CONSENSUS

Murat Demirbas, SUNY Buffalo

several slides borrowed/modified from Jeff Chase, Duke
http://www.cs.duke.edu/~chase/cps212/consensus.pdf

Consensus specification

N nodes; each with value v_i, decision d_i, completion t_i

Agreement: No two process can commit different decisions $\forall i,j : t_i \wedge t_j : d_i = d_j$

Validity: If all initial values are equal, nodes must decide on that value $\exists\, k :: (\forall i :: v_i = k) \Rightarrow (\forall i : t_i : d_i = v_i)$

Termination: Nodes decide eventually $\text{true} \rightarrowtail (\forall i :: t_i)$

Attacking generals problem

- Two armies are on opposite sides of a city in the valley
- The two generals should coordinate the attack; each has an initial value (attack or retreat)
- The only communication is through sending messengers which are prone to being captured/lost in the valley

Attacking generals problem

- Two armies are on opposite sides of a city in the valley
- The two generals should coordinate the attack; each has an initial value (attack or retreat)
- The only communication is through sending messengers which are prone to being captured/lost in the valley

Attacking generals problem

- Two armies are on opposite sides of a city in the valley
- The two generals should coordinate the attack; each has an initial value (attack or retreat)
- The only communication is through sending messengers which are prone to being captured/lost in the valley

No deterministic algorithm for reaching consensus !

Attacking generals ...

Proof is by contradiction Assume such an algorithm exists and completes in r rounds. The algorithm being tolerant to message loss should also work when last message is lost, making it r-1 rounds. This leads to algorithm that decides in 0 rounds, which can be shown to violate agreement.

Applies to both asynchronous & synchronous worlds

Assumes undetected message loss Result does not apply if the sender can reliably detect message loss (without having to rely on ack message which itself is prone to loss)

FLP impossibility result

So we assume reliable communication

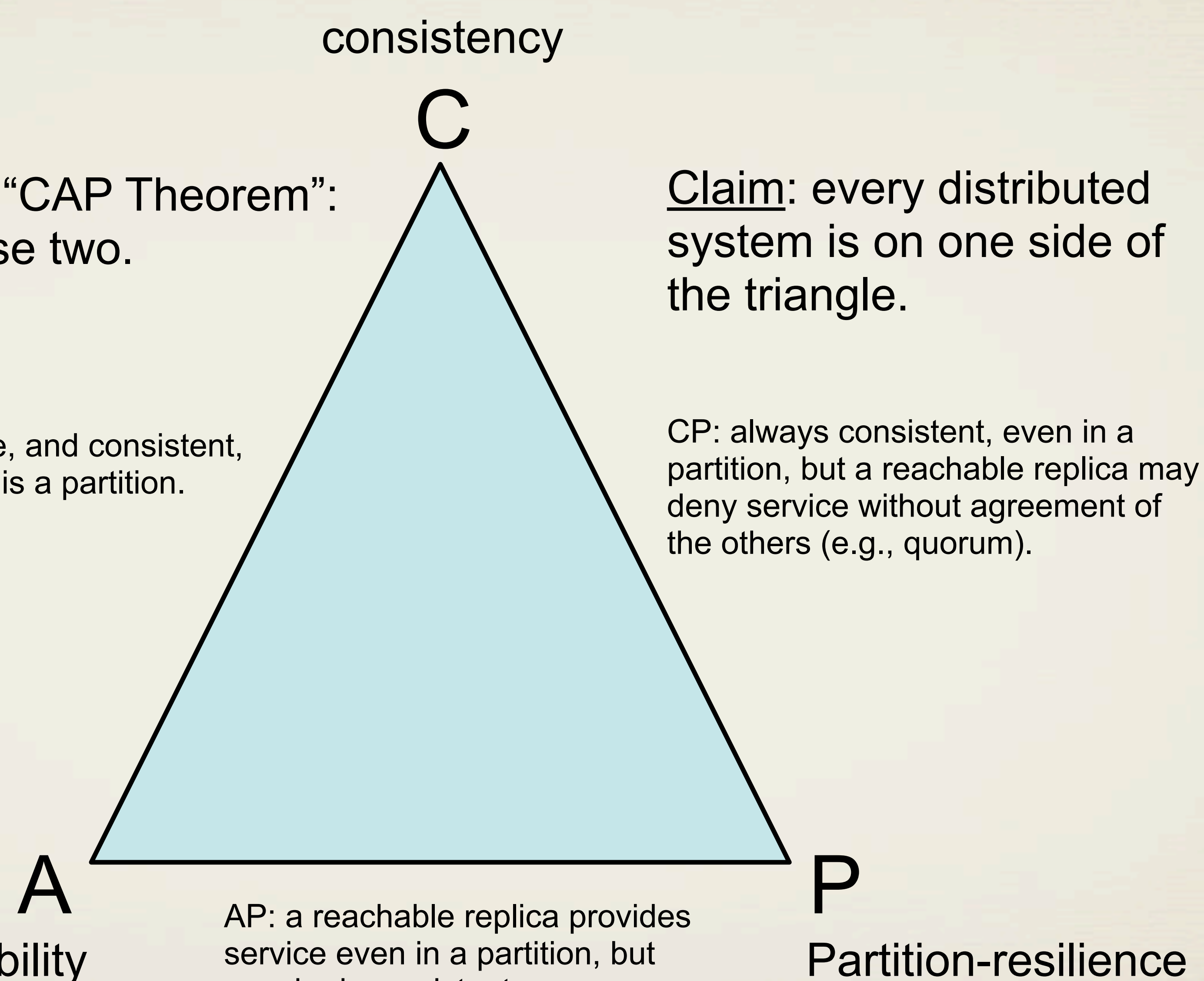
Still no deterministic algorithm for reaching consensus under the asynchronous system model in the presence of just 1 crash failure

The intuition for the proof is: it is impossible for a node to determine whether the critical node for decision is dead or taking too long to answer

What do these imply?

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

Claim: every distributed system is on one side of the triangle.



CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

AP: a reachable replica provides service even in a partition, but may be inconsistent.

CAP examples

CP: Paxos, or any consensus algorithm, or state machine replication with a quorum required for service

Always consistent, even in a partition. But might not be available, even without a partition.

AP: Bayou, Amazon-Dynamo

Always available if any replica is up and reachable, even in a partition. But might not be consistent, even without a partition.

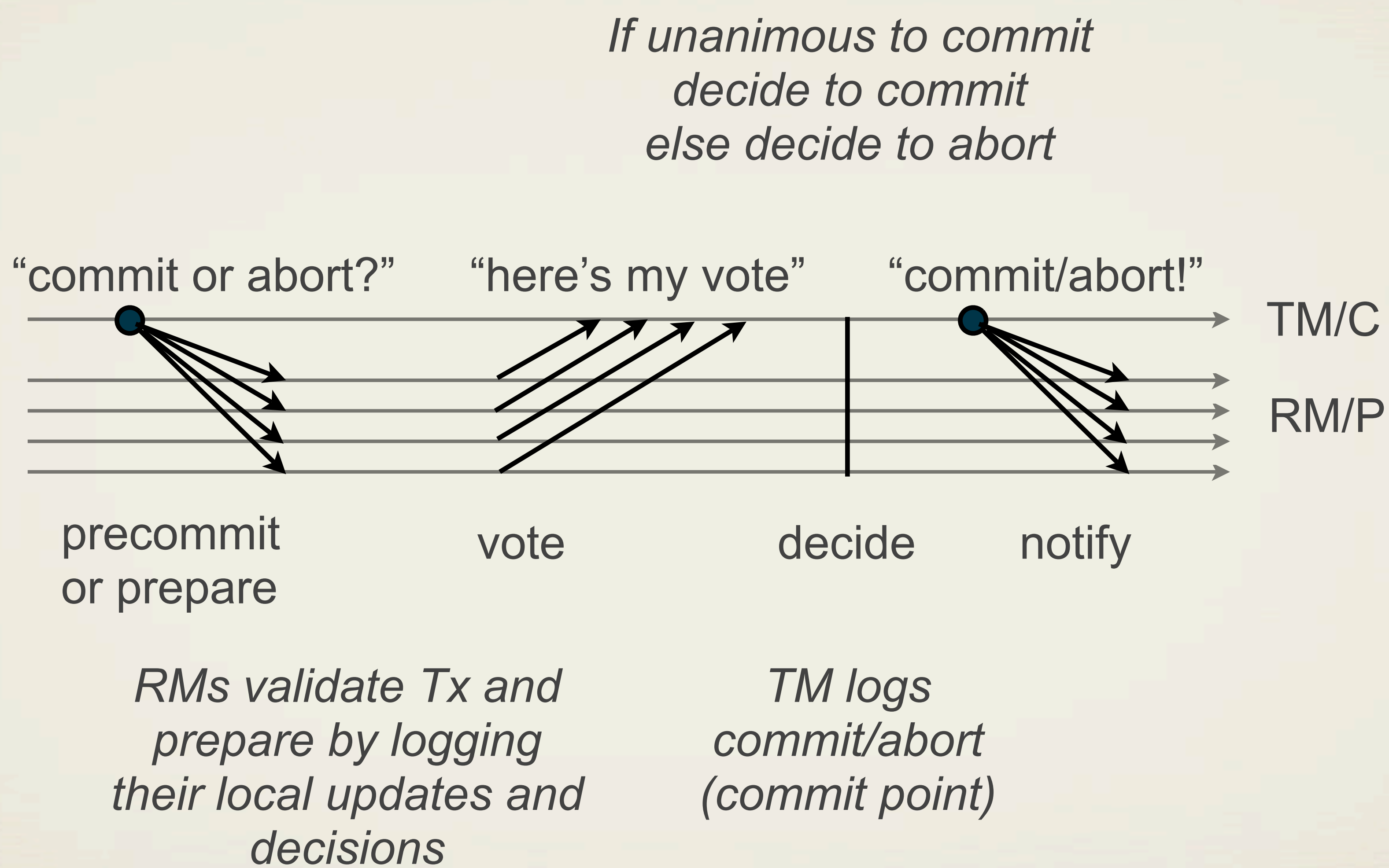
CA: consistent replication (e.g., state machine with CATOCS) with service from any replica

Always consistent in the absence of a partition. But may become inconsistent in a partition. Split brain syndrome

We will focus on CP

We will study the 2-phase commit, 3-phase commit, and Paxos algorithms next

2 phase commit



2PC blocks and becomes unavailable if TM fails !

Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

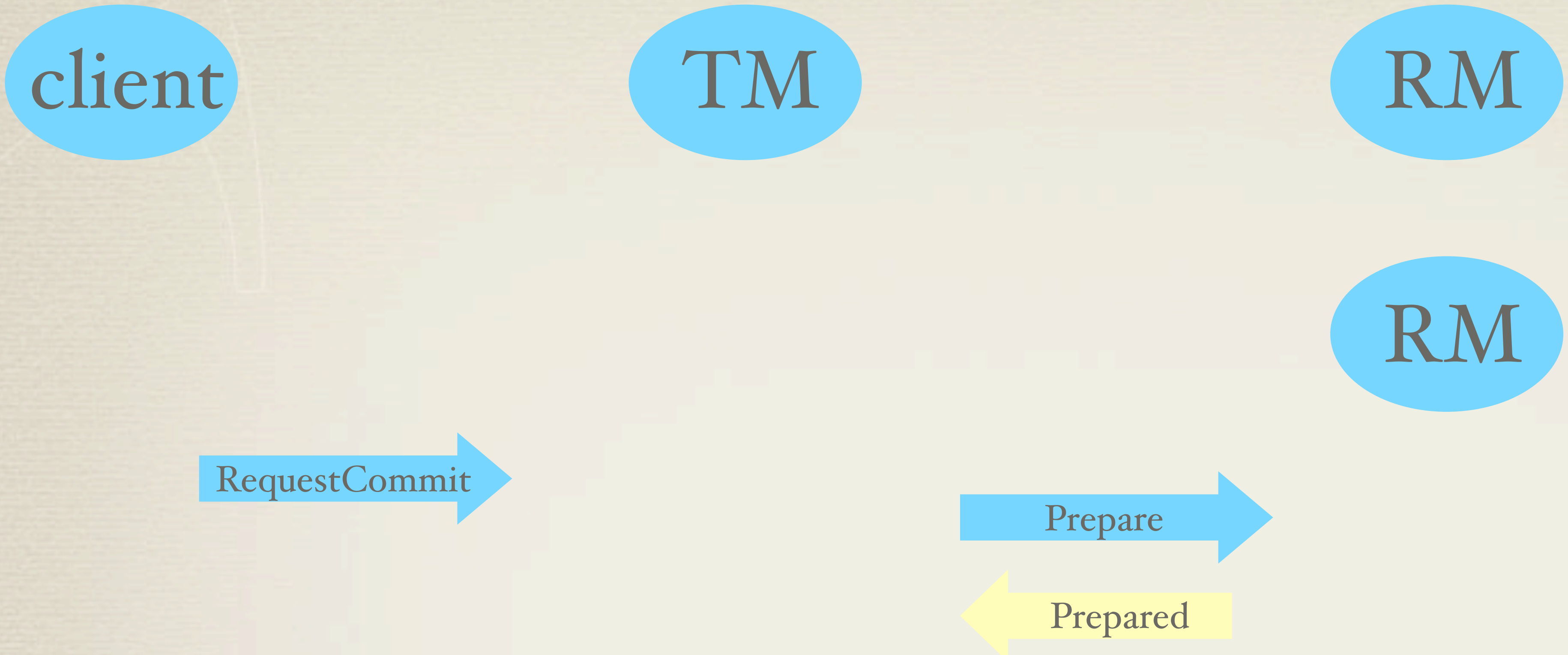
Fault-Tolerant Two Phase Commit

If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit

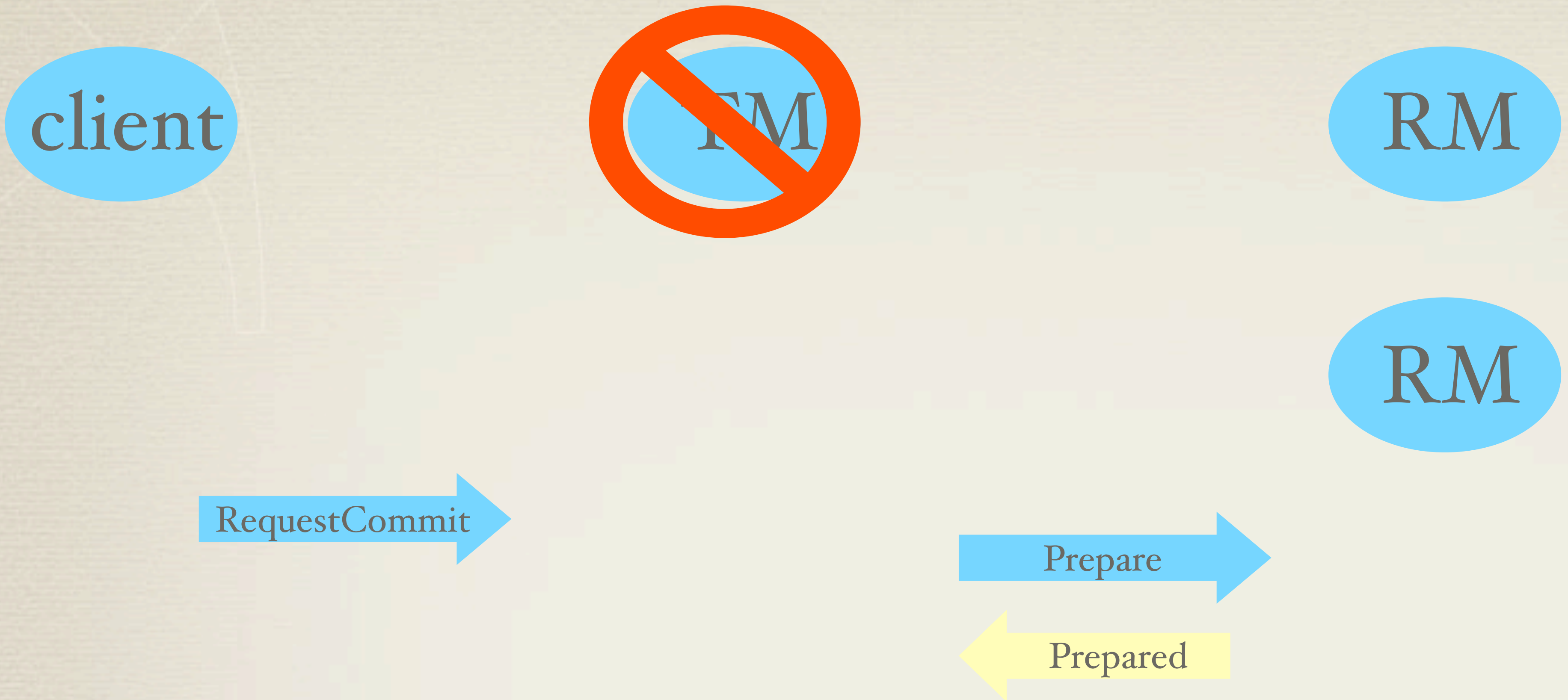
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit



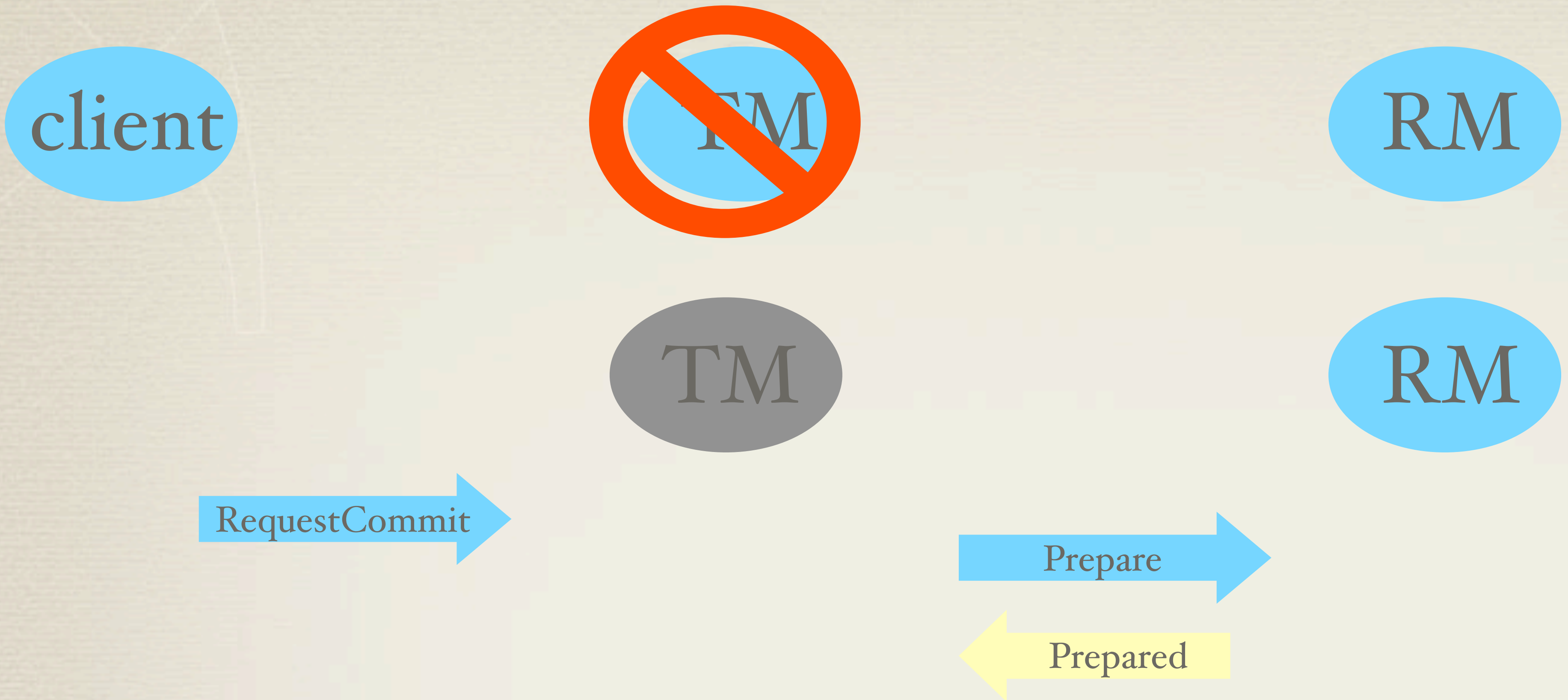
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit

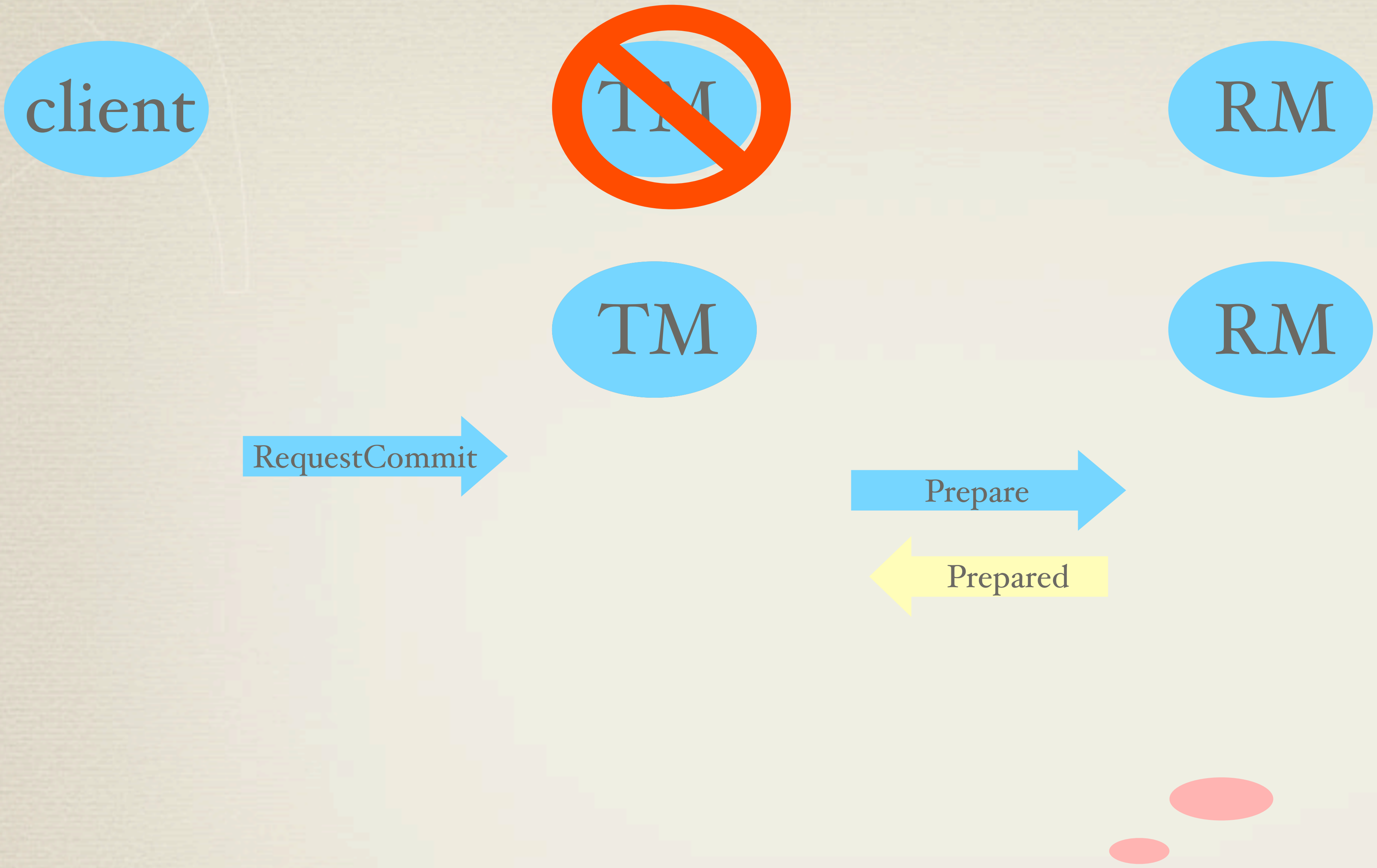


If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Solution: Add a "spare" transaction manager

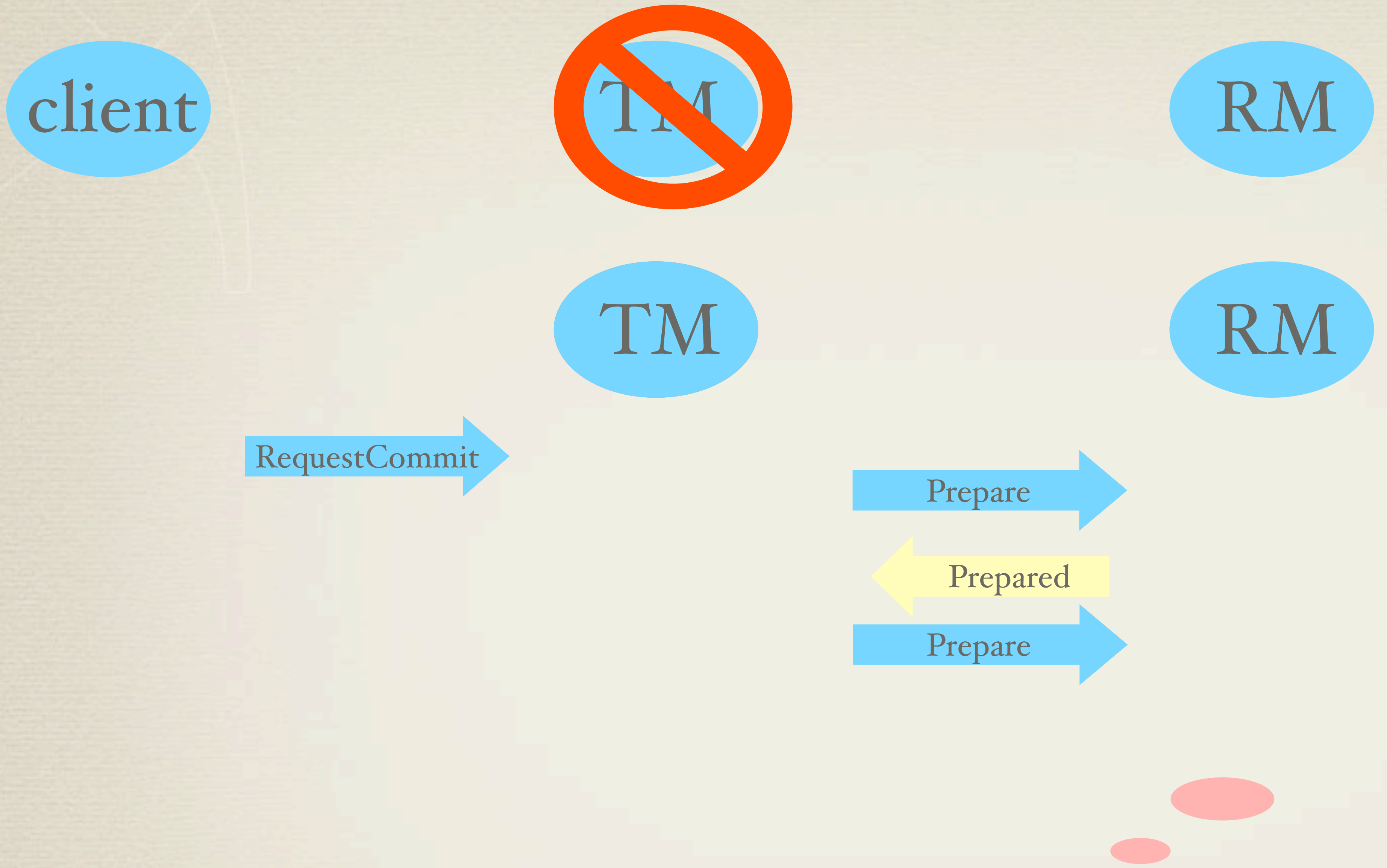
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



