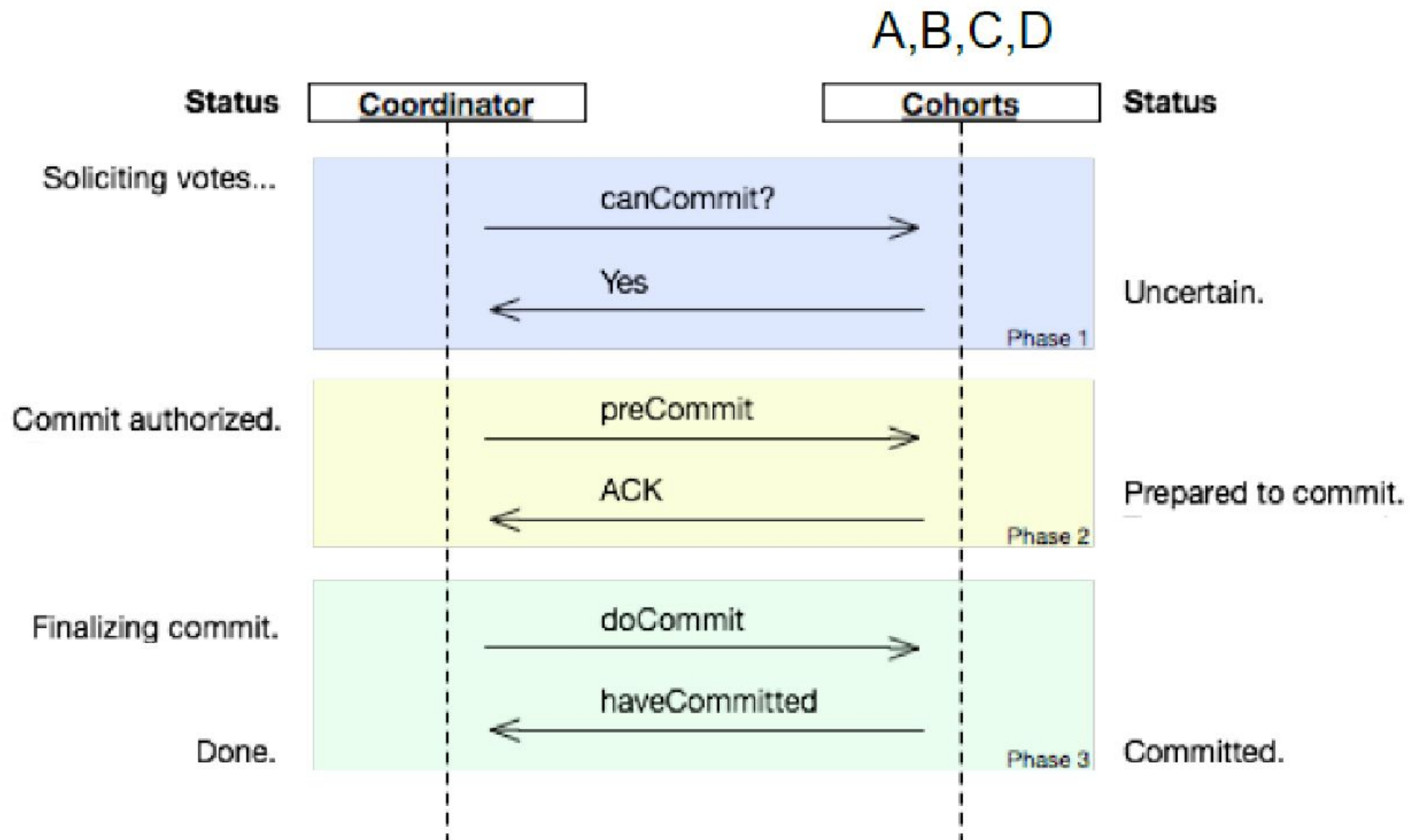
no byzantine

)

One attempt to fix: 3PC

- **Goal:** Turn 2PC into a live (non-blocking) protocol
 - 'force' progress after timeout!
 - our 3PC should never block on node failure as 2PC did
- **Insight:** 2PC suffers from allowing nodes to irreversibly commit an outcome *before ensuring that the others know the outcome, too*
- **Idea:** split the "commit/abort" phase into two phases
 - First communicate the outcome to everyone
 - Commit only after everyone knows the outcome





