Replicated State Machines
Primary-Backup

Arvind Krishnamurthy

University of Washington
Primary-Backup Replication

• Widely used
• Reasonably simple to implement
• Hard to get desired consistency and performance
• Will revisit this and consider other approaches later in the class
Fault Tolerance

- we'd like a service that continues despite failures!
- available: still useable despite *some class of* failures
- strong consistency: act just like a single server to clients
- very useful!
- very hard!
Failure Model

• What do we want to cope with?
  • Independent fail-stop computer failure
  • Site-wide power failure (and eventual reboot)
  • Network partition
  • No bugs, no malice
Core Idea: replication

- Two servers (or more)
- Each replica keeps state needed for the service
- If one replica fails, others can continue
Key Questions

• What state to replicate?
• How does replica get state?
• When to cut over to backup?
• Are anomalies visible at cut-over?
• How to repair/re-integrate?
Two Main Approaches

- State transfer
  - "Primary" replica executes the service
  - Primary sends [new] state to backups
- Replicated state machine
  - All replicas execute all operations
  - If same start state, same operations, same order, deterministic → then same end state

- There are tradeoffs: complexity, costs, consistency
Design Space

• Active or passive replicas
• Symmetric replicas or primary-backup
• Replicate commands or low-level inputs
State Machines

- $c$ is a Command
- $f$ is a Transition Function
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
SMR Requirements

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

Great!
SMR Requirements

put(x, 10)
**SMR Requirements**

- Replicas need to agree on the which requests have been handled.
SMR Requirements

\[ \text{put}(x, 10) \quad r0 \]
\[ \text{put}(x, 30) \quad r1 \]
SMR Requirements

OR

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

OR

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

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

\[
\text{put}(x, 30) \\
r_1
\]
SMR Requirements

\[ \text{put}(x, 10) \]
\[ r_0 \quad \text{r0} \]
\[ \text{X} = 10 \]
\[ \text{X} = 30 \]
\[ r_1 \quad \text{r1} \]

\[ \text{put}(x, 30) \]
\[ r_1 \quad \text{r1} \]
\[ r_0 \]
\[ \text{r0} \]
SMR Requirements

- 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

• 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

- put(x,30) → r0
- put(x,10) → r1
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
Replica Generated IDs

• 2 Phase ID generation
  • Every Replica proposes a candidate
  • One candidate is chosen and agreed upon by all replicas
1) Propose Candidates
### 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></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>
<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>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 $r1$

<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>$r0$</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>1.3</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>
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>

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>

<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>1.4</td>
<td>2.2</td>
</tr>
<tr>
<td>r0</td>
<td>2.4</td>
<td>2.4</td>
</tr>
</tbody>
</table>

5) Apply $r0$
Chain Replication

- Fault Tolerant Storage Service (Fail-Stop)
- Requests:
  - \( \text{Update}(x, y) \Rightarrow \text{set object } x \text{ to value } y \)
  - \( \text{Query}(x) \Rightarrow \text{read value of object } x \)
Chain Replication

\[
\begin{array}{cc}
X = 3 & X = 3 \\
X = 3 & X = 3
\end{array}
\]
Chain Replication

```
get(x)
```

Client

Head

Tail

X = 3
Chain Replication

Client

put(x, 30)
Chain Replication

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

1) Head assigns \textit{uid}

Client

\textbf{put}(x,30)
Chain Replication

2) Head sends message to next node

Client

put(x,30)

Head

Tail

 Req.  UID
r0    1

 Req.  UID
r0    1

X = 30

X = 30

X = 3

X = 3
Chain Replication

Client

put(x, 30)

3) Repeat until tail is reached

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

Head

Tail

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

X = 30

X = 30

X = 30

X = 3
Chain Replication

Client

\text{put}(x, 30)

\begin{tabular}{|c|c|}
\hline
\text{Req.} & \text{UID} \\
\hline
r0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\text{Req.} & \text{UID} \\
\hline
r0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\text{Req.} & \text{UID} \\
\hline
r0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
\text{Req.} & \text{UID} \\
\hline
r0 & 1 \\
\hline
\end{tabular}

4) respond to client with success

\begin{tabular}{|c|c|c|}
\hline
\text{X} = 30 & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\text{X} = 30 & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\text{X} = 30 & \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
\text{X} = 30 & \\
\hline
\end{tabular}
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 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: `put(x,30)`

Put operations: `put(x,10)`

Diagram showing chain replication with values `X = 3` in each node.
**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>

- **Head**: $X = 10$
- **Client**: put($x,30$)
- **Tail**: $X = 3$
- **Client**: put($x,10$)

- Request $r1$ with UID 1
- Request $r0$ with UID 2
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)

put(x, 10)

Head

X = 30

Tail

X = 3

Request UID

r1

r0

r1

r1

r0
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

<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)

put(x, 10)

Client

x = 10

Head

X = 30

Tail

X = 10

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

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

• Trusted Master
  • Fault-tolerant state machine
  • Trusted by all replicas
  • Monitors all replicas & issues commands
• How can you rely on this trusted master?
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
VMware’s FT Virtual Machines

- Whole-system replication
- Completely transparent to applications and clients
- High availability for any existing software
- Failure model:
  - independent hardware faults
  - site-wide power failure
- Limited to uniprocessor VMs
Overview

- two machines, primary and backup
- shared-disk for persistent storage
- back-up in "lock step" with primary
  - primary sends all inputs to backup
  - outputs of backup are dropped
- heart beats between primary and backup
  - if primary fails, start backup executing!
Challenges

• Making it look like a single reliable server
• How to avoid two primaries? ("split-brain syndrome")
• How to make backup an exact replica of primary
• What inputs must be sent to backup?
• How to deal with non-determinism?
Technique 1: Deterministic Replay

• Goal: make x86 platform deterministic
  • idea: use hypervisor to make virtual x86 platform deterministic

• Log all hardware events into a log
  • clock interrupts, network interrupts, i/o interrupts, etc.
  • for non-deterministic instructions, record additional info
    • e.g., log the value of the time stamp register
    • on replay: return the value from the log instead of the actual register
Deterministic Replay

- Replay: deliver inputs in the same order, at the same instructions
  - if during recording delivered clock interrupt at nth instr.
  - during replay also deliver the interrupt at the nth instr.
- Given an event log, deterministic replay recreates VM
  - hypervisor delivers first event
  - lets the machine execute to the next event
  - using special hardware registers to stop the processor at the right instruction
  - OS runs identical, applications runs identical
- Limitation: cannot handle multicore processors and interleaving
Applying Deterministic Replay to VM-FT

- Hypervisor at primary records
  - Sends log entries to backup over logging channel
- Hypervisor at backup replays log entries
  - We need to stop virtual x86 at instruction of next event
  - We need to know what is the next event
  - backup lags behind one event
Example

- Primary receives network interrupt
  - hypervisor forwards interrupt plus data to backup
  - hypervisor delivers network interrupt to OS kernel
  - OS kernel runs, kernel delivers packet to server
  - server/kernel write response to network card
  - hypervisor gets control and puts response on the wire
- Backup receives log entries
  - backup delivers network interrupt
  - ...
  - hypervisor does *not* put response on the wire
  - hypervisor ignores local clock interrupts
Technique 2: FT Protocol

• Primary delays any output until the backup acks
  • Log entry for each output operation
  • Primary sends output after backup acked receiving output operation

• Performance optimization:
  • primary keeps executing past output operations
  • buffers output until backup acknowledges
Questions

• Why send output events to backup and delay output until backup has acked?
• What happens when primary fails after receiving network input but before sending a corresponding log entry to backup?