Replication State Machines via Primary-Backup

CS 240: Computing Systems and Concurrency
Lecture 10

Marco Canini

From eventual to strong consistency

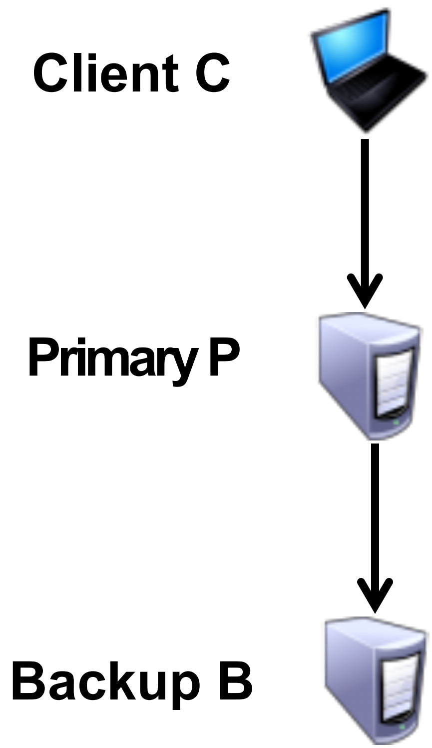
- Eventual consistency
  - Multi-master: Any node can accept operation
  - Asynchronously, nodes synchronize state
- Eventual consistency inappropriate for many applications
  - Imagine bank ledger as eventually consistent
- Stronger consistency makes applications easier to write

Plan

1. **Introduction to Primary-Backup replication**

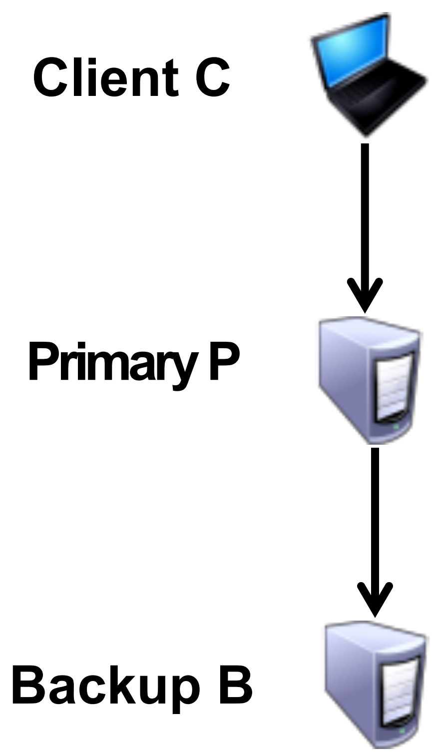
2. Case study: VMWare’s fault-tolerant virtual machine

Primary-Backup Replication



- **Mechanism:** Replicate and separate servers
- **Goal #1:** Reliable despite individual server failures
- **Goal #2:** Semantics of a single server

Primary-Backup Replication



- Nominate one replica ***primary***
- Other replicas are ***backup***
  - Only **one primary at a time**
- Clients send all operations to current **primary**
- Primary **orders** clients' operations

Need to keep clients, primary, and backup in sync: **who is primary** and **who is backup** (no Split Brain)

State machine replication

- **Insight: A replica** is essentially a ***state machine***
  - E.g., set of (key, value) pairs is **state**
  - Operations **transition** between states

- Each operation executed on **all** replicas, or **none at all**
  - *i.e.,* we need **distributed** **all-or-nothing atomicity**

- **Key assumption:** Operations are **deterministic**

- If op is deterministic, replicas will end in same state

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 imple-

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 or simply to facilitate cooperation without

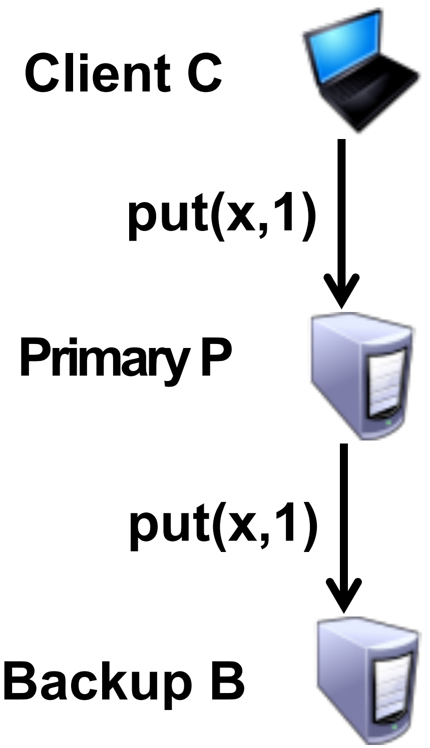
Replica coordination

All non-faulty state machines receive all operations in the same order

- **Agreement**: Every non-faulty state machine receives every operation

- **Order**: Every non-faulty state machine processes the operations it receives in the same order

