Replication Management using the State-Machine Approach
Fred B. Schneider

Summary and Discussion:
Hee Jung Kim and Ying Zhang

October 27, 2005
✓ Introduction
✓ State Machines
✓ Fault Tolerance
✓ Fault-tolerant State Machines
✓ Tolerating Faulty Output Devices
✓ Tolerating Faulty Clients
✓ Using Time to Make Request
✓ Reconfiguration
Introduction

- Why Replication?
- Two kinds of replication are..
- State machine Approach is..
- What can be discussed in each sections
State-Machine Approach

✓ A general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas.
State Machines

✓ State machine consist of
  - State Variables
  - Commands.

✓ Command might be implemented by
  - Sharing data amongst procedures,
  - Queuing requests
  - Using interrupt handlers.
Assumption!

✓ Requests from clients processed in causal order.
   - O1: Requests issued by a single client processed by $sm$ in the order they are issued
   - O2: $r1$ could have caused $r2 \Rightarrow r1$ processed by $sm$ before $r2$
“Outputs of a state machine are completely determined by the sequence of requests it processes, independent of time or any other activity of a system”
Is this a state machine?

pc: state-machine
  var q: real;
  adjust: command(sensor-val: real)
    q := F(q, sensor-val);
    send q to actuator
  end adjust
end pc

monitor: process
  do true -> val := sensor;
    <pc.adjust, val>;
    delay D
  od
end monitor
Fault Tolerance

✓ Byzantine failures: “arbitrary and malicious”
✓ Failstop failures: “other components [can] detect that a failure has occurred”
A system consisting of a set of distinct components is $t$ fault-tolerant if it satisfies its specification provided that no more than $t$ of those components become faulty during some interval of interest.
Fault-tolerant SM

✔ Replicate State Machines and run on separate processors.
✔ Each replica
  - Starts in the same initial state
  - Executes same requests in the same order
✔ Assuming independent failure
✔ Combine outputs of the replicas of this ensemble.
✓ Replica Coordination
All replicas receive and process the same sequence of requests.

- Agreement:
  Each Non-Fault replica receives every request.

- Order:
  Each Non-Fault replica processes the requests in the same relative order.
Agreement

Any protocol that allows a designated processor called the *transmitter* so that:

- IC1: All non-faulty processors agree on the same value.
- IC2: If the transmitter is non-faulty, then all non-faulty processors use its value as the one on which they agree.
Order and Stability

Order requirement can be satisfied by

- Assigning unique ids to requests.
- Processing the requests according to a total ordering on the unique ids.
Order Implementation

“A replica next processes the stable request with smallest unique ids.”

- Using Logical Clocks.
- Synchronized Real-Time Clocks.
- Using Replica-Generated Identifiers.
Using Logical Clocks

✓ A logical clock is a mapping \( T \) from events to the integers.

✓ LC1: \( T_p \) is incremented after each event at \( P \).

✓ LC2: Upon receipt of a message - with timestamp \( ts \), process \( p \) resets \( T_p \):
  \[ T_p := \max(T_p, ts) + 1. \]
Using Logical Clocks

✓ Assumption to property of communication channels.
  - FIFO channels between processors
  - Failure Detection Assumption (for fail-stop processors): A processor $p$ detects that a fail-stop processor $q$ has failed only after $p$ has received the last message sent to $p$ by $q$. 
Logical Clocks Stability Test

✓ Every client periodically makes some-possibly null-request to the state machine.

✓ Request stable at $smi$ if a request with larger timestamp has been received from every client running on a non-faulty processor.
Synchronized Real-time Clocks

- $T_p(e)$ : the real-time clock at processor $p$ when event $e$ occurs.
- Unique id: $T_p(e)$ appended by fixed bit string that uniquely identifies $p$.
- $O_1$ satisfied if only one request in between successive clock ticks
- $O_2$ satisfied if degree on synchronization is better than the minimum message delivery time.
Synchronized Real-time Clocks (cont’d)

✓ **Real-time Clock Stability Test I**
  
r is stable at $smi$ executed at $p$ if the local clock at $p$ reads $ts$ and $uid(r) < ts - td$

✓ **Real Clock Stability Test II**
  
r is stable at $smi$ if a request with larger $uid$ has been received from every client.
Using Replica-Generated Ids.

✓ Unique ids assigned by the replicas
  Two phase protocol
  - Replicas propose candidate unique ids
  - One candidate is selected
✓ Elaboration of the protocol
  - Seen: \textit{smi} has seen \textit{r} once it has received \textit{r} and proposed a candidate unique id for it.
  - Accepted: \textit{smi} has accepted \textit{r} once it knows the final choice of \textit{uid}(r).
Using Replica-Generated Ids.

