Paxos Explained from Scratch

Hein Meling and Leander Jehl

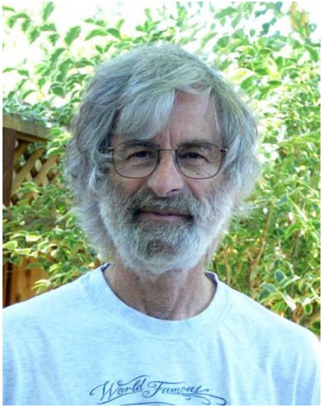
University of
Stavanger

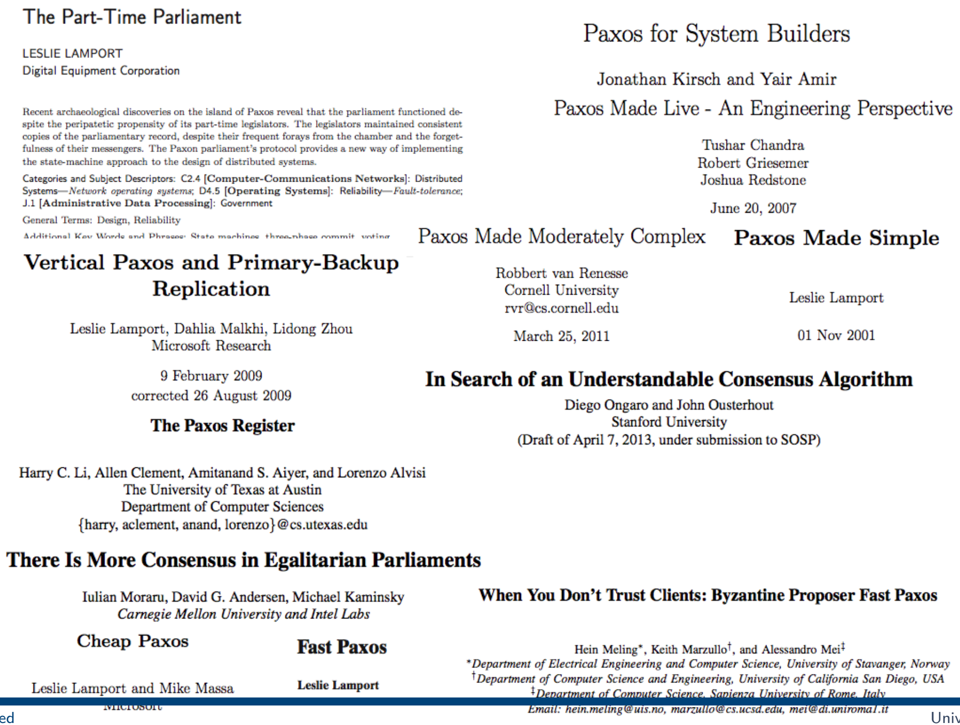
hein.meling@uis.no

DAT520 Distributed Systems 2024

Leslie Lamport

- Microsoft Research
- Many important contributions to distributed computing theory
- 2013 Turing Award winner
- But most know for LaTeX





The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [**Computer-Communications Networks**]: Distributed Systems—*Network operating systems*; D4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; J.1 [**Administrative Data Processing**]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

Paxos for System Builders

Jonathan Kirsch and Yair Amir

Paxos Made Live - An Engineering Perspective

Tushar Chandra
Robert Griesemer
Joshua Redstone

June 20, 2007

Vertical Paxos and Primary-Backup Replication

Leslie Lamport, Dahlia Malkhi, Lidong Zhou
Microsoft Research

9 February 2009
corrected 26 August 2009

Paxos Made Moderately Complex

Robbert van Renesse
Cornell University
rvr@cs.cornell.edu

March 25, 2011

Paxos Made Simple

Leslie Lamport

01 Nov 2001

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University
(Draft of April 7, 2013, under submission to SOSP)

The Paxos Register

Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
{harry, aclement, anand, lorenzo}@cs.utexas.edu

There Is More Consensus in Egalitarian Parliaments

Iulian Moraru, David G. Andersen, Michael Kaminsky
Carnegie Mellon University and Intel Labs

Cheap Paxos

Leslie Lamport and Mike Massa
Microsoft

Fast Paxos

Leslie Lamport

When You Don't Trust Clients: Byzantine Proposer Fast Paxos

Hein Meling*, Keith Marzullo†, and Alessandro Mei‡
**Department of Electrical Engineering and Computer Science, University of Stavanger, Norway*
†Department of Computer Science and Engineering, University of California San Diego, USA
‡Department of Computer Science, Sapienza University of Rome, Italy
Email: hein.meling@uis.no, marzullo@cs.ucsd.edu, mei@di.uniroma1.it

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 sent to 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 sent to 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 sent to 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
  - In a step-wise manner
  - With minimal changes in each step

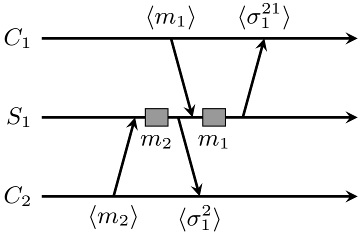
Objectives and Approach

- Explain Paxos
  - In a step-wise manner
  - With minimal changes in each step
- Objective
  - Understand why it works and why the solution is necessary
  - (not how to implement or formally prove it)

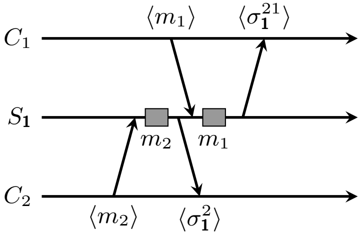
Objectives and Approach

- Explain Paxos
  - In a step-wise manner
  - With minimal changes in each step
- Objective
  - Understand why it works and why the solution is necessary
  - (not how to implement or formally prove it)
- Approach
  - Starting from a simple client/server system
  - Build fault tolerant server (replicated state machine)
  - Construct Multi-Paxos
  - Decompose Multi-Paxos into Paxos

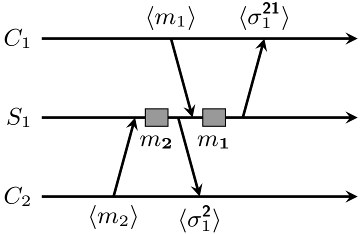
A Stateful Service: *SingleServer*



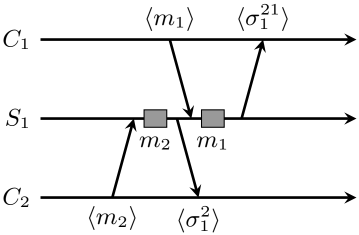
A Stateful Service: *SingleServer* (Subscript)


$\langle m_1 \rangle$
$\langle \sigma_{\mathbf{1}}^{21} \rangle$
C_1
$S_{\mathbf{1}}$
m_2
m_1
C_2
$\langle m_2 \rangle$
$\langle \sigma_{\mathbf{1}}^{2} \rangle$

A Stateful Service: *SingleServer* (Superscript)



A Stateful Service: *SingleServer*



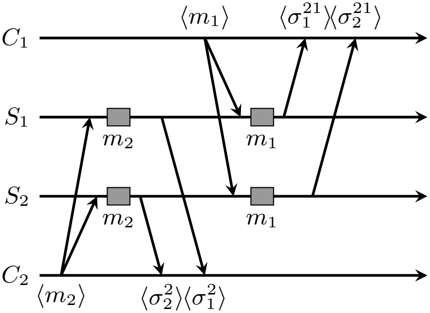
- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
    - σ^2 is a prefix of σ^{21}
- Corresponds to execution sequence: $m_2 m_1$

We Want to Make the Service Fault Tolerant!

Assumptions

- Asynchronous environment: No bounds on
  - Processing delay
  - Communication delay
  - Clock drift
  - Unreliable failure detectors
- Unreliable communication
  - May take arbitrarily long to deliver msg
  - Msgs can be duplicated and lost
- Deterministic operations
- Processes may crash

Fault Tolerance with Two Servers

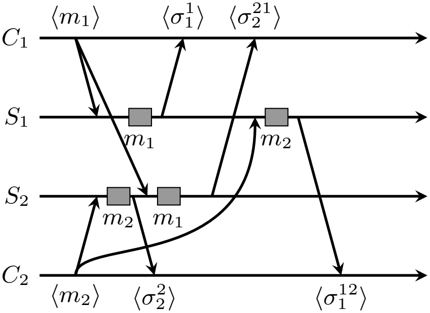

C_1
S_1
S_2
C_2
$\langle m_1 \rangle$
$\langle \sigma_1^{21} \rangle \langle \sigma_2^{21} \rangle$
m_2
m_1
m_2
m_1
$\langle m_2 \rangle$
$\langle \sigma_2^{2} \rangle \langle \sigma_1^{2} \rangle$

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

Deterministic State Machine

- Service implemented as a deterministic state machine
- Processing requests yield unique state transitions:
  - $\sigma_1^2 = \sigma_2^2$ and $\sigma_1^{21} = \sigma_2^{21}$.
- Clients suppress duplicate replies

Fault Tolerance with Two Servers: Whoops!

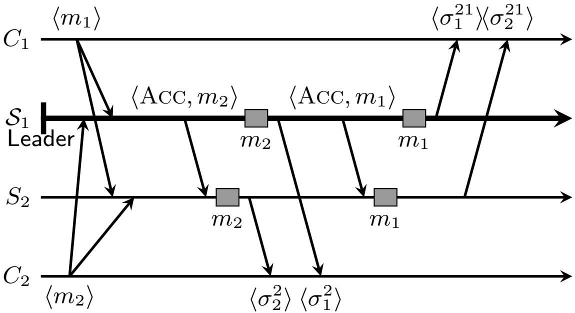


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle$
$\langle \sigma_2^{21} \rangle$
C_1
S_1
m_1
m_2
S_2
m_2
m_1
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^2 \rangle$
$\langle \sigma_1^{12} \rangle$

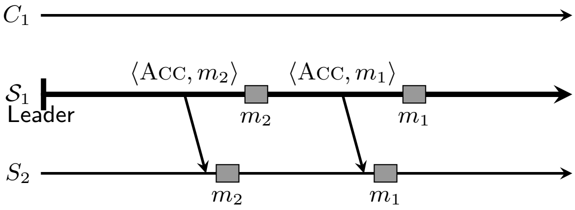
- Client C_2 sees: $\sigma^2\sigma^{12}$
  - σ^2 not a prefix of σ^{12}
- Client C_1 sees: $\sigma^1\sigma^{21}$
  - σ^1 not a prefix of σ^{21}
- Execution sequence
  - S_1: $m_1 m_2$
  - S_2: $m_2 m_1$

We Need to Order Client Requests!