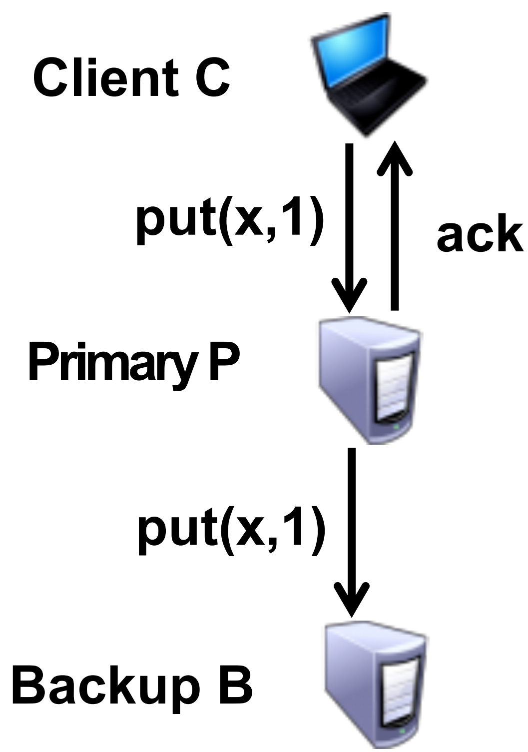
Primary-Backup Replication



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

When does the primary execute ops?

Primary-Backup Replication

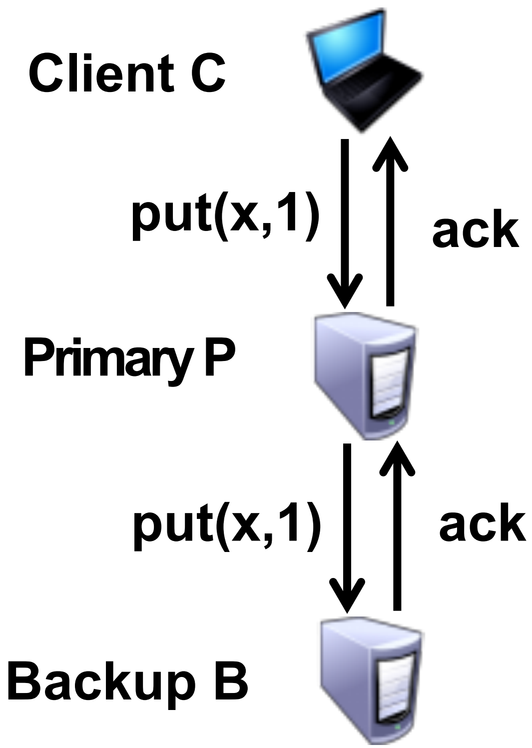


Asynchronous Replication

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

What can go wrong w/ async replication?

