Paxos Explained from Scratch

Hein Meling and Leander Jehl

University of Stavanger

hein.meling@uis.no

International Conference On Principles of Distributed Systems

What is Paxos and why is it Relevant?

- Fault tolerant consensus protocol

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

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

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

Objectives and Approach

- Explain Paxos
  - Using visual aids
  - In a step-wise manner
  - With minimal changes in each step

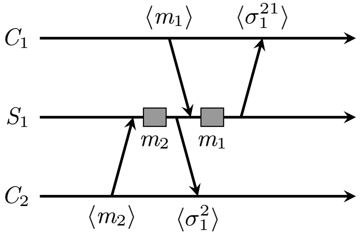
Objectives and Approach

- 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)

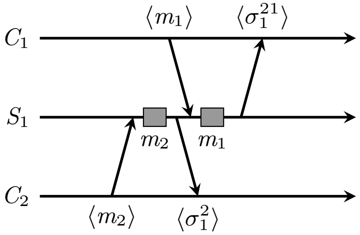
Objectives and Approach

- 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

A Stateful Service: *SingleServer*



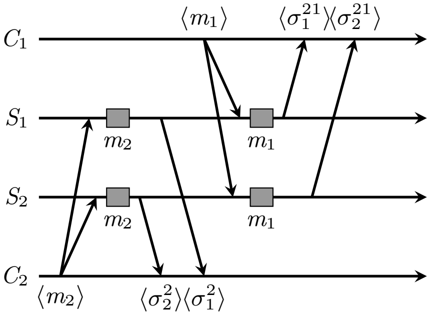
A Stateful Service: *SingleServer*



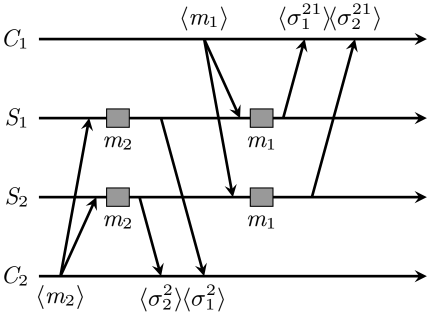
- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- Corresponds to execution sequence: $m_2 m_1$

We Want to Make the Service Fault Tolerant!

Fault Tolerance with Two Servers



Fault Tolerance with Two Servers

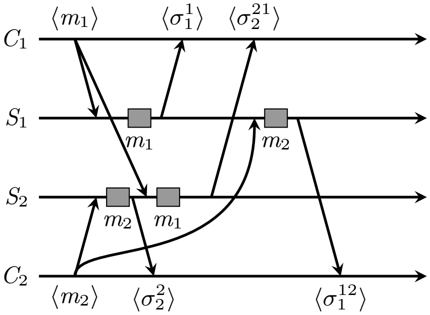


- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
  - σ^2 is a prefix of σ^{21}
- Corresponds to execution sequence: $m_2 m_1$

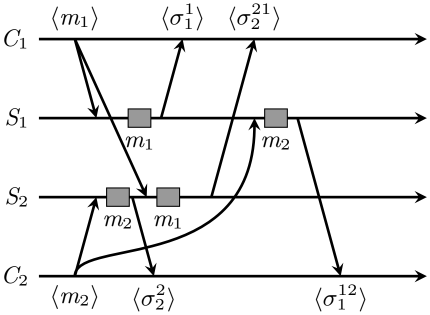
Deterministic State Machine

- 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

Fault Tolerance with Two Servers: Whoops!



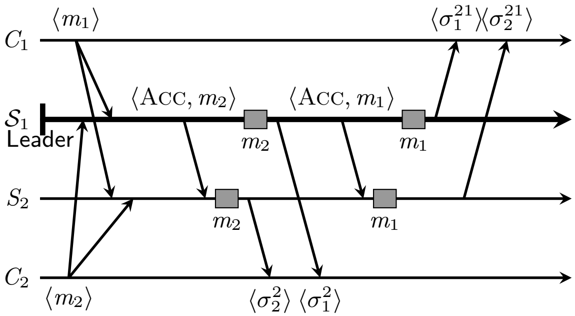
Fault Tolerance with Two Servers: Whoops!



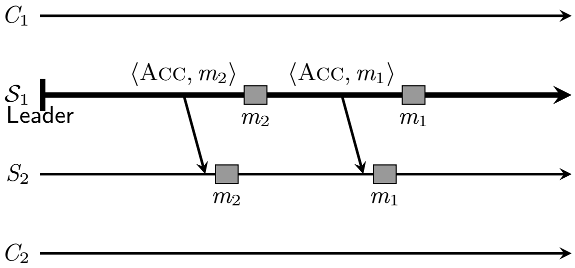
- Client C_2 sees: $\sigma^2\sigma^{12}$
  - σ^2 is not a prefix of σ^{12}
- Client C_1 sees: $\sigma^1\sigma^{21}$
  - σ^1 is not a prefix of σ^{21}
- Corresponds to execution sequence at
  - S_1: $m_1 m_2$
  - S_2: $m_2 m_1$

We Need to Order Client Requests!

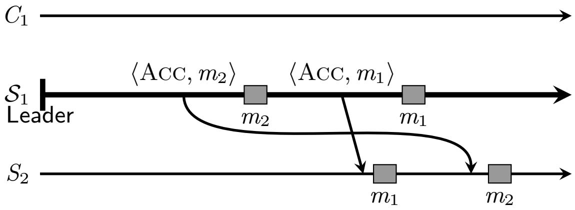
Let's Designate a Leader to Order Requests



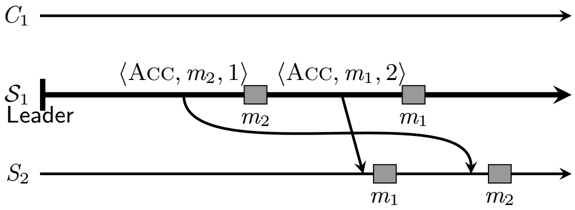
Without Clients



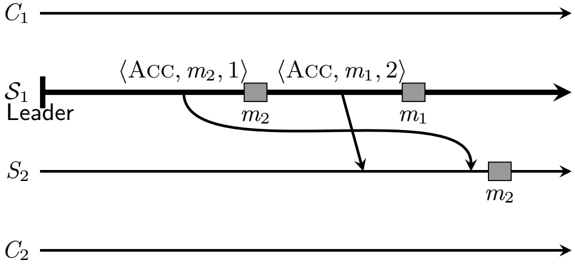
Problem: Also Accept Messages can be Reordered



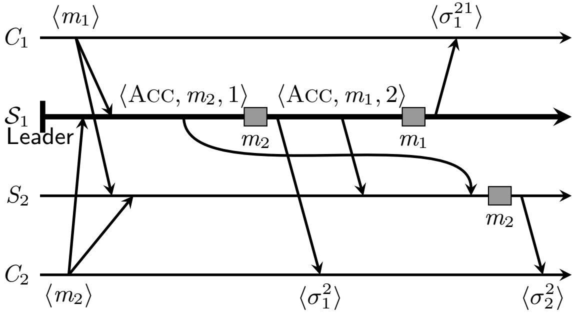
Add Sequence Numbers



Discard Out-of-Order Messages



Now with Clients



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?