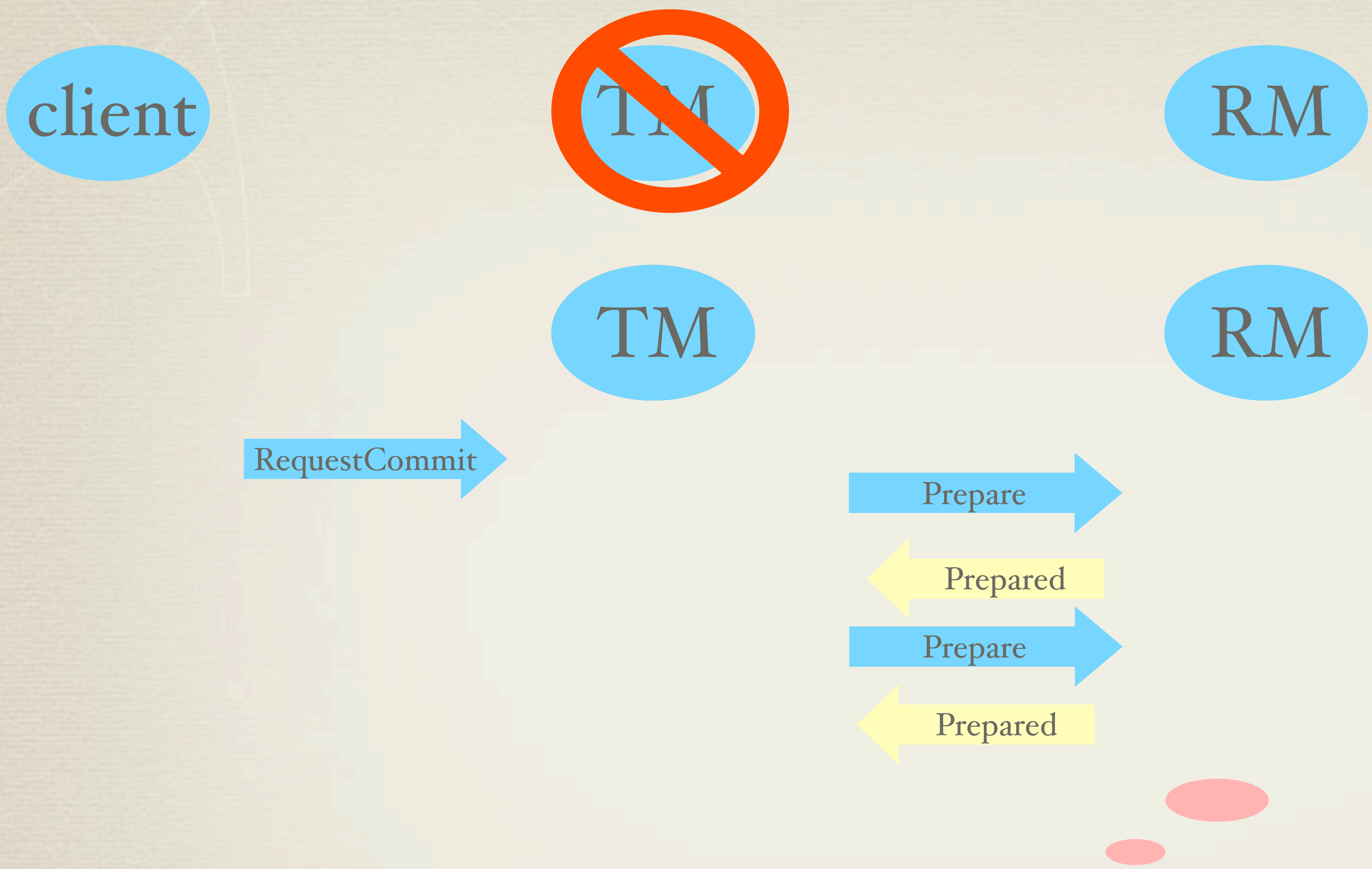
If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



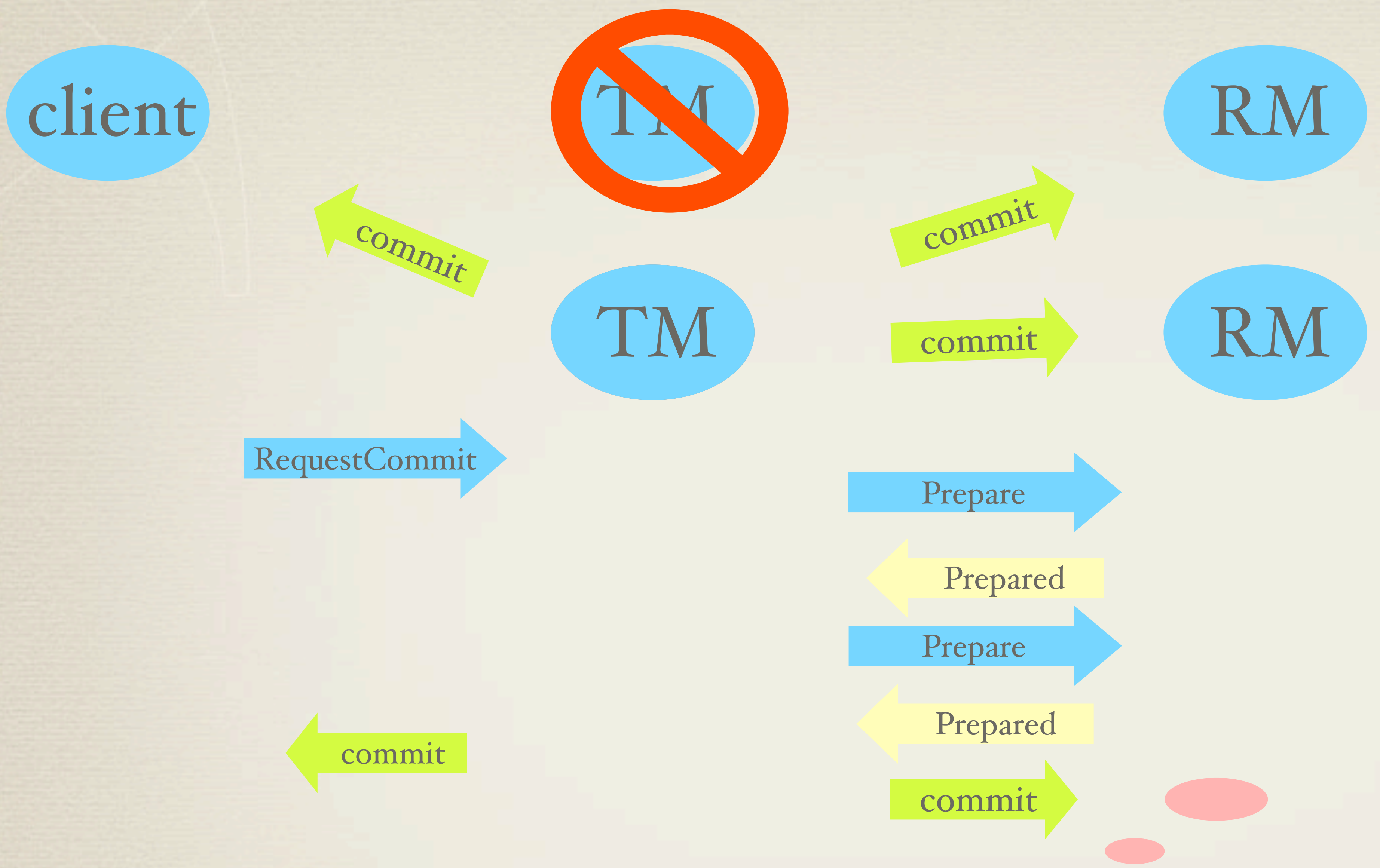
If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



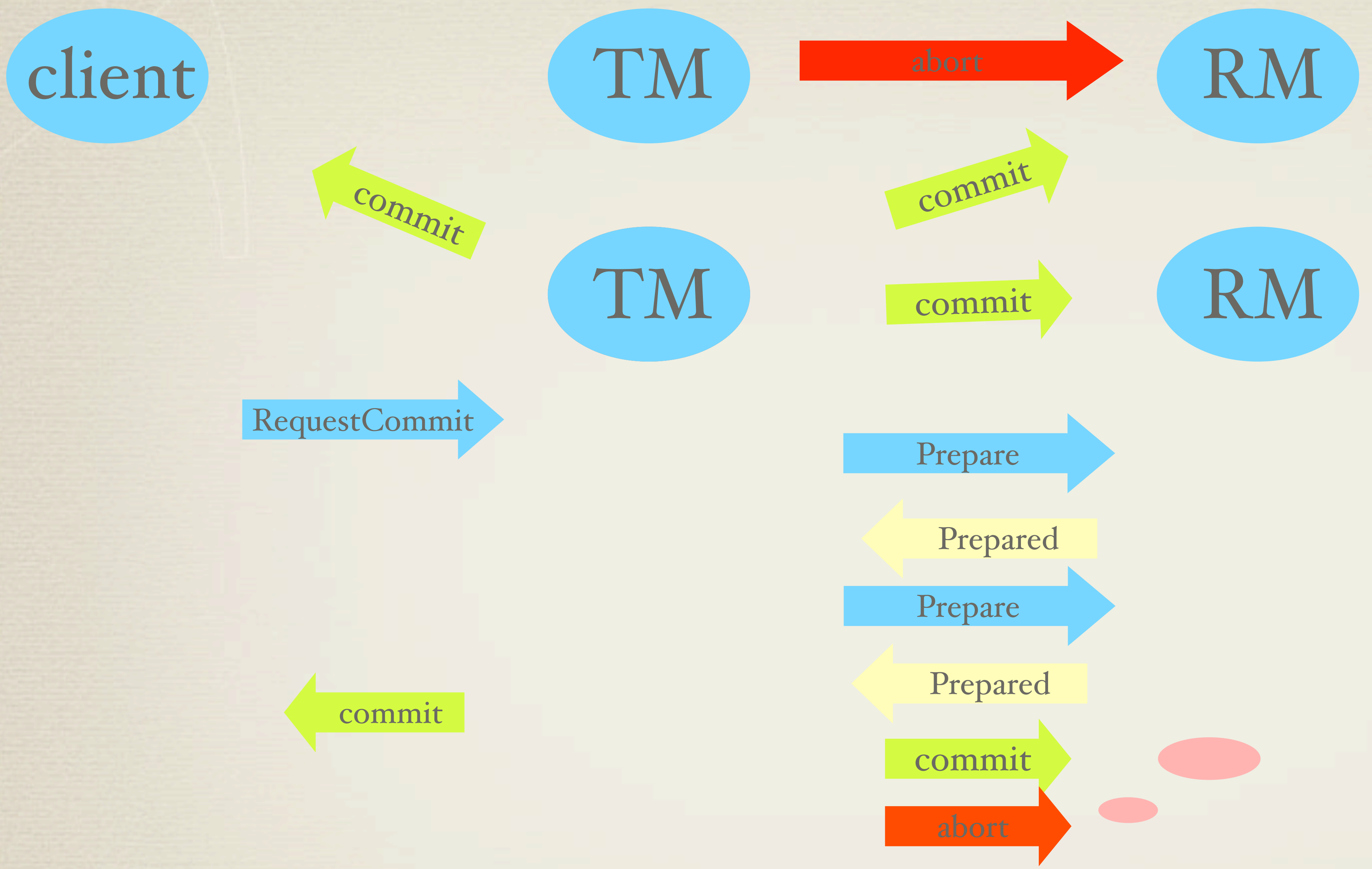
If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

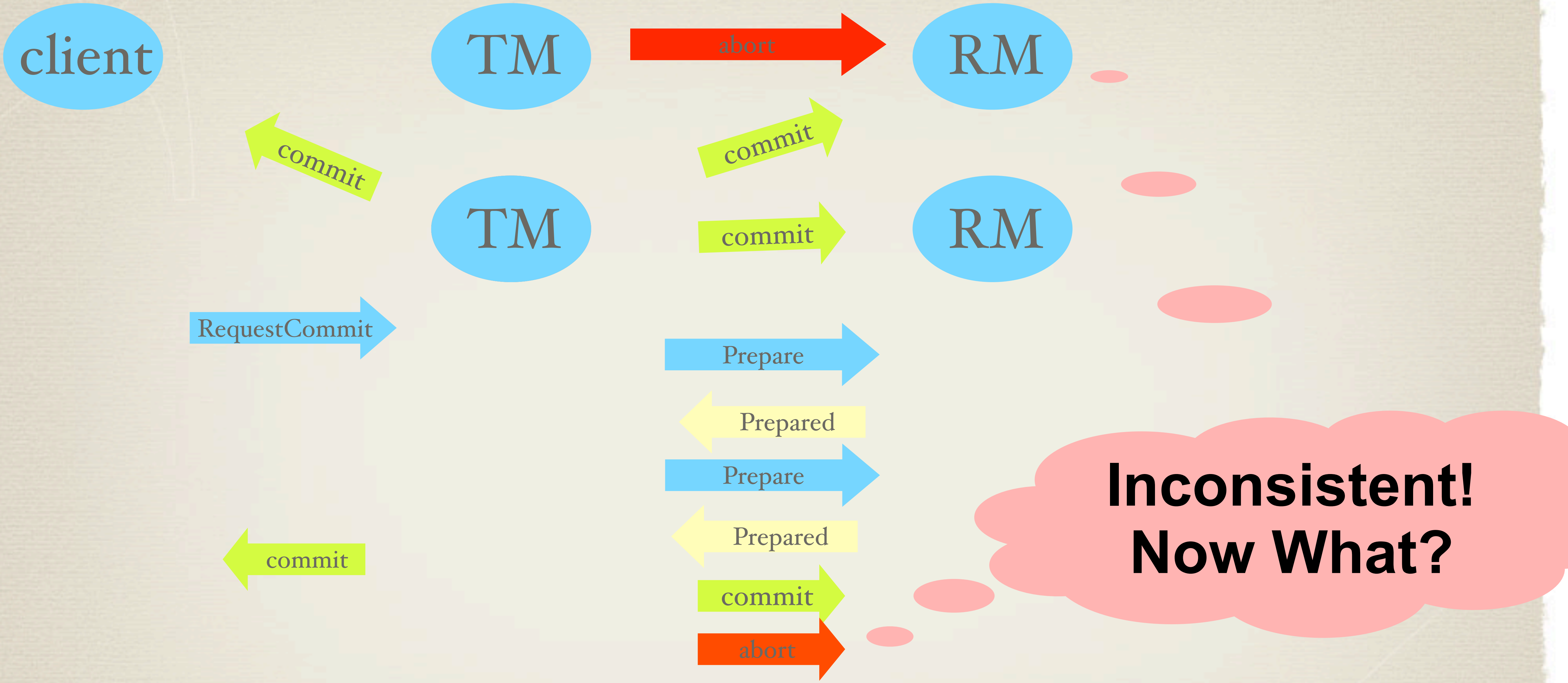
Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

But… What if….?

Fault-Tolerant Two Phase Commit



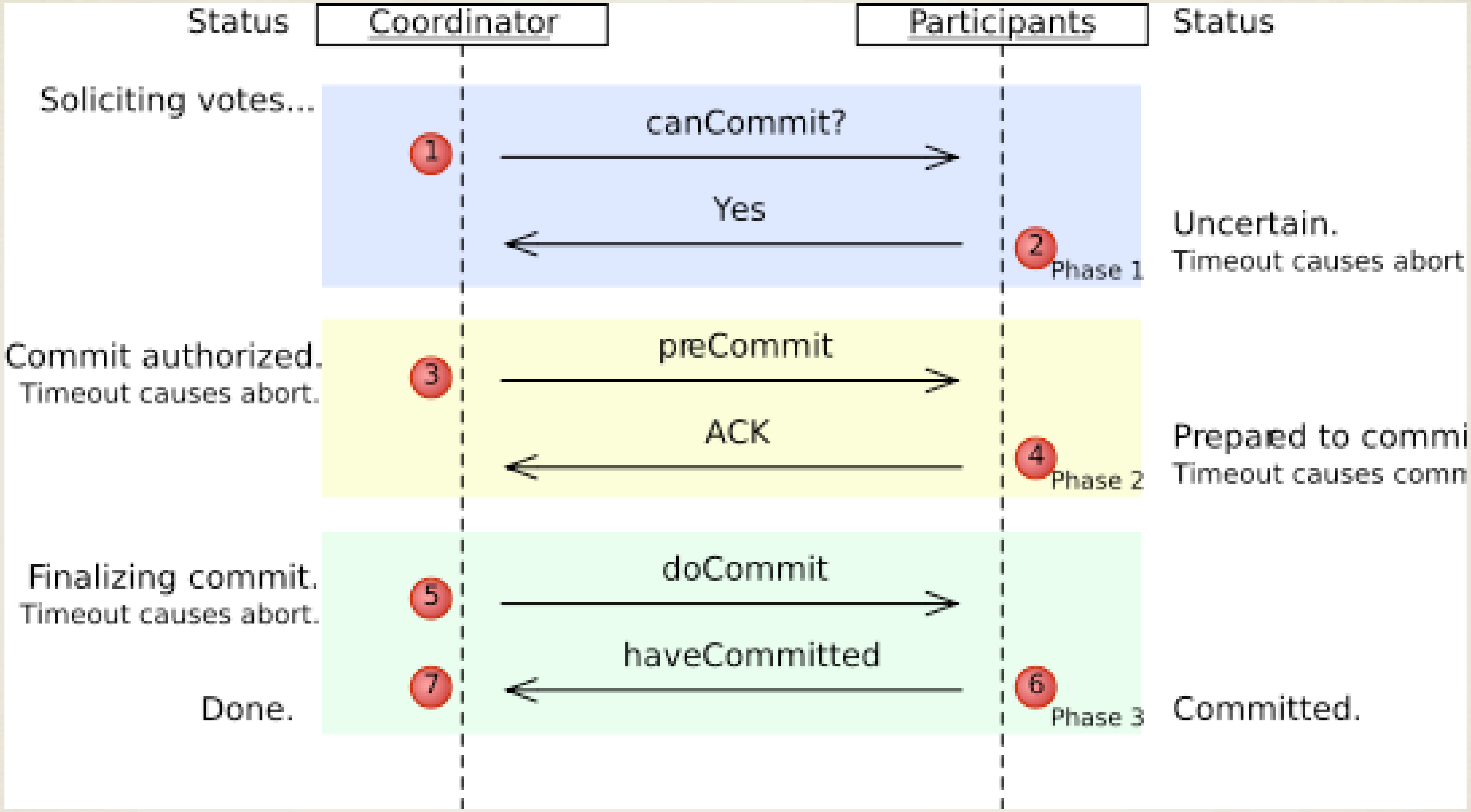
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Solution: Add a "spare" transaction manager

(non blocking commit, 3 phase commit)

But… What if….? The complexity is a mess.

3 phase commit



But, none of the 3PC proposals come with a complete algorithm and correctness proof. (Jim Gray 2004)

Paxos, the best so far...

- A 3PC protocol that works, with correctness proof
- Preserves safety against asynchrony & message loss
- Achieves progress when the conditions improve

How does Paxos work?

There are N nodes, some (ideally one) act as a leader

Leader presents a consensus value to the acceptors, counts the ballots for acceptance of the majority, and notifies acceptors of success

Paxos can mask failure of a minority of N nodes

Rounds and ballots

Paxos proceeds in rounds. Each round has a uniquely numbered ballot. Each round has 3 phases.

If no failures, then consensus is reached in one round.

Any would-be leader can start a new round on any (apparent) failure.

Consensus is reached when some leader successfully completes a round.

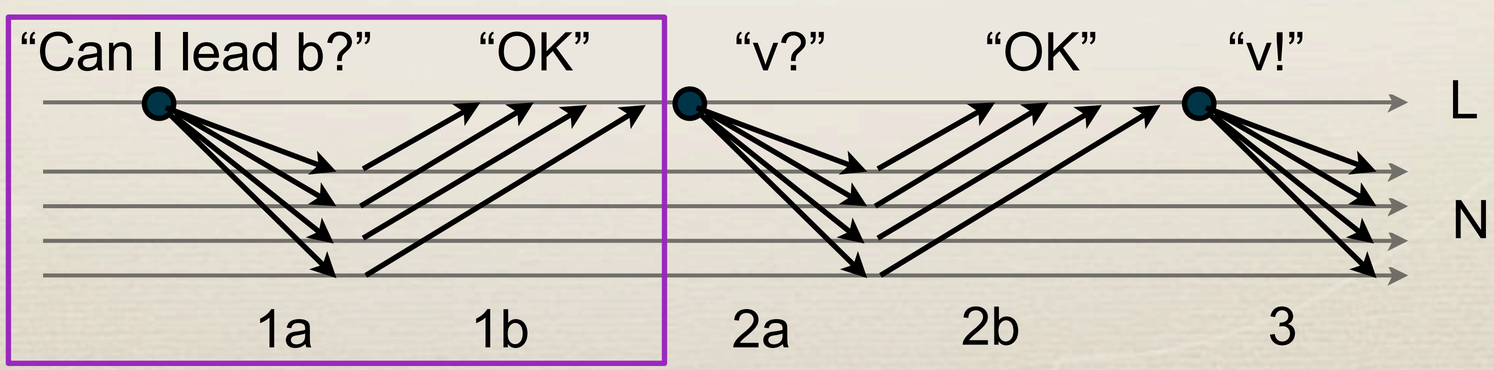
Phase 1: Proposing a round

Would-be leader chooses a unique ballot ID (round#)

Propose to the acceptors/agents (1a). Will you consider this ballot with me as leader?

Agents return the highest ballot ID seen so far (1b). Seen one higher than yours? That's a rejection.

If a majority respond and no one knows of a higher ballot number, then you are their leader (for this round)



Phase 2-3: Leading a round

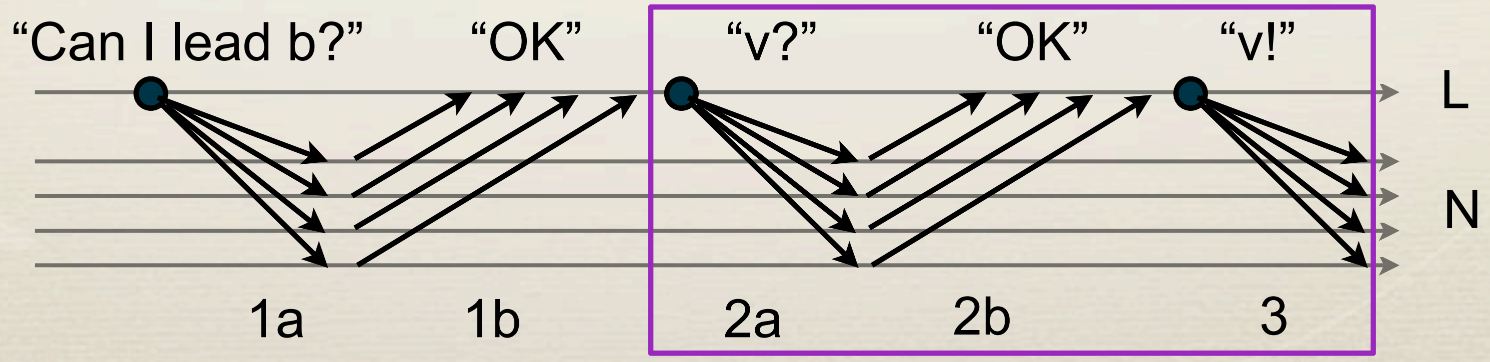
Choose a “suitable value”™ for this ballot.

Command the agents to accept the value (2a).

Did a majority respond (2b) and assent?

Yes: tell everyone the round succeeded (3).

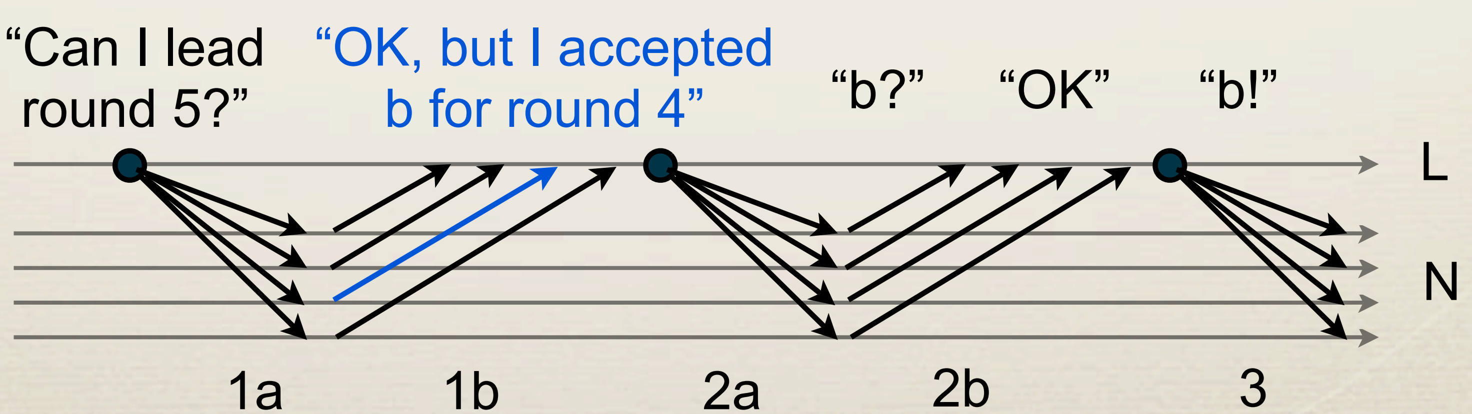
No: move on, e.g., ask for another round.



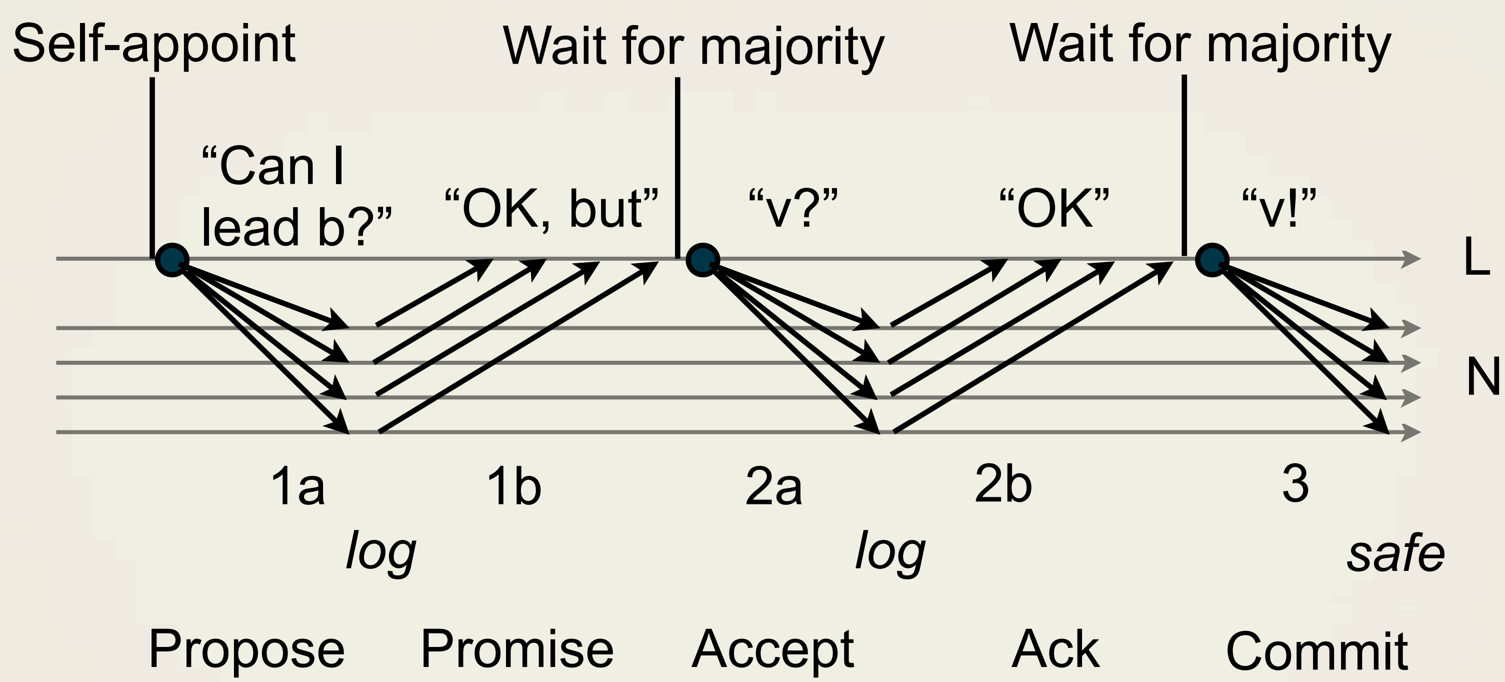
Choosing a suitable value

A majority of agents responded (1b). Did any accept a value for some previous ballot (in a 2b message)? No: choose any value you want.

Yes: they tell you the ballot ID and value. Find the most recent value that any responding agent accepted, and choose it for this ballot too.



Where is the anchor point?



Anchoring a value

A round anchors if a majority of agents hear the command (2a) and obey. (that value is chosen/anchored, even though no nodes may individually know of this!)

The round may then fail if many agents fail, many command messages are lost, or another leader usurps

But: safety requires that once some round anchors, no subsequent round can change it.

The system may have another round, possibly with a different leader, until all nodes learn of the success.

Class reenactment

5 students come to the board. Each student draws a ledger for round# and value.

scenario 1: fault-free, the leader completes in a round

scenario 2: leader dies after making majority accept, one node becomes leader, re-learns and commits value

scenario 3: leader dies before making majority accept, one node becomes leader, commits another value, old leader wakes up, re-learns and commits value.

Class reenactment ...

scenario 4: two leaders duel, progress violated, as the leaders keep undoing what the other tries to do. Finally, one leader drops the race, the remaining leader commits

scenario 5: the network is partitioning. If a majority partition exists it makes progress. Other partitions cannot make progress. Progress is made when network is connected again.

