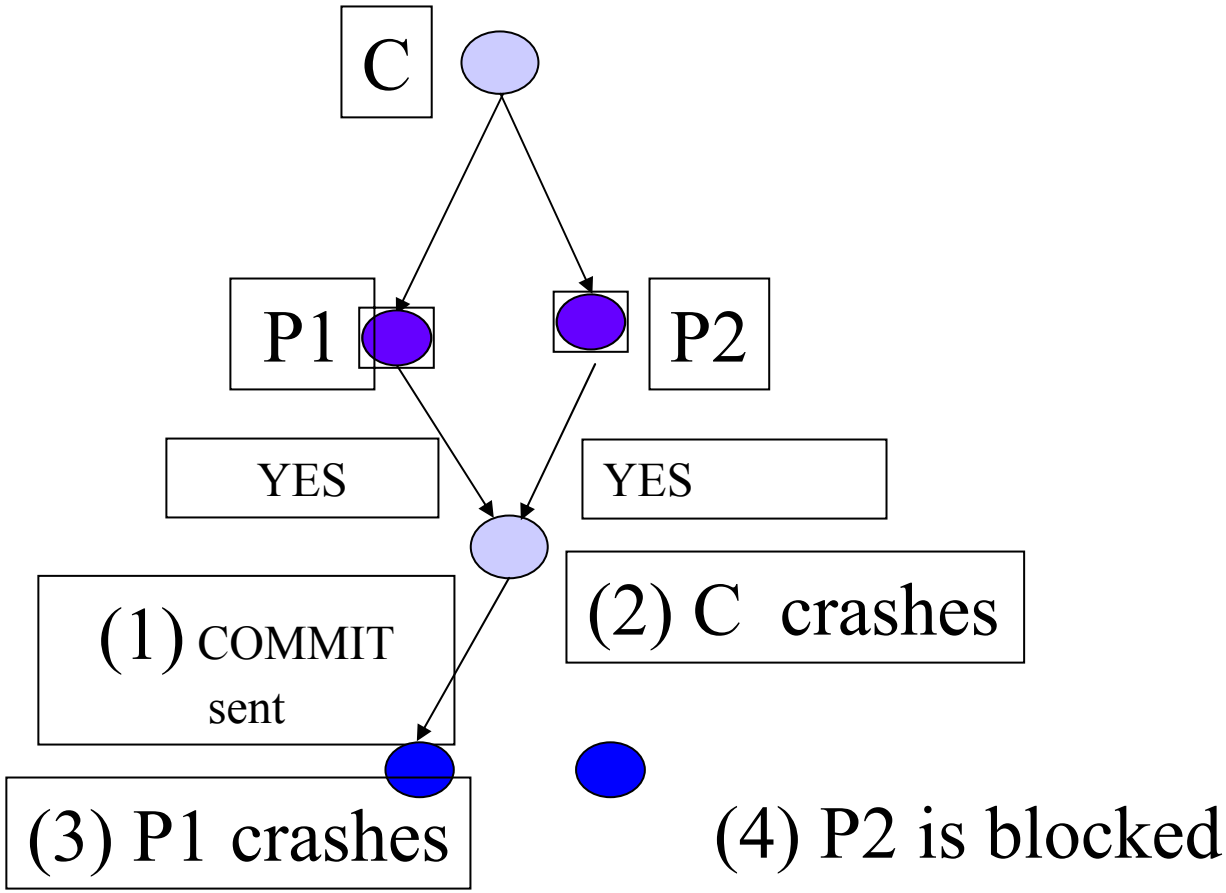
Distributed Transaction Management – 2003

Jyrki Nummenmaa
http://www.cs.uta.fi/~dtm
jyrki@cs.uta.fi

Blocking and non-blocking commit protocols

Based on the paper ”Nonblocking Commit Protocols” by Dale Skeen

Two-Phase Commit Blocks



Failure model

- Our Blocking Theorem from last week states that if network partitioning is possible, then any distributed commit protocol may block.
- Let's assume now that the network can not partition.
- Then we can consult other processes to make progress.
- However, if all participants fail, then we are, again, blocked.
- Let's further assume that total failure is not possible ie. not all participants are inoperational at the same time.

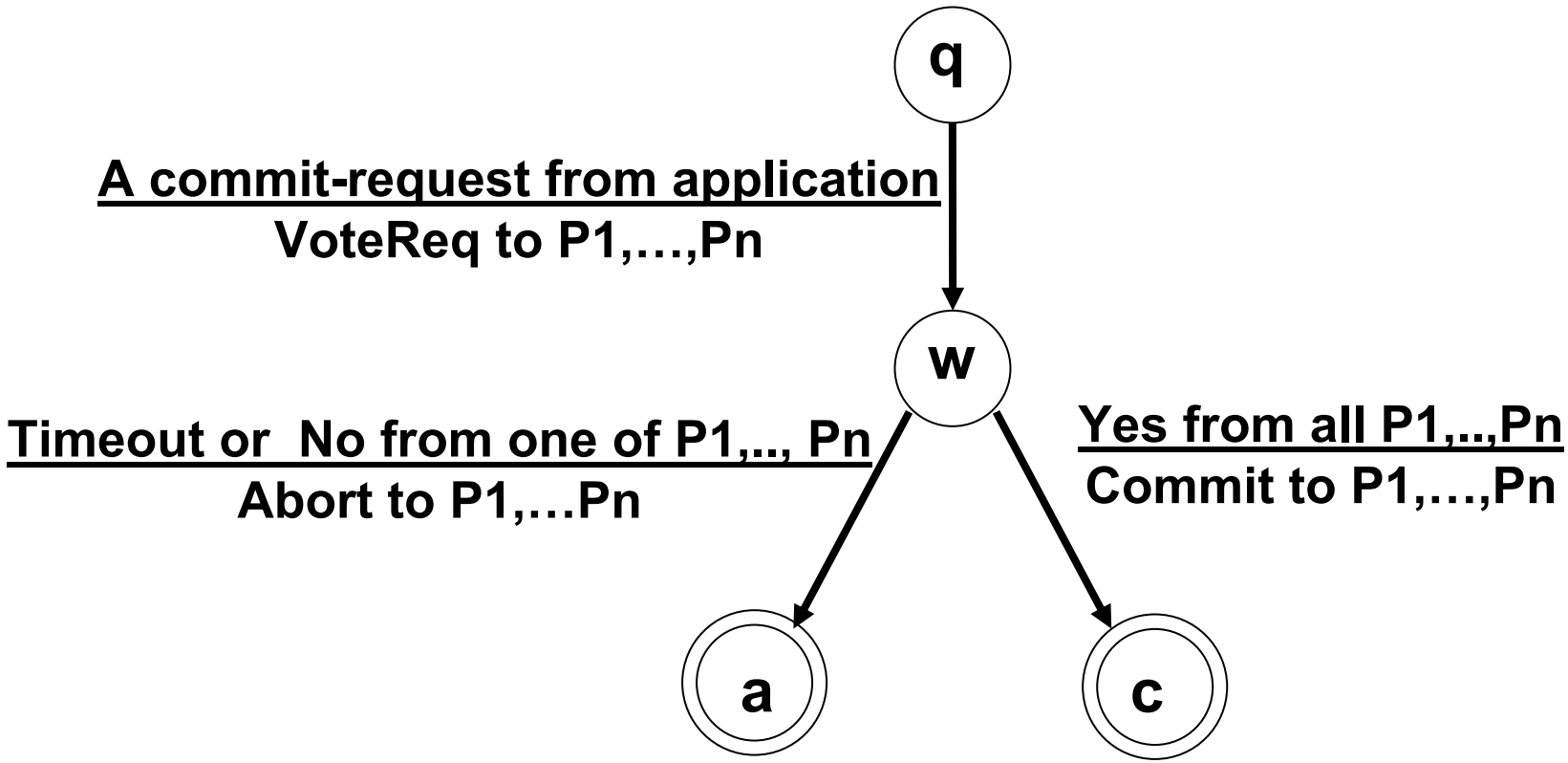
Automata representation

- We model the participants with finite state automata (FSA).
- The participants move from one state to another as a result of receiving one or several messages or as a result of a timeout event.
- Having received these messages, a participant may send some messages before executing the state transition.

