MC714 - Sistemas Distribuídos

Introdução ao algoritmo de Paxos

Islene Calciolari Garcia

Instituto de Computação - Unicamp

Primeiro Semestre de 2015

Sumário

Paxos Explained from Scratch

Hein Meling and Leander Jehl

University of
Stavanger

hein.meling@uis.no

International Conference On Principles of Distributed Systems

Fonte: Hein Meling e Leander Jehl - OPODIS2013

What is Paxos and why is it Relevant?

- Fault tolerant consensus protocol
- Used to order client requests in a fault tolerant server
  - For example a fault tolerant resource manager
- Used in production systems: Chubby, ZooKeeper, and Spanner
- It is always safe

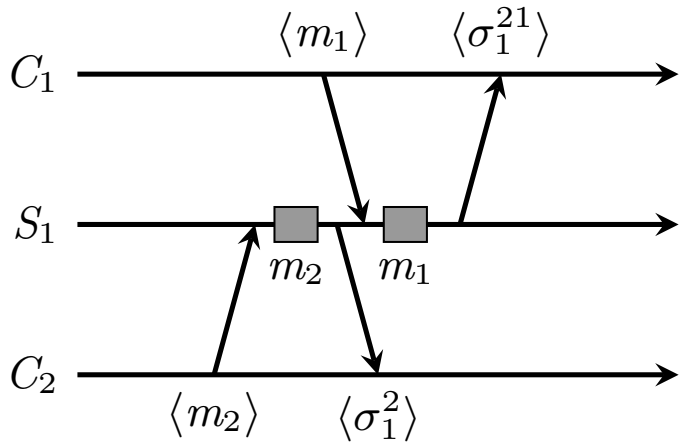
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Objectives and Approach

- Explain Paxos
  - Using visual aids
  - In a step-wise manner
  - With minimal changes in each step
- Objective
  - Understand why it works and why the solution is necessary
  - (no focus on how to implement or formally prove it)
- Approach
  - Use a simple client/server system as base
  - To build fault tolerant server (replicated state machine)
  - Construct Multi-Paxos
  - Decompose Multi-Paxos into Paxos

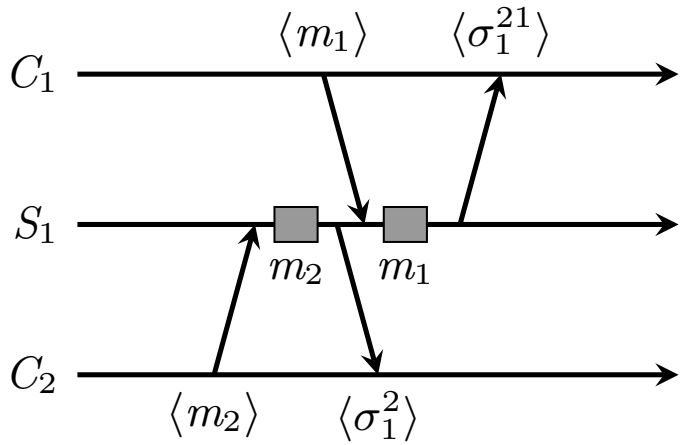
Fonte: Hein Meling e Leander Jehl - OPODIS2013

A Stateful Service: *SingleServer*



Fonte: Hein Meling e Leander Jehl - OPODIS2013

A Stateful Service: *SingleServer*



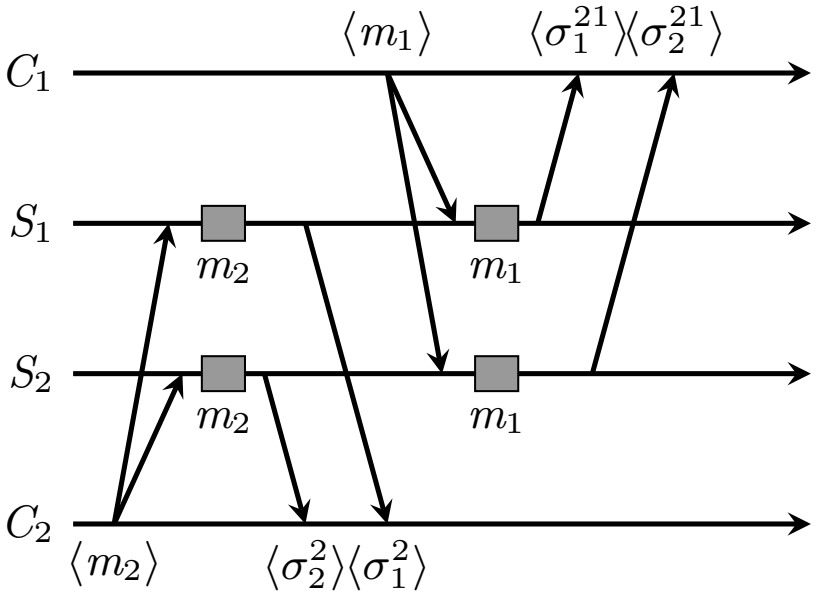
- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- Corresponds to execution sequence: $m_2 m_1$

Fonte: Hein Meling e Leander Jehl - OPODIS2013

We Want to Make the Service Fault Tolerant!

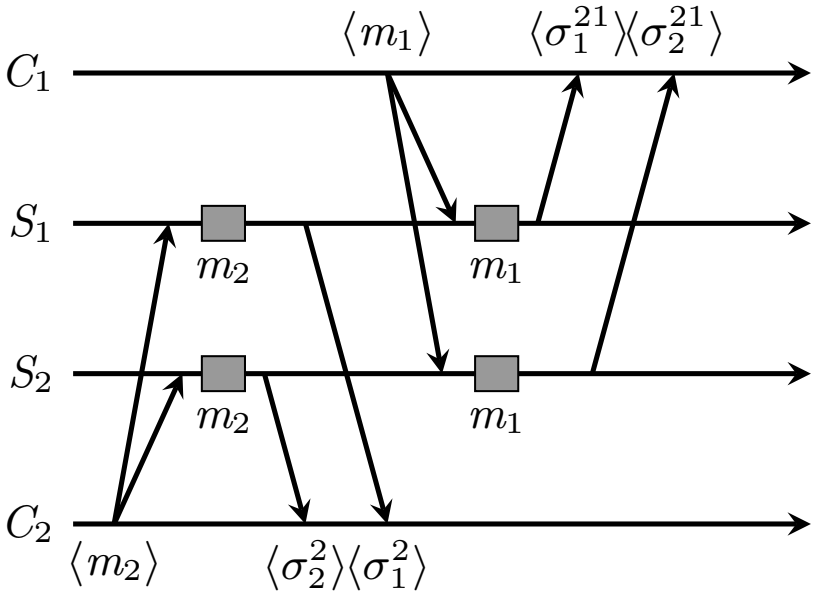
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Fault Tolerance with Two Servers



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Fault Tolerance with Two Servers



- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
  - σ^2 is a prefix of σ^{21}
- Corresponds to execution sequence: $m_2 m_1$

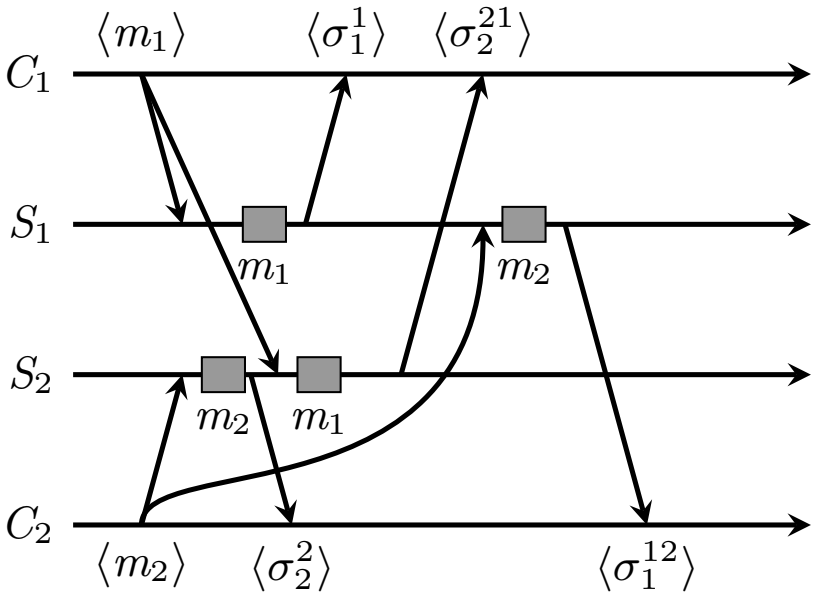
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Deterministic State Machine

- The service is implemented as a deterministic state machine
- Thus processing requests results in unique state transitions:
  - Therefore $\sigma_1^2 = \sigma_2^2$ and $\sigma_1^{21} = \sigma_2^{21}$.
- Clients can detect and suppress identical replies

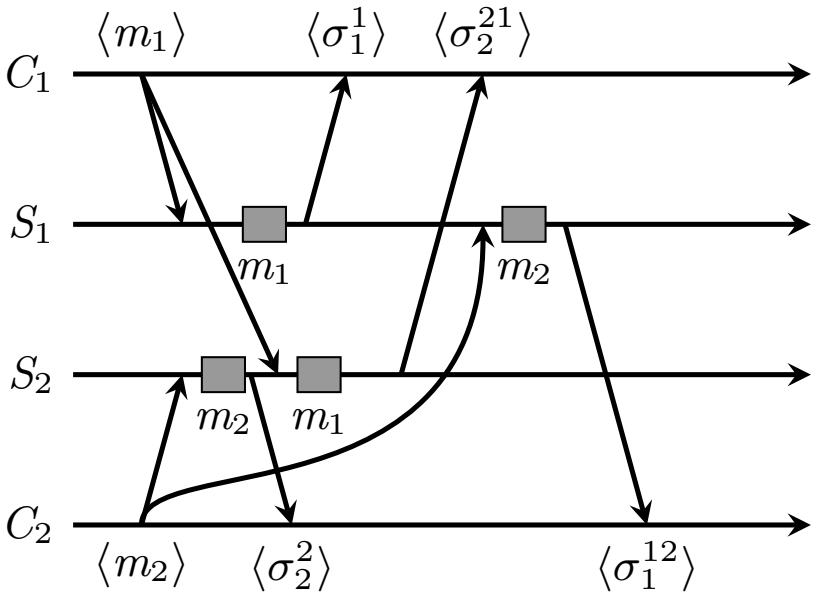
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Fault Tolerance with Two Servers: Whoops!



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Fault Tolerance with Two Servers: Whoops!



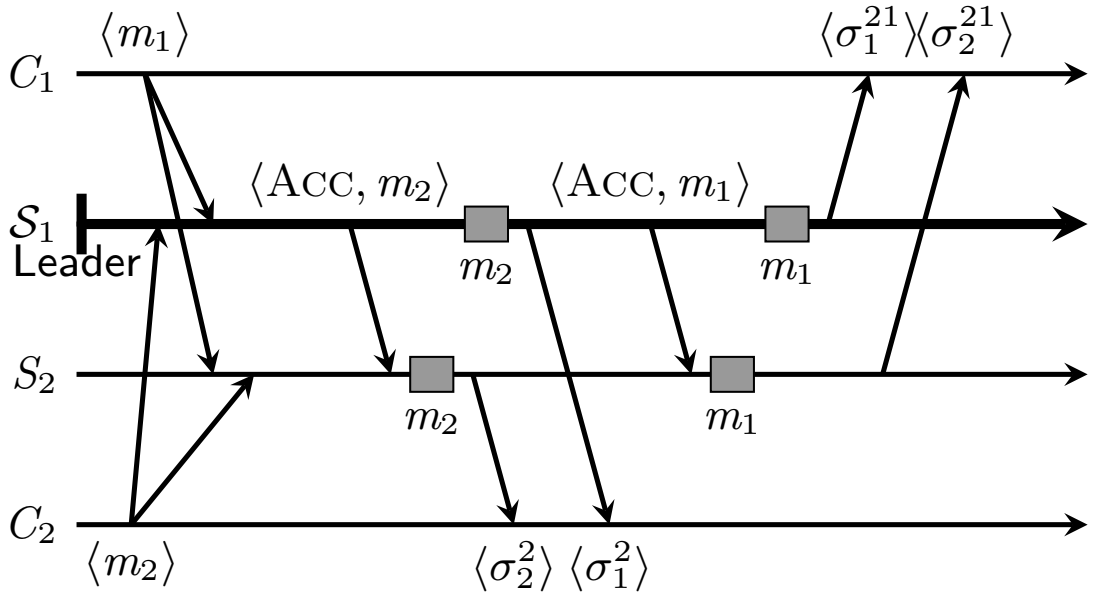
- Client C_2 sees: $\sigma^2\sigma^{12}$
  - σ^2 is not a prefix of σ^{12}
- Client C_1 sees: $\sigma^1\sigma^{21}$
  - σ^1 is not a prefix of σ^{21}
- Corresponds to execution sequence at
  - S_1: $m_1 m_2$
  - S_2: $m_2 m_1$

Fonte: Hein Meling e Leander Jehl - OPODIS2013

We Need to Order Client Requests!

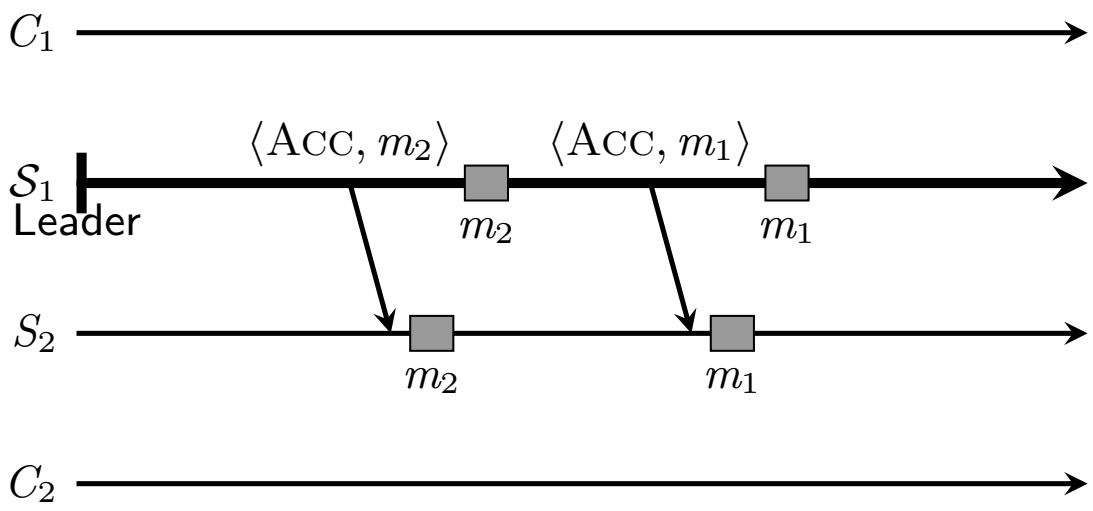
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Let's Designate a Leader to Order Requests



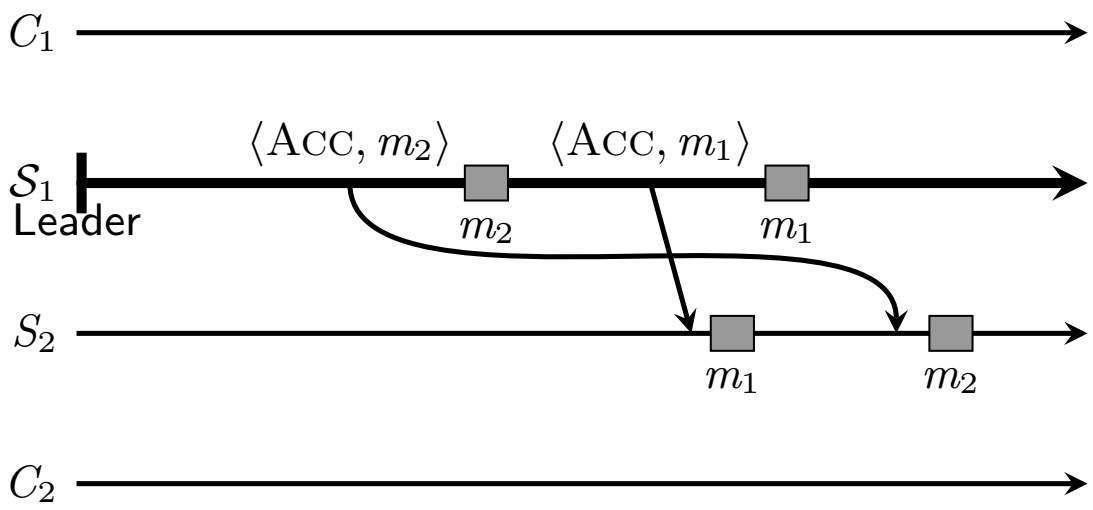
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Without Clients



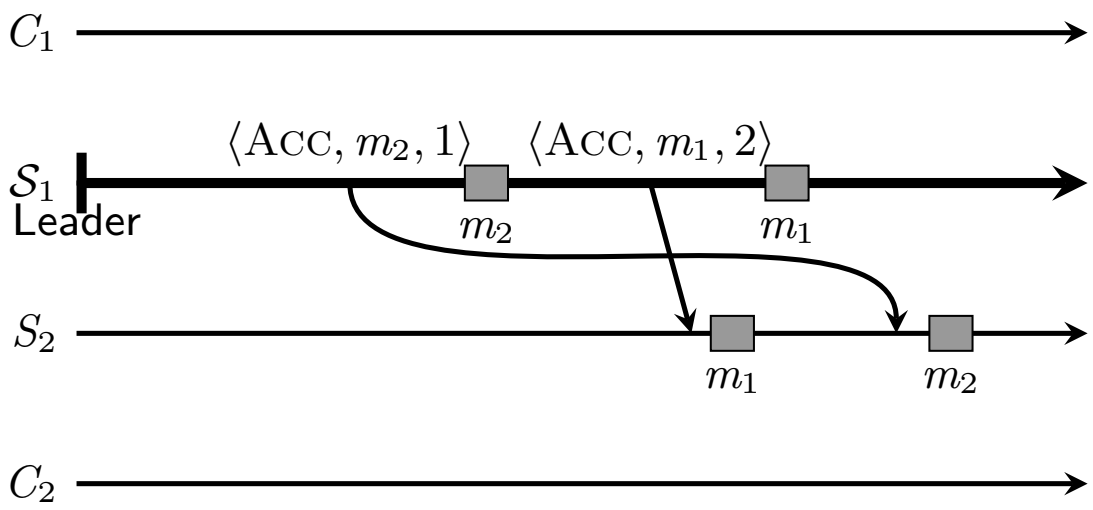
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Problem: Also Accept Messages can be Reordered



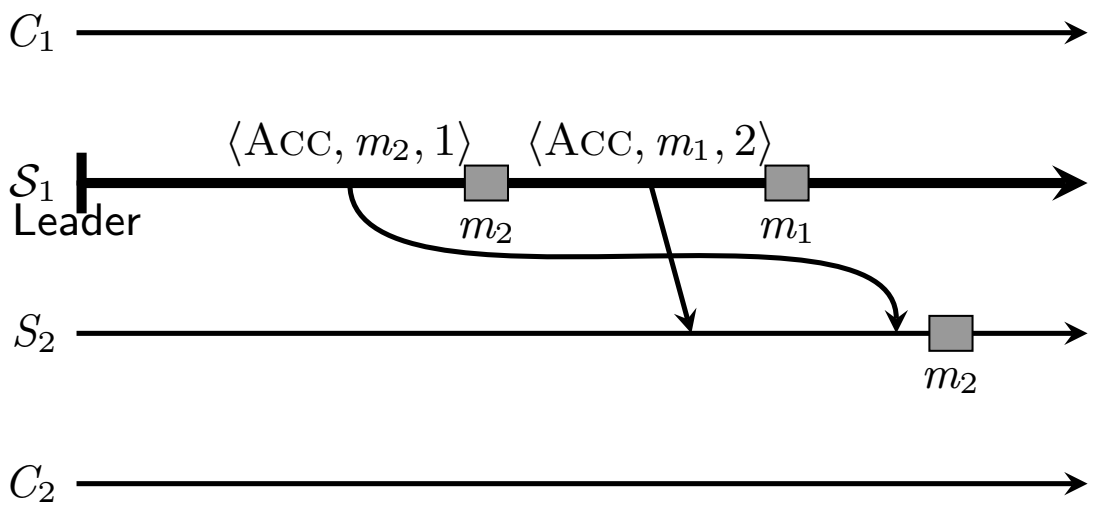
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Add Sequence Numbers



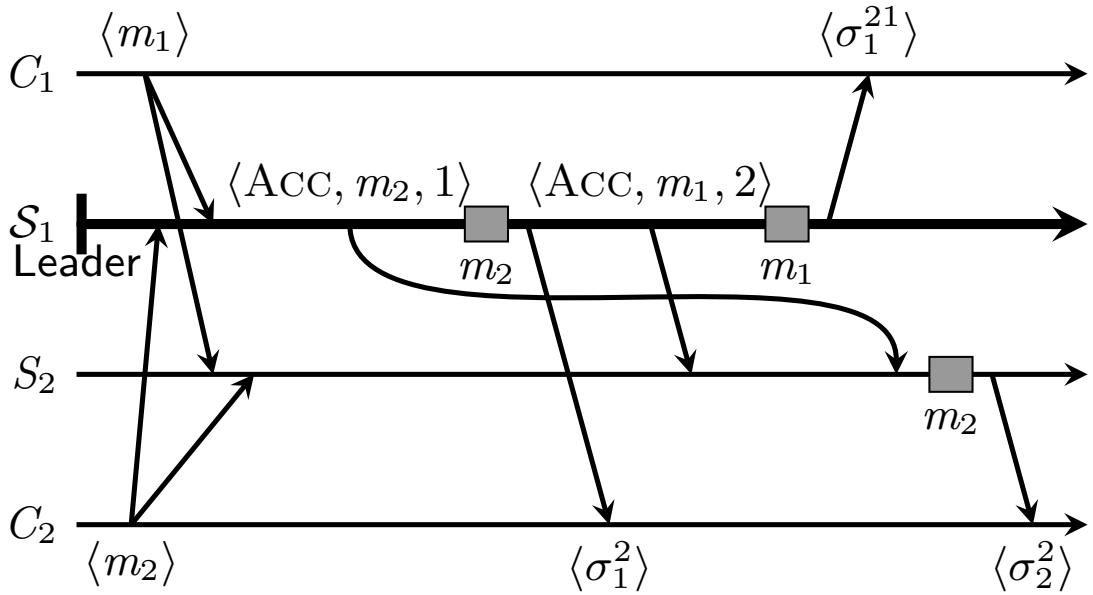
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Discard Out-of-Order Messages



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Now with Clients



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Clients Observe The Same Server States as Before

- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- However, S_2 didn't execute m_1
  - Q: What to do?

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Clients Observe The Same Server States as Before

- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- However, S_2 didn't execute m_1
  - Q: What to do?
  - A1: Buffer

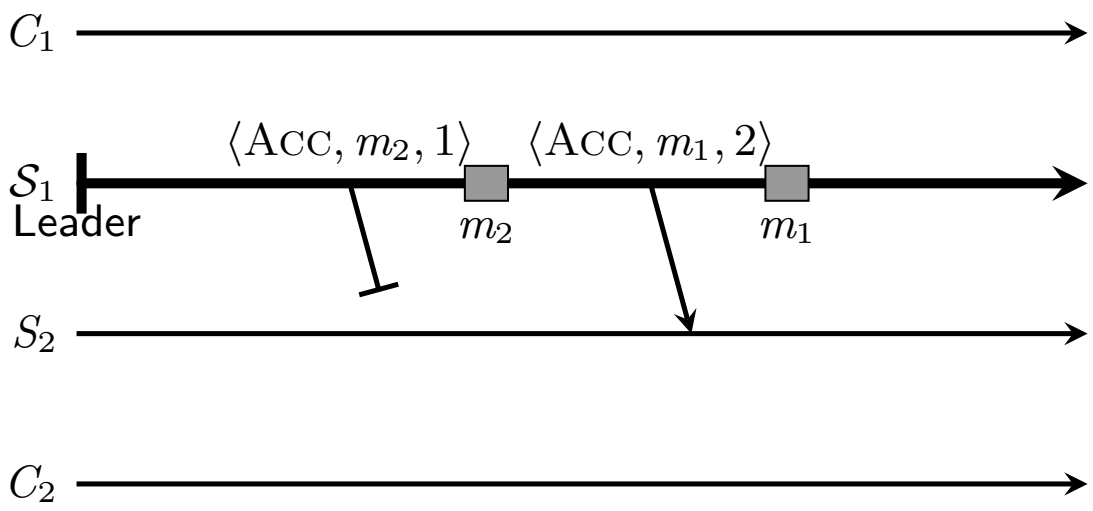
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Clients Observe The Same Server States as Before

- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- However, S_2 didn't execute m_1
  - Q: What to do?
  - A1: Buffer
  - A2: Retransmission mechanism

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Problem: Message Loss – S_2 Won't Execute Anything

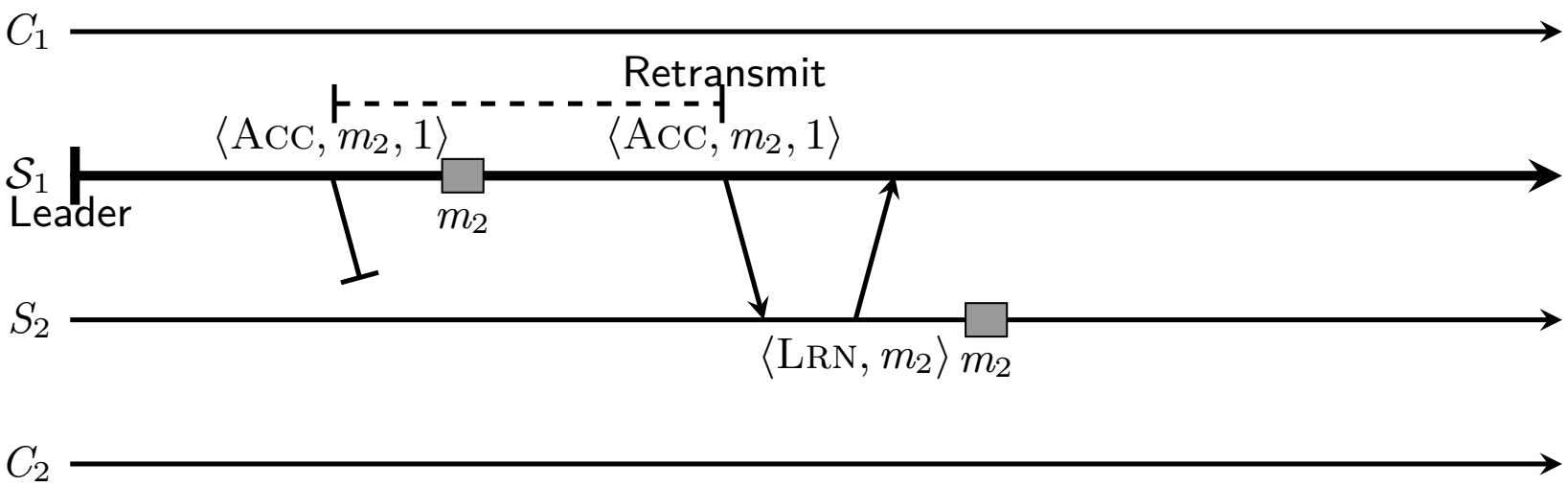


Fonte: Hein Meling e Leander Jehl - OPODIS2013

We Need a Retransmission Mechanism!

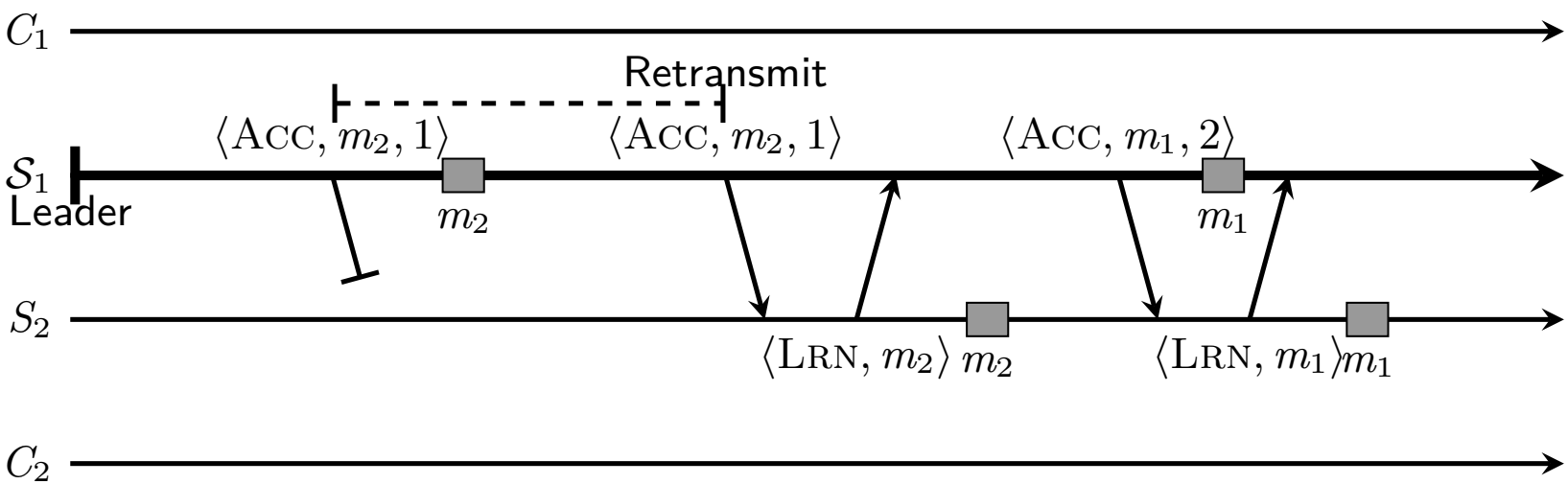
Fonte: Hein Meling e Leander Jehl - OPODIS2013

A Learn Stops Retransmission



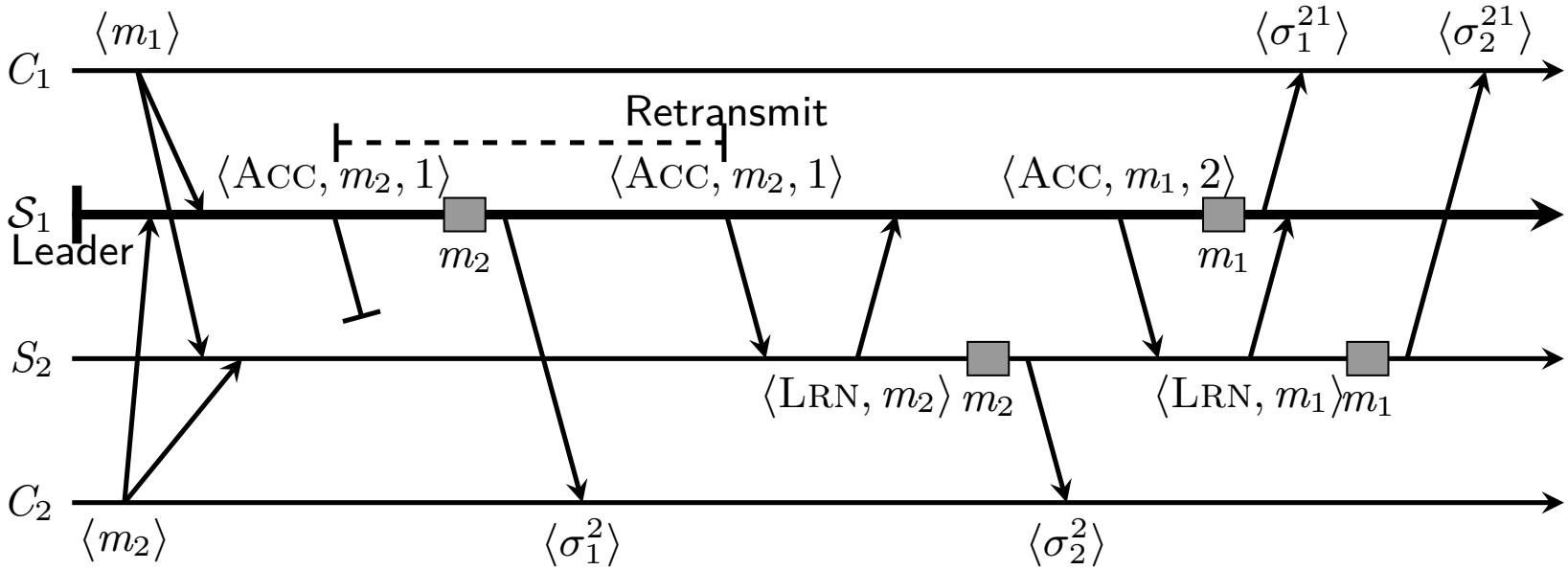
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Don't Send New Accept Until Learn



Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Clients



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap

- A leader
  - To decide the order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - To cope with message reordering
- Retransmission mechanism
  - To cope with message loss
  - Leader only sends next accept when learn from S_2
  - Allows leader to *make progress*, as long as messages are not lost infinitely often

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap

- A leader
  - To decide the order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - To cope with message reordering
- Retransmission mechanism
  - To cope with message loss
  - Leader only sends next accept when learn from S_2
  - Allows leader to *make progress*, as long as messages are not lost infinitely often

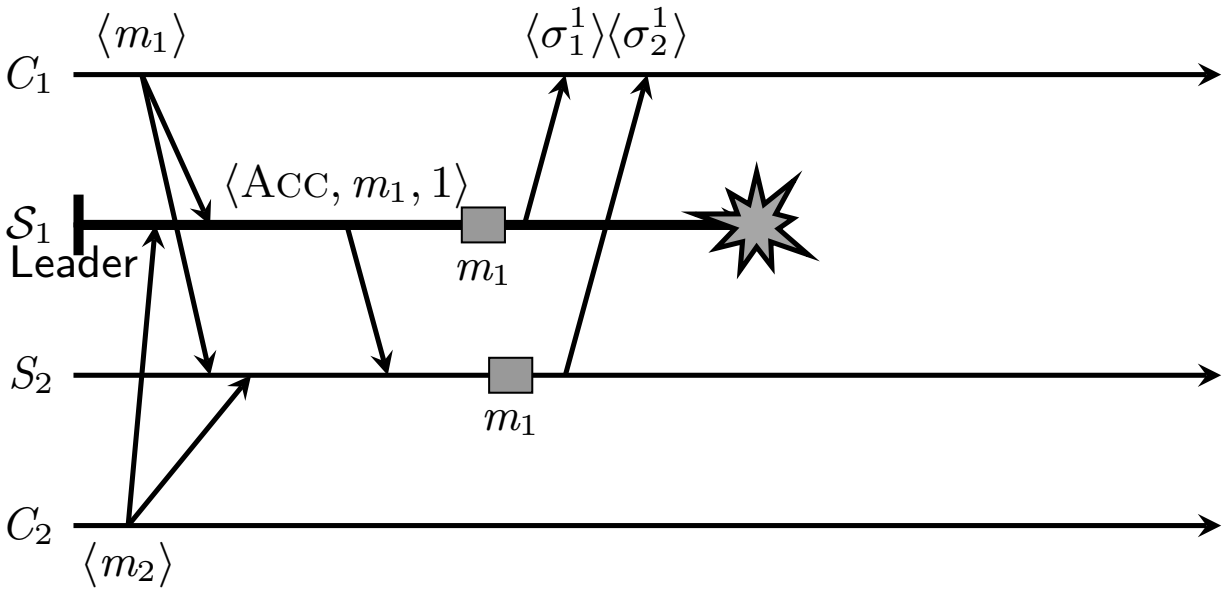
Combination of mechanisms:
RetransAccept protocol

Fonte: Hein Meling e Leander Jehl - OPODIS2013

What About Server Crashes?