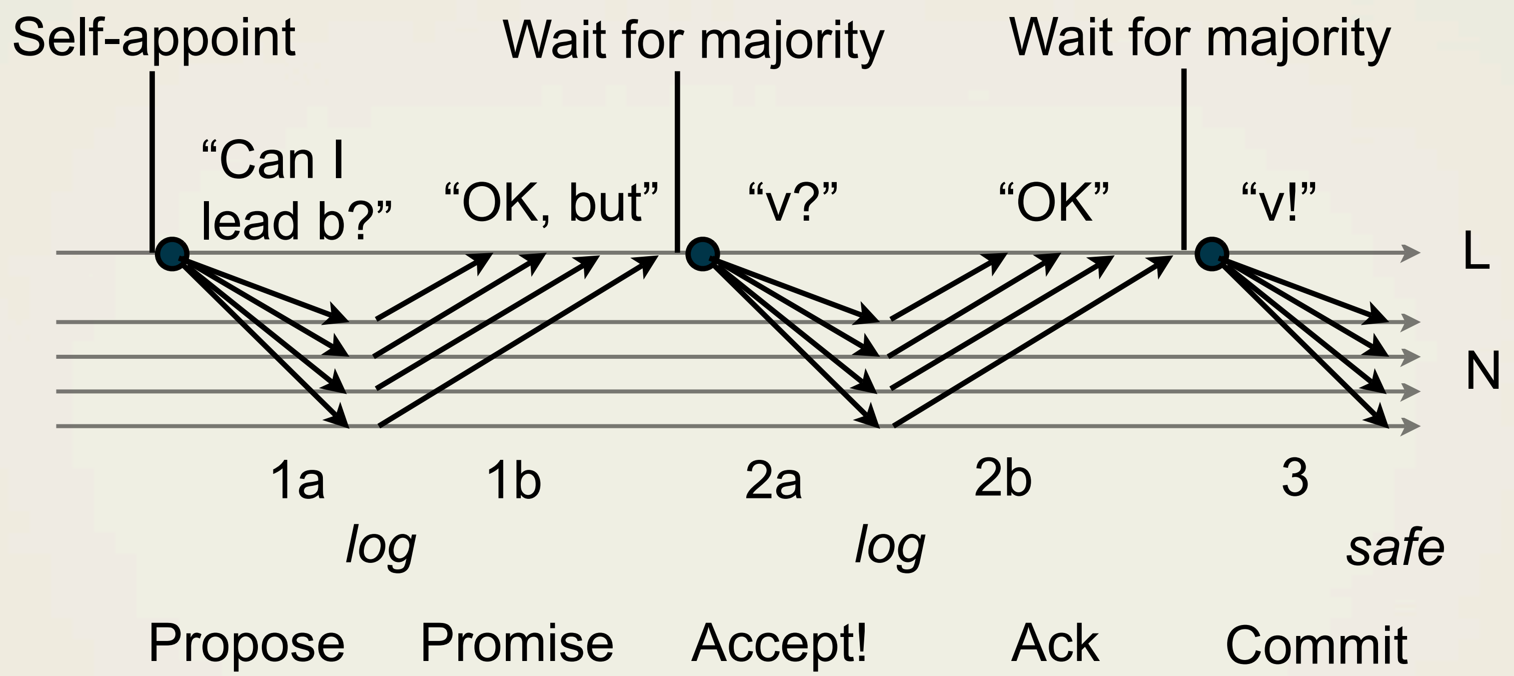
Why does Paxos work?

Key invariant: If some round commits, then any subsequent round chooses the same value, or it fails.

Consider leader L of a round R that follows a successful round P with value v, then either L learns of (P, v), or else R fails. Why? P got responses from a majority: if R does too, then some agent responds to both.

If L does learn of (P, v), then by the rules of Paxos L chooses v as a "suitable value"™ for R.

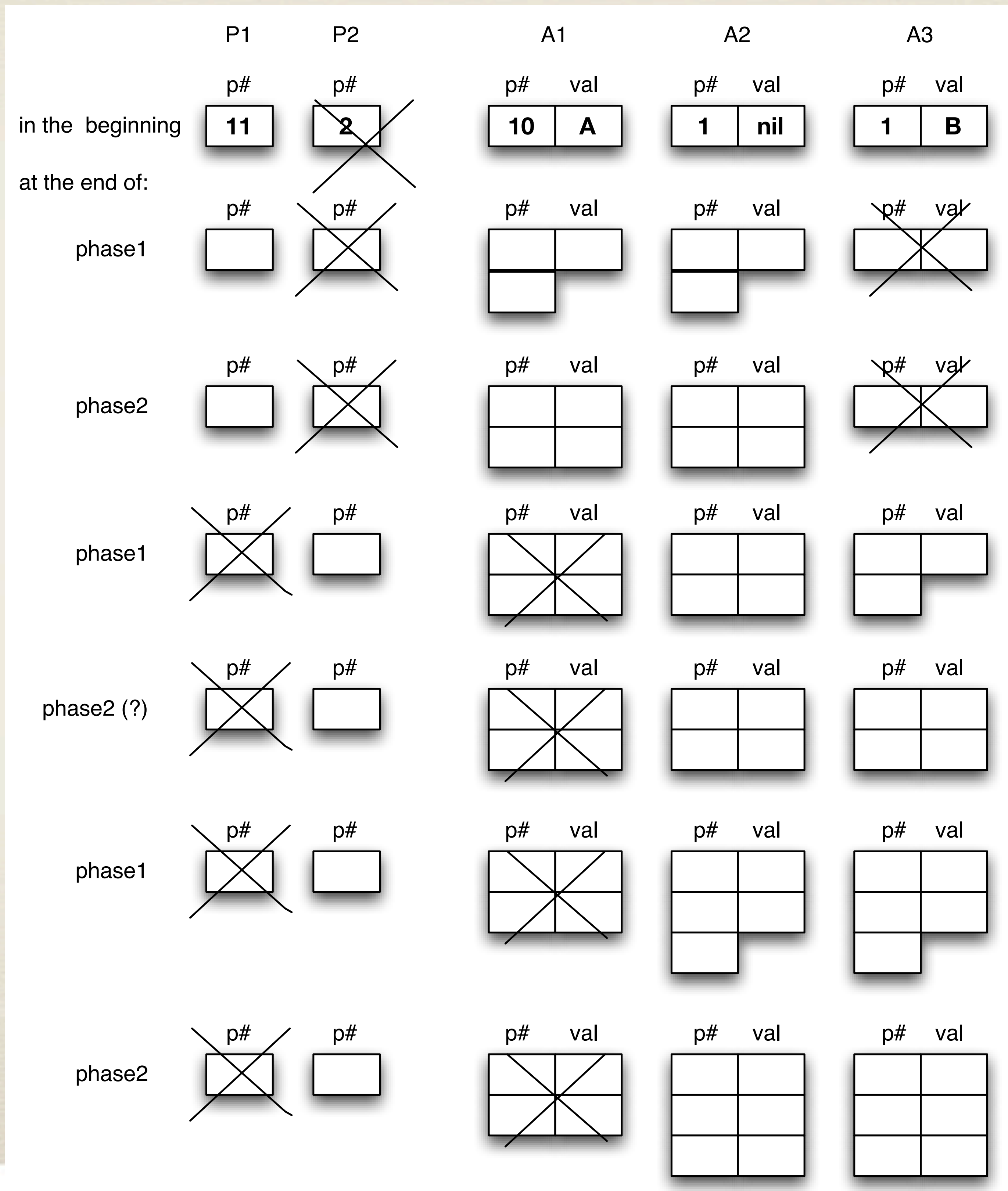
Paxos summary



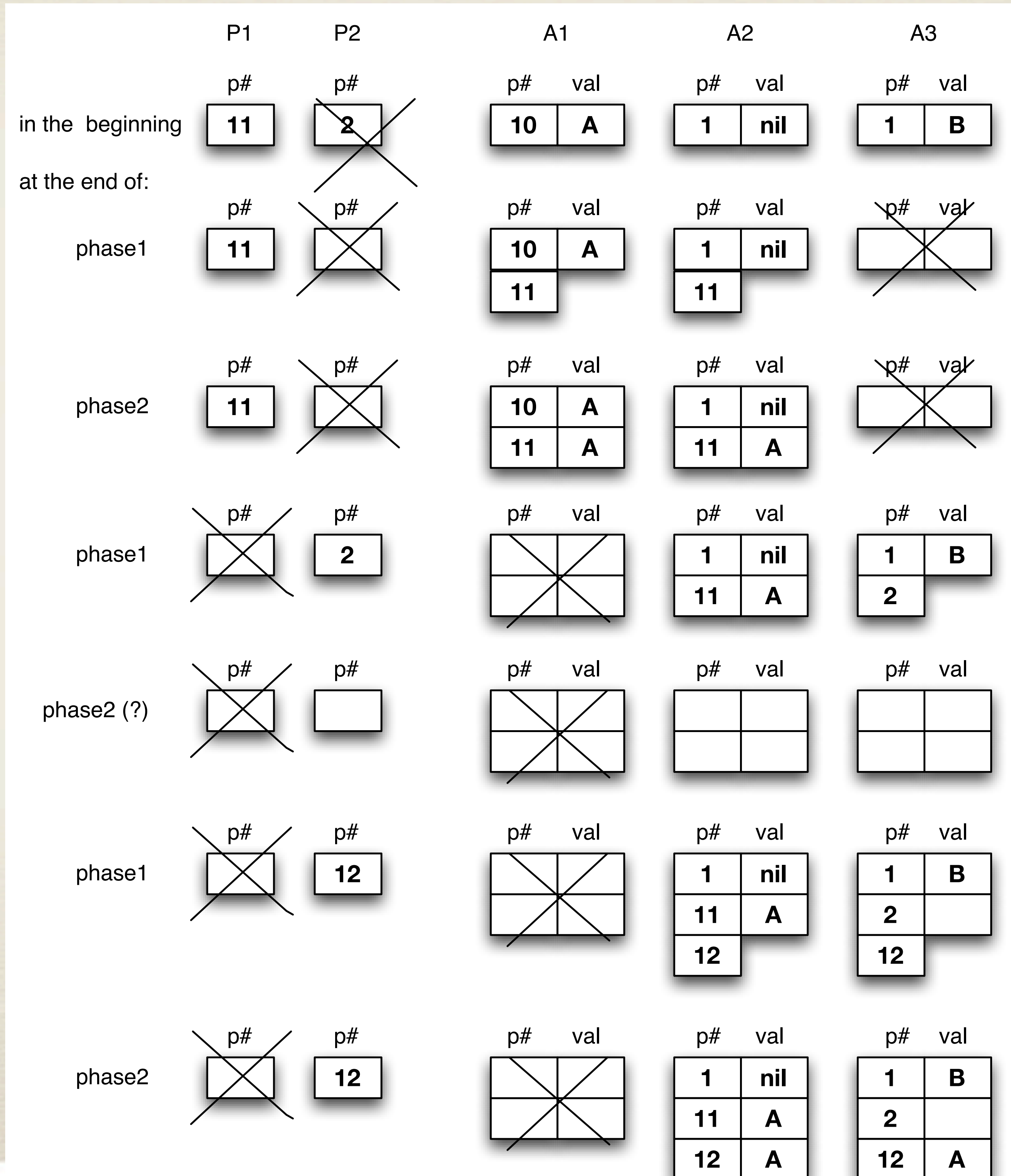
Paxos can be made efficient, and serve as the building block of highly consistent and partition-resilient systems with pretty good best-effort availability

Paxos exercise (again)

7. Paxos (10 points) Consider a system running the Paxos consensus algorithm with two proposers, $P1$, $P2$, and three acceptors, $A1$, $A2$, and $A3$. In the system the channels are reliable, so there are no message losses. However, nodes can crash, which we denote as putting a cross on the node. No learning takes place among acceptors. With these in mind, fill out the empty boxes in the following execution. $p\#$ denotes the proposal number and the val denotes the value of the proposal.



7. Paxos (10 points) Consider a system running the Paxos consensus algorithm with two proposers, $P1$, $P2$, and three acceptors, $A1$, $A2$, and $A3$. In the system the channels are reliable, so there are no message losses. However, nodes can crash, which we denote as putting a cross on the node. No learning takes place among acceptors. With these in mind, fill out the empty boxes in the following execution. $p\#$ denotes the proposal number and the val denotes the value of the proposal.



PAXOS MADE LIVE

Chandra, Griesemer, Redstone
Google 2007

Chubby

Chubby is a fault-tolerant system at Google. Typically there is one Chubby instance ("cell") per data center

Several Google systems (Google-Filesystem, Bigtable,..) use Chubby for distributed coordination/locking and to store a small amount of metadata.

Chubby ...

Chubby achieves fault-tolerance through replication. Chubby cell consists of 5 replicas. One replica is master

Every Chubby object (lock or file) is stored as an entry in a database. It is this database that is replicated.

Clients contact cell for service. Master replica serves all requests. If client contacts a replica, replica replies with master's network address. Client then contacts master.

How does Paxos fit here?

Each replica maintains a local copy of the request log

Paxos is used for ensuring all replicas have identical sequences of entries in their local logs despite faults

This is a standard replicated state machine approach to fault-tolerance

Multi-Paxos

This is an optimization to reduce the number of phases involved by chaining together multiple Paxos instances Propose messages can be omitted if the leader identity does not change between instances.

This does not interfere with the properties of Paxos because any replica at any time can still try to become a leader by broadcasting a propose message with a higher round/ballot number

Master leases

Reads of the data structure require executing Paxos

Read operations cannot be served out of the master's copy of the data structure because other replicas may have elected another master and modified the data structure without notifying the old master

Since read operations comprise a large fraction of all operations, serializing reads through Paxos is expensive

Master leases mechanism of Paxos solves this problem

Leader is chosen to serve until lease expires. Replicas refuse to process messages from another master while lease holds

Master leases ...

If the master has the lease, it is guaranteed that other replicas cannot successfully submit values to Paxos

Thus a master with the lease has up-to-date information in its local data structure which can be used to serve a read operation purely locally

By making the master attempt to renew its lease before it expires we can ensure that a master has a lease most of the time

If the master fails or gets stuck in a minority partition, when the lease expires another replica can become a master and continue execution as per Paxos rules

PAXOS COMMIT

Jim Gray, Leslie Lamport 2004

slides are from Gray's MS Research Techfest presentation

Paxos vs 2PC

The fundamental difference is that leader failure can block 2PC, while Paxos is non-blocking

There are some differences in problem setting

2PC: agents have multiple choice with veto power. Unanimity is required to commit. Agents are resource managers, rather than replicas.

Paxos: consensus value is dictated by the first leader to control a majority of replicas

Can we derive a nonblocking commit from Paxos?

Paxos Commit

Jim Gray
Leslie Lamport

Microsoft Research
Preview of a paper in preparation
Presented Microsoft Research Techfest
3 March 2004,
Redmond, WA

Article
MSR-TR-2003-96
Consensus on Transaction Commit
[***http://research.microsoft.com/research/pubs/view.aspx?tr_id=701***](http://research.microsoft.com/research/pubs/view.aspx?tr_id=701)

Commit is Common

- Marriage ceremony

Do you?
I do.
I now pronounce you…

- Theater

Ready on the set?
Ready!
Action!

- Contract law

Offer
Signature
Deal / lawsuit

The Common Picture

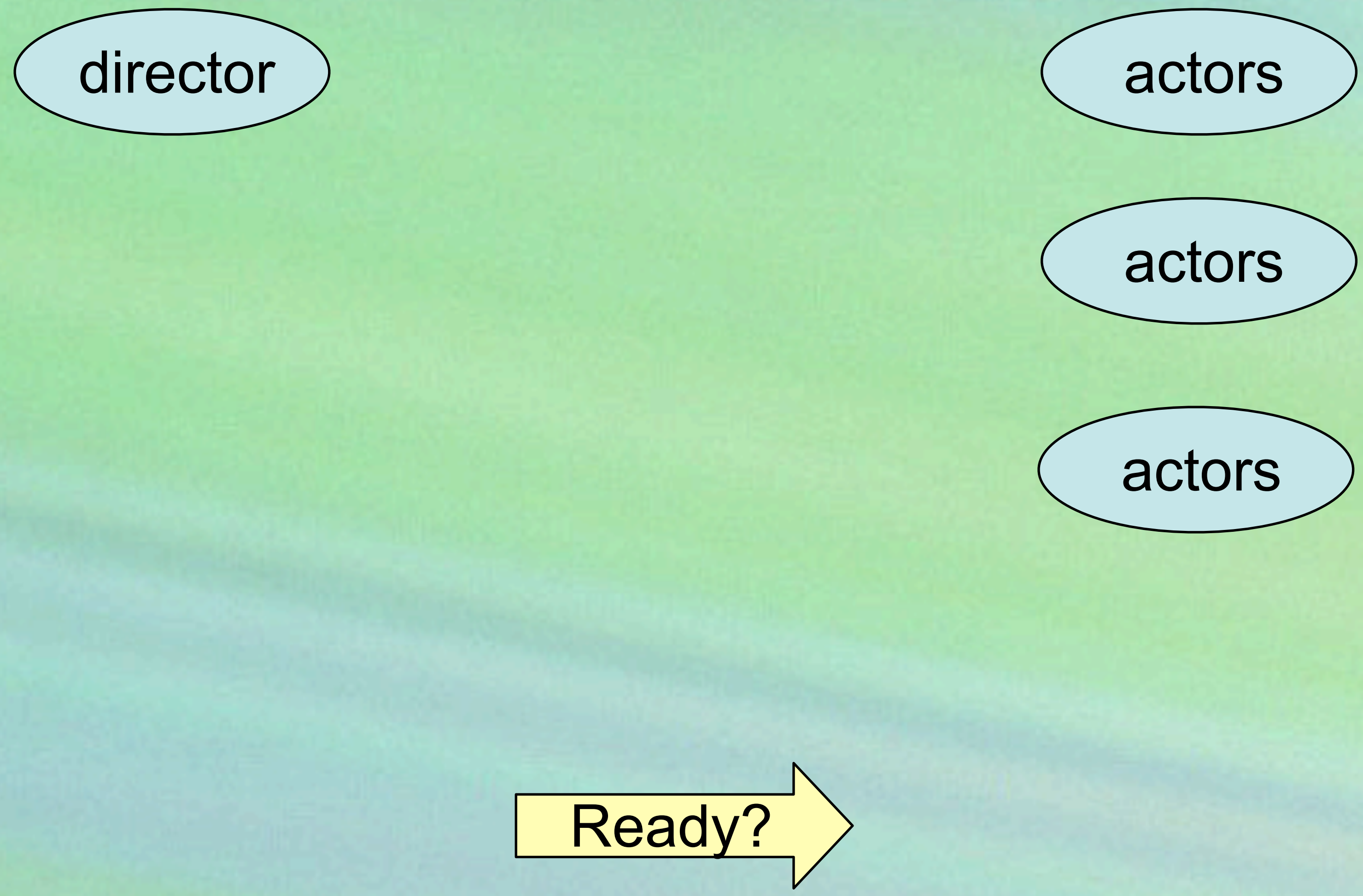
director

actors

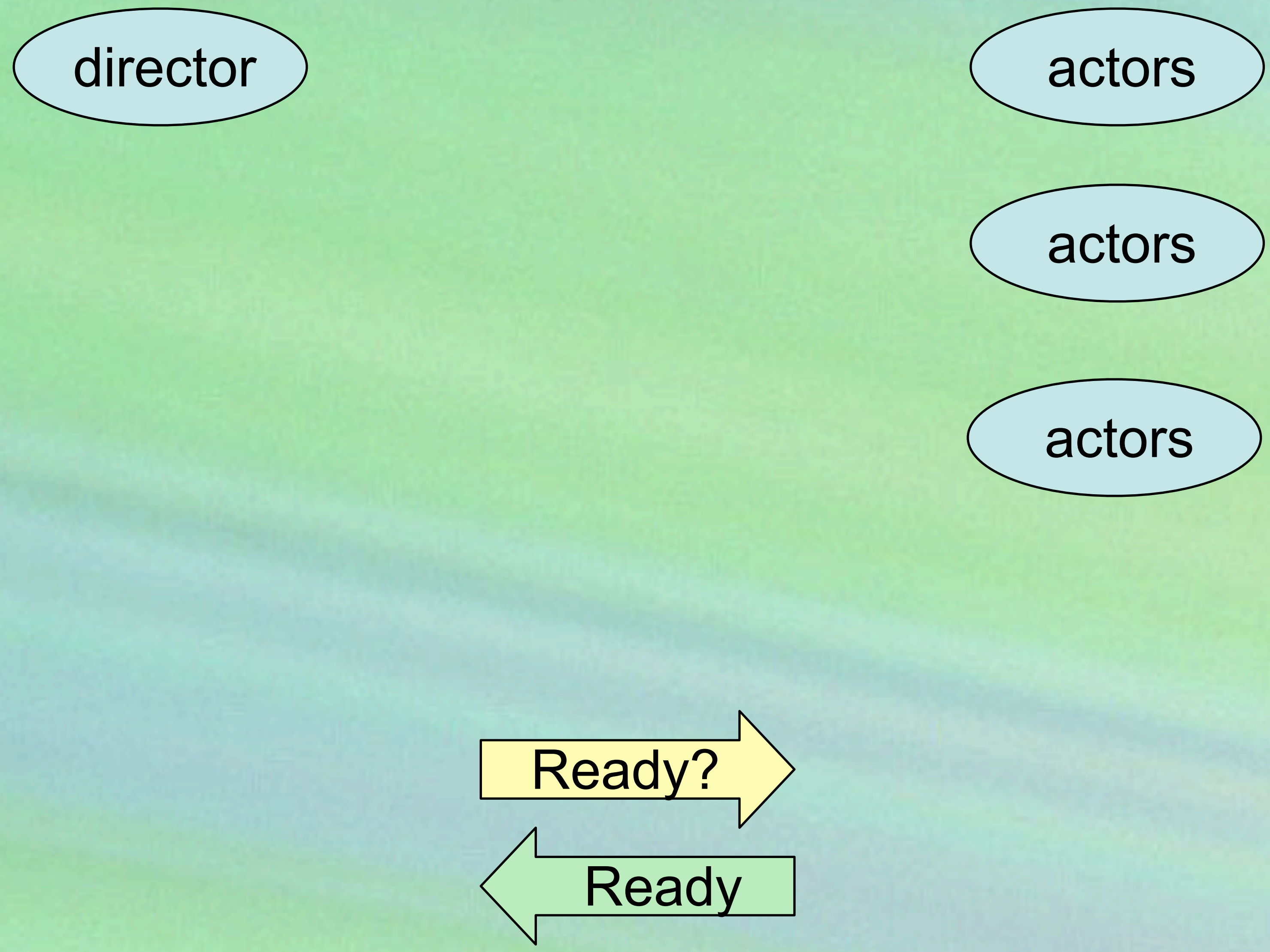
actors

actors

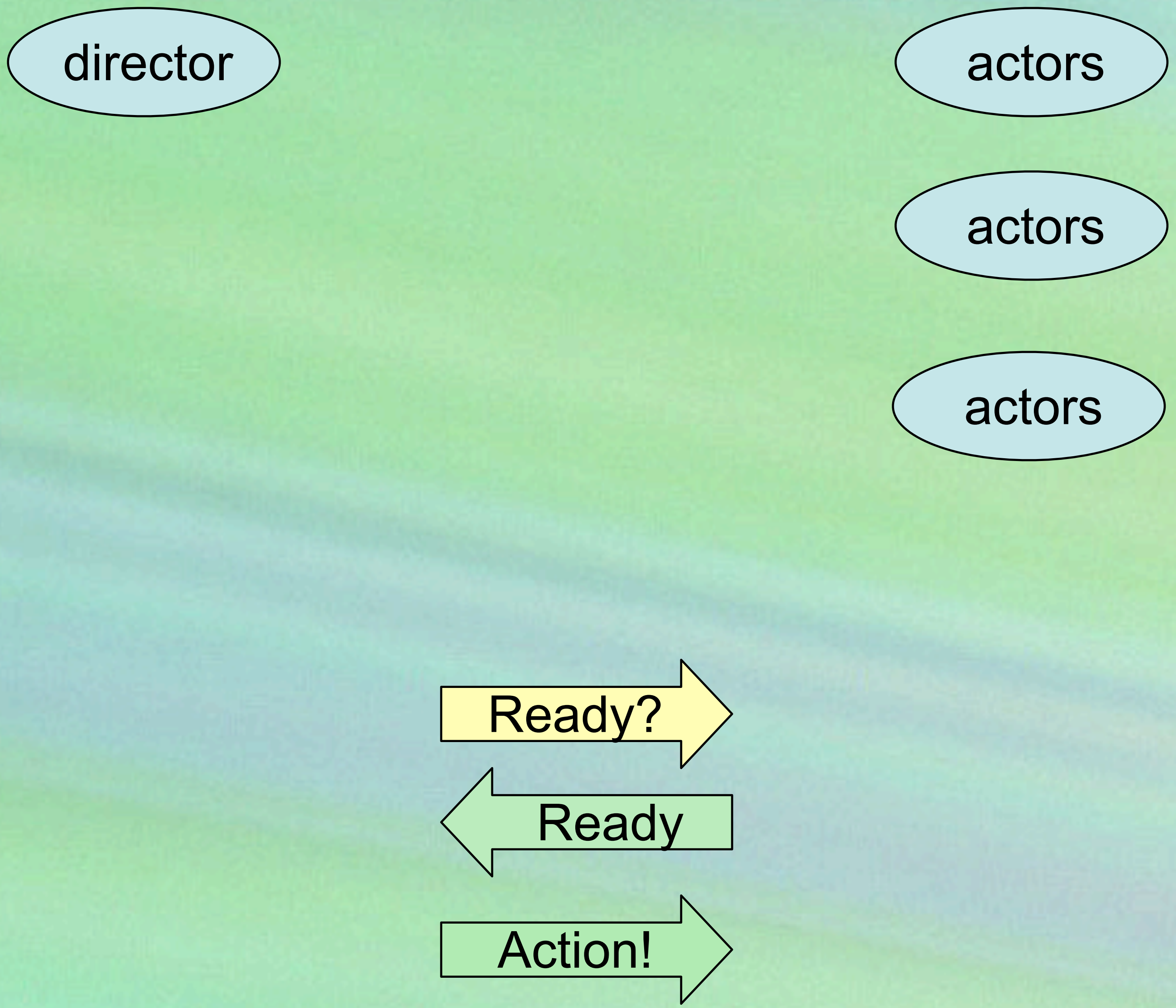
The Common Picture



The Common Picture



The Common Picture



All or Nothing:
If any actor says no the deal is off.

director

actors

actors

actors

All or Nothing:
If any actor says no the deal is off.

director

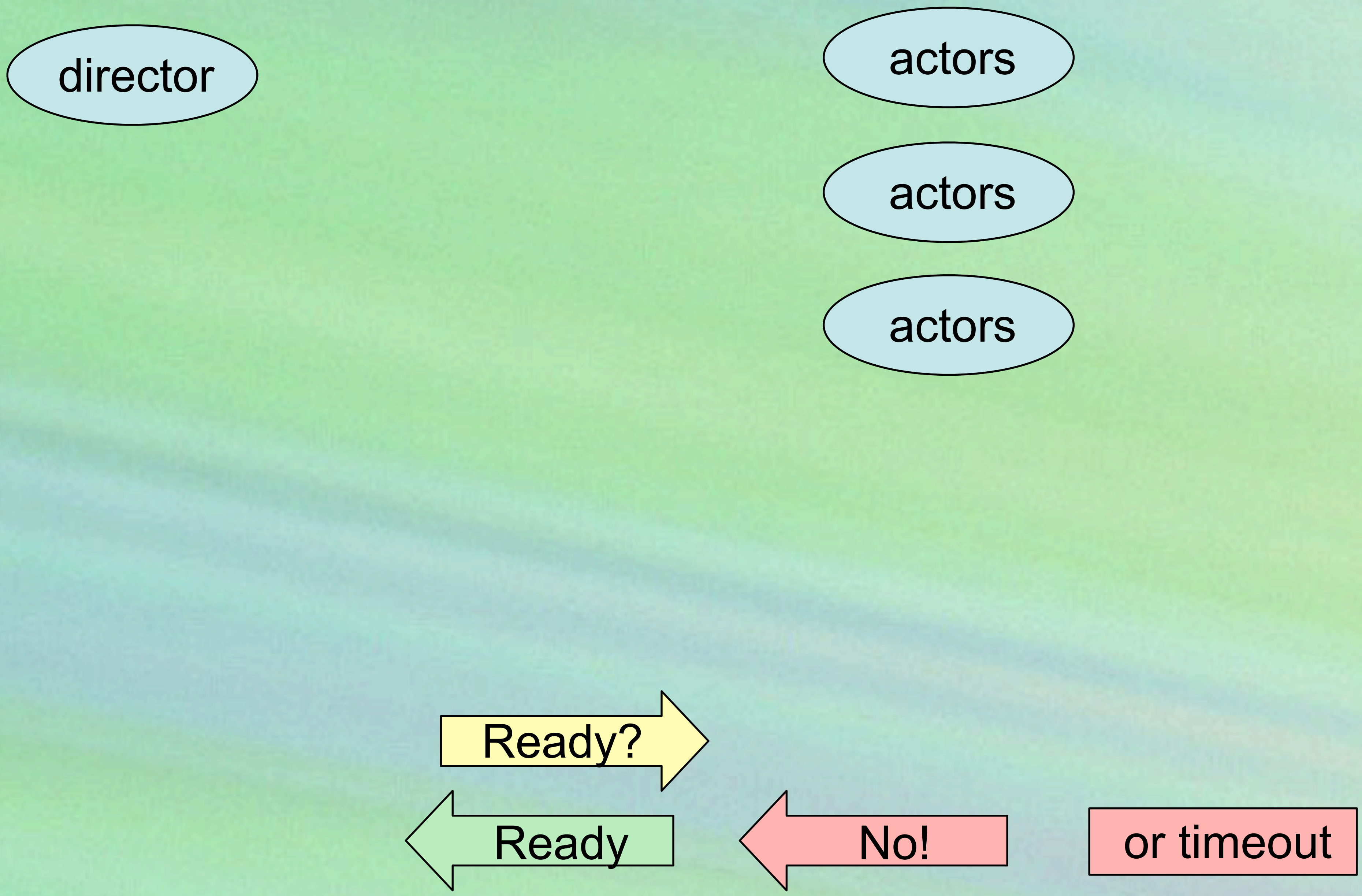
actors

actors

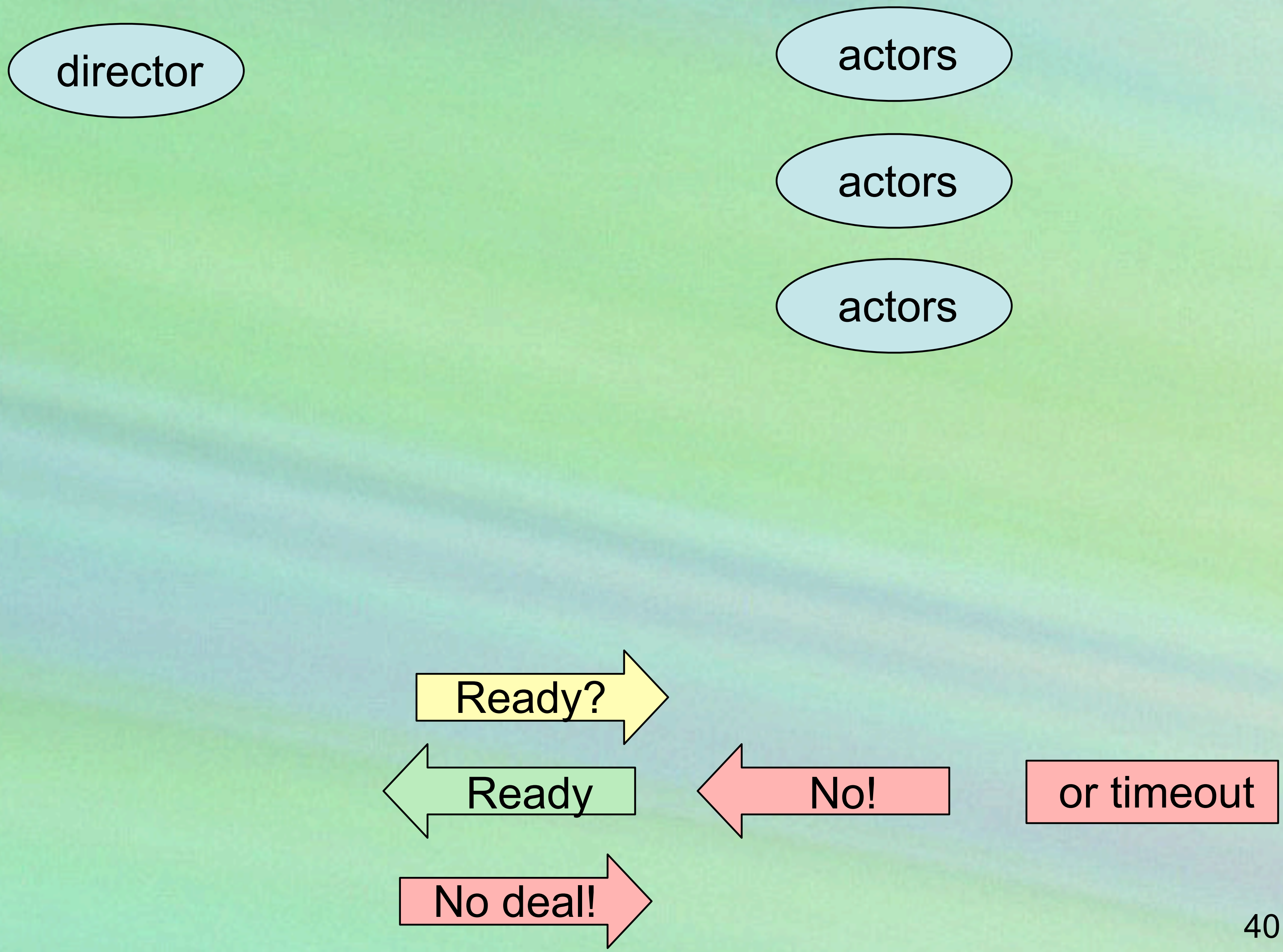
actors

Ready?