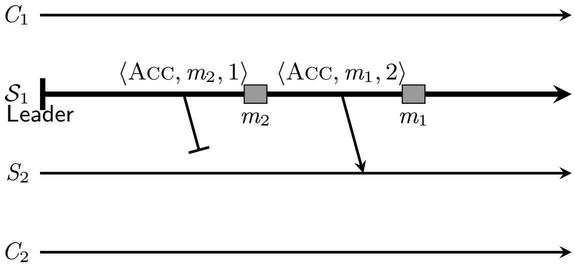
Clients Observe The Same Server States as Before

- 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

Clients Observe The Same Server States as Before

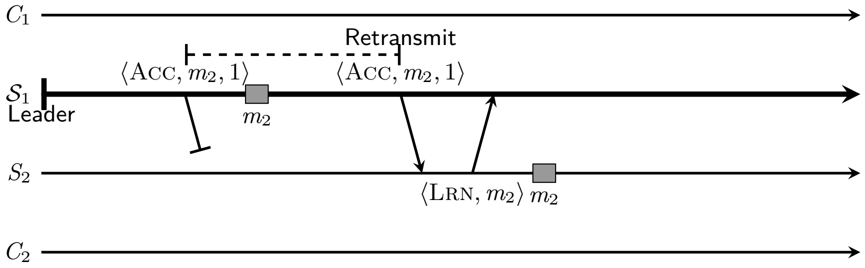
- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- However, S_2 didn't execute m_1
  - Q: What to do?
  - A1: Buffer
  - A2: Retransmission mechanism

Problem: Message Loss – S_2 Won't Execute Anything

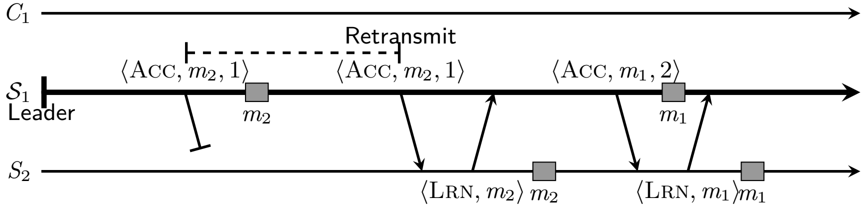


We Need a Retransmission Mechanism!

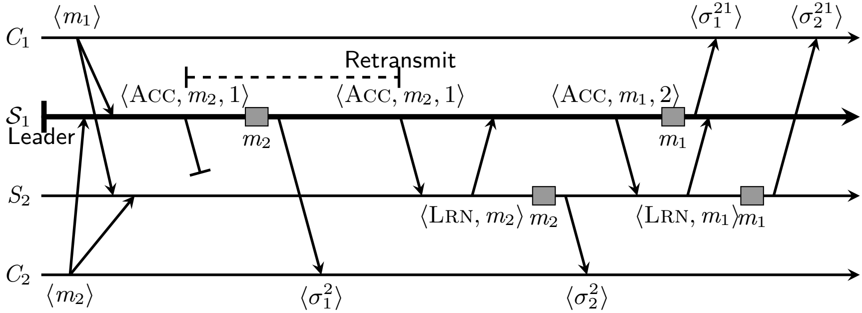
A Learn Stops Retransmission



Don't Send New Accept Until Learn



With Clients



Recap

- A leader
  - To decide the order of client requests
  - By sending an accept message to S_2

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

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

Recap

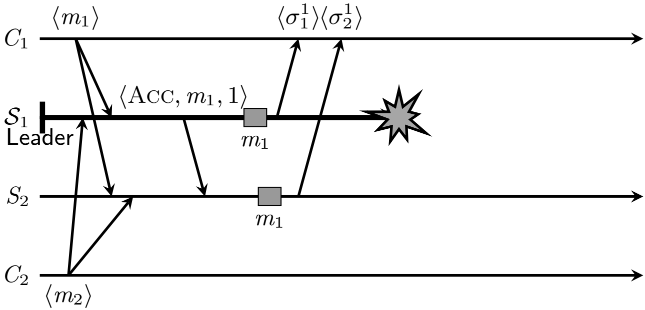
- A leader
  - To decide the order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - To cope with message reordering
- Retransmission mechanism
  - To cope with message loss
  - Leader only sends next accept when learn from S_2
  - Allows leader to *make progress*, as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

Combination of mechanisms:

RetransAccept protocol

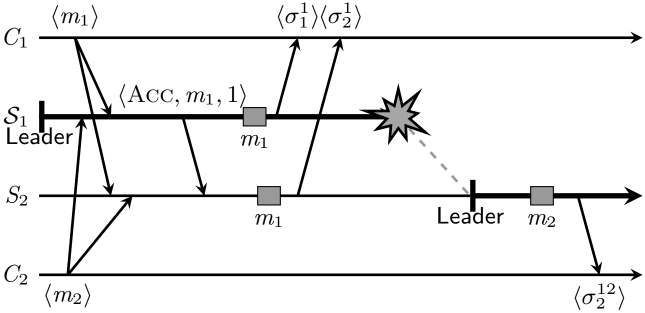
What About Server Crashes?

Crash

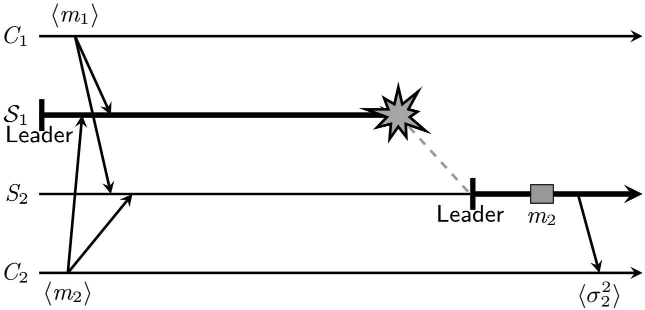


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle$
C_1
$\langle \text{ACC}, m_1, 1 \rangle$
S_1
Leader
m_1
S_2
m_1
C_2
$\langle m_2 \rangle$

