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*

\[ \langle m_2 \rangle \rightarrow S_1 \rightarrow \langle m_1 \rangle \rightarrow C_1 \rightarrow \langle \sigma_1 \rangle \]

\[ \langle m_2 \rangle \rightarrow C_2 \rightarrow \langle m_1 \rangle \rightarrow S_1 \rightarrow \langle \sigma_1 \rangle \]

Corresponds to execution sequence:

\[ m_2 \rightarrow m_1 \]

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS ’13
A Stateful Service: *SingleServer*

Client $C_2$ sees: $\sigma^{2}$

Client $C_1$ sees: $\sigma^{21}$

Corresponds to execution sequence: $m_2 m_1$
We Want to Make the Service Fault Tolerant!
Fault Tolerance with Two Servers

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS ’13 6 / 92
Fault Tolerance with Two Servers

Client $C_2$ sees: $\sigma^2$

Client $C_1$ sees: $\sigma^{21}$
  - $\sigma^2$ is a prefix of $\sigma^{21}$

Corresponds to execution sequence: $m_2m_1$
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!

Client $C_2$ sees: $\sigma_2 \sigma_{12}$

$\sigma_2$ is not a prefix of $\sigma_{12}$

Client $C_1$ sees: $\sigma_1 \sigma_{21}$

$\sigma_1$ is not a prefix of $\sigma_{21}$

Corresponds to execution sequence at $S_1$: $m_1 m_2$

$S_2$: $m_2 m_1$

$S_1$: $m_1$
Fault Tolerance with Two Servers: Whoops!

Client $C_2$ sees: $\sigma^2 \sigma^{12}$
- $\sigma^2$ is not a prefix of $\sigma^{12}$

Client $C_1$ sees: $\sigma^1 \sigma^{21}$
- $\sigma^1$ is not a prefix of $\sigma^{21}$

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

Hein Meling (Univ. of Stavanger)
Paxos Explained

\[ \langle \text{Acc}, m_2 \rangle \]
\[ \langle \text{Acc}, m_1 \rangle \]

\[ m_2 \]
\[ m_1 \]

\[ C_1 \]
\[ S_1 \]
\[ S_2 \]
\[ C_2 \]
Problem: Also Accept Messages can be Reordered

\[
C_1 \quad \langle \text{Acc, } m_2 \rangle \quad \langle \text{Acc, } m_1 \rangle \quad C_2
\]

\[
S_1 \quad \text{Leader}
\]

\[
S_2
\]

\[
m_2 \quad m_1 \quad m_1 \quad m_2
\]
Add Sequence Numbers

\[ \langle \text{Acc}, m_2, 1 \rangle \quad \langle \text{Acc}, m_1, 2 \rangle \]

\[ C_1 \]

\[ S_1 \text{ Leader} \]

\[ S_2 \]

\[ C_2 \]
Discard Out-of-Order Messages

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS '13  14 / 92
Now with Clients

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS '13 15 / 92
Clients Observe The Same Server States as Before

- Client $C_2$ sees: $\sigma^2$
- Client $C_1$ sees: $\sigma^{21}$
- However, $S_2$ didn’t execute $m_1$
  - Q: What to do?
Clients Observe The Same Server States as Before

- Client $C_2$ sees: $\sigma^2$
- Client $C_1$ sees: $\sigma^{21}$
- 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: $\sigma^2$
- Client $C_1$ sees: $\sigma^{21}$
- 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

\[ \langle \text{Acc}, m_2, 1 \rangle \quad \langle \text{Acc}, m_1, 2 \rangle \]

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS '13
We Need a Retransmission Mechanism!
A Learn Stops Retransmission

\[ \langle \text{Acc}, m_2, 1 \rangle \]

\[ \langle \text{LRN}, m_2 \rangle \]

C1

S1
Leader

m2

C2

S2
Don’t Send New Accept Until Learn

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS ’13  20 / 92
With Clients

\[ \langle m_1 \rangle \]

\( C_1 \)

\[ \langle \text{Acc}, m_2, 1 \rangle \]

\( S_1 \)

\[ m_2 \]