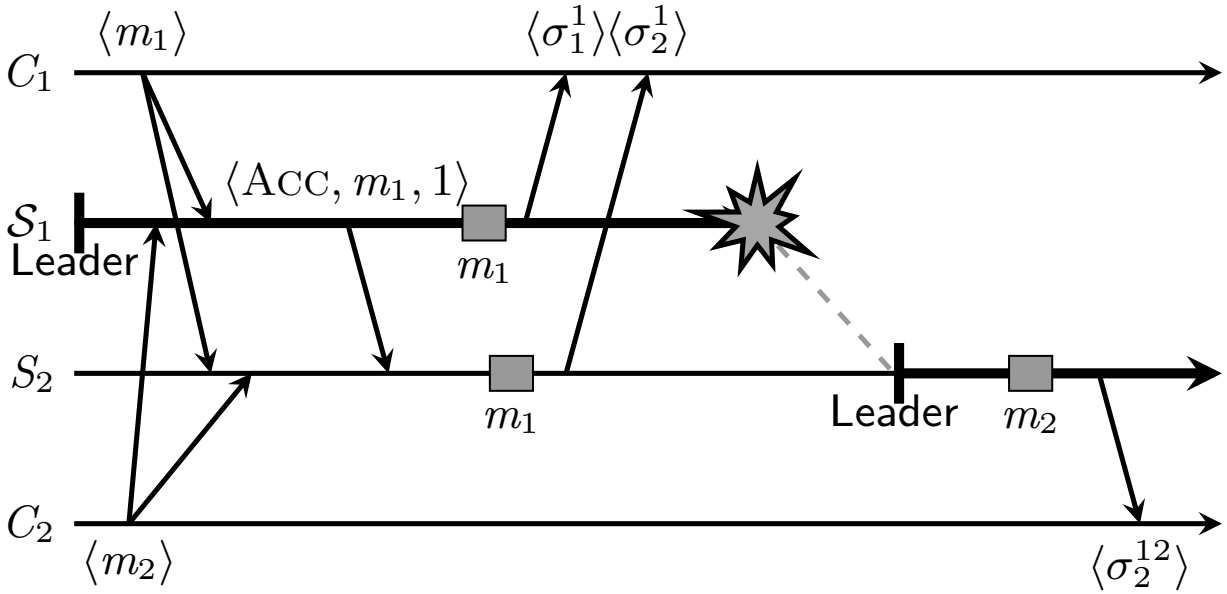
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Crash



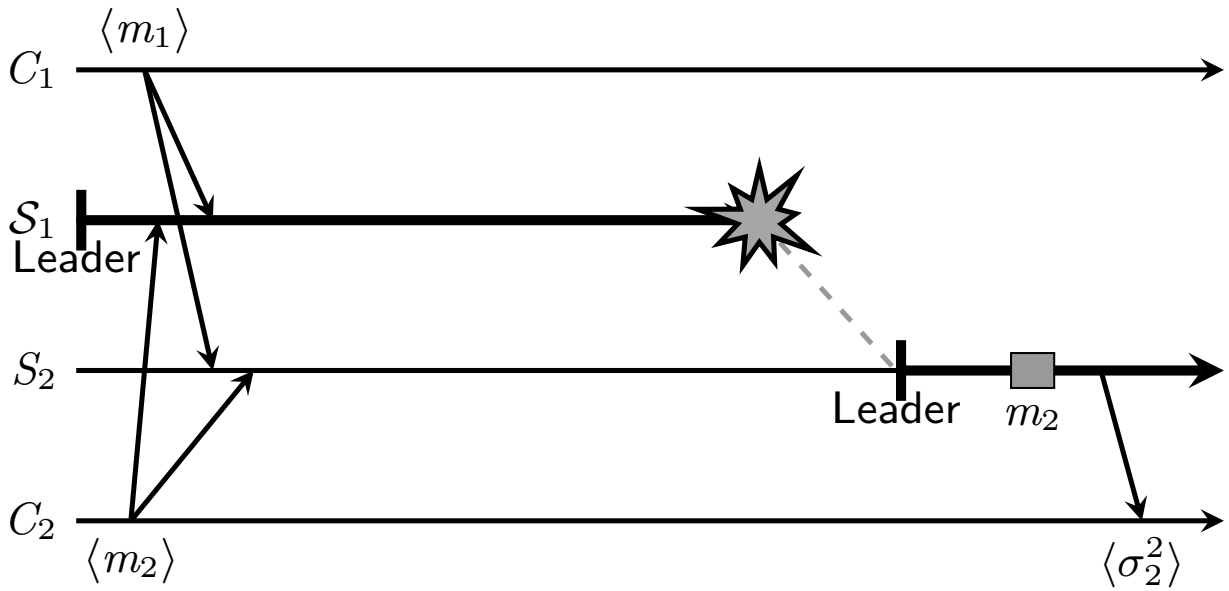
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Crash: Leader Takeover



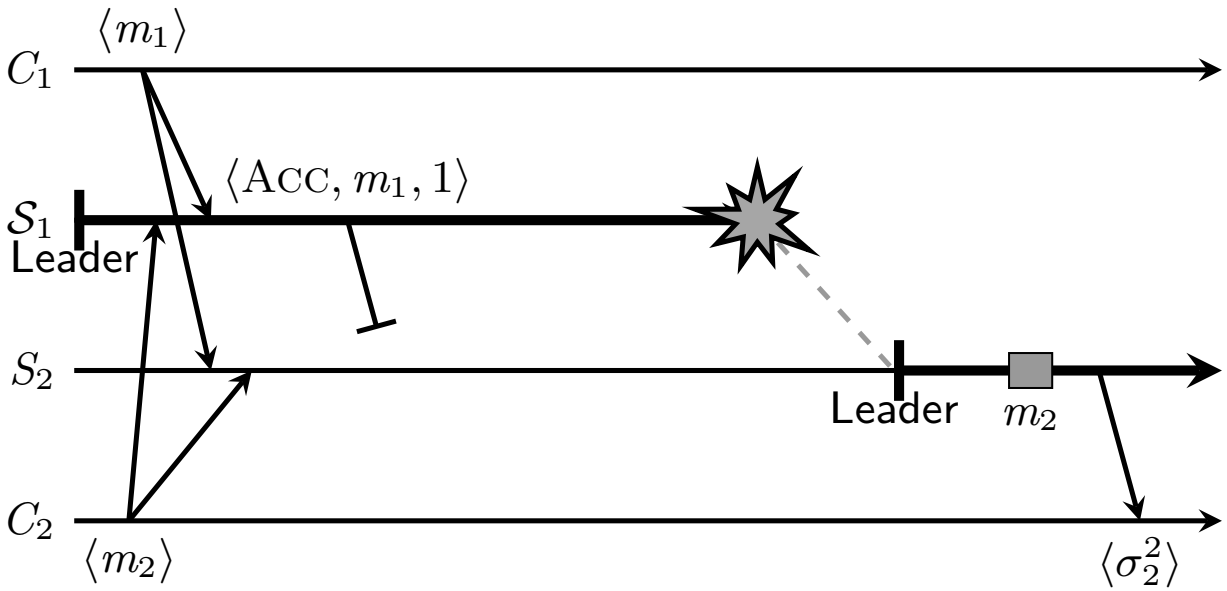
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Single Server Rule: Case 1



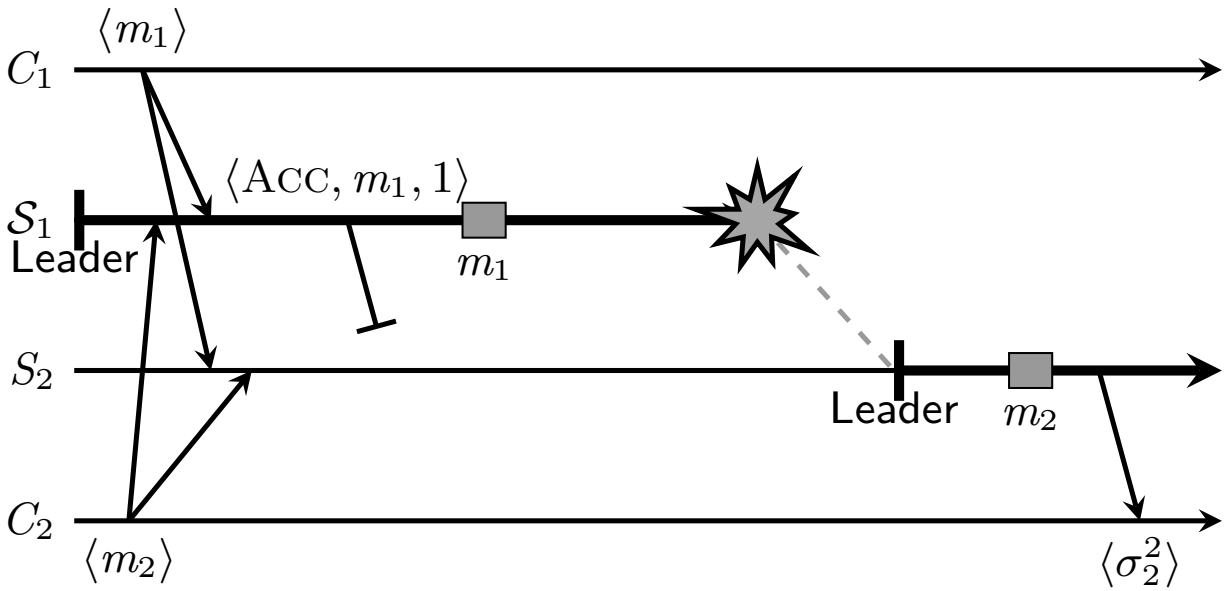
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Single Server Rule: Case 2



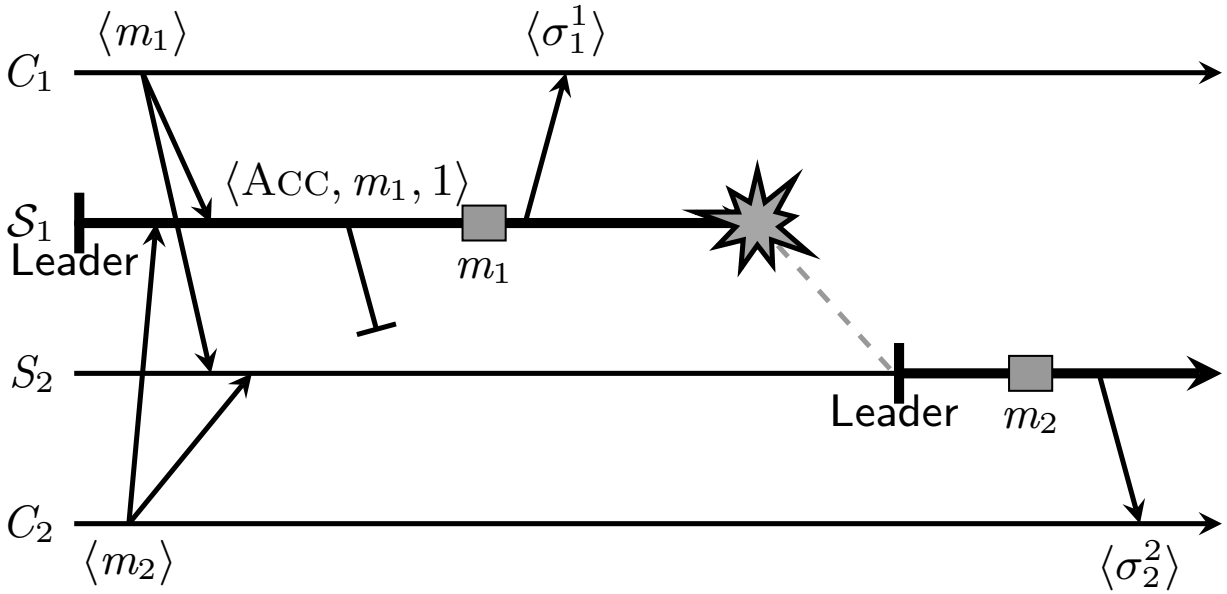
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Single Server Rule: Case 3



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Single Server Rule: Case 4 – A Problem



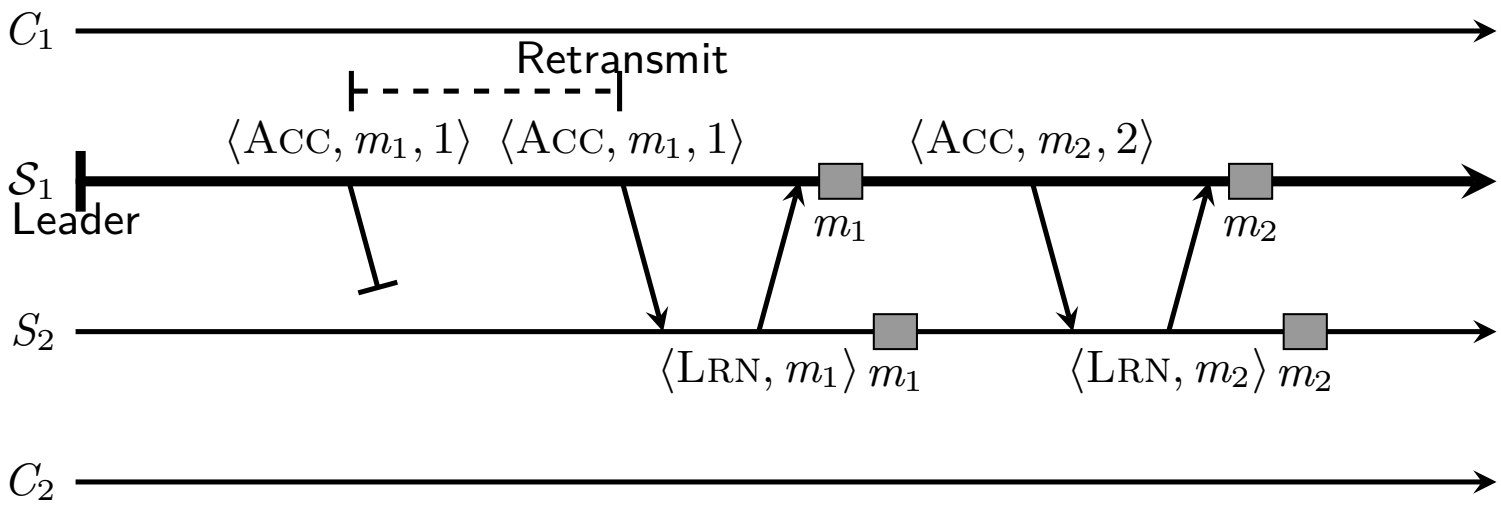
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Single Server Rule: Case 4 – A Problem

- Imagine that (S_1, S_2) implements a fault tolerant resource manager, e.g. a lock service
- Both clients could have gotten the lock

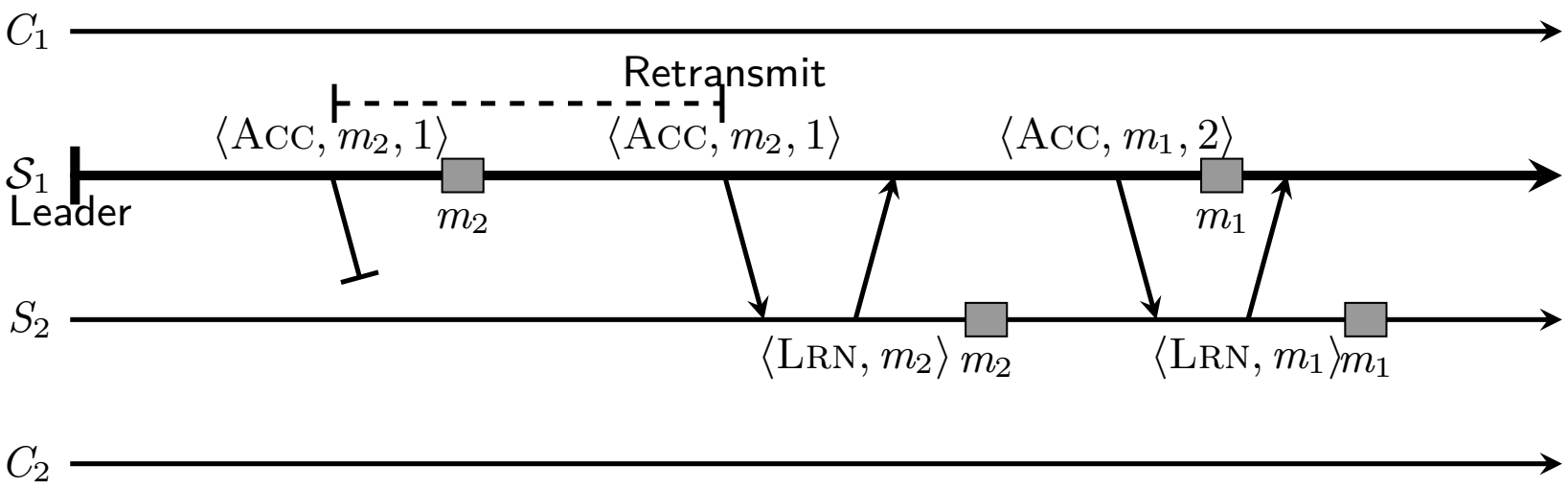
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Solution: Leader Waits for Learn Before Executing



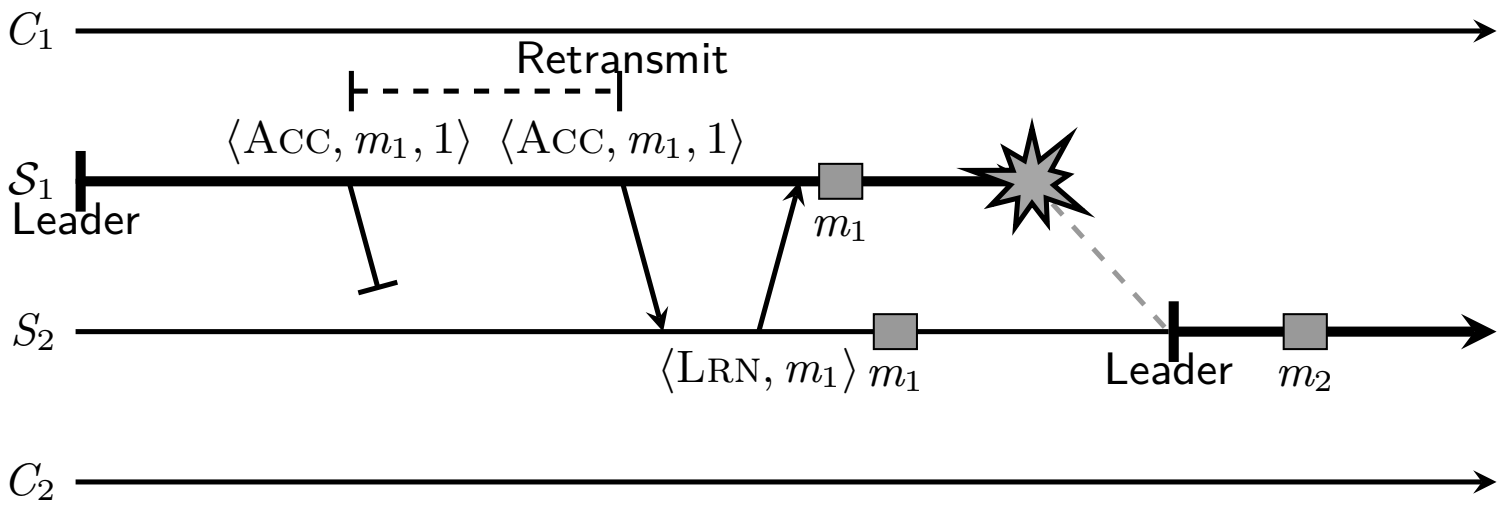
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recall Earlier Version



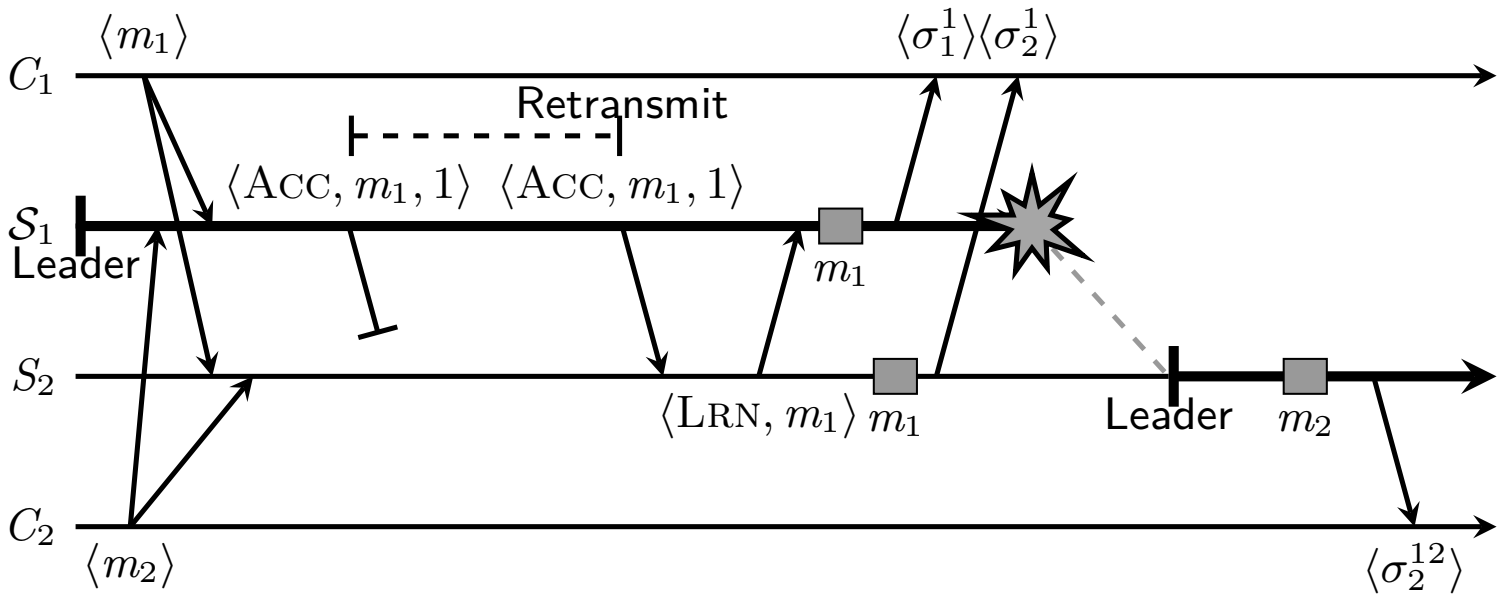
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Now Leader Takeover is Safe



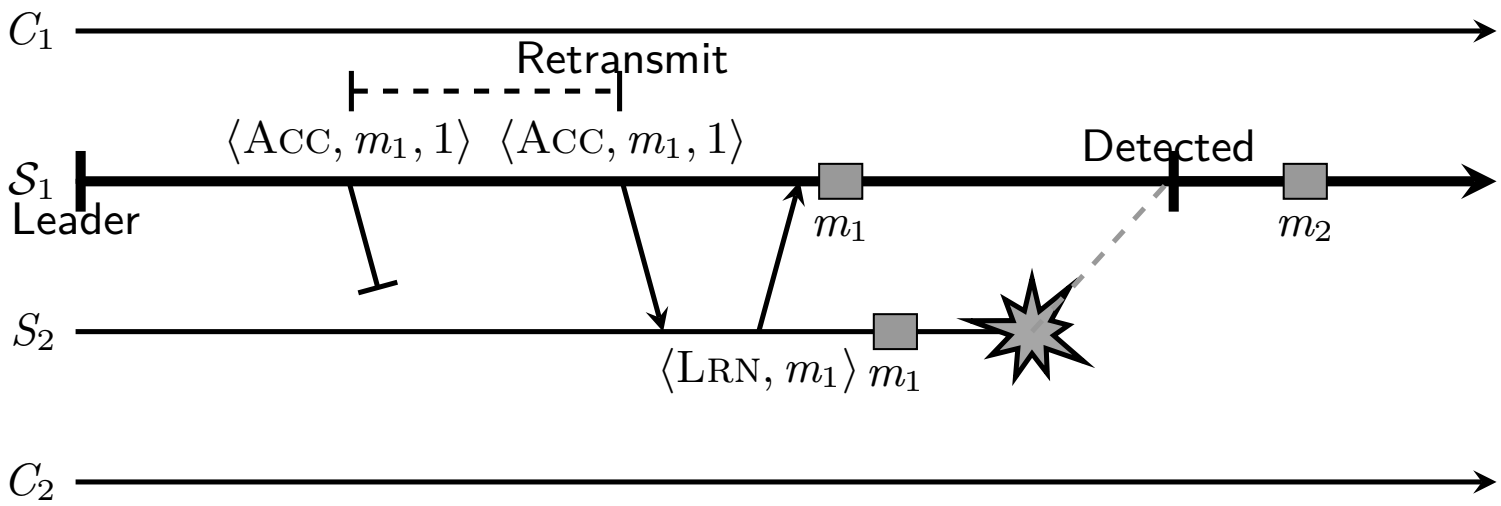
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Let's Add Client Messages



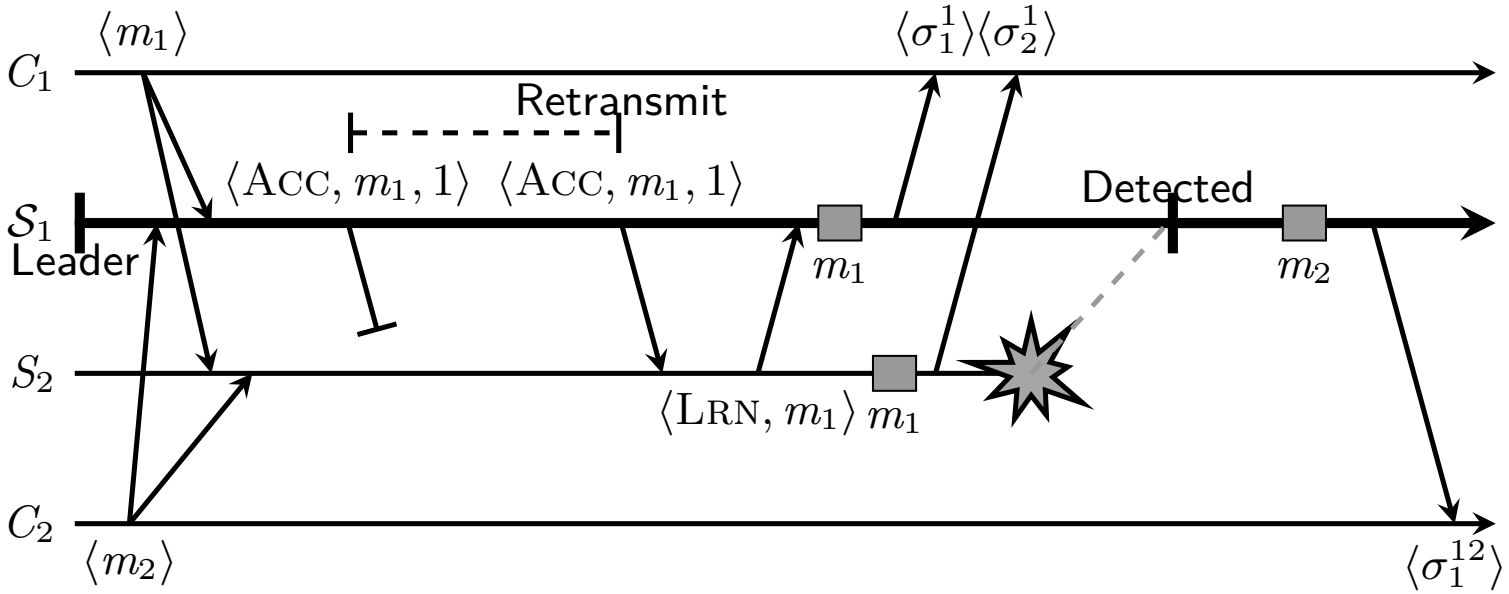
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Leader Remain in Control when S_2 Crash



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Let's Add Client Messages Again



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: The Problem

- When we detect a server crash
  - Adopt the *SingleServer* protocol

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: The Problem

- When we detect a server crash
  - Adopt the *SingleServer* protocol
- Problem with our *RetransAccept* protocol:
  - The leader might have replied to a client and then crashed, without ensuring that S_2 saw the accept
  - S_2 takes over and may execute a different request in *SingleServer* mode

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: WaitForLearn Protocol

- The leader always waits for a learn message from S_2
  - Think of it as an acknowledgement

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: WaitForLearn Protocol

- The leader always waits for a learn message from S_2
  - Think of it as an acknowledgement
- S_2 can execute after seeing an accept from the leader
  - This is because the accept message is also an implicit learn

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: WaitForLearn Protocol

- The leader always waits for a learn message from S_2
  - Think of it as an acknowledgement
- S_2 can execute after seeing an accept from the leader
  - This is because the accept message is also an implicit learn
- Q: What happens if the learn message to the leader is lost?

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: WaitForLearn Protocol

- The leader always waits for a learn message from S_2
  - Think of it as an acknowledgement
- S_2 can execute after seeing an accept from the leader
  - This is because the accept message is also an implicit learn
- Q: What happens if the learn message to the leader is lost?
- A: The leader uses *RetransAccept*; the accept will be retransmitted. So no need for another retransmit protocol.

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Somewhat Rougher Road Ahead!