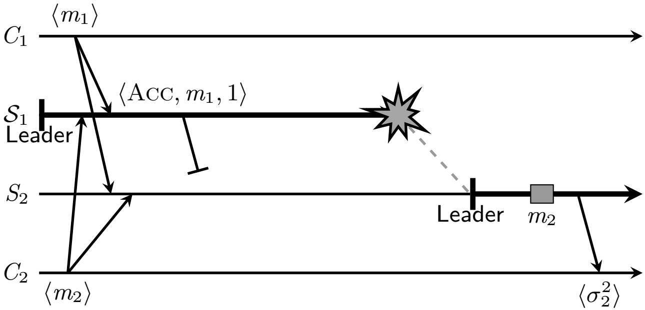
Crash: Leader Takeover



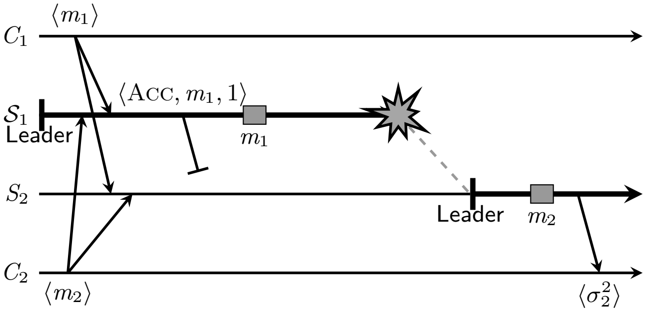
Single Server Rule: Case 1



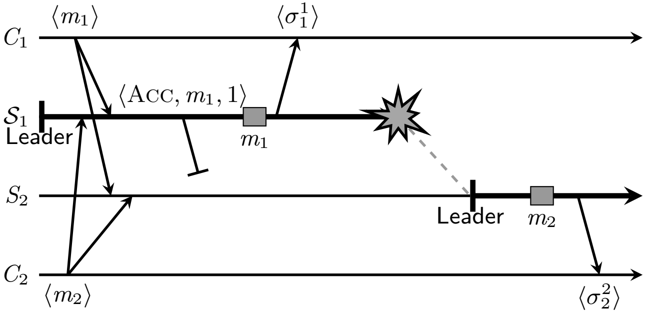
Single Server Rule: Case 2



Single Server Rule: Case 3



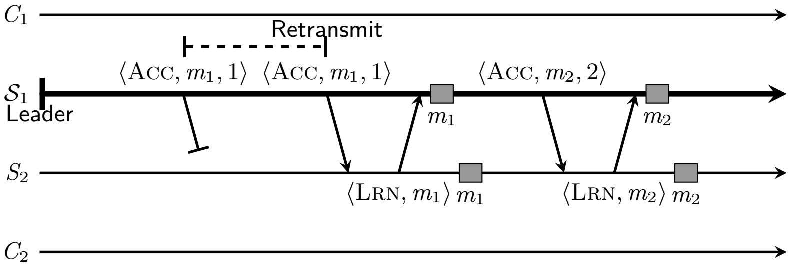
Single Server Rule: Case 4 – A Problem



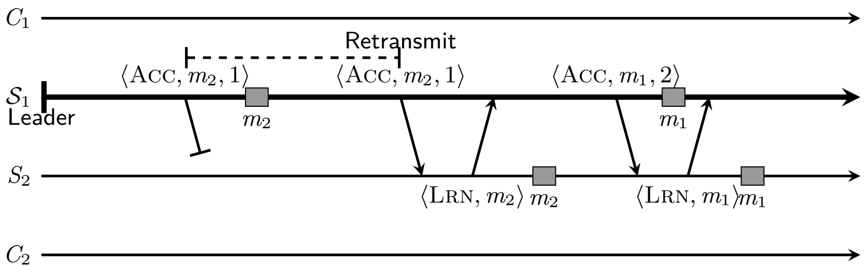
Single Server Rule: Case 4 – A Problem

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

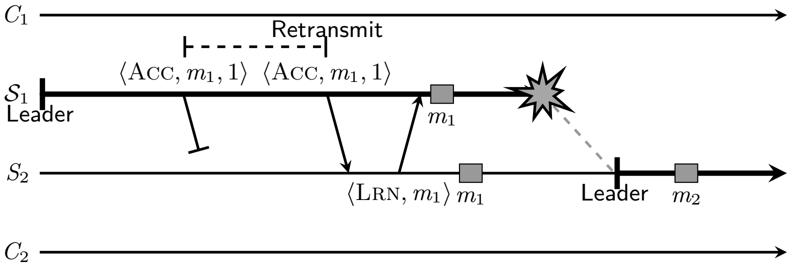
Solution: Leader Waits for Learn Before Executing



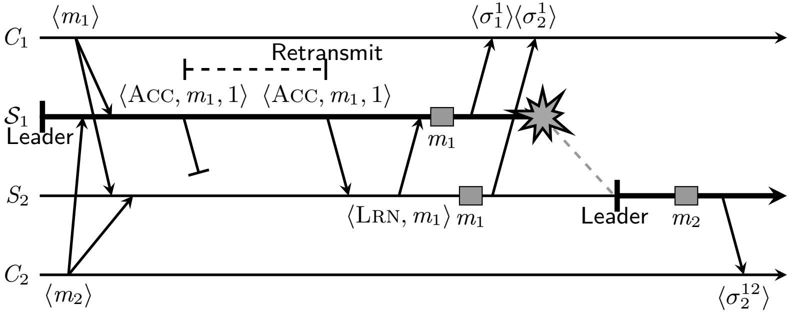
Recall Earlier Version



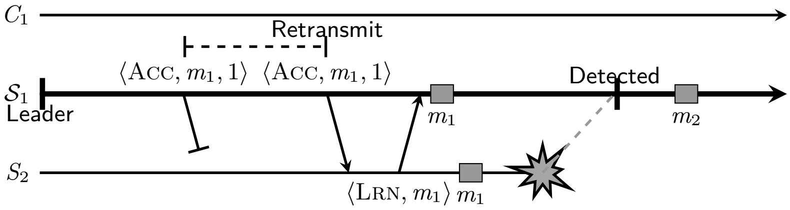
Now Leader Takeover is Safe



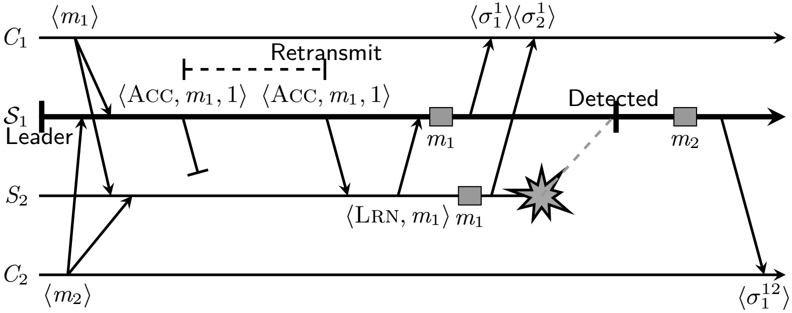
Let's Add Client Messages



Leader Remain in Control when S_2 Crash



Let's Add Client Messages Again



Recap: The Problem

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

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

Recap: WaitForLearn Protocol

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

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

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?

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.

Somewhat Rougher Road Ahead!

False Detection

- So far we have assumed that failure detection is accurate

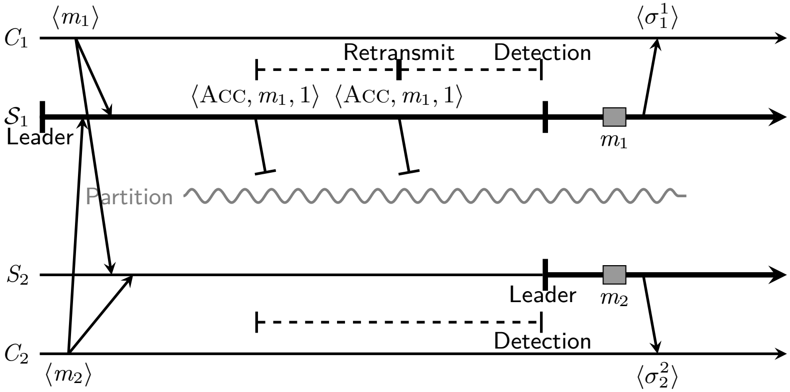
False Detection

- 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

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

Problem: Network Partitions



Network Partition

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

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

Network Partition

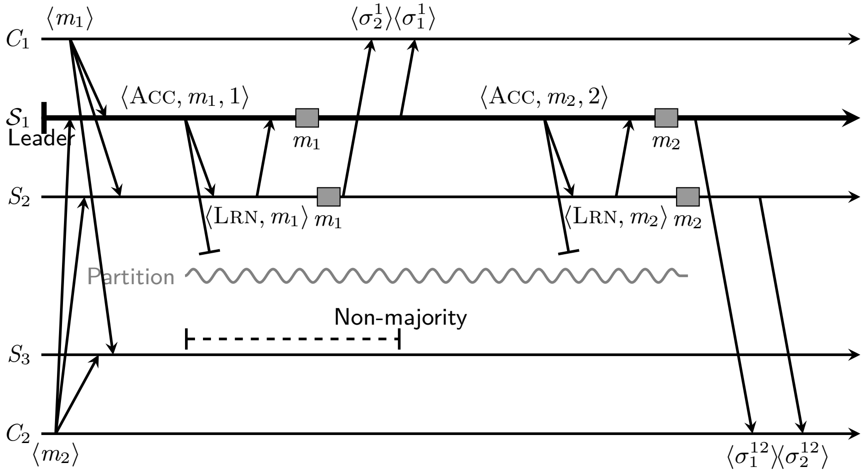
- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2
- Reconciling the state divergence
  - Involves rollback on multiple clients