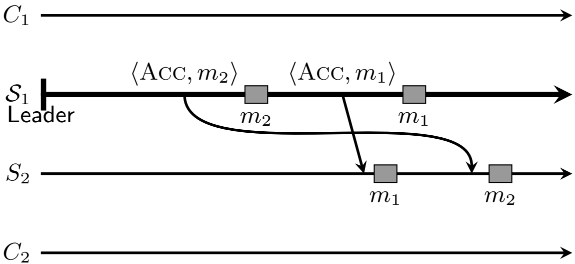
Let's Designate a Leader to Order Requests


$\langle m_1 \rangle$
$\langle \sigma_1^{21} \rangle \langle \sigma_2^{21} \rangle$
C_1
$\langle \text{Acc}, m_2 \rangle$
$\langle \text{Acc}, m_1 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_2
m_1
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^2 \rangle$ $\langle \sigma_1^2 \rangle$

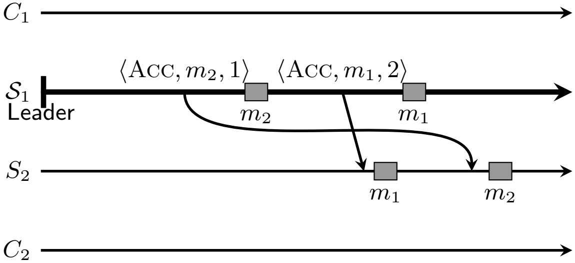
Without Clients


C_1
$\langle \textsc{Acc}, m_2 \rangle$
$\langle \textsc{Acc}, m_1 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_2
m_1
C_2

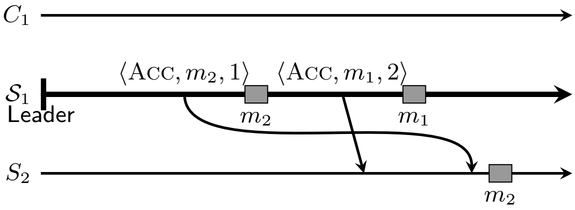
Problem: Also Accept Messages can be Reordered


C_1
$\langle \text{Acc}, m_2 \rangle$
$\langle \text{Acc}, m_1 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_1
m_2
C_2

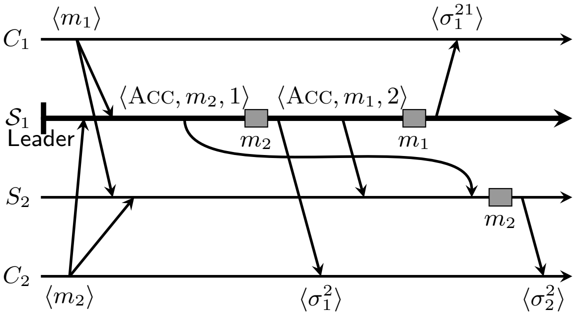
Add Sequence Numbers


C_1
$\langle \text{Acc}, m_2, 1\rangle$
$\langle \text{Acc}, m_1, 2\rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_1
m_2
C_2

Discard Out-of-Order Messages


C_1
$\langle \textsc{Acc}, m_2, 1 \rangle$
$\langle \textsc{Acc}, m_1, 2 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_2

Now with Clients


$\langle m_1 \rangle$
$\langle \sigma_1^{21} \rangle$
C_1
$\langle \text{ACC}, m_2, 1 \rangle$
$\langle \text{ACC}, m_1, 2 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
m_2
C_2
$\langle m_2 \rangle$
$\langle \sigma_1^2 \rangle$
$\langle \sigma_2^2 \rangle$

Clients Observe The Same Server States as Before

- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}

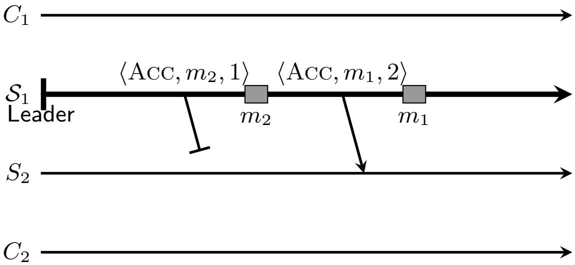
Clients Observe The Same Server States as Before

- 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

Problem: Message Loss – S_2 Won't Execute Anything

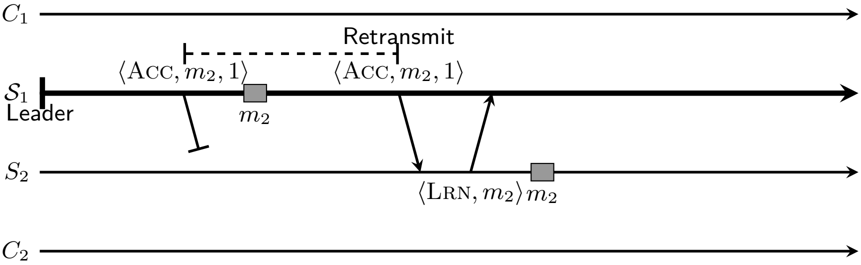

C_1
$\langle \textsc{Acc}, m_2, 1 \rangle$
$\langle \textsc{Acc}, m_1, 2 \rangle$
S_1
Leader
m_2
m_1
S_2
C_2

Clients Observe The Same Server States as Before

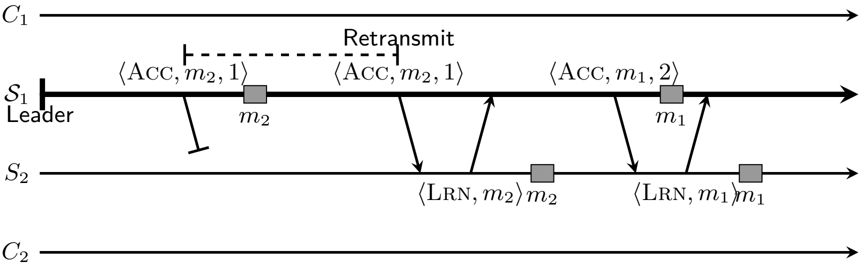
- Client C_2 sees: σ^2
- Client C_1 sees: σ^{21}
- However, S_2 didn't execute m_1
  - Q: What to do?
  - A1: Buffer
  - A2: Retransmission mechanism

We Need a Retransmission Mechanism!

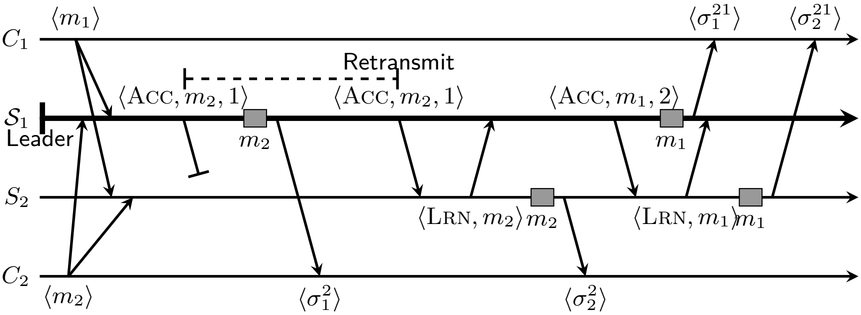
A Learn Stops Retransmission


C_1
Retransmit
$\langle \text{Acc}, m_2, 1\rangle$
$\langle \text{Acc}, m_2, 1\rangle$
$\mathcal{S}_1$
Leader
m_2
S_2
$\langle \text{Lrn}, m_2\rangle$
m_2
C_2

Don’t Send New Accept Until Learn


C_1
Retransmit
$\langle \textsc{Acc}, m_2, 1\rangle$
$\langle \textsc{Acc}, m_2, 1\rangle$
$\langle \textsc{Acc}, m_1, 2\rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
$\langle \textsc{Lrn}, m_2\rangle$
m_2
$\langle \textsc{Lrn}, m_1\rangle$
m_1
C_2

With Clients


$\langle m_1 \rangle$
$\langle \sigma_1^{21} \rangle$
$\langle \sigma_2^{21} \rangle$
C_1
Retransmit
$\langle \textsc{Acc}, m_2, 1 \rangle$
$\langle \textsc{Acc}, m_2, 1 \rangle$
$\langle \textsc{Acc}, m_1, 2 \rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
$\langle \textsc{Lrn}, m_2 \rangle$
m_2
$\langle \textsc{Lrn}, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$
$\langle \sigma_1^2 \rangle$
$\langle \sigma_2^2 \rangle$

Recap

- A leader
  - Decides order of client requests
  - By sending an accept message to S_2

Recap

- A leader
  - Decides order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - Cope with message reordering

Recap

- A leader
  - Decides order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - Cope with message reordering
- Retransmission mechanism
  - Cope with message loss
  - Leader only sends next accept when learn from S_2

Recap

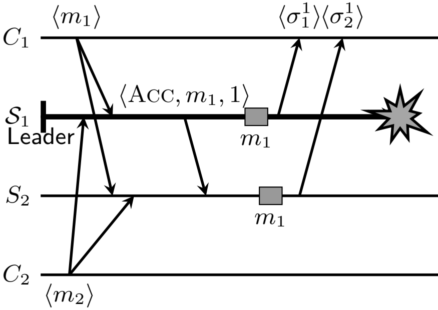
- A leader
  - Decides order of client requests
  - By sending an accept message to S_2
- Sequence numbers
  - Cope with message reordering
- Retransmission mechanism
  - Cope with message loss
  - Leader only sends next accept when learn from S_2

Combination of mechanisms:

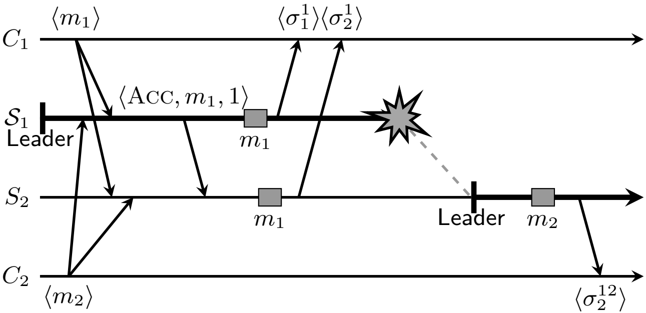
RetransAccept protocol

What About Server Crashes?