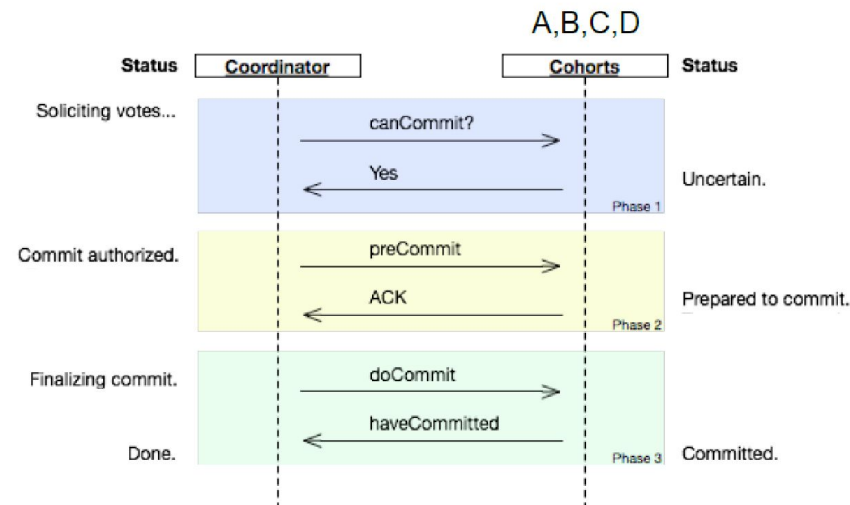
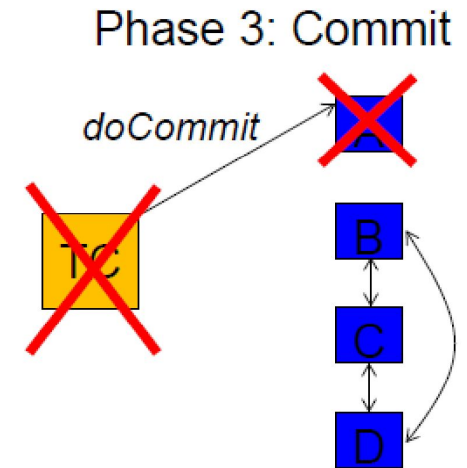
Does 3PC solve the blocking issue?

Assuming same scenario as before
(TC, A crash)

Can B/C/D reach a safe decision when
they time out?

- 1. If one of them has received preCommit, ...
- 2. If none of them has received preCommit

...



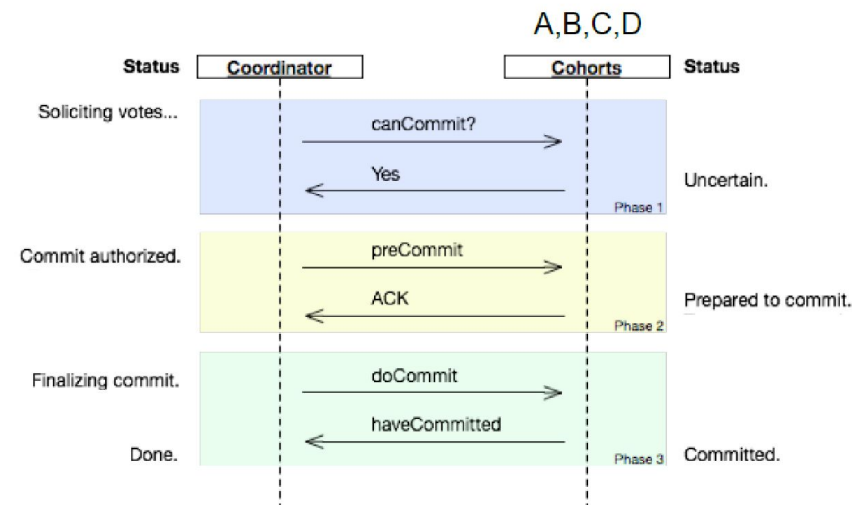
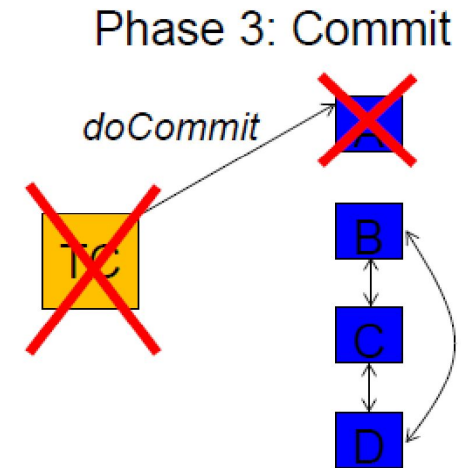
Does 3PC solve the blocking issue?

Assuming same scenario as before
(TC, A crash)

Can B/C/D reach a safe decision when
they time out?

- 1. If one of them has received preCommit, ...
can commit
- 2. If none of them has received preCommit, ...
abort

Blocking solved! 😊





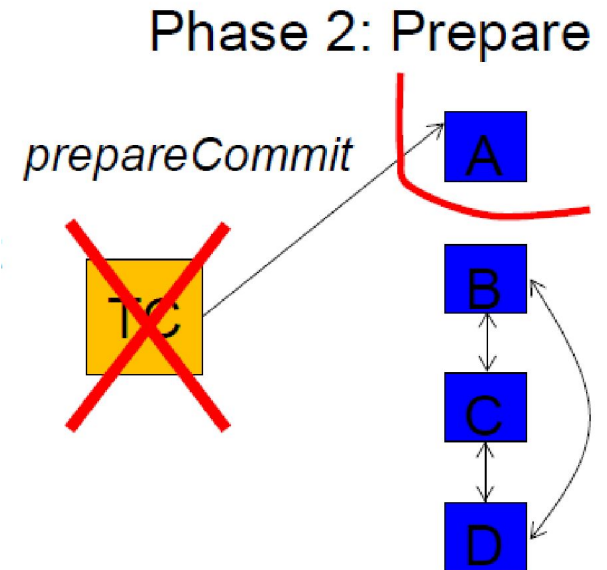
But is 3PC always safe (correct)?

