Fault-Tolerant State Machine Replication

Drew Zagieboylo
Authors

• Fred Schneider
Takeaways

• Can represent **deterministic** distributed system as *Replicated State Machine*

• 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*
Outline

• Motivation
• State Machine Replication
• Implementation
• Fault Tolerance Requirements
• An Example - Chain Replication
• Evaluation
Motivation

Client

Server

\[ X = 10 \]

get(x)

…No response
Motivation

Server

Client

\[ X = 10 \]

\[ \Box = 10 \]

\[ X = 10 \]

\[ X = 10 \]
Motivation

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

- \( c \) is a Command
- \( f \) is a Transition Function

\[
\begin{align*}
X &= Y \\
f(c) &\downarrow  \\
X &= Z
\end{align*}
\]
State Machine Replication (SMR)

• The State Machine Approach to a fault tolerant distributed system

• Keep around N copies of the state machine
State Machine Replication (SMR)

- The State Machine Approach to a fault tolerant distributed system
- Keep around $N$ copies of the state machine
put(x, 10)
SMR
Requirements

Great!
SMR
Requirements

\[ \text{put}(x, 10) \]
SMR Requirements

- Replicas need to agree on the requests that have been handled

```
get(x)  10
          X = 10
          X = 10
          X = 10
          X = 3
get(x)  3
```

Problem!
SMR
Requirements

\( \text{put}(x, 10) \rightarrow r0 \rightarrow \begin{array}{c} X = 3 \\ X = 3 \\ X = 3 \\ X = 3 \end{array} \)

\( \text{put}(x, 30) \rightarrow r1 \rightarrow \begin{array}{c} X = 3 \\ X = 3 \\ X = 3 \\ X = 3 \end{array} \)
SMR
Requirements

X = 10
X = 10
X = 10
X = 10

OR

X = 30
X = 30
X = 30
X = 30
SMR Requirements

\[ \text{put}(x, 10) \rightarrow r0 \]

\[ X = 3 \]

\[ X = 3 \]

\[ X = 3 \]

\[ X = 3 \]

\[ \text{put}(x, 30) \rightarrow r1 \]
SMR

Requirements

\( \text{put}(x, 10) \)
\( r_0 \)

\( r_0 \)

\( r_0 \)

\( X = 3 \)

\( X = 3 \)

\( X = 3 \)

\( X = 3 \)

\( \text{put}(x, 30) \)
\( r_1 \)

\( r_1 \)
SMR
Requirements

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

\[ r_0 \]

\[ \text{put}(x,30) \]

\[ r_1 \]
SMR
Requirements

E = 10
E = 30
r0
r1

put(x, 10)

r0
r1

E = 10
E = 30
r0
r1

put(x, 30)

r1
r0
• Replicas need to handle requests in the same order.
SMR

• 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
Implementation

- Agreement
  - Someone proposes a request; if that person is nonfaulty all servers will accept that request
  - Strong and Dolev [1983] and Schneider [1984] for implementations
  - Client or Server can propose the request
SMR Implementation

\[ \text{put}(x, 10) \]
SMR Implementation

Non-faulty Transmitter

put(x, 10)
Implementation

• 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?
SMR Requirements

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>
SMR
Requirements

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>
SMR Requirements

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>
SMR

Requirements

<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>

Assign Total Ordering

X = 30

r0 is now stable!
SMR Requirements

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!
Implementation

Client Generated IDs

• Order via Clocks (Client timestamps represent IDs)
  • Logical Clocks
  • Synchronized Clocks

• Ideas from last week! [Lamport 1978]
Implementation

Replica Generated IDs

- 2 Phase ID generation
  - Every Replica proposes a candidate
  - One candidate is chosen and agreed upon by all replicas
Replica ID Generation

\[
\text{put}(x, 30) \rightarrow r_0
\]

\[
\begin{array}{ccc}
\text{X = 3} & \text{X = 3} \\
\text{X = 3} & \text{X = 3}
\end{array}
\]

\[
\text{put}(x, 10) \rightarrow r_1
\]
Replica ID Generation

1) Propose Candidates
Replica ID Generation

2) Accept \( r0 \)
Replica ID Generation

<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>2.2</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>r0</td>
<td>1.2</td>
<td>2.4</td>
</tr>
<tr>
<td>r1</td>
<td>2.2</td>
<td>2.2</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>

3) Accept $r_1$
Replica ID Generation

$r1$ is now stable
Replica ID Generation

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>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>2.3</td>
<td>2.4</td>
</tr>
</tbody>
</table>

5) Apply r0
Implementation

Replica Generated IDs

• 2 Rules for Candidate Generation/Selection

  • Any new candidate ID must be > the id of any accepted request.