✓ Constraints on the proposed ids\(\text{cuid}(smi,r)\)
  - UID1: \(\text{cuid}(smi,r) \leq \text{uid}(r)\)
  - UID2: if \(r'\) SEEN at \(smi\) after \(r\) has been accepted then \(\text{uid}(r) < \text{cuid}(smi,r')\)

✓ Replica-Generated Id Stability Test:
  \(r\) that has been accepted by \(smi\) is stable provided there is no request \(r'\) that has
  i) Been seen by \(smi\)
  ii) Not been accepted by \(smi\)
  iii) \(\text{cuid}(smi,r') \leq \text{uid}(r)\)
Using Replica-Generated Ids.

✓ Replica-generated Unique Identifiers:

- \( sm_i \) maintains
  - \( \text{SEEN}_i \): largest \( \text{cuid}(sm_i, r) \) so far assigned by \( sm_i \)
  - \( \text{ACCEPT}_i \): largest \( \text{uid}(r) \) so far assigned by \( sm_i \) on receipt of \( r \)
  - \( \text{cuid}(sm_i, r) = \max(\_) + 1 + i \)
  - Disseminates \( \text{cuid}(sm_i, r) \) to other replicas, awaits receipt of a candidate uid from every non-faulty replica.
  - \( \text{uid}(r) = \max_j(\text{cuid}(sm_j, r)) \)
Tolerating Faulty Output Devices

✔ Outputs used outside system:
   Use replicated voters and output devices.

✔ Outputs used inside system:
   The client need not gather a majority of responses to its request to the state machine. It can use the single response produced locally.
Tolerating Faulty Clients

✓ Replicate the client
  - However, requires changes to state machines that handle requests from that client.

✓ Defensive programming
  - Sometimes, a client cannot be made fault-tolerant by using replication.
  - Careful design of state machine can limit the effects of requests from faulty clients.
Using Time to Make Request

✓ Assume that
  - All clients and state machine replicas have clocks synchronized to within $r$, and
  - Election starts at time $strt$ and known to all clients and state machine replicas.

✓ Transmitting a default vote
  - If client has not made a request by time $strt + r$, then a request with that client’s default vote has been made.
An ensemble of state machine replicas can tolerate more than $t$ faults if it is possible to remove state machine replicas running on faulty processors from the ensemble and add replicas running on repaired processors.”
Reconfiguration

✓ Combining Condition:

\[ P(t) - F(t) > X \text{ for all } 0 \leq t \]

where \( X \):

- \( P(t)/2 \) (Byzantine failure)
- \( 0 \) (fail-stop failure)

\( P(t) = \text{total number of processors at time } t \)
\( F(t) = \text{faulty number of processors at time } t \)
Unbounded total number of faults is possible if...

F1: Byzantine failures, removed faulty replica from the ensemble before the Combining Condition is violated by subsequent processor failures.

F2: Replicas running on repaired processors are added to the ensemble before the Combining Condition is violated by subsequent processor failures.
Configuration

The configuration of the system is defined as:

- **C**: The clients
- **S**: The state-machine replicas
- **O**: The output devices

To change system configuration:
- the value of C, S, O must be available
- whenever C, S, O added, state must be updated
Managing Configuration

A non-faulty configurator satisfies

C1: Only a faulty element is removed from the configuration.
C2: Only a non-faulty element is added to the configuration.
Integration with Failstop Processors and Logical Clocks

If $e$ is a client or output device, then $sm_i$ sends the state variables to before sending any output with $ids > r_{\text{join}}$.

If $e$ is a state-machine replica, $sm_{\text{new}}$, then $sm_i$:  
1. sends state variables and copies of any pending requests to $sm_{\text{new}}$, 
2. sends $sm_{\text{new}}$ subsequent request $r$ received from $c$ such that $\text{uid}(r) < \text{uid}(r_c)$, where $r_c$ is the first request that $sm_{\text{new}}$ received directly from $c$ after being restarted.
Integration with Failstop Processors and Realtime Clocks

If $e$ is a client or output device, then $s_{mi}$ sends the state variables to before sending any output with $ids > r_{join}$.

If $e$ is a state-machine replica, $s_{new}$, then $s_{mi}$:
1. sends state variables and copies of any pending requests to $s_{new}$,
2. sends to $s_{new}$ every request received during the next interval of duration.

Simplified!!
Stability Revised

When requests made by a client can be received from two sources—the client and via a relay. The stability test must be changed...

Stability Test During Restart:

\( r \) received directly from \( c \) by a restarting \( sm_{new} \) is stable only after the last request from \( c \) relayed by another processor has been received by \( sm_{new} \)
Summary

- State Machines approach is ..
- Coping with failures (Byzantine, Failstop) ..
  - Fault-tolerant State Machines
  - Tolerating Faulty Output Devices
  - Tolerating Faulty Clients
- Optimization :
  - Using time to request
- Dynamic reconfiguration
  - Managing the configuration
  - Integrating a repaired object
Thank you !!!

Any question ???