- Liveness (availability): Yep
 - Doesn't block, it always makes progress by/after timing out
- Safety (correctness): ???
 - Can you think of scenarios in which 3PC would result in inconsistent decisions?
- Cases / network partition that may lead to unsafety:
 - A hasn't crashed, it's just offline [network partition]
 - TC hasn't crashed, it's just offline [network partition]

One example


One example scenario:

- A receives *prepareCommit* from TC
- Then,
 - A gets partitioned from B/C/D and
 - TC crashes
- On timeout:
 - None of B/C/D have not received *prepareCommit*, hence they all Abort
 - A is prepared to commit, hence, according to protocol, it unilaterally decides to commit



Similar scenario with partitioned, not crashed, TC

Note: this scenario is impossible in a snchroneous system



2PC: can block even when one (or a few) machines fail (i.e., the whole system can not make progress during the failure).

- **Safety** (correctness) – YES;
- **Liveness** (availability) – MOSTLY – except in some failure cases

3PC: liveness but at the cost of safety

- **Safety** (correctness) – MOSTLY except sometimes when partitions
- **Liveness** (availability) – YES

Can one design a protocol that's both safe and live?

It turns out that it's impossible in the most general case!

■ Fischer-Lynch-Paterson [FLP'85] Impossibility Result

It is impossible for a set of processors in an asynchronous system to agree on a binary value, [even if only a single process is subject to an unannounced failure]

The core of the problem is **asynchrony**

- makes it impossible to tell whether or not a machine has crashed (and therefore it will launch recovery and coordinate with you safely) or it is just unreachable now (and therefore it's running separately from you, potentially doing stuff in disagreement with you)

[For **synchronous systems**, 3PC can be made to guarantee both safety and liveness!

- When you know the upper bound of message delays, you can infer when something has crashed with certainty

]



FLP – Translation

- What FLP says: [in the general case] you can't guarantee both safety and progress when there is even a single fault at an inopportune moment
- What FLP doesn't say: in practice, how close can you get to the ideal (safe and live)?

Next: Paxos algorithm, which in practice gets close



Two 'flavors' for the distributed agreement problem:

Participants need to agree on a value / action ...

- [consensus problem] ... and they are willing and capable to accept any value.
- [atomic commitment problem] ... and they have specific constraints on whether they can accept any particular value.

■ Paxos

- One of the few **safe** and **largely-live** agreement protocols
- All nodes agree on the same value despite node failures, network failures, and delays
 - Only blocks in exceptional circumstances that are rare in practice

■ Usage:

- Locks, coordinator election, next operation, next version, RSM
- Examples:
 - Google: Chubby (Paxos-based distributed lock service)
 - Most Google services use Chubby directly or indirectly
 - Yahoo: Zookeeper (Paxos-based distributed lock service)
 - MSR: Frangipani (Paxos-based distributed lock service)


vs. RAFT vs. B-PAXOS



10,000 feet view

Two Core Differentiating Mechanisms

- 1. Majority voting
 - 2PC needs all nodes to vote 'Yes' before committing
 - As a result, 2PC may block when a single node fails
 - Paxos requires only a majority of the acceptors ($\text{half} + 1$) to accept a proposal
 - As a result, in Paxos up to half the nodes can fail to reply and the protocol continues to work correctly
 - Moreover, since no two majorities can exist simultaneously, network partitions do not cause problems (as they did for 3PC)
- 2. Proposal ordering
 - Lets nodes decide which of several concurrent proposals to accept and which to reject



2PC: can block even when one (or a few) machines fail (i.e., the whole system can not make progress during the failure).

- **Safety** (correctness) – YES;
- **Liveness** (availability) – MOSTLY – except in some failure cases

3PC: liveness but at the cost of safety

- **Safety** (correctness) – MOSTLY *except when partitions*
- **Liveness** (availability) – YES

Paxos: fixes the problem for most cases you'll see in practice

- **Safety** (correctness) – YES
- **Liveness** (availability) – ALMOST ALWAYS
 - If less than half the nodes fail, the rest nodes reach agreement *eventually*



Paxos in more detail



Paxos

- Safety
 - Only a single value is chosen
 - Only a proposed value can be chosen
 - Only chosen values are learned by processes
- Liveness ***
 - Some proposed value **eventually** chosen if fewer than half of processes fail
 - If value is chosen, a process eventually learns it

Paxos: Roles of a Process

- Three conceptual roles
 - **Proposers** propose values
 - **Acceptors** accept values, where chosen if majority accept
 - **Learners** learn the outcome (chosen value)
- In reality, a process can play any/all roles

Strawman

- 3 proposers, 1 acceptor
 - Acceptor accepts first value received
 - No liveness on failure
- 3 proposers, 3 acceptors
 - Accept first value received, acceptors choose common value known by majority
 - But no such majority is guaranteed

Paxos

- Each acceptor accepts *multiple proposals*
 - Hopefully one of multiple accepted proposals will have a majority vote (and we determine that)
 - If not, rinse and repeat (more on this)
- How do we select among multiple proposals?
- **Ordering:** proposal is tuple (proposal #, value) = (n, v)
 - Proposal # strictly increasing, globally unique
 - Globally unique? Trick: set low-order bits to proposer's ID