Primary-Backup Replication



Synchronous Replication

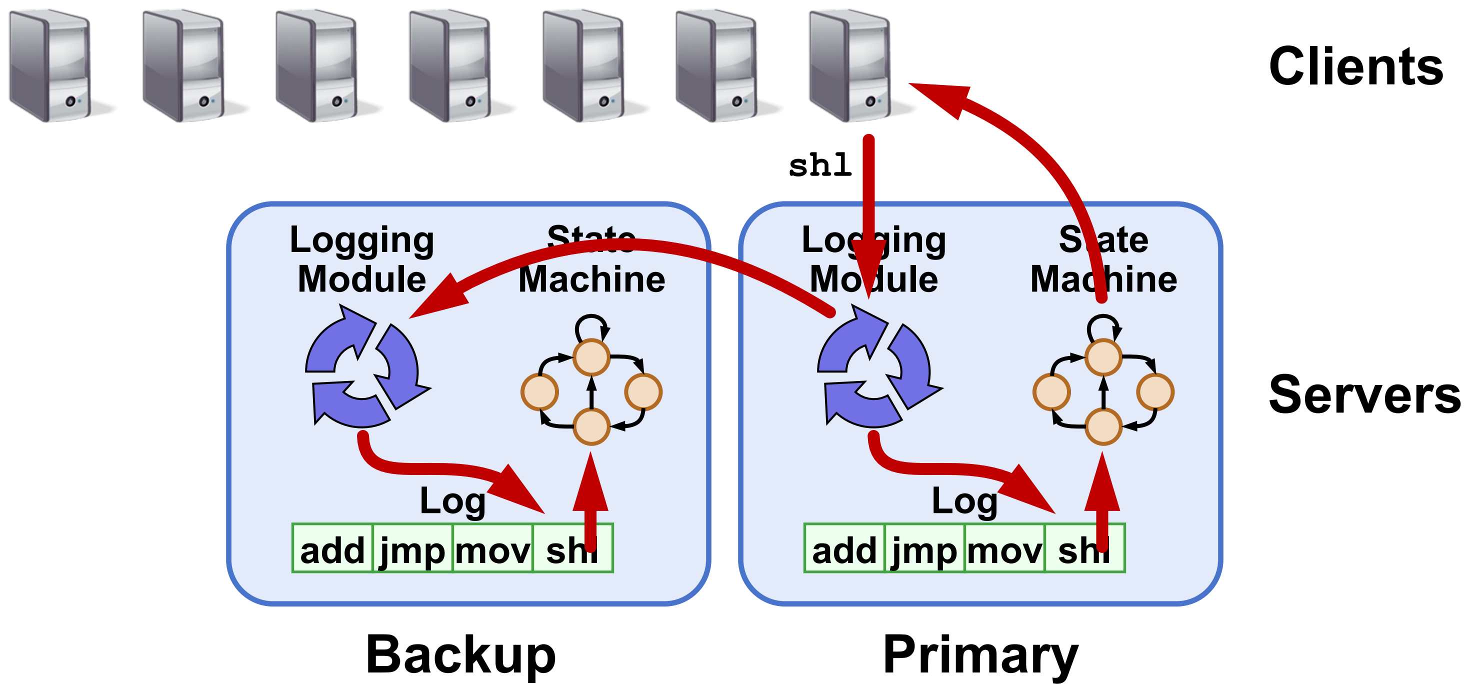
1. Primary gets operations
2. Primary orders ops into log
3. Replicates log of ops to backup
4. Backup exec's op or writes to log
5. **Primary gets ack, execs ops**