Fonte: Hein Meling e Leander Jehl - OPODIS2013

False Detection

- So far we have assumed that failure detection is accurate

Fonte: Hein Meling e Leander Jehl - OPODIS2013

False Detection

- So far we have assumed that failure detection is accurate
- But in an asynchronous environment
  - There is always a chance of false detection
  - Because it is impossible to pick the right timeout delay

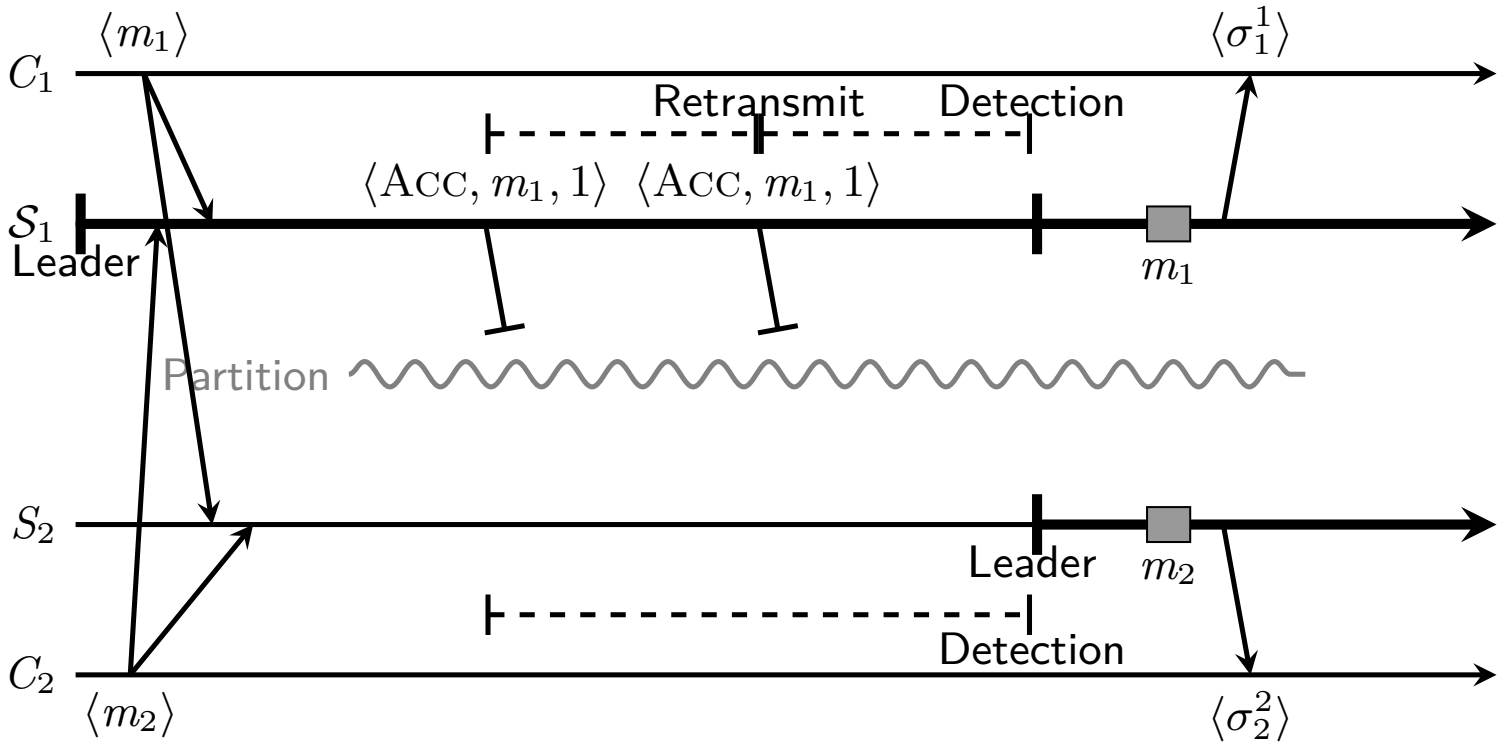
Fonte: Hein Meling e Leander Jehl - OPODIS2013

False Detection

- So far we have assumed that failure detection is accurate
- But in an asynchronous environment
  - There is always a chance of false detection
  - Because it is impossible to pick the right timeout delay
- We now consider false detection in the context of network partitions

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Problem: Network Partitions



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Network Partition

- Each server can switch to *SingleServer* mode (no coordination) and make progress

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Network Partition

- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Network Partition

- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2
- Reconciling the state divergence
  - Involves rollback on multiple clients

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Network Partition

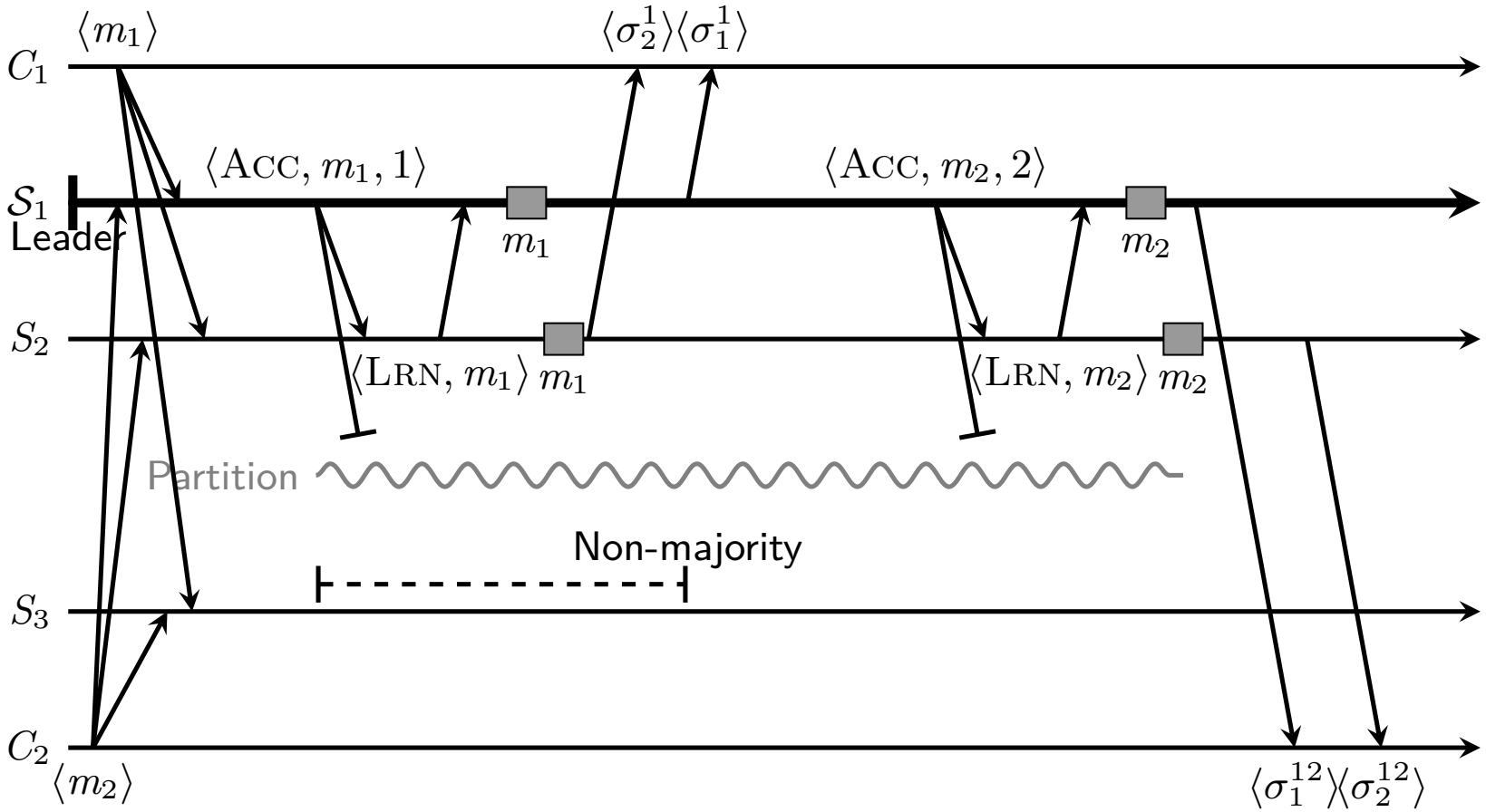
- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2
- Reconciling the state divergence
  - Involves rollback on multiple clients
  - Quickly becomes unmanageable

Fonte: Hein Meling e Leander Jehl - OPODIS2013

We Want to Avoid Relying on Clients!

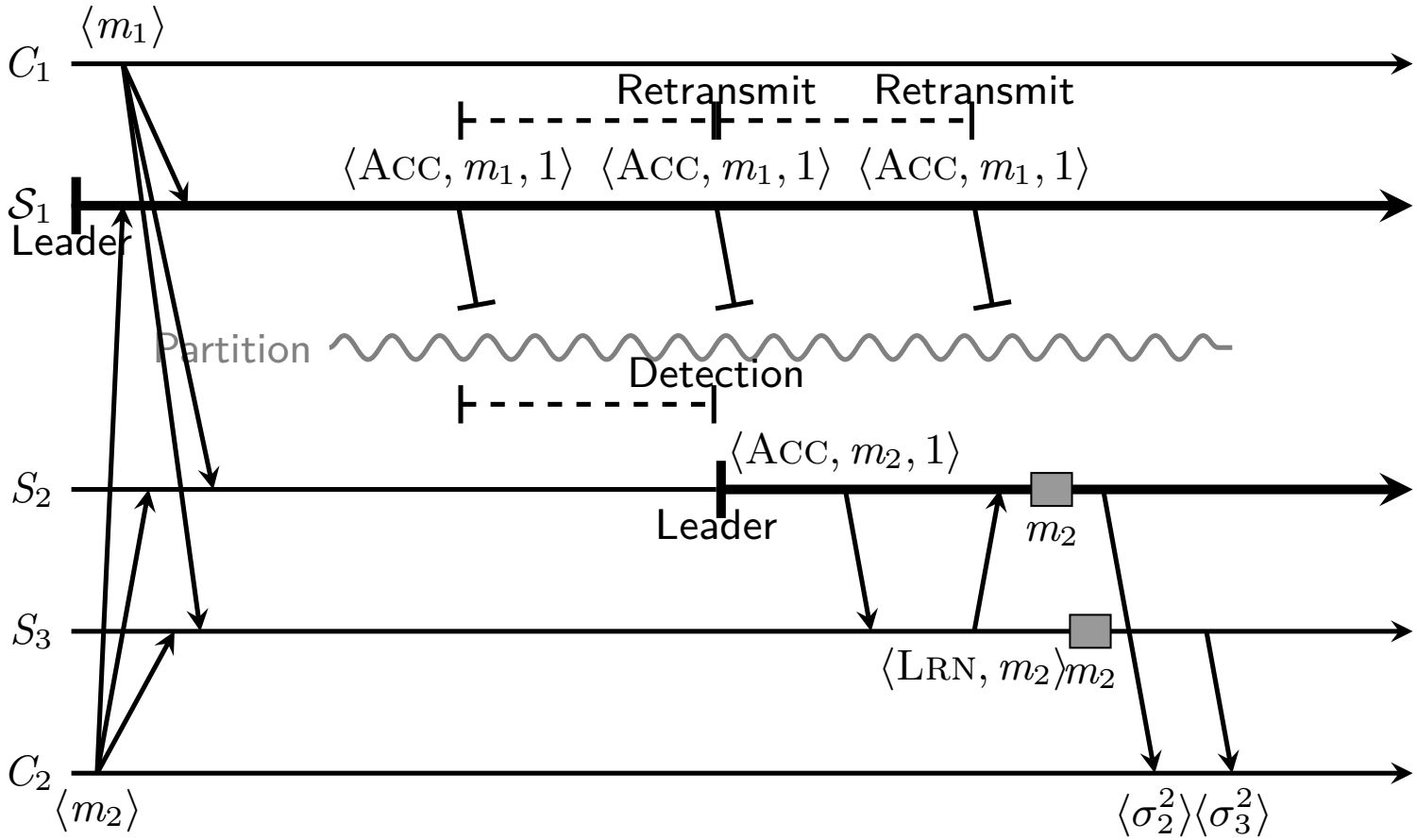
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Add Another Server; Make Progress in Majority Partition



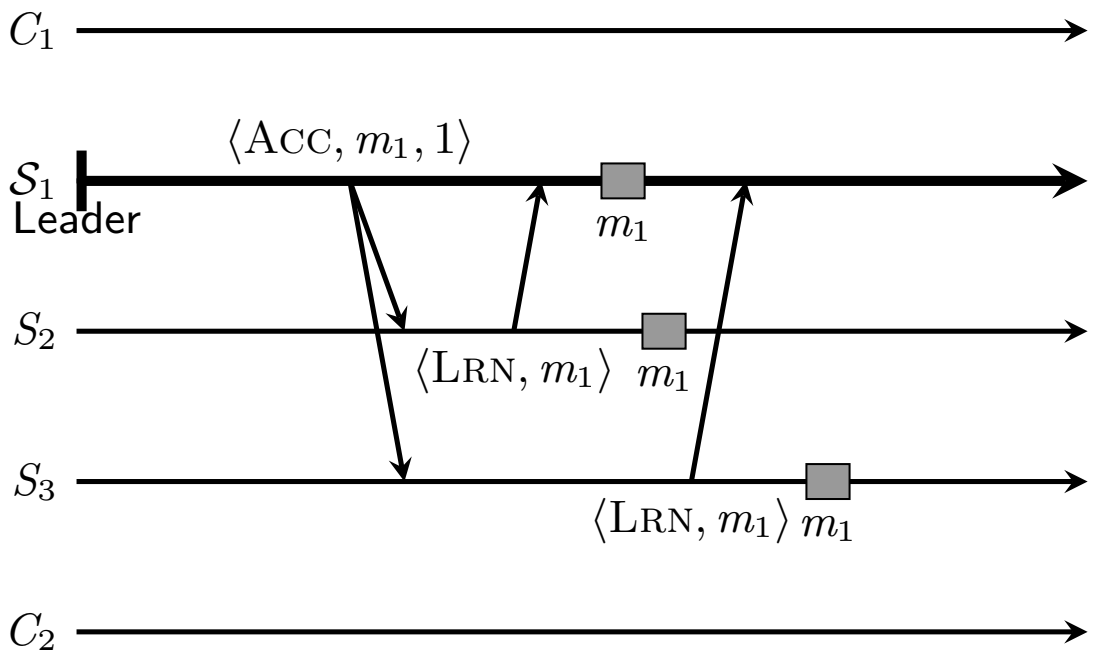
Fonte: Hein Meling e Leander Jehl - OPODIS2013

New Leader in Majority Partition



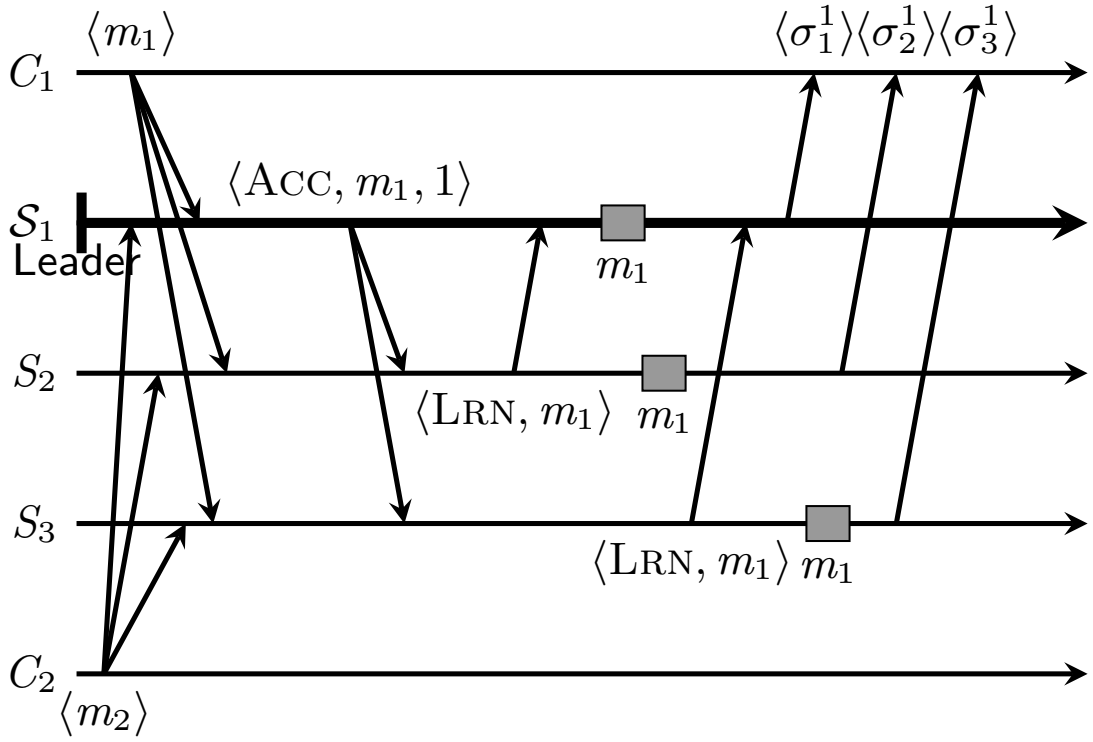
Fonte: Hein Meling e Leander Jehl - OPODIS2013

WaitForLearn Without Partition



Fonte: Hein Meling e Leander Jehl - OPODIS2013

WaitForLearn With Clients



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Network Partition

- We added another server, S_3
  - To avoid rollback using clients

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Network Partition

- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Network Partition

- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
  - when another server has crashed or is temporarily unavailable

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Network Partition

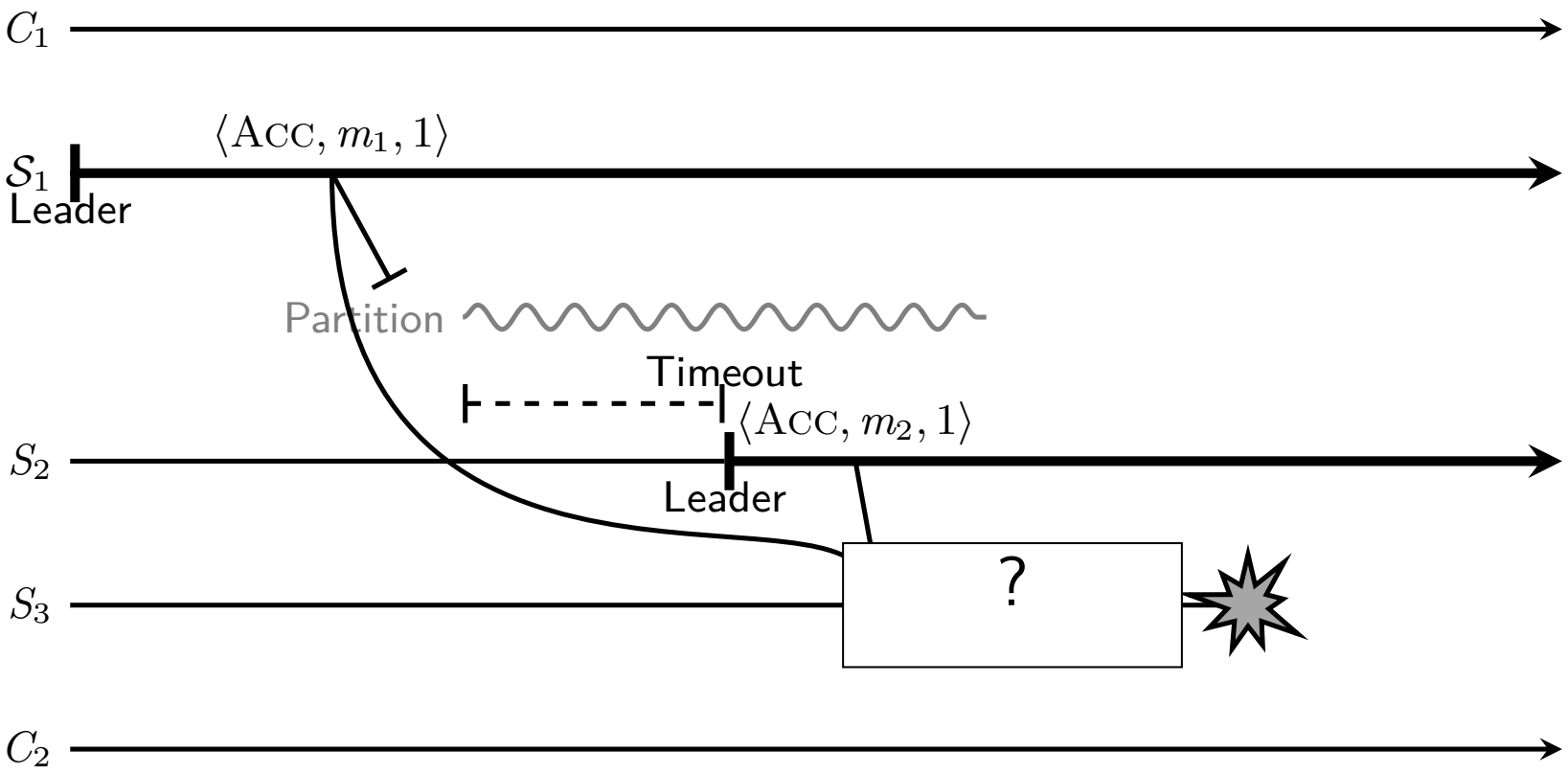
- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
  - when another server has crashed or is temporarily unavailable
- But we still only tolerate one concurrent failure
  - Either a crash or a network partition

Fonte: Hein Meling e Leander Jehl - OPODIS2013

What can go Wrong: Concurrent Crash and Partition

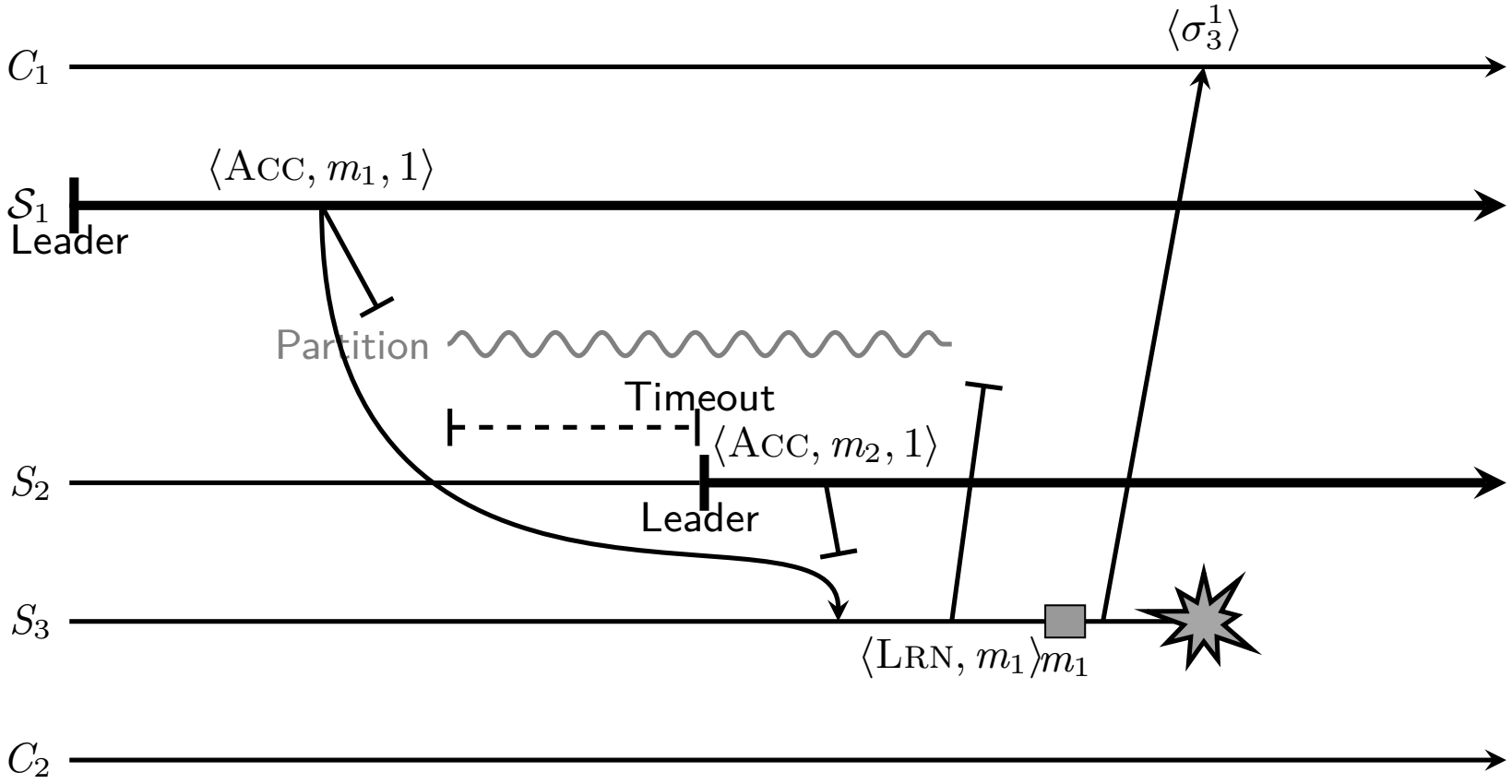
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Concurrent Crash and Partition



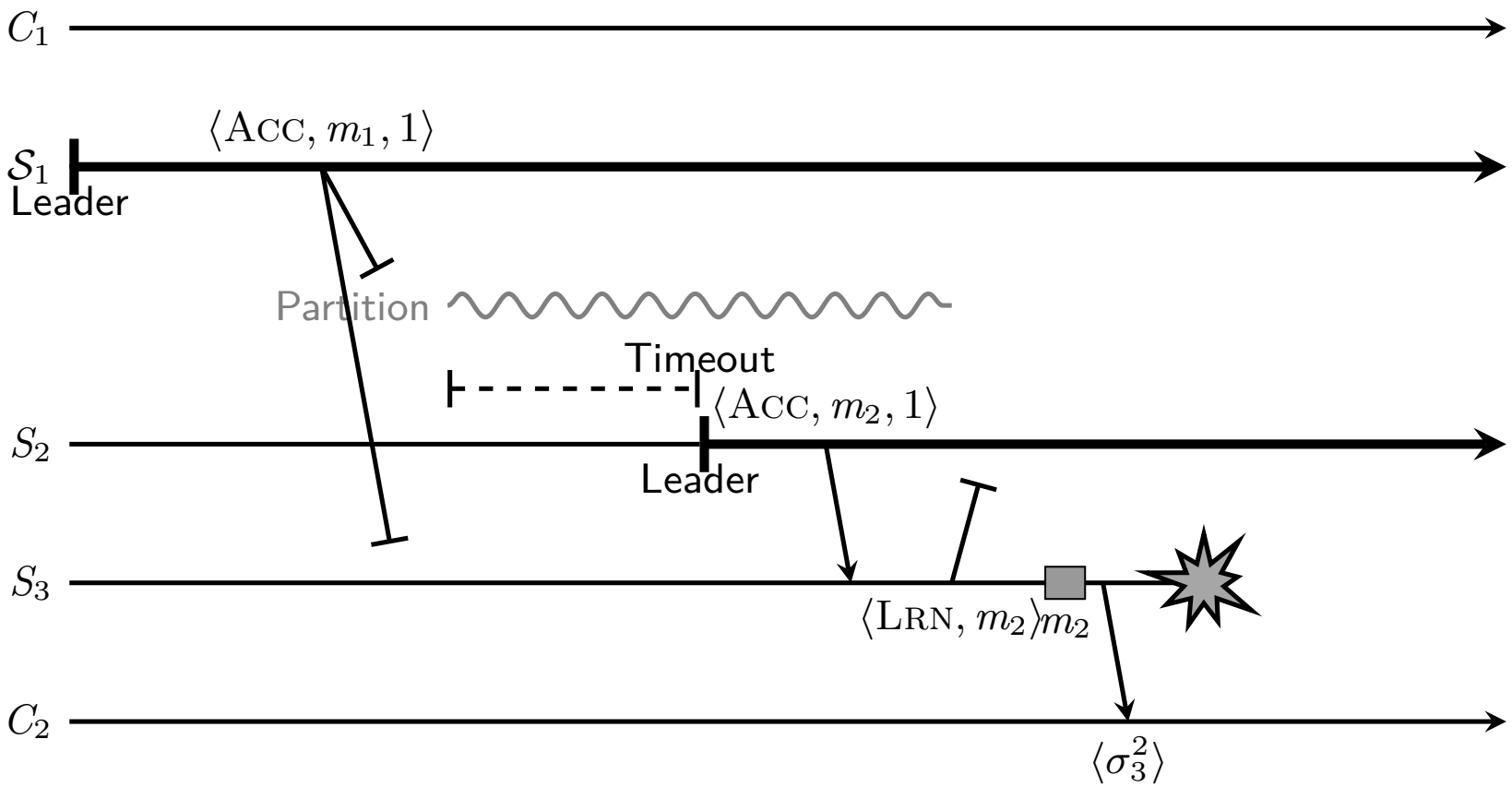
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Crash and Partition: Outcome 1 – m_1 Executed



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Crash and Partition: Outcome 2 – m_2 Executed



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Crash and Partition

- S_3 crashed
  - But *it could* have executed either m_1 or m_2
  - And replied to a client

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Crash and Partition

- S_3 crashed
  - But *it could* have executed either m_1 or m_2
  - And replied to a client
- Other servers cannot determine which message, if any, was executed

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Crash and Partition

- S_3 crashed
  - But *it could* have executed either m_1 or m_2
  - And replied to a client
- Other servers cannot determine which message, if any, was executed
  - Maybe we could talk to clients?
  - We don't want to rely on clients!