Network Partition

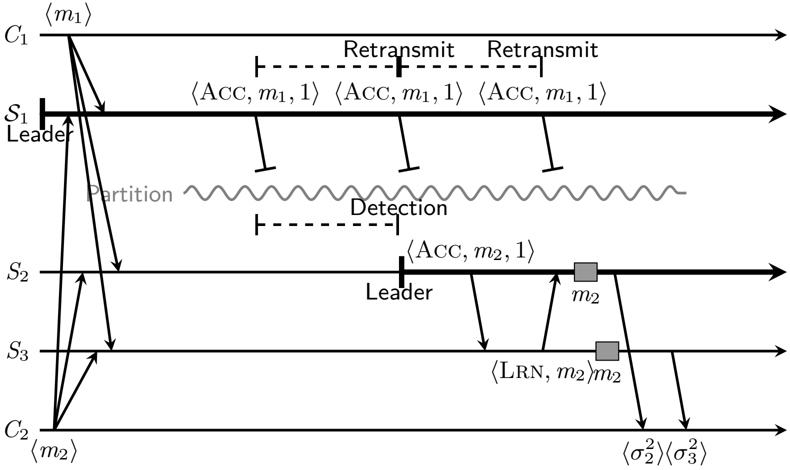
- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2
- Reconciling the state divergence
  - Involves rollback on multiple clients
  - Quickly becomes unmanageable

We Want to Avoid Relying on Clients!

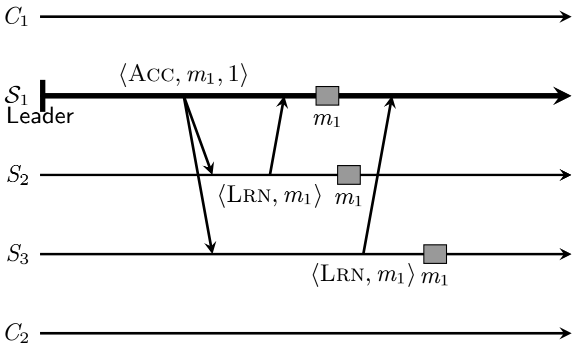
Add Another Server; Make Progress in Majority Partition



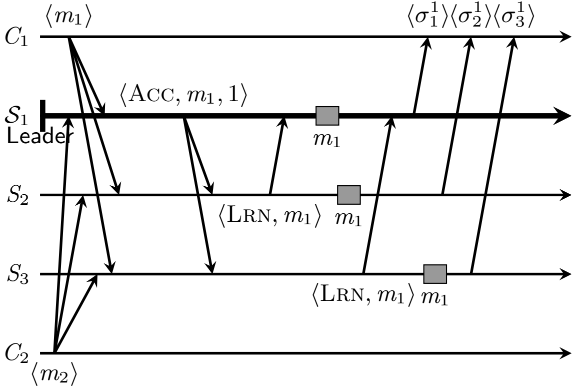
New Leader in Majority Partition



WaitForLearn Without Partition



WaitForLearn With Clients



Recap: Network Partition

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

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

Recap: Network Partition

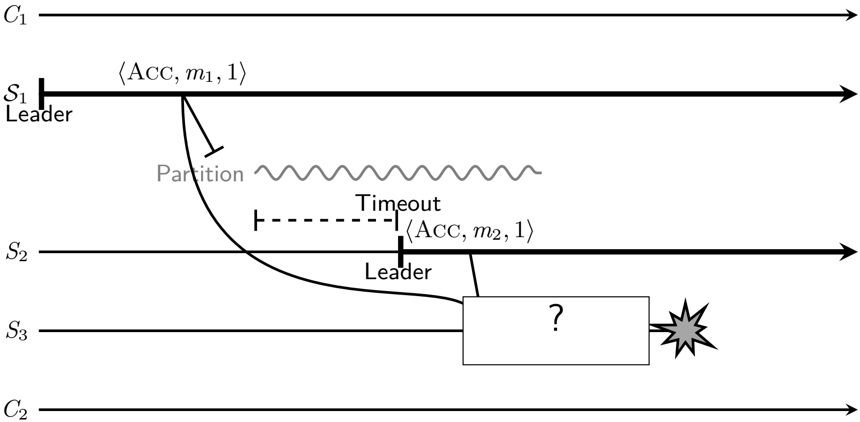
- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
  - when another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

Recap: Network Partition

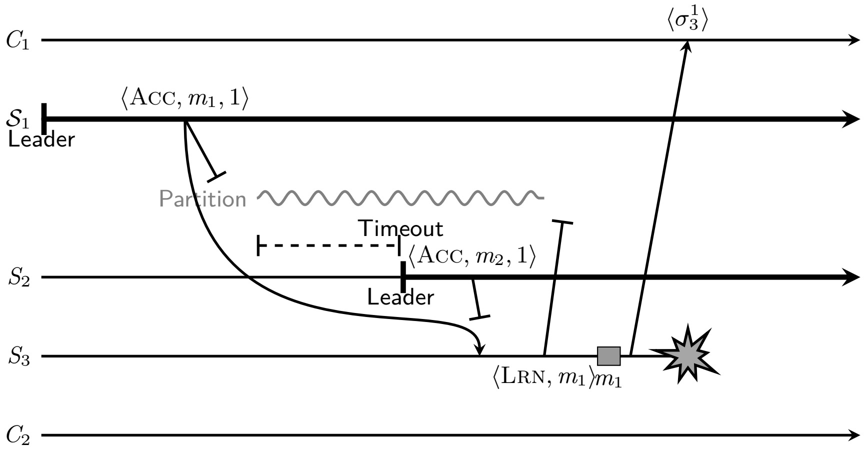
- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
  - when another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

What can go Wrong: Concurrent Crash and Partition

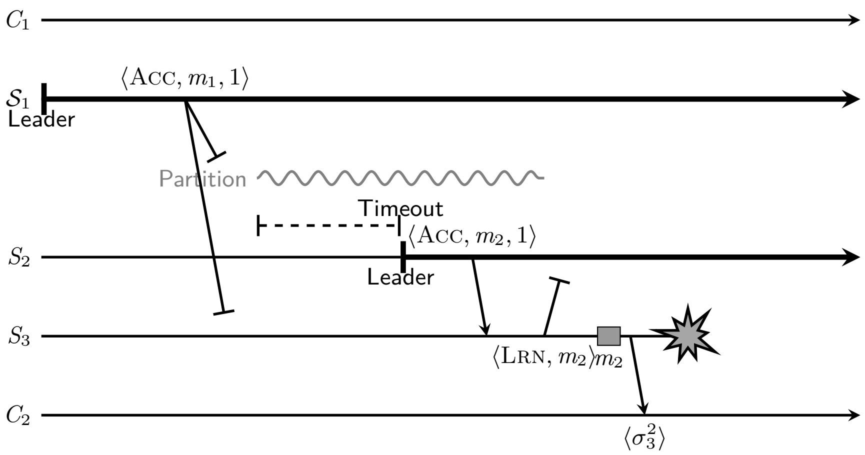
Concurrent Crash and Partition



Crash and Partition: Outcome 1 – m_1 Executed



Crash and Partition: Outcome 2 – m_2 Executed



Recap: Crash and Partition

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

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

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!

