State Machine Replication
Key Ideas

To tolerate faults...
... replicate functionality!

• Can represent deterministic distributed system as replicated state machine (SMR)
• Each replica reaches the same conclusion about the system independently
• Key examples of distributed algorithms that generically implement SMR
• Formalizes notions of fault-tolerance in SMR
Motivation

Client

![Diagram showing client interacting with server]

Server

X = 10

get(x)

...No response

Client

get(x)

10
Motivation

Client → Server

X = 10

X = 10
Motivation

- Need replication for fault tolerance
- What happens in these scenarios without replication?
  - Storage - disk failure
  - Web service - network failure
- Be able to reason about failure tolerance
- How badly can things go wrong and have our system continue to function?
State Machines

- State variables
- Deterministic commands
Requests and Causality

Process order consistent with potential causality

- Client A sends $r$, then $r'$
- $r$ is processed before $r'$
- $r$ causes Client B to send $r'$
- $r$ is processed before $r'$. 
Coding State Machines

- State machines are procedures
- Client calls procedure
- Avoid loops
- More flexible structure
State Machine Replication
State Machine Replication
Write

\[ \text{put}(x, 10) \]
After the Write

Great!
Write

\[ \text{put}(x, 10) \]
Need Agreement

Replicas need to agree which requests have been handled

Problem!
Two Writes

put(x, 10)

put(x, 30)

r0

r1

X = 3

X = 3

X = 3

X = 3
Either Outcome is Fine

- Either $X = 10$ or $X = 30$
Order Matters

\[ \text{put}(x,10) \rightarrow r0 \]
\[ \text{put}(x,30) \rightarrow r1 \]
Order Matters

- `put(x, 10)`
  - `r0` connects to `X = 3`
  - `r0` connects to `X = 3`

- `put(x, 30)`
  - `r1` connects to `X = 3`
  - `r1` connects to `X = 3`
Order Matters

put(x, 10)

r0

X = 10

X = 30

r1

put(x, 30)

r1
Order Matters

put(x, 10)

put(x, 30)
Order Matters

Replicas need to handle requests in the same order.
Requirements

All non-faulty servers need...

◆ Agreement
  - Every replica needs to accept the same set of requests

◆ Order
  - All replicas process requests in the same relative order
Idea for Agreement

- Someone proposes a request
- If the proposer is non-faulty, all servers will accept that request
Agreement

$\text{put}(x, 10)$
Agreement

Non-faulty Transmitter

\[
\text{put}(x, 10)
\]
Idea for Order

Assign unique ids to requests, process them in ascending order

◆ How do we assign unique ids in a distributed system?
◆ How do we know when every replica has processed a given request?
Order

put(x,30)

r0

X = 3

X = 3

X = 3

X = 3

put(x,10)

r1
Order

Assign Total Ordering

<table>
<thead>
<tr>
<th>Request</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1</td>
</tr>
<tr>
<td>r1</td>
<td>2</td>
</tr>
</tbody>
</table>

put(x, 30)

put(x, 10)
Order

Assign Total Ordering

<table>
<thead>
<tr>
<th>Request</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1</td>
</tr>
<tr>
<td>r1</td>
<td>2</td>
</tr>
</tbody>
</table>
Order

Assign Total Ordering

<table>
<thead>
<tr>
<th>Request</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1</td>
</tr>
<tr>
<td>r1</td>
<td>2</td>
</tr>
</tbody>
</table>
Order

Assign Total Ordering

<table>
<thead>
<tr>
<th>Request</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1</td>
</tr>
<tr>
<td>r1</td>
<td>2</td>
</tr>
</tbody>
</table>

Cannot receive request with smaller ID

r0 is now stable!
Order

Assign Total Ordering

<table>
<thead>
<tr>
<th>Request</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1</td>
</tr>
<tr>
<td>r1</td>
<td>2</td>
</tr>
</tbody>
</table>

r0 is now stable!
r1 is now stable!
Generating IDs

- Order via clocks (client timestamp = id)
  - Logical clocks
  - Synchronized clocks

- Two-phase ID generation
  - Every replica proposes a candidate
  - One candidate is chosen and agreed upon by all replicas
Replica ID Generation

- put(x, 30) to r0
- put(x, 10) to r1
## Replica ID Generation

1) Propose candidates

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>2.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>2.2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>2.3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>2.4</td>
<td></td>
</tr>
</tbody>
</table>
## Replica ID Generation

### Table 1: Request IDs and Requests

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.1</td>
<td>2.4</td>
</tr>
<tr>
<td>r1</td>
<td>2.1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.2</td>
<td>2.4</td>
</tr>
<tr>
<td>r1</td>
<td>2.2</td>
<td></td>
</tr>
</tbody>
</table>

### Table 2: Request IDs and Requests

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>2.3</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>2.4</td>
<td>2.4</td>
</tr>
</tbody>
</table>

2) Accept r0
Replica ID Generation

3) Accept $r_1$
Replica ID Generation

$r1$ is now stable
Replica ID Generation

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>2.1</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>1.1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>2.3</td>
<td>2.4</td>
</tr>
</tbody>
</table>

X = 10

4) Apply r1
Replica ID Generation

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>2.1</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>1.1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>1.2</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>2.3</td>
<td>2.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1.4</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>2.4</td>
<td>2.4</td>
</tr>
</tbody>
</table>

5) Apply $r_0$
Rules for Replica-Generated IDs

- Any new candidate ID must be > ID of any accepted request
- The ID selected from the candidate list must be >= each candidate

When is a candidate stable?
- It has been accepted
- No other pending request with a smaller candidate ID
Faults

- **Fail-Stop**
  - A faulty server can be detected as faulty

- **Byzantine**
  - Faulty servers can do arbitrary, perhaps malicious things
  - This includes crash failures (server can stop responding without notification)
Fail-Stop Tolerance

\( \text{put}(x, 30) \)
Fail-Stop Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.1</td>
<td></td>
</tr>
</tbody>
</table>

1) Propose Candidates....
Fail-Stop Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

2) Accept $r_0$
Fail-Stop Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

2) Apply $r0$
Fail-Stop Tolerance

2) Apply $r_0$

GAME OVER!!!
Fail-Stop Fault Tolerance

- To tolerate $t$ failures, need $t+1$ servers.
- As long as 1 server remains, we’re OK
- Only need to participate in protocols with other live servers
Byzantine Fault Tolerance

- To tolerate $t$ failures, need $2t + 1$ servers
- Protocols now involve votes
  - Can only trust server response if the majority of servers say the same thing
- $t + 1$ servers need to participate in replication protocols
This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a State Machine,
Fault-Tolerant State Machines

- Implement the state machine on multiple processors
- State machine replication
  - Each starts in the same initial state
  - Executes the same requests
  - Requires consensus to execute in same order
  - Deterministic, each will do the exact same thing
  - Produce the same output
Consensus

- Termination
- Validity
- Integrity
- Agreement

Ensures procedures are called in same order across all machines