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Explicit Leader Change Mechanism

- Above problem is rooted in possibility of false detection
  - Can lead to several servers thinking they are leaders
  - And sending accept messages concurrently

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Explicit Leader Change Mechanism

- Above problem is rooted in possibility of false detection
  - Can lead to several servers thinking they are leaders
  - And sending accept messages concurrently
- It can be solved by an explicit leader takeover protocol

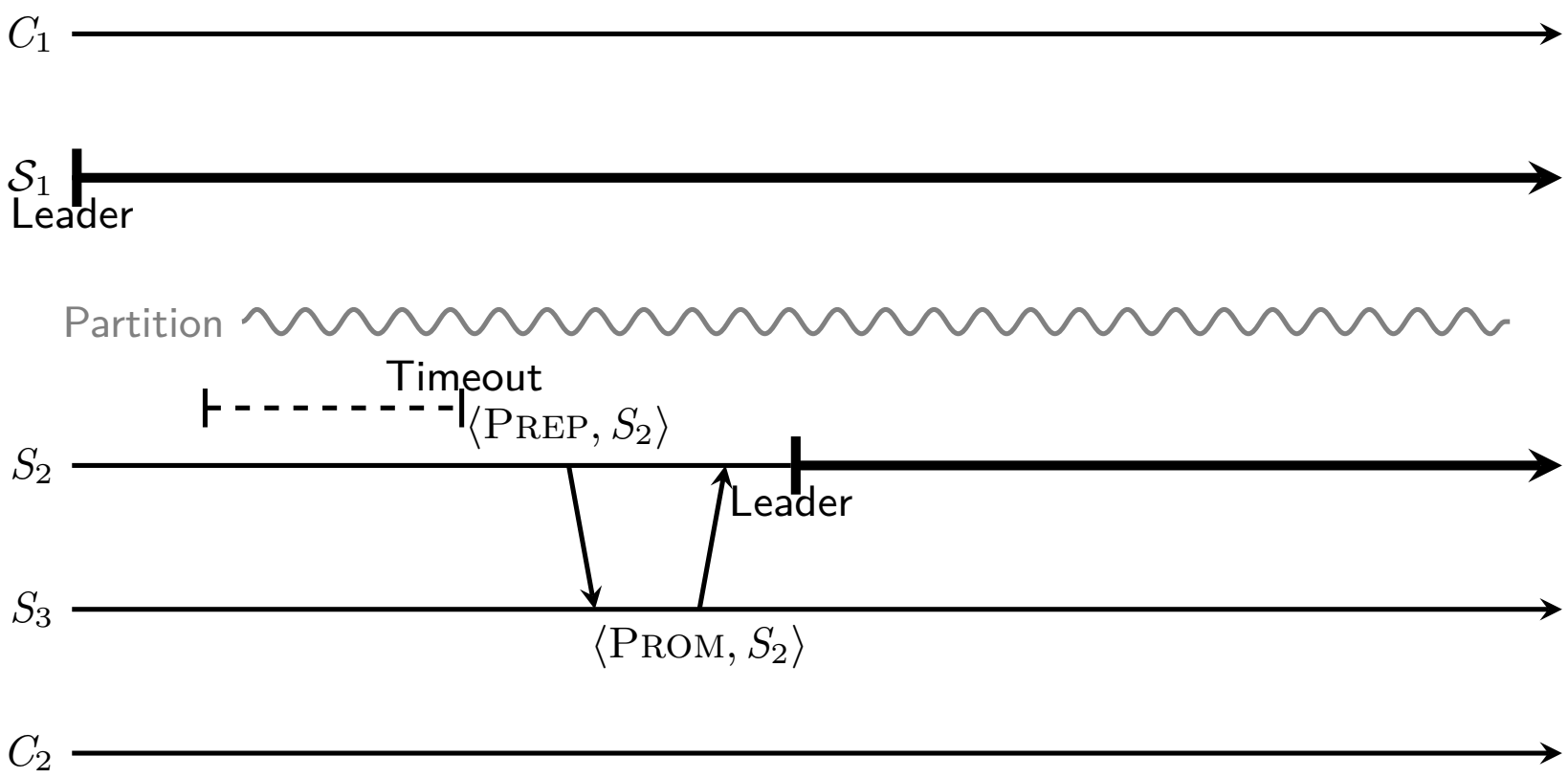
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Explicit Leader Change Mechanism

- Above problem is rooted in possibility of false detection
  - Can lead to several servers thinking they are leaders
  - And sending accept messages concurrently
- It can be solved by an explicit leader takeover protocol
- We need a way to
  - Distinguish messages from different leaders
  - Change the leader

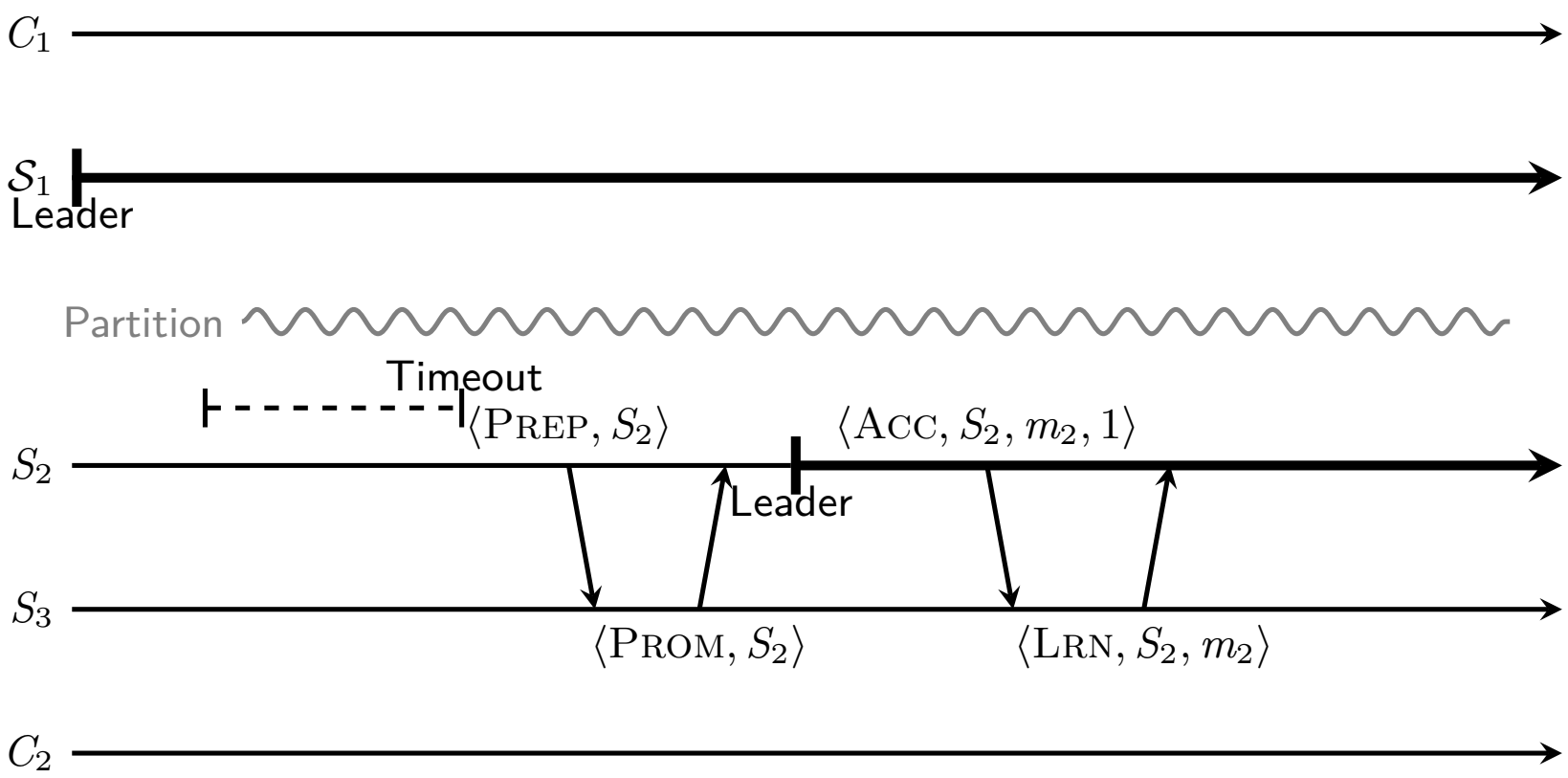
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Explicit Leader Change



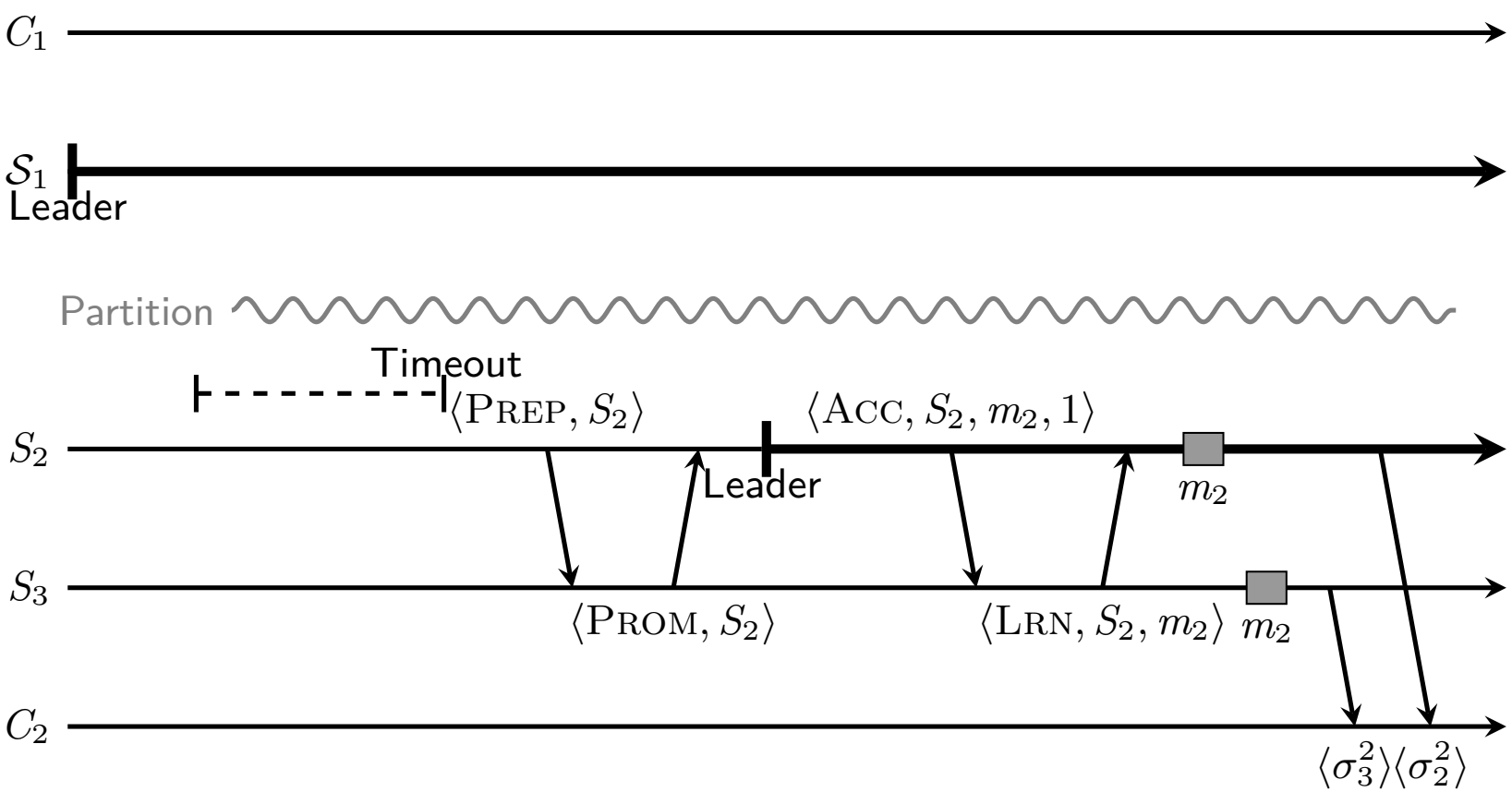
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Leader Identifiers in Accept and Learn Messages



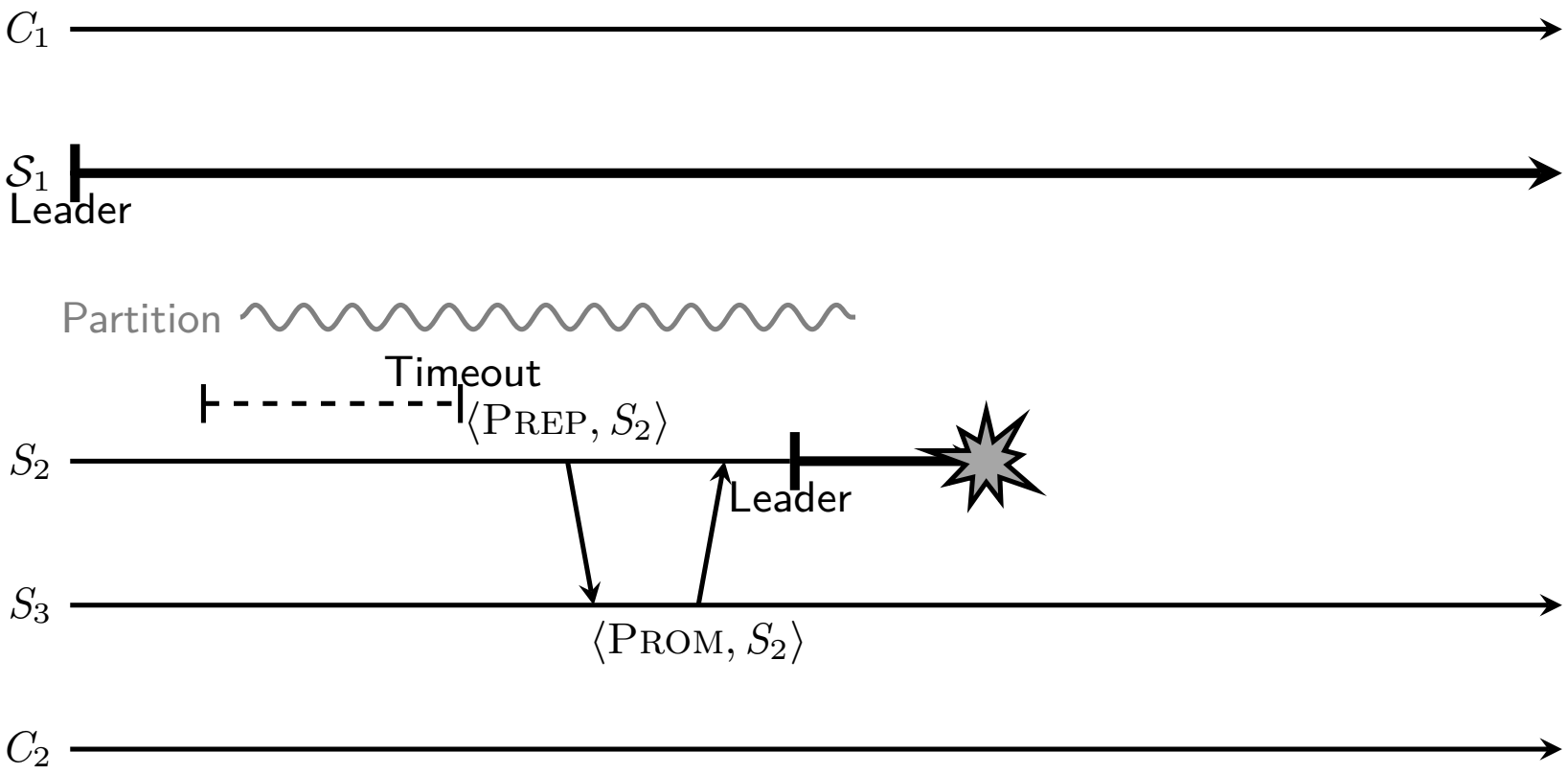
Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Client Replies



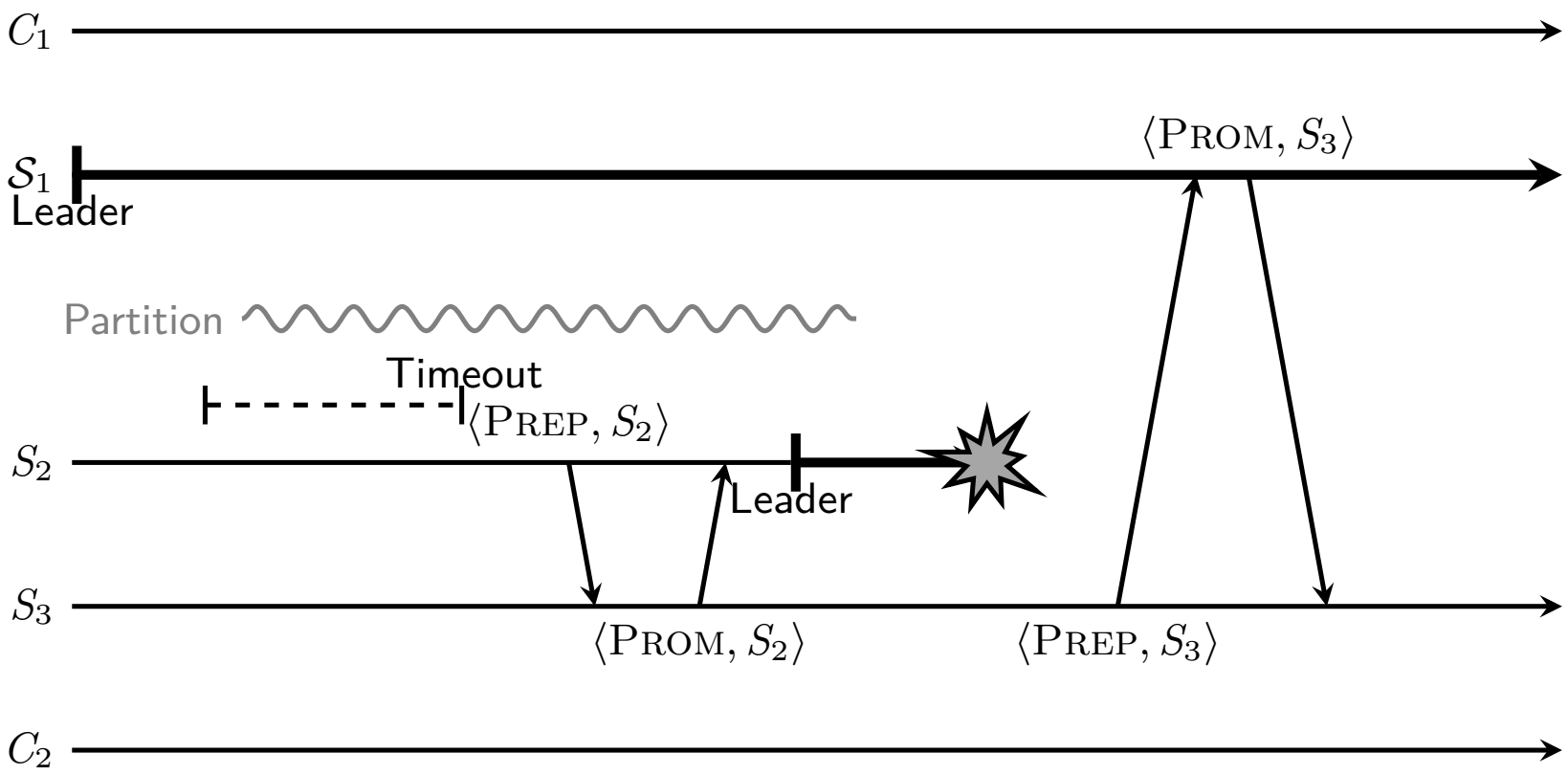
Fonte: Hein Meling e Leander Jehl - OPODIS2013

What Happens Now?



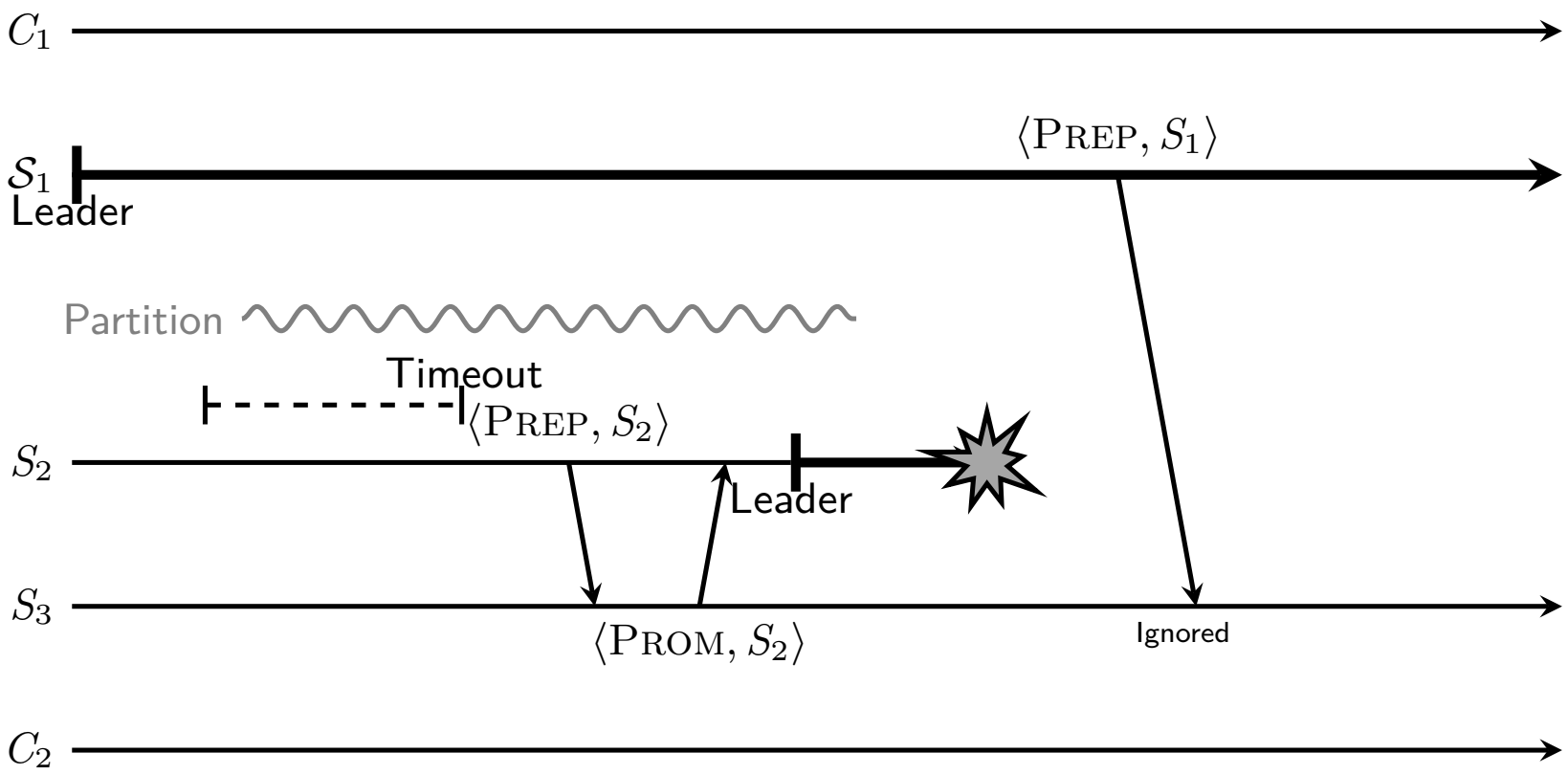
Fonte: Hein Meling e Leander Jehl - OPODIS2013

S_3 Takes Over?



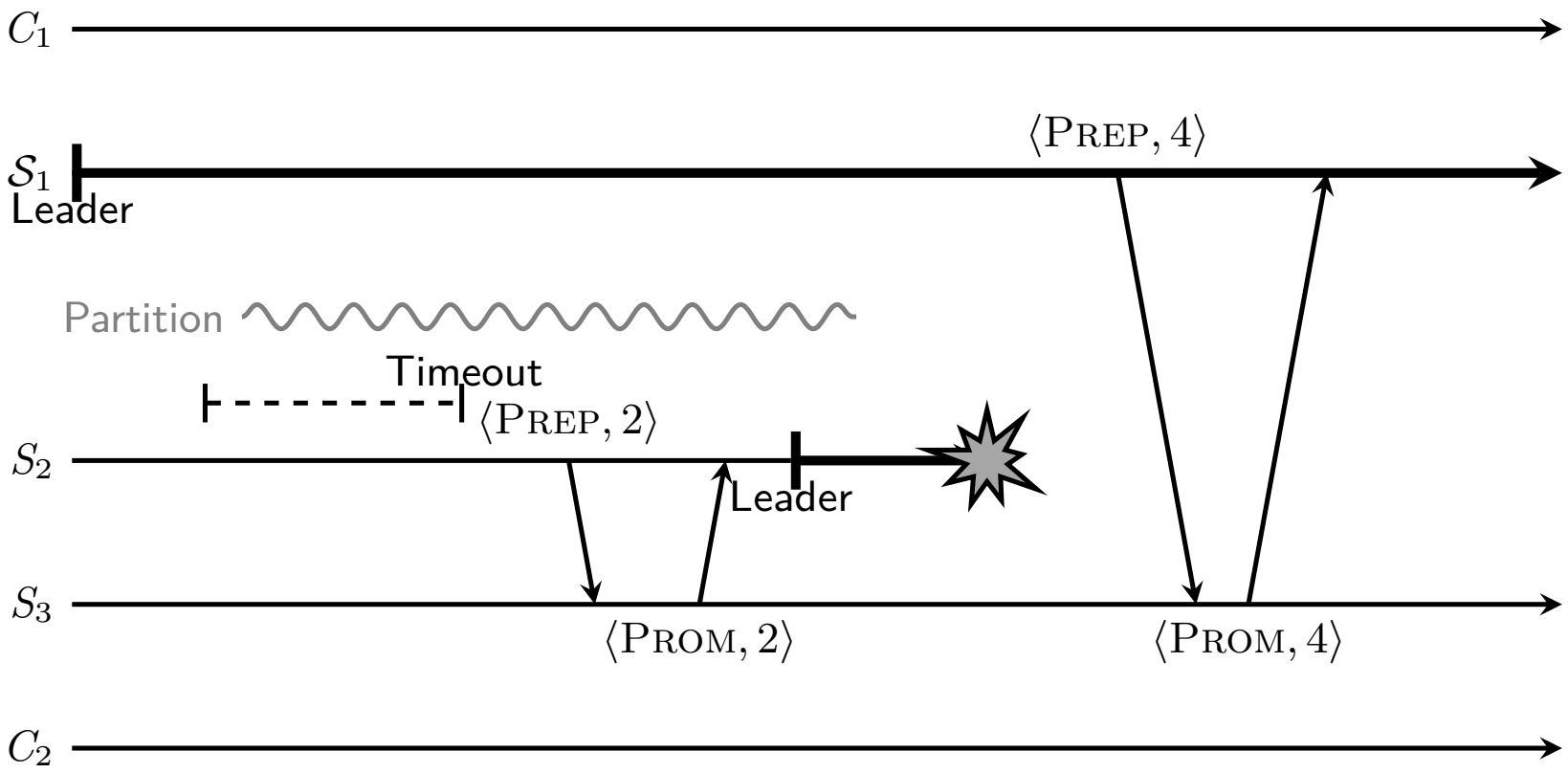
Fonte: Hein Meling e Leander Jehl - OPODIS2013

S_1 Takes Over Again?



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Replace Leader Identifiers With Round Numbers



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Leader Change

- Added round number rnd in messages
  - To identify the leader
    - $\langle \text{ACC}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \text{LRN}, rnd, m \rangle$: Sent to leader of round rnd

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Leader Change

- Added round number rnd in messages
  - To identify the leader
    - $\langle \text{ACC}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \text{LRN}, rnd, m \rangle$: Sent to leader of round rnd
  - Round numbers are assigned:
    - S_1: $1, 4, 7, \ldots$
    - S_2: $2, 5, 8, \ldots$
    - S_3: $3, 6, 9, \ldots$
  - Skipping rounds is possible

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Leader Change

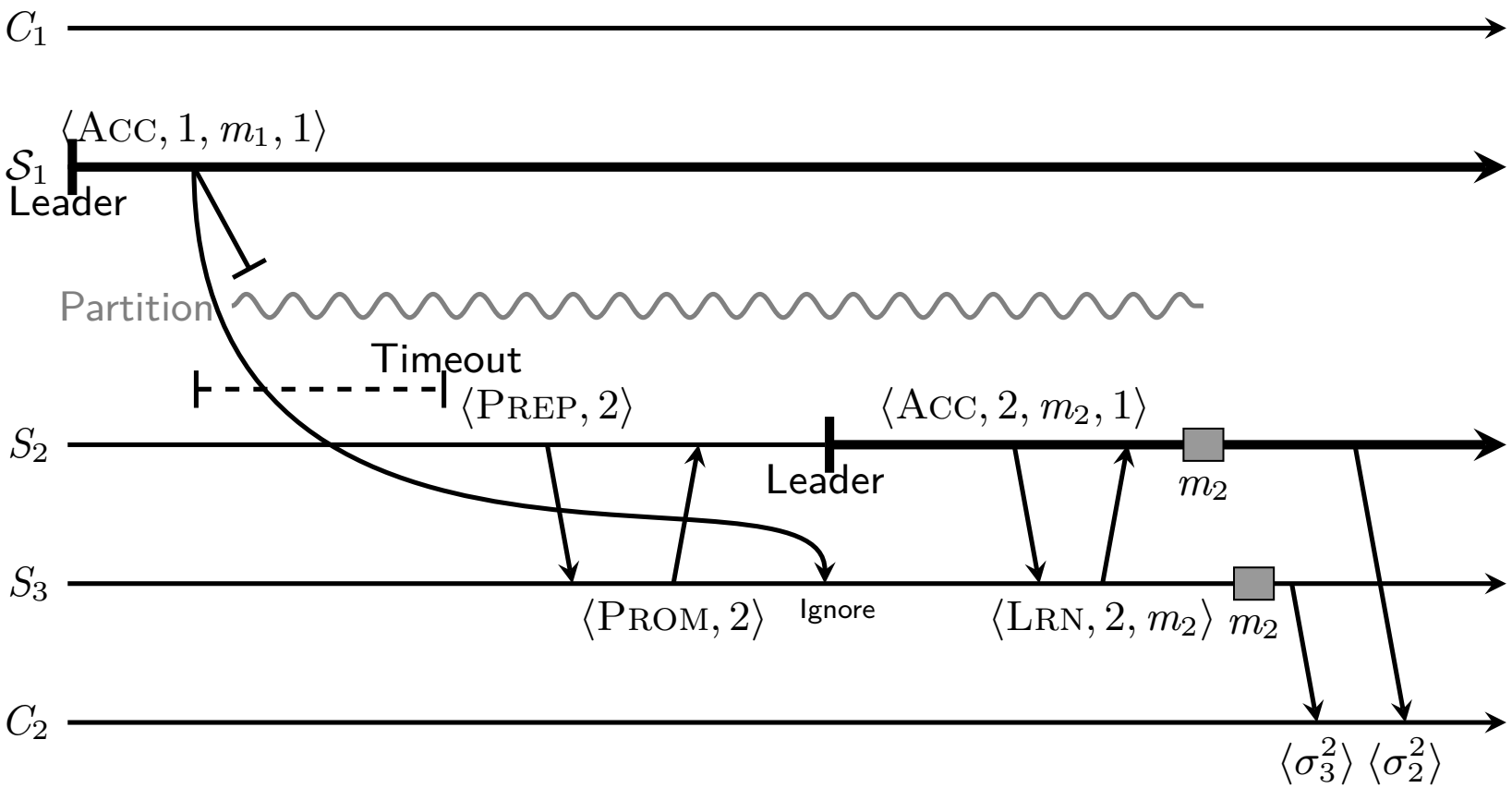
- Added round number rnd in messages
  - To identify the leader
    - $\langle \text{ACC}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \text{LRN}, rnd, m \rangle$: Sent to leader of round rnd
  - Round numbers are assigned:
    - S_1: $1, 4, 7, \ldots$
    - S_2: $2, 5, 8, \ldots$
    - S_3: $3, 6, 9, \ldots$
  - Skipping rounds is possible
- Added two new messages
  - $\langle \text{PREP}, rnd \rangle$: Request to become leader for round rnd
  - $\langle \text{PROM}, rnd \rangle$: Promise not to accept messages from a lower round than rnd (i.e. an older leader)

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Let's Apply This Together With Accept and Learn

Fonte: Hein Meling e Leander Jehl - OPODIS2013

S_3 Ignores Accept Message From Old Leader

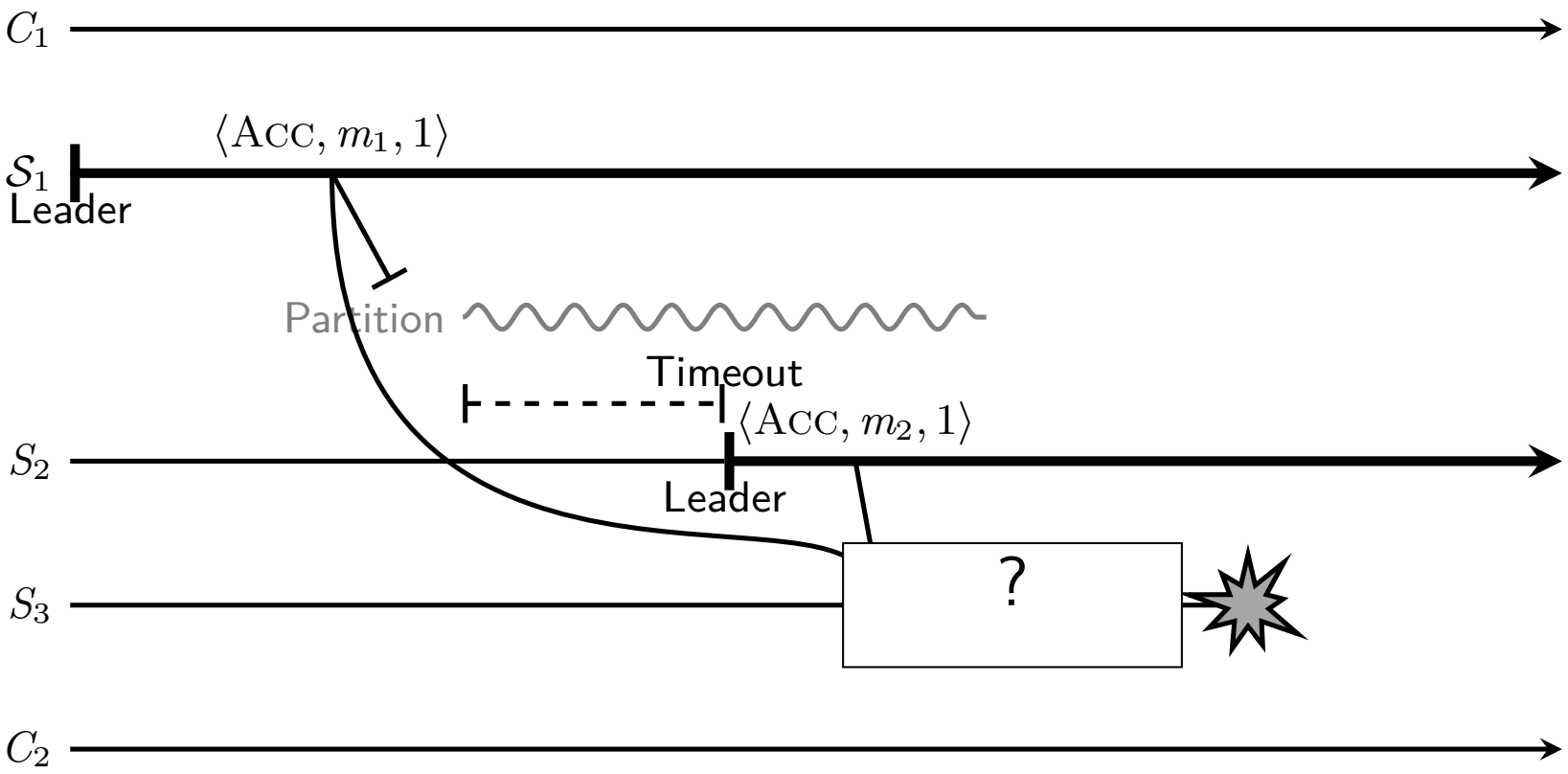


Fonte: Hein Meling e Leander Jehl - OPODIS2013

Let's Recall the Problem we are Trying to Solve

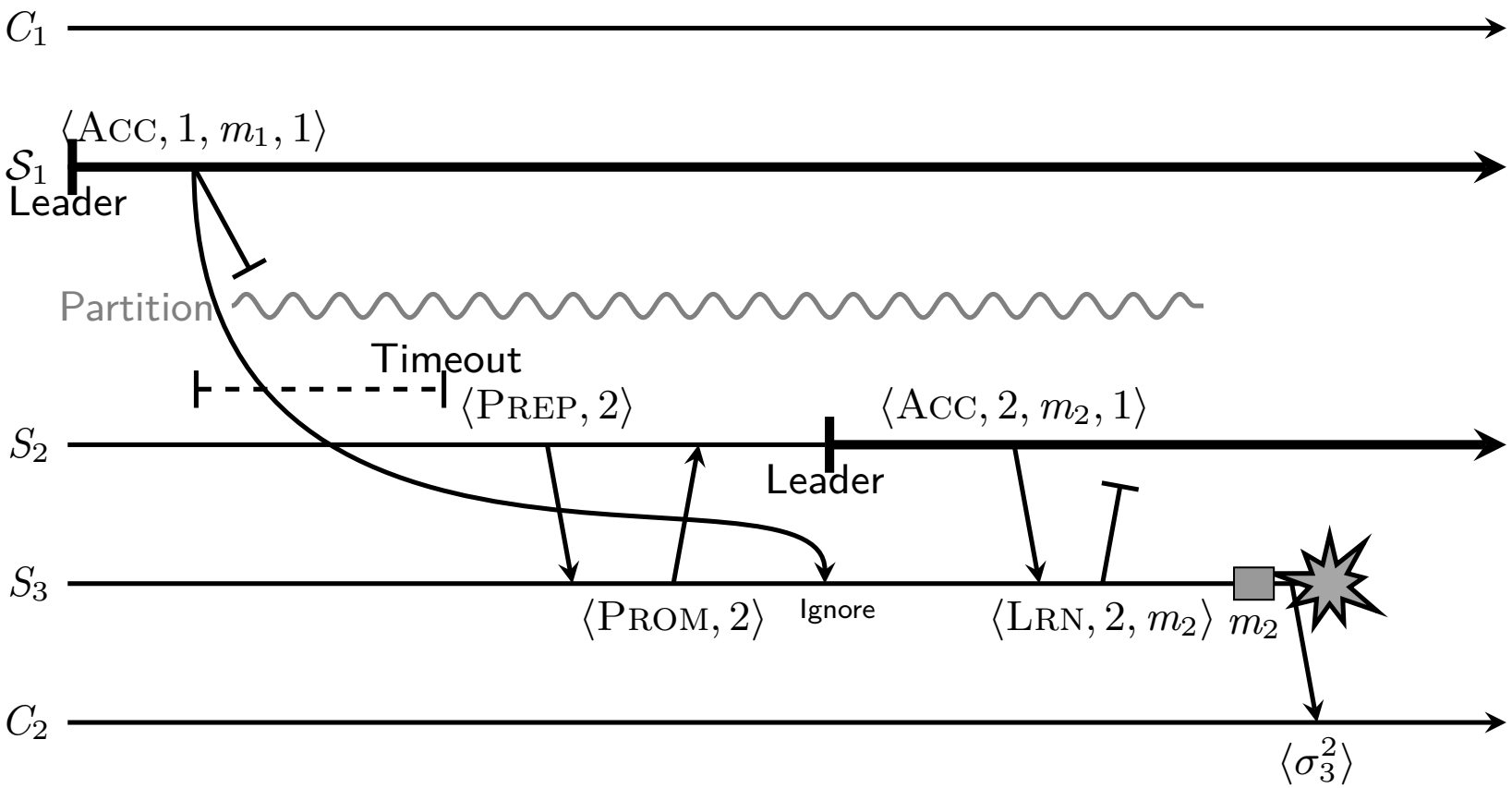
Fonte: Hein Meling e Leander Jehl - OPODIS2013

We Don't Know What S_3 Did Before Crashing



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Do We Know Now?