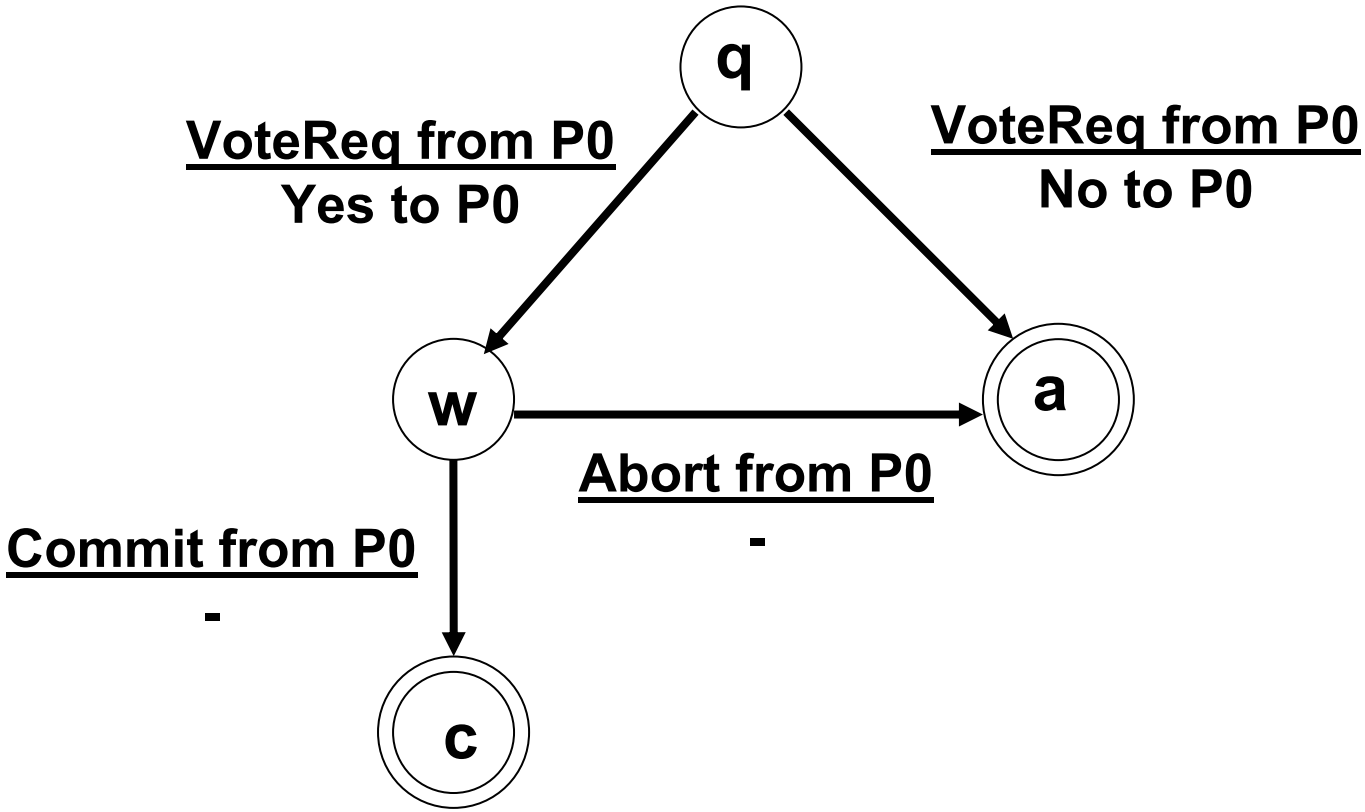
Commit Protocol Automata

- Final states are divided into Abort states and Commit states (finally, either Abort or Commit takes place).
- Once an Abort state is reached, it is not possible to do a transition to a non-Abort state. (Abort is irreversible). Similarly for Commit states (Commit is also irreversible).
- The state diagram is acyclic.
- We denote the initial state by **q**, the terminal states are **a** (an abort/rollback state) and **c** (a commit state). Often there is a wait-state, which we denote by **w**.
- Assume the participants are P1,…,Pn. Possible coordinator is P0, when the protocol starts.

2PC Coordinator



2PC Participant



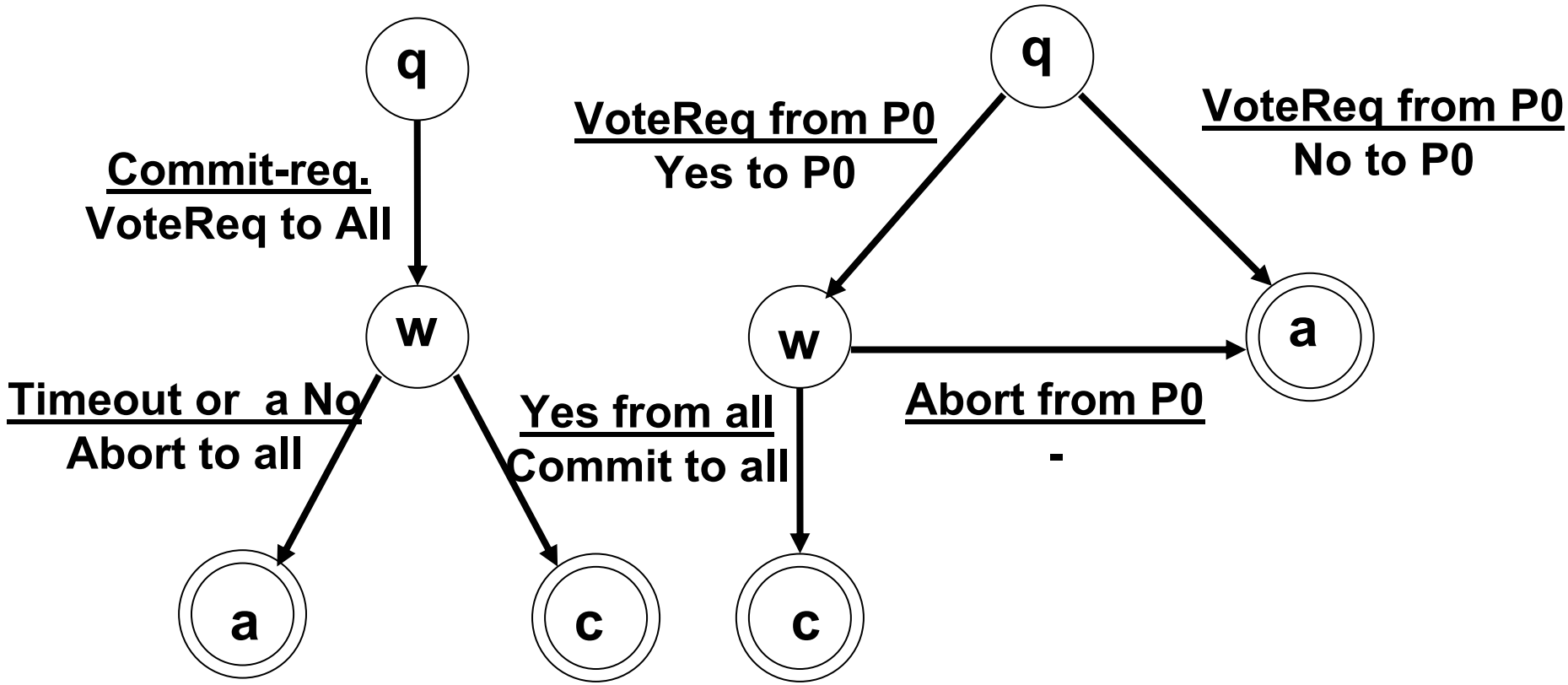
Commit Protocol State Transitions

- In a commit protocol, the idea is to inform other participants on local progress.
- In practice commit protocols exchange data on their progress.
- The results given later in this lecture cover also commit protocols, for which this is not true.
- However, in known protocols it is customary to send messages to other participants about any change of state (unless it is a change into a terminal state).

Concurrency set

- A concurrency set of a state s is the set of possible states among all participants, if some participant is in state **s**.
- In other words, the concurrency set of state **s** is the set of all states that can co-exist with state **s**.

2PC Concurrency Sets



Concurrency_set(q) = {q,w,a}, Concurrency_set(a) = {q,w,a}
Concurrency_set(w) = {q,w,a,c}, Concurrency_set(c) = (w,c)

Committable states

- We say that a state is committable, if the existence of a participant in this state means that everyone has voted Yes.
- If a state is not committable, we say that it is non-committable.
- In 2PC, **c** is the only committable state.

How can a site terminate when there is a timeout?

- Either (1) one of the operational sites knows the fate of the transaction, or (2) the operational sites can decide the fate of the transaction.
- Knowing the fate of the transaction means, in practice, that there is a participant in a terminal state.
- Start by considering a single participant s. Participant s must infer the possible states of other participants from its own state. This can be done using concurrency sets.

When can't a single participant unilaterally abort?

- Suppose a participant is in a state, which has a commit state in its concurrency set. Then, it is possible that some other participant is in a commit state.
- A participant in a state, which has a commit state in its concurrency set, should not unilaterally abort.

When can't a single participant unilaterally commit?

- Suppose a participant is in a state, which has an abort state in its concurrency set. Then, some participant may be in an abort state.
- A participant in a state, which has an abort state in its concurrency set, should not unilaterally commit.
- Also, a participant that is not in a committable state should not commit.

The Fundamental Non-Blocking Theorem

- A protocol is non-blocking, if and only if it satisfies the following conditions:
  (1) There exists no local state such that its concurrency set contains both an abort and a commit state, and
  (2) there exists no noncommittable state, whose concurrency set contains a commit state.

Showing the Fundamental Non-Blocking Theorem

- From our discussion above it follows that Conditions (1) and (2) are necessary.
- We discuss their sufficiency later by showing how to terminate a commit protocol fulfilling conditions (1) and (2).

Observations on 2PC

- As the participants exchange messages as they progress, they progress in a synchronised fashion.
- In fact, there is always at most one step difference between the states of any two live participants.
- We say that the participants keep a one-step synchronisation.
- It is easy to see by Fundamental Nonblocking Theorem that 2PC is blocking.