Paxos Protocol Overview

- Proposers:

1. Choose a proposal number n
2. Ask acceptors if any accepted proposals with $n_a < n$
3. If existing proposal v_a returned, propose same value (n, v_a)
4. Otherwise, propose own value (n, v)

Note **altruism**: goal is to reach consensus, not “win”

- Acceptors try to accept value with highest proposal n
- Learners are passive and wait for the outcome

Paxos Phase 1

- Proposer:
 - Choose proposal number n , send $\langle \text{prepare}, n \rangle$ to acceptors
- Acceptors:
 - If $n > n_h$
 - $n_h = n$ ← promise not to accept any new proposals $n' < n$
 - If no prior proposal accepted
 - Reply $\langle \text{promise}, n, \emptyset \rangle$
 - Else
 - Reply $\langle \text{promise}, n, (n_a, v_a) \rangle$
 - Else
 - Reply $\langle \text{prepare-failed} \rangle$

Paxos Phase 2

- Proposer:

- If receive promise from majority of acceptors,
 - Determine v_a returned with highest n_a , if exists
 - Send $\langle \text{accept}, (n, v_a \parallel v) \rangle$ to acceptors

- Acceptors:

- Upon receiving (n, v) , if $n \geq n_h$,
 - Accept proposal and notify learner(s)