\( S_2 \)

\[ \langle \text{Lrn}, m_2 \rangle m_2 \]

\[ \langle \text{Lrn}, m_1 \rangle m_1 \]

\[ \langle \sigma_1^2 \rangle \]

\[ \langle \sigma_2^2 \rangle \]

\[ \langle \sigma_1^{21} \rangle \]

\[ \langle \sigma_2^{21} \rangle \]

Hein Meling (Univ. of Stavanger)
Paxos Explained
OPODIS '13 21 / 92
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

Hein Meling (Univ. of Stavanger)
Paxos Explained

\[ C_1 \langle m_1 \rangle \rightarrow \langle \sigma_1^1 \sigma_2^1 \rangle \]

\[ S_1 \langle \text{Acc, } m_1, 1 \rangle \rightarrow m_1 \]

\[ S_2 \rightarrow \langle m_2 \rangle \]

\[ C_2 \]
Crash: Leader Takeover

\[ \langle m_1 \rangle \rightarrow \langle \text{Acc, } m_1, 1 \rangle \rightarrow \langle \sigma^1_1 \rangle \rightarrow \langle \sigma^1_2 \rangle \]

\[ \langle m_2 \rangle \rightarrow \langle \sigma^2_2 \rangle \]

Hein Meling (Univ. of Stavanger)

Paxos Explained
Single Server Rule: Case 1

\[ \langle m_1 \rangle \]

\[ \langle m_2 \rangle \]

\[ \langle \sigma_2^2 \rangle \]
Single Server Rule: Case 2

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS ’13 27 / 92
Single Server Rule: Case 3

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS '13  28 / 92
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

\[
\langle \text{Acc}, m_1, 1 \rangle \quad \langle \text{Acc}, m_1, 1 \rangle
\]

Retransmit

\[
\langle \text{LRN}, m_1 \rangle m_1
\]

\[
\langle \text{Acc}, m_2, 2 \rangle
\]

\[
\langle \text{LRN}, m_2 \rangle m_2
\]
Recall Earlier Version

\[
\begin{align*}
\langle \text{Acc}, m_2, 1 \rangle & \quad \langle \text{Acc}, m_2, 1 \rangle \\
\langle \text{LRN}, m_2 \rangle & \quad \langle \text{LRN}, m_1 \rangle
\end{align*}
\]
Now Leader Takeover is Safe
Let’s Add Client Messages

![Diagram showing client messages and leader messages in a Paxos system.]

- C1 sends \( \langle m_1 \rangle \) to S1, the leader.
- S1 receives \( \langle Acc, m_1, 1 \rangle \) and forwards it to C1.
- C1 retransmits \( \langle m_1 \rangle \) due to a network failure.
- S1 receives \( \langle LRN, m_1 \rangle \) and verifies the message.
- \( \langle m_2 \rangle \) is sent from C2 to S2.
- S2 forwards \( \langle m_2 \rangle \) to the leader.
- The leader sends \( \langle \sigma_1^{12} \rangle \) to all followers.
Leader Remain in Control when $S_2$ Crash

\[ \langle \text{Acc}, m_1, 1 \rangle \langle \text{Acc}, m_1, 1 \rangle \]

Retransmit

\[ \langle \text{LRN}, m_1 \rangle m_1 \]

Detected

$C_1$

$S_1$

Leader

$S_2$

$C_2$

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS '13
Let’s Add Client Messages Again

\[
\langle m_1 \rangle \langle m_2 \rangle \langle \text{Acc, } m_1, 1 \rangle \langle \text{Acc, } m_1, 1 \rangle \langle \text{Lrn, } m_1 \rangle \langle \text{Lrn, } m_1 \rangle \langle \sigma_1 \rangle \langle \sigma_2 \rangle \langle \sigma_1^2 \rangle
\]
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

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.
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!
So far we have assumed that failure detection is accurate.
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.
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

\[
\begin{align*}
C_1 &\quad \langle m_1 \rangle \\
S_1 &\quad \text{Leader} \\
C_2 &\quad \langle m_2 \rangle \\
S_2 &\quad \text{Leader}
\end{align*}
\]

\[
\begin{align*}
\langle \text{Acc}, m_1, 1 \rangle &\quad \langle \text{Acc}, m_1, 1 \rangle \\
\langle \sigma_1^1 \rangle &\quad \text{Retransmit} \\
\langle \sigma_2^2 \rangle &\quad \text{Detection}
\end{align*}
\]

Hein Meling (Univ. of Stavanger)
Paxos Explained
OPODIS '13 41 / 92
Each server can switch to *SingleServer* mode (no coordination) and make progress.