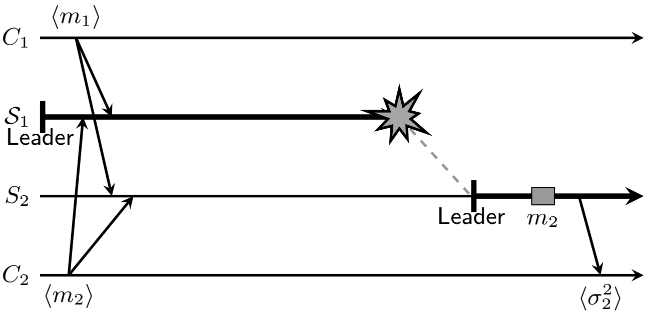
Crash


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle$
C_1
$\langle \text{Acc}, m_1, 1 \rangle$
$\mathcal{S}_1$
Leader
m_1
S_2
m_1
C_2
$\langle m_2 \rangle$

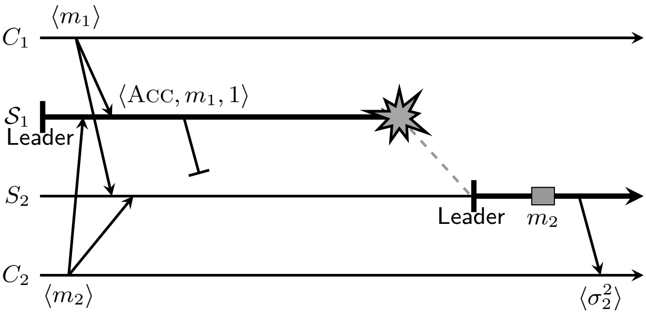
Crash: Leader Takeover


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle$
C_1
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\mathcal{S}_1$
Leader
m_1
S_2
m_1
Leader
m_2
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^{12} \rangle$

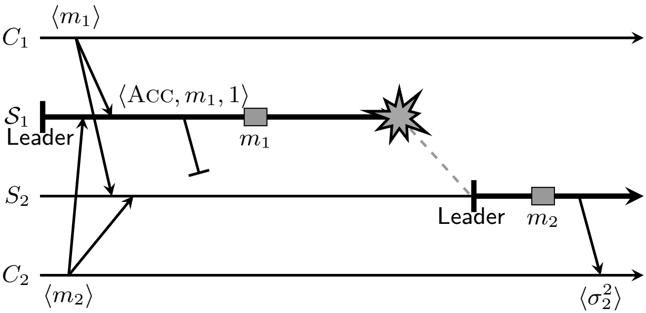
Single Server Rule: Case 1



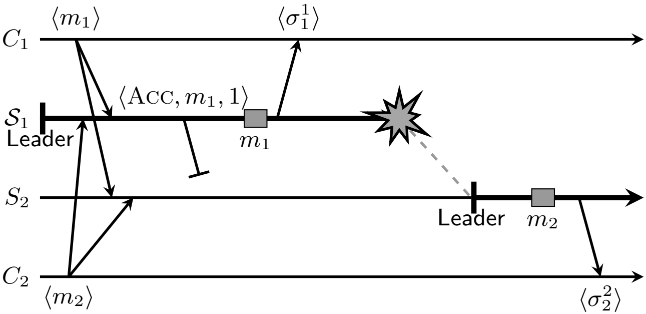
Single Server Rule: Case 2



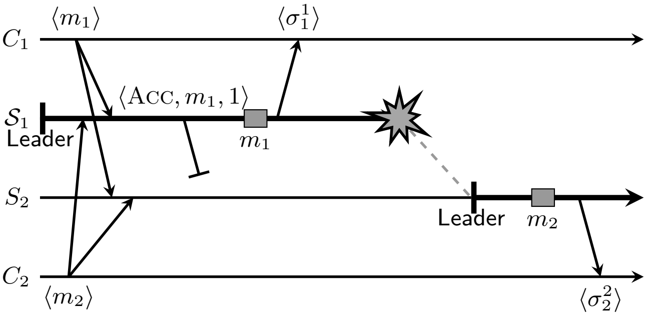
Single Server Rule: Case 3



Single Server Rule: Case 4 – A Problem



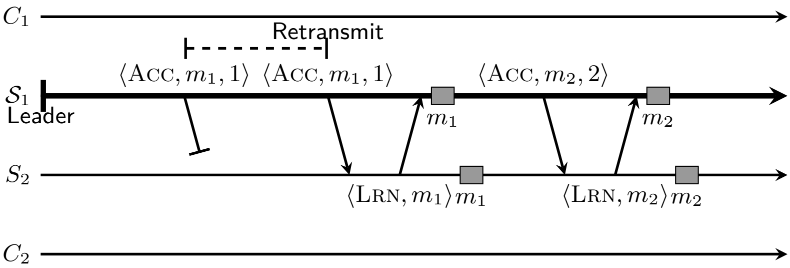
Lock Service: Both clients get the lock



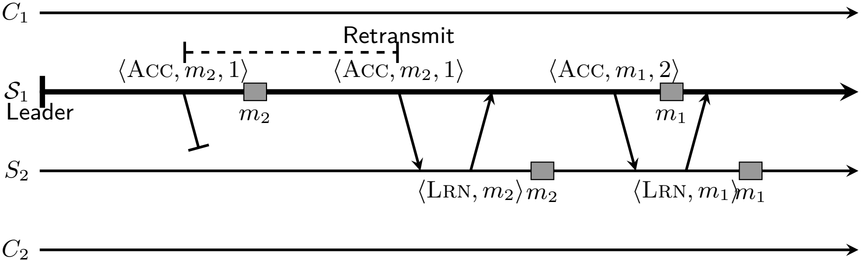
Single Server Rule: Case 4 – A Problem

- Imagine that (S_1, S_2) is a fault tolerant lock service
- Both clients could have gotten the lock

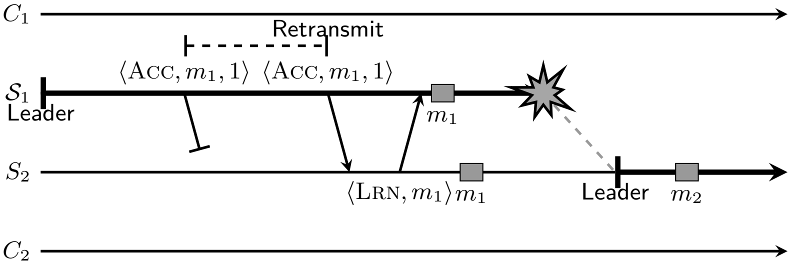
Solution: Leader Waits for Learn Before Executing


C_1
Retransmit
$\langle \textsc{Acc}, m_1, 1\rangle$
$\langle \textsc{Acc}, m_1, 1\rangle$
$\langle \textsc{Acc}, m_2, 2\rangle$
S_1
Leader
m_1
m_2
S_2
$\langle \textsc{Lrn}, m_1\rangle$
m_1
$\langle \textsc{Lrn}, m_2\rangle$
m_2
C_2

Recall Earlier Version

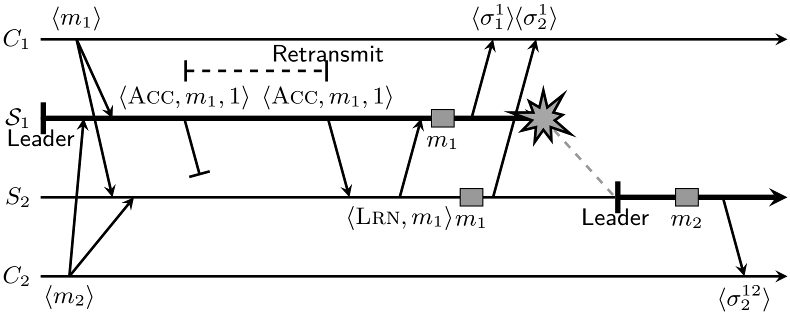

C_1
Retransmit
$\langle \textsc{Acc}, m_2, 1\rangle$
$\langle \textsc{Acc}, m_2, 1\rangle$
$\langle \textsc{Acc}, m_1, 2\rangle$
$\mathcal{S}_1$
Leader
m_2
m_1
S_2
$\langle \textsc{Lrn}, m_2\rangle$
m_2
$\langle \textsc{Lrn}, m_1\rangle$
m_1
C_2

Now Leader Takeover is Safe

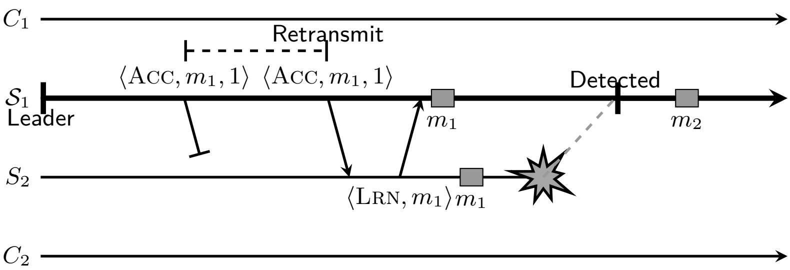


C_1
Retransmit
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\mathcal{S}_1$
Leader
m_1
S_2
$\langle \textsc{Lrn}, m_1 \rangle$
m_1
Leader
m_2
C_2

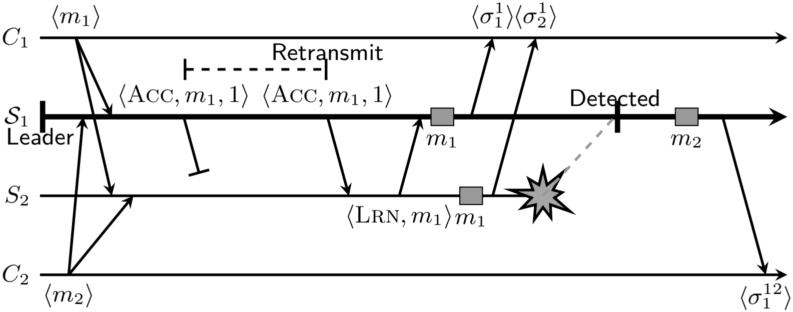
Let's Add Client Messages


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle$
C_1
Retransmit
$\langle \text{Acc}, m_1, 1 \rangle$
$\langle \text{Acc}, m_1, 1 \rangle$
S_1
Leader
m_1
S_2
$\langle \text{Lrn}, m_1 \rangle$
m_1
Leader
m_2
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^{12} \rangle$

Leader Remain in Control when S_2 Crash


C_1
Retransmit
$\langle \text{Acc}, m_1, 1\rangle$
$\langle \text{Acc}, m_1, 1\rangle$
Detected
S_1
Leader
m_1
m_2
S_2
$\langle \text{Lrn}, m_1\rangle$
m_1
C_2

Let's Add Client Messages Again



$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle$
C_1
Retransmit
$\langle \text{Acc}, m_1, 1 \rangle$
$\langle \text{Acc}, m_1, 1 \rangle$
Detected
$\mathcal{S}_1$
Leader
m_1
m_2
S_2
$\langle \text{Lrn}, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$
$\langle \sigma_1^{12} \rangle$

Recap: The Problem

- When we detect a server crash
  - Switch to *SingleServer* mode

Recap: The Problem

- When we detect a server crash
  - Switch to *SingleServer* mode
- Problem with *RetransAccept* protocol:
  - Leader might have replied to a client and then crashed, without ensuring that S_2 saw the accept

Recap: The Problem

- When we detect a server crash
  - Switch to *SingleServer* mode
- Problem with *RetransAccept* protocol:
  - 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

- Leader waits for a learn from S_2

Recap: WaitForLearn Protocol

- Leader waits for a learn from S_2
- S_2 can execute after seeing an accept from the leader
  - Because the accept is also an implicit learn

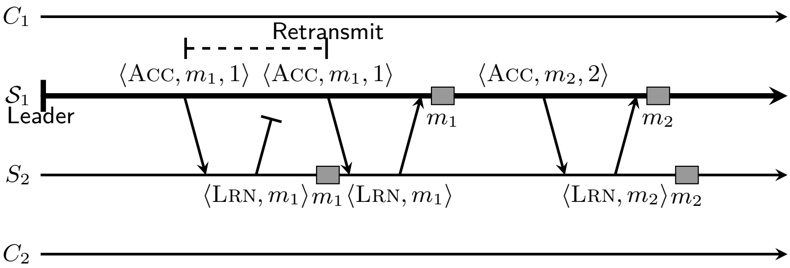
Question

- Q: What happens if the learn message to the leader is lost?