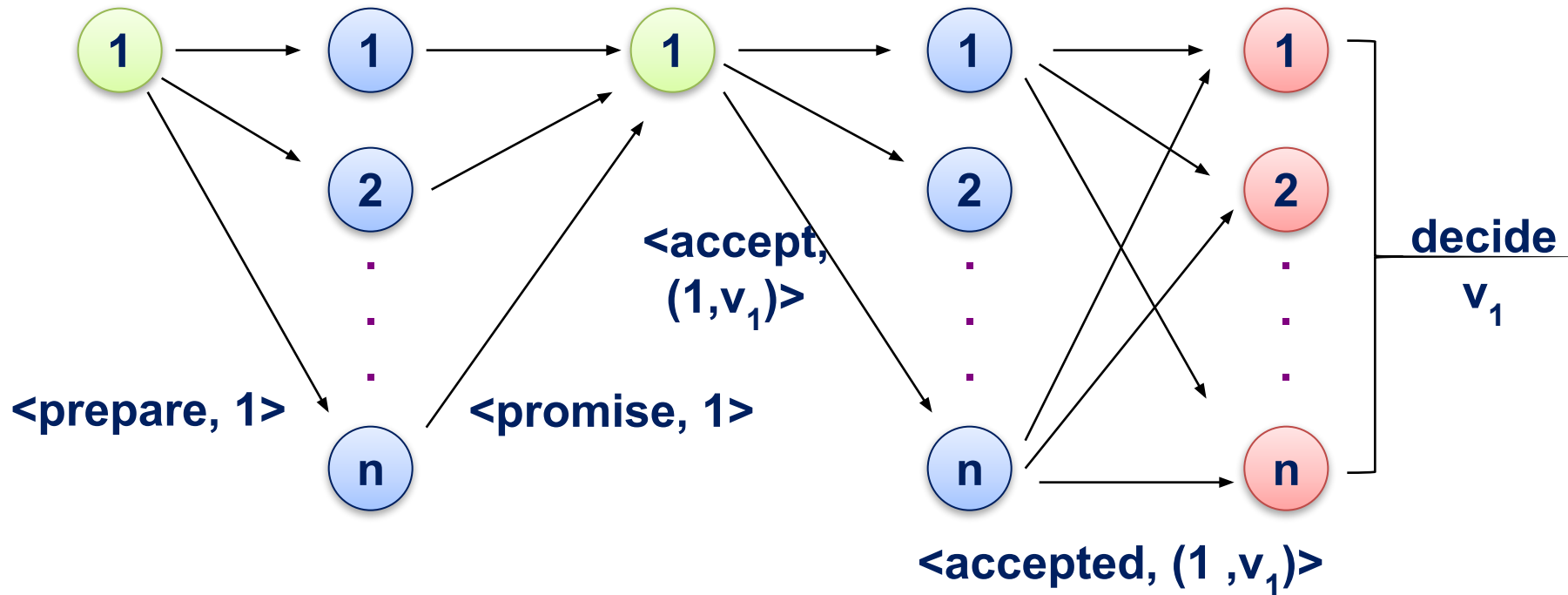
$$n_a = n_h = n$$

$$v_a = v$$

Paxos Phase 3

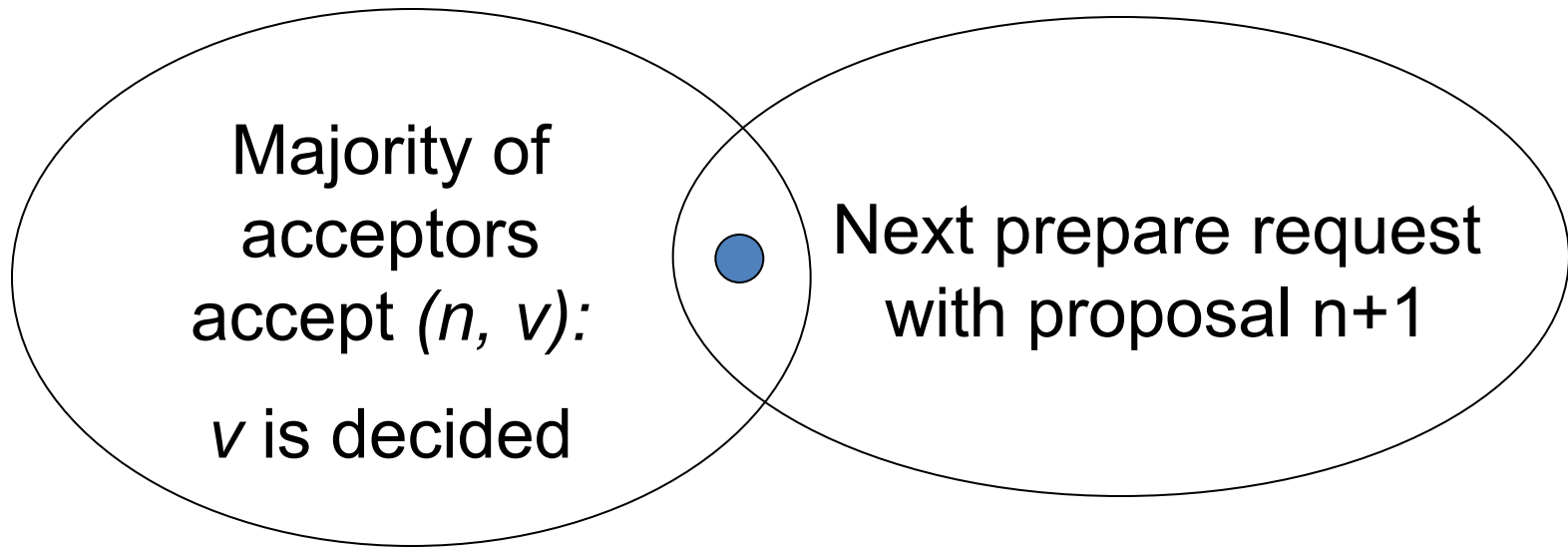
- **Learners** need to know which value chosen
- Approach #1
 - Each acceptor notifies all learners
 - More expensive
- Approach #2
 - Elect a “distinguished learner”
 - Acceptors notify elected learner, which informs others
 - Failure-prone

Paxos: Well-behaved Run



Paxos is safe

- Intuition: if proposal with value v decided, then every higher-numbered proposal issued by any proposer has value v .