But it will lead to inconsistencies:

- $S_1$ has state $\sigma_1$
- $S_2$ has state $\sigma_2$

Reconciling the state divergence involves rollback on multiple clients, which quickly becomes unmanageable.
Network Partition

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

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

\[
\begin{align*}
C_1 & \langle m_1 \rangle \\
S_1 & \langle \text{Acc, } m_1, 1 \rangle \\
S_2 & \langle \text{LRN, } m_1 \rangle m_1 \\
S_3 & \\
C_2 & \langle m_2 \rangle \\
\langle \sigma_2^1 \rangle \langle \sigma_1^1 \rangle & \\
\langle \text{Acc, } m_2, 2 \rangle & \\
\langle \text{LRN, } m_2 \rangle m_2 & \\
\langle \sigma_1^{12} \rangle \langle \sigma_2^{12} \rangle & \\
\end{align*}
\]
New Leader in Majority Partition

\[ \langle m_1 \rangle \]

\[ \langle \text{Acc}, m_1, 1 \rangle \]

\[ \langle \text{Acc}, m_1, 1 \rangle \]

\[ \langle \text{Acc}, m_1, 1 \rangle \]

\[ \langle \text{Acc}, m_1, 1 \rangle \]

\[ \langle \text{Lrn}, m_2 \rangle m_2 \]

\[ \langle \sigma_2^2 \rangle \langle \sigma_3^2 \rangle \]
WaitForLearn Without Partition

$S_1$ Leader

$\langle \text{Acc, } m_1, 1 \rangle$

$\langle \text{LRN, } m_1 \rangle m_1$

$C_1$

$S_2$

$\langle \text{LRN, } m_1 \rangle m_1$

$C_2$

$S_3$
WaitForLearn With Clients

\[ \langle \text{Acc, } m_1, 1 \rangle \]

\[ \langle \text{LRN, } m_1 \rangle m_1 \]

\[ \langle \text{LRN, } m_1 \rangle m_1 \]

\[ \langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \]

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS ’13
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

\[ \langle \text{Acc}, m_1, 1 \rangle \]

Partition

\[ \langle \text{Acc}, m_2, 1 \rangle \]

Timeout

Leader

?
Crash and Partition: Outcome 1 – $m_1$ Executed

\[
\langle \sigma_1^3 \rangle
\]

\[
\langle \text{Acc}, m_1, 1 \rangle
\]

\[
\langle \text{Acc}, m_2, 1 \rangle
\]

\[
\langle \text{LRN}, m_1 \rangle m_1
\]

\[
\text{Leader}
\]

\[
\text{Partition}
\]

\[
\text{Timeout}
\]
Crash and Partition: Outcome 2 – $m_2$ Executed

$S_1$  
Leader  
\langle \text{Acc, } m_1, 1 \rangle

$S_2$

Partition

\langle \text{Acc, } m_2, 1 \rangle

Timeout

\langle \text{LRN, } m_2 \rangle_{m_2}$

$S_3$

\langle \sigma_3^2 \rangle$

$C_1$

$C_2$
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 $S_1$ to $S_2$

Partition $S_1$ to $S_2$

Timeout $\langle \text{Prep}, S_2 \rangle$

Leader $\langle \text{PROM}, S_2 \rangle$
Leader Identifiers in Accept and Learn Messages

\[ C_1 \]
\[ S_1 \]
\[ \text{Leader} \]
\[ S_2 \]
\[ \text{Partition} \]
\[ \text{Timeout} \]
\[ \langle \text{PREP}, S_2 \rangle \]
\[ \langle \text{ACC}, S_2, m_2, 1 \rangle \]
\[ \langle \text{PROM}, S_2 \rangle \]
\[ \langle \text{LRN}, S_2, m_2 \rangle \]
\[ S_3 \]
\[ C_2 \]
With Client Replies

\[ C_1 \]

\[ S_1 \] Leader

\[ S_2 \]

\[ \langle \text{PREP}, S_2 \rangle \]

\[ \langle \text{PROM}, S_2 \rangle \]

\[ \langle \text{ACC}, S_2, m_2, 1 \rangle \]

\[ \langle \text{LRN}, S_2, m_2 \rangle \]

\[ m_2 \]

\[ m_2 \]

\[ \langle \sigma^2_3 \rangle \langle \sigma^2_2 \rangle \]
What Happens Now?

