Replicated State Machine
What is the problem?
Fault Tolerance by Replication

- Replica takes over on failure.
- Or in other scenarios.
- Challenge:
  - Ensure replicas are equivalent.
- Why?
Replication Requirements

• Replicas must have the same state/be equivalent.

• A simple way to build this out
  
  • Ensure software running at each replica is **deterministic**.
  
  • Ensure commands/operations are executed in the **same order**.
Determinism

- Ensure that equivalent replicas executing the same operation remain equivalent.
- What does it mean to be equivalent?
  - Depends on what you are running.
- What does it mean to be deterministic?
  - Depends on what you are running.
- **State machines** are an abstraction over these details.
  - Think back to ADTs from linearizability.
What is the Problem

In what order should these commands be run?

Client

\[
\begin{align*}
\text{set("s1", ...)} & \quad \text{set("s0", ...)} \\
\text{get("s0")} \rightarrow & \quad \text{set("s1", ...)} \\
\text{get("s0")} \rightarrow & \quad \text{get("s1")} \rightarrow \\
\end{align*}
\]
A Possible Solution

**Diagram:**
- KV-Store
  - set("s1", 42)
  - set("s1", 1729)
  - set("s1", 25)
- Client
  - set("s1", 25)
  - set("s1", 42)
  - set("s1", 1729)
A Possible Solution

set("s1", 42)

Client
set("s1", 25)
set("s1", 42)
set("s1", 1729)

KV-Store

set("s1", 25)
set("s1", 42)
set("s1", 1729)

Client

KV-Store

set("s1", 1729)
set("s1", 42)

Client
set("s1", 1729)
A Possible Solution

- `set("s1", 42)`
- `set("s1", 25)`
- `set("s1", 1729)`

Diagram:
- Two KV-Store nodes connected by arrows.
- Arrows from KV-Store to Client nodes:
  - Client 1: `set("s1", 25)`
  - Client 2: `set("s1", 42)`
  - Client 3: `set("s1", 1729)`
How to build fault tolerant oracles?
What Do We Need?

- **Agreement** on operation order.
- **Validity** to ensure operations executed were actually issued.
Consensus Protocols

• **Termination**: All correct nodes *eventually* decide on a value to output.

• **Agreement**: All decided nodes decide on the *same* value.

• **Validity**: The decision must be one of the inputs.
Consensus Protocols

• **Termination**: All correct nodes *eventually* decide on a value to output.

• **Eventual Agreement**: All decided nodes *eventually* decide on the *same* value.

• **Validity**: The decision must be one of the inputs.
Welcome to Paxos
Outline

• Going to go over single-decree Paxos.
  • Lamport's paper. Idea is to understand when and why it works.
  • Then look at how to apply this idea to build out a RSM.
Outline

• Going to go over single-decree Paxos.
  • Lamport's paper. Idea is to understand when and why it works.
  • Then look at how to apply this idea to build out a RSM.
Single Decree Paxos
Three Types of Participants

- Proposers
- Acceptors
- Learners
Three Types of Participants

- **Proposers**
  - Propose values that should be selected from.

- **Acceptors**
  - Decide what value is ultimately accepted.

- **Learners**
  - Are told what decision was made and can then act on the decision.

Proposers | Acceptors | Learners
Paxos: Requirements

- **Validity**: Acceptors should only choose values that are proposed.
- **Agreement**: Only one value should be chosen.
Achieving Agreement

- Relies on both proposers and acceptors.
- Acceptors make sure that a chosen value cannot be forgotten.
  - How?
- Proposers make sure that they don't try to override a chosen value.
  - How?
Paxos Invariants

Proposal: (id, value)

• Each proposal has a unique ID. [For example use machine ID to ensure this].

• Need to make sure proposals are **totally ordered**.

• If some proposal with ID $i$ and value $v$ is chosen then

• all proposals with $ID > i$ must also have value $v$. 

<table>
<thead>
<tr>
<th>ID</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
</tr>
</tbody>
</table>
Paxos Protocol: Phase 1

Proposal ID: (<index>, <Sequence #>)
Prepare Message: prepare <proposal ID>
Paxos Protocol: Phase 1

Want to propose cake

Promise Message: promise <proposal ID> <accepted value>
Paxos Protocol: Phase 2

Want to propose cake

Accept Message: accept <proposal ID> <value>
Paxos Protocol: Phase 2

Want to propose cake

Proposal: (1, a)
Accepted: cake

Proposal: (1, a)
Accepted: cake

Proposal: (1, a)
Accepted: cake

Proposal: (0, z)
Accepted: ∅
Paxos Protocol: Phase 1

Prepare Message: prepare <proposal ID>

Want to propose ice cream

Proposal: (1, a)  Accepted: cake
Proposal: (1, a)  Accepted: cake
Proposal: (1, a)  Accepted: cake
Proposal: (0, z)  Accepted: ∅
Paxos Protocol: Phase 1