Race condition leads to liveness problem

Process 0 Process 1

Completes phase 1
with proposal n_0

Performs phase 2,
acceptors reject

Restarts and completes phase
1 with proposal $n_2 > n_1$

Starts and completes phase
1 with proposal $n_1 > n_0$

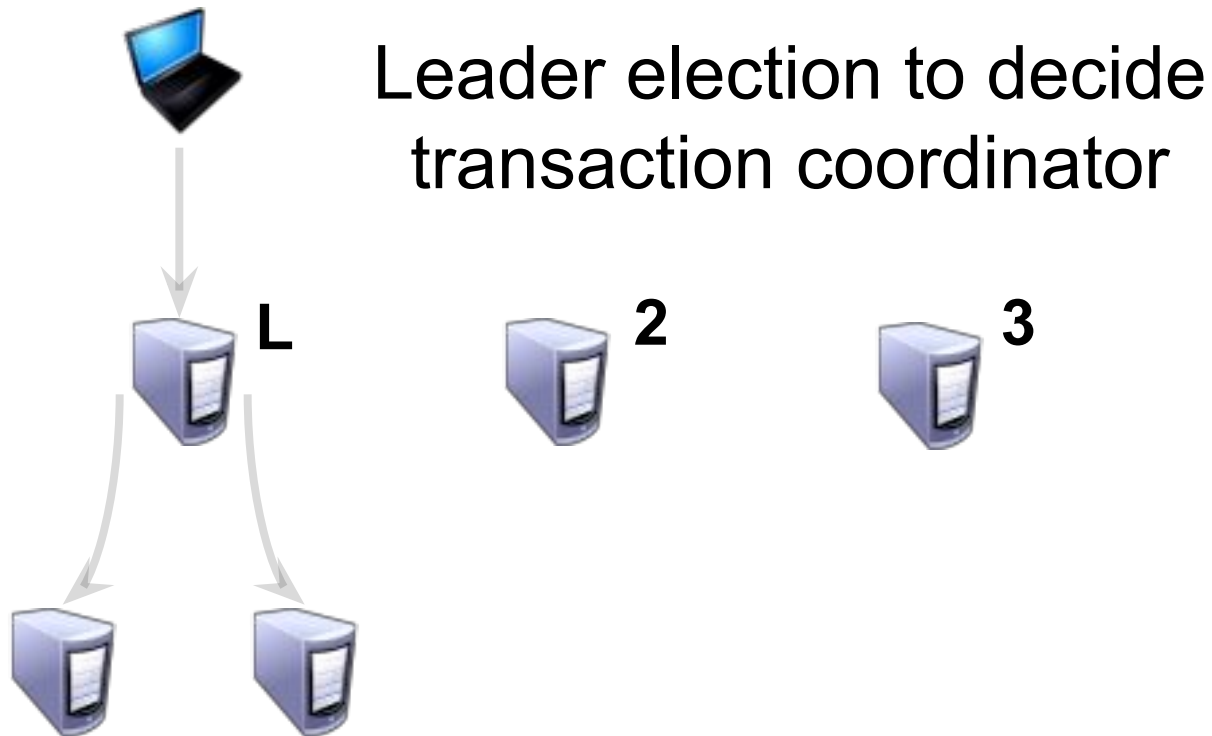
Performs phase 2,
acceptors reject

... can go on indefinitely ...

Paxos with leader election

- Simplify model with each process playing all three roles
- If elected proposer can communicate with a majority, protocol guarantees liveness
- Paxos can tolerate failures $f < N / 2$

Using Paxos in system



Using Paxos in system



The Part-Time Parliament

Leslie Lamport

This article appeared in *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133-169. Minor corrections were made on 29 August 2000.

- Tells mythical story of Greek island of Paxos with “legislators” and “current law” passed through parliamentary voting protocol
- Misunderstood paper: submitted 1990, published 1998
- Lamport won the Turing Award in 2013

The Paxos story...

As Paxos prospered, legislators became very busy.

Parliament could no longer handle all details of government, so a bureaucracy was established.

Instead of passing a decree to declare whether each lot of cheese was fit for sale, Parliament passed a decree appointing a cheese inspector to make those decisions.