Fonte: Hein Meling e Leander Jehl - OPODIS2013

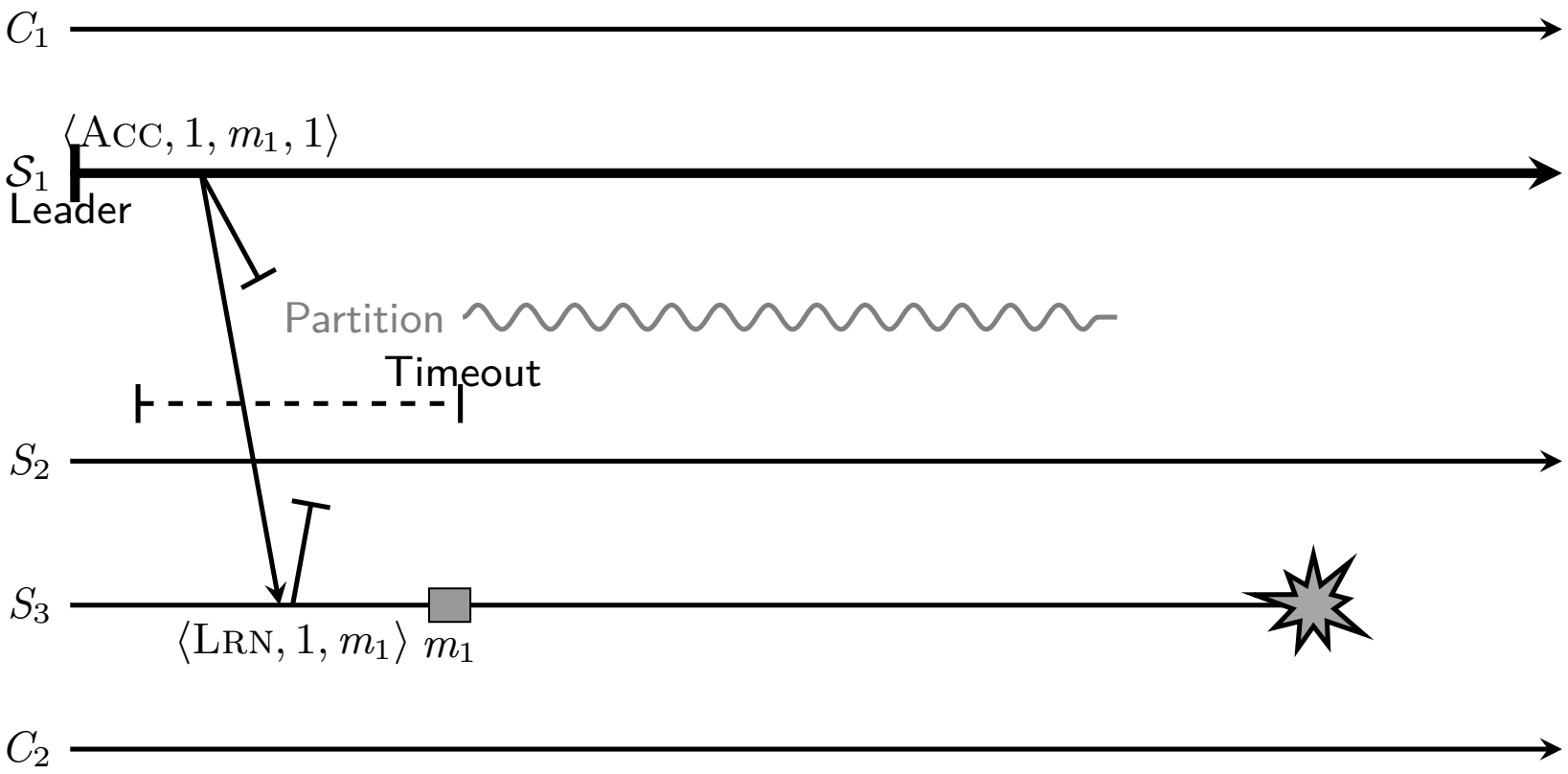
No we don't!

Fonte: Hein Meling e Leander Jehl - OPODIS2013

But it is Safe to Continue as If m_2 Had Been Executed

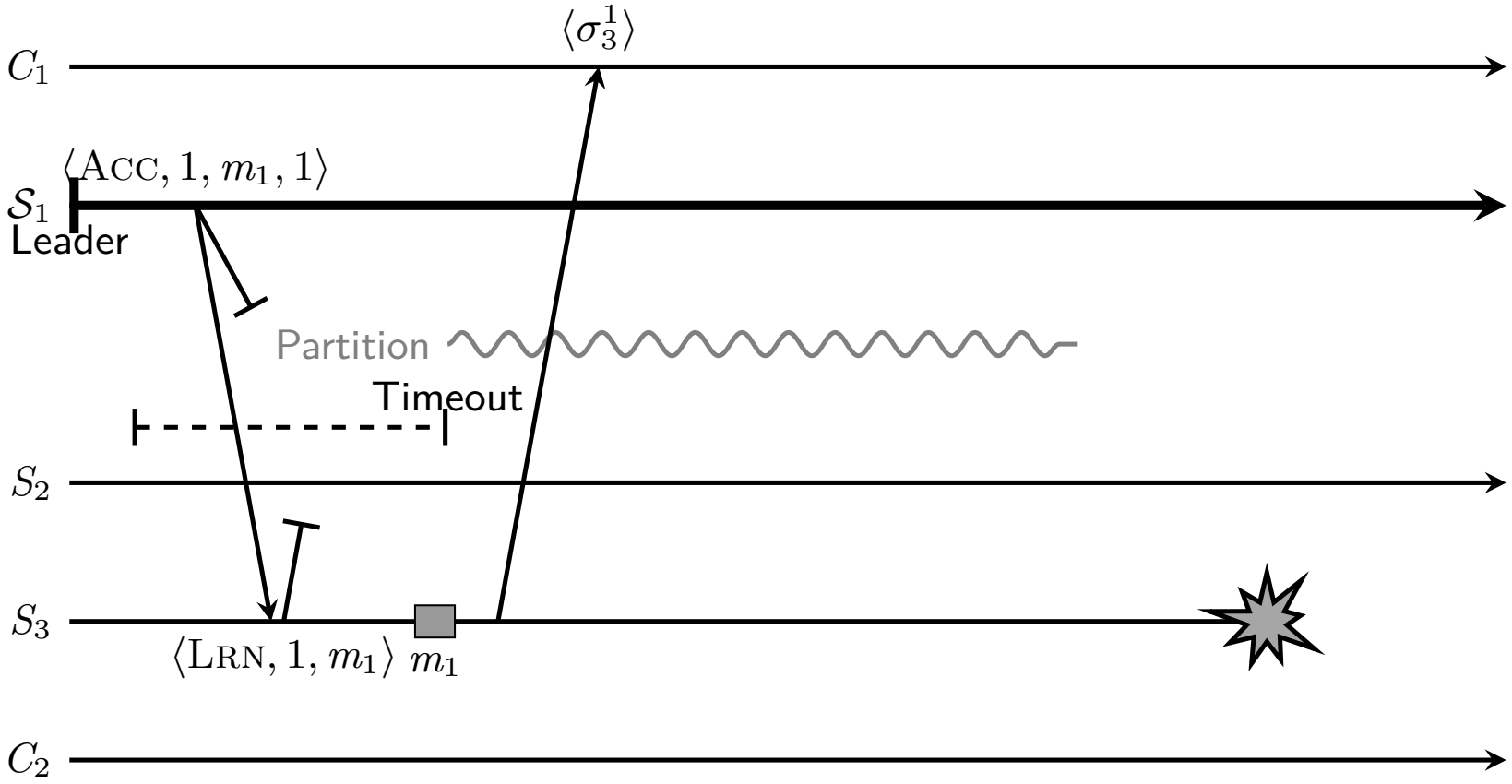
Fonte: Hein Meling e Leander Jehl - OPODIS2013

What Happens If S_3 Learn m_1?



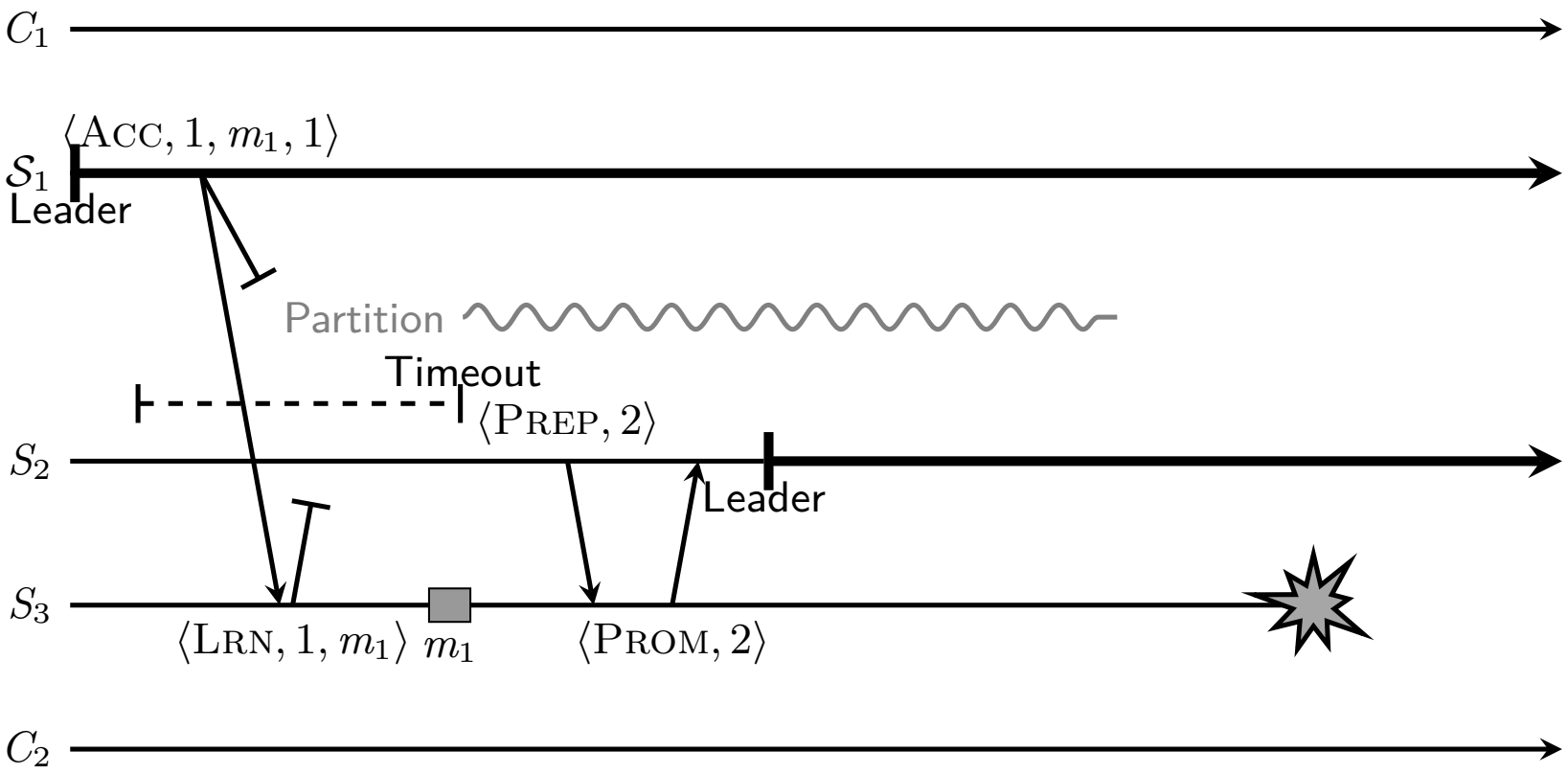
Fonte: Hein Meling e Leander Jehl - OPODIS2013

What Happens If S_3 Learn m_1?



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Does Leader Change Help?



Fonte: Hein Meling e Leander Jehl - OPODIS2013

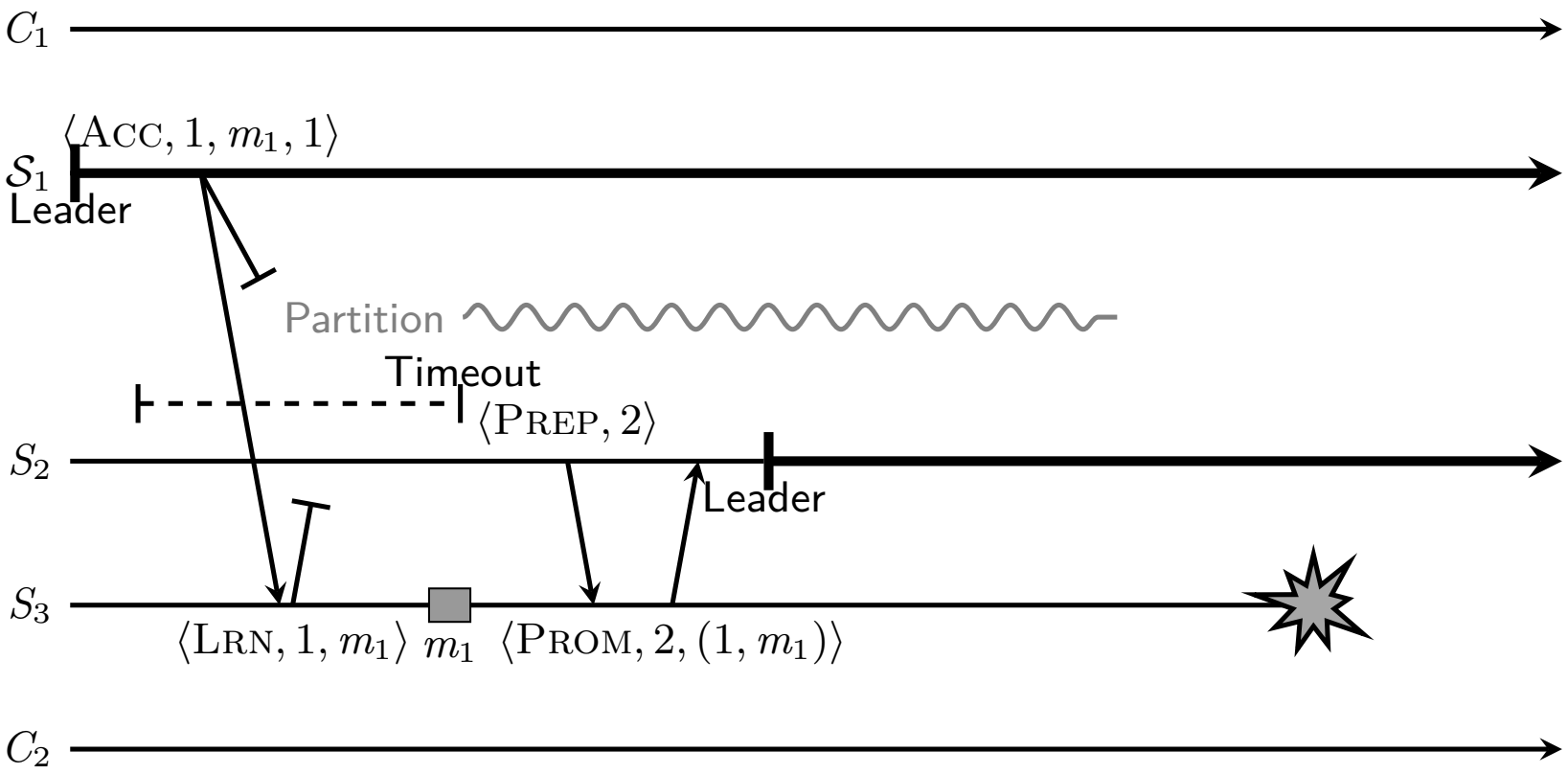
No!
We Still don't Know What
S_3 Did Before Crashing.

Fonte: Hein Meling e Leander Jehl - OPODIS2013

But the fix is Easy!

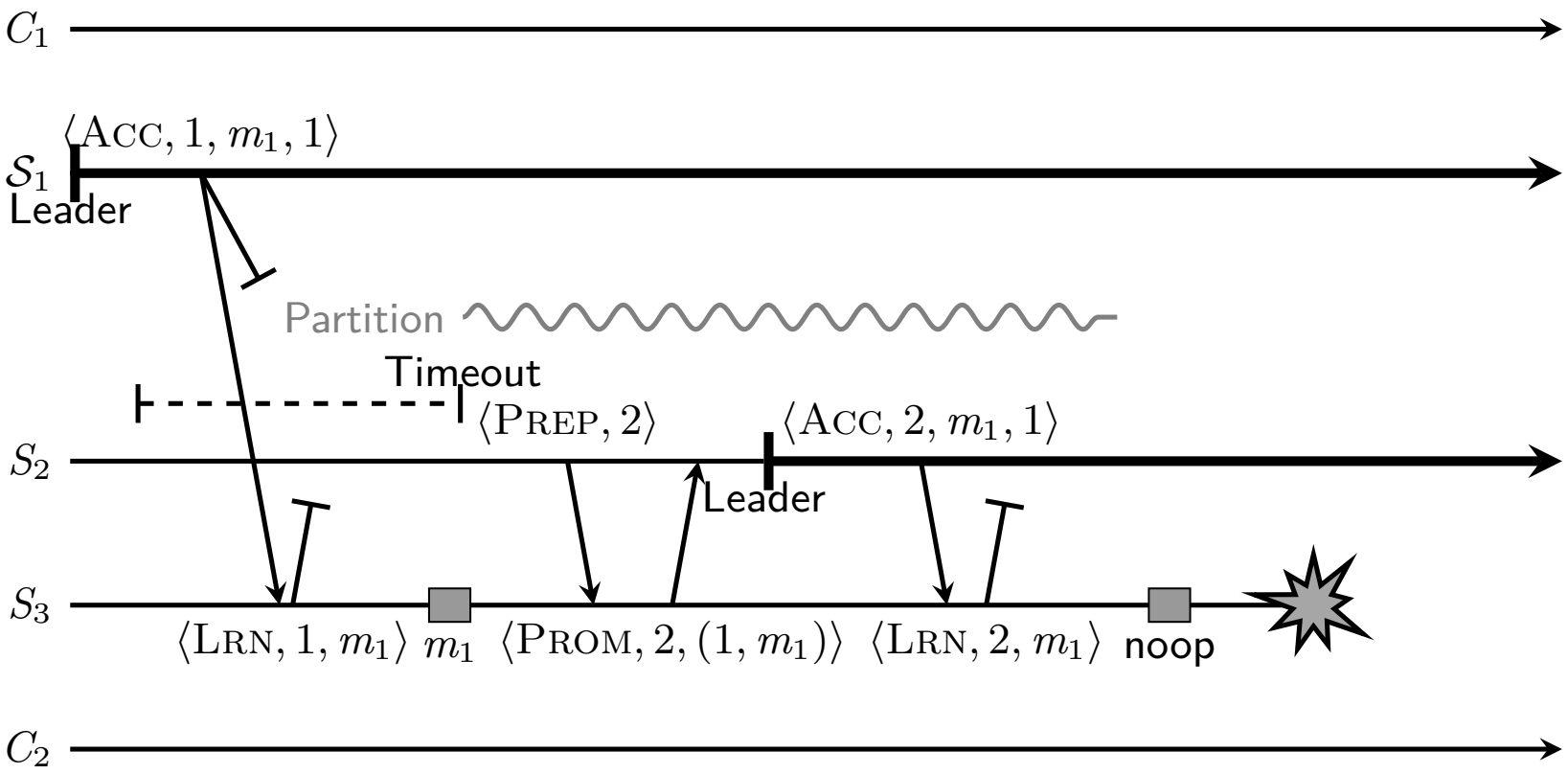
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Tell new Leader About Accepted Messages



Fonte: Hein Meling e Leander Jehl - OPODIS2013

The new Leader Resends Accept for Those Messages

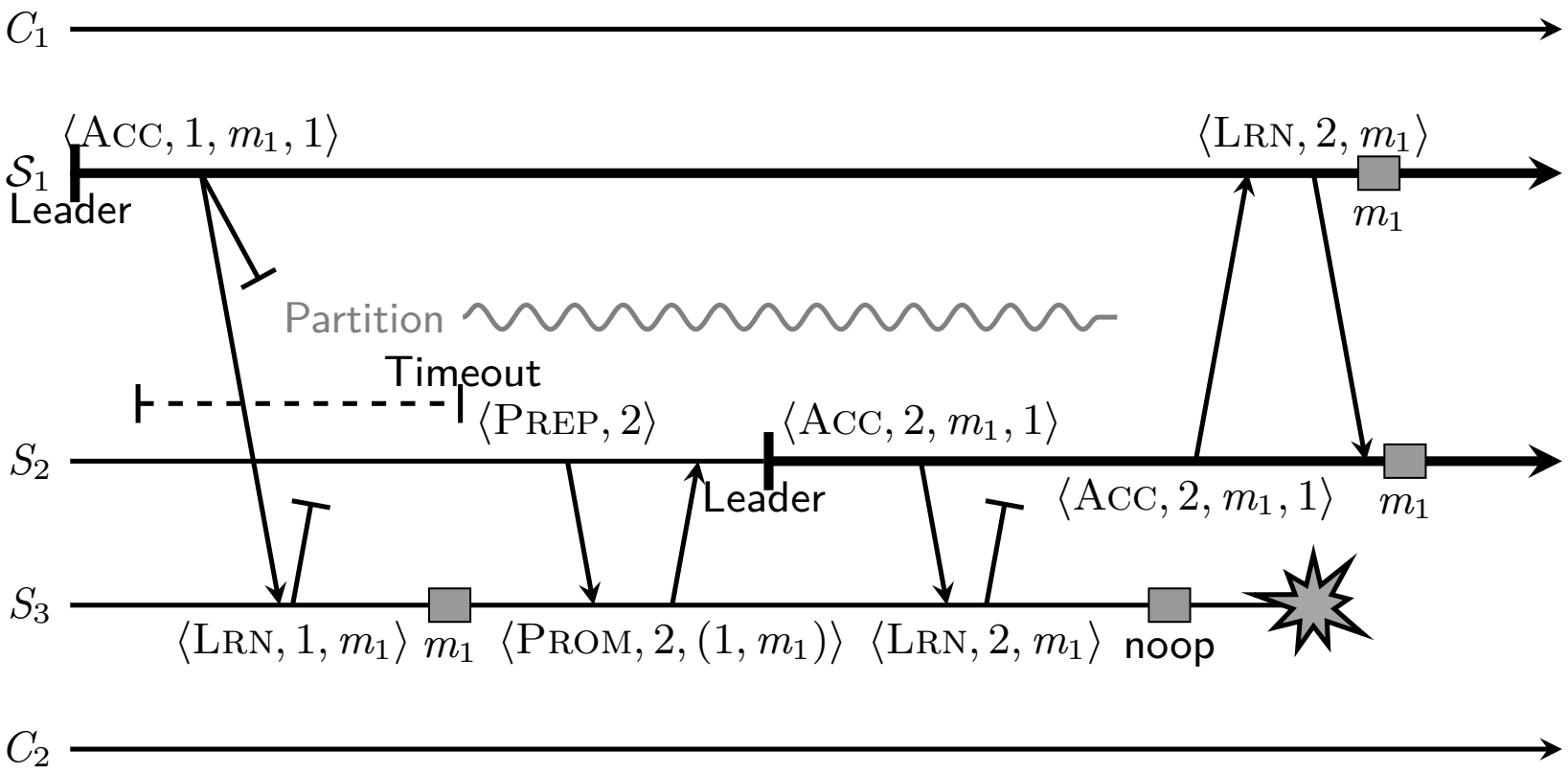


Fonte: Hein Meling e Leander Jehl - OPODIS2013

Learn was Lost and S_3 Crashed.
Leader Still can't Execute m_1.

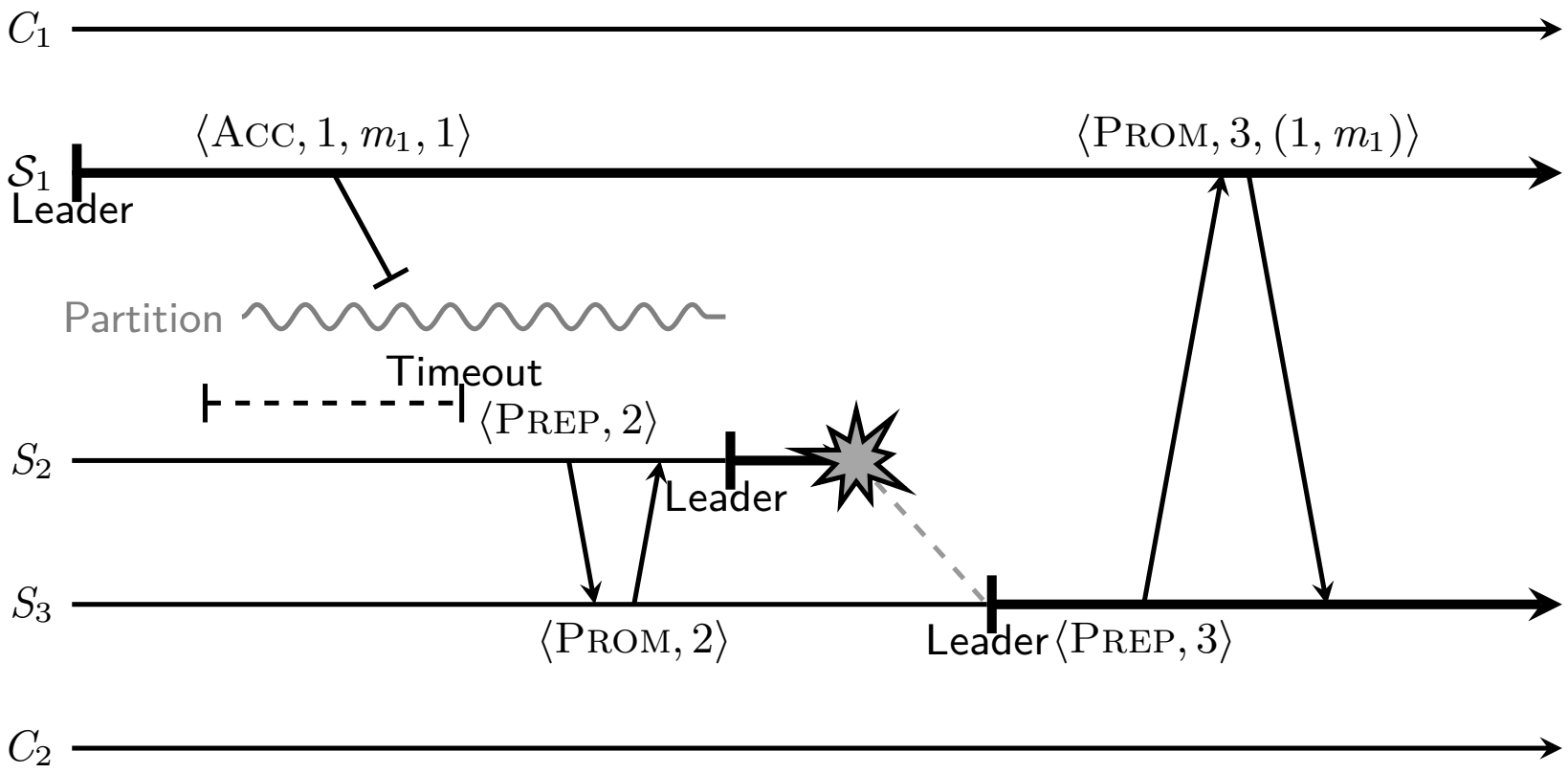
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Leader Also Resends Accept After Merge



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Promise from old Leader Includes Accepted Messages



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Leader Change 2

- Added information about accept from previous leader: $\langle \textsc{Prom}, rnd, (1, m_1) \rangle$
  - Promise not to accept messages from a lower round than rnd
  - Last leader did send m_1 in round 1
  - Typical naming: $\langle \textsc{Prom}, rnd, (vrnd, vval) \rangle$

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Recap: Leader Change 2

- Added information about accept from previous leader: $\langle \text{PROM}, rnd, (1, m_1) \rangle$
  - Promise not to accept messages from a lower round than rnd
  - Last leader did send m_1 in round 1
  - Typical naming: $\langle \text{PROM}, rnd, (vrnd, vval) \rangle$
- Leader resends accept for messages identified in the promise message
  - After receiving the promise
  - After a partition merge

Fonte: Hein Meling e Leander Jehl - OPODIS2013

What About More Than one Crash?