All or Nothing:
If any actor says no the deal is off.



All or Nothing:
If any actor says no the deal is off.



The Database Version

TM: Transaction Manager
RM: Resource Manager

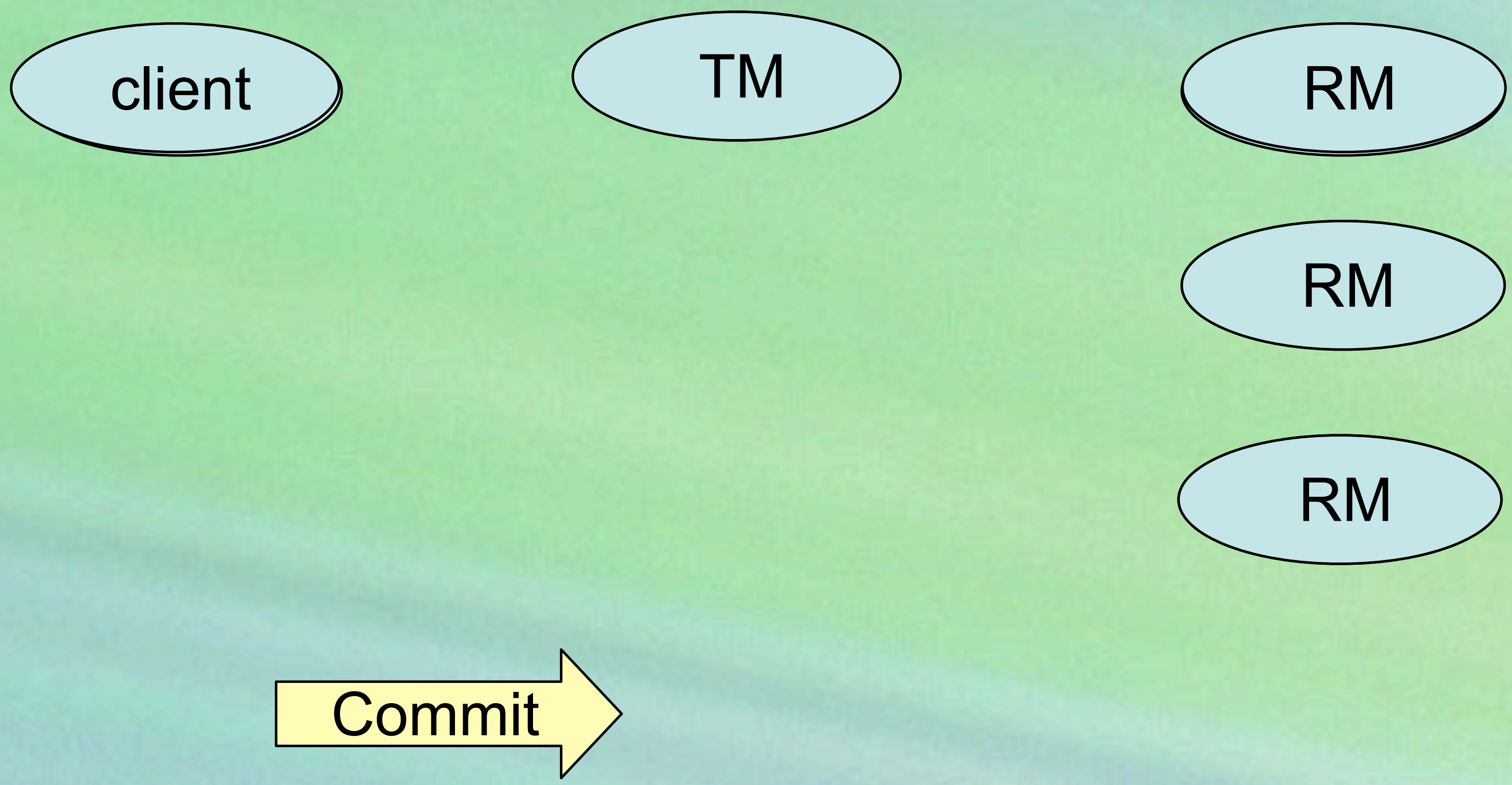
The Database Version

TM: Transaction Manager
RM: Resource Manager

The Database Version

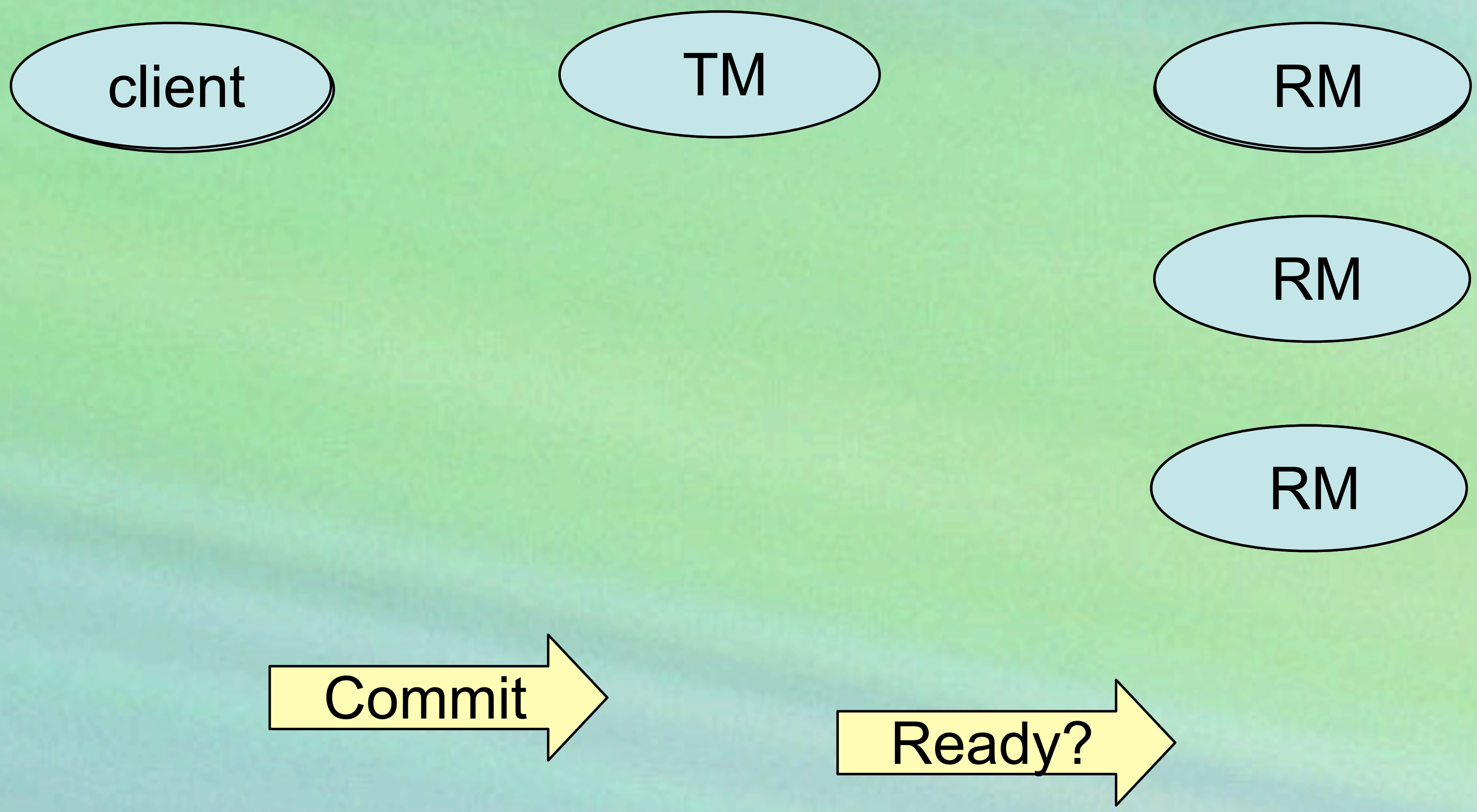
TM: Transaction Manager
RM: Resource Manager

The Database Version



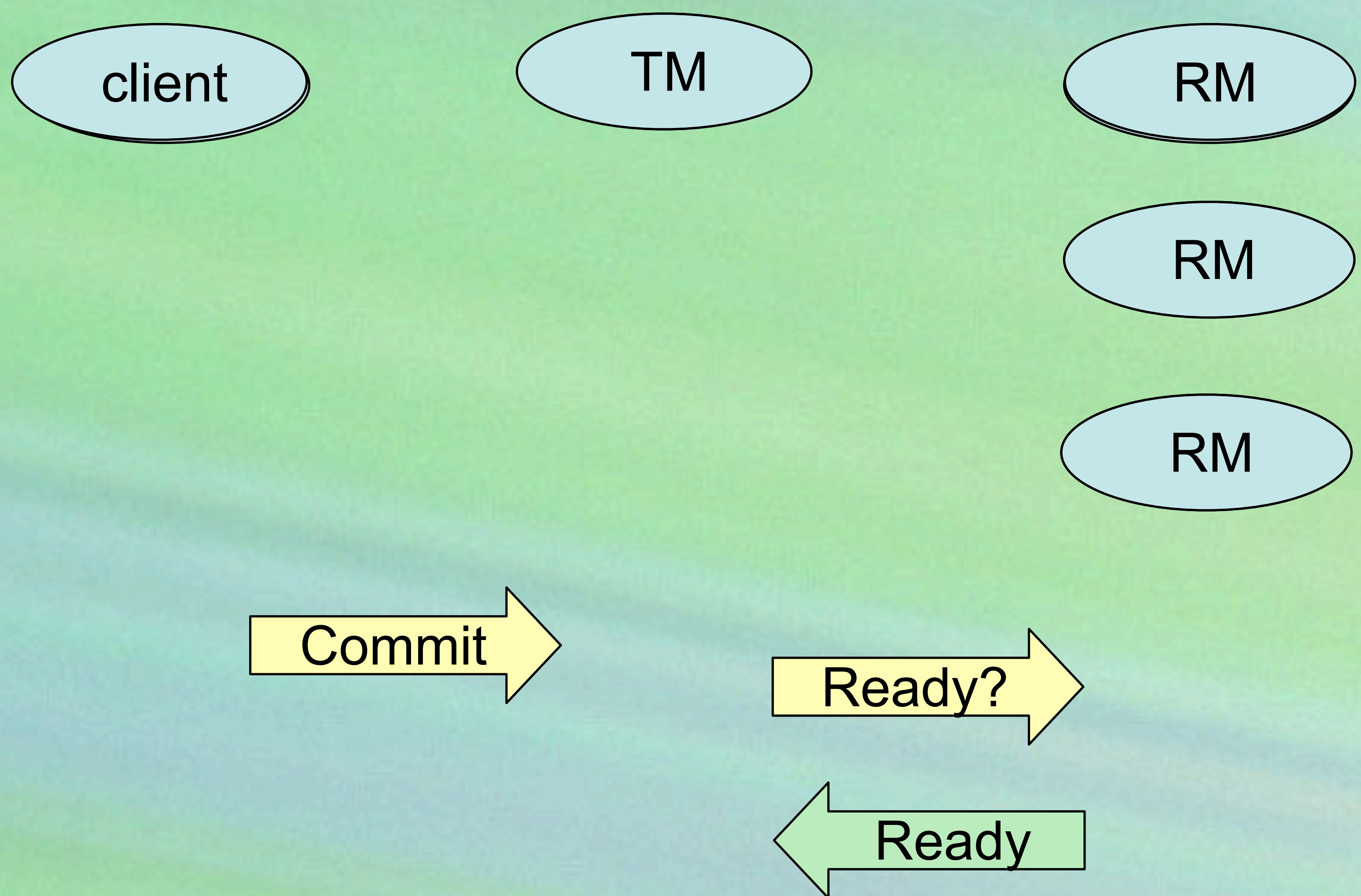
TM: Transaction Manager
RM: Resource Manager

The Database Version



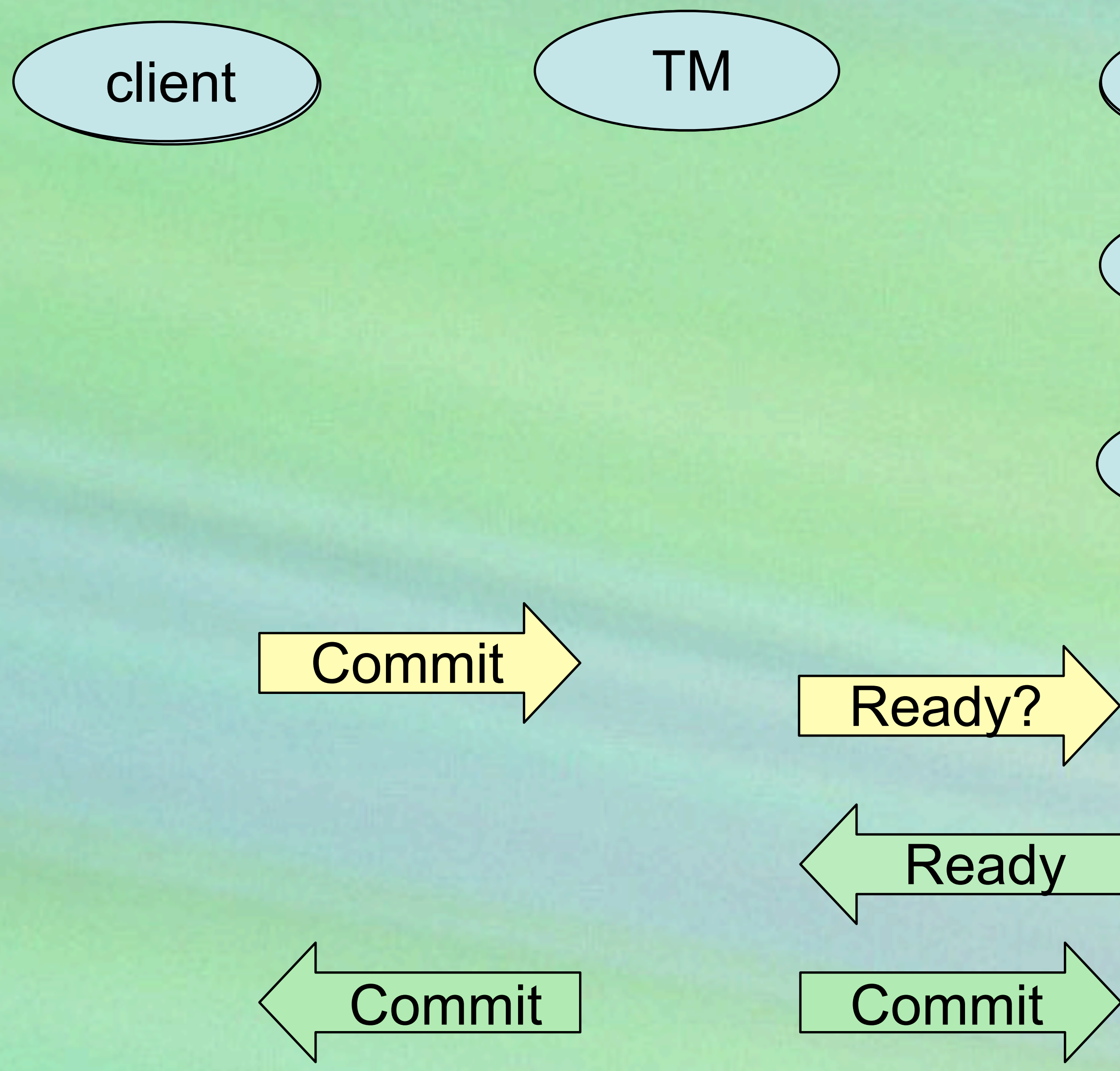
TM: Transaction Manager
RM: Resource Manager

The Database Version



TM: Transaction Manager
RM: Resource Manager

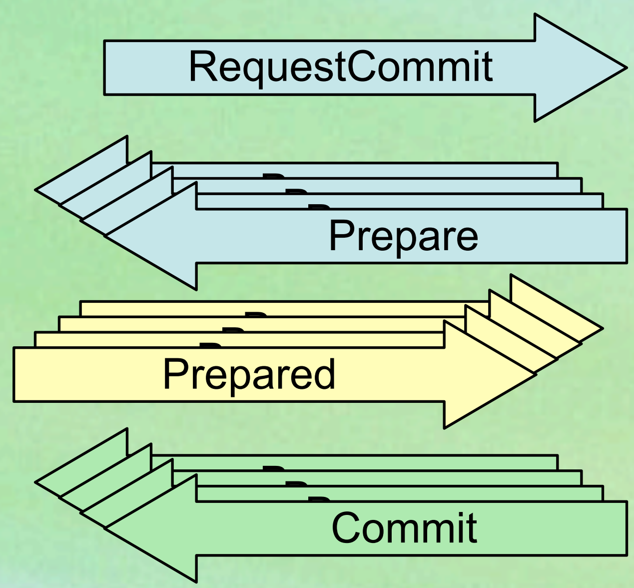
The Database Version

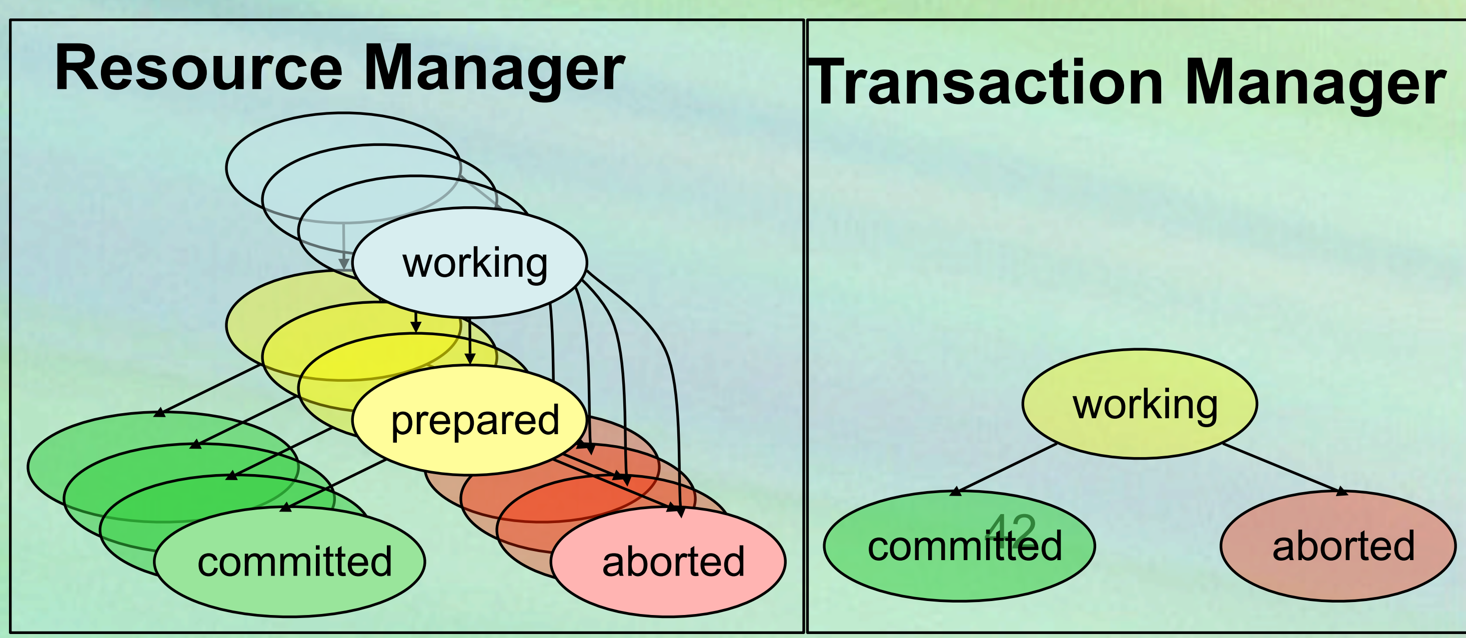


TM: Transaction Manager
RM: Resource Manager

Two Phase Commit

- *N* Resource Managers (RMs)
- Want all RMs to commit or all abort.
- Coordinated by Transaction Manager (TM)
  TM sends Prepare, Commit-Abort
- RM responds Prepared, Aborted
- *3N+1* messages
- *N+1* stable writes
- Delay
  - 4 message
  - 2 stable write
- Blocking:
  if TM fails,
  Commit-Abort stalls





The Problem With 2PC

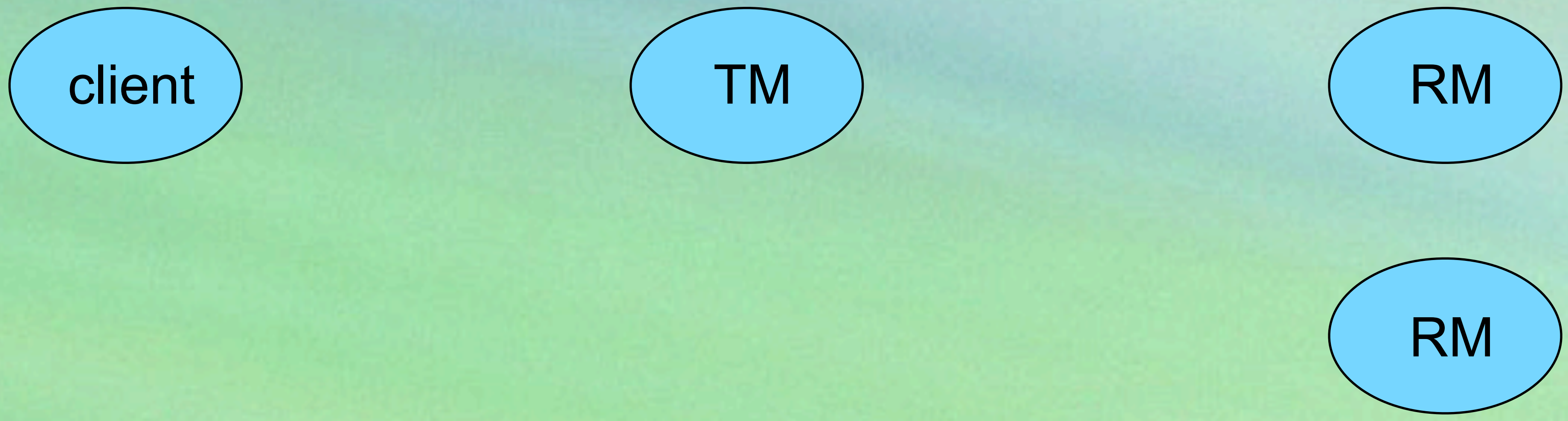
- Atomicity – all or nothing
- Consistency – does right thing
- Isolation – no concurrency anomalies
- Durability / Reliability – state survives failures
- Availability: always up

Blocks if TM fails

Problem Statement

- ACID Transactions make error handling easy.
- One fault can make 2-Phase Commit block.
- Goal: ACID and Available.
  Non-blocking despite *F* faults.