One-step synchronisation and non-blocking property

- If a commit protocol keeps one-step synchronisation, then the concurrency set of state **s** consists of **s** and the states adjacent to **s**.
- By applying this observation and the Fundamental Non-blocking Theorem, we get a useful Lemma:

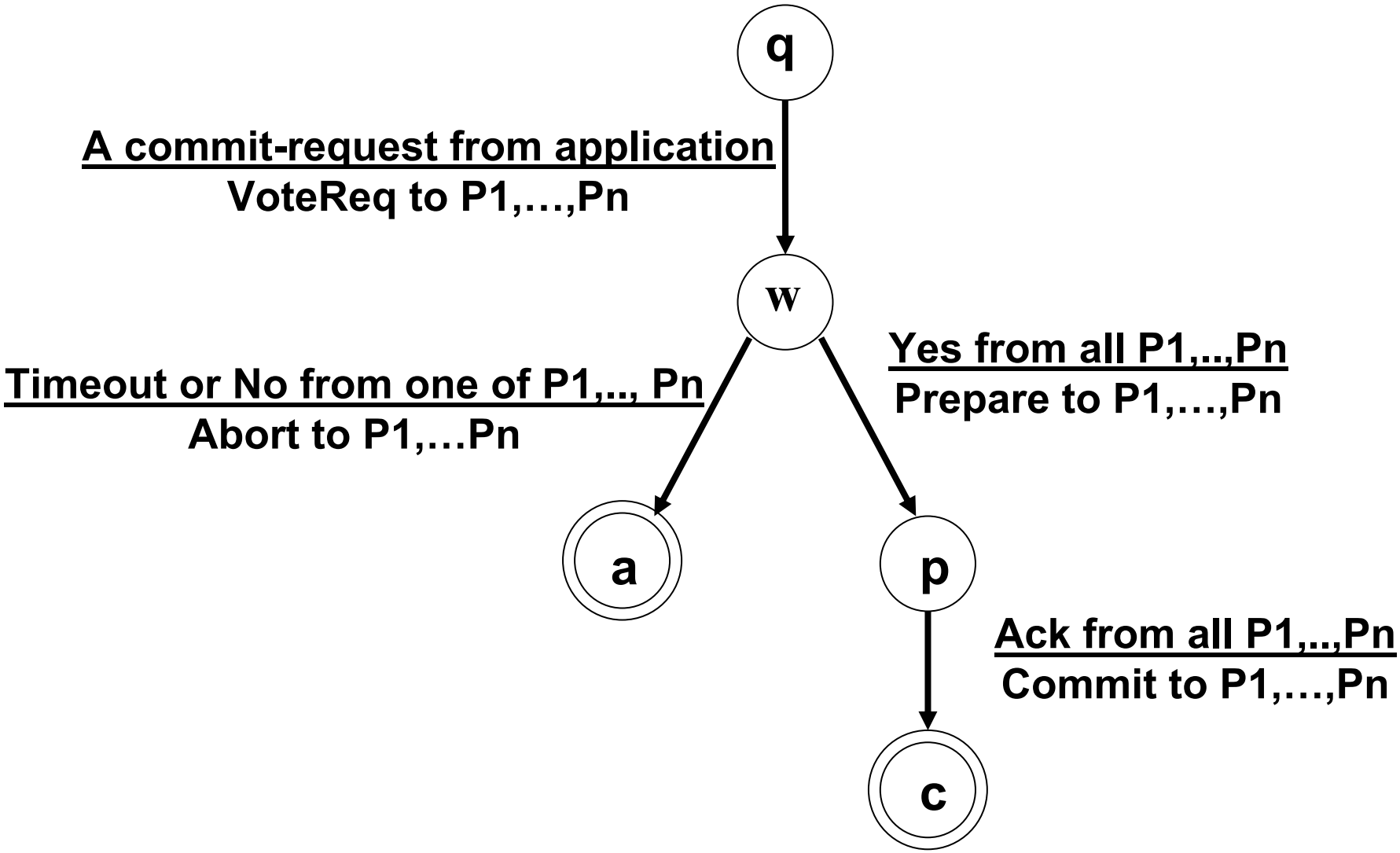
Lemma

- A protocol that is synchronous within one state transition is non-blocking, if and only if
  (1) it contains no state adjacet to both a Commit and an Abort state, and
  (2) it contains non non-committable state that is adjacet to a commit state.

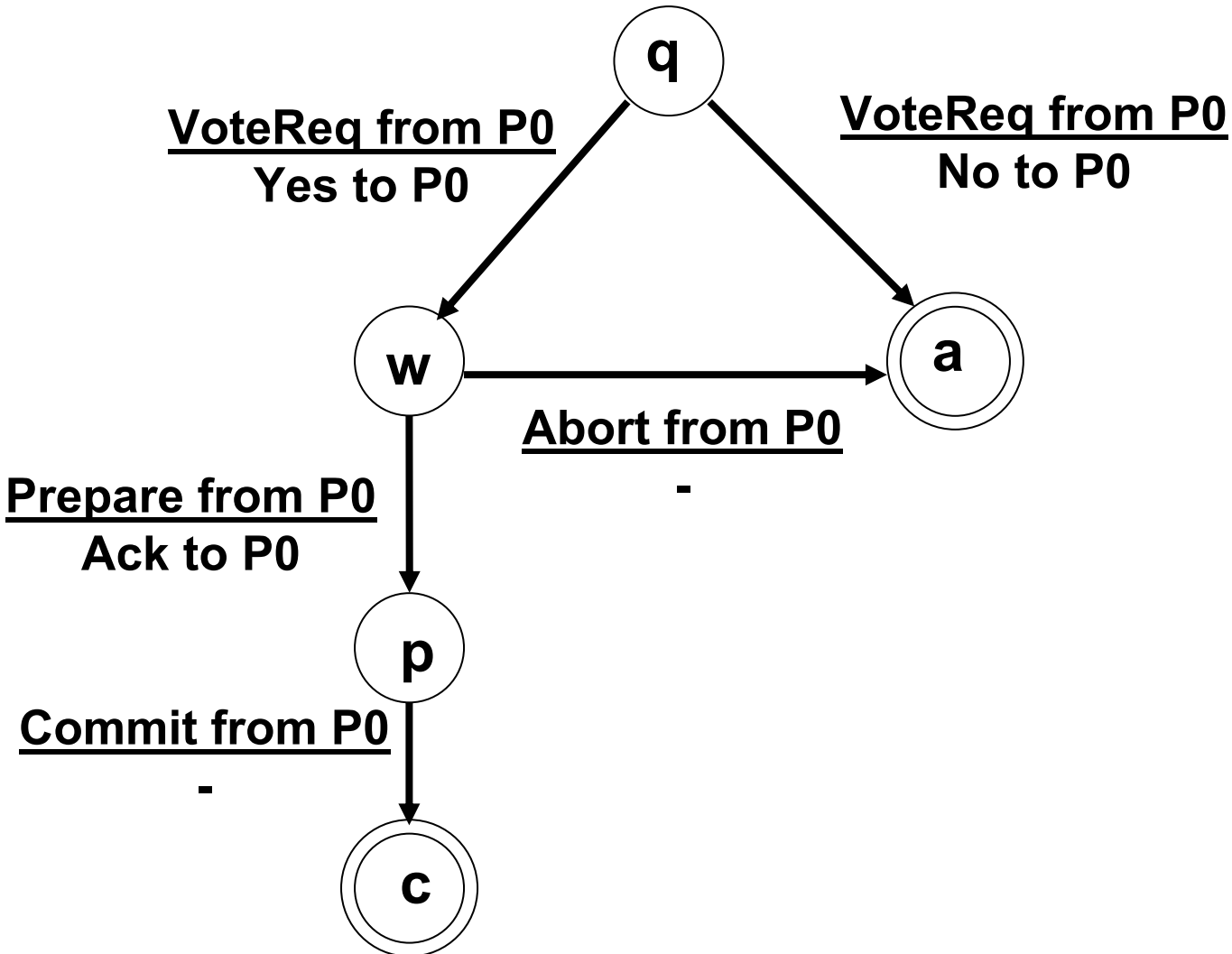
How to improve 2PC to get a non-blocking protocol

- It is easy to see that the state w is the problematic state – and in two ways:
  - it has both Abort and Commit in its concurrency set, and
  - it is a non-committable state, but it has Commit in its concurrency set.
- Solution: add an extra state between **w** and **c** (adding between **w** and **a** would not do – why?)
- We are primarily interested in the centralised protocol, but similar decentralised improvement is possible.

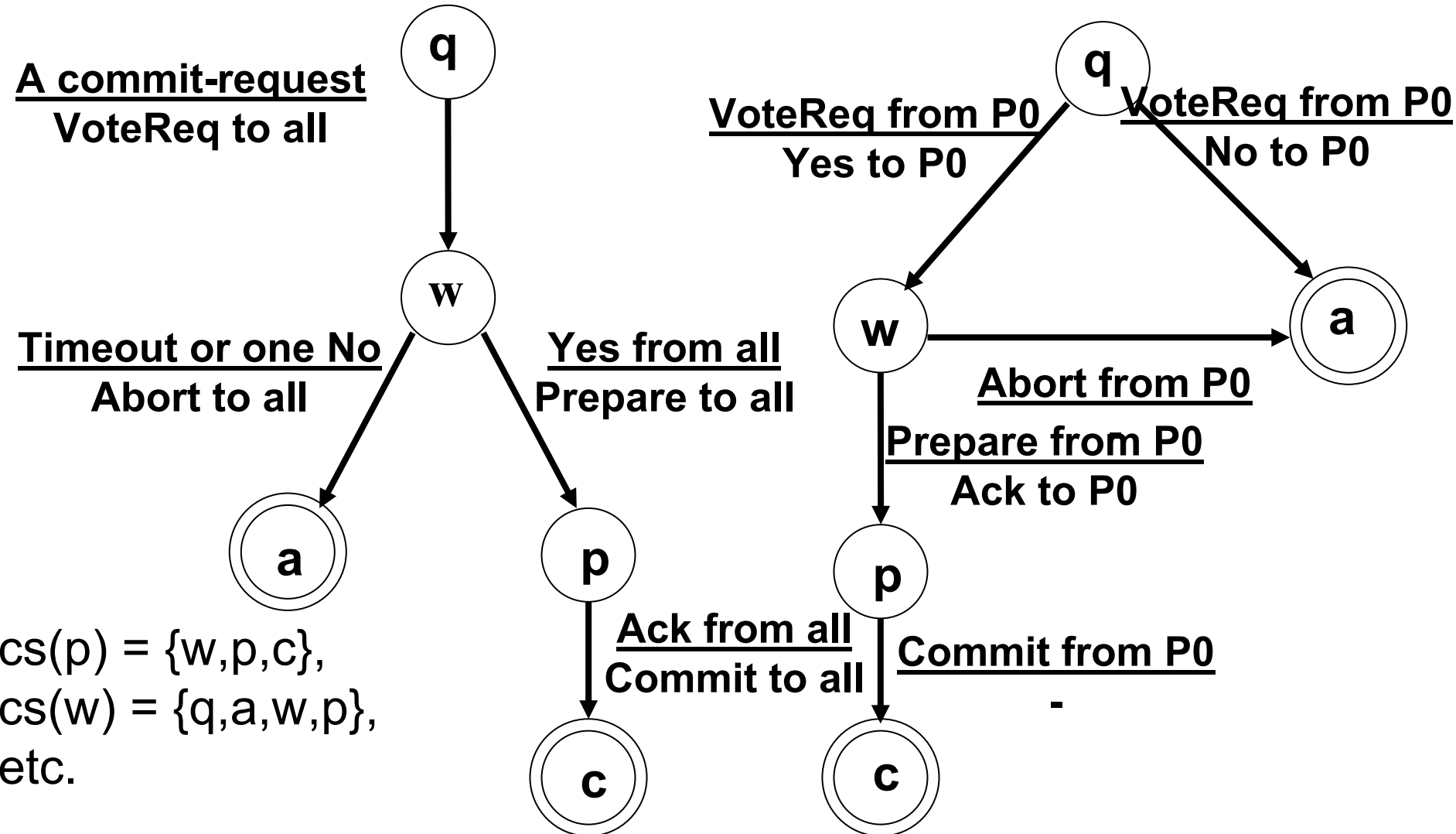
3PC Coordinator



3PC Participant



3PC Concurrency sets (cs)



cs(p) = {w,p,c},
cs(w) = {q,a,w,p},
etc.