Question

- 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.

If the Learn is Lost, Retrans will fix it


C_1
Retransmit
$\langle \text{Acc}, m_1, 1\rangle$
$\langle \text{Acc}, m_1, 1\rangle$
$\langle \text{Acc}, m_2, 2\rangle$
$\mathcal{S}_1$
Leader
m_1
m_2
S_2
$\langle \text{Lrn}, m_1\rangle$
m_1
$\langle \text{Lrn}, m_1\rangle$
$\langle \text{Lrn}, m_2\rangle$
m_2
C_2

Somewhat Rougher Road Ahead!

False Detection

- So far we have assumed that failure detection is accurate

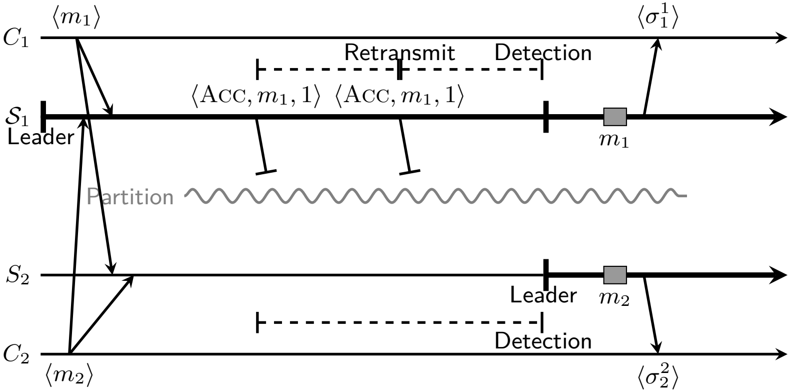
False Detection

- 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


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle$
C_1
Retransmit
Detection
$\langle \text{Acc}, m_1, 1 \rangle$
$\langle \text{Acc}, m_1, 1 \rangle$
S_1
Leader
m_1
Partition
S_2
Leader
m_2
Detection
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^2 \rangle$

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

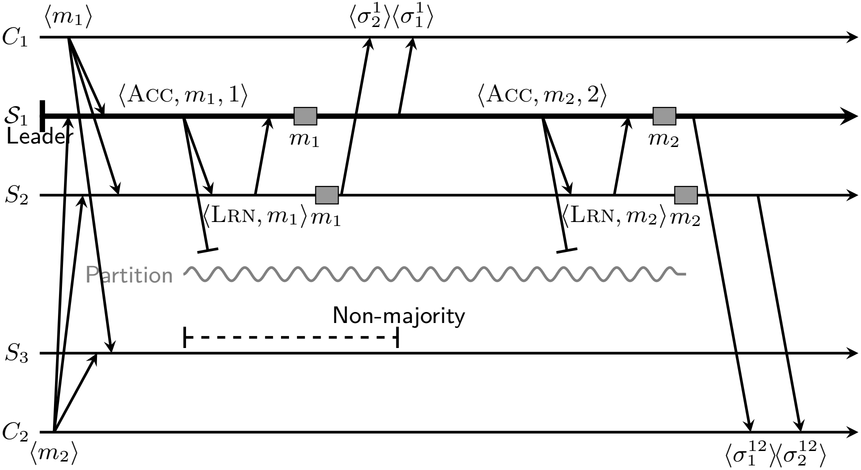
- Each server can switch to *SingleServer* mode (no coordination) and make progress
- But it will lead to inconsistencies
  - S_1 has state σ^1
  - S_2 has state σ^2
- Reconciling the state divergence
  - Involves rollback on multiple clients

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

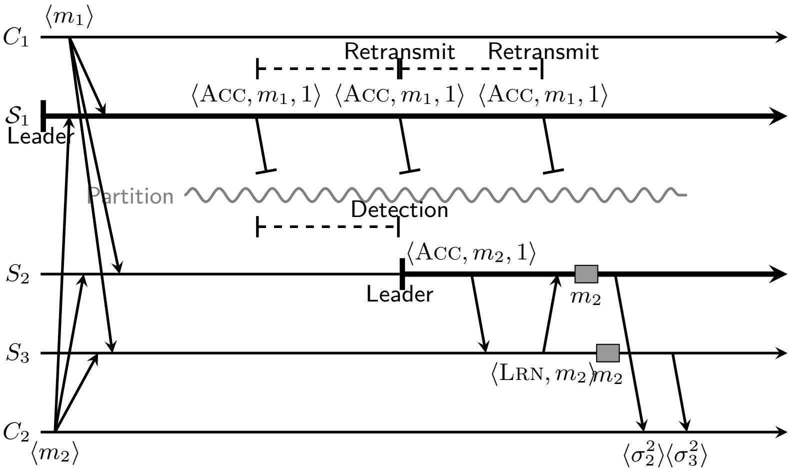
We Want to Avoid Relying on Clients!

Add Another Server; Make Progress in Majority Partition

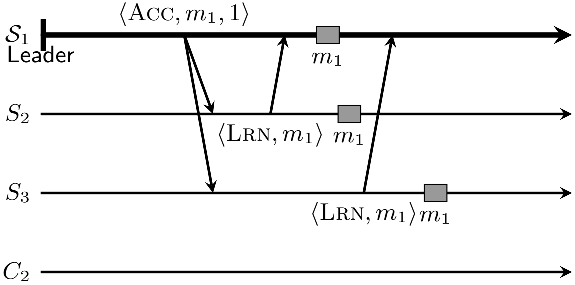


$\langle m_1 \rangle$
$\langle \sigma_2^1 \rangle \langle \sigma_1^1 \rangle$
C_1
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\langle \textsc{Acc}, m_2, 2 \rangle$
S_1
Leader
m_1
m_2
S_2
$\langle \textsc{Lrn}, m_1 \rangle$
m_1
$\langle \textsc{Lrn}, m_2 \rangle$
m_2
Partition
Non-majority
S_3
C_2
$\langle m_2 \rangle$
$\langle \sigma_1^{12} \rangle \langle \sigma_2^{12} \rangle$

New Leader in Majority Partition

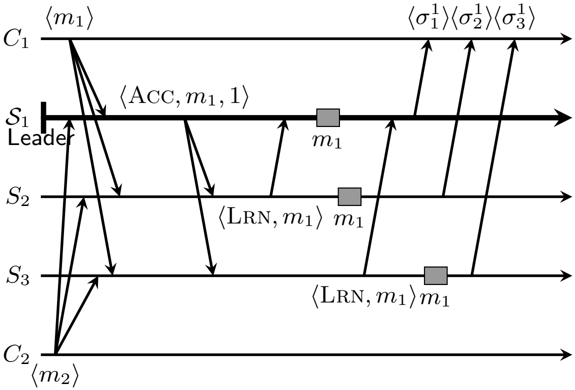

$\langle m_1 \rangle$
C_1
Retransmit
Retransmit
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\langle \textsc{Acc}, m_1, 1 \rangle$
S_1
Leader
Partition
Detection
$\langle \textsc{Acc}, m_2, 1 \rangle$
S_2
Leader
m_2
S_3
$\langle \textsc{Lrn}, m_2 \rangle$
m_2
C_2
$\langle m_2 \rangle$
$\langle \sigma_2^2 \rangle$
$\langle \sigma_3^2 \rangle$

WaitForLearn Without Partition



C_1
$\langle \textsc{Acc}, m_1, 1 \rangle$
$\mathcal{S}_1$
Leader
m_1
S_2
$\langle \textsc{Lrn}, m_1 \rangle$
m_1
S_3
$\langle \textsc{Lrn}, m_1 \rangle$
m_1
C_2

WaitForLearn With Clients


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle$
C_1
$\langle \text{ACC}, m_1, 1 \rangle$
S_1
Leader
m_1
S_2
$\langle \text{LRN}, m_1 \rangle$
m_1
S_3
$\langle \text{LRN}, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$

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

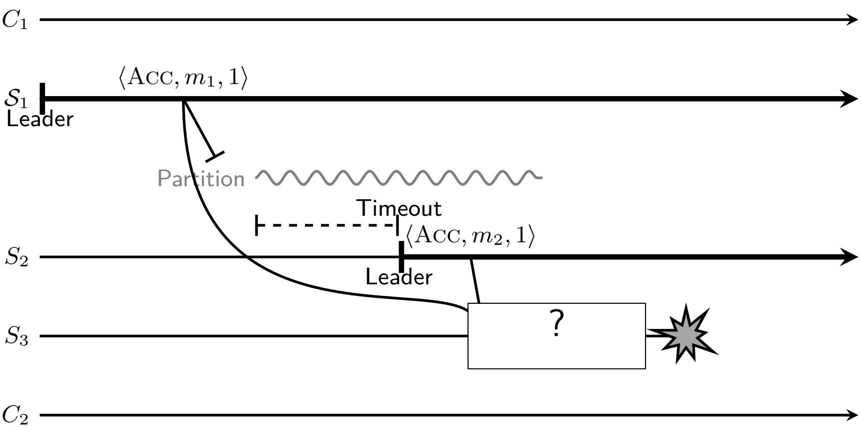
- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
    when one of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

Recap: Network Partition

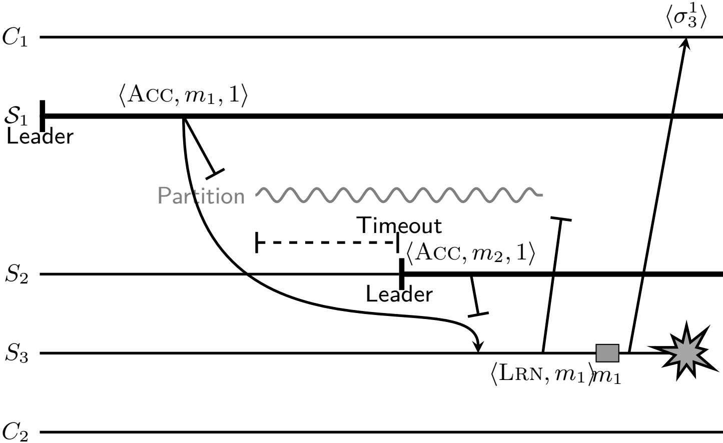
- We added another server, S_3
  - To avoid rollback using clients
- We still use the *WaitForLearn* protocol
  - To ensure that another server has seen the accept message
- Leader only needs to wait for *one* learn before executing the request
  - Allows the leader to make progress,
    when one of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

