Replication State Machines via Primary-Backup

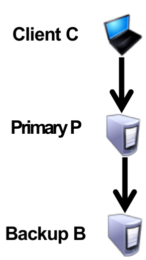
COS 418: *Distributed Systems*
Lecture 10

Michael Freedman

From eventual to strong consistency

- Eventual consistency
  - Multi-master: Any node can accept operation
  - Asynchronously, nodes synchronize state
- Eventual consistency inappropriate for many applications
  - Imagine NFS file system as eventually consistent
  - NFS clients can read/write to different masters, see different versions of files
- Stronger consistency makes applications easier to write
  - (More on downsides later)

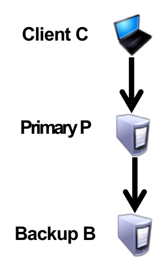
Primary-Backup Replication



- **Mechanism:** Replicate and separate servers
- **Goal #1:** Provide a highly reliable service (despite failures)
- **Goal #2:** Servers should behave just like a single, more reliable server

---

Primary-Backup Replication



- Nominate one replica ***primary,*** other ***backup***
  - Clients send all ops to current **primary**
  - Primary **orders** clients' operations
- Only **one primary at a time**

Need to keep clients, primary, and backup in sync:
who is primary and **who is backup**

State machine replication

- **Idea: A replica** is essentially a ***state machine***
  - Set of (key, value) pairs is **state**
  - Operations **transition** between states
- Need an op to be executed on all replicas, or none at all
  - *i.e.,* we need **distributed all-or-nothing atomicity**
  - If op is deterministic, replicas will end in same state
- **Key assumption:** Operations are **deterministic**

5

More reading: ACM Computing Surveys, Vol. 22, No. 4, December 1990 (pdf)

Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial

FRED B. SCHNEIDER

Department of Computer Science, Cornell University, Ithaca, New York 14853

The state machine approach is a general method for implementing fault-tolerant services in distributed systems. This paper reviews the approach and describes protocols for two different failure models—Byzantine and fail stop. System reconfiguration techniques for removing faulty components and integrating repaired components are also discussed.

Categories and Subject Descriptors: C.2.4 **[Computer-Communication Networks]**: Distributed Systems—*network operating systems*; D.2.10 **[Software Engineering]**: Design—*methodologies*; D.4.5 **[Operating Systems]**: Reliability—*fault tolerance*; D.4.7 **[Operating Systems]**: Organization and Design—*interactive systems, real-time systems*

General Terms: Algorithms, Design, Reliability

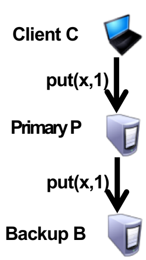
Additional Key Words and Phrases: Client-server, distributed services, state machine approach

INTRODUCTION

Distributed software is often structured in terms of *clients* and *services*. Each service comprises one or more *servers* and exports *operations* that clients invoke by making *requests*. Although using a single, centralized, server is the simplest way to implement service by replicating servers and coordinating client interactions with server replicas.[1] The approach also provides a framework for understanding and designing replication management protocols. Many protocols that involve replication of data or software—be it for masking failures

6

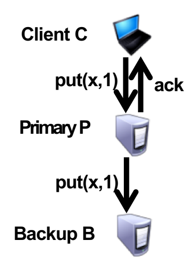
Primary-Backup Replication



1. Primary gets operations
2. Primary orders ops into log
3. Replicates log of ops to backup
4. Backup exec's ops or writes to log

7

Primary-Backup Replication

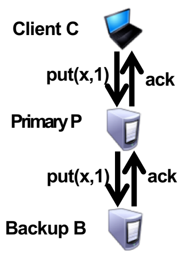


Asynchronous Replication

1. Primary gets operations
2. **Primary exec's ops**
3. Primary orders ops into log
4. Replicates log of ops to backup
5. Backup exec's ops or writes to log

8

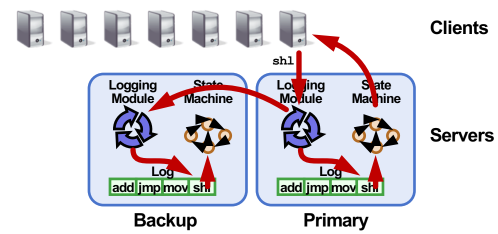
Primary-Backup Replication



Synchronous Replication

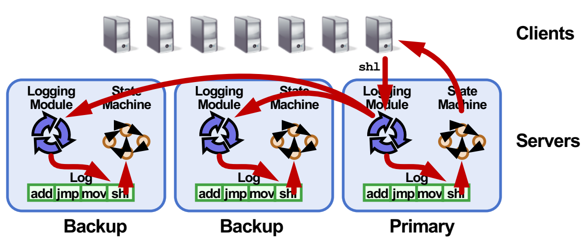
1. Primary gets operations
2. Primary orders ops into log
3. Replicates log of ops to backup
4. Backup exec's op or writes to log
5. **Primary gets ack, execs ops**

Why does this work? **Synchronous Replication**



- Replicated log => replicated state machine
  - All servers execute same commands in same order

Why does this work? **Synchronous Replication**



- Replicated log => replicated state machine
  - All servers execute same commands in same order

Need determinism? Make it so!

- Operations are deterministic
  - No events with ordering based on local clock
    - Convert timer, network, user into logged events
  - Nothing using random inputs
- Execution order of ops is identical
  - Most RSMs are single threaded

Example: Make random() deterministic

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

`random.`**`seed`**(*a=None*)
Initialize internal state of the random number generator.