3PC and failures

- If there are no failures, then clearly 3PC is correct.
- In the presence of failures, the operational participants should be able to terminate their execution.
- In the centralised case, a need for termination protocol implies that the coordinator is no longer operational.
- We discuss a general termination protocol. It makes the assumption that at least one participant remains operational and that the participants obey the Fundamental Non-Blocking Theorem.

Termination

- Basic idea: Choose a backup coordinator B – vote or use some preassigned ids.
- Backup Coordinator Decision Rule:
  If the B's state contains commit in its concurrency set, commit the transaction. Else abort the transaction.
- Reasoning behind the rule: If B's state contains commit in the concurrency set, then it is possible that some site has performed commit – otherwise not.

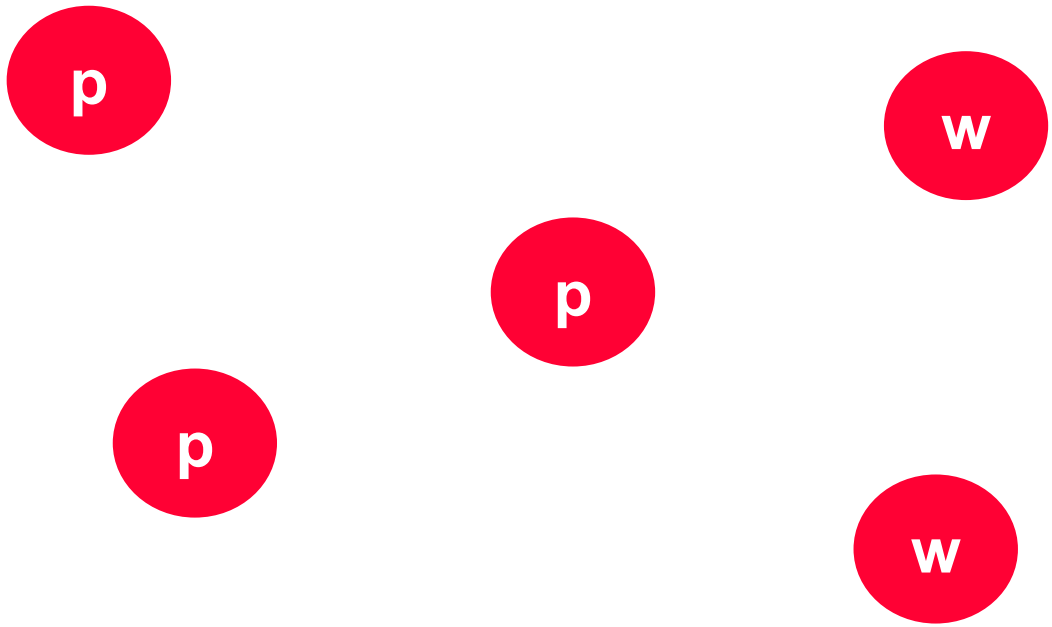
Re-executing termination

- It is, of course, possible the backup coordinator fails.
- For this reason, the termination protocol should be executed in such a way that it can be re-executed.
- In particular, the termination protocol must not break the one-step synchronisation.

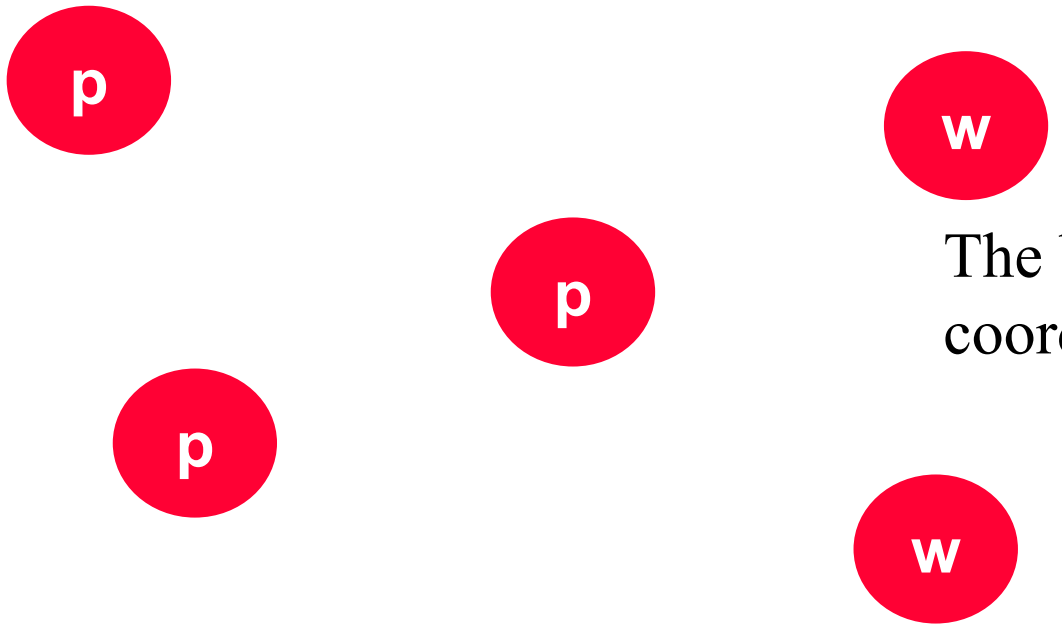
Implementing termination

- To keep one-step synchronisation, the termination protocol should be executed in two steps:
- 1. The backup coordinator B tells the others to make a transition to B's state. Others answer Ok. (This is not necessary if B is in Commit or Abort state.)
  2. B tells the others to commit or abort by the decision rule.

What can happen, if we break one-step synchronisation?

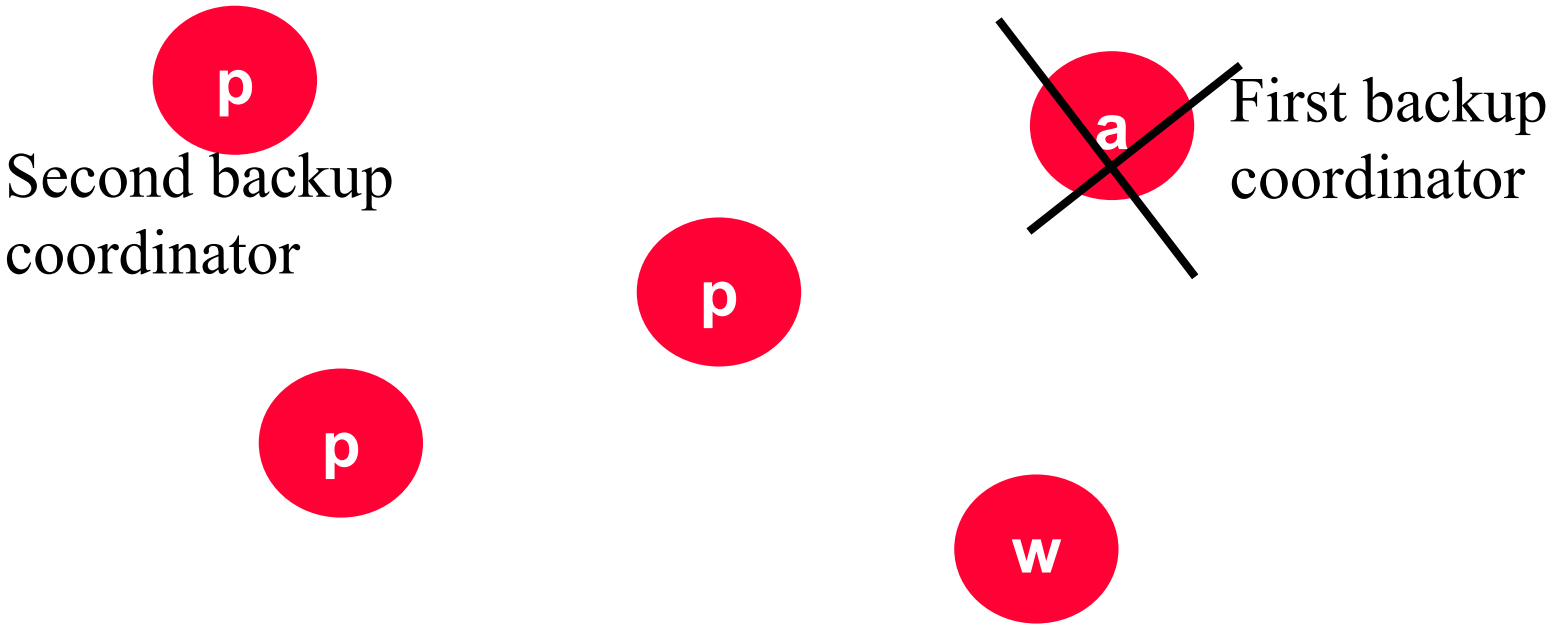


The circles are live participants and the letters are their states. Suppose that the coordinator is not alive and not in the picture.