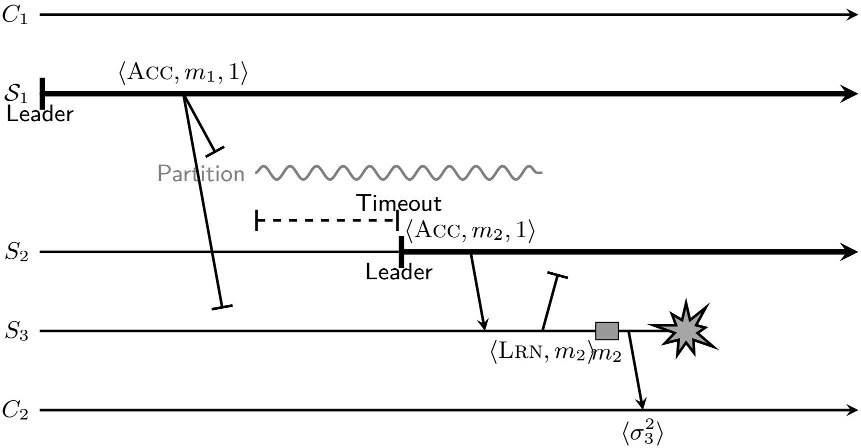

C_1
$\langle \textsc{Acc}, m_1, 1 \rangle$
S_1
Leader
Partition
Timeout
$\langle \textsc{Acc}, m_2, 1 \rangle$
S_2
Leader
?
S_3
C_2

Crash and Partition: Outcome 1 – m_1 Executed



$\langle\sigma_3^1\rangle$
C_1
$\langle\text{ACC}, m_1, 1\rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle\text{ACC}, m_2, 1\rangle$
S_2
Leader
S_3
$\langle\text{LRN}, m_1\rangle$
m_1
C_2

Crash and Partition: Outcome 2 – m_2 Executed


C_1
$\langle \text{Acc}, m_1, 1\rangle$
S_1
Leader
Partition
Timeout
$\langle \text{Acc}, m_2, 1\rangle$
S_2
Leader
S_3
$\langle \text{Lrn}, m_2\rangle$
m_2
C_2
$\langle \sigma_3^2\rangle$

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

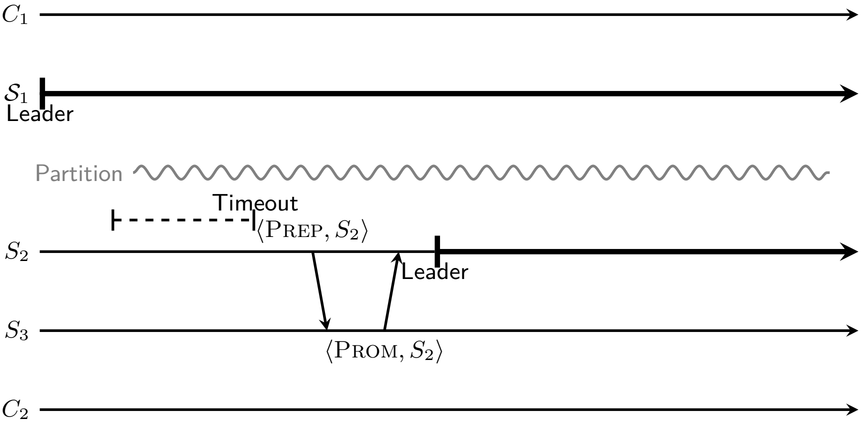
Explicit Leader Change Mechanism

- 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

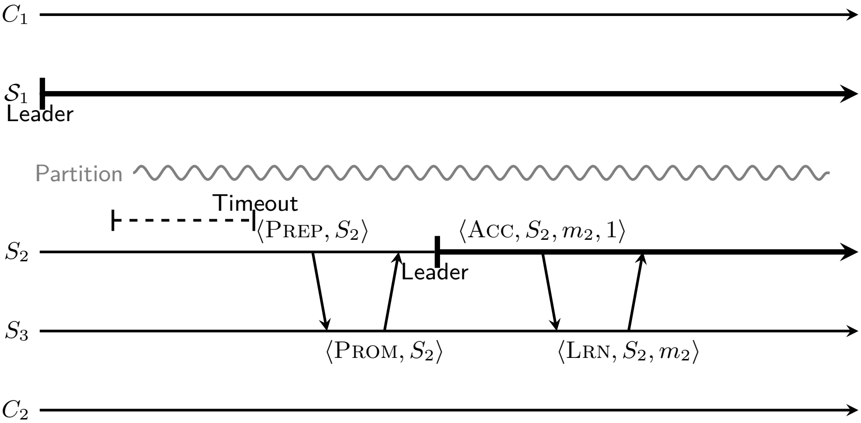
Explicit Leader Change Mechanism

- 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

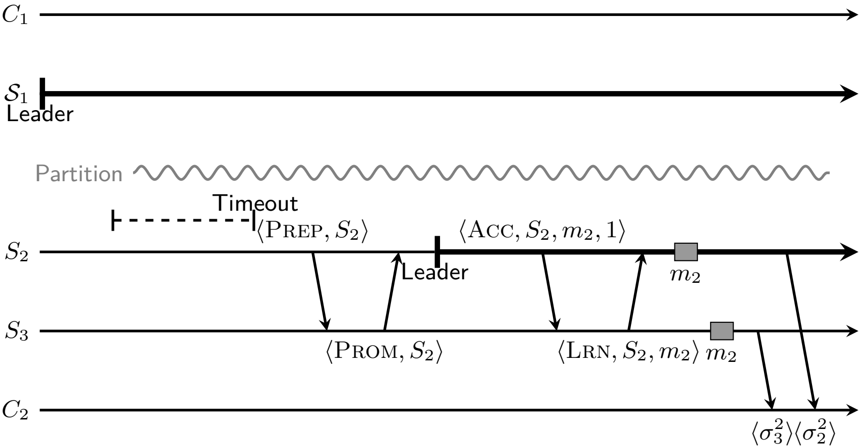

C_1
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \textsc{Prep}, S_2 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, S_2 \rangle$
C_2

Leader Identifiers in Accept and Learn Messages

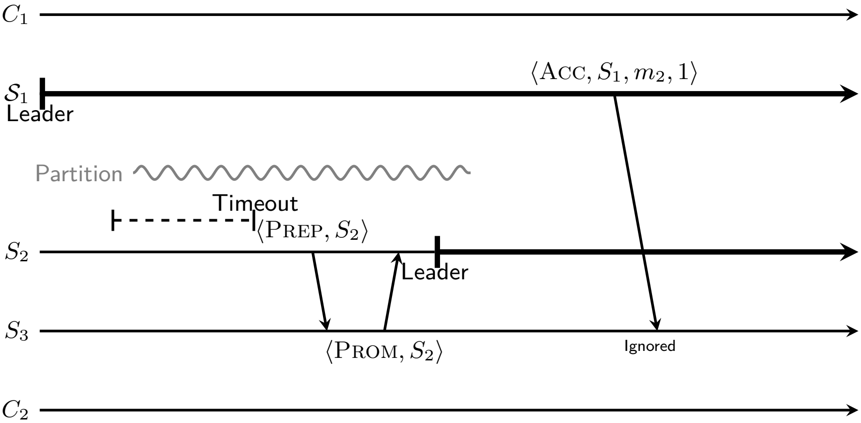


C_1
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \textsc{Prep}, S_2 \rangle$
$\langle \textsc{Acc}, S_2, m_2, 1 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, S_2 \rangle$
$\langle \textsc{Lrn}, S_2, m_2 \rangle$
C_2

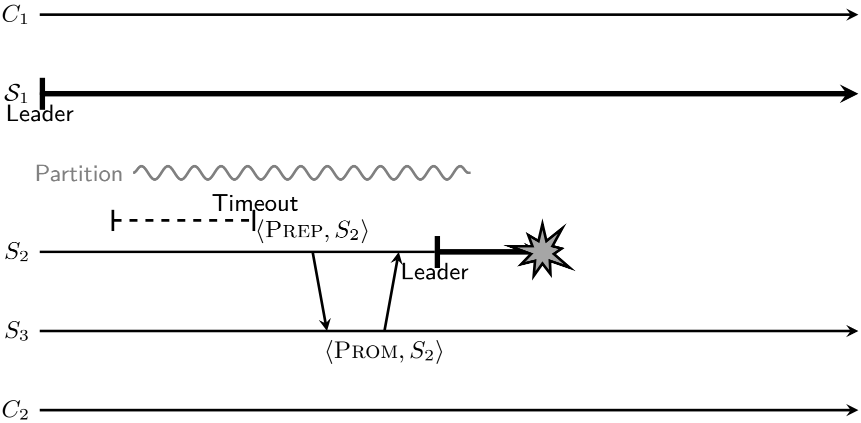
With Client Replies


C_1
S_1
Leader
Partition
Timeout
$\langle \textsc{Prep}, S_2 \rangle$
$\langle \textsc{Acc}, S_2, m_2, 1 \rangle$
S_2
Leader
m_2
S_3
$\langle \textsc{Prom}, S_2 \rangle$
$\langle \textsc{Lrn}, S_2, m_2 \rangle$
m_2
C_2
$\langle \sigma_3^2 \rangle \langle \sigma_2^2 \rangle$

Ignore Accept From Old Leader

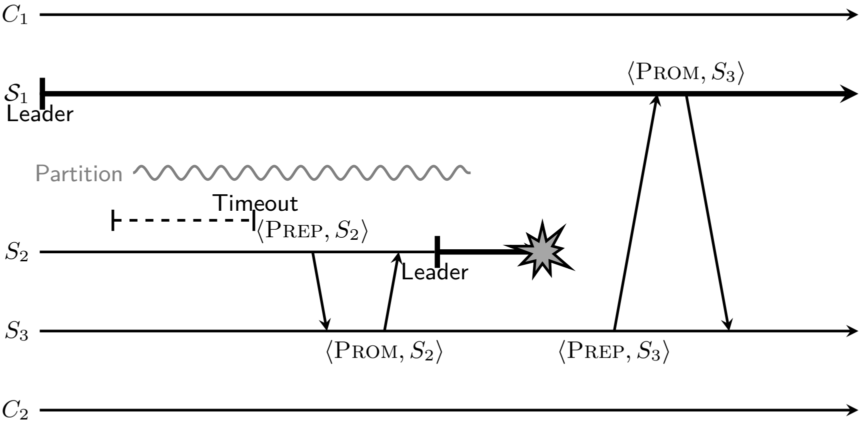

C_1
$\langle \text{Acc}, S_1, m_2, 1 \rangle$
S_1
Leader
Partition
Timeout
$\langle \text{Prep}, S_2 \rangle$
S_2
Leader
S_3
$\langle \text{Prom}, S_2 \rangle$
Ignored
C_2

What Happens Now?