Want to propose ice cream

.promise (1, b) cake
.promise (1, b) cake
.promise (1, b) ∅

Proposal: (1, a)
Accepted: cake

Proposal: (1, b)
Accepted: cake

Proposal: (1, b)
Accepted: ∅
Paxos Protocol: Phase 2

Prepare Message: prepare <proposal ID>

Proposal: (1, a)
Accepted: cake

Proposal: (1, b)
Accepted: cake

Proposal: (1, b)
Accepted: cake

Proposal: (1, b)
Accepted: ∅
Paxos Protocol: Phase 2

- Proposal: (1, a)  Accepted: cake
- Proposal: (1, b)  Accepted: cake
- Proposal: (1, b)  Accepted: cake
- Proposal: (1, b)  Accepted: cake
Paxos: Some Questions

- Why do proposers need to pick the last committed value returned in Phase 1?
Paxos: Some Questions

Is it possible to reach this situation?
Paxos: Some Questions

Proposal: (1, a)  
Accepted: cake

Proposal: (1, a)  
Accepted: cake

Proposal: (1, b)  
Accepted: cannoli

Proposal: (1, b)  
Accepted: cannoli

Proposal: (1, b)  
Accepted: cannoli

Proposal: (1, c)  
Accepted: cake

Proposal: (1, c)  
Accepted: cake

Proposal: (1, c)  
Accepted: cake

Want to propose cake
Paxos: Non-Termination
Paxos Protocol: Phase 1

prepare (1, a)

Proposal: (0, z)
Accepted: ∅

Proposal: (0, z)
Accepted: ∅

Proposal: (0, z)
Accepted: ∅
Paxos Protocol: Phase 1

Proposal: (1, a)
Accepted: ∅

Proposal: (1, a)
Accepted: ∅

Proposal: (1, a)
Accepted: ∅
Paxos Protocol: Phase 1

Proposal: (1, a)
Accepted: ∅
Paxos Protocol: Phase 1

Proposal: (1, b)
Accepted: ∅
Paxos Protocol: Phase 1

a. Accept for (1, a) will fail.

b. Proposal: (1, b)
   Accepted: ∅

c. Proposal: (1, b)
   Accepted: ∅
Paxos Protocol: Phase 1

```
prepare (2, a)
```

Proposal: (1, b)
Accepted: ∅
Paxos Protocol: Phase 1

Proposal: (2, a)  
Accepted: ∅  
promise (2, a) ∅  
promise (2, a) ∅  
promise (2, a) ∅
Paxos Protocol: Phase 1

a

b Accept for (1, b) will fail.

c

Proposal: (2, a)
Accepted: ∅
How to Resolve this Problem?

- Elect a leader.
  - Introduce random timeouts to ensure someone eventually wins.
- Leader is the only proposer (by and large).
  - Still need acceptors and quorum to make sure future leaders don't forget.
- Elect a new leader in response to failure/timeout/etc.
Extending to State Machine
What is Going on With This?

• Return to RSMs: we want a consensus algorithm to decide order of operations.
  • Without knowing all operations a-priori -- so not deciding just one value.
• Model sequence of commands as an array with slots.
  • "Run" an instance of Paxos for each slot in this array.
But Use a Leader

• Rather than doing this naively, we are going to rely on a leader.

• Allow leader to avoid the promise phase.
Multi Paxos: Phase 1

a  
Can I be leader?

b  
c  

Yes, you can be leader.
Can I be leader?

Ballot: (0,z)

Ballot: (0,z)

Ballot: (0,z)
Multi Paxos: Phase 1

Can I be leader?

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

p1b((1,a), accepted)

p1b((1,a), accepted)

p1b((1,a), accepted)
Multi Paxos: Phase 1

- Ballot: (1, a)  
  Accepted: [...] 

- Ballot: (1, a)  
  Accepted: [...] 

- Ballot: (1, a)  
  Accepted: [...]
Multi Paxos: Phase 2

Ballot: (1, a)  Accepted: [...]

p2a(a, <(1,a), 1, x>)

Ballot: (1, a)  Accepted: [...]

p2a(a, <(1,a), 1, x>)

Ballot: (1, a)  Accepted: [...]

p2a(a, <(1,a), 1, x>)
Multi Paxos: Phase 2

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

p2b((1, a))
Multi Paxos: Phase 2

Ballot: (1, a)  Accepted: [...]
Multi Paxos: Phase 2

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

Ballot: (1, a)
Accepted: [...]

p2b((1, a))
Multi Paxos: Phase 1

Can I be leader?

Ballot: (1,a)
Ballot: (1,a)
Ballot: (1,a)
Multi Paxos: View Change

Ballot: (1, b)

Ballot: (1, b)

Ballot: (1, b)
Multi Paxos: View Change

Ballot: (1, b)
Accepted: [...]

Ballot: (1, b)
Accepted: [...]

Ballot: (1, b)
Accepted: [...]

p2b((1, b))
p2b((1, b))
p2b((1, b))
Multi Paxos: View Change

Ballot: (1, b)
Accepted: [...]
As an aside: how does one build a reusable version of this system?

Most common abstraction now: build a key-value store.

- Popularized by Chubby at Google, implemented multipaxos.

Can use key-value store to implement locks, indicate what is alive, etc.

- Often extended with leases to make sure state is cleaned up despite failures.
Summary

• Replicated state machines are a powerful abstraction for fault tolerance.
• However, require an oracle that can order commands across all replicas.
• Enter consensus protocols.