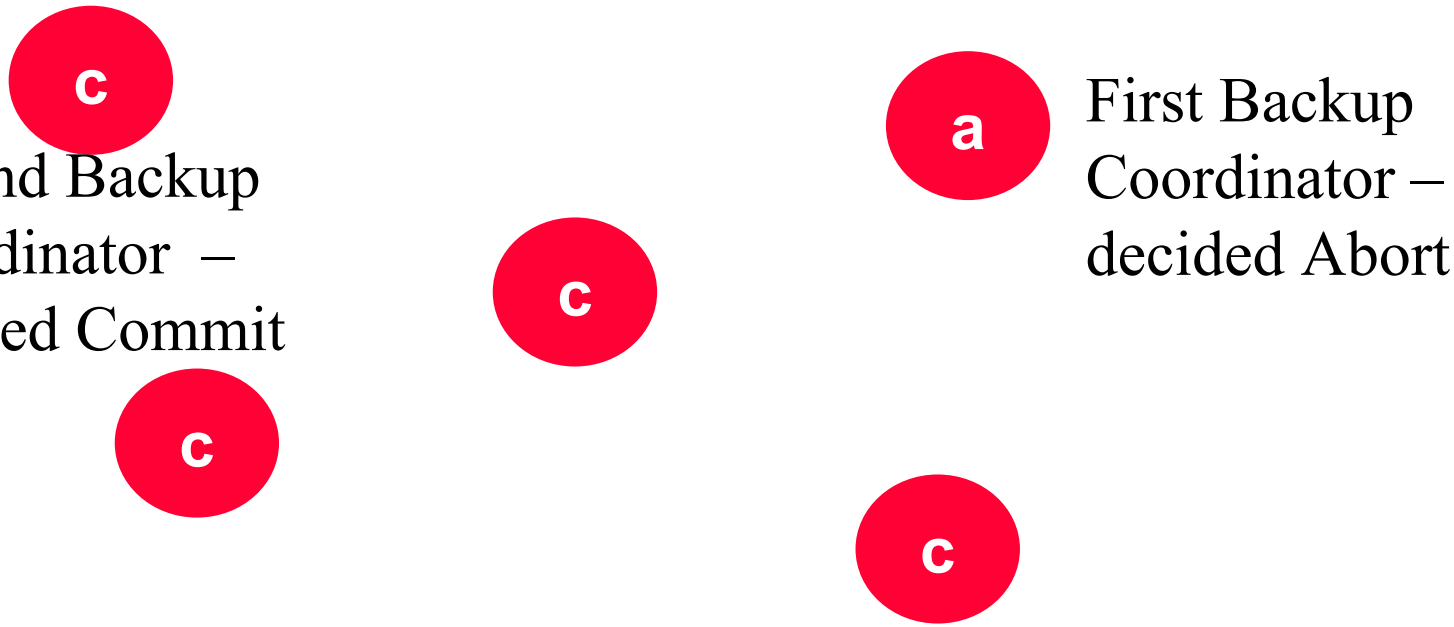
The backup coordinator sends others Abort messages and unilaterally rolls back (according to decision rule). The messages are lost and the backup coordinator crashes.

Now the one-step synchronisaton is lost



By the decision rule, the second backup coordinator decides Commit.

…final result.



This is incorrect.

Fundamental Non-Blocking Theorem Proof - Sufficiency

- The basic termination procedure and decision rule is valid for any protocol that fulfills the conditions given in the Fundamental Non-Blocking Theorem.
- The existence of a termination protocol completes the proof.

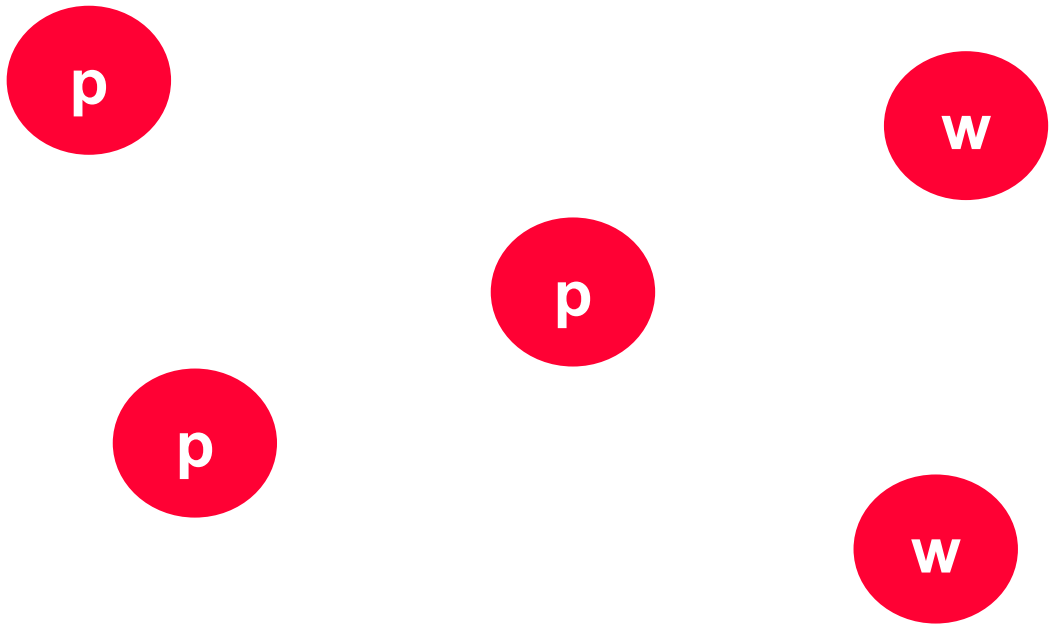
Discussion

- Clearly, an Abort decision can be reached in 3PC similarly as in 2PC in the straightforward case.
- However, Commit requires extra messages.
- The bad thing here is that in practice nearly all of the txns end doing a Commit. This way, nearly all of the commits require extra messages.
- Unfortunately, we can not create a non-blocking protocol by adding a “pre-abort” state instead of the “pre-commit” state.

Back to real world

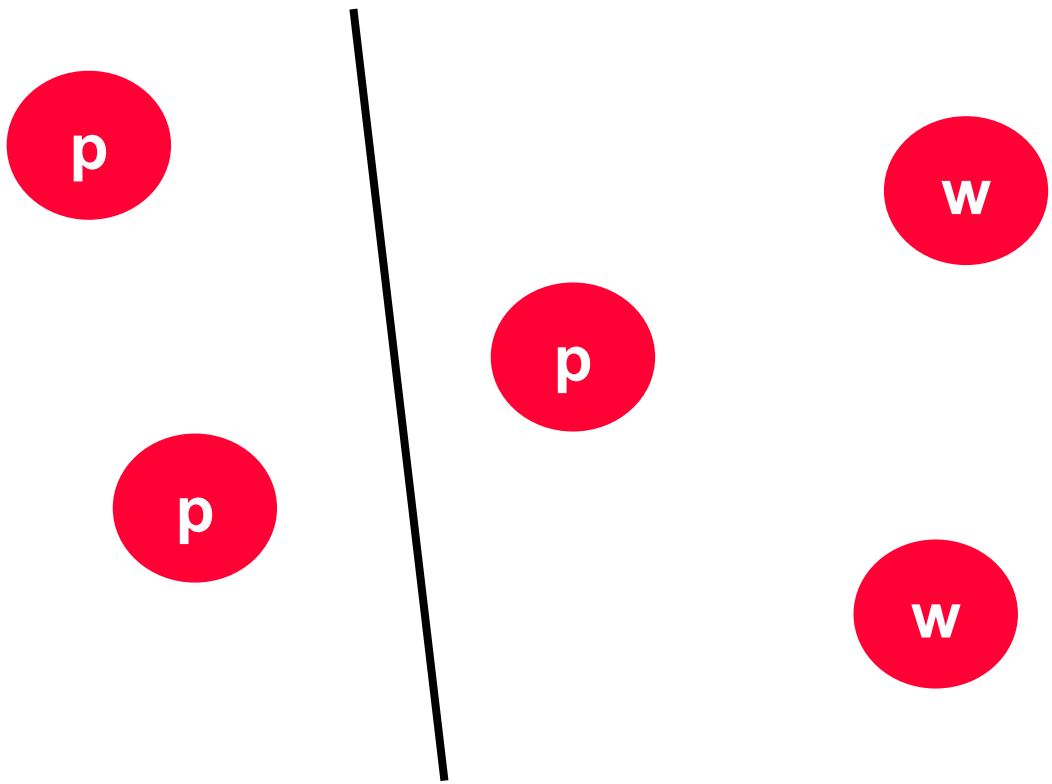
- For the creation of non-blocking protocols, Skeen's work is based on the assumption that the network may not partition.
- However, in reality it is possible that the network partitions.
- In 2PC this means simply blocking.
- Let's see what 3PC does, if the network blocks.