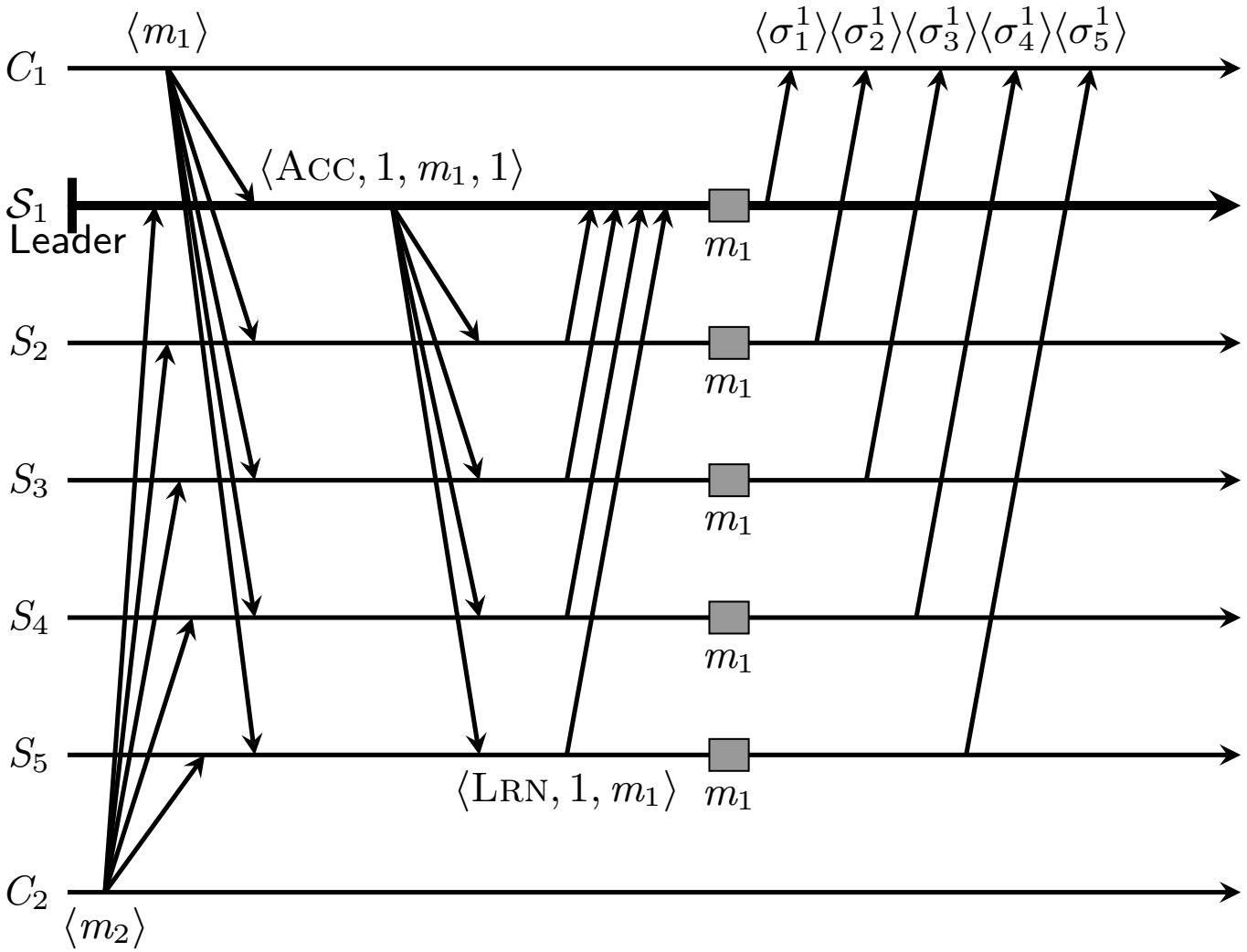
Fonte: Hein Meling e Leander Jehl - OPODIS2013

What About More Than one Crash?

- Increase the number of servers
- To limit progress to a majority partition:
  - We can only tolerate fewer than half of the servers fail
  - To tolerate f crashes, we need at least $2f + 1$

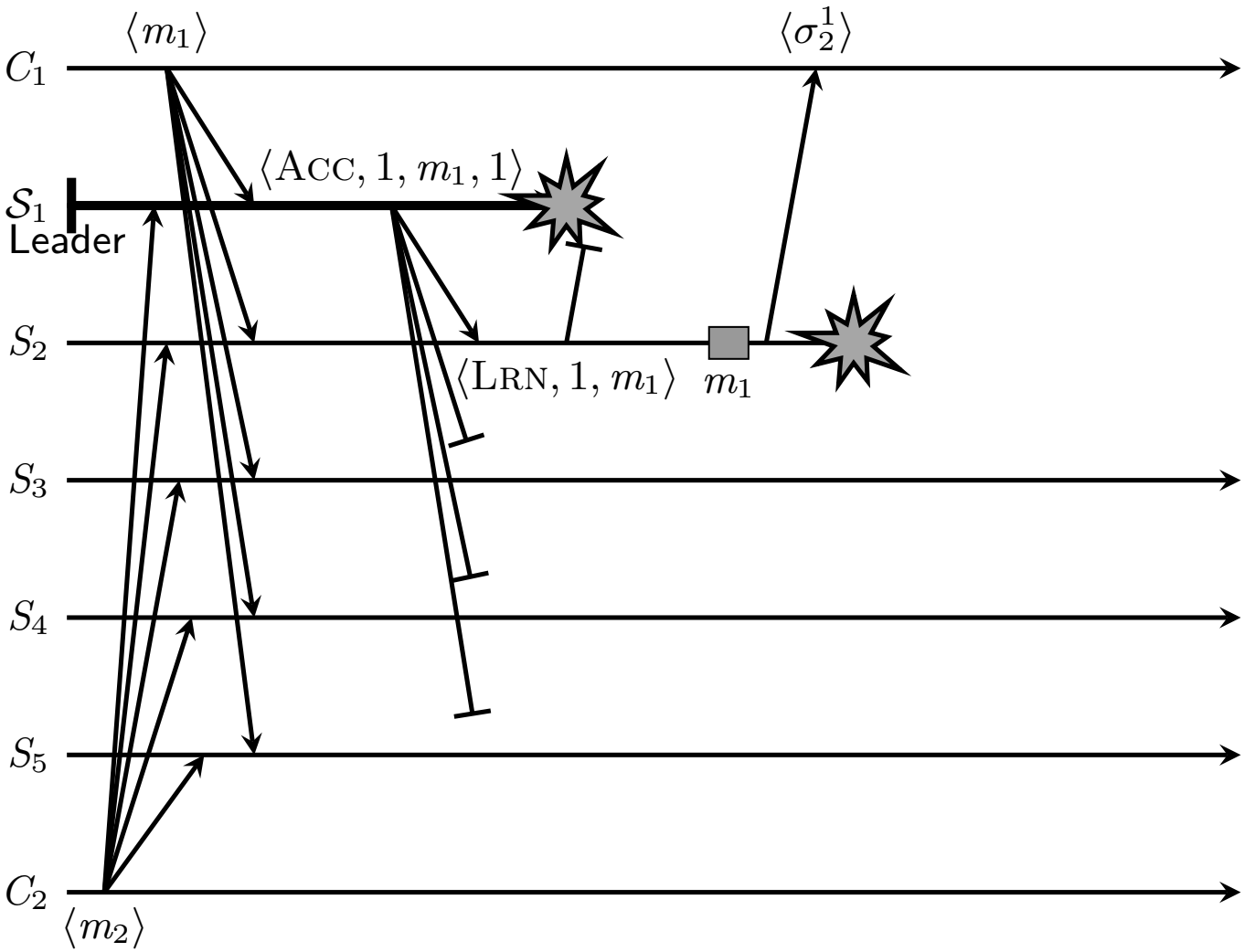
Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Five Servers



Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Five Servers, S_2 Cannot Execute After Accept



Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Five Servers, S_2 Cannot Execute After Accept

- A combination of message loss and crashes
  - Prevent non-leader servers from executing after receiving an accept

Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Five Servers, S_2 Cannot Execute After Accept

- A combination of message loss and crashes
  - Prevent non-leader servers from executing after receiving an accept
  - This was not necessary for the three server case
    - The accept from the leader is an implicit learn
    - And together with its own "learn", can execute!

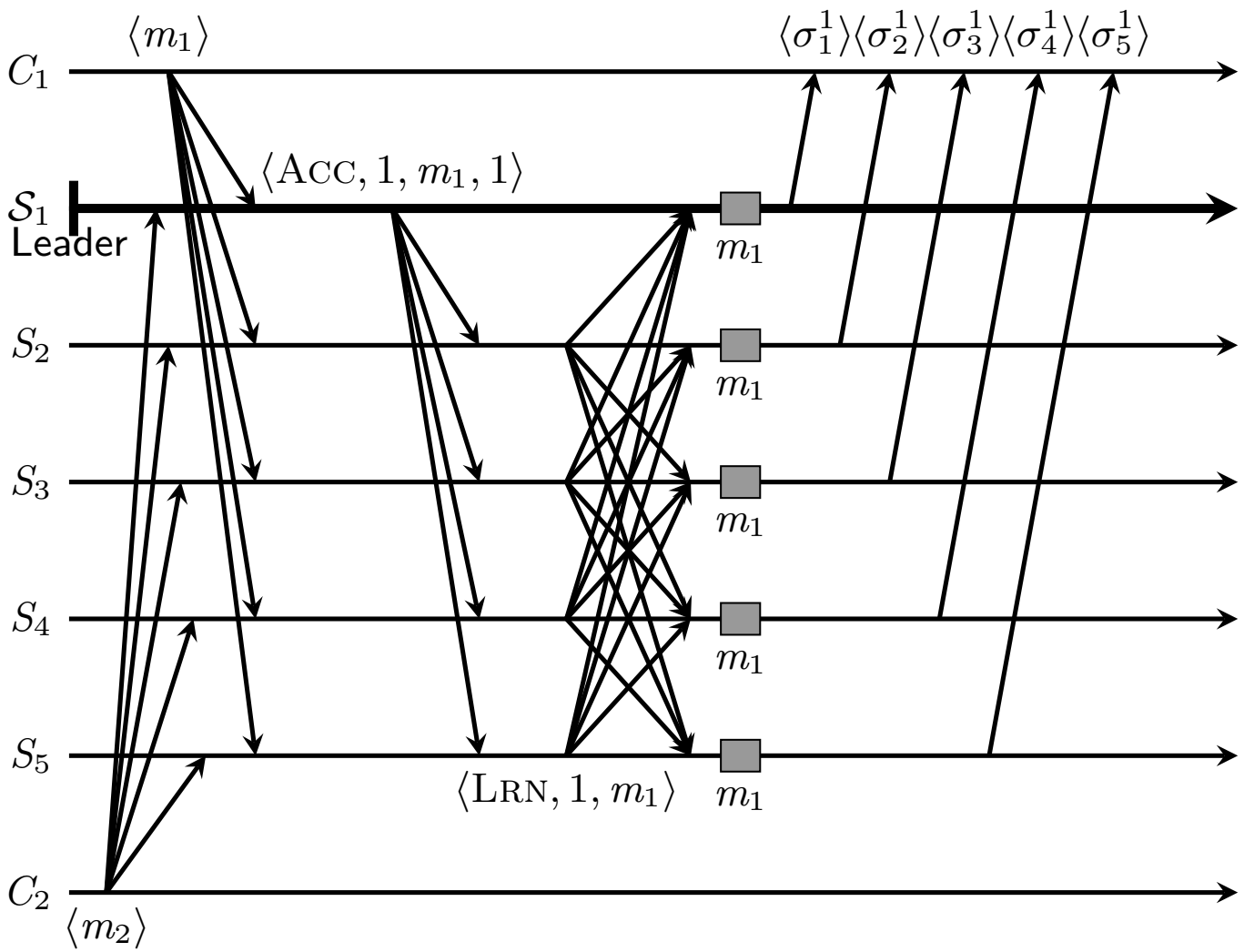
Fonte: Hein Meling e Leander Jehl - OPODIS2013

With Five Servers, S_2 Cannot Execute After Accept

- A combination of message loss and crashes
  - Prevent non-leader servers from executing after receiving an accept
  - This was not necessary for the three server case
    - The accept from the leader is an implicit learn
    - And together with its own "learn", can execute!
- There are two solutions:
  - Wait for all-to-all learn
  - Wait for commit from leader

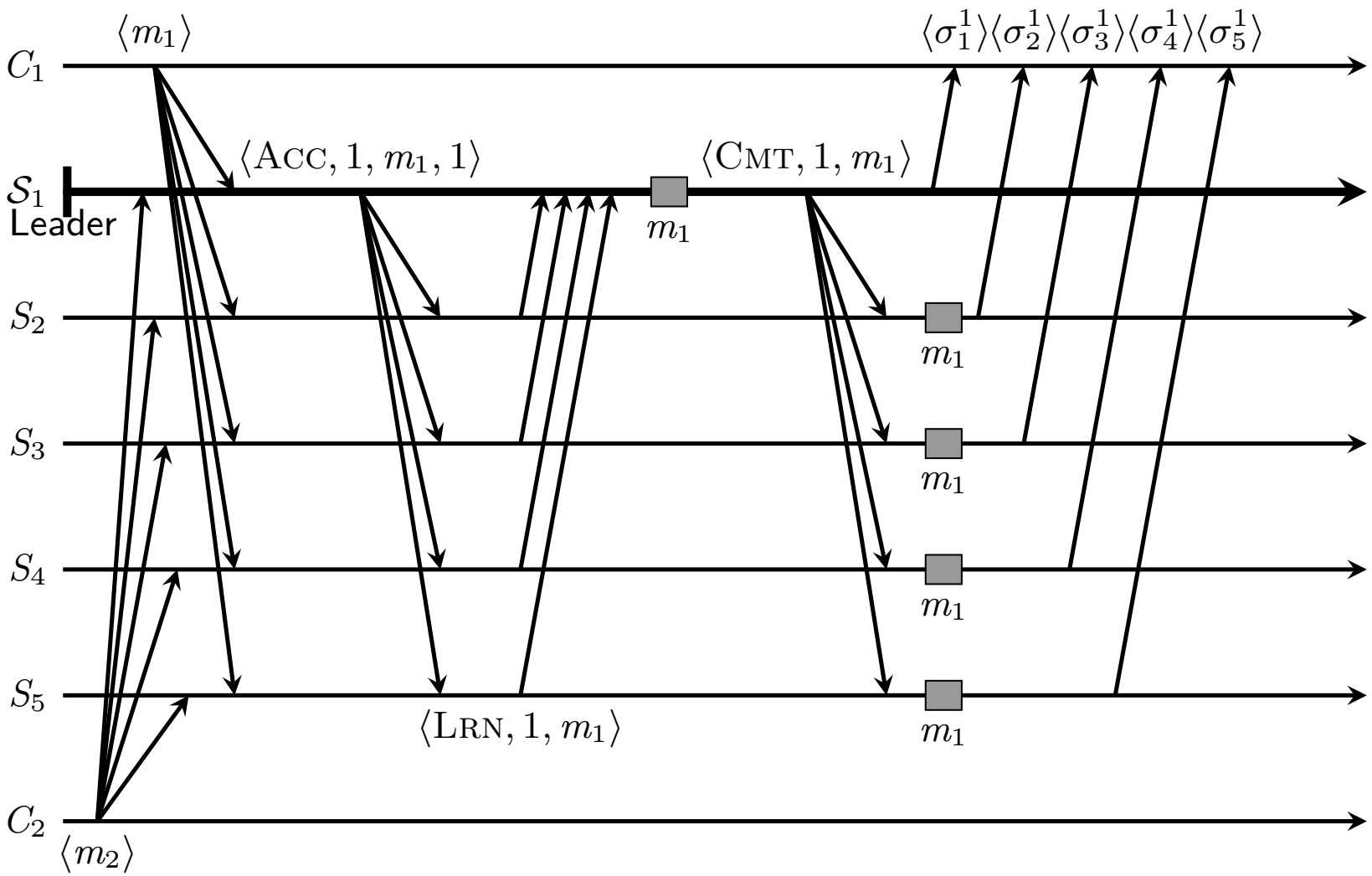
Fonte: Hein Meling e Leander Jehl - OPODIS2013

All-to-All Learn Before Execute



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Await Commit Before Execute

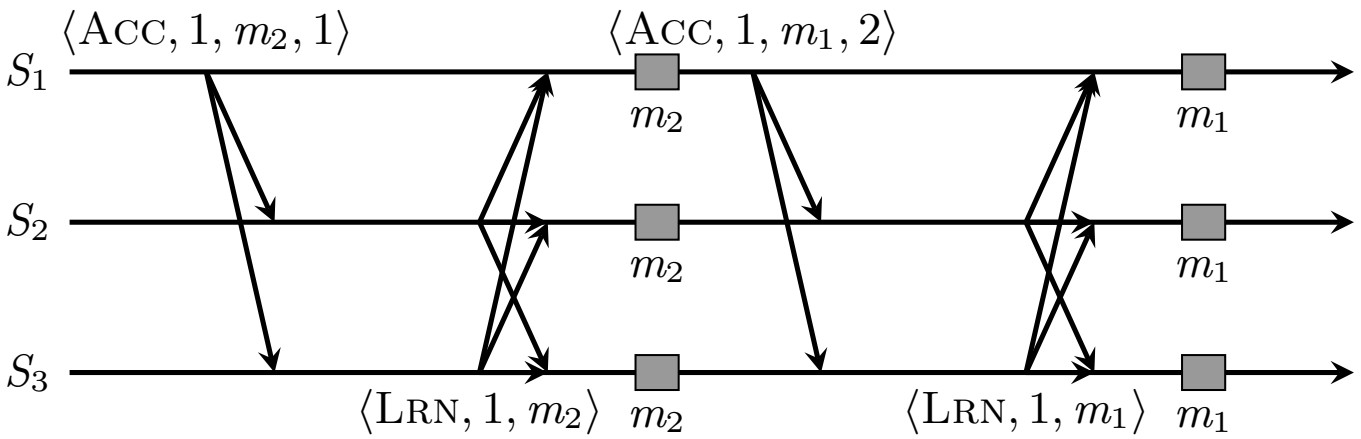


Fonte: Hein Meling e Leander Jehl - OPODIS2013

Wrapping it up!

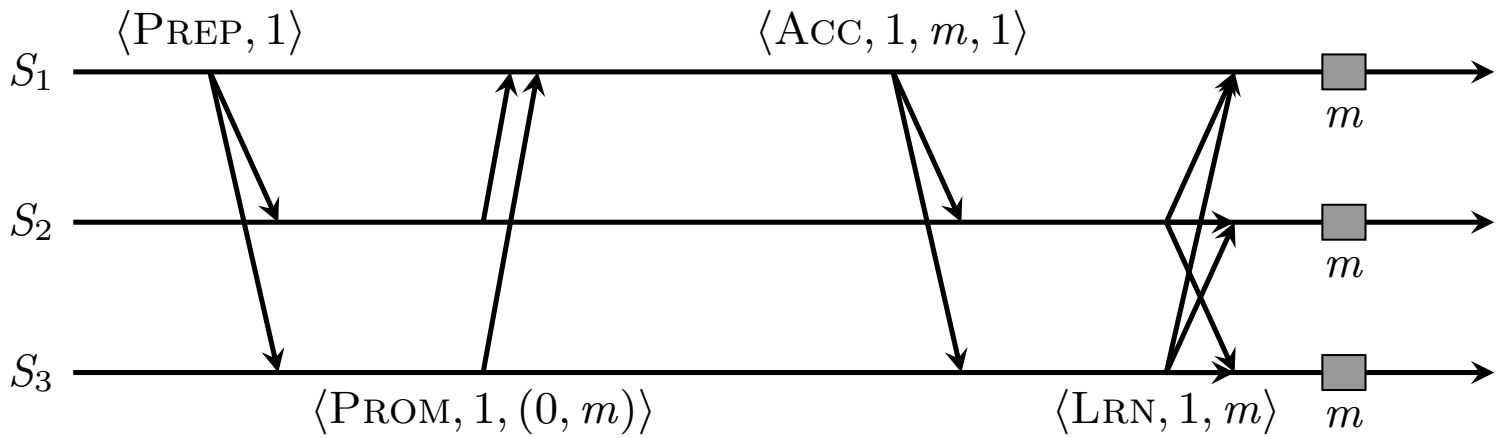
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Multi-Paxos



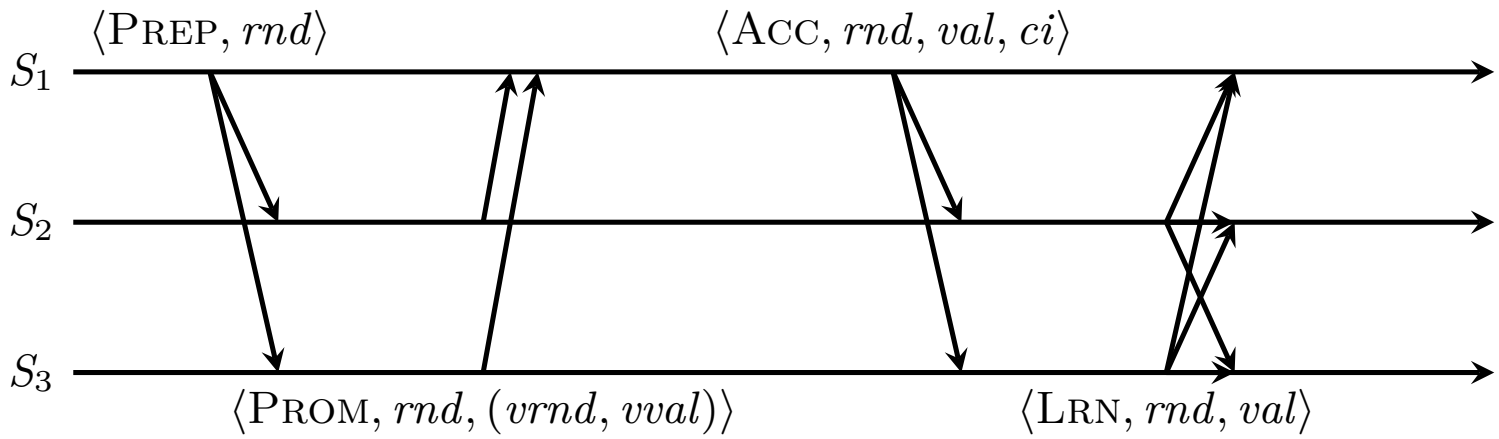
Fonte: Hein Meling e Leander Jehl - OPODIS2013

Paxos



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Paxos



Fonte: Hein Meling e Leander Jehl - OPODIS2013

Paxos Agents

- Proposer = Leader
  - Sends prepare and accept messages
  - Receive promise messages
- Acceptor
  - Receive accept messages
  - Sends learn messages
- Learner
  - Receive learn messages

Fonte: Hein Meling e Leander Jehl - OPODIS2013

Agreement in Distributed Systems: Possible Solutions

- Two-Phase Commit (2PC)
- Paxos

Fonte: Amir H. Payberah

Two-Phase Commit (2PC)

Fonte: Amir H. Payberah

The Two-Phase Commit (2PC) Problem

- The problem first was encountered in database systems.
- Suppose a database system is updating some complicated data structures that include parts residing on more than one machine.

Fonte: Amir H. Payberah

The Two-Phase Commit (2PC) Problem

- The problem first was encountered in database systems.
- Suppose a database system is updating some complicated data structures that include parts residing on more than one machine.
- System model:
  - Concurrent processes and uncertainty of timing, order of events and inputs (asynchronous systems).
  - Failure and recovery of machines/processors, of communication channels.

Fonte: Amir H. Payberah

Intuitive Example (1/3)

- You want to organize outing with 3 friends at 6pm Tuesday.
  - Go out only if all friends can make it.

Fonte: Amir H. Payberah

Intuitive Example (2/3)

- What do you do?

Fonte: Amir H. Payberah

Intuitive Example (2/3)

- What do you do?
  - Call each of them and ask if can do 6pm on Tuesday (voting phase)

Fonte: Amir H. Payberah

Intuitive Example (2/3)

- 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)

Fonte: Amir H. Payberah

Intuitive Example (2/3)

- 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 cannot do Tuesday, call other three to cancel (abort)

Fonte: Amir H. Payberah

Intuitive Example (3/3)

- Critical details

Fonte: Amir H. Payberah

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.

Fonte: Amir H. Payberah

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.
  - You need to remember the decision and tell anyone whom you have not been able to reach during commit/abort phase.

Fonte: Amir H. Payberah

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.
  - You need to remember the decision and tell anyone whom you have not been able to reach during commit/abort phase.
- That is exactly how 2PC works.

Fonte: Amir H. Payberah

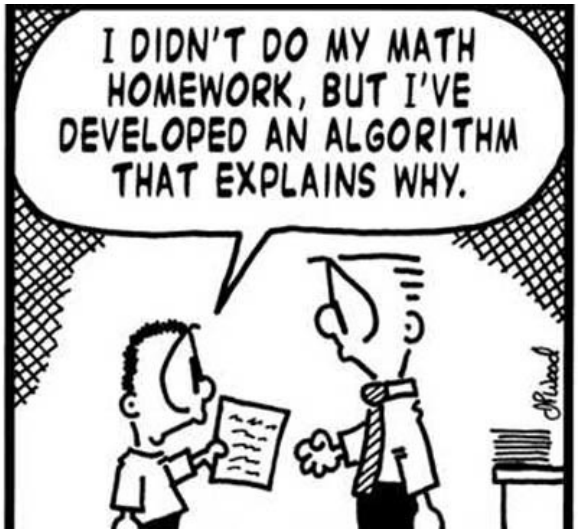
The 2PC Players

- Coordinator (Transaction Manager)
  - Begins transaction.
  - Responsible for commit/abort.
- Participants (Resource Managers)
  - The servers with the data used in the distributed transaction.

Fonte: Amir H. Payberah

The 2PC Algorithm

- Phase 1 - prepare phase
- Phase 2 - commit phase



Fonte: Amir H. Payberah

The 2PC Algorithm - Prepare Phase

- Coordinator asks each participant `canCommit`.

Fonte: Amir H. Payberah

The 2PC Algorithm - Prepare Phase

- Coordinator asks each participant `canCommit`.
- Participants must prepare to commit using permanent storage before answering `yes`.

Fonte: Amir H. Payberah

The 2PC Algorithm - Prepare Phase

- Coordinator asks each participant `canCommit`.
- Participants must prepare to commit using permanent storage before answering `yes`.
  - Lock the objects.
  - Participants are not allowed to cause an abort after it replies `yes` to `canCommit`.

Fonte: Amir H. Payberah

The 2PC Algorithm - Prepare Phase

- Coordinator asks each participant `canCommit`.
- Participants must prepare to commit using permanent storage before answering `yes`.
  - Lock the objects.
  - Participants are not allowed to cause an abort after it replies `yes` to `canCommit`.
- Outcome of the transaction is uncertain until `doCommit` or `doAbort`.
  - Other participants might still cause an abort.

Fonte: Amir H. Payberah

The 2PC Algorithm - Commit Phase

- The coordinator collects all votes.
  - If unanimous yes, causes commit.
  - If any participant voted no, causes abort.

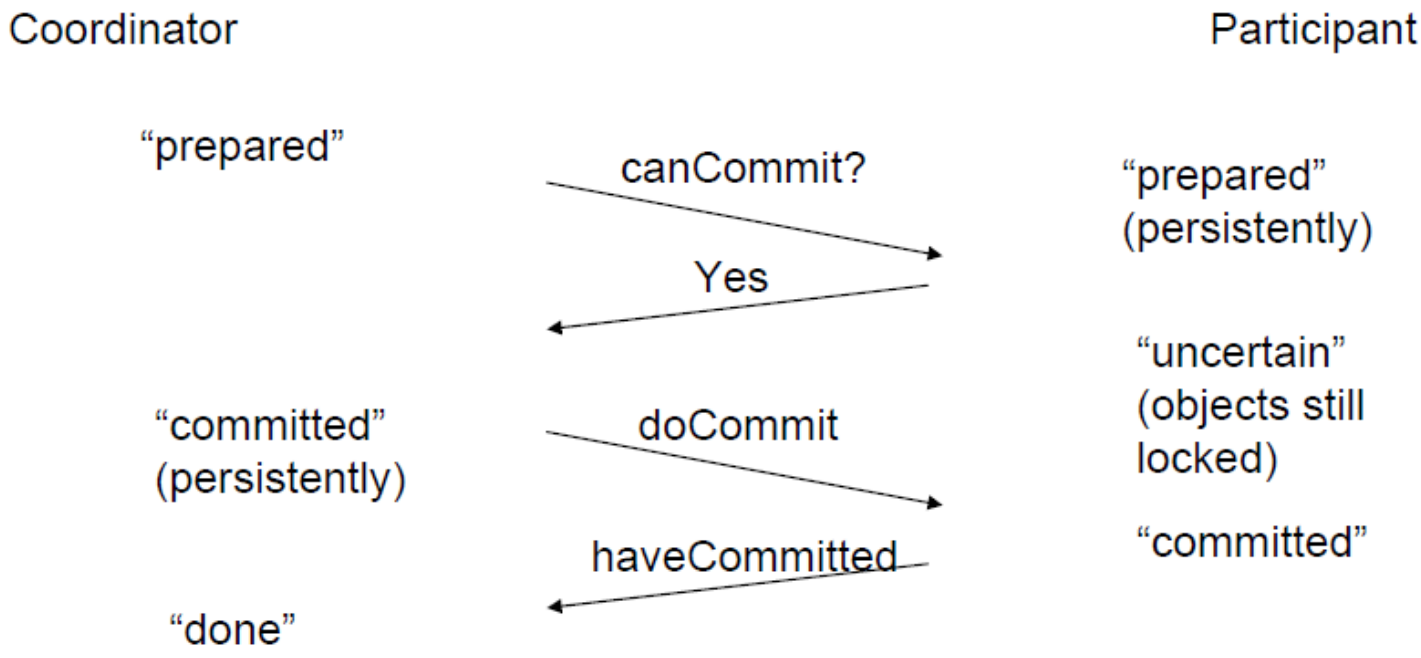
Fonte: Amir H. Payberah

The 2PC Algorithm - Commit Phase

- The coordinator collects all votes.
  - If unanimous yes, causes commit.
  - If any participant voted no, causes abort.
- The fate of the transaction is decided atomically at the coordinator, once all participants vote.
  - Coordinator records fate using permanent storage.
  - Then broadcasts `doCommit` or `doAbort` to participants.

Fonte: Amir H. Payberah

2PC Sequence of Events



Fonte: Amir H. Payberah

Recovery in 2PC

- Recovery after timeouts.
- Recovery after crashes and reboot.

Fonte: Amir H. Payberah

Recovery in 2PC

- Recovery after timeouts.
- Recovery after crashes and reboot.
- Note: you cannot differentiate between the above in a realistic asynchronous network.

Fonte: Amir H. Payberah

Handling Timeout

- To avoid processes blocking for ever.

Fonte: Amir H. Payberah

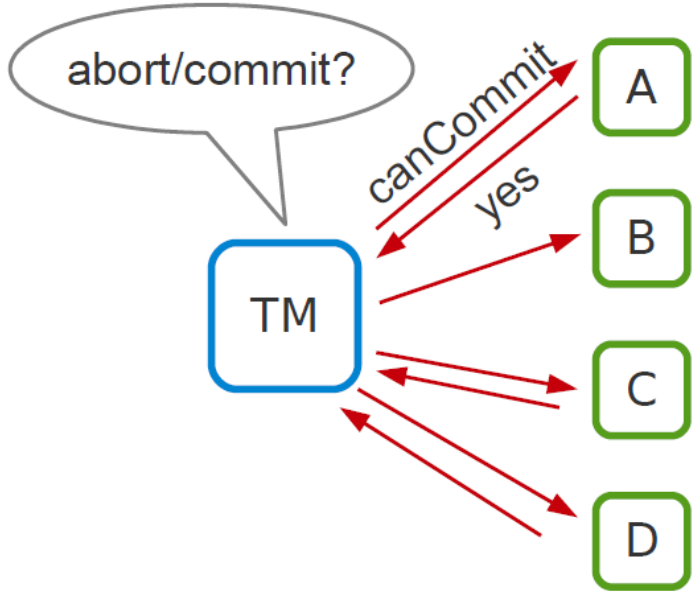
Handling Timeout

- To avoid processes blocking for ever.
- Two scenarios:
  - Coordinator waits for votes from participants.
  - Participant is waiting for the final decision.

Fonte: Amir H. Payberah

Handling Timeout at Coordinator

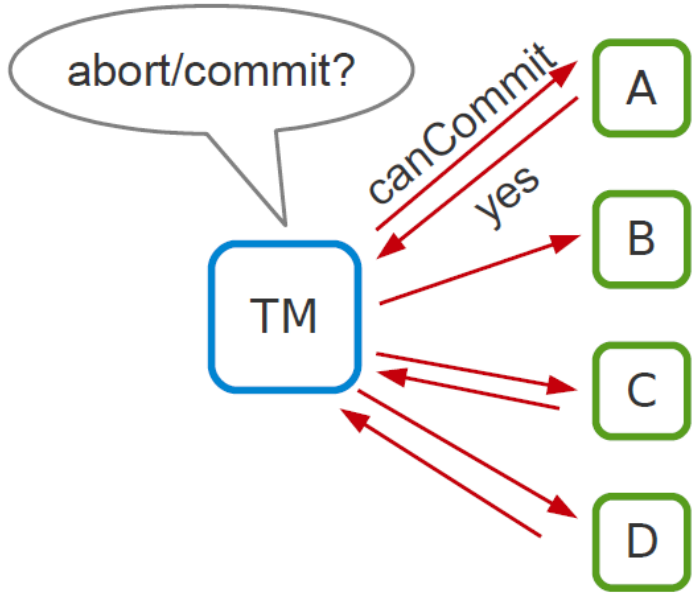
- If B voted no, can coordinator unilaterally abort?
- If B voted yes, can coordinator unilaterally abort/commit?