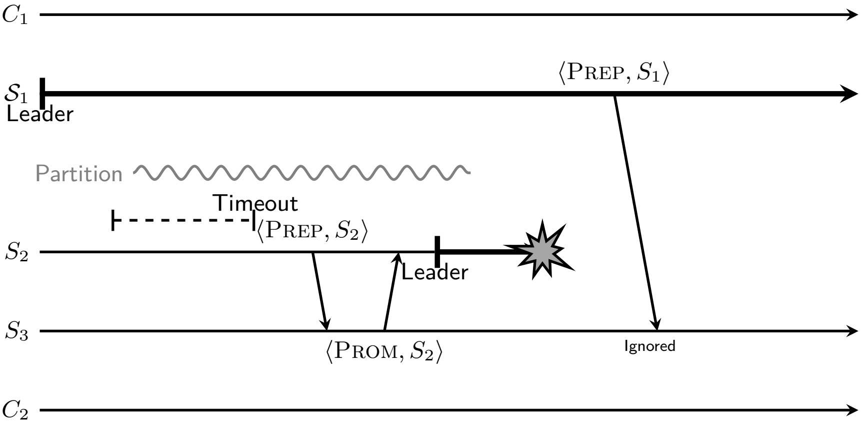


C_1
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \textsc{Prep}, S_2 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, S_2 \rangle$
C_2

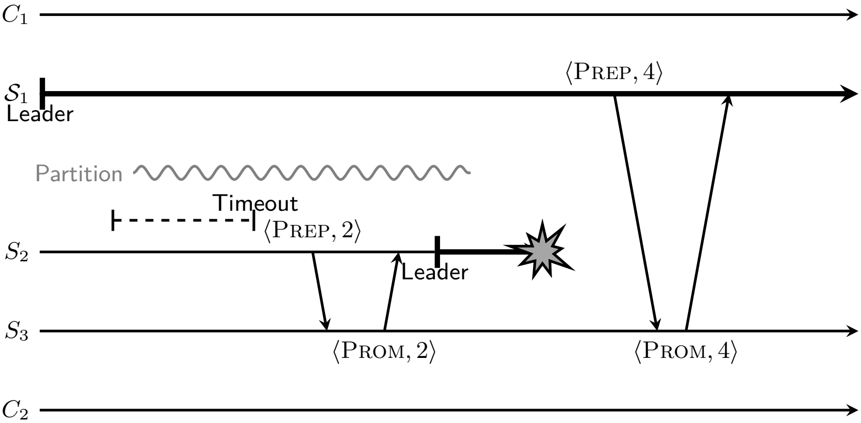
Case 1: S_3 Takes Over?


C_1
S_1
Leader
$\langle \text{Prom}, S_3 \rangle$
Partition
Timeout
$\langle \text{Prep}, S_2 \rangle$
S_2
Leader
S_3
$\langle \text{Prom}, S_2 \rangle$
$\langle \text{Prep}, S_3 \rangle$
C_2

Case 2: S_1 Takes Over Again?


C_1
$\langle \textsc{Prep}, S_1 \rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \textsc{Prep}, S_2 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, S_2 \rangle$
Ignored
C_2

Replace Leader Identifiers With Round Numbers


C_1
$\langle \textsc{Prep}, 4 \rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \textsc{Prep}, 2 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, 2 \rangle$
$\langle \textsc{Prom}, 4 \rangle$
C_2

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

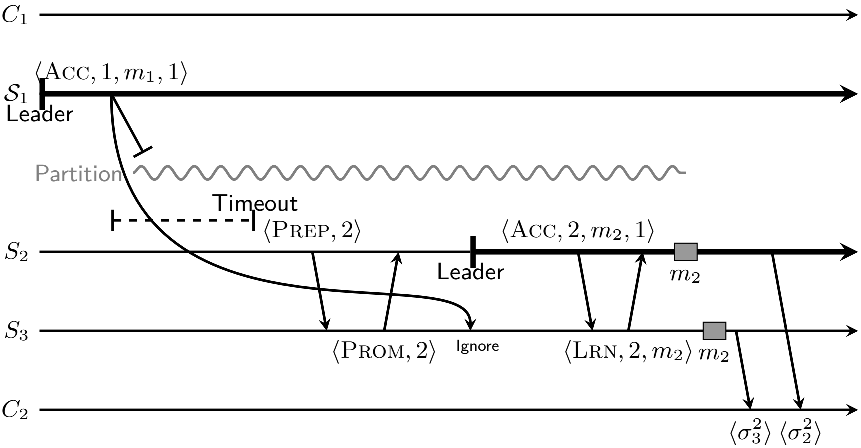
- Added round number rnd in messages
  - To identify the leader
    - $\langle \textsc{Acc}, rnd, m, seqno \rangle$: Sent by leader of round rnd
    - $\langle \textsc{Lrn}, rnd, m \rangle$: Sent to leader of round rnd
  - Round numbers are assigned:
    - S_1: $1, 4, 7, \ldots$
    - S_2: $2, 5, 8,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

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

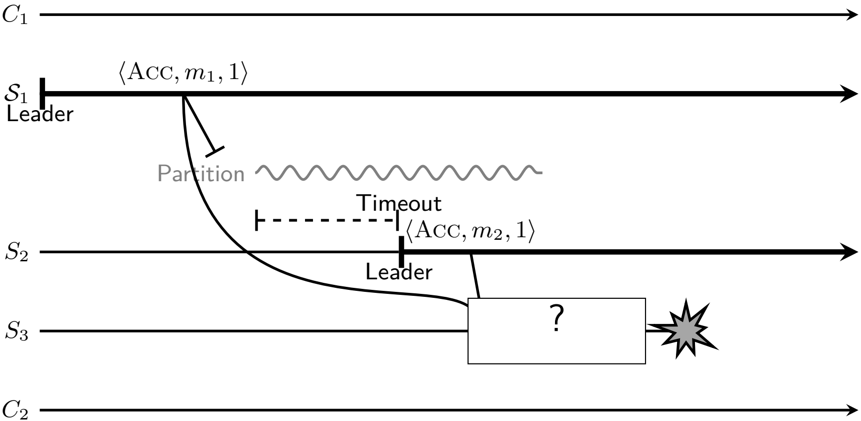
S_3 Ignores Accept Message From Old Leader



C_1
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
S_1
Leader
Partition
Timeout
$\langle \textsc{Prep}, 2 \rangle$
$\langle \textsc{Acc}, 2, m_2, 1 \rangle$
S_2
Leader
m_2
S_3
$\langle \textsc{Prom}, 2 \rangle$
Ignore
$\langle \textsc{Lrn}, 2, m_2 \rangle$
m_2
C_2
$\langle \sigma_3^2 \rangle$
$\langle \sigma_2^2 \rangle$

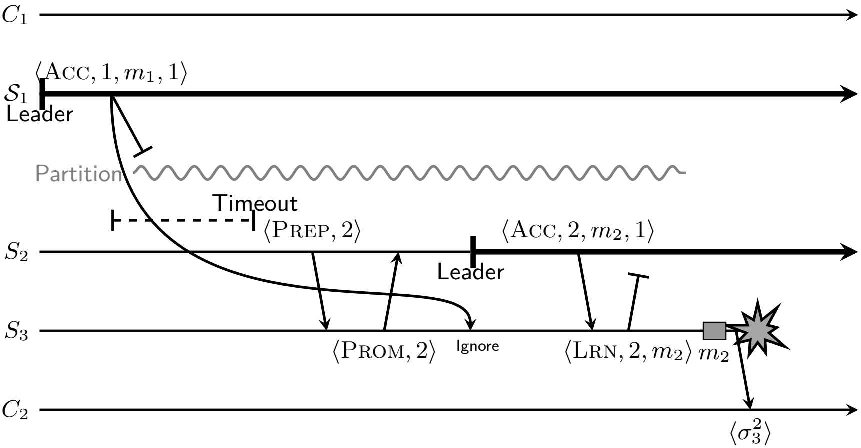
Let's Recall the Problem we are Trying to Solve

We Don't Know What S_3 Did Before Crashing



C_1
$\langle \text{Acc}, m_1, 1\rangle$
S_1
Leader
Partition
Timeout
$\langle \text{Acc}, m_2, 1\rangle$
S_2
Leader
?
S_3
C_2

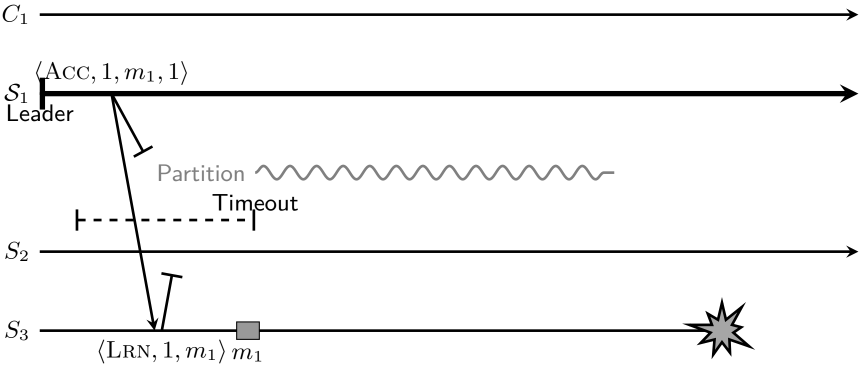
Do We Know Now?


C_1
$\langle \text{Acc}, 1, m_1, 1 \rangle$
S_1
Leader
Partition
Timeout
$\langle \text{Prep}, 2 \rangle$
$\langle \text{Acc}, 2, m_2, 1 \rangle$
S_2
Leader
S_3
$\langle \text{Prom}, 2 \rangle$
Ignore
$\langle \text{Lrn}, 2, m_2 \rangle$
m_2
C_2
$\langle \sigma_3^2 \rangle$

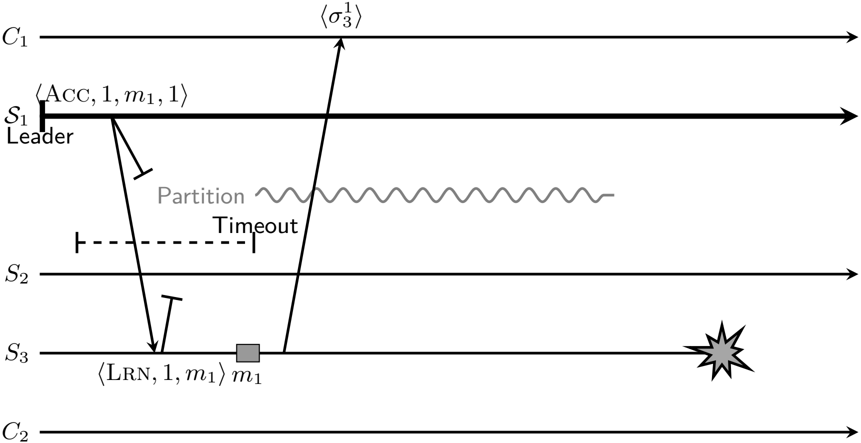
No we don’t!