Assume the following situation…



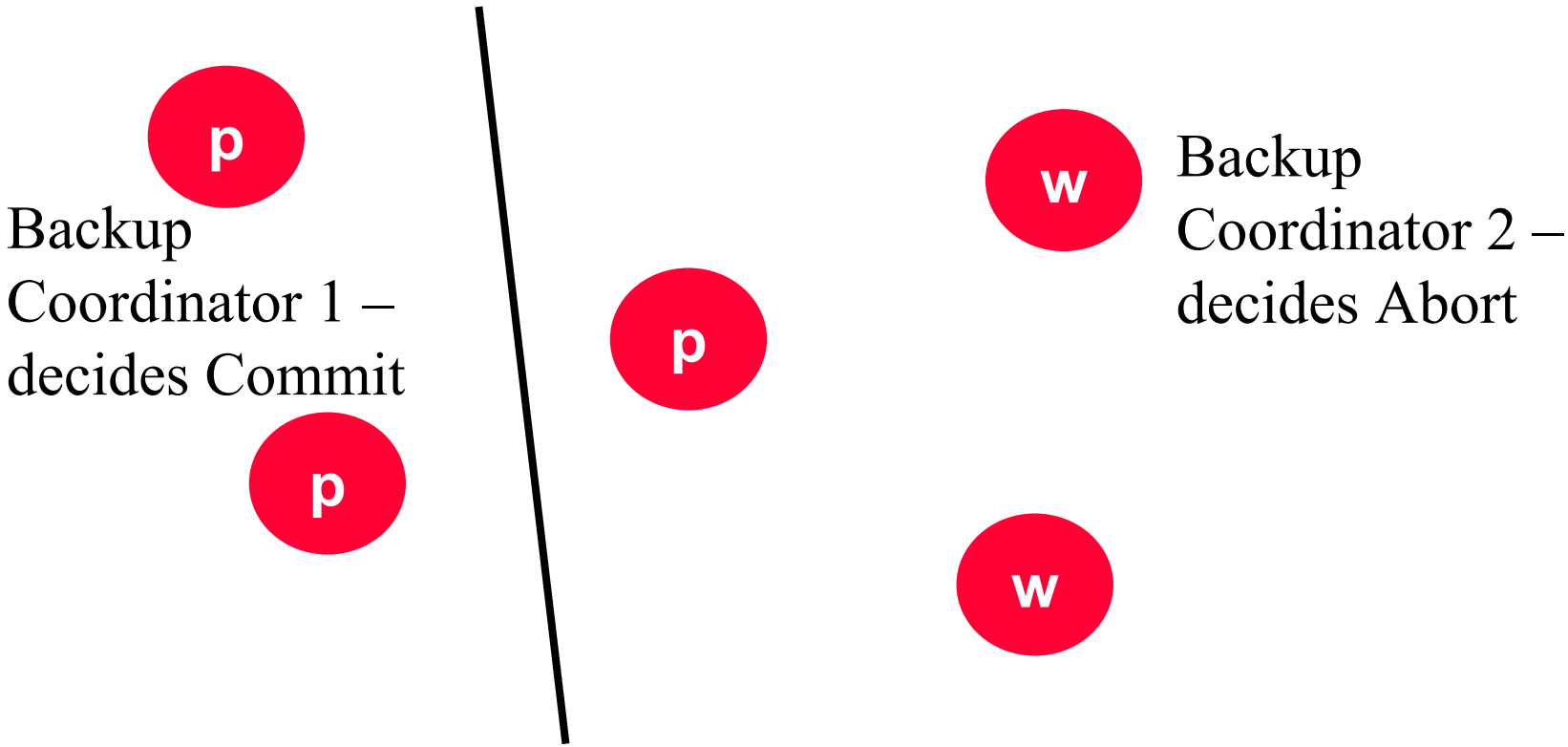
The circles are participants and the letters are their states.

…then the network partitions…



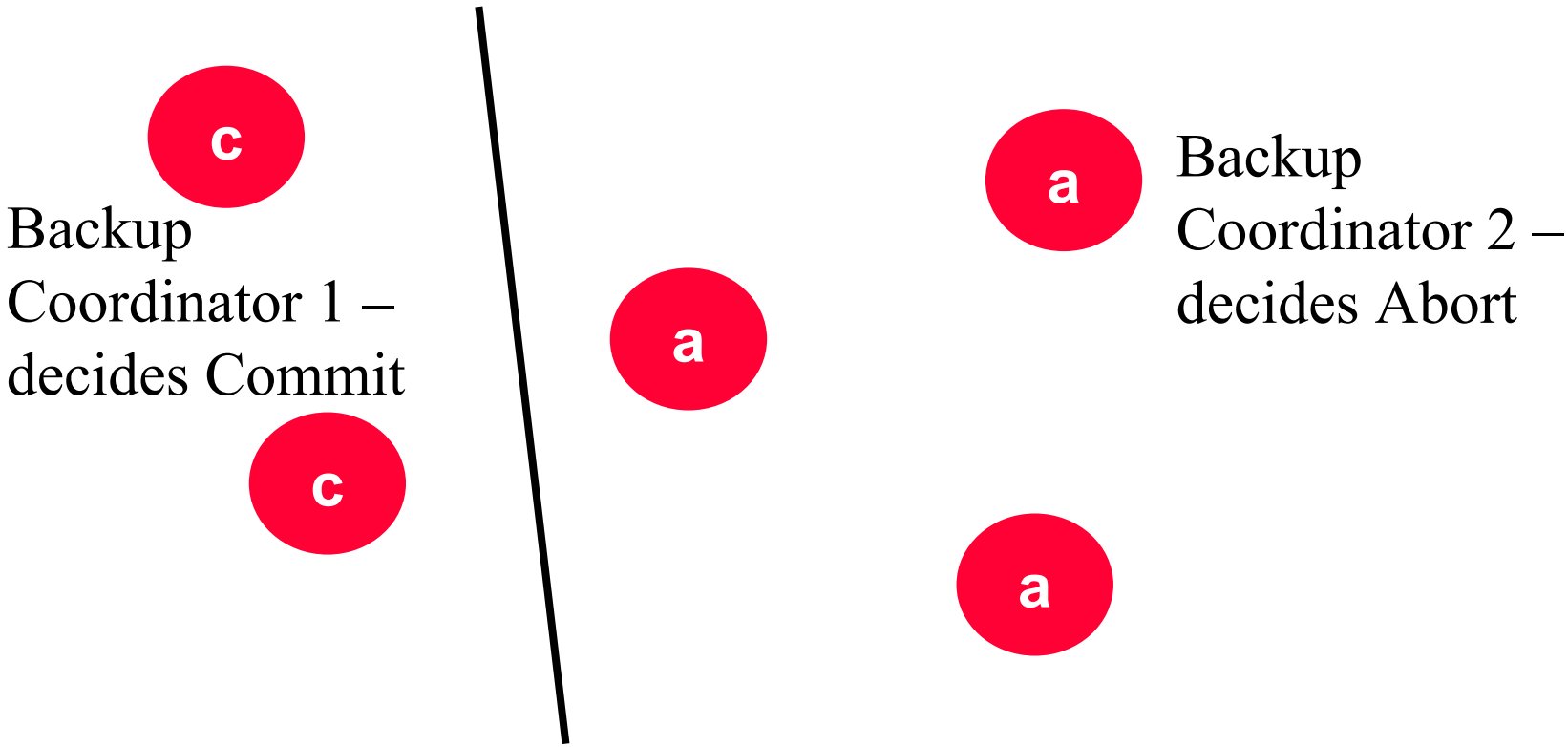
The circles are participants and the letters are their states.

…3PC termination kicks in…



Termination starts in both sets, as they think that the non-responsive sites have failed.

…final result.

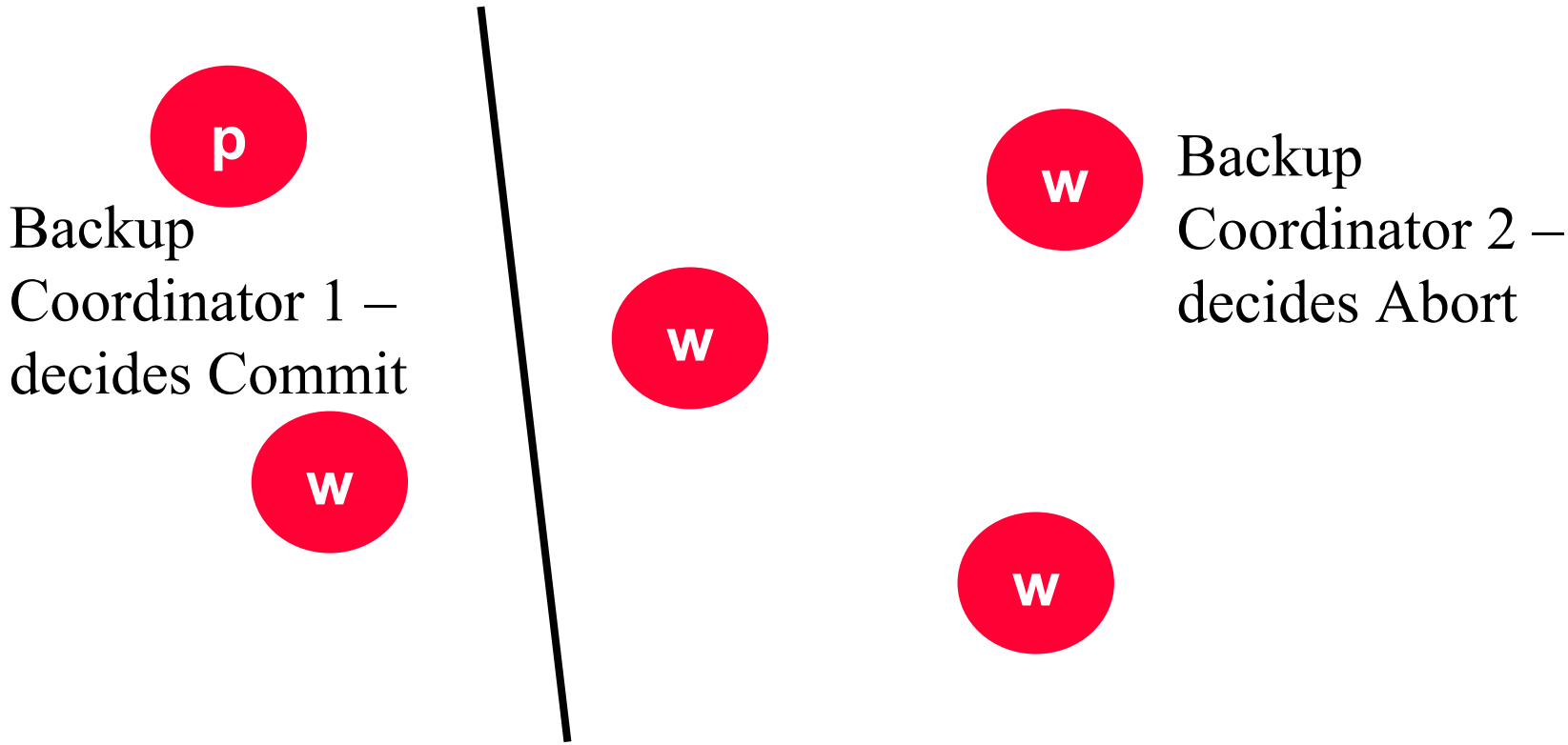


Conclusion: We achieved non-blocking if the network does not partition. If it does, the protocol is no longer correct.