Fonte: Amir H. Payberah

Handling Timeout at Coordinator

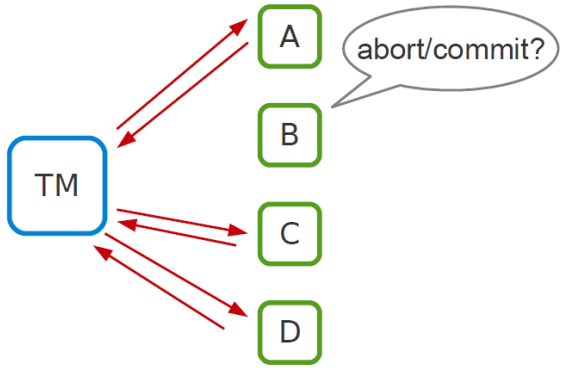
- If B voted no, can coordinator unilaterally abort?
- If B voted yes, can coordinator unilaterally abort/commit?
  - Coordinator waits for votes from participants.
  - Participant is waiting for the final decision.
  - Coordinator timeout abort and send `doAbort` to participants.



Fonte: Amir H. Payberah

Handling Timeout at Participants

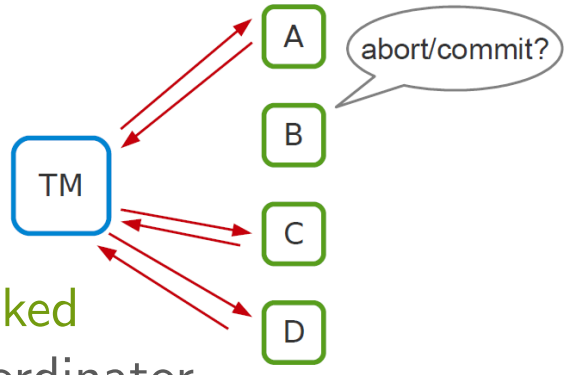
- If B times out on TM and has voted yes, then execute termination protocol.



Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.

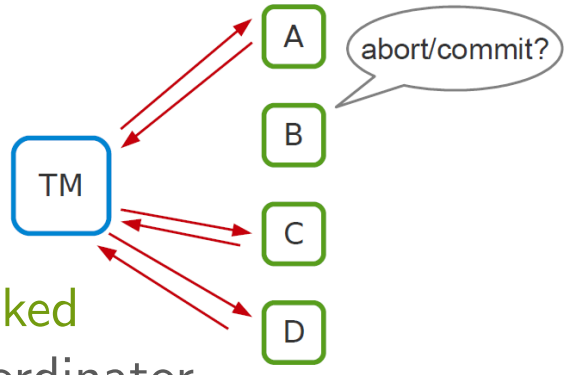


- Simple protocol: participant is remained blocked until it can establish communication with coordinator.

Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.

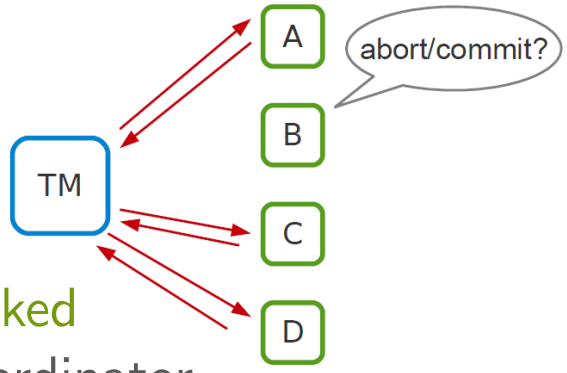


- Simple protocol: participant is remained blocked until it can establish communication with coordinator.
- Cooperative protocol: participant sends a decision-request message to other participants.

Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.

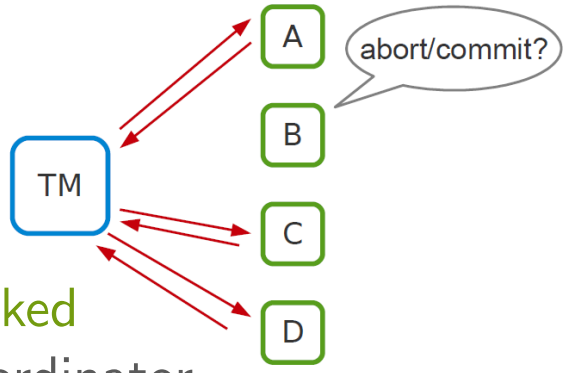


- Simple protocol: participant is remained blocked until it can establish communication with coordinator.
- Cooperative protocol: participant sends a decision-request message to other participants.
  - B sends status message to A

Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.

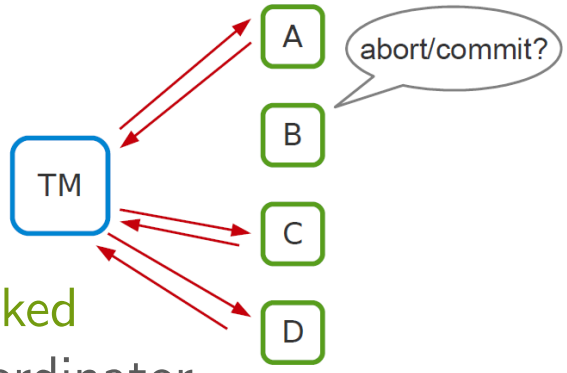


- Simple protocol: participant is remained blocked until it can establish communication with coordinator.
- Cooperative protocol: participant sends a decision-request message to other participants.
  - B sends status message to A
  - If A has received commit/abort from TM, ...

Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.

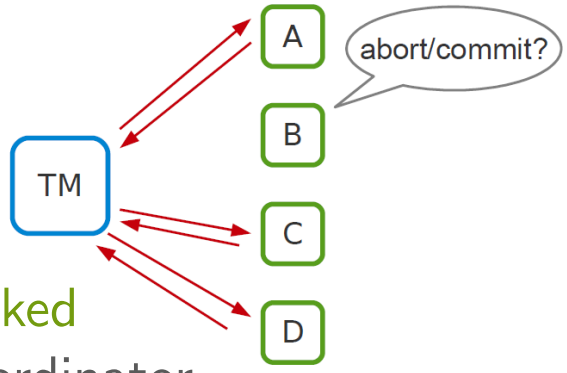


- Simple protocol: participant is remained blocked until it can establish communication with coordinator.
- Cooperative protocol: participant sends a decision-request message to other participants.
  - B sends status message to A
  - If A has received commit/abort from TM, ...
  - If A has not responded to TM, ...

Fonte: Amir H. Payberah

Handling Timeout at Participants

- If B times out on TM and has voted yes, then execute termination protocol.



- Simple protocol: participant is remained blocked until it can establish communication with coordinator.
- Cooperative protocol: participant sends a decision-request message to other participants.
  - B sends status message to A
  - If A has received commit/abort from TM, ...
  - If A has not responded to TM, ...
  - If A has responded with no/yes ...

Fonte: Amir H. Payberah

Handling Crash and Recovery (1/2)

- All nodes must log protocol progress.
  - Participants: prepared, uncertain, committed/aborted
  - Coordinator: prepared, committed/aborted, done

Fonte: Amir H. Payberah

Handling Crash and Recovery (1/2)

- All nodes must log protocol progress.
  - Participants: prepared, uncertain, committed/aborted
  - Coordinator: prepared, committed/aborted, done
- Nodes cannot back out if commit is decided.

Fonte: Amir H. Payberah

Handling Crash and Recovery (2/2)

- Coordinator crashes:
  - If it finds no commit on disk, it aborts.
  - If it finds commit, it commits.

Fonte: Amir H. Payberah

Handling Crash and Recovery (2/2)

- Coordinator crashes:
  - If it finds no commit on disk, it aborts.
  - If it finds commit, it commits.
- Participant crashes:
  - If it finds no yes on disk, it aborts.
  - If it finds yes, runs termination protocol to decide.

Fonte: Amir H. Payberah

Fault-Tolerance Limitations of 2PC

- Even with recovery enabled, 2PC is not really fault-tolerant (or live), because it can be blocked even when one (or a few) machines fail.
- Blocking means that it does not make progress during the failures.

Fonte: Amir H. Payberah

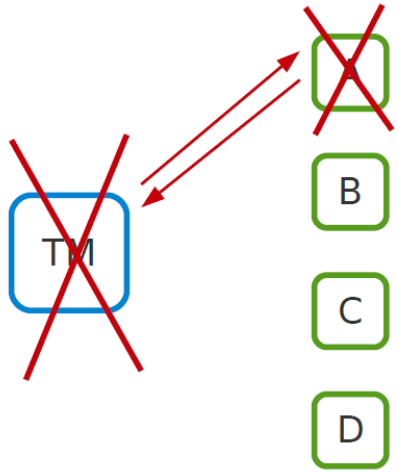
Fault-Tolerance Limitations of 2PC

- Even with recovery enabled, 2PC is not really fault-tolerant (or live), because it can be blocked even when one (or a few) machines fail.
- Blocking means that it does not make progress during the failures.
- Any scenarios?

Fonte: Amir H. Payberah

2PC Blocking Scenario

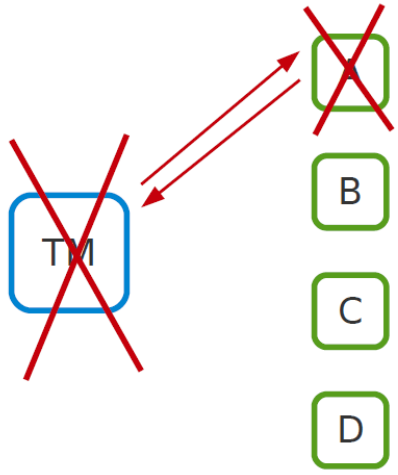
- TM sends `doCommit` to A, A gets it and commits, and then both TM and A die.



Fonte: Amir H. Payberah

2PC Blocking Scenario

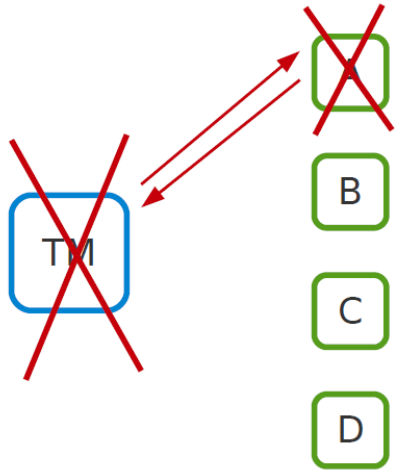
- TM sends `doCommit` to A, A gets it and commits, and then both TM and A die.
- B, C, D have already also replied `yes`, have locked their mutexes, and now need to wait for TM or A to reappear.
  - They cannot recover the decision with certainty until TM or A are online.



Fonte: Amir H. Payberah

2PC Blocking Scenario

- TM sends `doCommit` to A, A gets it and commits, and then both TM and A die.
- B, C, D have already also replied `yes`, have locked their mutexes, and now need to wait for TM or A to reappear.
  - They cannot recover the decision with certainty until TM or A are online.
- This is why 2PC is called a blocking protocol: 2PC is safe, but not live.



Fonte: Amir H. Payberah

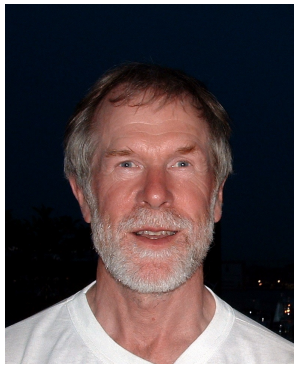
Impossibility of Distributed Consensus with One Faulty Process

Fonte: Amir H. Payberah

FLP

- Fischer-Lynch-Paterson (FLP)
  - M.J. Fischer, N.A. Lynch, and M.S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM, 1985.





Fonte: Amir H. Payberah

FLP 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.

Fonte: Amir H. Payberah

FLP Impossibility Result

- What FLP says: you cannot guarantee both safety and progress when there is even a single fault at an inopportune moment.

Fonte: Amir H. Payberah

FLP Impossibility Result

- What FLP says: you cannot guarantee both safety and progress when there is even a single fault at an inopportune moment.
- What FLP does not say: in practice, how close can you get to the ideal (always safe and live)?

Fonte: Amir H. Payberah

FLP Impossibility Result

- What FLP says: you cannot guarantee both safety and progress when there is even a single fault at an inopportune moment.
- What FLP does not say: in practice, how close can you get to the ideal (always safe and live)?
- So, Paxos ...

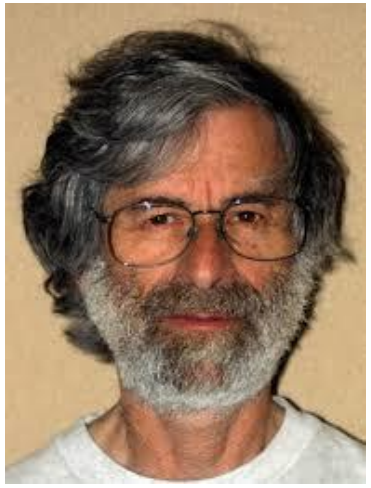
Fonte: Amir H. Payberah

Paxos

Fonte: Amir H. Payberah

Paxos

- The only known completely-safe and largely-live agreement protocol.
- L. Lamport, The part-time parliament, ACM Transactions on Computer Systems, 1998.



Fonte: Amir H. Payberah

The Paxos Players

- Proposers
  - Suggests values for consideration by acceptors.
- Acceptors
  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.
- Learners
  - Learns the chosen value.

Fonte: Amir H. Payberah

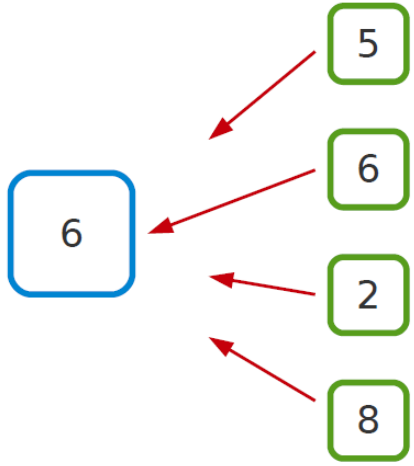
The Paxos Players

- Proposers
  - Suggests values for consideration by acceptors.
- Acceptors
  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.
- Learners
  - Learns the chosen value.
- A node can act as more than one roles (usually 3).

Fonte: Amir H. Payberah

Single Proposal, Single Acceptor

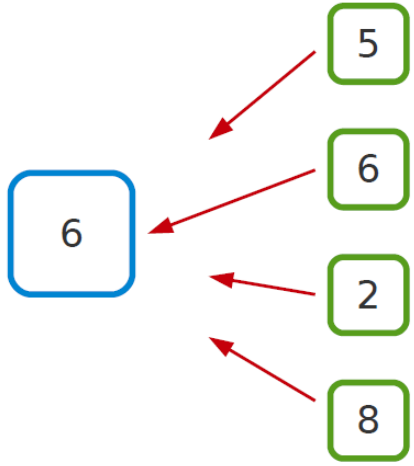
- Use just one acceptor
  - Collects proposers' proposals.
  - Decides the value and tells everyone else.



Fonte: Amir H. Payberah

Single Proposal, Single Acceptor

- Use just one acceptor
  - Collects proposers' proposals.
  - Decides the value and tells everyone else.

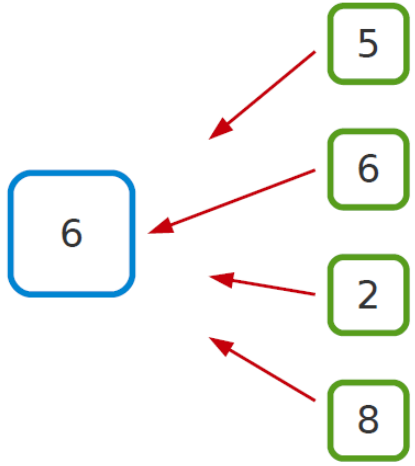


- Sounds familiar?

Fonte: Amir H. Payberah

Single Proposal, Single Acceptor

- Use just one acceptor
  - Collects proposers' proposals.
  - Decides the value and tells everyone else.



- Sounds familiar?
  - two-phase commit (2PC)
  - acceptor fails = protocol blocks

Fonte: Amir H. Payberah

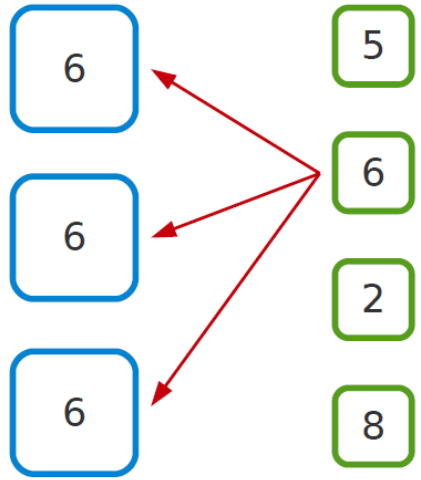
Single Proposal, Multiple Acceptors

- One acceptor is not fault-tolerant enough.
- Let's have multiple acceptors.

Fonte: Amir H. Payberah

Single Proposal, Multiple Acceptors

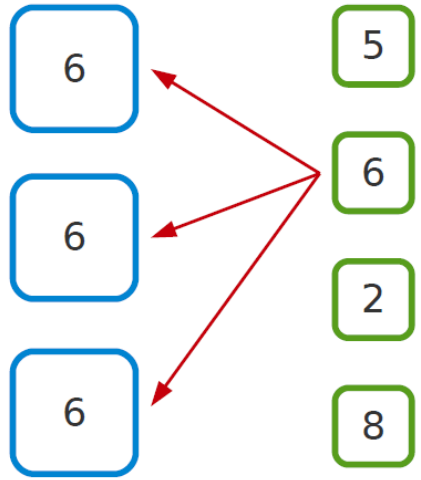
- One acceptor is not fault-tolerant enough.
- Let's have multiple acceptors.



Fonte: Amir H. Payberah

Single Proposal, Multiple Acceptors

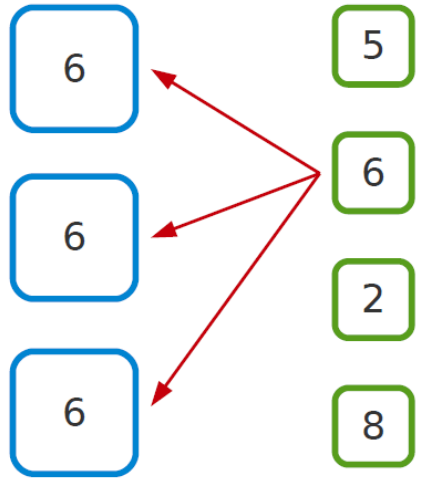
- One acceptor is not fault-tolerant enough.
- Let's have multiple acceptors.
- From there, must reach a decision. How?



Fonte: Amir H. Payberah

Single Proposal, Multiple Acceptors

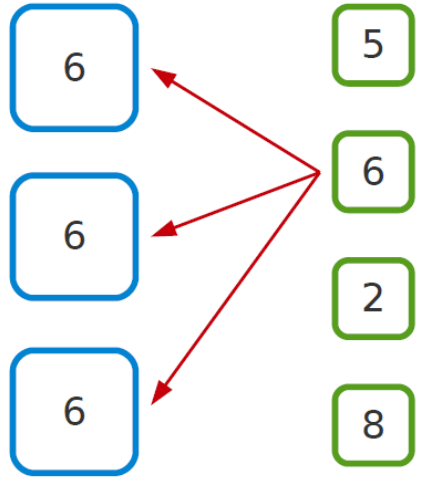
- One acceptor is not fault-tolerant enough.
- Let's have multiple acceptors.
- From there, must reach a decision. How?
- Decision = value accepted by the majority.



Fonte: Amir H. Payberah

Single Proposal, Multiple Acceptors

- One acceptor is not fault-tolerant enough.
- Let's have multiple acceptors.

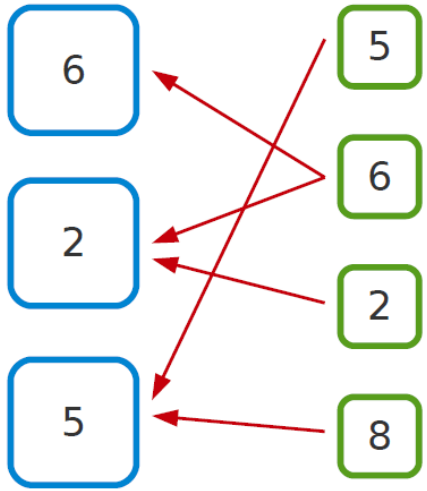


- From there, must reach a decision. How?
- Decision = value accepted by the majority.
- P1: an acceptor must accept first proposal it receives.

Fonte: Amir H. Payberah

Multiple Proposals, Multiple Acceptors

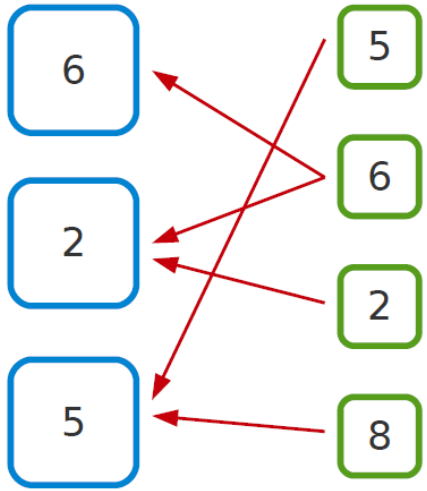
- If there are multiple proposals, no proposal may get the majority.
  - 3 proposals may each get 1/3 of the acceptors.



Fonte: Amir H. Payberah

Multiple Proposals, Multiple Acceptors

- If there are multiple proposals, no proposal may get the majority.
  - 3 proposals may each get 1/3 of the acceptors.



- Solution: acceptors can accept multiple proposals, distinguished by a unique proposal number.

Fonte: Amir H. Payberah

Handling Multiple Proposals

- All chosen proposals must have the same value.

Fonte: Amir H. Payberah

Handling Multiple Proposals

- All chosen proposals must have the same value.
- P2: If a proposal with value v is chosen, then every higher-numbered proposal that is chosen also has value v.

Fonte: Amir H. Payberah

Handling Multiple Proposals

- All chosen proposals must have the same value.
- P2: If a proposal with value v is chosen, then every higher-numbered proposal that is chosen also has value v.
  - P2a: ... accepted ...

Fonte: Amir H. Payberah

Handling Multiple Proposals

- All chosen proposals must have the same value.
- P2: If a proposal with value v is chosen, then every higher-numbered proposal that is chosen also has value v.
  - P2a: ... accepted ...
  - P2b: ... proposed ...

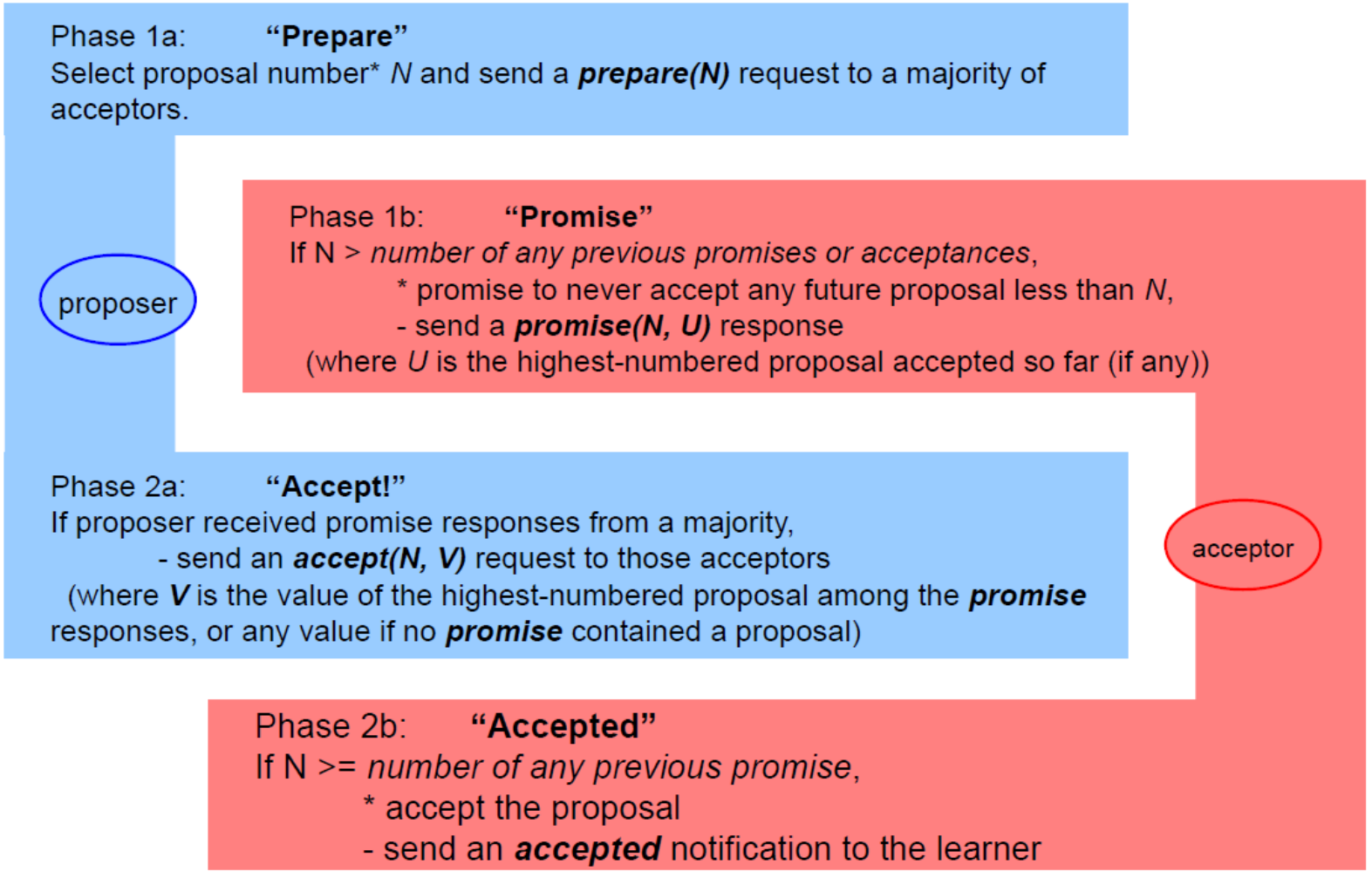
Fonte: Amir H. Payberah

The Paxos Algorithm

- Phase 1a - prepare phase
- Phase 1b - promise phase
- Phase 2a - accept phase
- Phase 2b - accepted phase

Fonte: Amir H. Payberah

Paxos Algorithm



Fonte: Amir H. Payberah

Paxos Algorithm - Prepare Phase

- A proposer selects a proposal number n and sends a `prepare` request with number n to majority of acceptors.

Fonte: Amir H. Payberah

Paxos Algorithm - Promise Phase

- If an acceptor receives a `prepare` request with number n greater than that of any prepare request it saw

Fonte: Amir H. Payberah

Paxos Algorithm - Promise Phase

- If an acceptor receives a `prepare` request with number n greater than that of any prepare request it saw
  - It responses `yes` to that request with a promise not to accept any more proposals numbered less than n.

Fonte: Amir H. Payberah

Paxos Algorithm - Promise Phase

- If an acceptor receives a `prepare` request with number n greater than that of any prepare request it saw
  - It responses `yes` to that request with a promise not to accept any more proposals numbered less than n.
  - It includes the highest-numbered proposal (if any) that it has accepted.

Fonte: Amir H. Payberah

Paxos Algorithm - Accept Phase

- If the proposer receives a response `yes` to its `prepare` requests from a majority of acceptors

Fonte: Amir H. Payberah

Paxos Algorithm - Accept Phase

- If the proposer receives a response `yes` to its `prepare` requests from a majority of acceptors
  - It sends an `accept` request to each of those acceptors for a proposal numbered n with a values v, which is the value of the highest-numbered proposal among the responses.

Fonte: Amir H. Payberah

Paxos Algorithm - Accepted Phase

- If an acceptor receives an `accept` request for a proposal numbered n

Fonte: Amir H. Payberah

Paxos Algorithm - Accepted Phase

- If an acceptor receives an `accept` request for a proposal numbered n
  - It accepts the proposal unless it has already responded to a `prepare` request having a number greater than n.

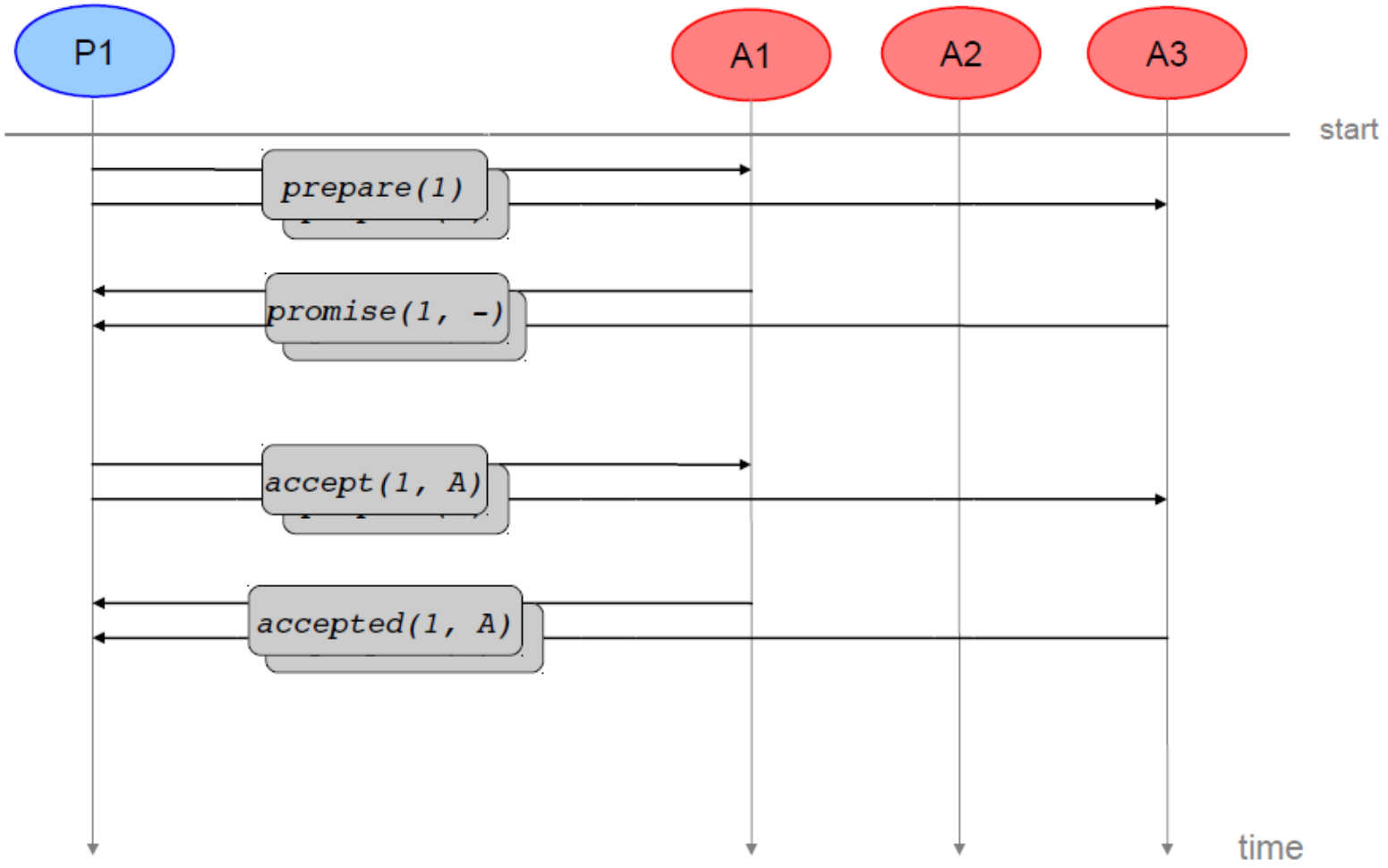
Fonte: Amir H. Payberah

Definition of Chosen

- A value is chosen at proposal number n, iff majority of acceptors accept that value in phase 2 of the proposal number.