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

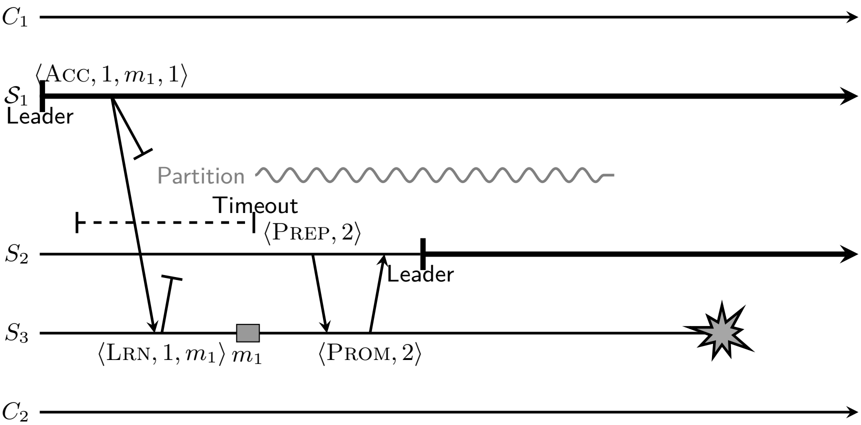
What Happens If S_3 Learn m_1?


C_1
$\langle \textsc{Acc}, 1, m_1, 1\rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
S_2
S_3
$\langle \textsc{Lrn}, 1, m_1\rangle$
m_1
C_2

What Happens If S_3 Learn m_1?


$\langle\sigma_3^1\rangle$
C_1
$\langle \textsc{Acc}, 1, m_1, 1\rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
S_2
S_3
$\langle \textsc{Lrn}, 1, m_1\rangle$
m_1
C_2

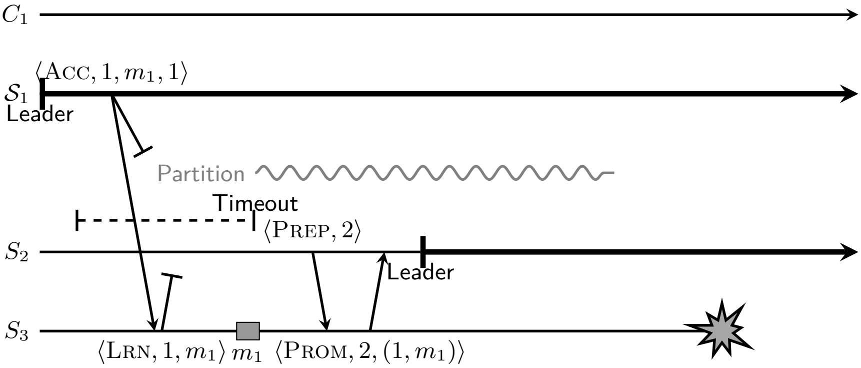
Does Leader Change Help?


C_1
$\langle \text{Acc}, 1, m_1, 1\rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \text{Prep}, 2\rangle$
S_2
Leader
S_3
$\langle \text{Lrn}, 1, m_1\rangle$
m_1
$\langle \text{Prom}, 2\rangle$
C_2

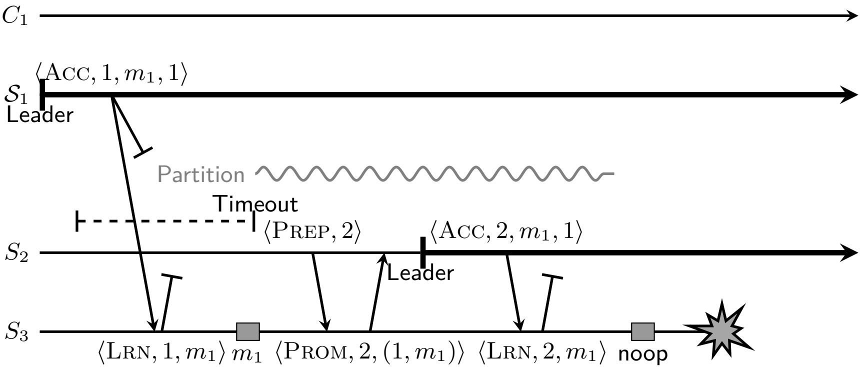
No!
We Still don't Know What
S_3 Did Before Crashing.

But the fix is Easy!

Tell new Leader About Accepted Messages


C_1
$\langle \text{ACC}, 1, m_1, 1 \rangle$
$\mathcal{S}_1$
Leader
Partition
Timeout
$\langle \text{PREP}, 2 \rangle$
S_2
Leader
S_3
$\langle \text{LRN}, 1, m_1 \rangle$
m_1
$\langle \text{PROM}, 2, (1, m_1) \rangle$
C_2

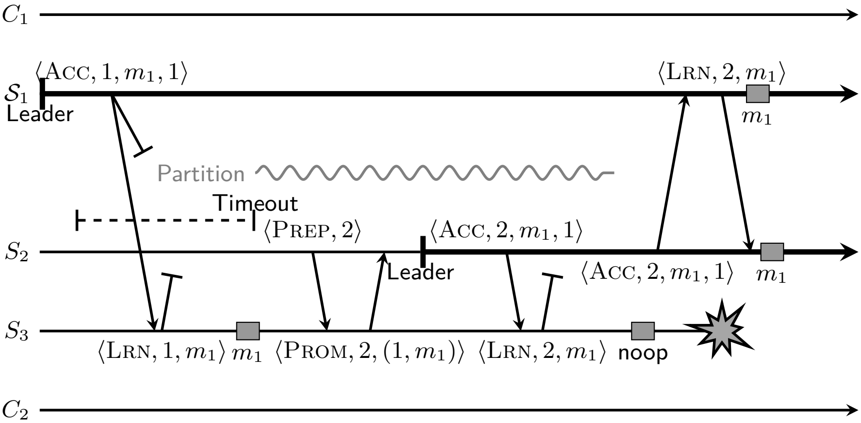
The new Leader Resends Accept for Those Messages



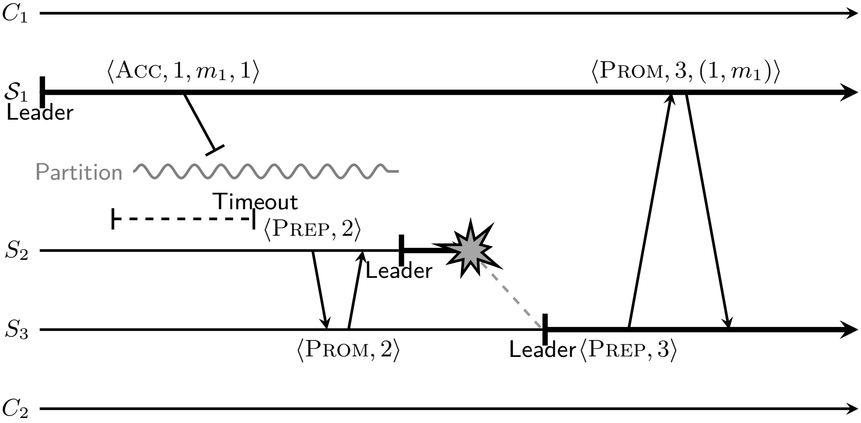
C_1
$\langle \textsc{Acc}, 1, m_1, 1\rangle$
S_1
Leader
Partition
Timeout
$\langle \textsc{Prep}, 2\rangle$
$\langle \textsc{Acc}, 2, m_1, 1\rangle$
S_2
Leader
S_3
$\langle \textsc{Lrn}, 1, m_1\rangle$
m_1
$\langle \textsc{Prom}, 2, (1, m_1)\rangle$
$\langle \textsc{Lrn}, 2, m_1\rangle$
noop
C_2

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

Leader Also Resends Accept After Merge


C_1
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
$\langle \textsc{Lrn}, 2, m_1 \rangle$
$\mathcal{S}_1$
Leader
m_1
Partition
Timeout
$\langle \textsc{Prep}, 2 \rangle$
$\langle \textsc{Acc}, 2, m_1, 1 \rangle$
S_2
Leader
$\langle \textsc{Acc}, 2, m_1, 1 \rangle$
m_1
S_3
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1
$\langle \textsc{Prom}, 2, (1, m_1) \rangle$
$\langle \textsc{Lrn}, 2, m_1 \rangle$
noop
C_2

Promise from old Leader Includes Accepted Messages


C_1
S_1
Leader
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
$\langle \textsc{Prom}, 3, (1, m_1) \rangle$
Partition
Timeout
$\langle \textsc{Prep}, 2 \rangle$
S_2
Leader
S_3
$\langle \textsc{Prom}, 2 \rangle$
Leader
$\langle \textsc{Prep}, 3 \rangle$
C_2

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
  - Previous leader sent m_1 in round 1
  - Typical naming: $\langle \textsc{Prom}, rnd, (vrnd, vval) \rangle$

Recap: Leader Change 2

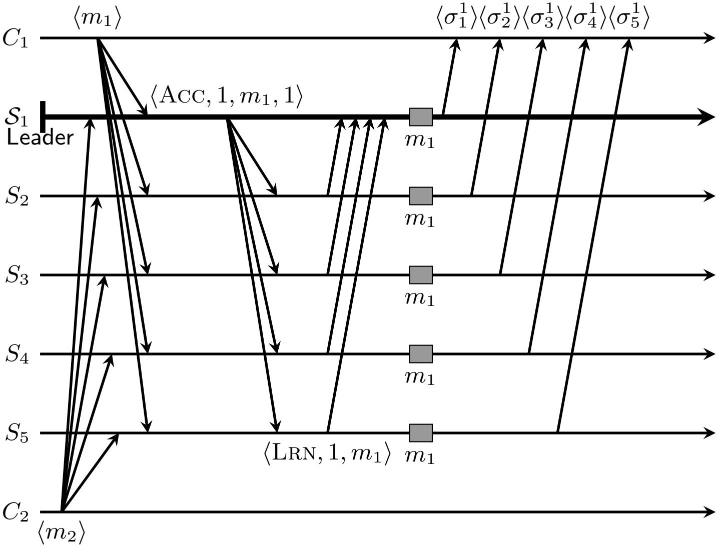
- Added information about accept from previous leader: $\langle \textsc{Prom}, rnd, (1, m_1) \rangle$
  - Promise not to accept messages from a lower round than rnd
  - Previous leader sent m_1 in round 1
  - Typical naming: $\langle \textsc{Prom}, rnd, (vrnd, vval) \rangle$
- Leader resends accept for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

What About More Than one Crash?