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

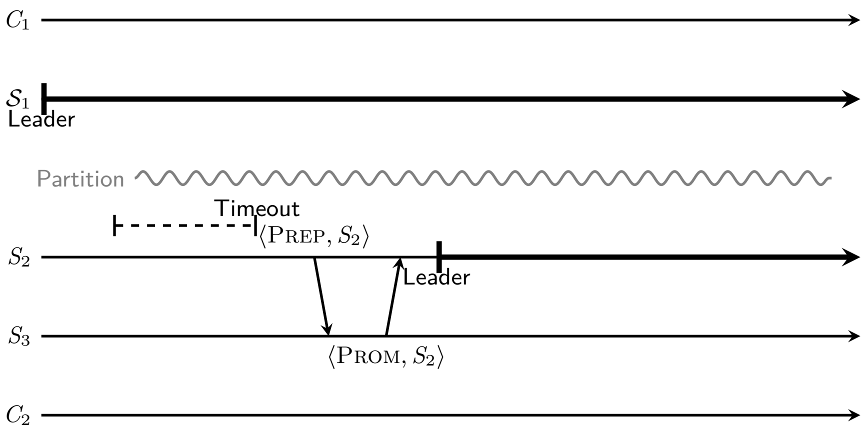
Explicit Leader Change Mechanism

- 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

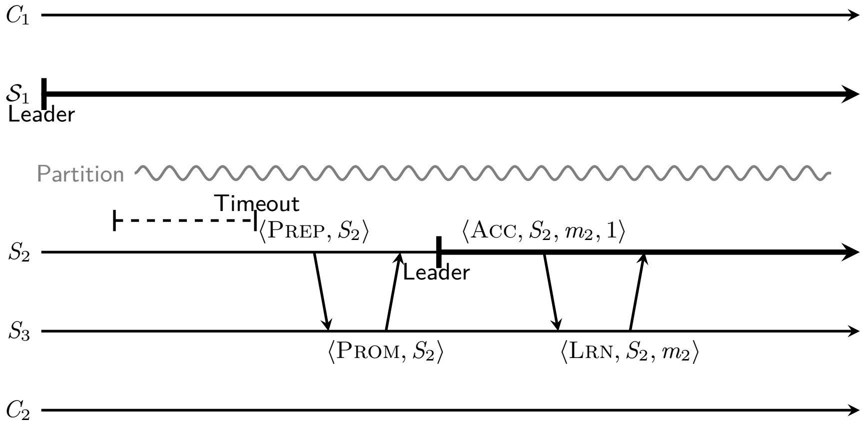
Explicit Leader Change Mechanism

- 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

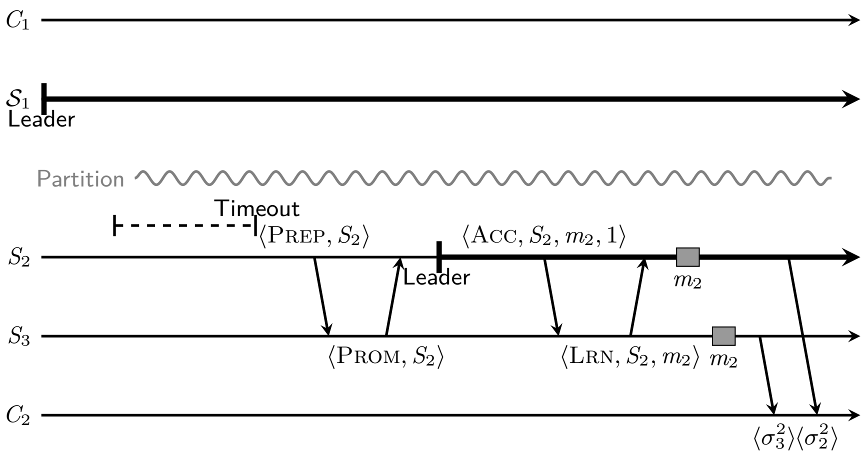
Explicit Leader Change



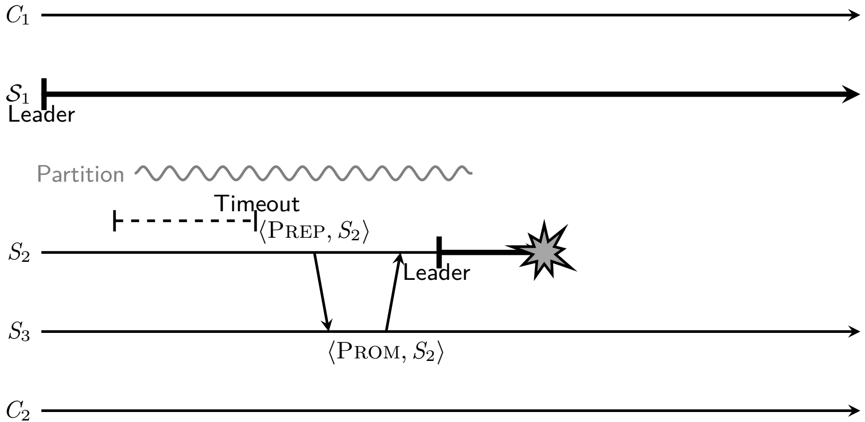
Leader Identifiers in Accept and Learn Messages



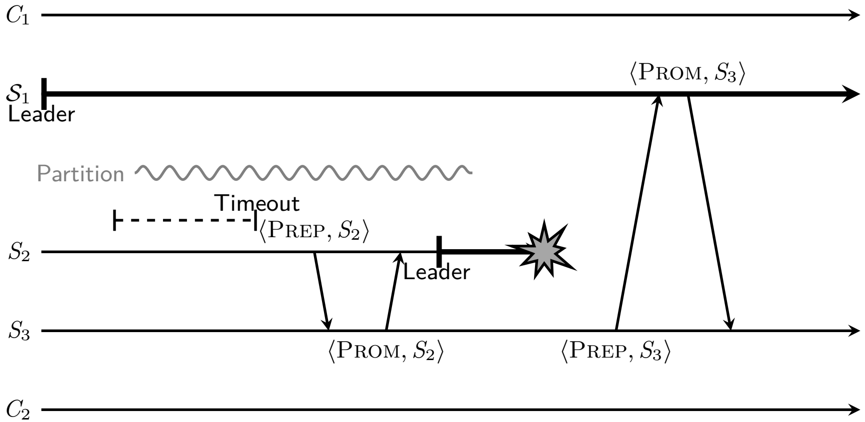
With Client Replies



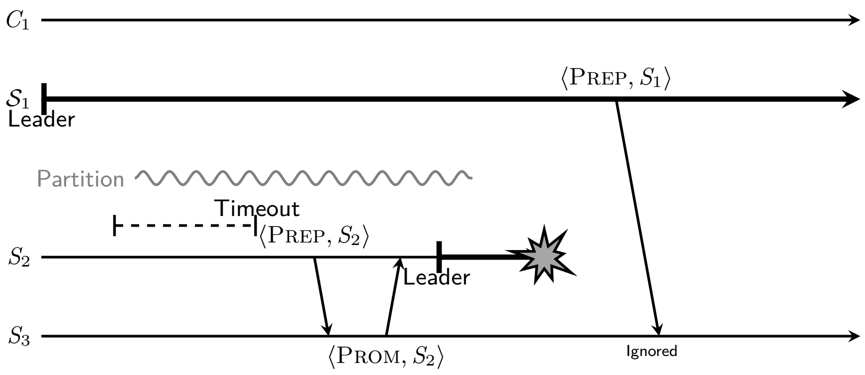
What Happens Now?



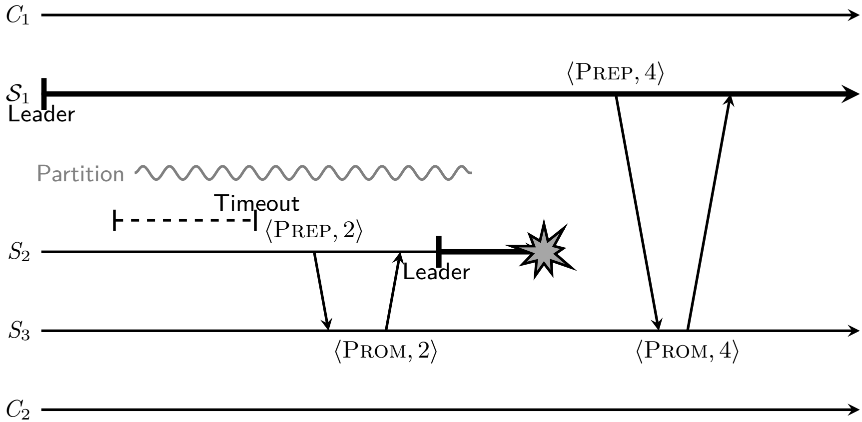
S_3 Takes Over?