$C_1$

$S_1$

Leader

Partition

$S_2$

Timeout $\langle \text{PREP}, S_2 \rangle$

$S_3$

$\langle \text{PROM}, S_2 \rangle$

$C_2$
$S_3$ Takes Over?

$C_1$  

$S_1$  
Leader

Partition

$S_2$  

Timeout

$S_3$  

$C_2$  

$\langle \text{Prep}, S_2 \rangle$  

$\langle \text{Prom}, S_3 \rangle$  

$\langle \text{Prom}, S_2 \rangle$  

$\langle \text{Prep}, S_3 \rangle$  

Hein Meling (Univ. of Stavanger)

Paxos Explained

OPODIS '13 59 / 92
$S_1$ Takes Over Again?

\[\langle \text{Prep}, S_1 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]

\[\langle \text{Prep}, S_2 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]

\[\langle \text{Prep}, S_1 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]

\[\langle \text{Prep}, S_2 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]

\[\langle \text{Prep}, S_1 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]

\[\langle \text{Prep}, S_2 \rangle\]

\[\langle \text{PROM}, S_2 \rangle\]
Replace Leader Identifiers With Round Numbers

$C_1$

$S_1$
Leader

Partition

$S_2$

Timeout

$S_3$

Timeout

$C_2$

$\langle \text{Prep}, 4 \rangle$

$\langle \text{Prep}, 2 \rangle$

$\langle \text{Prom}, 2 \rangle$

$\langle \text{Prom}, 4 \rangle$

Hein Meling (Univ. of Stavanger) Paxos Explained
Recap: Leader Change

- Added round number $rnd$ in messages
  - To identify the leader
    - $\langle Acc, rnd, m, seqno \rangle$: Sent by leader of round $rnd$
    - $\langle LRN, rnd, m \rangle$: Sent to leader of round $rnd$
Recap: Leader Change

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

- Added round number $rnd$ in messages
  - To identify the leader
    - $\langle \text{Acc}, rnd, m, \text{seqno} \rangle$: 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)
Let’s Apply This Together
With Accept and Learn
$S_3$ Ignores Accept Message From Old Leader

Hein Meling (Univ. of Stavanger) Paxos Explained OPODIS '13 64 / 92
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

$\sim$ Partition

$\langle \text{Acc, } m_2, 1 \rangle$

$S_2$ Leader

$S_3$ $\sim$

$C_2$ ?
Do We Know Now?

\[ \langle \text{Acc}, 1, m_1, 1 \rangle \]

\[ \langle \text{PROM}, 2 \rangle \quad \text{Ignore} \]

\[ \langle \text{Lrn}, 2, m_2 \rangle \]

\[ \langle \text{Acc}, 2, m_2, 1 \rangle \]

\[ \langle \text{PREP}, 2 \rangle \]

\[ \langle \text{Timeout} \rangle \]

\[ \langle \sigma_3^2 \rangle \]

Hein Meling (Univ. of Stavanger)
No we don’t!
But it is Safe to Continue as If $m_2$ Had Been Executed
What Happens If $S_3$ Learn $m_1$?

$C_1$

$S_1$

Leader

$\langle \text{Acc}, 1, m_1, 1 \rangle$

$S_2$

$S_3$

$\langle \text{LRN}, 1, m_1 \rangle m_1$

$C_2$

Partition

Timeout

$\langle \sigma_1 3 \rangle \langle \sigma_2 3 \rangle \langle \text{Acc}, 1, m_1, 1 \rangle \langle \text{Lrn}, 1, m_1 \rangle$
What Happens If $S_3$ Learn $m_1$?
Does Leader Change Help?