Cheese inspector \approx leader
using quorum-based voting protocol

The Paxos story...

Parliament passed a decree making Δῖκοτρα the first cheese inspector. After some months, merchants complained that Δῖκοτρα was too strict and was rejecting perfectly good cheese.

Parliament then replaced him by passing the decree

1375: Γωυδα is the new cheese inspector

But Δῖκοτρα did not pay close attention to what Parliament did, so he did not learn of this decree right away.

There was a period of confusion in the cheese market when both Δῖκοτρα and Γωυδα were inspecting cheese and making conflicting decisions.

Split-brain!

The Paxos story...

To prevent such confusion, the Paxons had to guarantee that a position could be held by at most one bureaucrat at any time.

To do this, a president included as part of each decree the time and date when it was proposed.

A decree making Δῖκοτρα the cheese inspector might read

2716: 8:30 15 Jan 72 – Δῖκοτρα is cheese inspector for 3 months.

Leader gets a lease!

The Paxos story...

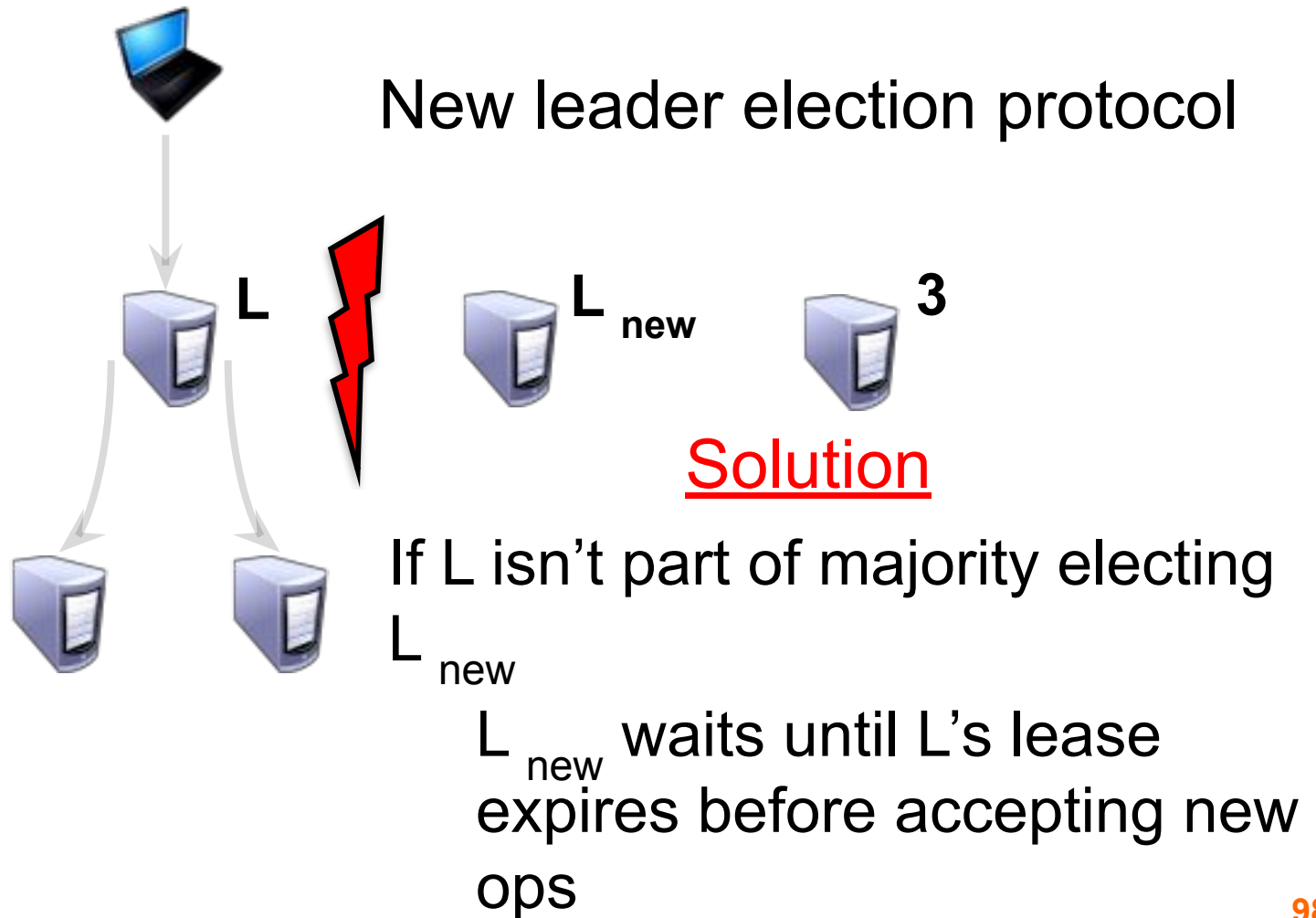
A bureaucrat needed to tell time to determine if he currently held a post. Mechanical clocks were unknown on Paxos, but Paxons could tell time accurately to within 15 minutes by the position of the sun or the stars.