S_1 Takes Over Again?



Replace Leader Identifiers With Round Numbers



Recap: Leader Change

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

Recap: Leader Change

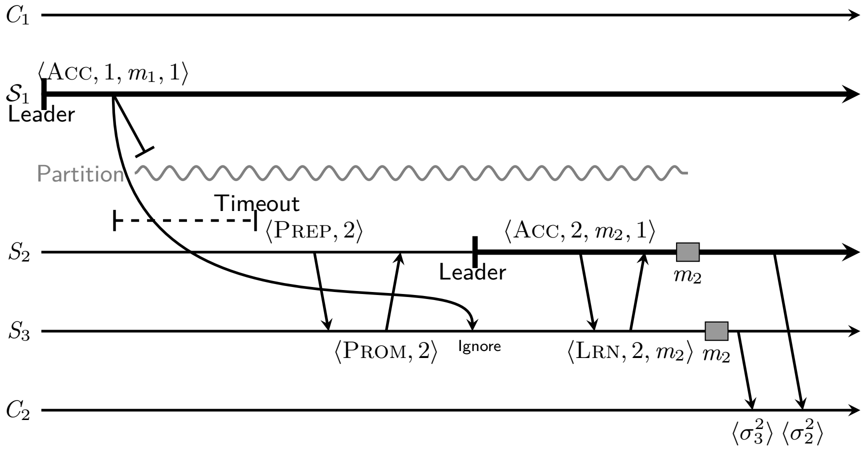
- Added round number rnd in messages
  - To identify the leader
    - $\langle \text{Acc}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \text{Lrn}, rnd, m \rangle$: Sent to leader of round rnd
  - Round numbers are assigned:
    - S_1: $1, 4, 7, \ldots$
    - S_2: $2, 5, 8, \ldots$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

Recap: Leader Change

- Added round number rnd in messages
  - To identify the leader
    - $\langle \textsc{Acc}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \textsc{Lrn}, rnd, m \rangle$: 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
  - $\langle \textsc{Prep}, rnd \rangle$: Request to become leader for round rnd
  - $\langle \textsc{Prom}, rnd \rangle$: Promise not to accept messages from a lower round than rnd (i.e. an older leader)

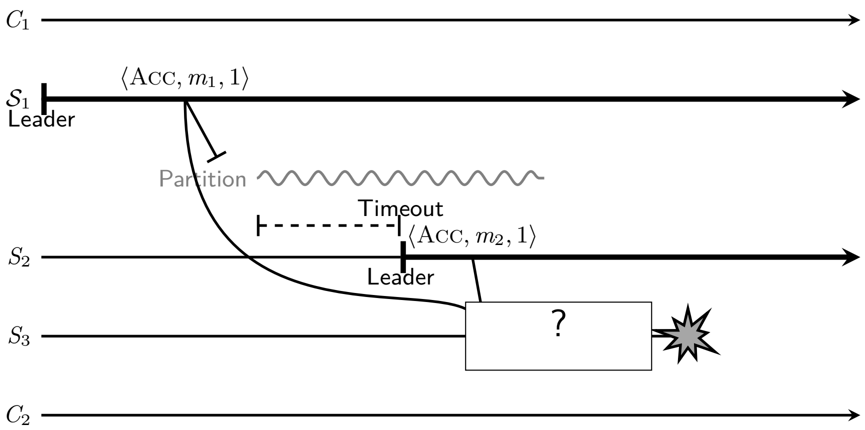
Let's Apply This Together With Accept and Learn

S_3 Ignores Accept Message From Old Leader

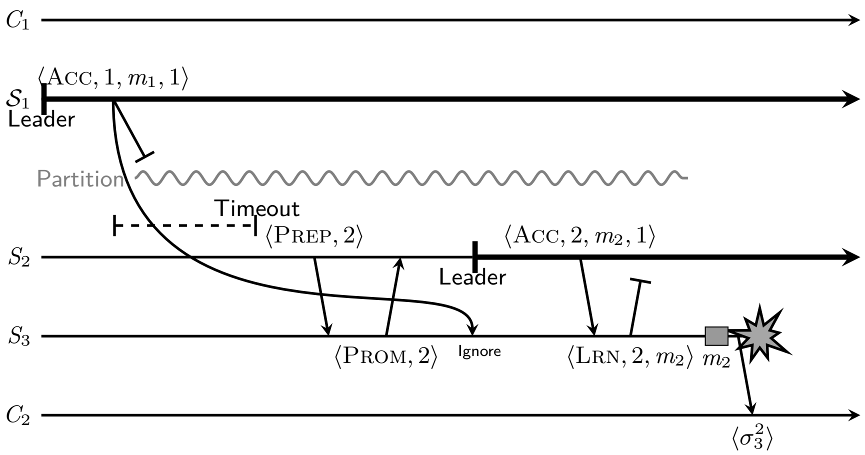


Let's Recall the Problem we are Trying to Solve

We Don't Know What S_3 Did Before Crashing



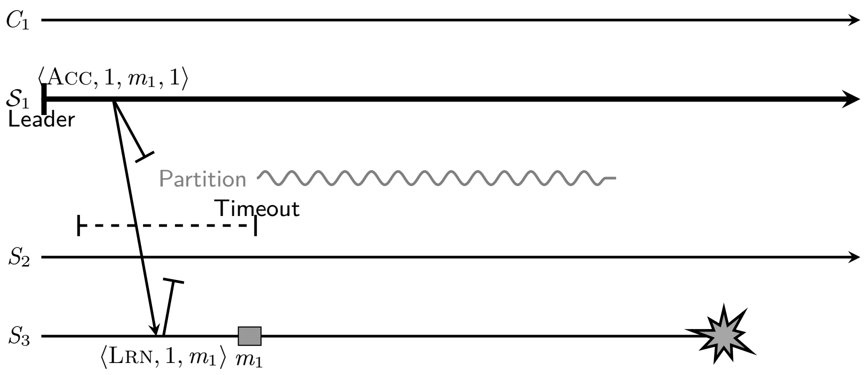
Do We Know Now?



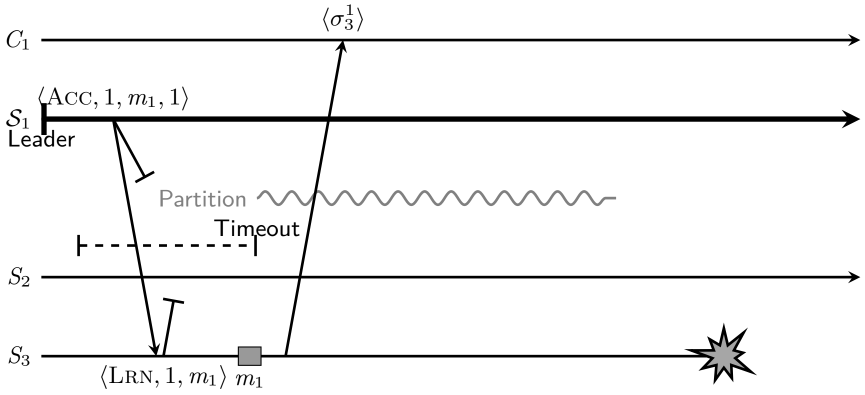
No we don't!