Fault-Tolerant Two Phase Commit

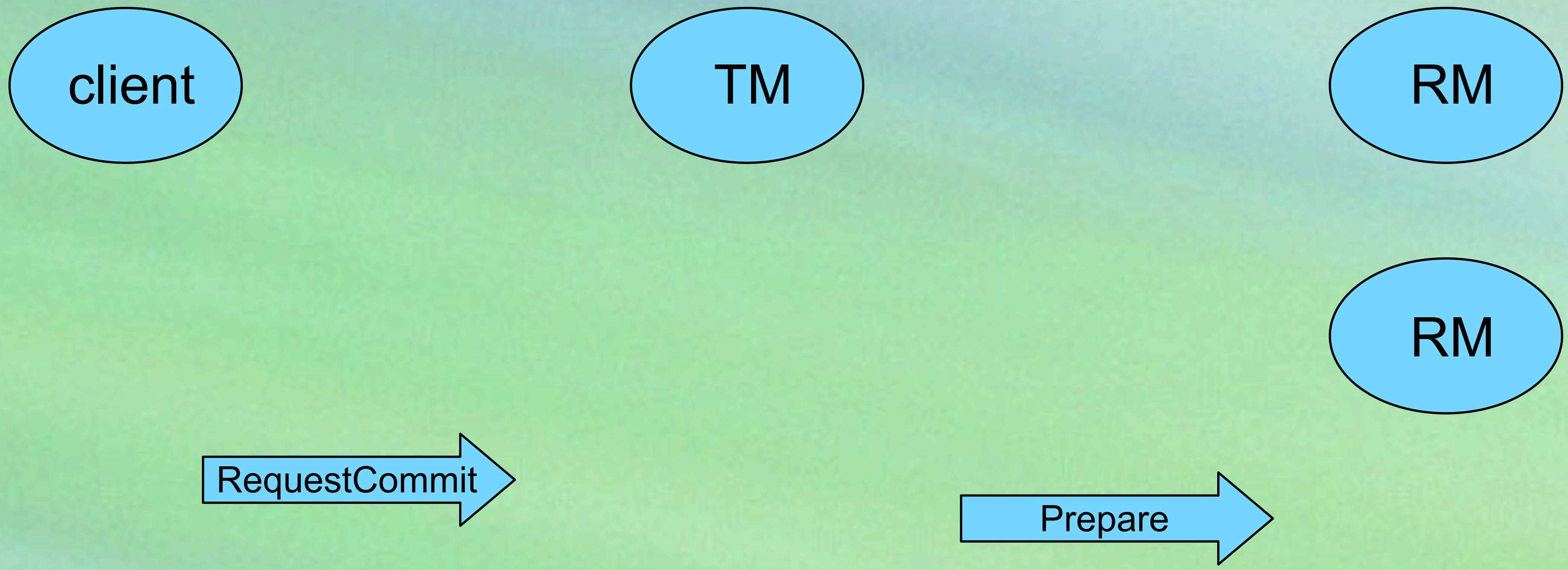


If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit

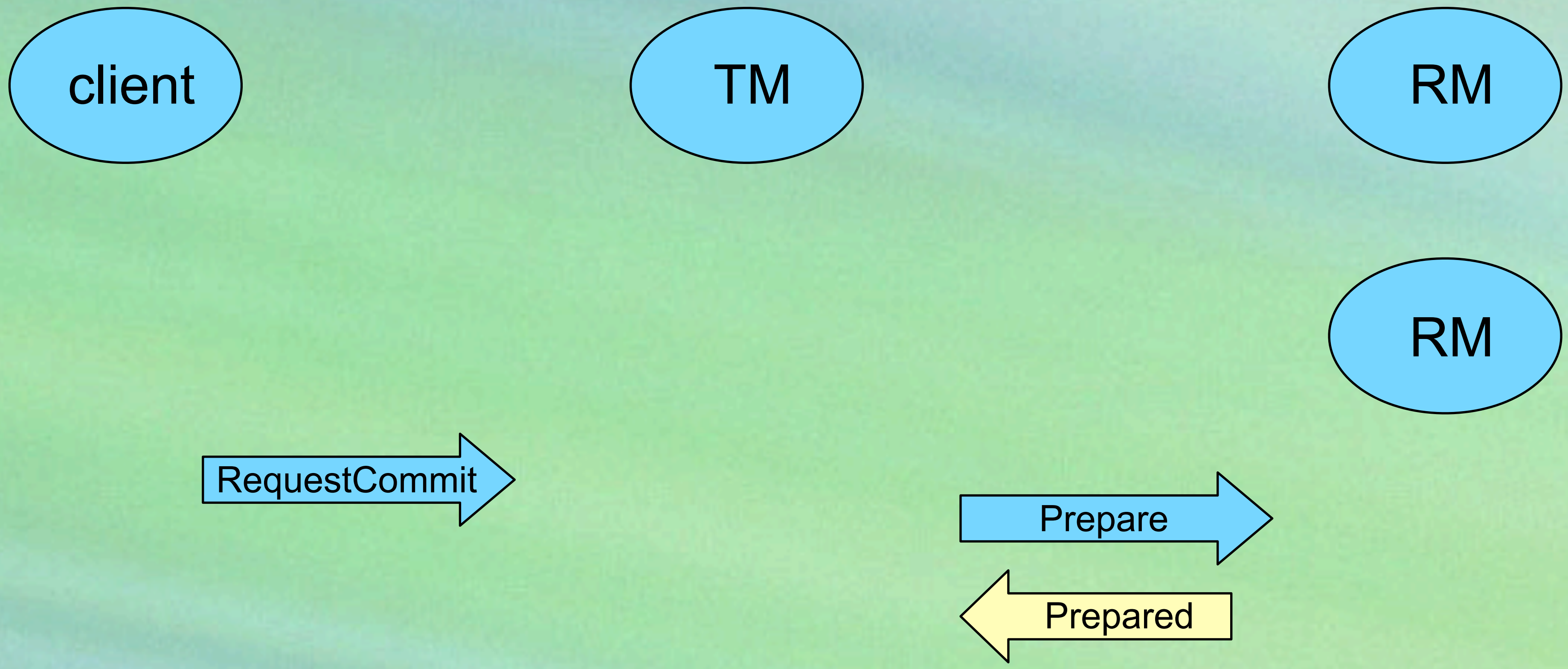
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit



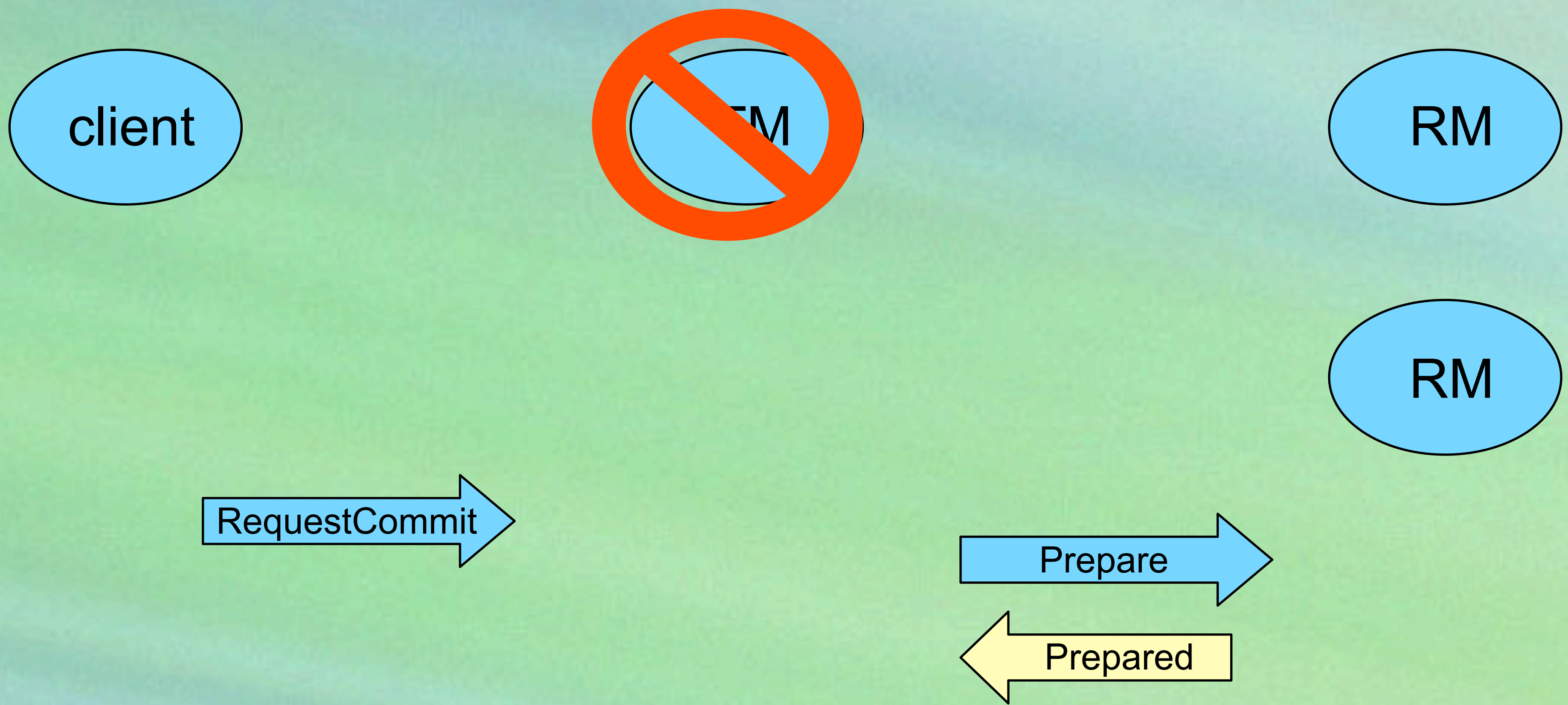
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit



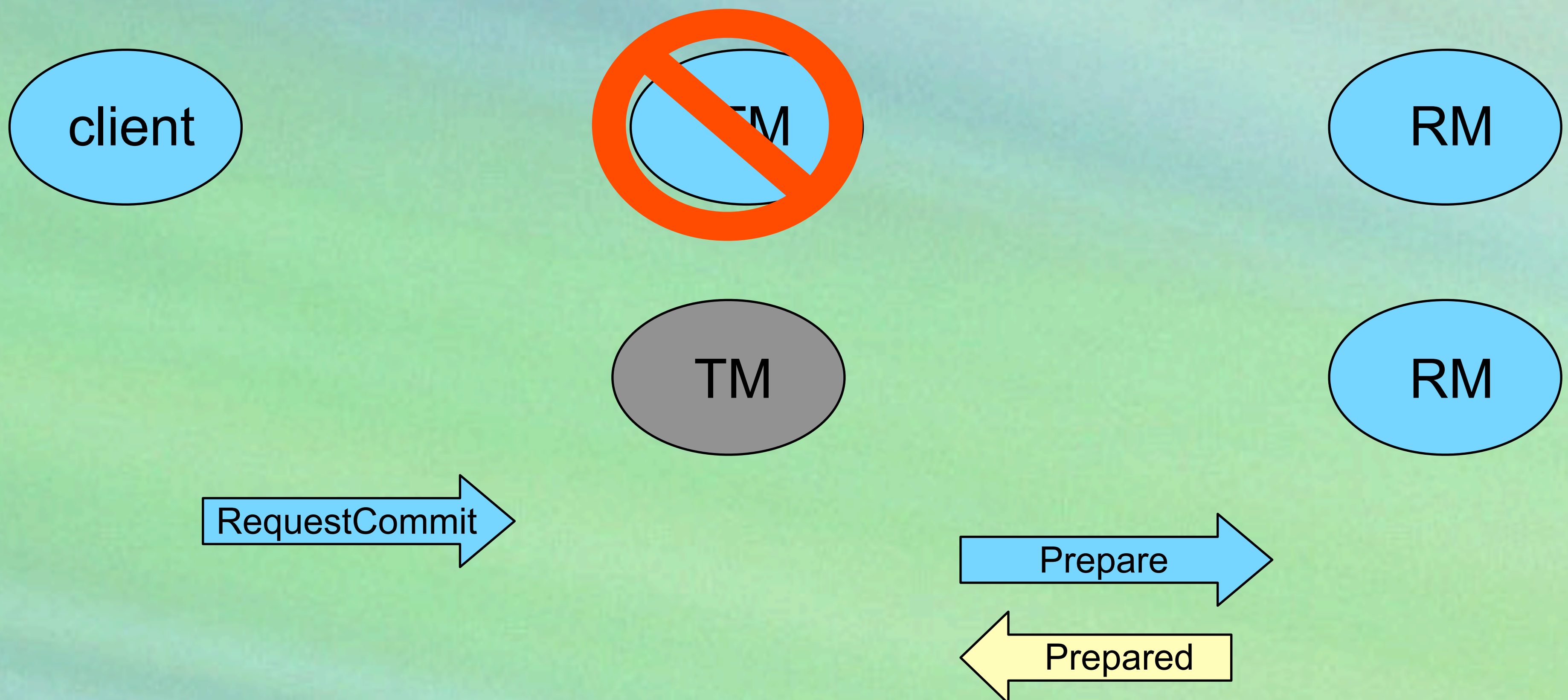
If the 2PC Transaction Manager (TM) Fails, transaction blocks.

Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.

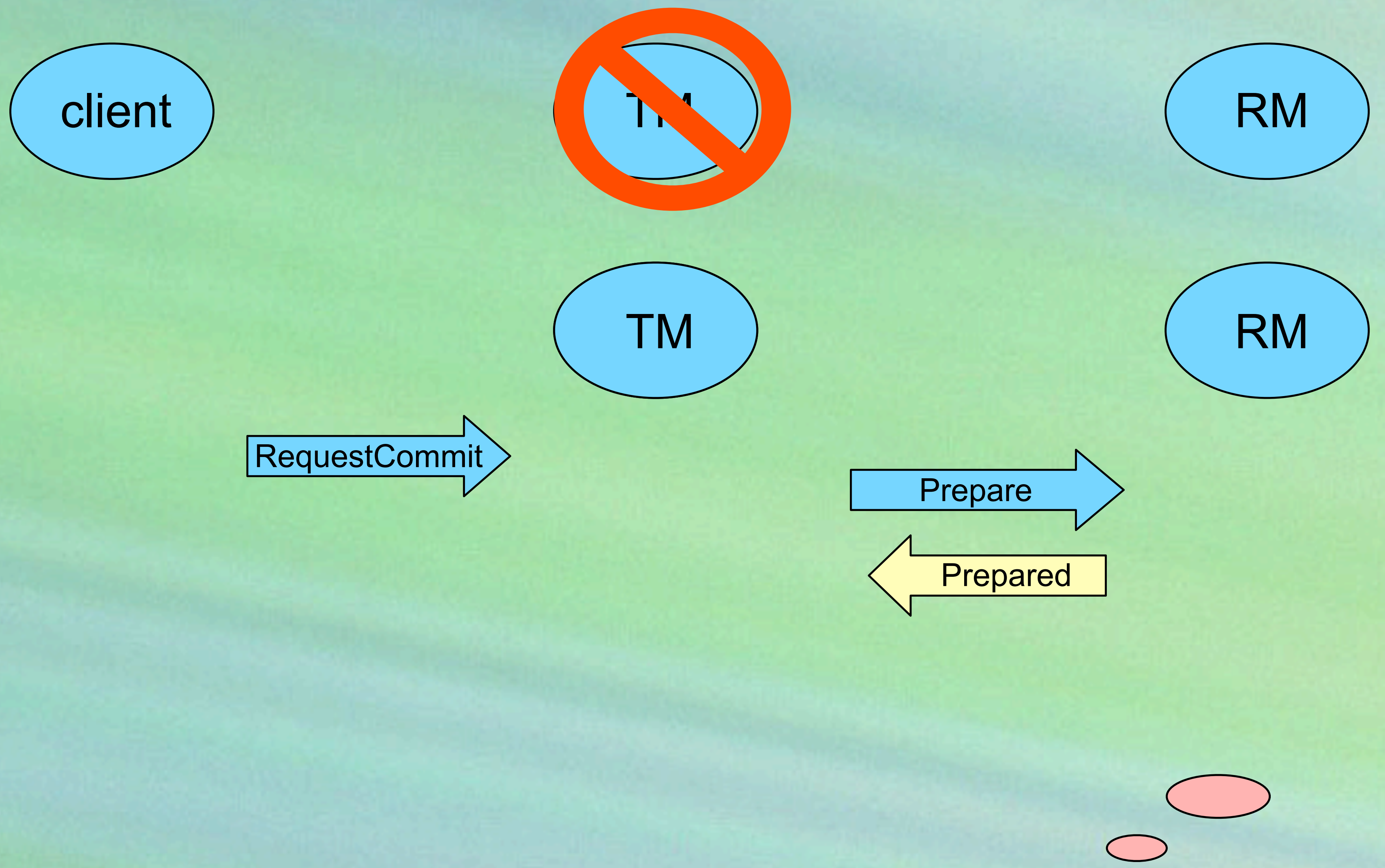
Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.

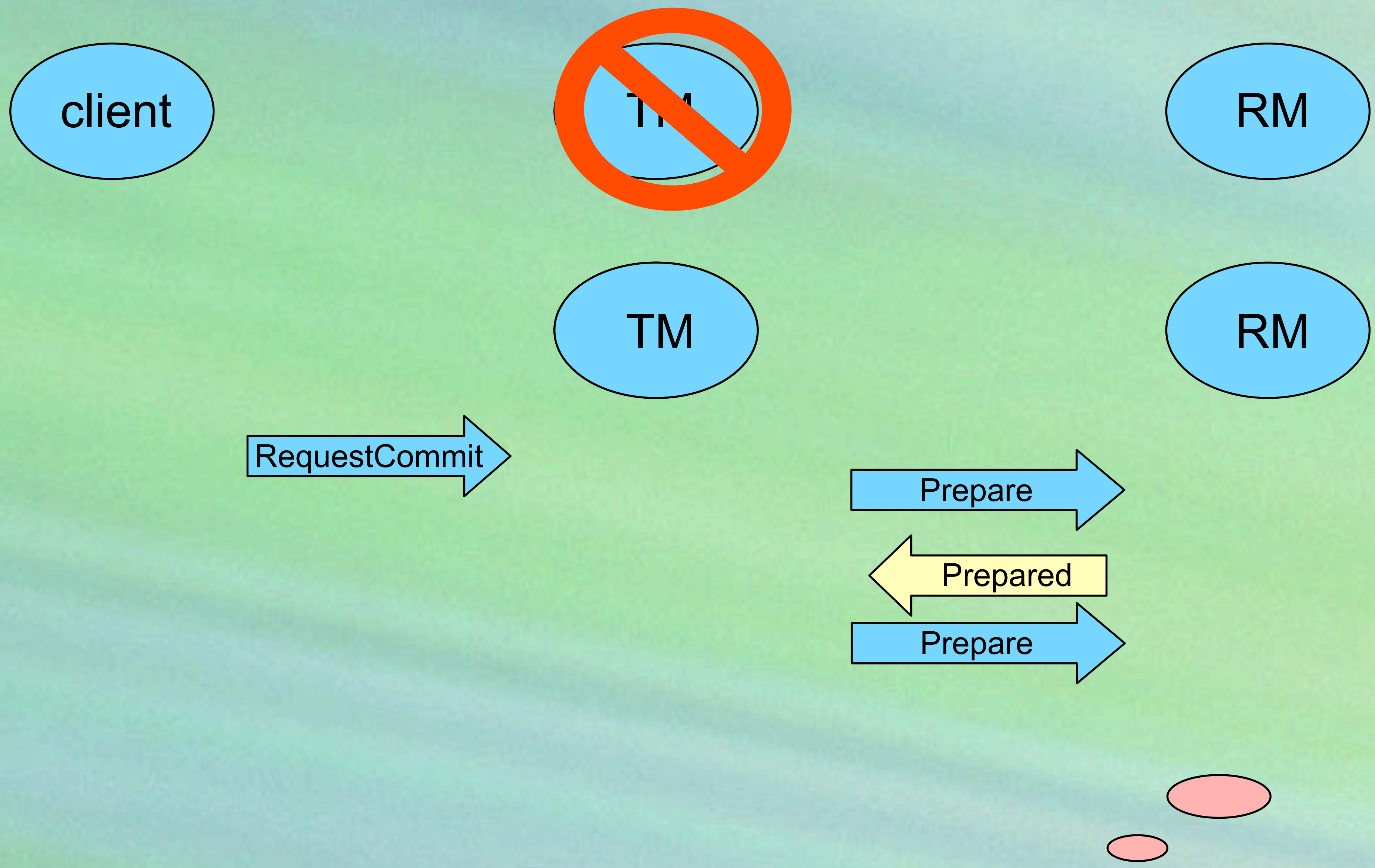
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



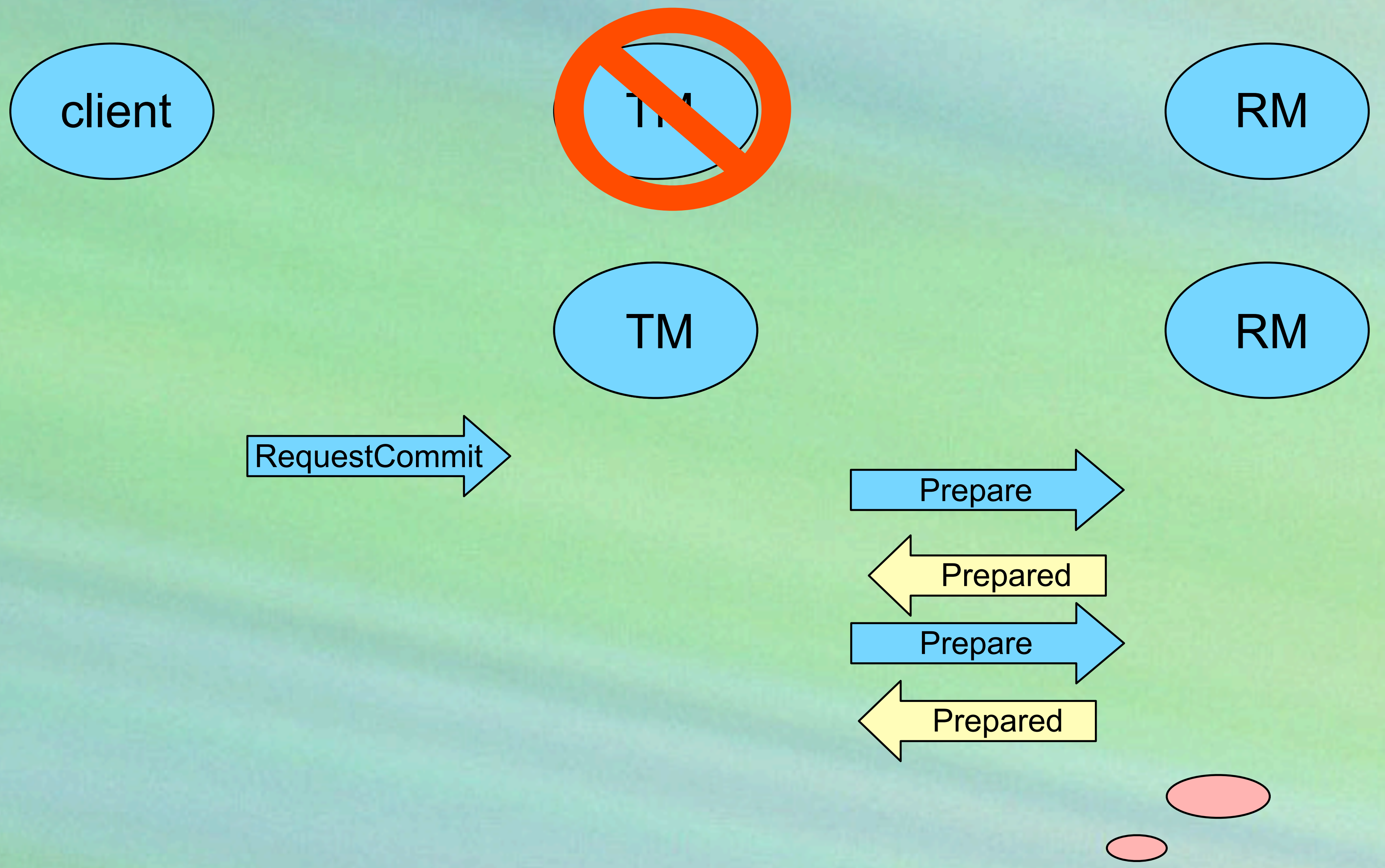
If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



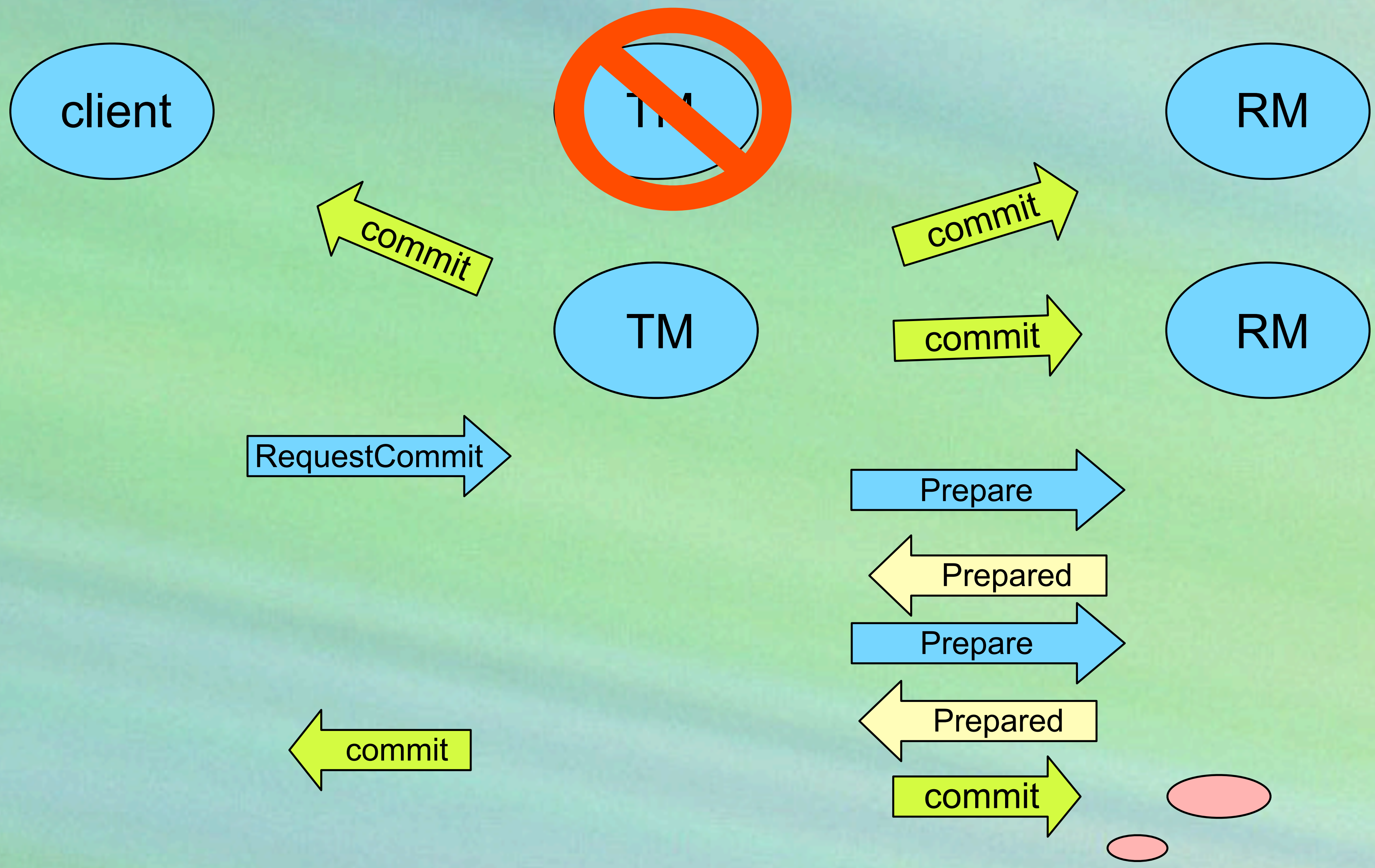
If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)

Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a "spare" transaction manager
(non blocking commit, 3 phase commit)

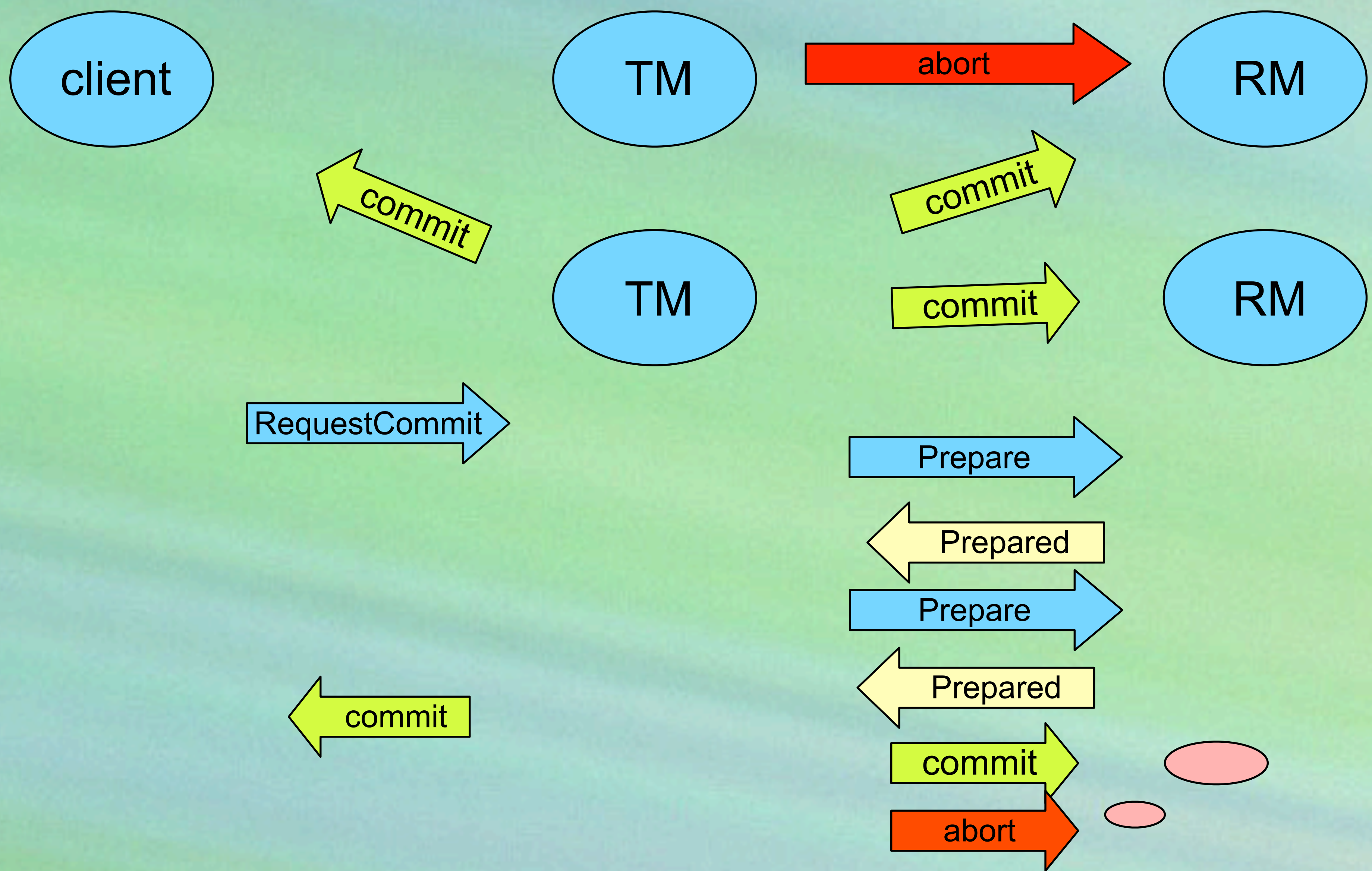
Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a “spare” transaction manager

(non blocking commit, 3 phase commit)

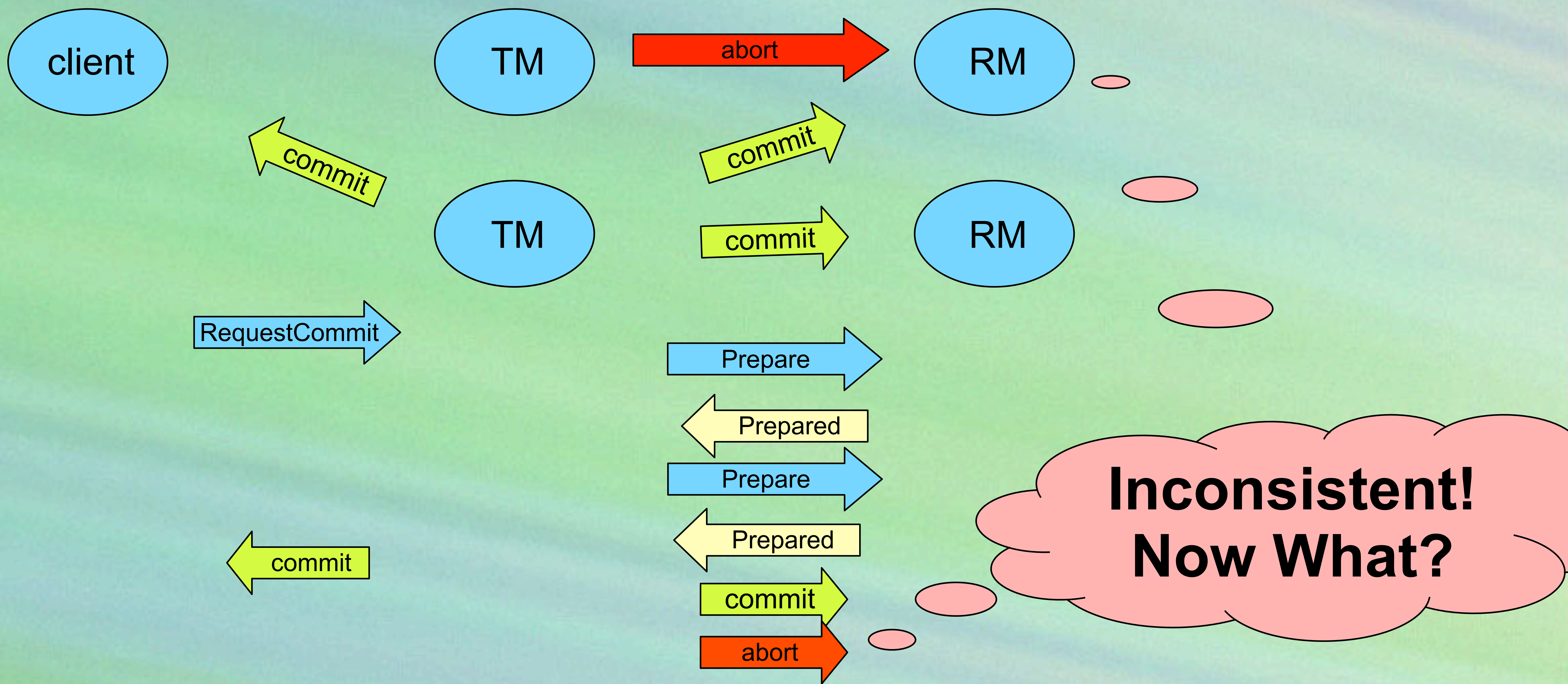
Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)

But… What if….?