  • The ID selected from the candidate list must be >= each candidate

• In the paper these are written as:

  • If a request \( r' \) is seen by a replica \( sm_i \) after \( r \) has been accepted by \( sm_i \) then \( uid(r) < cuid(sm_i, r') \)

  • \( cuid(sm_i, r) \leq uid(r) \)
Implementation

Replica Generated IDs

• When do we know a candidate is *stable*?
  
  • A candidate is *accepted*
  
  • No other pending requests with smaller candidate ids
Fault Tolerance

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

- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things

- Crash Failures
  - Server can stop responding without notification (subset of Byzantine)
Fault Tolerance

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

• Byzantine
  • Faulty servers can do arbitrary, perhaps malicious things

• Crash Failures - NOT covered in paper
  • Server can stop responding without notification (subset of Byzantine)
Fail-Stop Tolerance

put\((x,30)\)

\(r0\)
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 \( r0 \)
Fail-Stop Tolerance

2) Apply $r0$

<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>
Fail-Stop Tolerance

2) Apply $r0$

GAME OVER!!!
Fail-Stop 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 Tolerance

$\text{get}(x)$

$X = 3$

$r0$

$X = 3$

$X = 3$

$X = 3$
Byzantine Tolerance

\[ \text{get}(x) \]

\[ r0 \]
Byzantine Tolerance
Byzantine Tolerance

Client trusts the majority =>
Need majority to participate in replication
Byzantine Tolerance

Who to trust?? 3 or 7?
Byzantine Tolerance

\[ \text{put}(x,30) \]

\[ r0 \]

\[
\begin{array}{ccc}
X = 3 & & \text{skull}
\end{array}
\]

\[
\begin{array}{ccc}
X = 3 & & X = 3
\end{array}
\]
Byzantine Tolerance

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>r0</td>
<td>1.2</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>??</td>
<td>??</td>
</tr>
<tr>
<td>r0</td>
<td>1.4</td>
<td></td>
</tr>
</tbody>
</table>
Byzantine Tolerance

a) No response

a) Wait for majority candidates
Timeout long requests & notify others
Byzantine Tolerance

a) No response

a) Accept r0
### Byzantine Tolerance

#### Small ID

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>r0</td>
<td>1.2</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>-5</td>
<td>???</td>
</tr>
<tr>
<td>r0</td>
<td>1.4</td>
<td></td>
</tr>
</tbody>
</table>
Byzantine Tolerance
Small ID

uid = max(cuid(sm_i, r))
Ignore low candidates!

2) Accept r0
Byzantine Tolerance
Small ID

\[ uid = \max(\text{cuid}(sm_i, r)) \]
Ignore low candidates!

2) Accept r0

<table>
<thead>
<tr>
<th>Req.</th>
<th>CUID</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r0 )</td>
<td>1.1</td>
<td>1.4</td>
</tr>
<tr>
<td>( r0 )</td>
<td>1.2</td>
<td>1.4</td>
</tr>
<tr>
<td>( r0 )</td>
<td>-5</td>
<td>???</td>
</tr>
<tr>
<td>( r0 )</td>
<td>1.4</td>
<td>1.4</td>
</tr>
</tbody>
</table>
Byzantine Tolerance
Large ID

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>r0</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>r0</td>
<td>1.4</td>
<td></td>
</tr>
</tbody>
</table>
Byzantine Tolerance
Large ID

Large numbers follow protocol!
Byzantine Tolerance

Large ID

2) Accept r0
Fault Tolerance

• Byzantine Failures
  • 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
Other Contributions

• Tolerating Faulty Output Devices
  • (e.g. a faulty network, or user-facing i/o)

• Tolerating Faulty Clients

• Reconfiguration
Takeaways

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,

Lamport 1978
Takeaways

• Can represent *deterministic* distributed system as *Replicated State Machine*

• 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*
Chain Replication

• Authors
  • Robert Van Renesse (RVR)
  • Fred Schneider
Chain Replication

- Fault Tolerant Storage Service (Fail-Stop)

- Requests:
  - Update(x, y) => set object x to value y
  - Query(x) => read value of object x
Chain Replication

\[ X = 3 \]

\[ X = 3 \]

\[ X = 3 \]

\[ X = 3 \]
Chain Replication

Head

Tail

$X = 3$

get($x$)

3

Client
Chain Replication

Client

put(x, 30)

Head

Tail

X = 3

X = 3

X = 3

X = 3
Chain Replication

Client

put(x,30)

Head

X = 30

Tail

X = 3

X = 3

X = 3

1) Head assigns uid

 Req. | UID
-------|-------
  r0   |  1    

Put (x,30)
Chain Replication

2) Head sends message to next node
Chain Replication

Client

put(x, 30)

3) Repeat until tail is reached

Head

X = 30

Req. UID

r0 1

Tail

X = 3

Req. UID

r0 1

Req. UID

r0 1

Req. UID

r0 1
Chain Replication

Client:

```
put(x, 30)
```

4) respond to client with success
Chain Replication

• How does Chain Replication implement State Machine Replication?