If Δίκστρο's term began at 8:30, he would not start inspecting cheese until his celestial observations indicated that it was 8:45.

Handle clock skew:

Lease doesn't end until expiry + max skew

Solving Split Brain

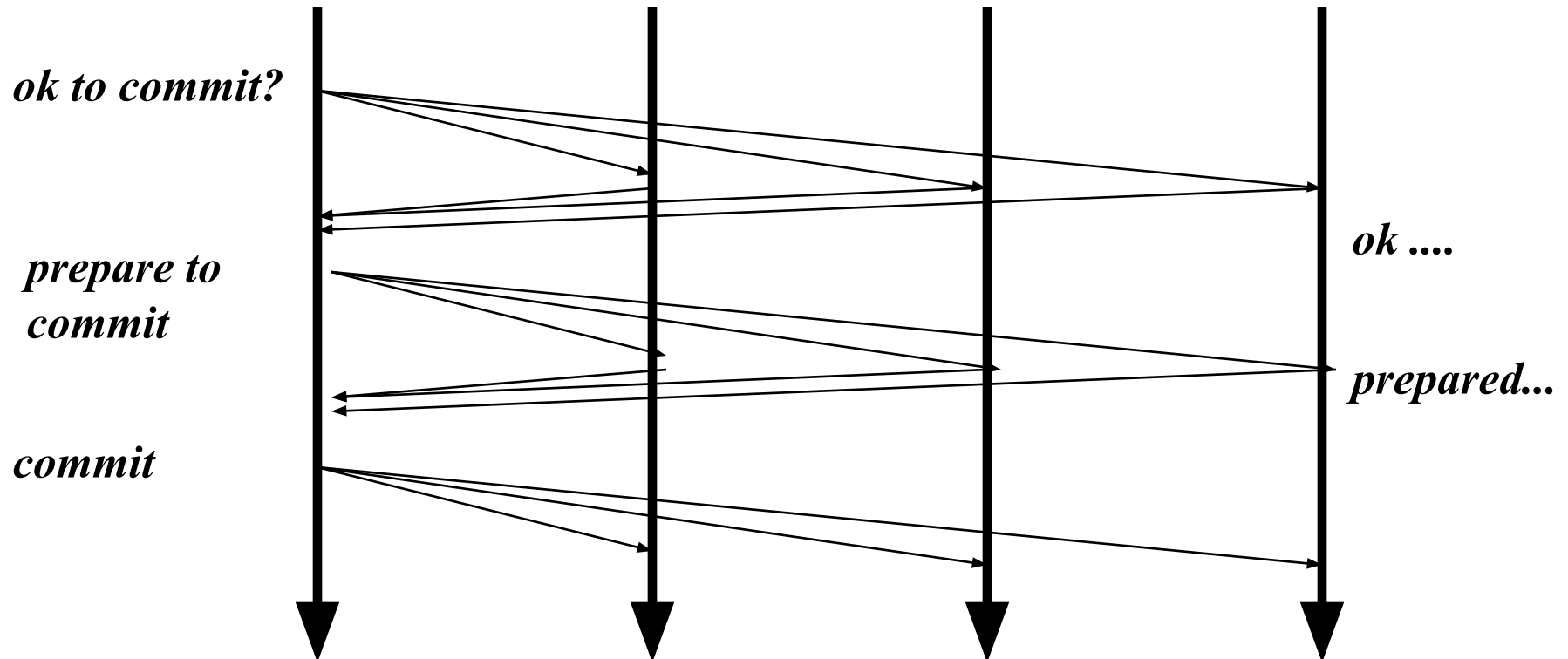




Backup Slides



Three phase commit protocol illustrated



Note: garbage collection protocol not shown here

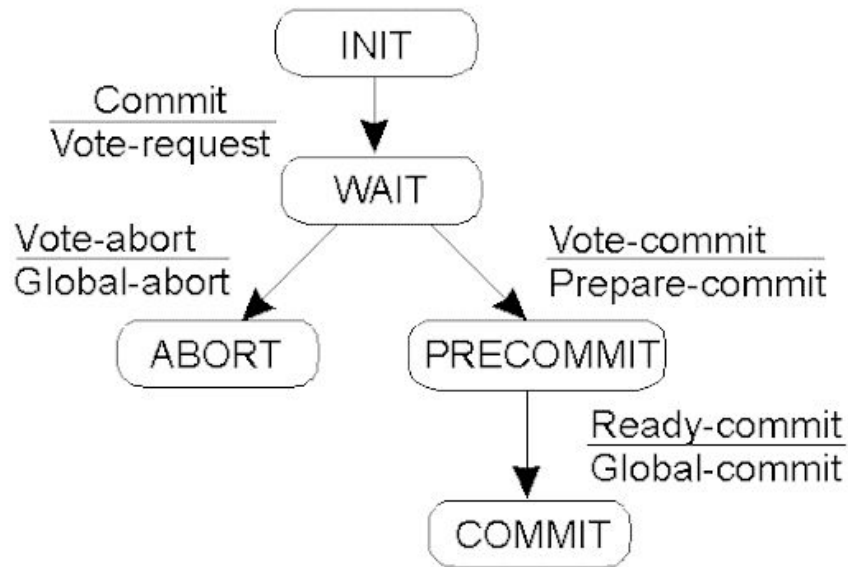


Three-Phase Commit (1/2)

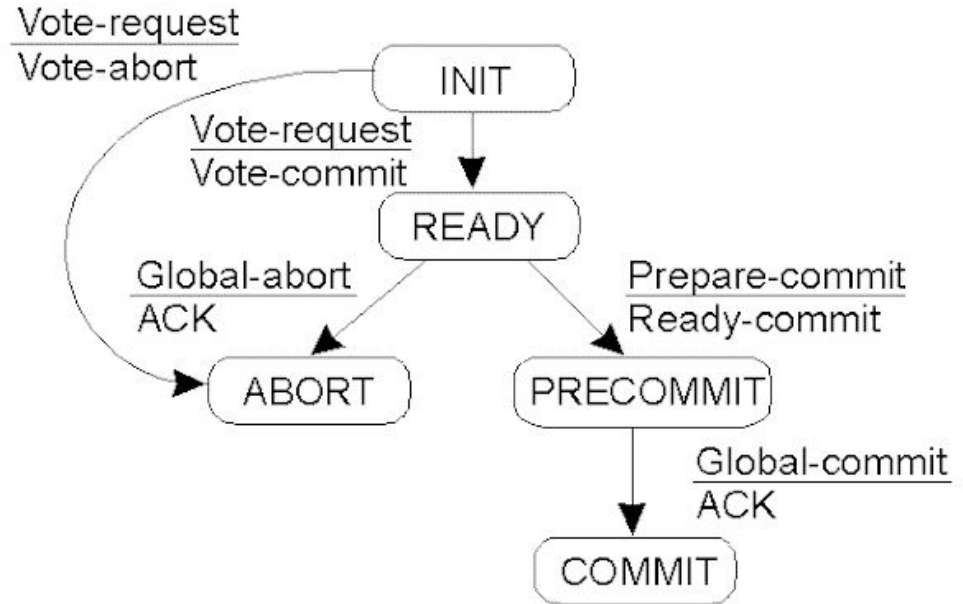
- **Phase 1a:** Coordinator sends `VOTE_REQUEST` to participants
- **Phase 1b:** When participant receives `VOTE_REQUEST` it returns either YES or NO to coordinator. If it sends NO, it aborts its local computation
- **Phase 2a:** Coordinator collects all votes; if all are YES, it sends `PREPARE` to all participants, otherwise it sends `ABORT`, and halts
- **Phase 2b:** Each participant waits for `PREPARE`, or waits for `ABORT` after which it halts
- **Phase 3a:** (Prepare to commit) Coordinator waits until all participants have ACKed receipt of `PREPARE` message, and then sends `COMMIT` to all
- **Phase 3b:** (Prepare to commit) Participant waits for `COMMIT`

3PC □ Finite state machines

Coordinator



Participant



- If any process is in "PRECOMMIT" all voted for commit
 - Protocol commits only when all surviving processes have acknowledged prepare to commit
- If coordinator fails, run the protocol forward to commit state (or back to abort state)

3PC □ Recap. Failing Coordinator and Participant

Problem: Can P find out what it should do after a crash in the ready or pre-commit state, even if other participants or the coordinator failed?

Solution idea: Coordinator and participants on their way to commit, never differ by more than one state transition

Consequences

- If a participant times out in ready state: it can find out at the coordinator or other participants whether it should abort, or enter pre-commit state.
- If a participant already made it to the pre-commit state, it can always safely commit (but is not allowed to do so for the sake of failing other processes)