Fault-Tolerant Two Phase Commit



If the 2PC Transaction Manager (TM) Fails, transaction blocks.
Solution: Add a “spare” transaction manager
(non blocking commit, 3 phase commit)

But… What if….? The complexity is a mess.

Fault Tolerant 2PC

- Several workarounds proposed in database community:
- Often called "3-phase" or "non-blocking" commit.
- None with complete algorithm and correctness proof.

"Reaching Agreement in the Presence of Faults"

Shostak, Pease, & Lamport
JACM, 1980

“Reaching Agreement in the Presence of Faults”

Shostak, Pease, & Lamport
JACM, 1980

- 25 years of theory

“Reaching Agreement in the Presence of Faults”

Shostak, Pease, & Lamport
JACM, 1980

- 25 years of theory

“Reaching Agreement in the Presence of Faults”

Shostak, Pease, & Lamport
JACM, 1980

- 25 years of theory
- Now called the **Consensus** problem

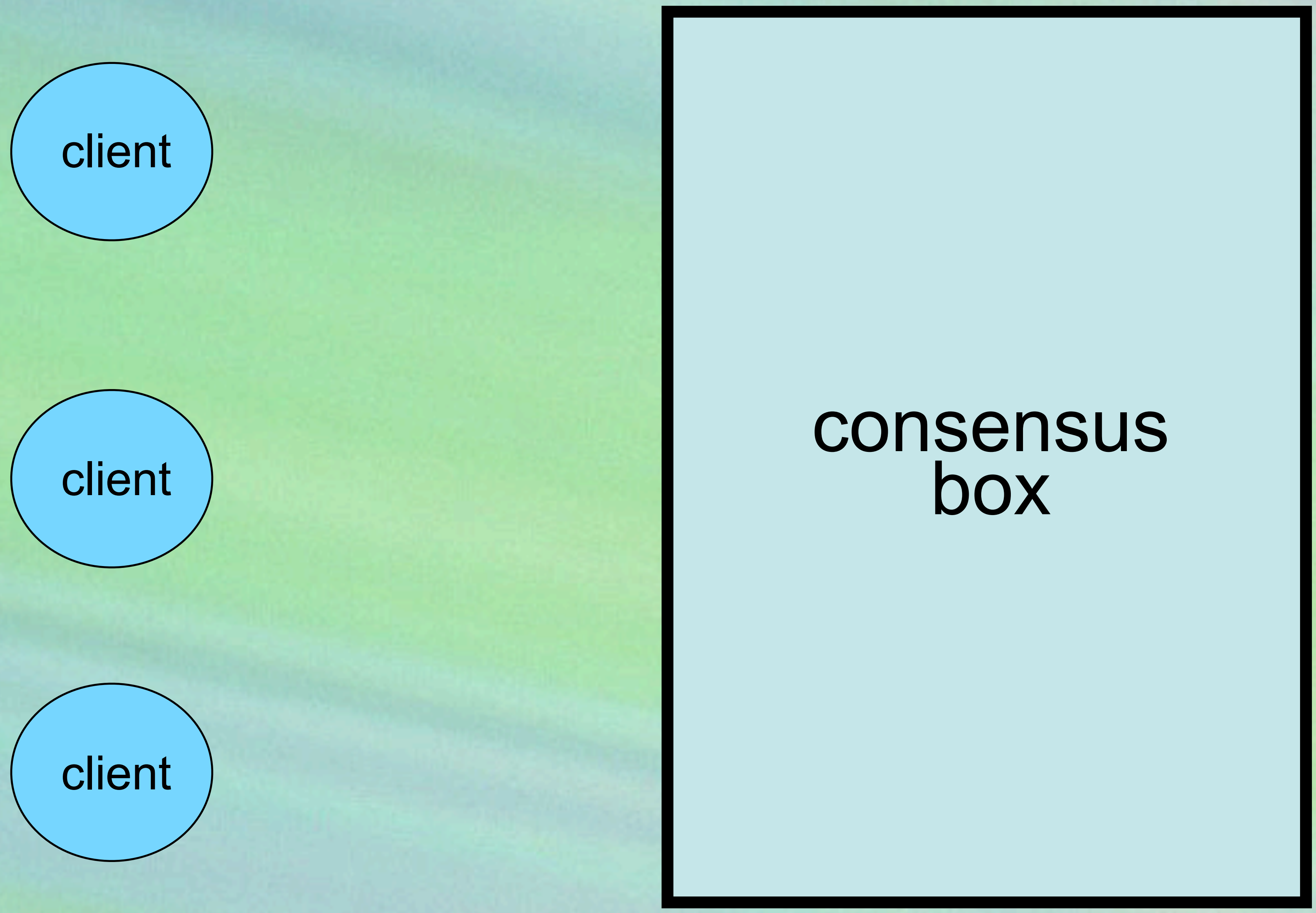
“Reaching Agreement in the Presence of Faults”

Shostak, Pease, & Lamport
JACM, 1980

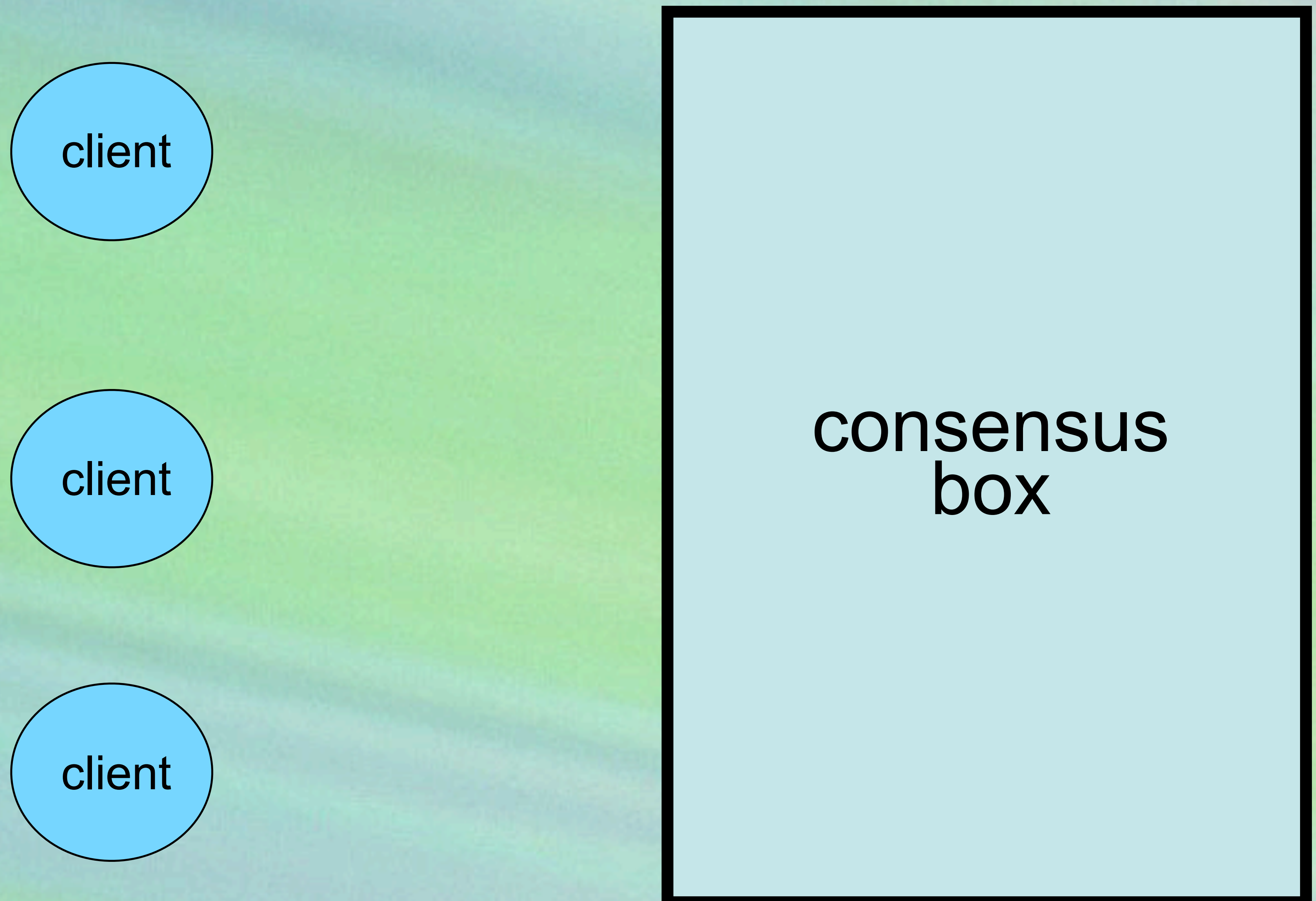
- 25 years of theory
- Now called the **Consensus** problem
- *N* processes want to agree on a value, even if *F* of them have failed.

Consensus

Consensus

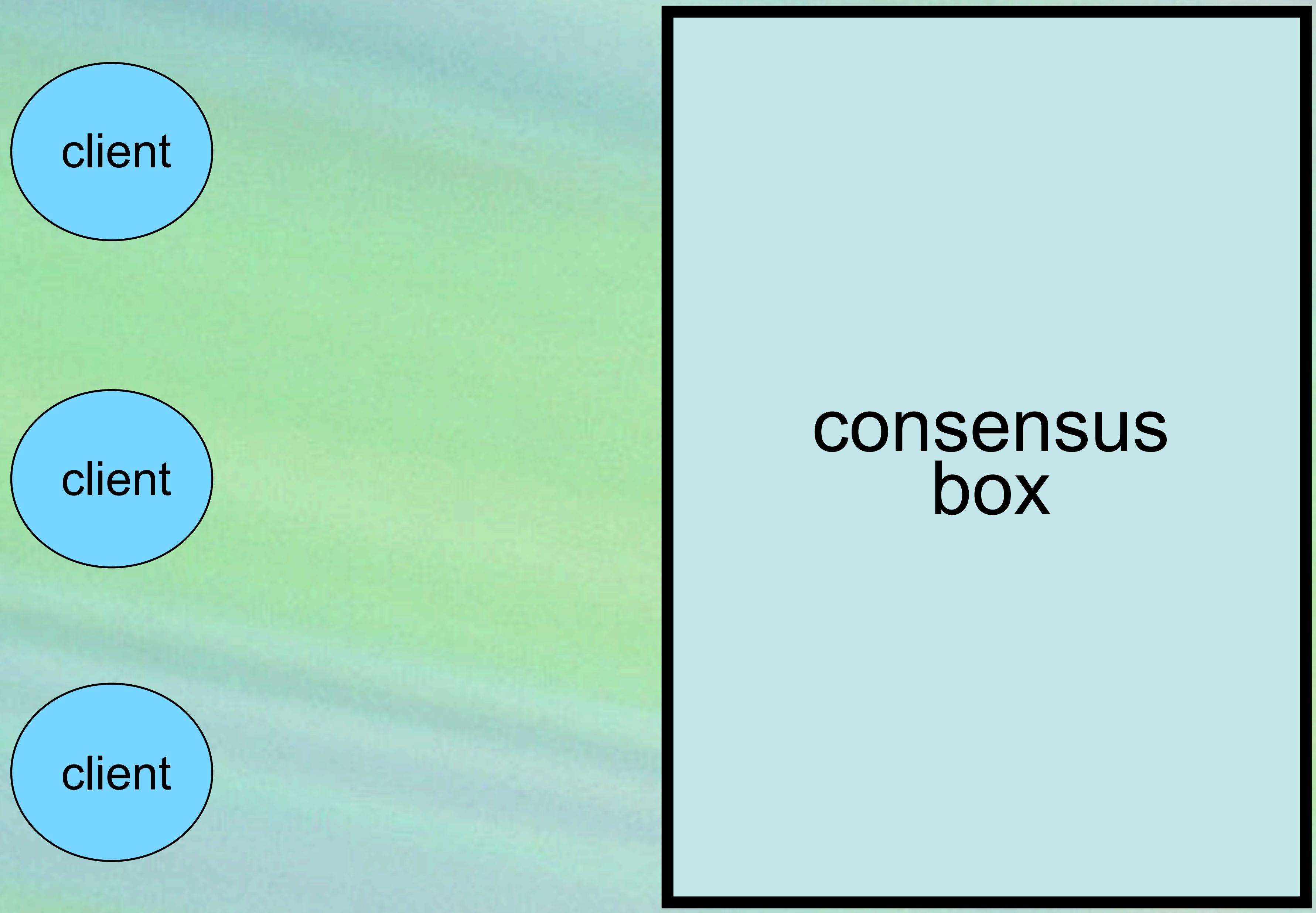


Consensus



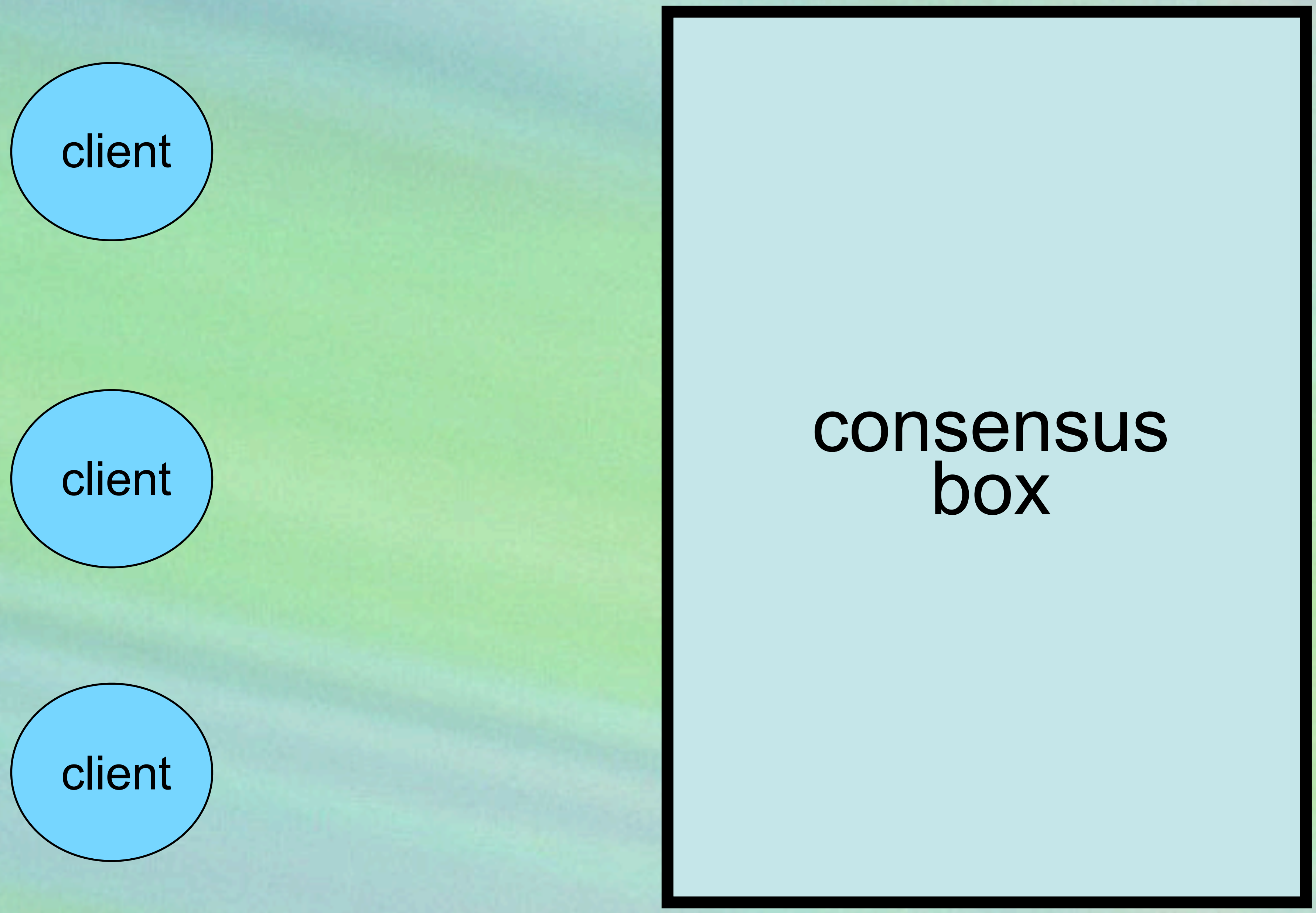
- collects proposed values

Consensus



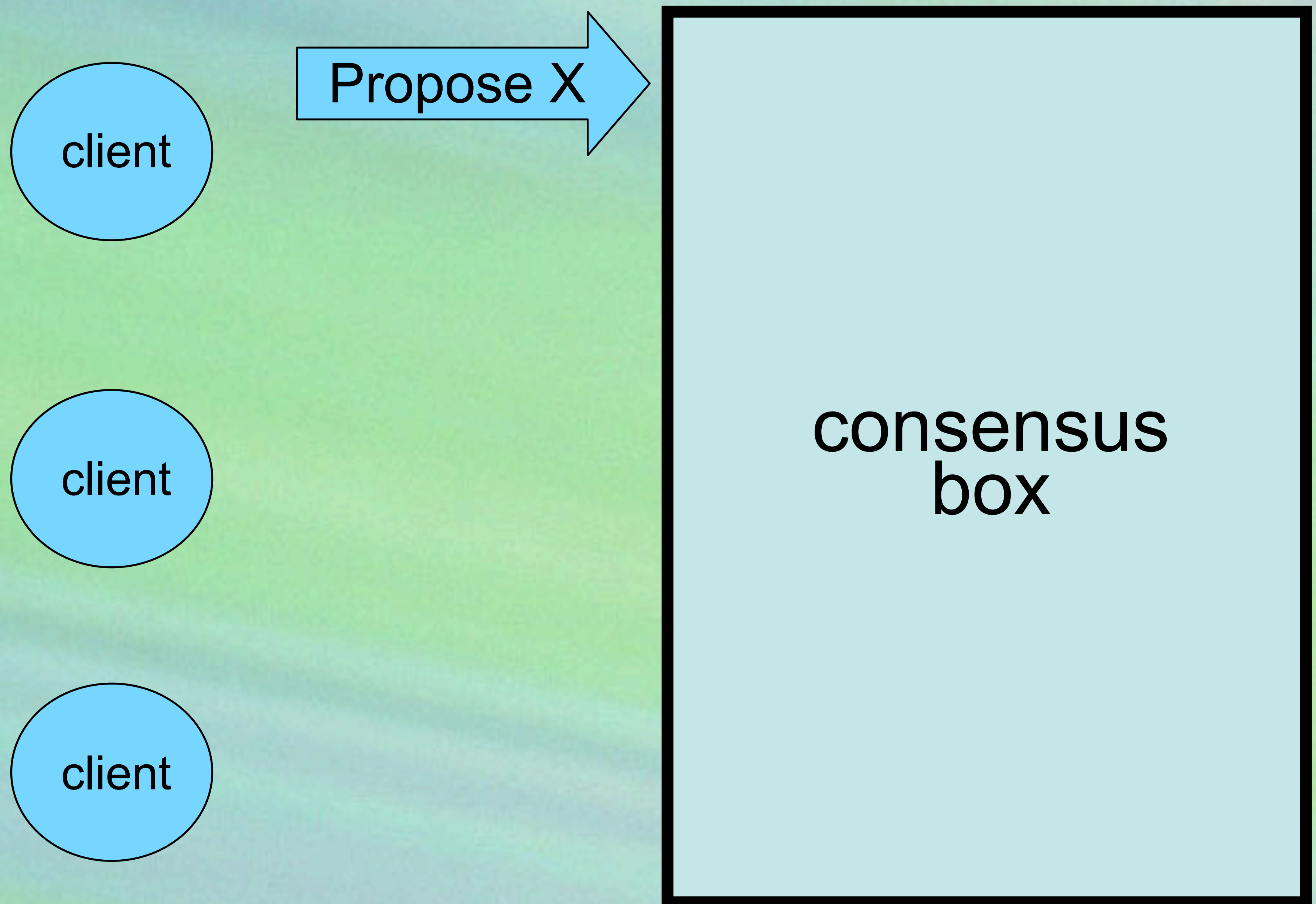
- collects proposed values
- Picks one proposed value

Consensus



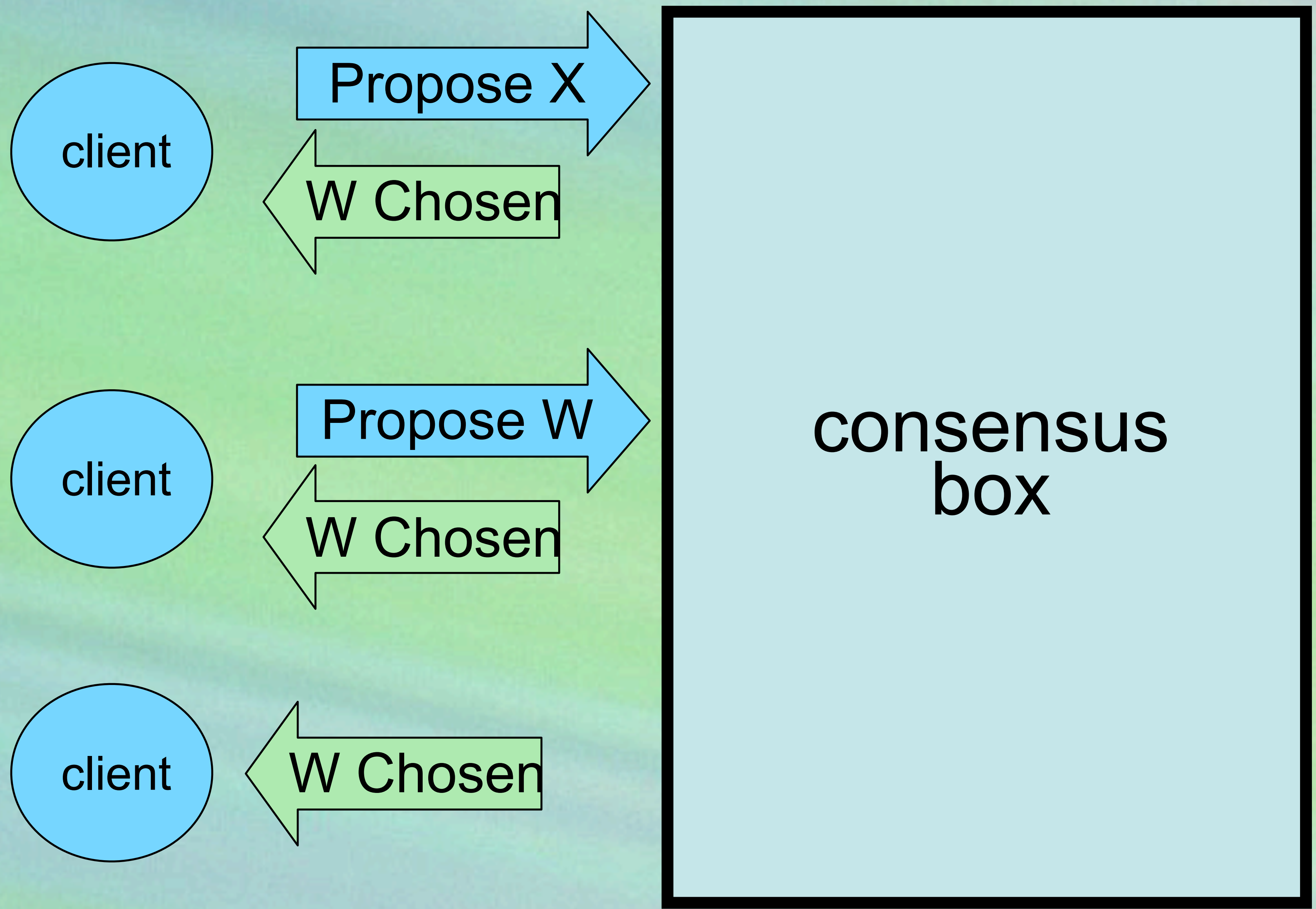
- collects proposed values
- Picks one proposed value
- remembers it forever

Consensus



- collects proposed values
- Picks one proposed value
- remembers it forever

Consensus



- collects proposed values
- Picks one proposed value
- remembers it forever

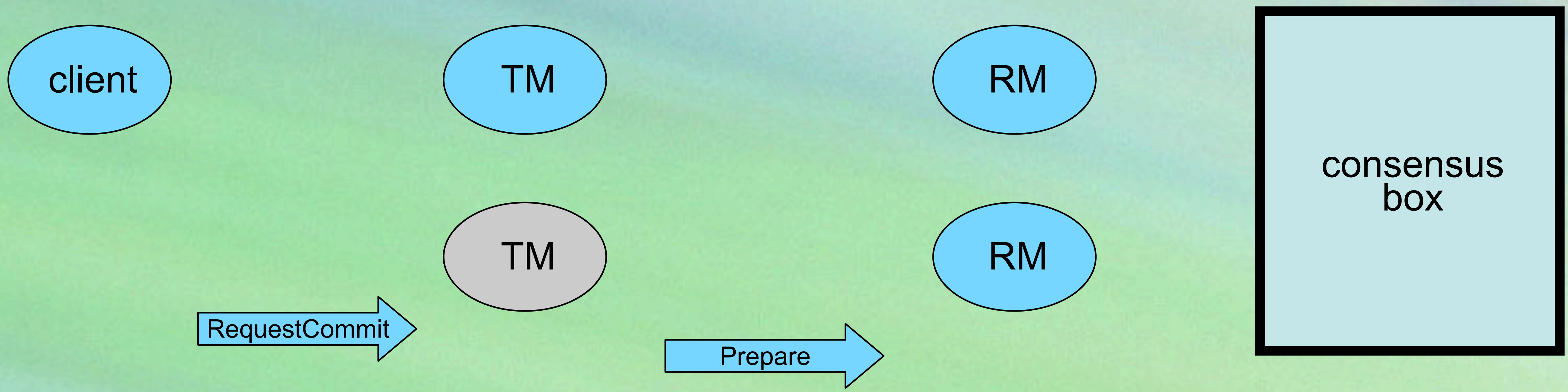
Consensus for Commit
The Obvious Approach

- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

Consensus for Commit
The Obvious Approach

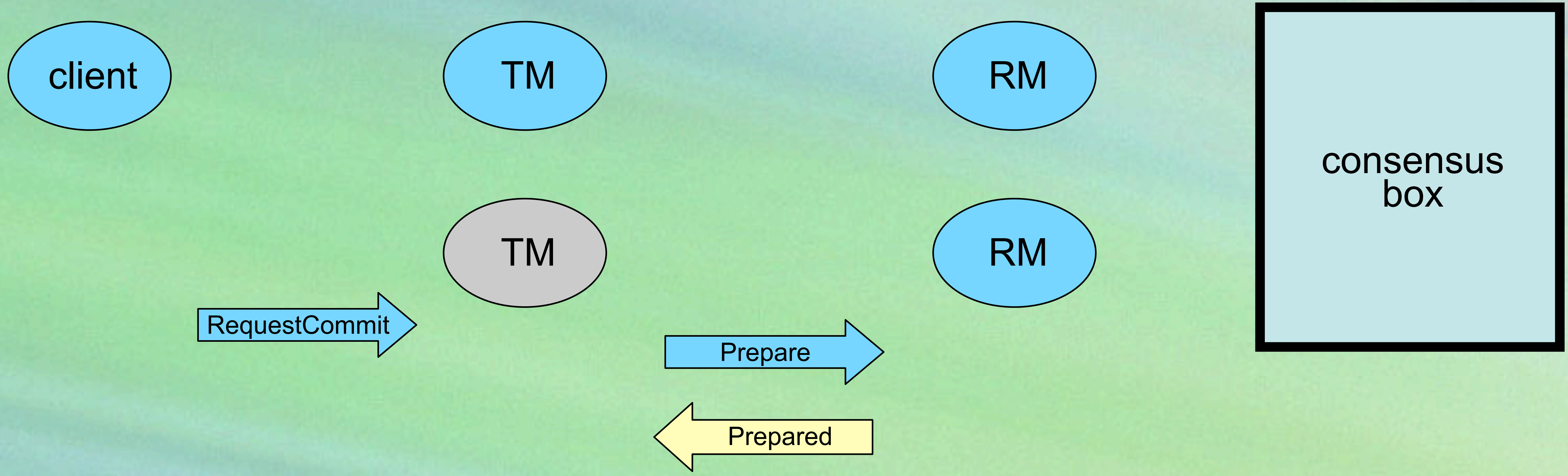
- Get consensus on TM's decision.
- TM just learns consensus value.
- TM is "stateless"

Consensus for Commit
The Obvious Approach



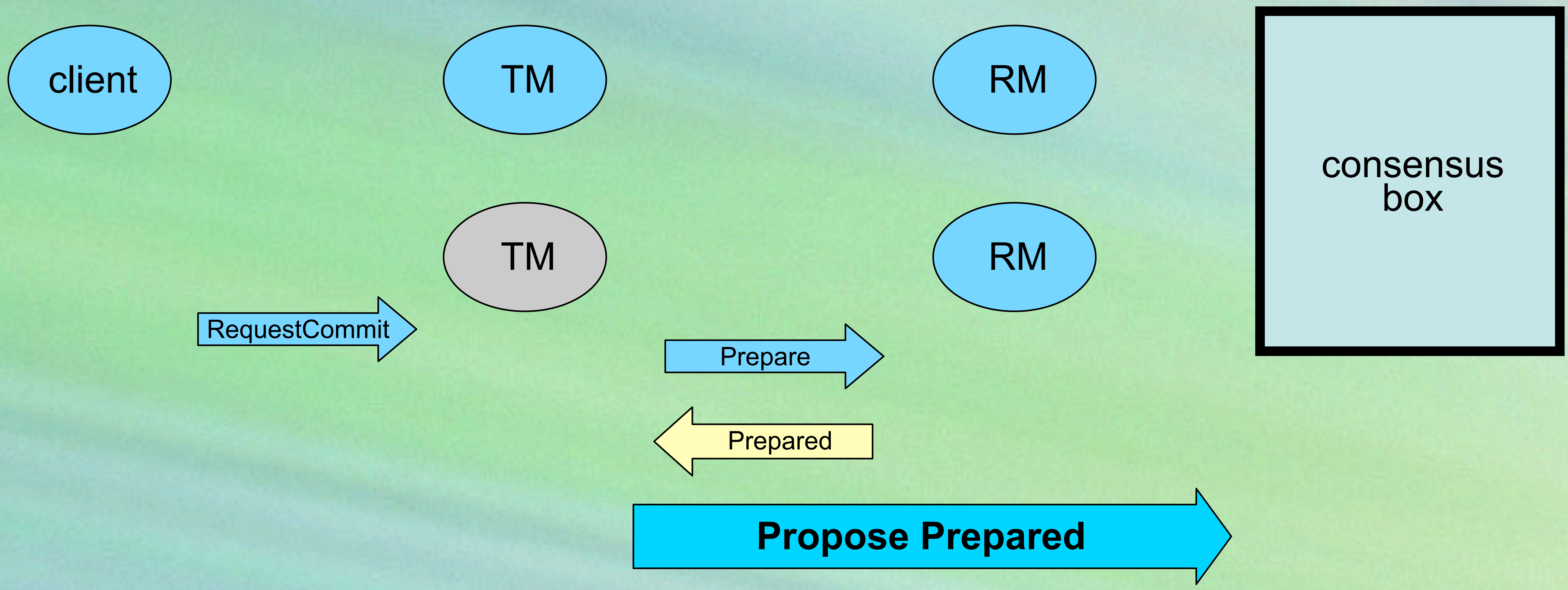
- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

Consensus for Commit
The Obvious Approach



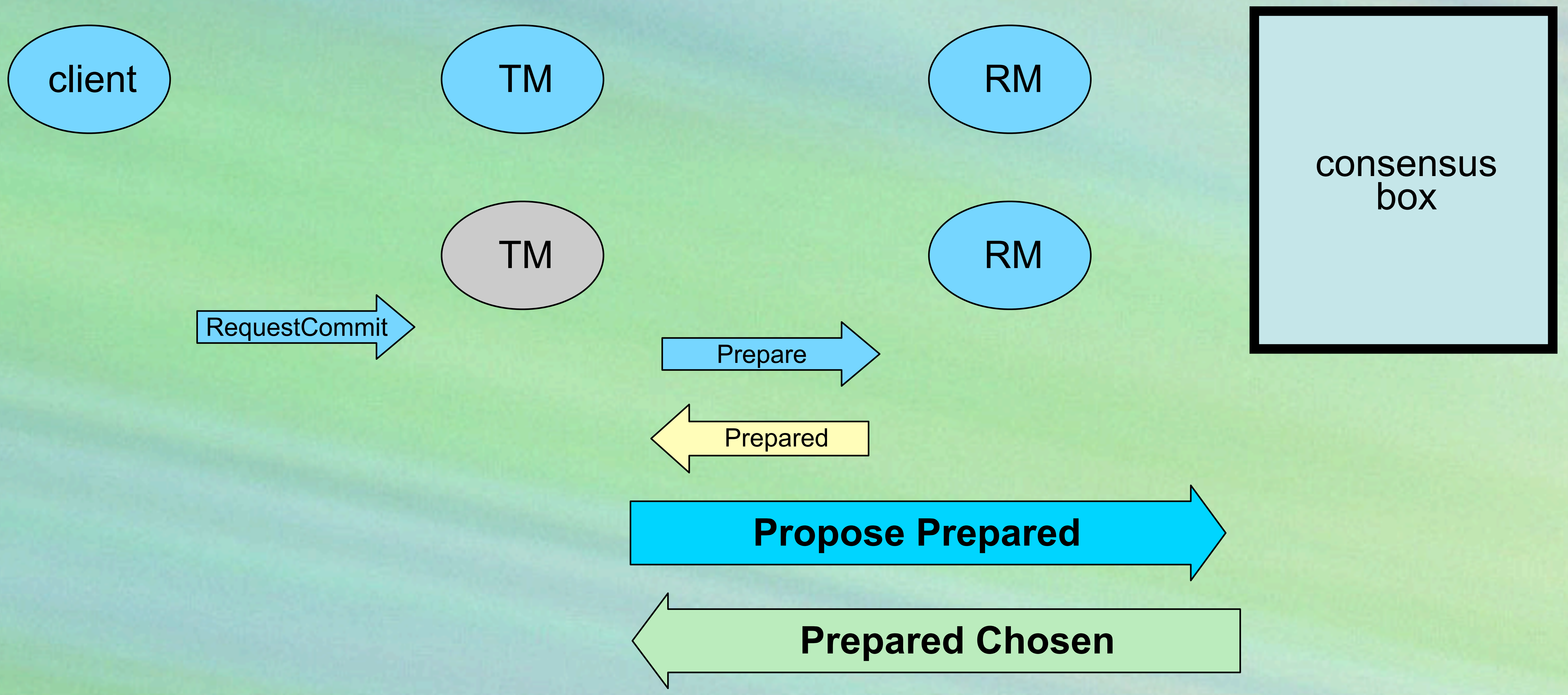
- Get consensus on TM's decision.
- TM just learns consensus value.
- TM is "stateless"

Consensus for Commit
The Obvious Approach



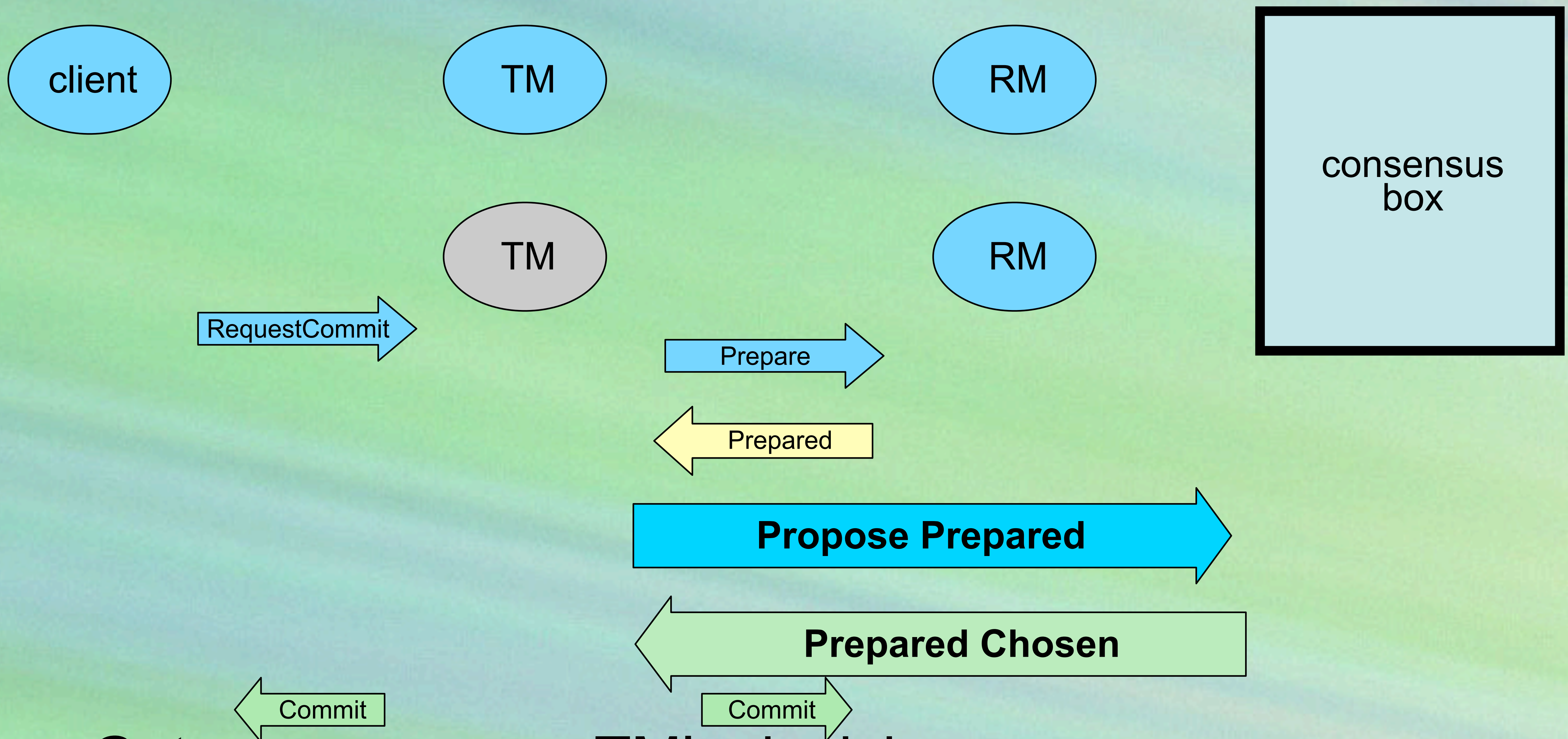
- Get consensus on TM's decision.
- TM just learns consensus value.
- TM is "stateless"

Consensus for Commit
The Obvious Approach



- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

Consensus for Commit
The Obvious Approach



- Get consensus on TM’s decision.
- TM just learns consensus value.
- TM is “stateless”

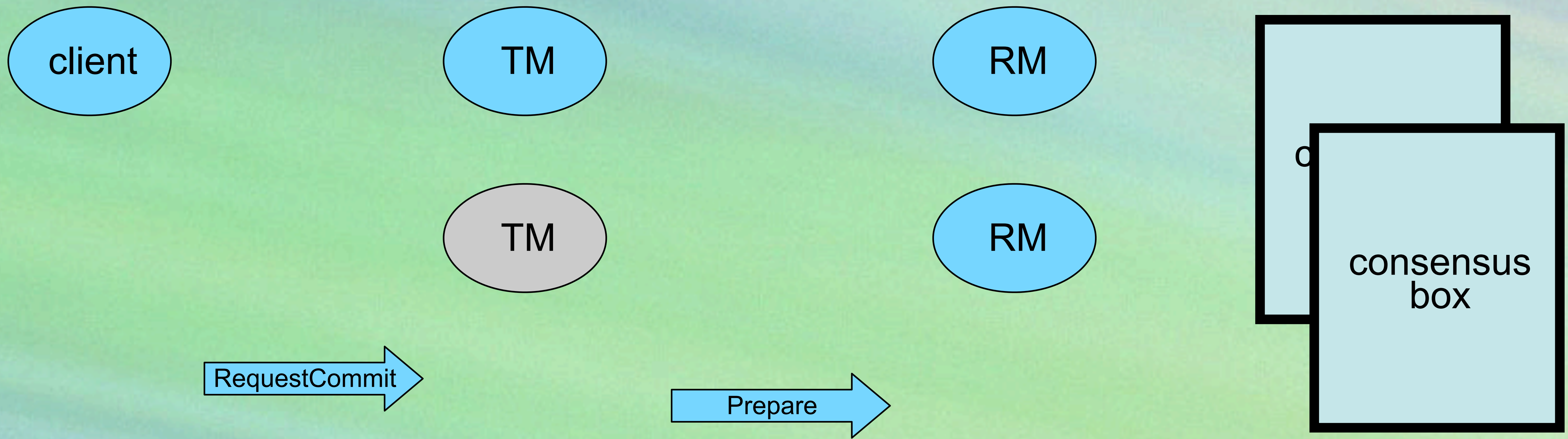
Consensus for Commit
The Paxos Commit Approach

- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

Consensus for Commit
The Paxos Commit Approach

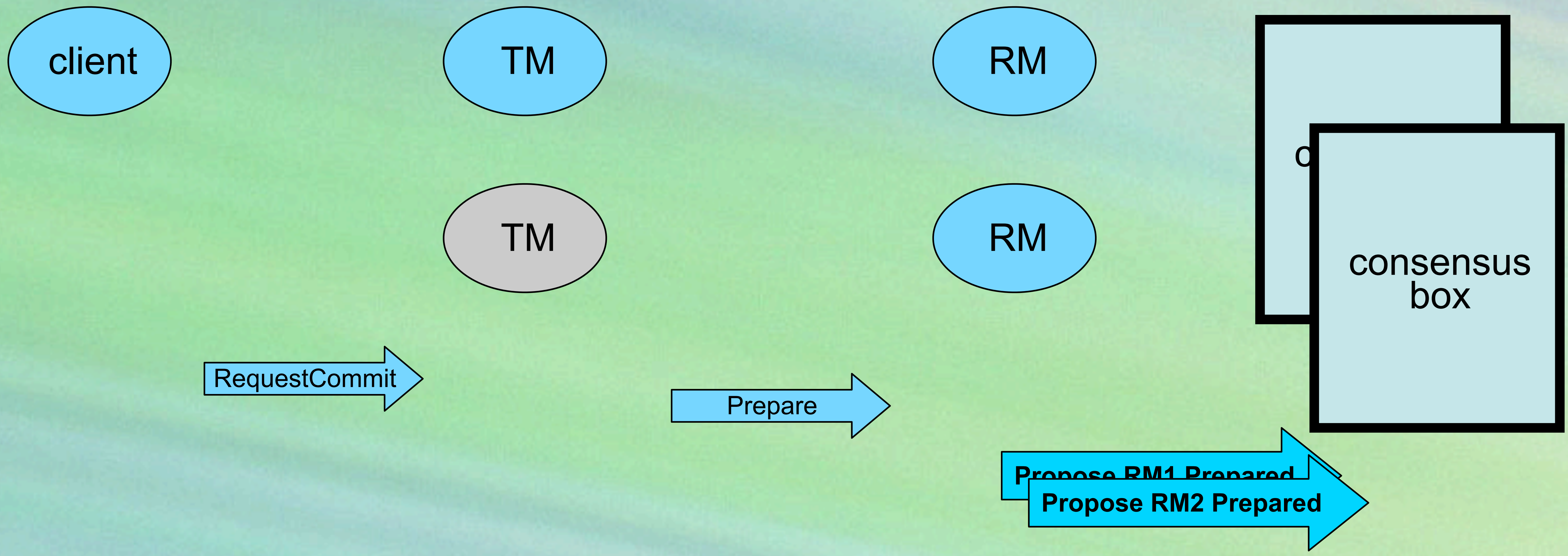
- Get consensus on each RM's choice.
- TM just combines consensus values.
- TM is "stateless"

Consensus for Commit
The Paxos Commit Approach



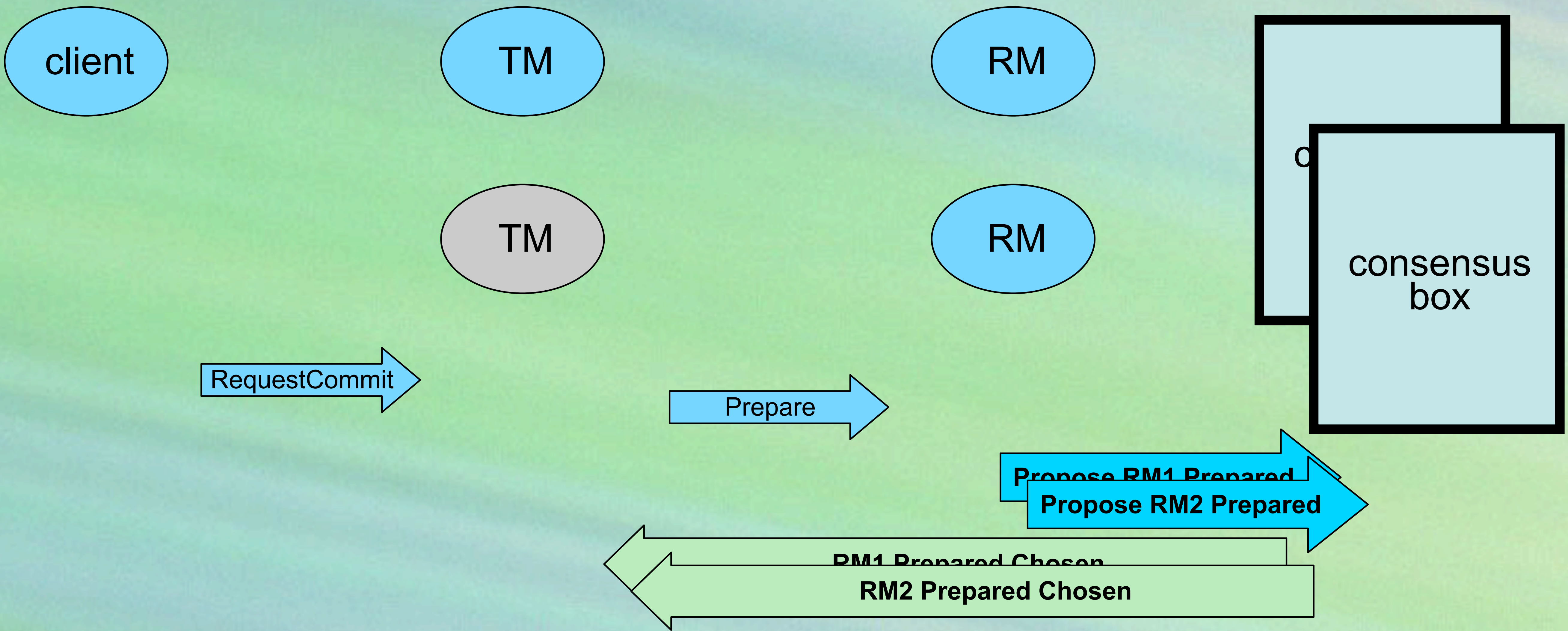
- Get consensus on each RM's choice.
- TM just combines consensus values.
- TM is "stateless"

Consensus for Commit
The Paxos Commit Approach



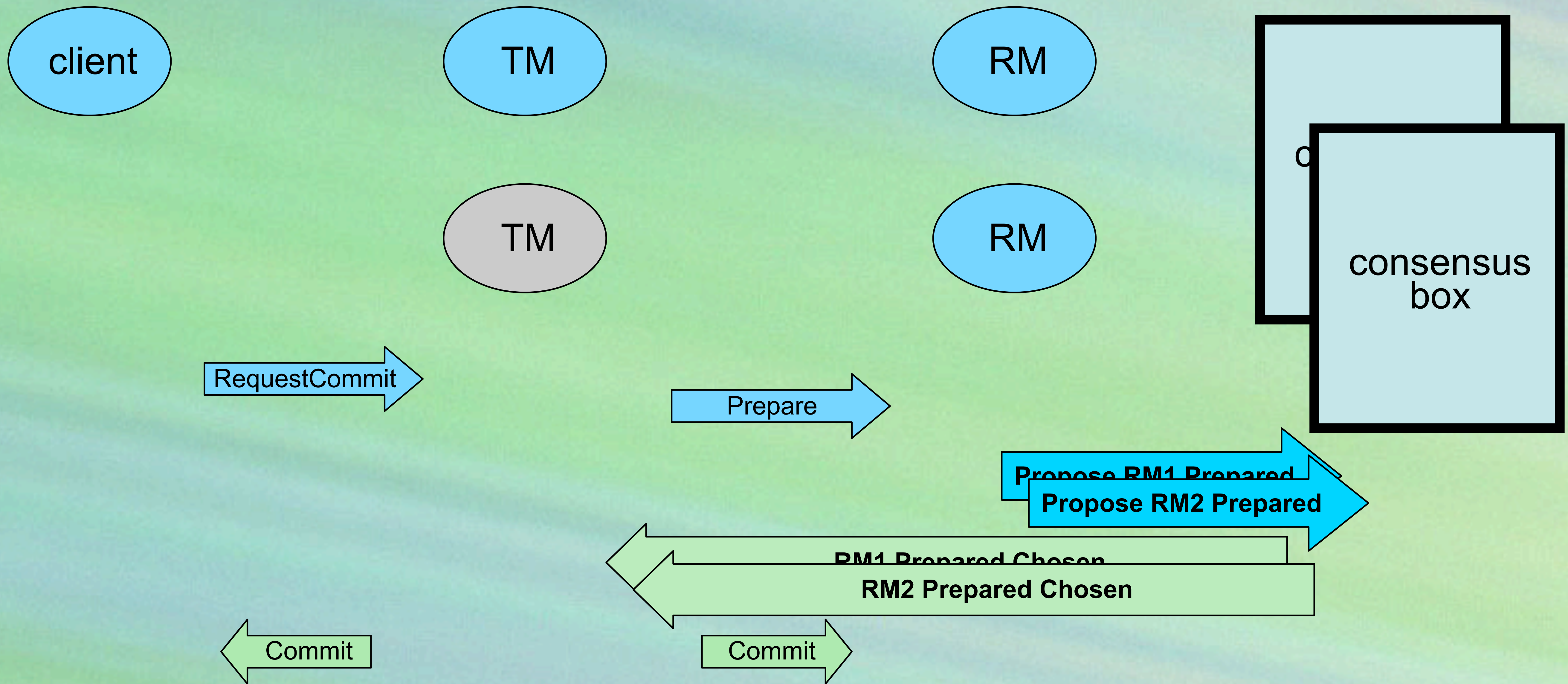
- Get consensus on each RM's choice.
- TM just combines consensus values.
- TM is "stateless"

Consensus for Commit
The Paxos Commit Approach



- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

Consensus for Commit
The Paxos Commit Approach



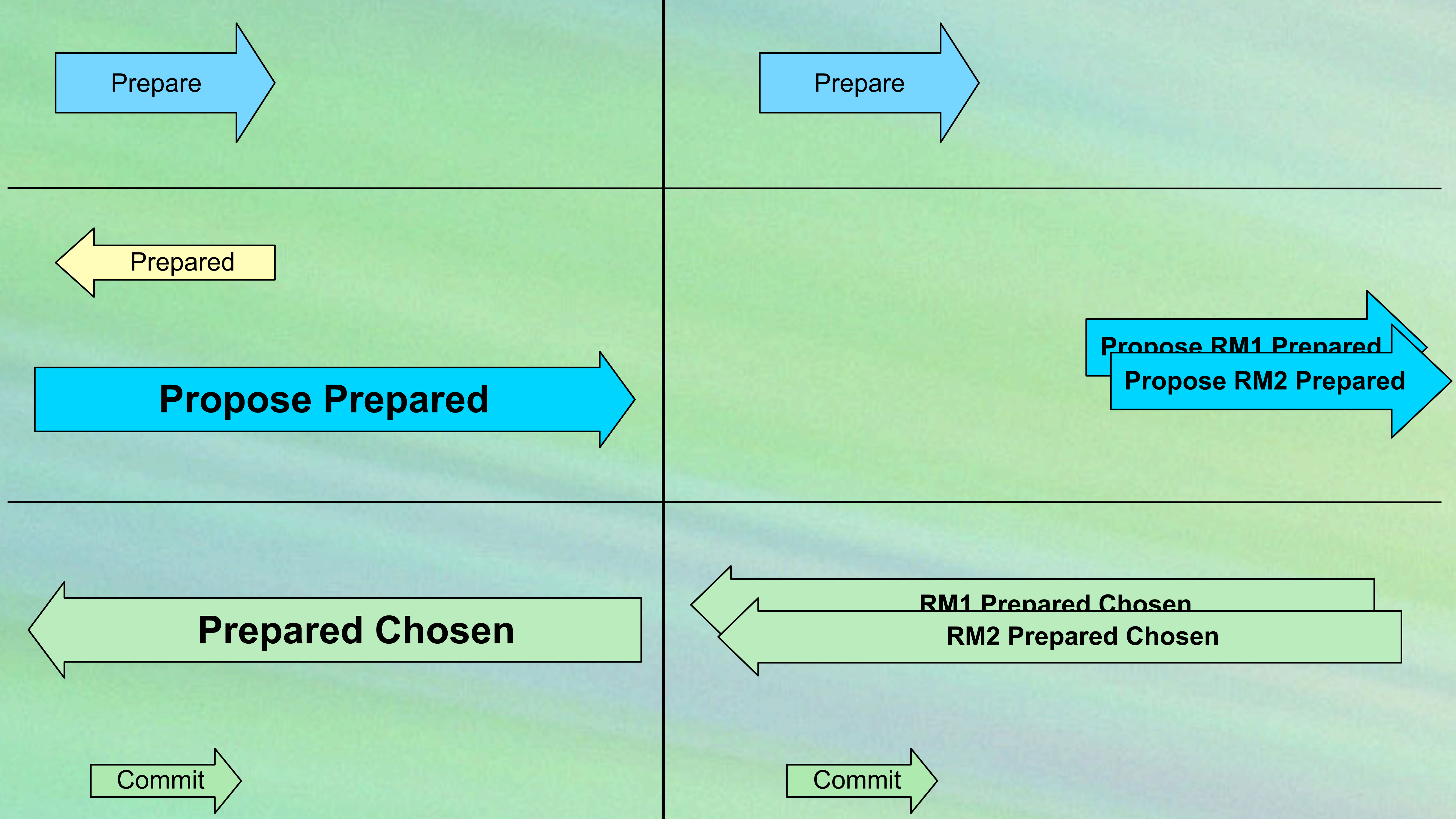
- Get consensus on each RM’s choice.
- TM just combines consensus values.
- TM is “stateless”