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

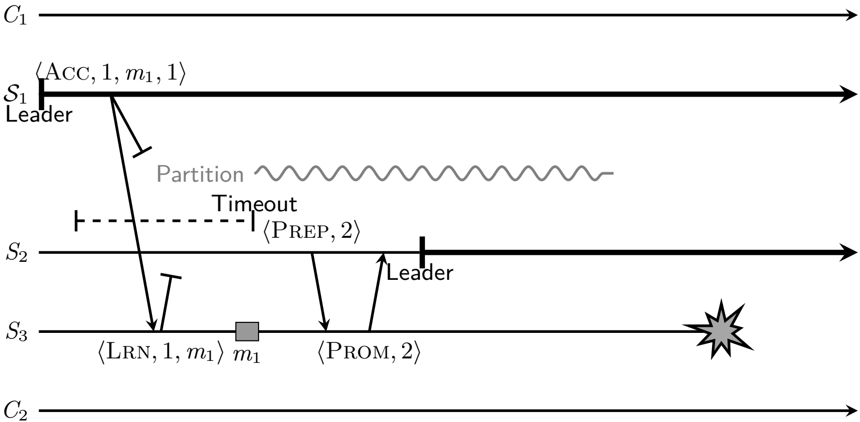
What Happens If S_3 Learn m_1?



What Happens If S_3 Learn m_1?



Does Leader Change Help?



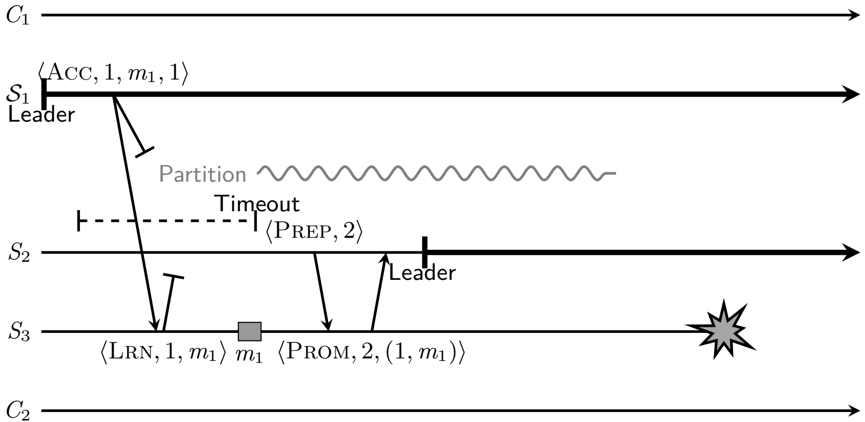
No!

We Still don't Know What

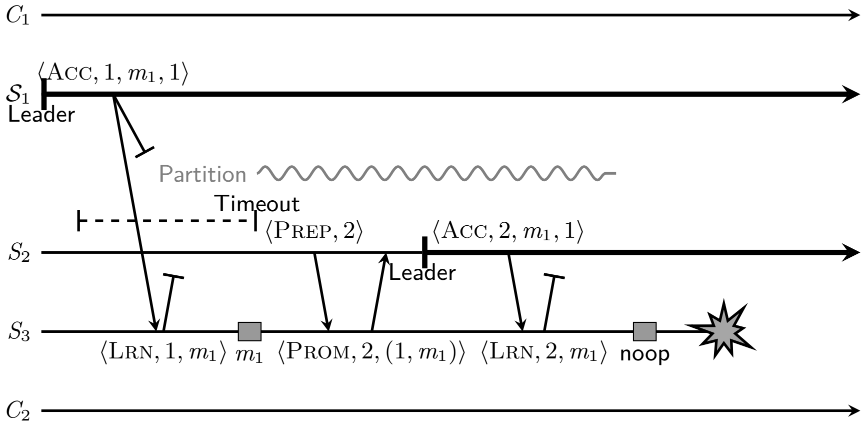
S_3 Did Before Crashing.

But the fix is Easy!

Tell new Leader About Accepted Messages

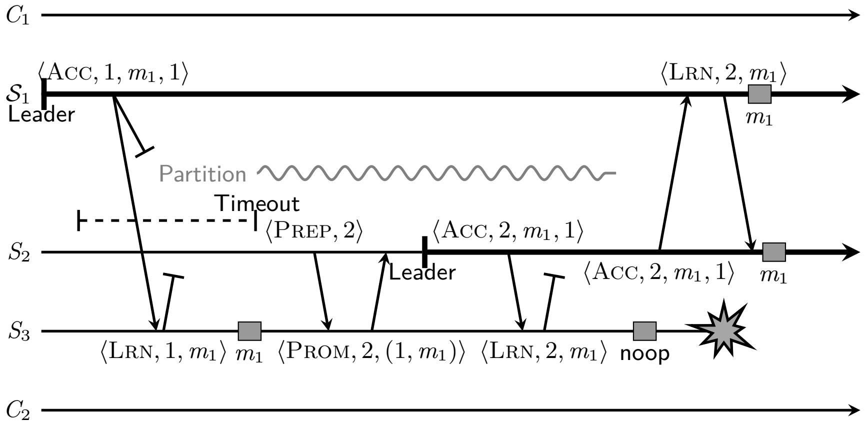


The new Leader Resends Accept for Those Messages

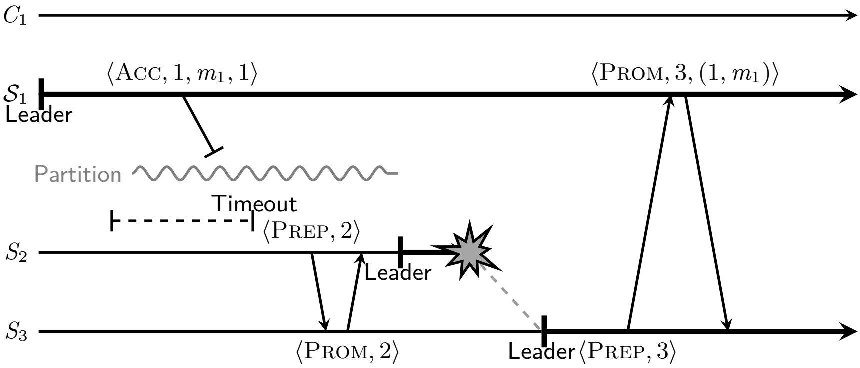


Learn was Lost and S_3 Crashed. Leader Still can't Execute m_1.

Leader Also Resends Accept After Merge



Promise from old Leader Includes Accepted Messages



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: $\langle \textsc{Prom}, rnd, (vrnd, vval) \rangle$

Recap: Leader Change 2

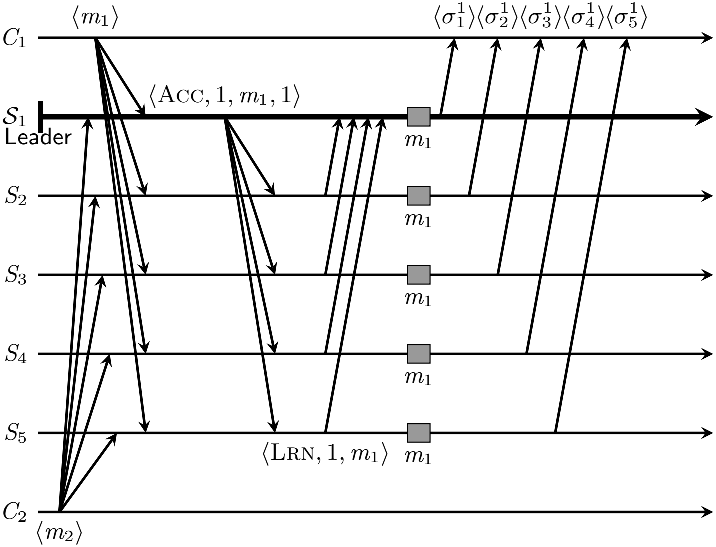
- Added information about accept from previous leader: $\langle \textsc{Prom}, rnd, (1, m_1) \rangle$
  - Promise not to accept messages from a lower round than rnd
  - Last leader did send m_1 in round 1
  - Typical naming: $\langle \textsc{Prom}, rnd, (vrnd, vval) \rangle$
- Leader resends accept for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

What About More Than one Crash?