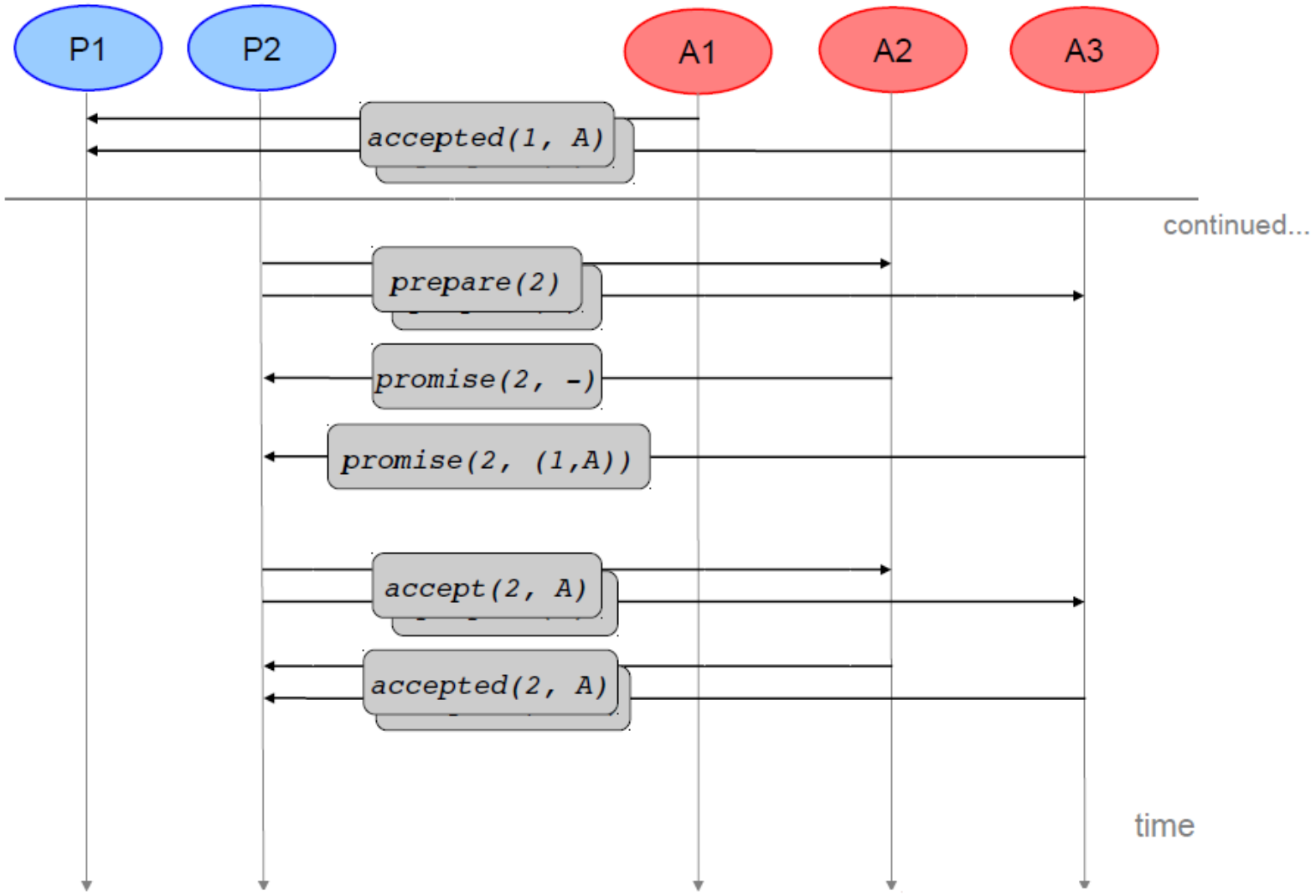
Fonte: Amir H. Payberah

Paxos Example



Fonte: Amir H. Payberah

Paxos Example



Fonte: Amir H. Payberah

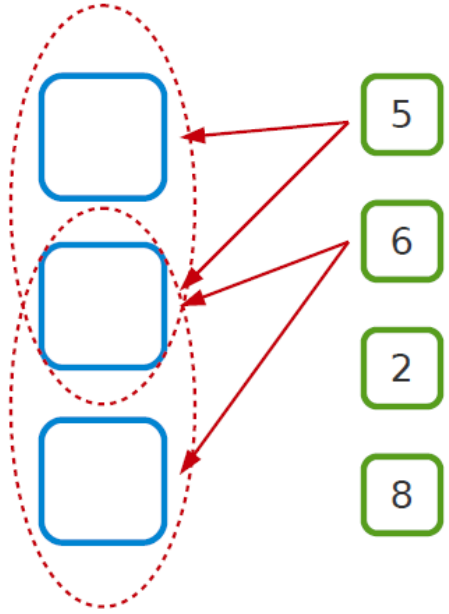
Paxos - Safety (1/3)

- If a value v is chosen at proposal number n, any value that is sent out in phase 2 of any later proposal numbers must be also v.

Fonte: Amir H. Payberah

Paxos - Safety (2/3)

- Decision = Majority (any two majorities share at least one element)
- Therefore after the first round in which there is a decision, any subsequent round involves at least one acceptor that has accepted v.



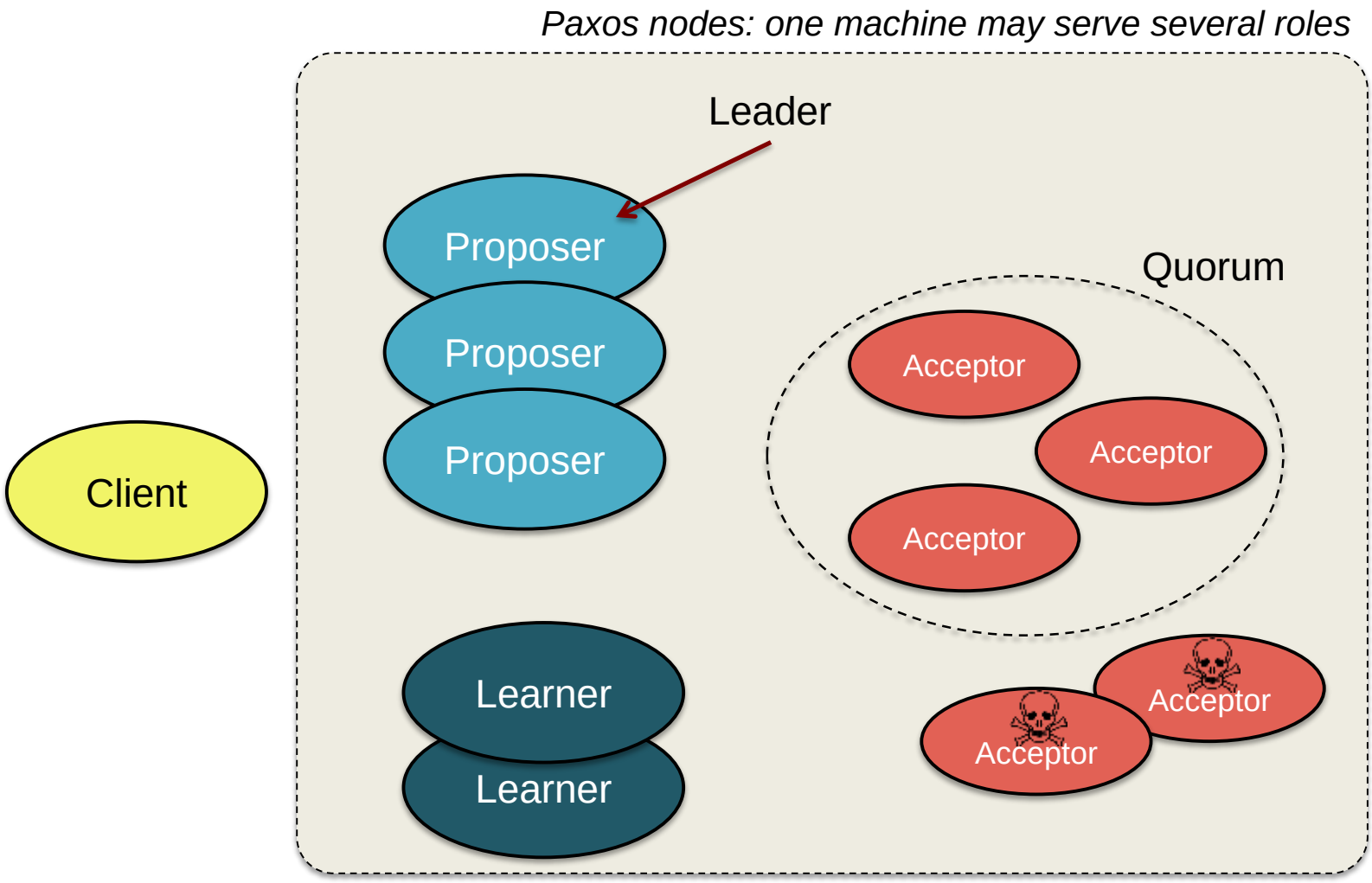
Fonte: Amir H. Payberah

Paxos - Safety (3/3)

- Now suppose our claim is not true, and let m is the first proposal number that is later than n and in 2nd phase, the value sent out is $w \neq v$.
- This is not possible, because if the proposer P was able to start 2nd phase for w, it means it got a majority to accept round for m (for $m > n$). So, either:
  - v would not have been the value decided, or
  - v would have been proposed by P
- Therefore, once a majority accepts v, that never changes.

Fonte: Amir H. Payberah

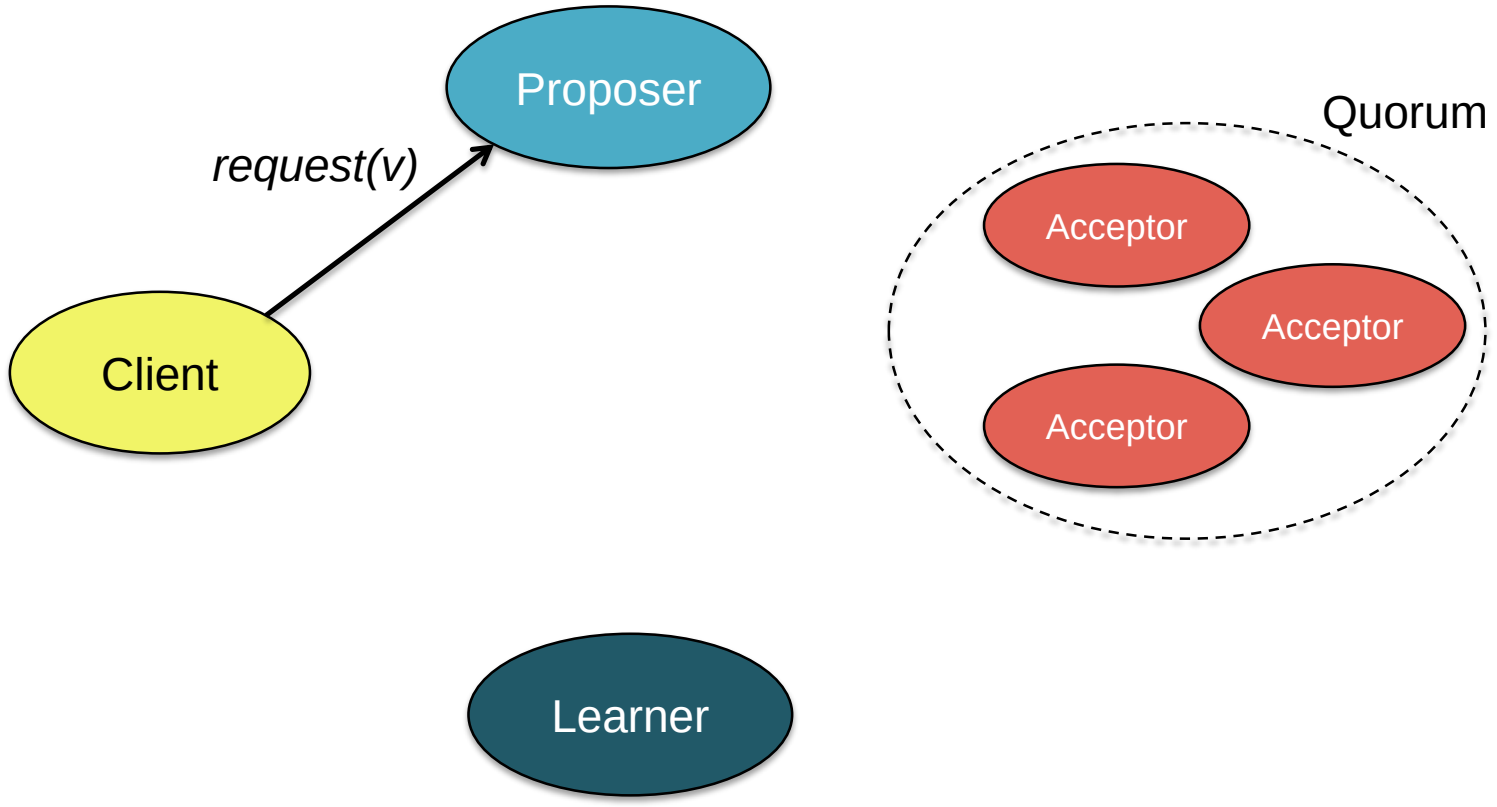
Paxos in action



Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 0

Client sends a request to a proposer

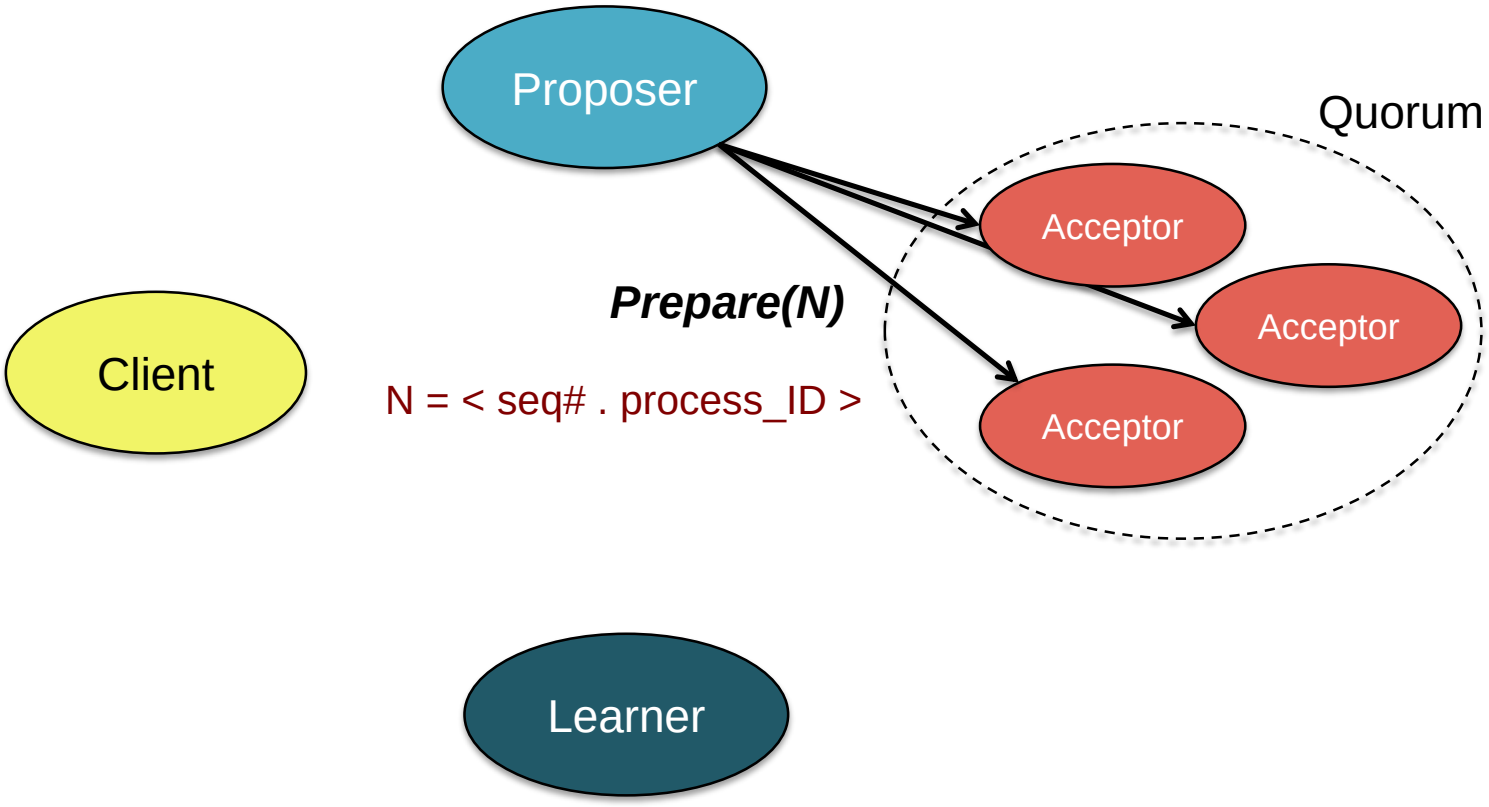


Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 1a

Proposer: creates a *proposal #N (N acts like a Lamport time stamp)*, where *N* is greater than any previous proposal number used by this proposer

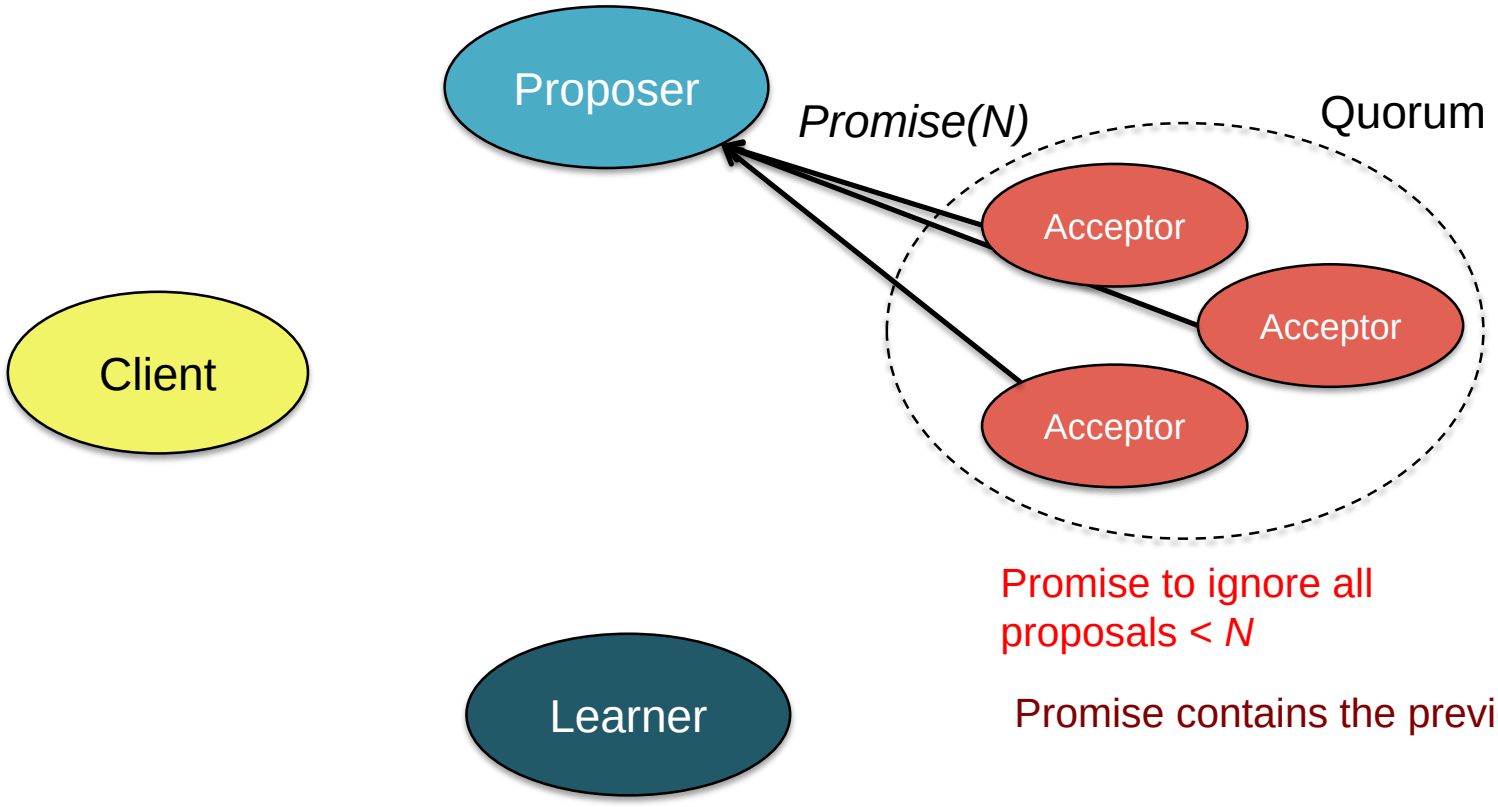
Send to Quorum of Acceptors (however many you can reach – but a majority)



Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 1b

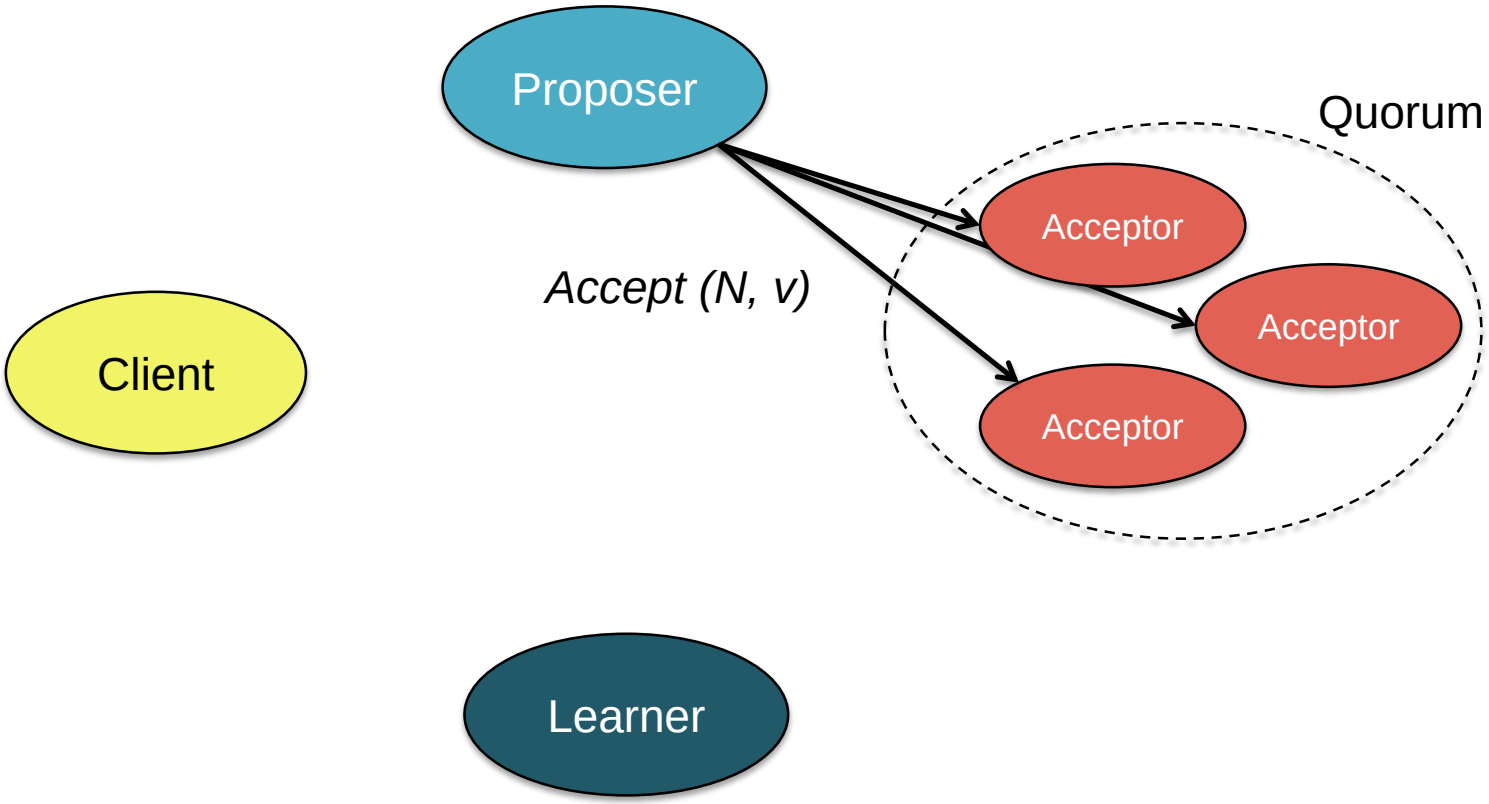
Acceptor: if proposer's ID > any previous proposal
promise to ignore all requests with IDs < N
reply with info about highest past proposal: { *N, value* }



Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 2a

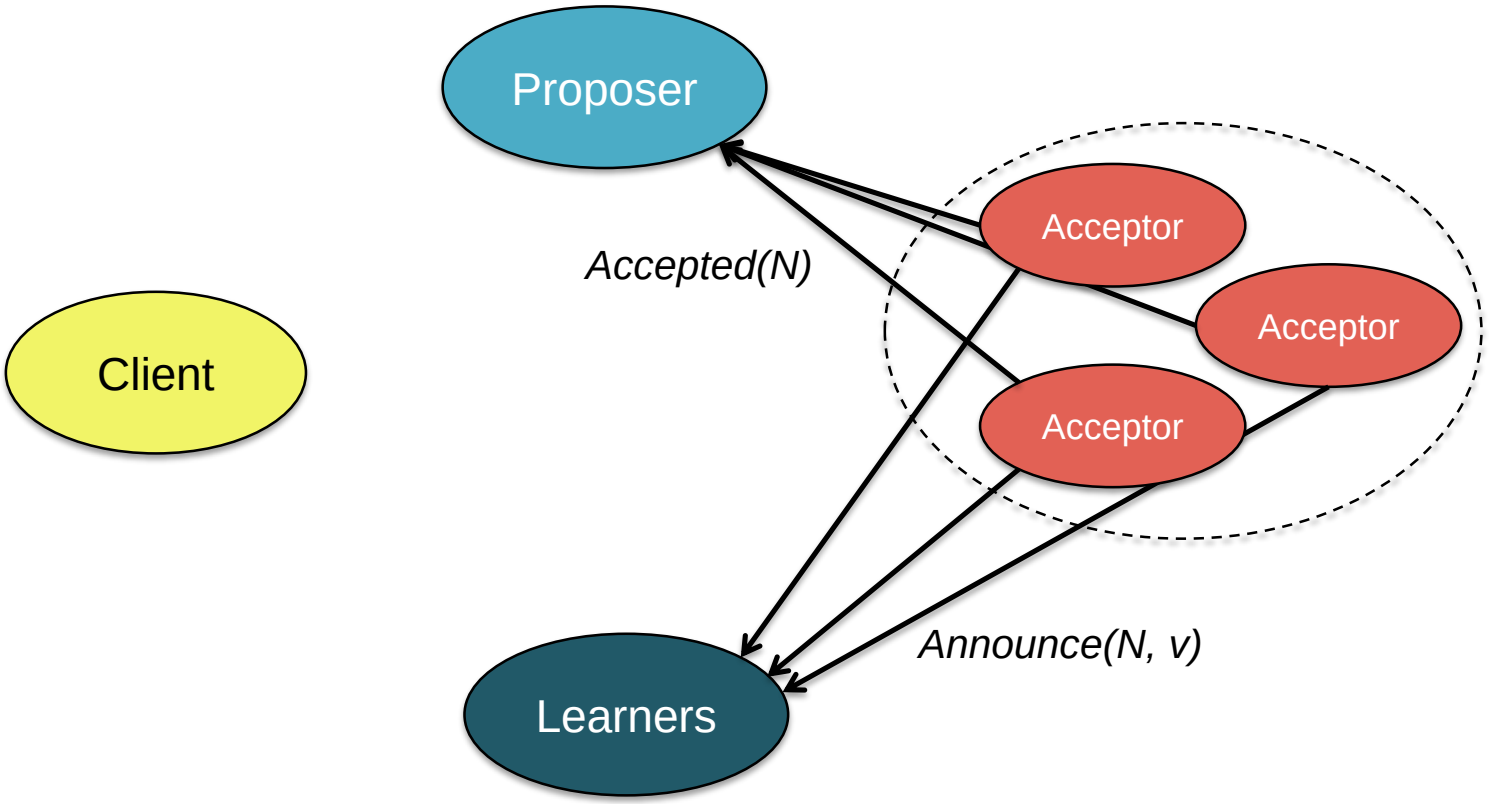
Proposer: if proposer receives promises from the quorum (majority):
Attach a value *v* to the proposal (the event).
Send ***Accept*** to quorum with the **chosen** value
If promise was for another {N, v}, proposer MUST accept that



Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 2b

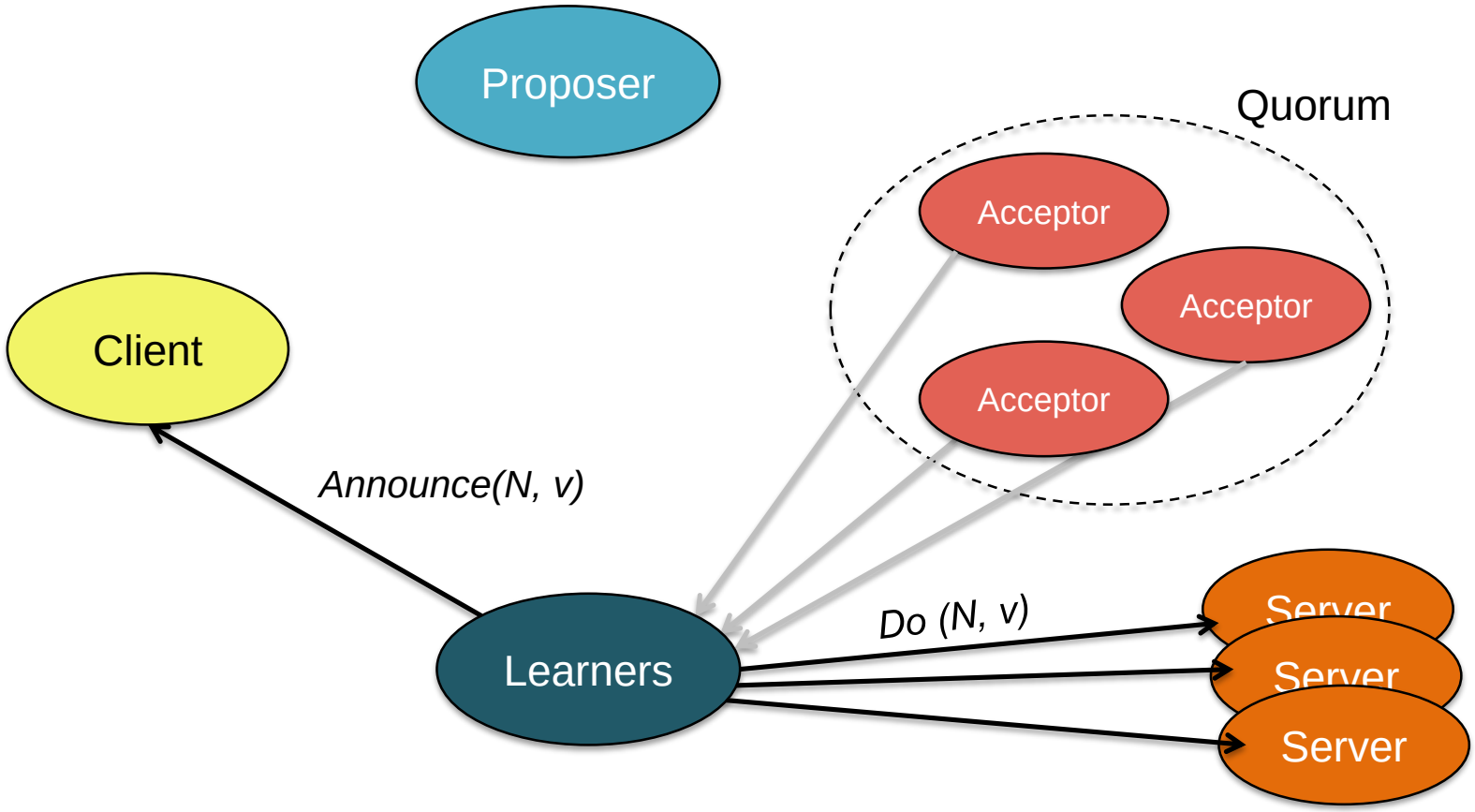
Acceptor: if the promise still holds, then announce the value *v*
Send ***Accepted*** message to Proposer and every Learner
else ignore the message (or send *NACK*)



Fonte: Prof. Paul Krzyzanowski

Paxos in action: Phase 3

Learner: Respond to client and/or take action on the request



Fonte: Prof. Paul Krzyzanowski

Paxos: Keep trying

- A proposal *N* may fail because
  - The acceptor may have made a new promise to ignore all proposals less than some value *M* >*N*
  - A proposer does not receive a quorum of responses: either *promise* (phase 1b) or *accept* (phase 2b)
- Algorithm then has to be restarted with a higher proposal #

Fonte: Prof. Paul Krzyzanowski