• Agreement
  
  • Only *Update* modifies state, can ignore *Query*
  
  • Client always sends *update* to *Head*. *Head* propagates request down chain to *Tail*.
  
  • Everyone accepts the request!
Chain Replication

• How does Chain Replication implement State Machine Replication?

• Order
  • Unique IDs generated implicitly by Head's ordering
  • FIFO order preserved down the chain
  • Tail interleaves Query requests
  • How can clients test stability? (How can clients tell when their Updates have been handled)
Chain Replication

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Client

\[
\text{put}(x, 30)
\]

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

\[
\text{put}(x, 30)
\]

\[
\text{put}(x, 10)
\]
Chain Replication

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Client: put(x,30)  
Client: put(x,10)
Chain Replication

Req. | UID
---|---
1 | r1
2 | r0

Client

put(x,30) r0

Head

X = 30

X = 10

X = 3

Tail

put(x,10) r1

Client
Chain Replication

Client

put(x, 30)

r0

Head

X = 30

req. | UID
---|---
 1. r1
 2. r0

Tail

X = 10

req. | UID
---|---
 1. r1

Client

put(x, 10)

r1
Chain Replication

Client  put(x,30)  r0  Head  X = 30  X = 10  Request  UID

Client  put(x,10)  r1  X = 10  Request  UID

X = 10

req.  uid

r1  1

r0  2

req.  uid

r1  1

r1  1

x=10
Chain Replication

Client

put(x,30)  r0

X = 30  X = 10  X = 10  X = 10

Head  Tail

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>
Chain Replication

Client

put(x,30) r0

Head

Tail

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

X = 30

X = 30

X = 30

X = 10
Chain Replication

Client

```
put(x,30)
```

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

```
r0
```

```
X = 30
```

Head

```
X = 30
```

```
X = 30
```

```
X = 30
```

Tail

```
X = 10
```

```
r0
```

```
X = 10
```

```
r0
```

```
r0
```

```
r0
```

```
r1
```

```
r1
```

```
r1
```

```
r1
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r1
```

```
r1
```

```
r1
```

```
r1
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```

```
r0
```
Chain Replication

Client

put(x,30)

r0

x = 30

Head

X = 30

Tail

X = 30

r0

r0

r1

r1

r0

r0

Req.  UID

r1  1

r0  2

Req.  UID

r1  1

r0  2

Req.  UID

r1  1

r0  2

Req.  UID

r1  1

r0  2

X = 30
Fault Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Head

Tail

\[
X = 30 \rightarrow X = 3 \rightarrow X = 3 \rightarrow X = 3
\]
Fault Tolerance

Dropped requests $r1$ and $r0$
Fault Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Head

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Tail

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>
Fault Tolerance

New tail is *stable* for superset of old tail’s requests
Fault Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

X = 30 → X = 30 → X = 10 → X = 10
Fault Tolerance

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

X = 30

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

X = 10

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

X = 30

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

X = 10
Fault Tolerance

Need to re-send r0
Fault Tolerance

Head

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
<tr>
<td>r0</td>
<td>2</td>
</tr>
</tbody>
</table>

Tail

<table>
<thead>
<tr>
<th>Req.</th>
<th>UID</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>1</td>
</tr>
</tbody>
</table>

Need to re-send r0

How is all of this assignment managed?
Chain Replication
Fault Tolerance

- Trusted Master
  - Fault-tolerant state machine
- Trusted by all replicas
- Monitors all replicas & issues commands
Chain Replication
Fault Tolerance

• Failure cases:
  
  • Head Fails
    
    • *Master* assigns 2nd node as Head
  
  • Tail Fails
    
    • *Master* assigns 2nd to last node as Tail
  
  • Intermediate Node Fails
    
    • *Master* coordinates chain link-up
Chain Replication Evaluation

• Compare to other primary/backup protocols

• Tradeoffs?
  • Latency
  • Consistency

• Trusted Master
Conclusions

• Implements the “exercise left to the reader” hinted at by Lamport’s paper

• Provides some of the concrete details needed to actually implement this idea

  • But still a fair number of details in real implementations that would need to be considered

    • Chain replication illustrates a “simple” example with fully concrete details

• Does some work to justify why such synchronization might be useful (plane actuators)

• A key contribution that bridges the gap between academia and practicality for SMR