`None` or no argument seeds from current time or from an operating system specific randomness source if available (see the `os.urandom()` function for details on availability).

`random.`**`getstate`**()
Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

Example: Make random() deterministic

- Primary:
  - Initiates PRNG with OS-supplied randomness, gets initial seed
  - Sends initial seed to to backup
- Backup
  - Initiates PRNG with seed from primary

`random.`**`seed`**(*a=None*)
Initialize internal state of the random number generator.

`None` or no argument seeds from current time or from an operating system specific randomness source if available (see the `os.urandom()` function for details on availability).

`random.`**`getstate`**()
Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

Case study

The design of a practical system for fault-tolerant virtual machines

D. Scales, M. Nelson, G. Venkitachalam, VMWare

SIGOPS Operating Systems Review 44(4), Dec. 2010 (pdf)

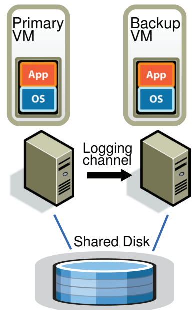
VMware vSphere Fault Tolerance (VM-FT)

Goals:

1. Replication of the **whole virtual machine**
2. **Completely transparent** to apps and clients
3. **High availability** for any existing software

vSphere Overview

- Two virtual machines (***primary***, ***backup***) on different bare metal
- ***Logging channel*** runs over network
- ***Shared disk*** via fiber channel

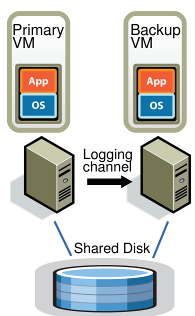


Virtual Machine I/O

- **VM inputs**
  - Incoming network packets
  - Disk reads
  - Keyboard and mouse events
  - Clock timer interrupt events
- **VM outputs**
  - Outgoing network packets
  - Disk writes

Overview

- **Primary** sends **inputs** to backup
- **Backup outputs** dropped
- Primary-backup **heartbeats**
  - If primary fails, backup takes over



VM-FT: Challenges

1. **Making the backup an exact replica of primary**
2. Making the system behave like a single server
3. Avoiding two primaries (Split Brain)

Log-based VM replication

- **Step 1:** Hypervisor at primary logs causes of non-determinism

1. Log results of **input events**
   - Including current program counter value for each
2. Log results of **non-deterministic instructions**
   - *e.g.* log **result** of timestamp counter read

21

Log-based VM replication

- **Step 2:** Primary hypervisor sends log entries to backup
- Backup hypervisor **replays** the log entries
  - **Stops backup VM** at next input event or non-deterministic instruction
    - Delivers **same input** as primary
    - Delivers **same non-deterministic instruction result** as primary

22

VM-FT Challenges

1. Making the backup an exact replica of primary
2. **Making the system behave like a single server**
   - **FT Protocol**
3. Avoiding two primaries (Split Brain)

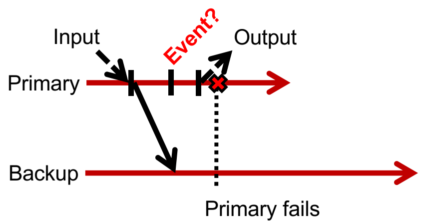
23

Primary to backup failover

- When backup takes over, non-determinism makes it **execute differently** than primary would have
  - **This is okay!**
- **Output requirement**
  - When backup takes over, execution is **consistent** with **outputs** the primary has already sent

24

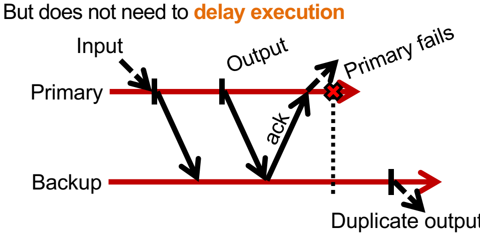
The problem of inconsistency



25

VM-FT protocol

- Primary logs each output operation
- Delays sending output until Backup acknowledges it
  - But does not need to **delay execution**

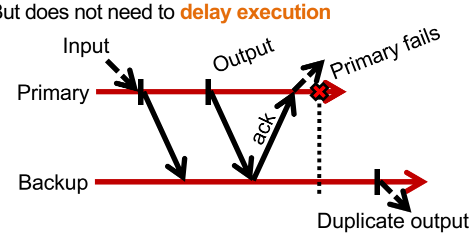


26

VM-FT protocol

"If a tree falls in forest" metaphor: If event happens and nobody sees it yet, did it really happen?

- Primary logs each output operation
- Delays sending output until Backup acknowledges it
  - But does not need to **delay execution**



Can restart execution at an output event

27

VM-FT: Challenges

1. Making the backup an exact replica of primary
2. Making the system behave like a single server
3. **Avoiding two primaries (Split Brain)**
   - Logging channel may **break**

28

Detecting and responding to failures

- Primary and backup each run UDP heartbeats, monitor logging traffic from their peer
- Before “going live” (backup) or finding new backup (primary), execute **atomic test-and-set** on variable in shared storage
- If the replica finds variable already set, it **aborts**

29

VM-FT: Conclusion

- Challenging application of primary-backup replication
- Design for correctness and consistency of replicated VM outputs despite failures
- Performance results show generally **high performance, low logging bandwidth overhead**

30