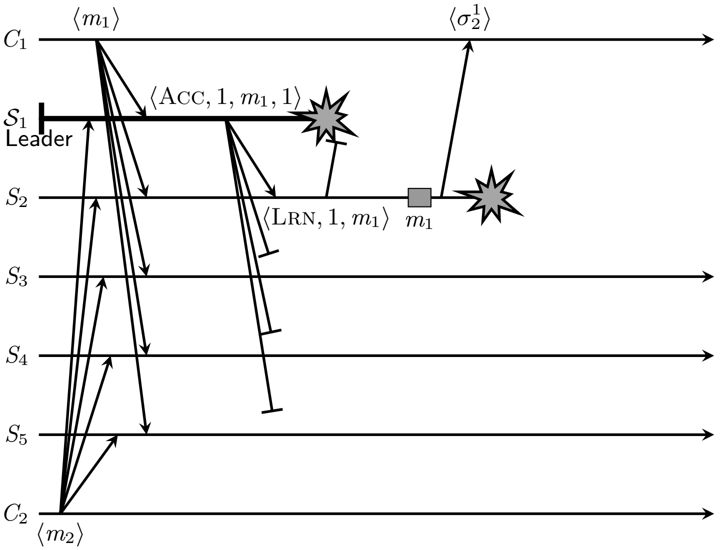
What About More Than one Crash?

- 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 $2f + 1$

With Five Servers



With Five Servers, S_2 Cannot Execute After Accept



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

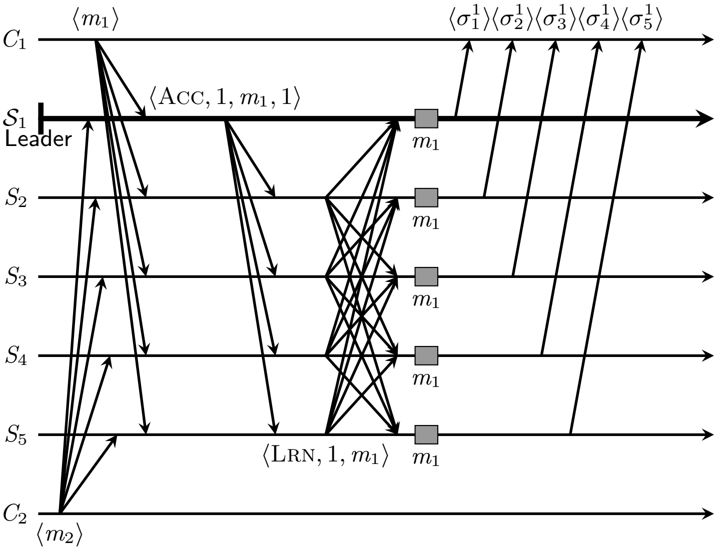
With Five Servers, S_2 Cannot Execute After Accept

- 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!

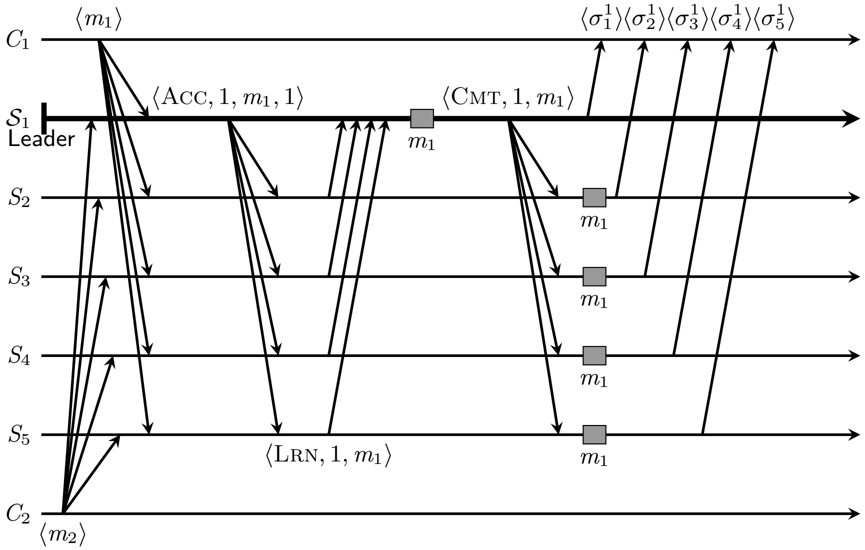
With Five Servers, S_2 Cannot Execute After Accept

- 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

All-to-All Learn Before Execute

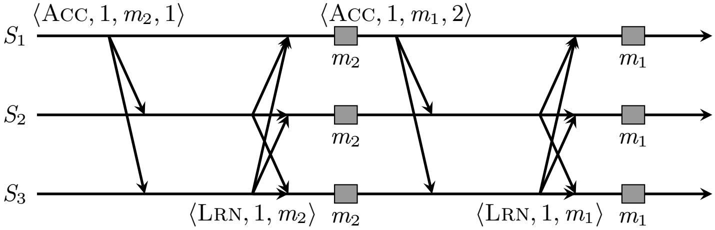


Await Commit Before Execute



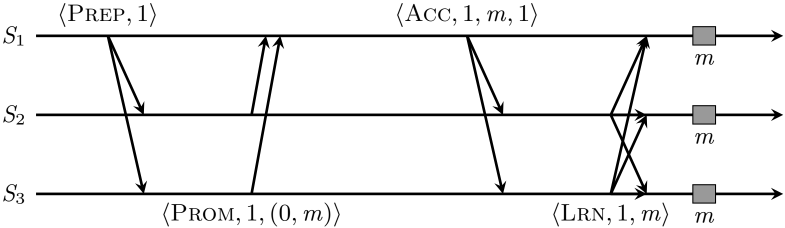
Wrapping it up!

Multi-Paxos

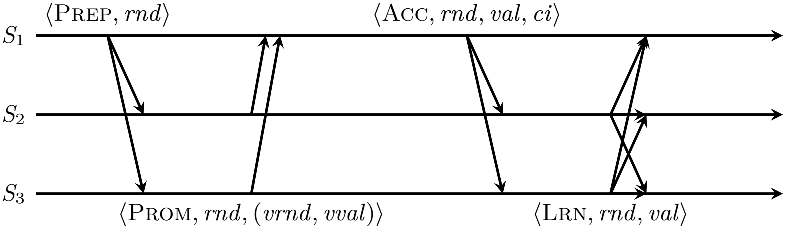


$\langle \text{Acc}, 1, m_2, 1\rangle$
$\langle \text{Acc}, 1, m_1, 2\rangle$
S_1
S_2
S_3
m_2
m_2
m_2
m_1
m_1
m_1
$\langle \text{Lrn}, 1, m_2\rangle$
$\langle \text{Lrn}, 1, m_1\rangle$

Paxos


$\langle \textsc{Prep}, 1 \rangle$
$\langle \textsc{Acc}, 1, m, 1 \rangle$
S_1
S_2
S_3
m
m
$\langle \textsc{Prom}, 1, (0, m) \rangle$
$\langle \textsc{Lrn}, 1, m \rangle$
m

Paxos



$\langle$Prep, $rnd\rangle$
$\langle$Acc, rnd, val, $ci\rangle$
S_1
S_2
S_3
$\langle$Prom, rnd, $(vrnd, vval)\rangle$
$\langle$Lrn, rnd, $val\rangle$

Paxos Agents

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

That's It! Thank You!