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS '13  72 / 92
No!
We Still don’t Know What $S_3$ Did Before Crashing.
But the fix is Easy!
Tell new Leader About Accepted Messages

$C_1$

$S_1$

Leader

$\langle \text{Acc}, 1, m_1, 1 \rangle$

Partition

$\langle \text{Prep}, 2 \rangle$

Timeout

$\langle \text{Prom}, 2, (1, m_1) \rangle$

$\langle \text{LRN}, 1, m_1 \rangle$

$S_2$

$S_3$

$\langle \text{Lrn}, 1, m_1 \rangle$

$m_1$

$C_2$

Hein Meling (Univ. of Stavanger)  Paxos Explained  OPODIS '13
The new Leader Resends Accept for Those Messages

\( \langle \text{Acc, 1, } m_1, 1 \rangle \)

\( \langle \text{Acc, 2, } m_1, 1 \rangle \)

\( \langle \text{Lrn, 1, } m_1 \rangle \)

\( \langle \text{PROM, 2, (1, } m_1 \rangle \)

\( \langle \text{Lrn, 2, } m_1 \rangle \)

\( \text{noop} \)
Learn was Lost and $S_3$ Crashed. Leader Still can’t Execute $m_1$. 
Leader Also Resends Accept After Merge

\begin{align*}
S_1 & \langle \text{Acc}, 1, m_1, 1 \rangle \\
S_2 & \langle \text{PRep}, 2 \rangle \\
S_3 & \langle \text{Lrn}, 1, m_1 \rangle \\
C_1 & \langle \text{Lrn}, 2, m_1 \rangle \\
C_2 & \langle \text{Lrn, 2, } m_1 \rangle \\
\end{align*}
Promise from old Leader Includes Accepted Messages

\[ \langle \text{Acc}, 1, m_1, 1 \rangle \]

\[ \langle \text{Prom}, 3, (1, m_1) \rangle \]
Added information about accept from previous leader:
\[ \langle \text{PROM}, \text{rnd}, (1, m_1) \rangle \]
- Promise not to accept messages from a lower round than \( \text{rnd} \)
- Last leader did send \( m_1 \) in round 1
- Typical naming: \( \langle \text{PROM}, \text{rnd}, (v rnd, v val) \rangle \)
Added information about accept from previous leader:
\langle \text{PROM}, \text{rnd}, (1, m_1) \rangle

- Promise not to accept messages from a lower round than \text{rnd}
- Last leader did send \text{m}_1 in round 1
- Typical naming: \langle \text{PROM}, \text{rnd}, (\text{vrnd}, \text{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

\[\langle m_1 \rangle\]
\[\langle m_2 \rangle\]
\[\langle \text{Acc, 1, } m_1, 1 \rangle\]
\[\langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \langle \sigma_4^1 \rangle \langle \sigma_5^1 \rangle\]
\[\langle \text{LRN, 1, } m_1 \rangle\]
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

\[ \langle m_1 \rangle \}

\[ \langle \text{Acc}, 1, m_1, 1 \rangle \]

\[ \langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \langle \sigma_4^1 \rangle \langle \sigma_5^1 \rangle \]

\[ m_1 \]

\[ \langle \text{LRN}, 1, m_1 \rangle \]

\[ m_1 \]
Await Commit Before Execute

\[ \langle m_1 \rangle \]

\[ \langle Acc, 1, m_1, 1 \rangle \]

\[ \langle CMT, 1, m_1 \rangle \]

\[ \langle \sigma_1^1 \rangle \langle \sigma_2^1 \rangle \langle \sigma_3^1 \rangle \langle \sigma_4^1 \rangle \langle \sigma_5^1 \rangle \]
Wrapping it up!
Multi-Paxos

\[ \langle \text{Acc}, 1, m_2, 1 \rangle \] \langle \text{Acc}, 1, m_1, 2 \rangle

\[ \langle \text{Lrn}, 1, m_2 \rangle \] \langle \text{Lrn}, 1, m_1 \rangle

\[ m_2 \] \[ m_1 \]
Paxos Explained

\[\langle \text{PREP}, 1 \rangle\]  \[\langle \text{PROM}, 1, (0, m) \rangle\]  \[\langle \text{ACC}, 1, m, 1 \rangle\]  \[\langle \text{LRN}, 1, m \rangle\]
Paxos

$S_1$ → $\langle \text{PREP}, \text{rnd} \rangle$

$S_2$ → $\langle \text{PROM, rnd, } (\text{vrnd, vval}) \rangle$

$S_3$ → $\langle \text{Acc, rnd, val, ci} \rangle$

$\langle \text{Lrn, rnd, val} \rangle$
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!