Usefulness of 3PC

- The previous example suggests that 3PC is in fact not useful at all in practice. Can we find a fix?
- Possibility 1: Allow a group to elect a backup coordinator and terminate only, if they contain a majority of the original sites.
- Possibility 2: Allow a group to recover only, if it contains two different states (and there can be at most 2). This may have further practical complications.
- Note that Possibilities 1 and 2 are not compatible.

Possibilities 1 and 2 used at the same time



Group on left contains two different states, group on right contains a majority of the original sites.

3PC Recovery

- Similarly as in 2PC, a recovering participant needs to consult the operational participants and ask about the fate of the txn.
- Also, the participants need to write logs as in 2PC – details are left as an exercise.

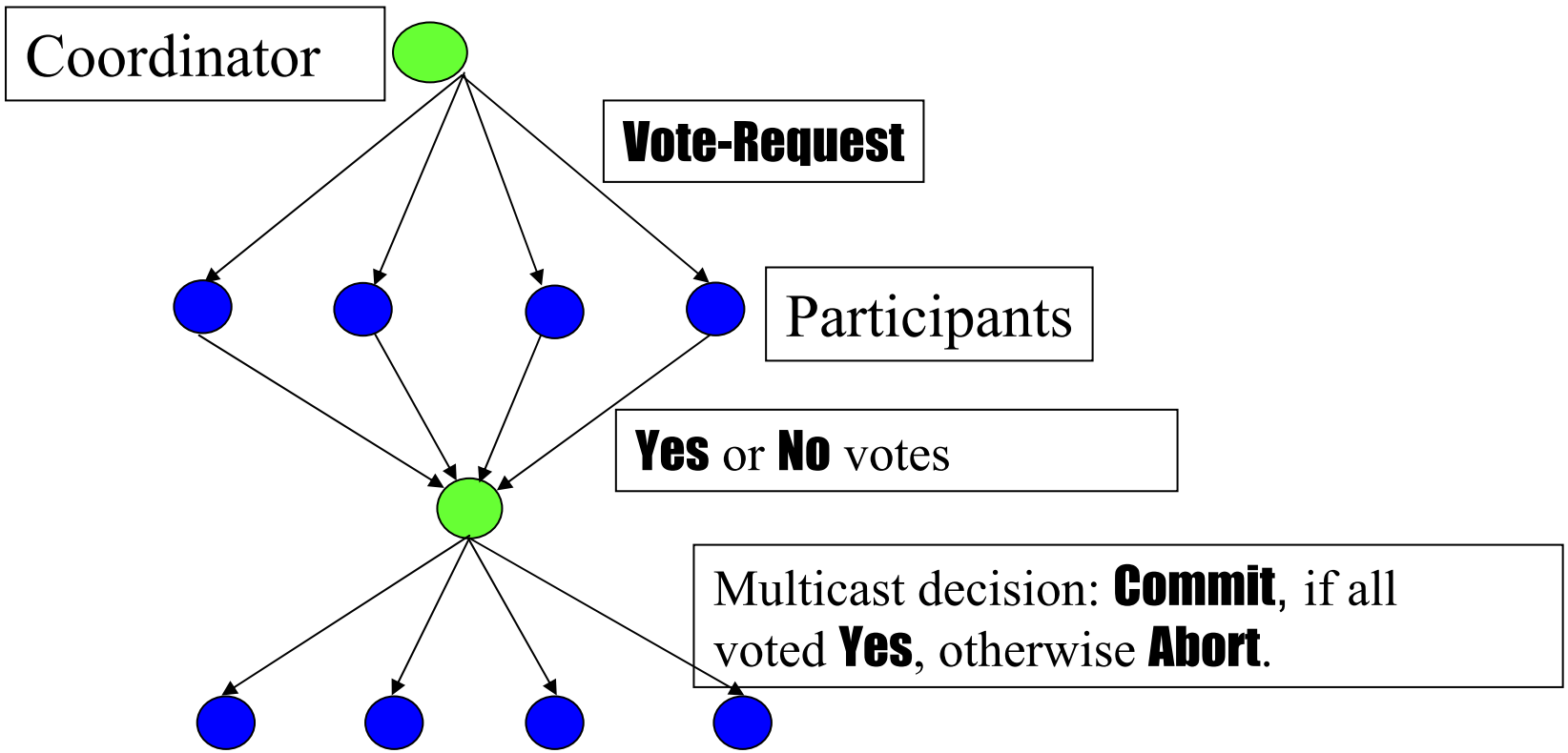
Conclusions on 3PC

- 3PC can in practice be used to solve some situations, where 2PC would block.
- The costs are increased communication and implementation complexity.
- Also, one needs to understand when termination is possible in practice.

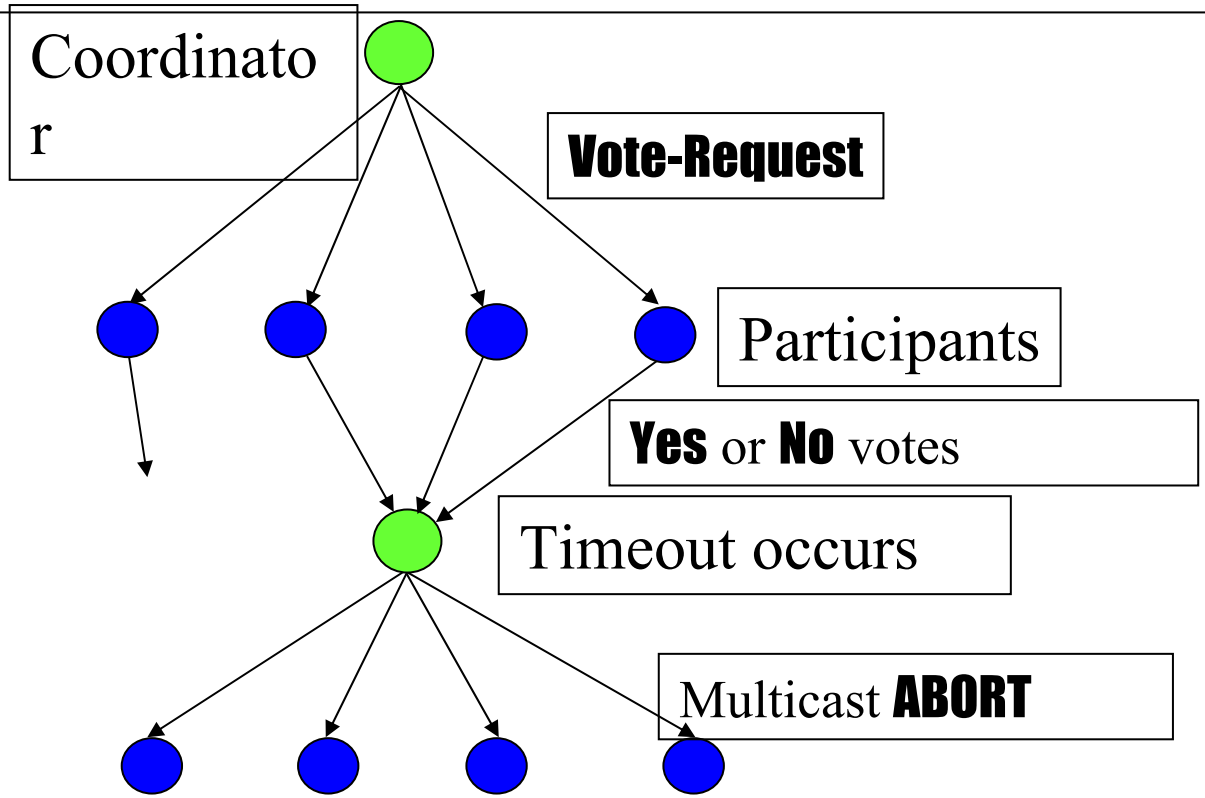
PRACTICAL DISTRIBUTED COMMIT

Based on paper “Practical distributed commit in modern environments” by Nummenmaa and Thanisch

2PC for Distributed Commit



2PC - a timeout occurs



Q: Is this good?

A: (as we will see): Maybe

Why would the timeout mechanism be good?

- Because it may be that some of the participating processes are holding resources, which are needed for other transactions.
  - Holding these resources may reduce throughput of transaction processing, which, of course, is a bad thing.
  - Timeout mechanism may help to find out that something is wrong.

Why would the timeout mechanism not be good? / 1

- Because, given the different types of failures, it may extremely difficult to figure out a ”good” timeout period, even with dynamically adjustable statistics.
  - This is, assuming that timeout is meant to be used to detect failures.

Why would the timeout mechanism not be good? / 2

- Because it may be that none of the participating processes are holding resources, which are needed for other transactions.
  - In this case, we should allow the processes to hold their locks for resources.
  - Rolling the transaction back will only lead to either unnecessarily repeating some processing or a lost transaction.

Why would the timeout mechanism not be good? / 3

- Because it may be that some of the participating processes are holding resources, which are needed for other transactions, and the timeout comes too late to save the performance.

Why is this happening?