The Obvious Approach | Paxos Commit

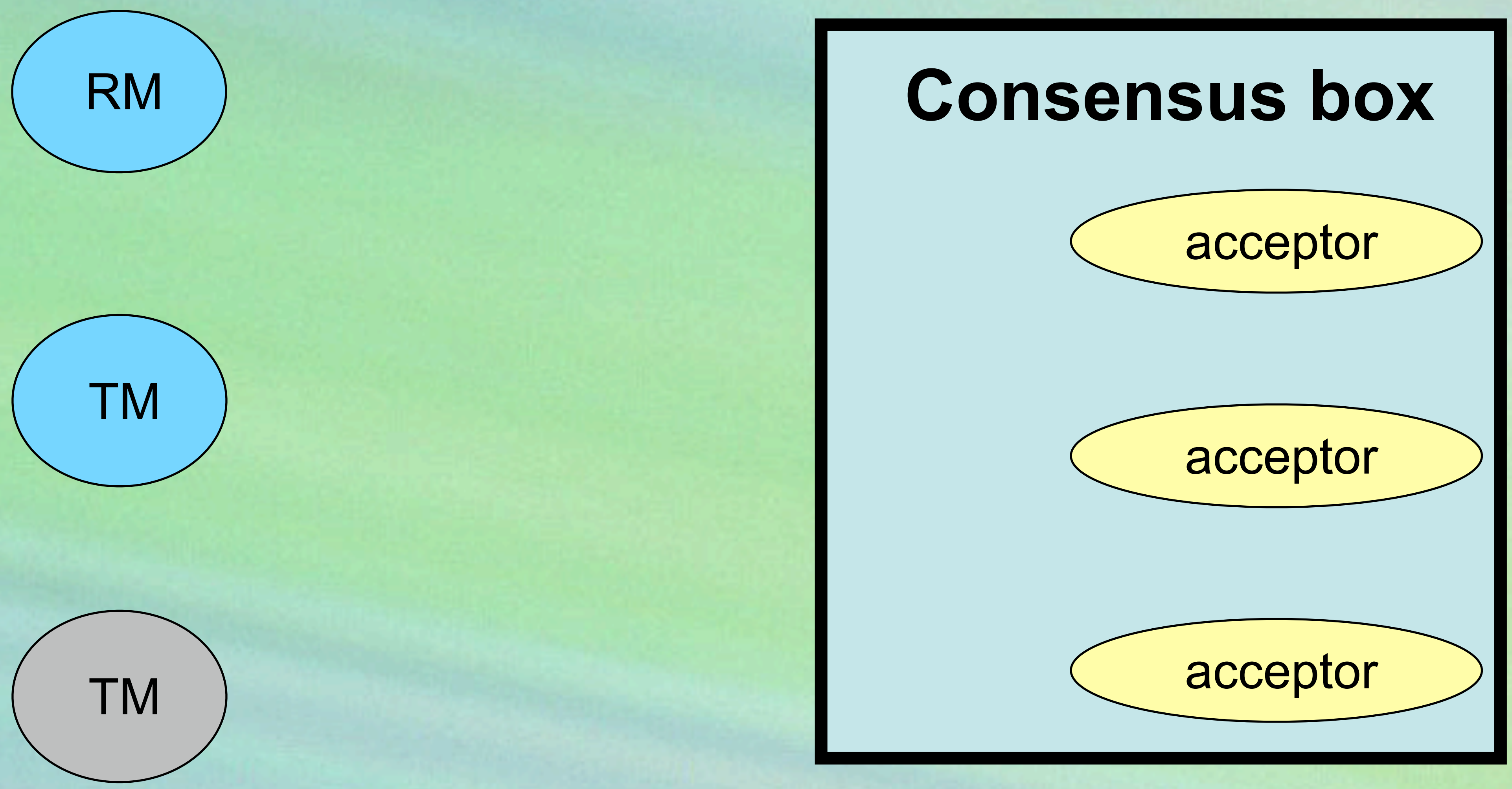
One fewer message delay

The Obvious Approach | Paxos Commit

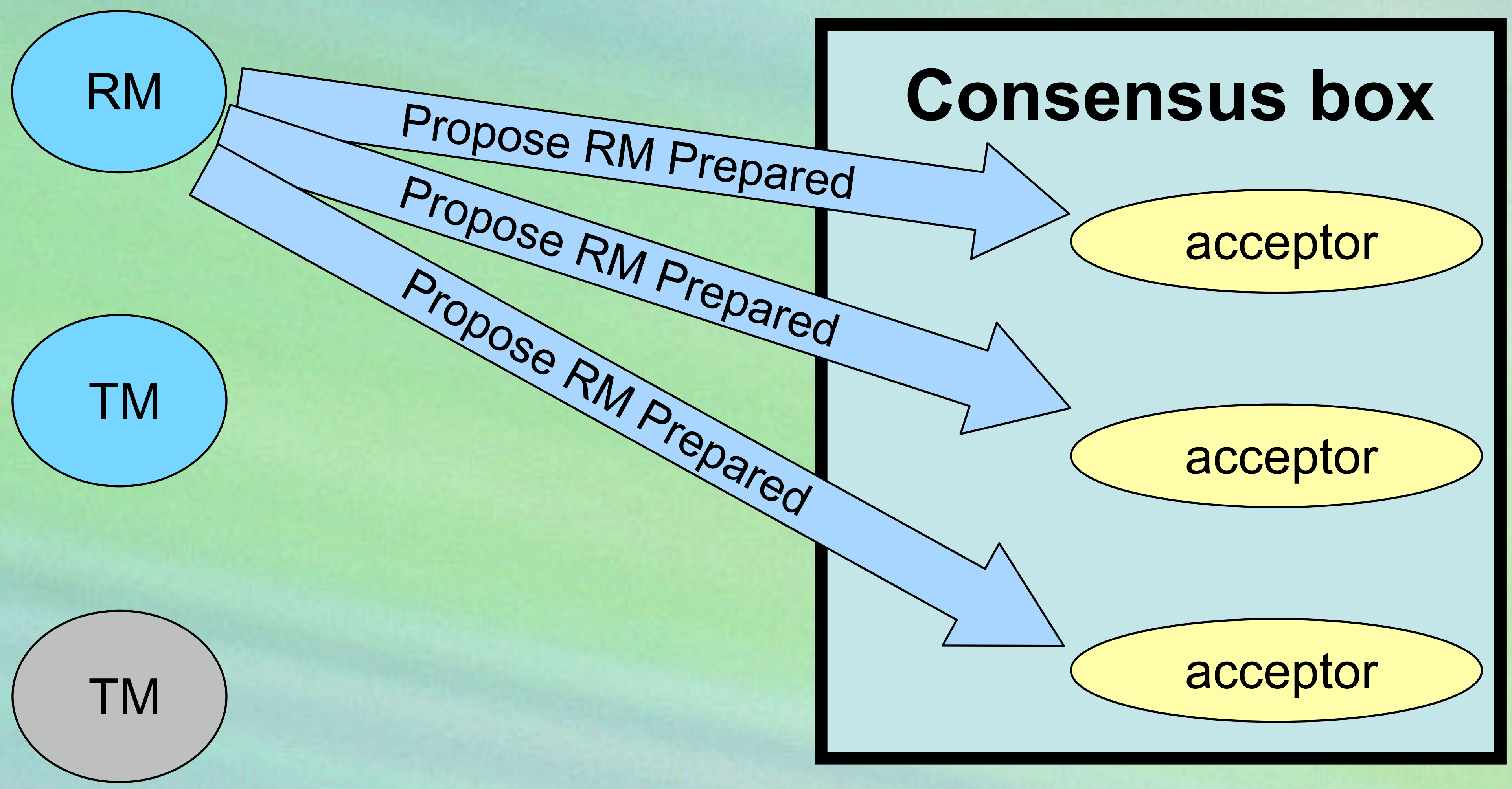
One fewer message delay



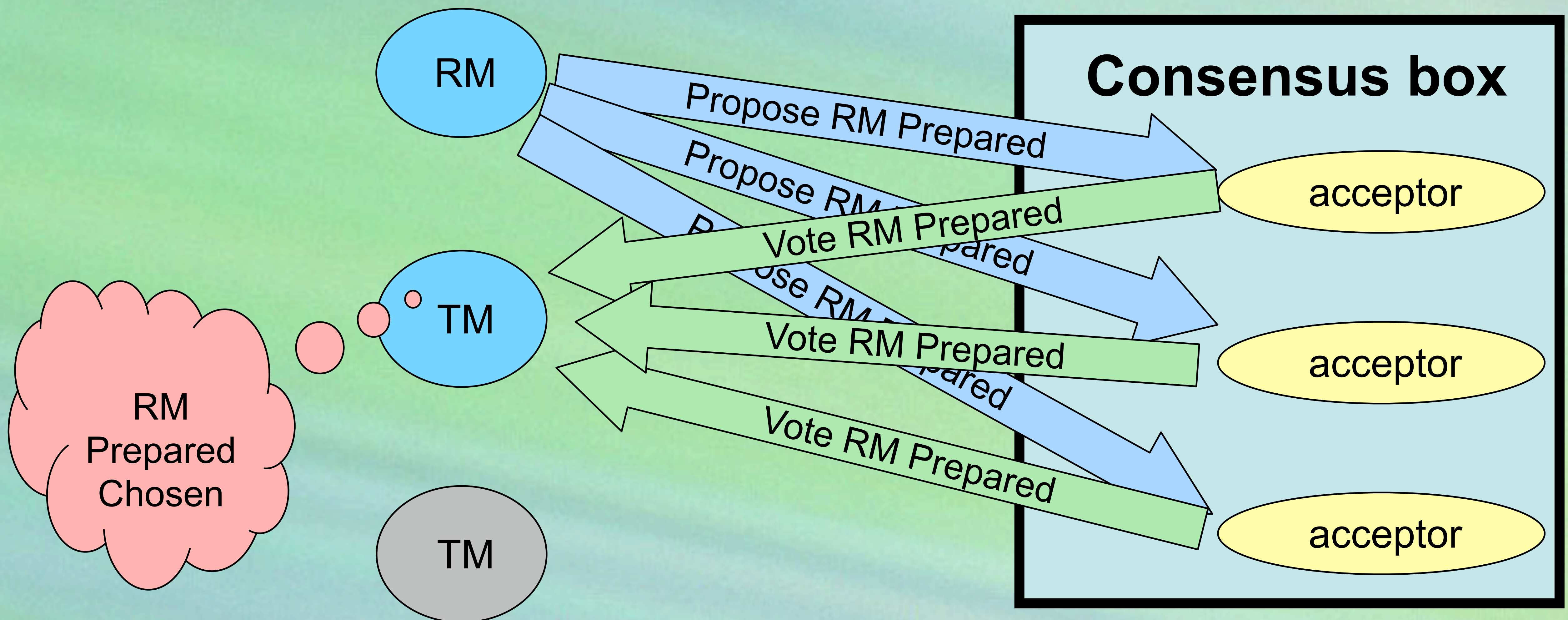
Consensus in Action



Consensus in Action

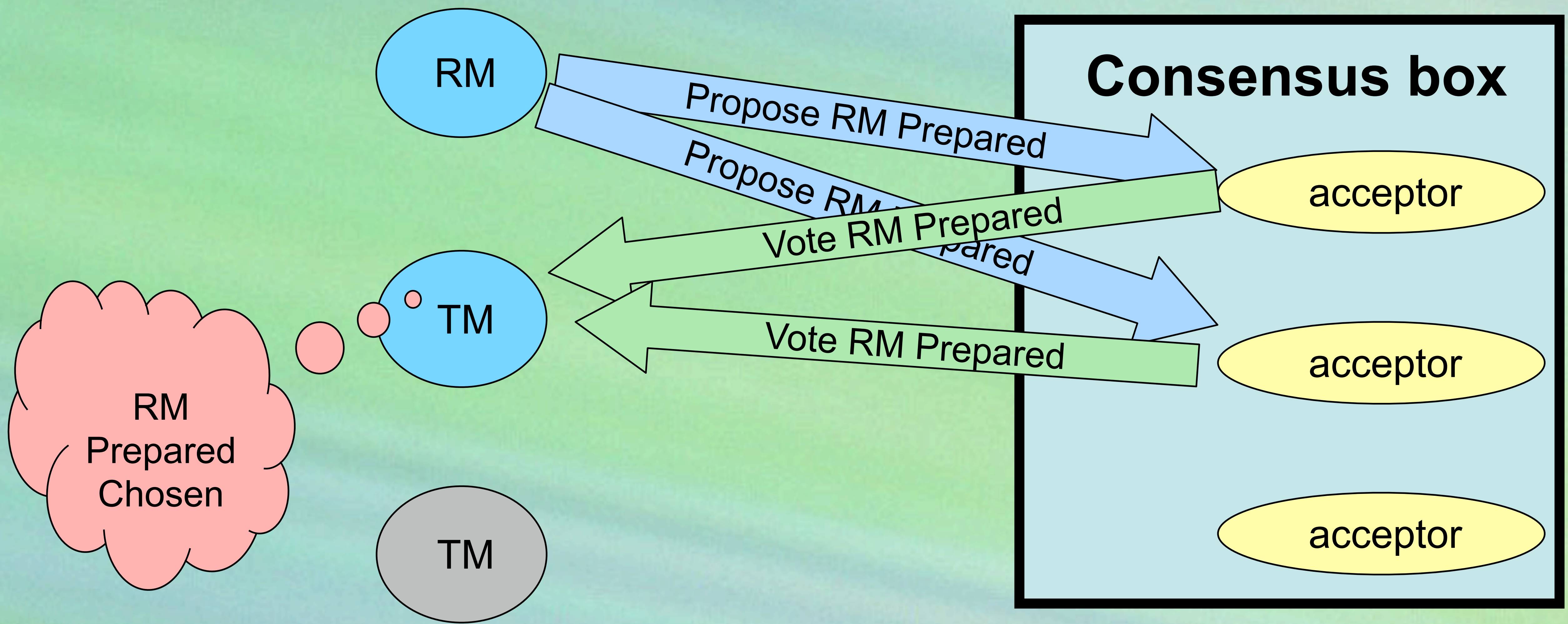


Consensus in Action



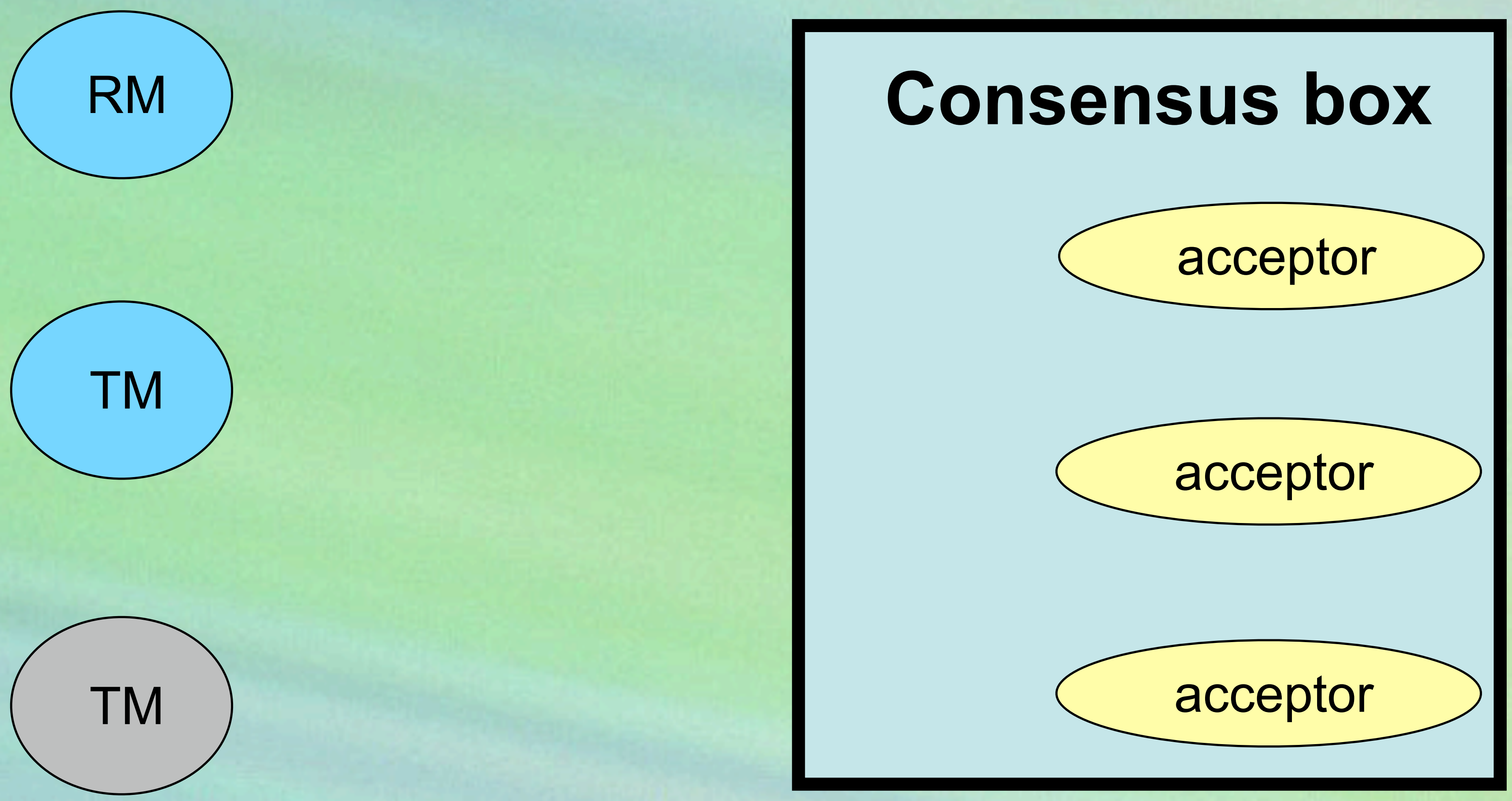
- The normal (failure-free) case

Consensus in Action

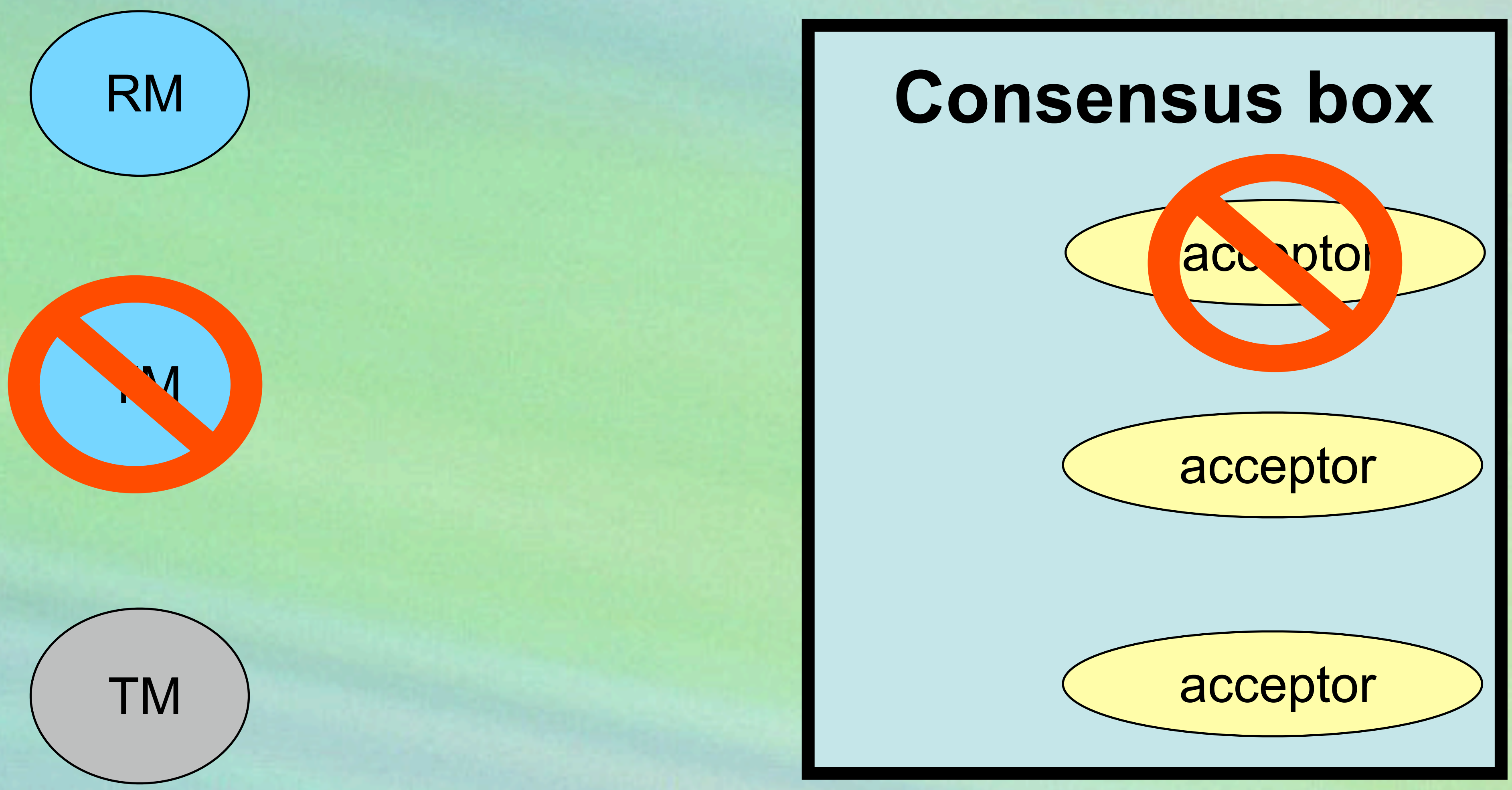


- The normal (failure-free) case
- Two message delays
- Can optimize

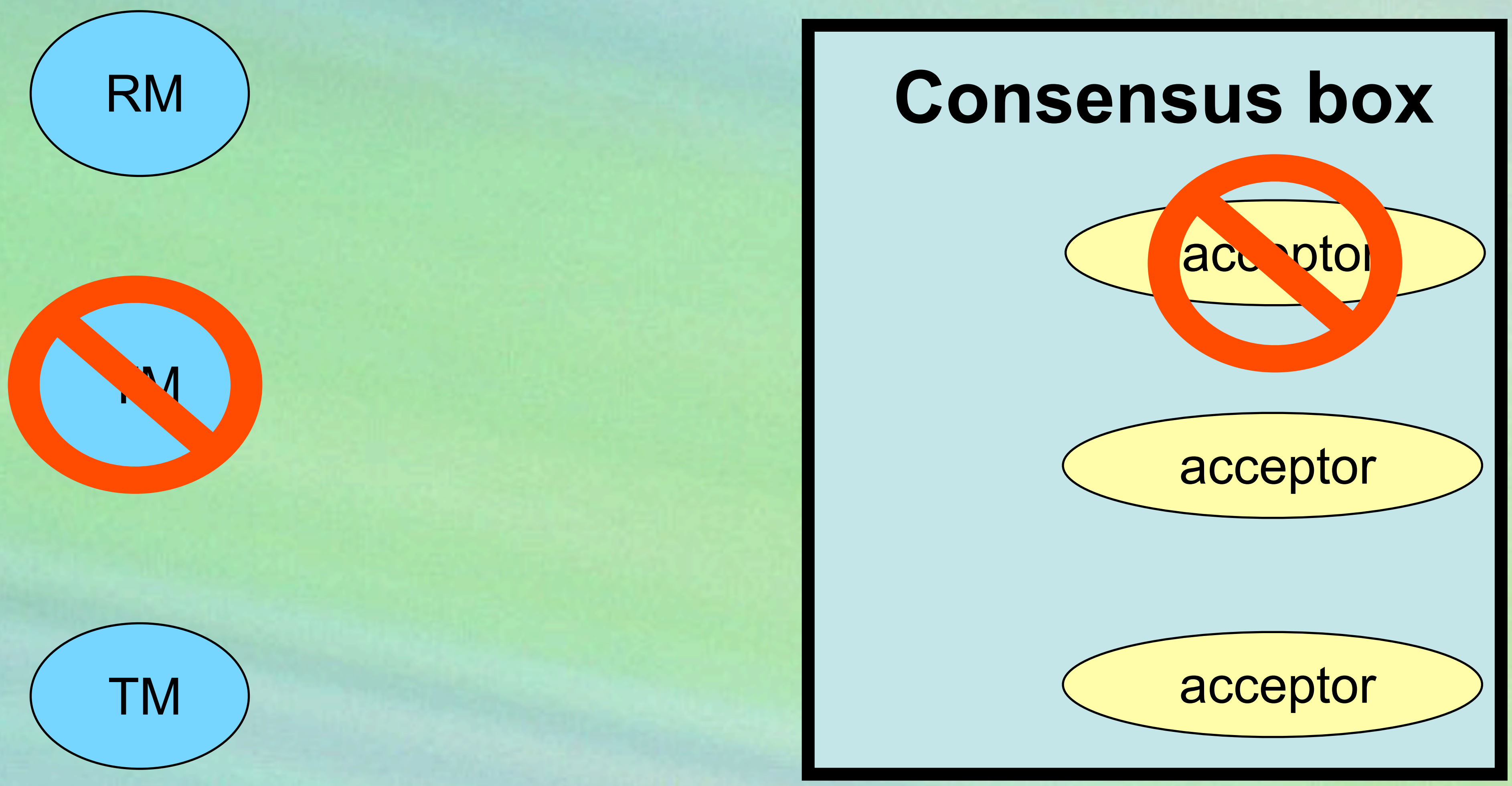
Consensus in Action



Consensus in Action



Consensus in Action



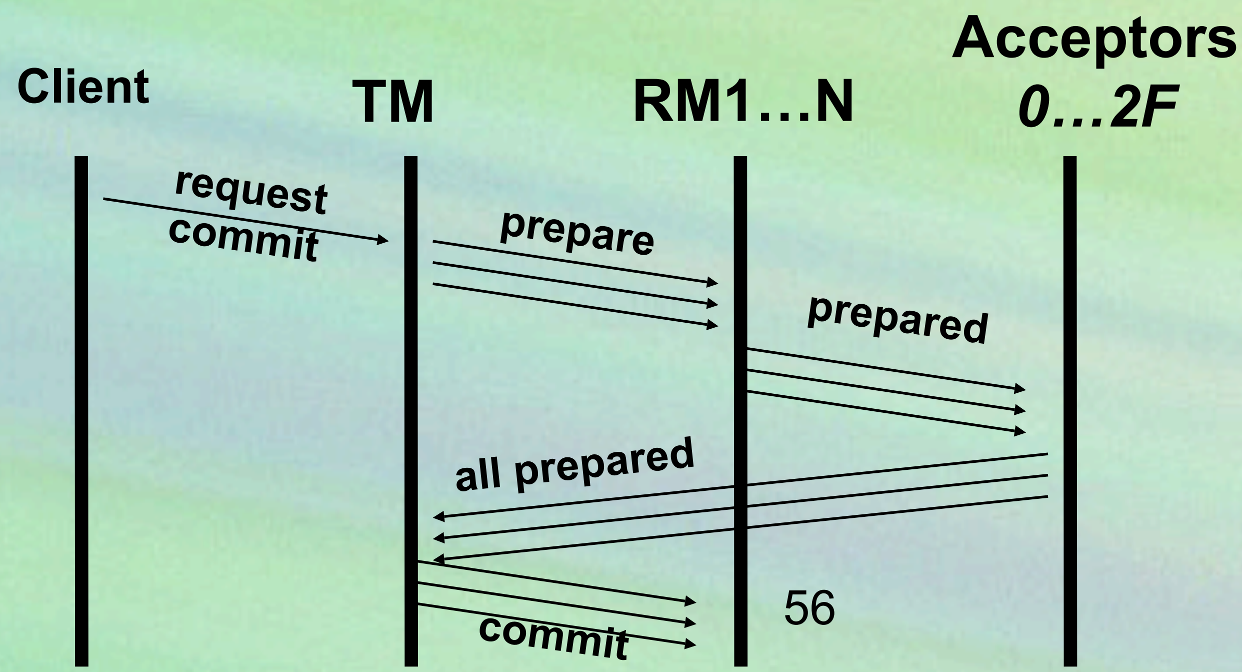
TM can always learn what was chosen,
or get *Aborted* chosen if nothing chosen yet;
if majority of acceptors working .

The Complete Algorithm

- Subtle.
- More weird cases than most people imagine.
- Proved correct.

Paxos Commit

- *N* RMs
- *2F+1* acceptors (~*2F+1* TMs)
- If *F+1* acceptors see all RMs prepared, then transaction committed.
- *2F(N+1)* + *3N* + *1* messages
  5 message delays
  2 stable write delays.



Two-Phase Commit

- $3N+1$ messages
- $N+1$ stable writes
- 4 message delays
- 2 stable-write delays

Paxos Commit

tolerates F faults

- $3N + 2F(N+1) + 1$ messages
- $N+2F+1$ stable writes
- 5 message delays
- 2 stable-write delays

Two-Phase Commit

- $3N+1$ messages
- $N+1$ stable writes
- 4 message delays
- 2 stable-write delays

Paxos Commit

tolerates F faults

- $3N+ 2F(N+1) +1$ messages
- $N+2F+1$ stable writes
- 5 message delays
- 2 stable-write delays

Same algorithm when $F=0$ and
TM = Acceptor

Summary

- Commit is common
- Two Phase commit is good but…
  It is the un-availability protocol
- Paxos commit is non-blocking
  if there are at most *F* faults.
- When *F=0* (no fault-tolerance),
  Paxos Commit == 2PC