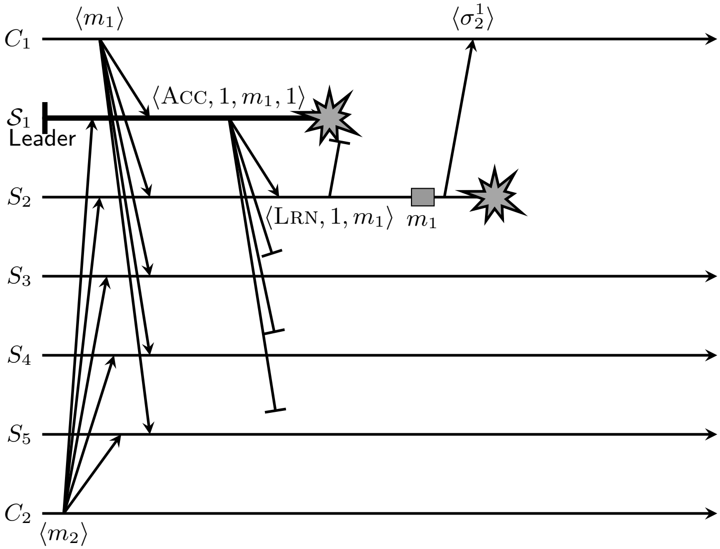
What About More Than one Crash?

- Increase the number of servers
- Limit progress to a majority partition:
  - Can only tolerate that fewer than half of the servers fail
  - To tolerate f crashes, we need $2f+1$
  - Majority: $f+1$

With Five Servers


$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \langle \sigma_4^1 \rangle \langle \sigma_5^1 \rangle$
C_1
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
S_1
Leader
m_1
S_2
m_1
S_3
m_1
S_4
m_1
S_5
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$

With Five Servers, S_2 Cannot Execute After Accept


$\langle m_1 \rangle$
$\langle \sigma_2^1 \rangle$
C_1
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
S_1
Leader
S_2
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1
S_3
S_4
S_5
C_2
$\langle m_2 \rangle$

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

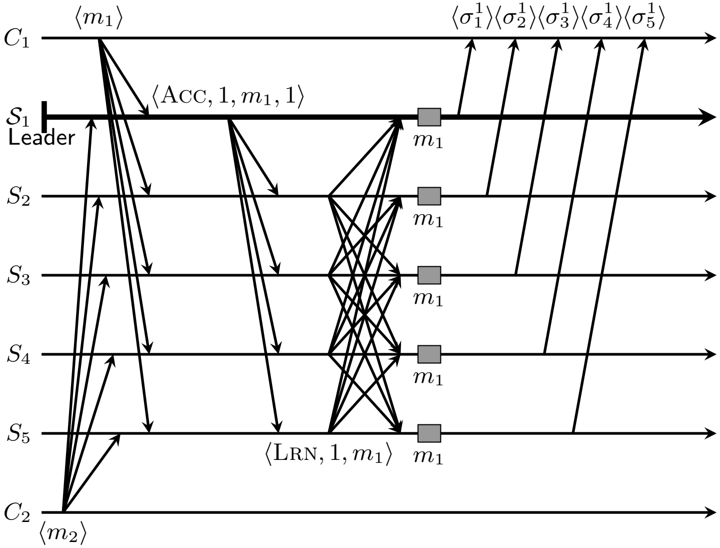
With Five Servers, S_2 Cannot Execute After Accept

- 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!

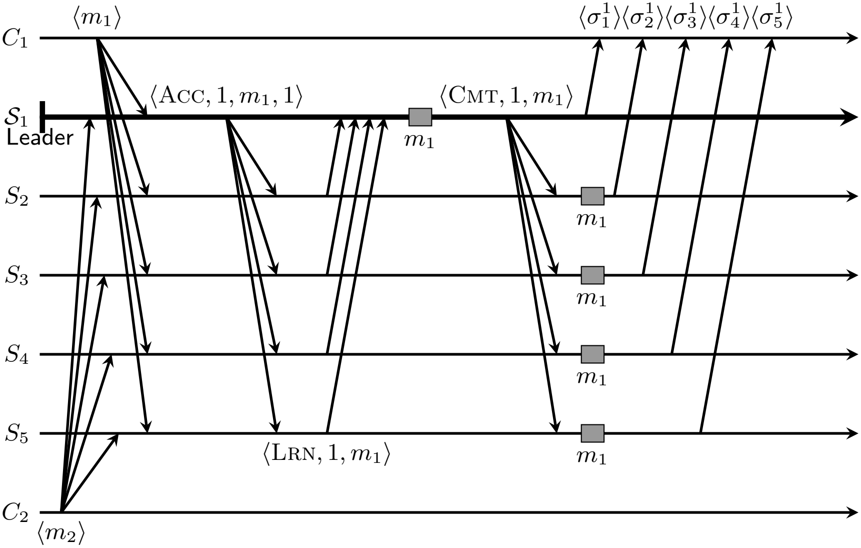
With Five Servers, S_2 Cannot Execute After Accept

- 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

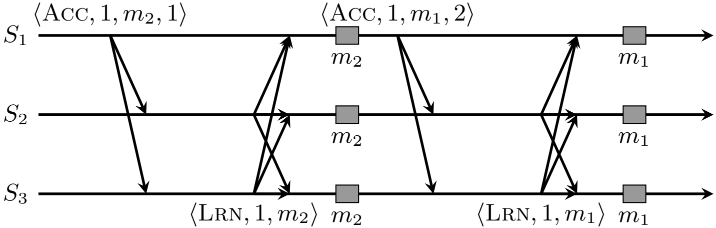

$\langle m_1 \rangle$
$\langle\sigma_1^1\rangle\langle\sigma_2^1\rangle\langle\sigma_3^1\rangle\langle\sigma_4^1\rangle\langle\sigma_5^1\rangle$
C_1
$\langle \text{Acc}, 1, m_1, 1 \rangle$
S_1
Leader
m_1
S_2
m_1
S_3
m_1
S_4
m_1
S_5
$\langle \text{Lrn}, 1, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$

Await Commit Before Execute

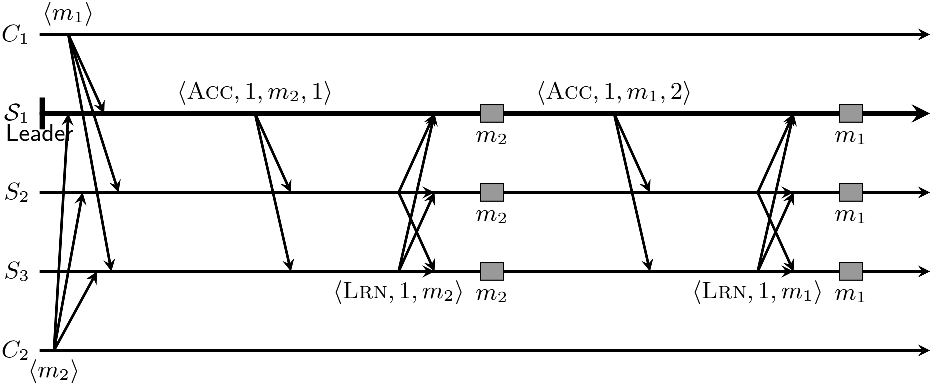

$\langle m_1 \rangle$
$\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \langle \sigma_4^1 \rangle \langle \sigma_5^1 \rangle$
C_1
$\langle \textsc{Acc}, 1, m_1, 1 \rangle$
$\langle \textsc{Cmt}, 1, m_1 \rangle$
S_1
Leader
m_1
S_2
m_1
S_3
m_1
S_4
m_1
S_5
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$

Wrapping it up!

Multi-Paxos

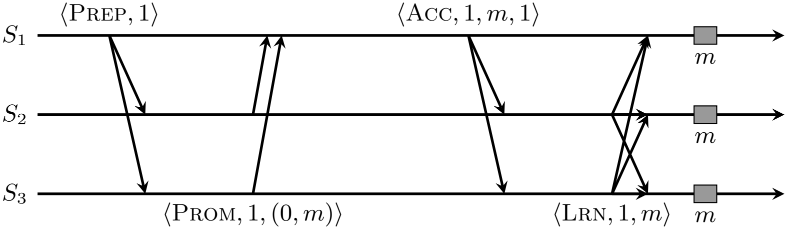

$\langle \textsc{Acc}, 1, m_2, 1 \rangle$
$\langle \textsc{Acc}, 1, m_1, 2 \rangle$
S_1
S_2
S_3
m_2
m_1
m_2
m_1
$\langle \textsc{Lrn}, 1, m_2 \rangle$
m_2
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1

Multi-Paxos with Clients

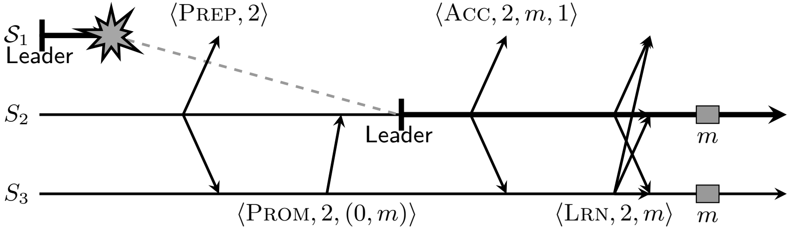


$\langle m_1 \rangle$
C_1
$\langle \textsc{Acc}, 1, m_2, 1 \rangle$
$\langle \textsc{Acc}, 1, m_1, 2 \rangle$
S_1
Leader
m_2
m_1
S_2
m_2
m_1
S_3
$\langle \textsc{Lrn}, 1, m_2 \rangle$
m_2
$\langle \textsc{Lrn}, 1, m_1 \rangle$
m_1
C_2
$\langle m_2 \rangle$

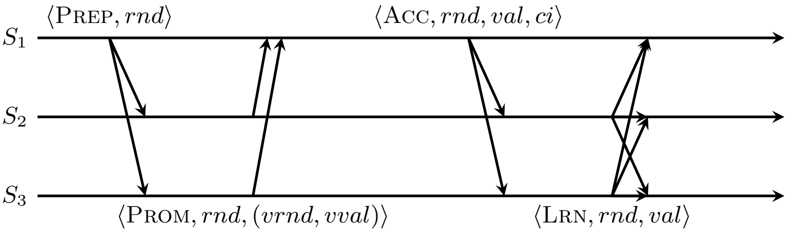
Paxos


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

Paxos with Failure


$\langle \textsc{Prep}, 2 \rangle$
$\langle \textsc{Acc}, 2, m, 1 \rangle$
$\mathcal{S}_1$
Leader
S_2
Leader
m
S_3
$\langle \textsc{Prom}, 2, (0, m) \rangle$
$\langle \textsc{Lrn}, 2, m \rangle$
m

Paxos


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

That's It! Thank You!