- The traditional problem definition for atomic distributed commit is not really related to overall system performance.
- The impractical problem definition gives impractical protocols.
- Currently, the protocols now first try to reach a commit decision, and after a timeout they will try to reach an abort decision, regardless of other factors.

Traditional problem definition for distributed atomic commit /1

- (1) A participant can vote **Yes** or **No** and may not change the vote.
- (2) A participant can decide either *Abort* or *Commit* and may not change it.
- (3) If any participant votes **No**, then the global decision must be *Abort*.
- (4) It must never happen that one participant decides *Abort* and another decides *Commit*.

Traditional problem definition for distributed atomic commit / 2

- (5) All participants, which execute sufficiently long, must eventually decide, regardless whether they have failed earlier or not.
- (6) If there are no failures or suspected failures and all participants vote **Yes**, then the decision must not be *Abort*.

  The participants are not allowed to create artificial failures or suspicion.

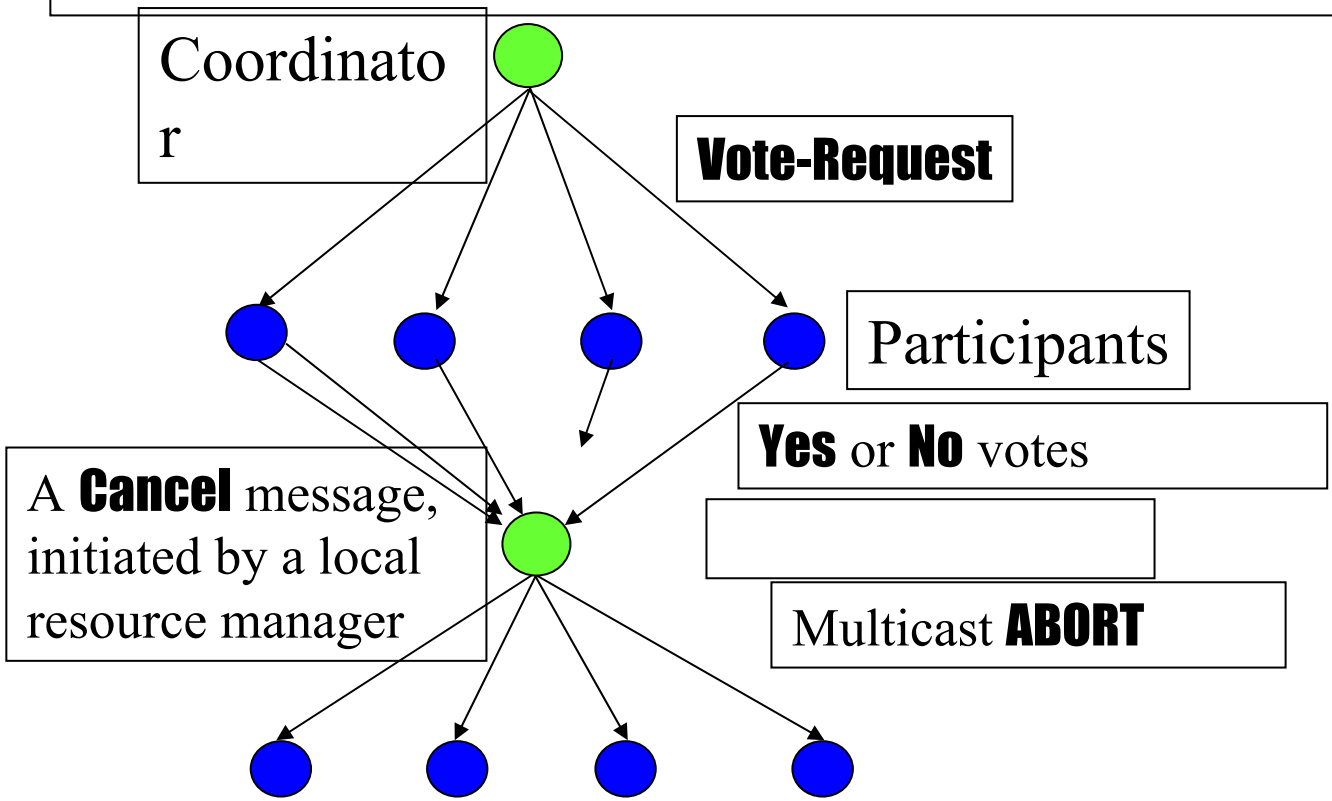
What kind of protocols does the traditional problem definition give?

- First, the protocols try to reach a commit decision, regardless of overall system performance.
- After a timeout, the protocols will try to reach an abort decision, regardless of overall system performance (again).

What should be changed in the problem definition?

- (1) A participant can vote **Yes** or **No**. Having voted, it can try to change its vote.
- (6) If the transaction can be committed and it is feasible to do so for overall efficiency, the decision must be *Commit*.
  If this is not the case and it is still possible to abort the transaction, the decision must be *Abort*.
  - Earlier version of (6) was about failures.

Interactive 2PC



If the Coordinator gets a **Cancel** message before multicasting a decision, it decides to abort.

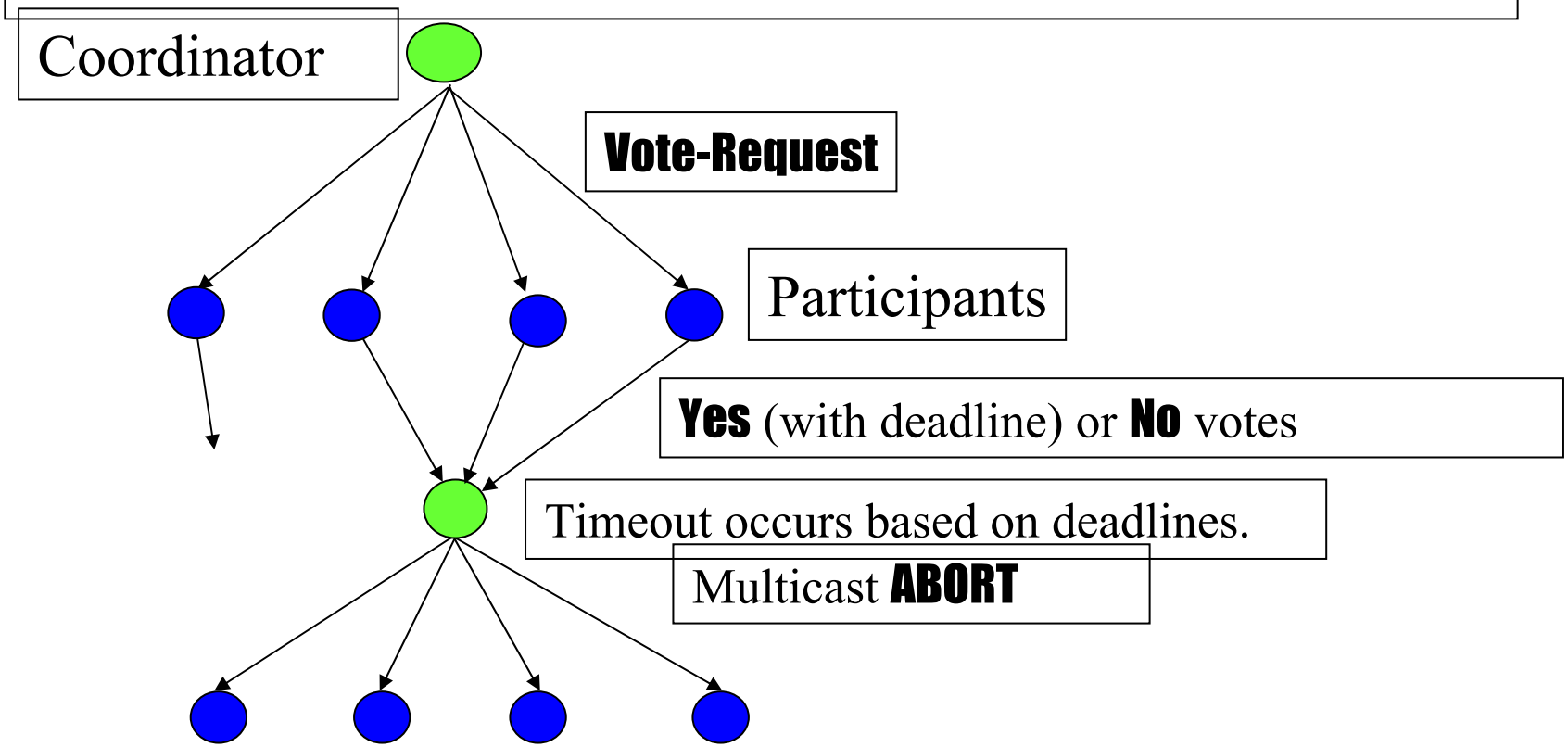
Interactive 2PC - Observations

- There is no need for timeouts for participant failure (only for coordinator).
- There is no need to estimate the transaction duration.
- The mechanism works regardless of the duration of the transaction.
- It is possible to adjust the opinion about the feasibility of the transaction based on the changing situation with lock requests.

Interactive 2PC - Termination

- If a participant gets fed up waiting for the coordinator, it has better check up that the coordinator is still alive – the coordinator may just not have been able to decide and therefore has not sent any messages.

2PC with deadlines



Along with the commit votes, the participants tell how long they are willing to wait, based on local resource manager estimation.

Number of messages

- The deadline protocol does not imply extra messages.
- The interactive protocol only implies extra messages for cancel.
  - The need for extra messages is low.
  - If the information about the abort (due to a **Cancel** message) reaches some participants before they have voted, then the overall number of messages may drop.

Overall performance

- It is easy to see that the new protocols provide more flexibility, which supports overall performance.
- The more often the example situations occur, the more the overall performance improves.

Conclusions

- The interactive protocol provides gives most flexibility.
- There is no real advantage of using basic 2PC over Interactive 2PC (I2PC).
- To benefit from I2PC, the dialogue between the local resource manager and the local participant needs to be improved.
- If you want to use timeouts, it might be better to set them based on deadlines.