If the primary fails? A backup is promoted to new primary

Hot vs. Cold backups

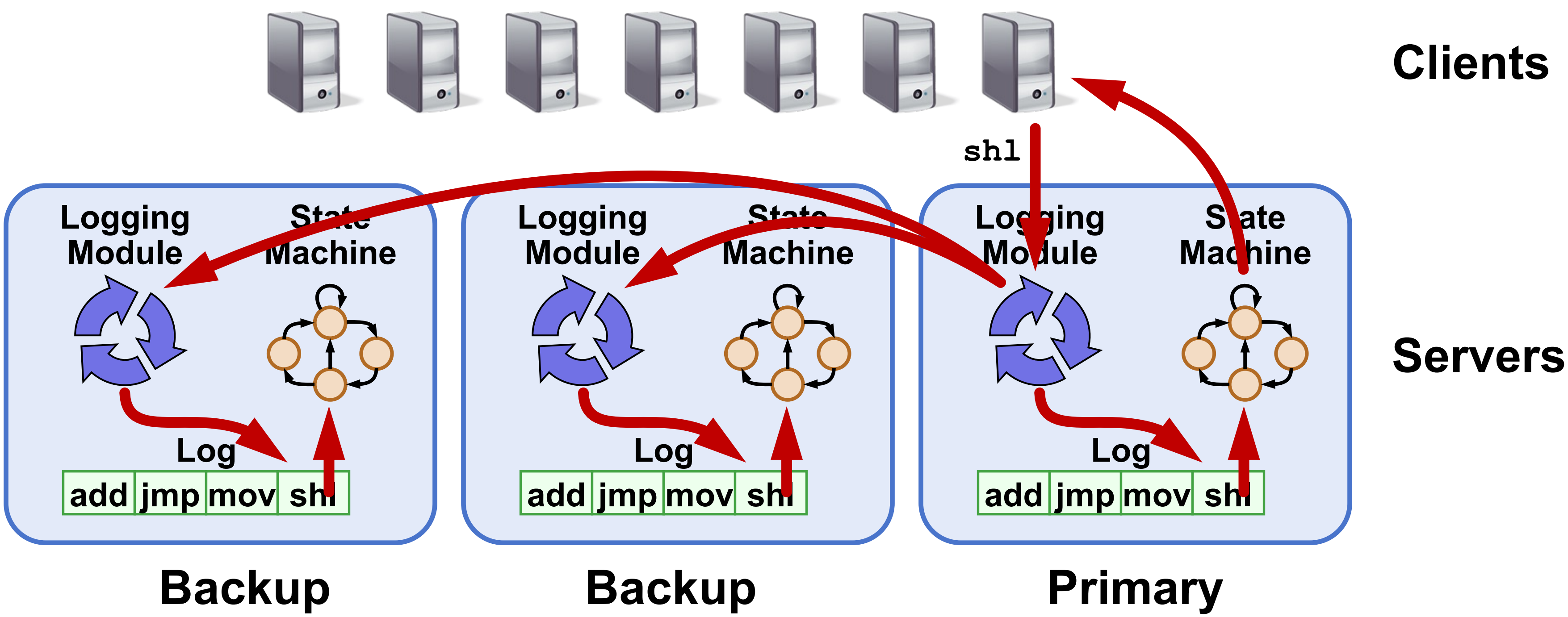
- “Backup exec’s op or writes to log”
- **Hot** Backups execute operations from the primary as soon as they receive it
- **Cold** Backups log operations received from primary, and execute them only if primary fails

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
  - e.g. replicated state machines (RSMs) is 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.

Primary-Backup: Summary

- First step in our goal of making **stateful** replicas **fault-tolerant**

- Allows replicas to provide **continuous service** despite **persistent net and machine failures**

- Finds repeated application in **practical systems**

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)

Where should RSM be implemented?

- In hardware
  - Sensitive to architecture changes
- At the OS level
  - State transition are hard to track and coordinate
- At the application level
  - Requires sophisticated application programmers

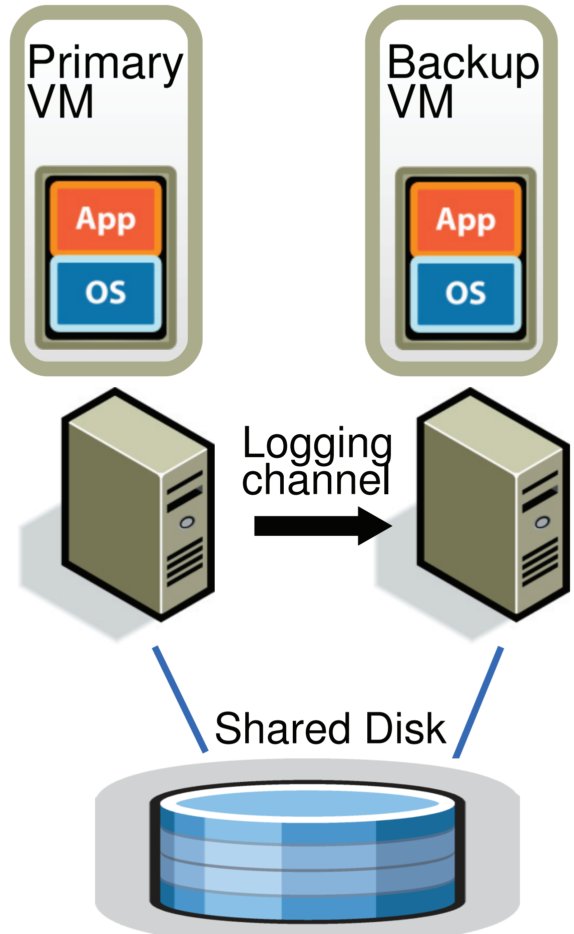
VMware vSphere Fault Tolerance (VM-FT)

Goals:

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

Overview

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



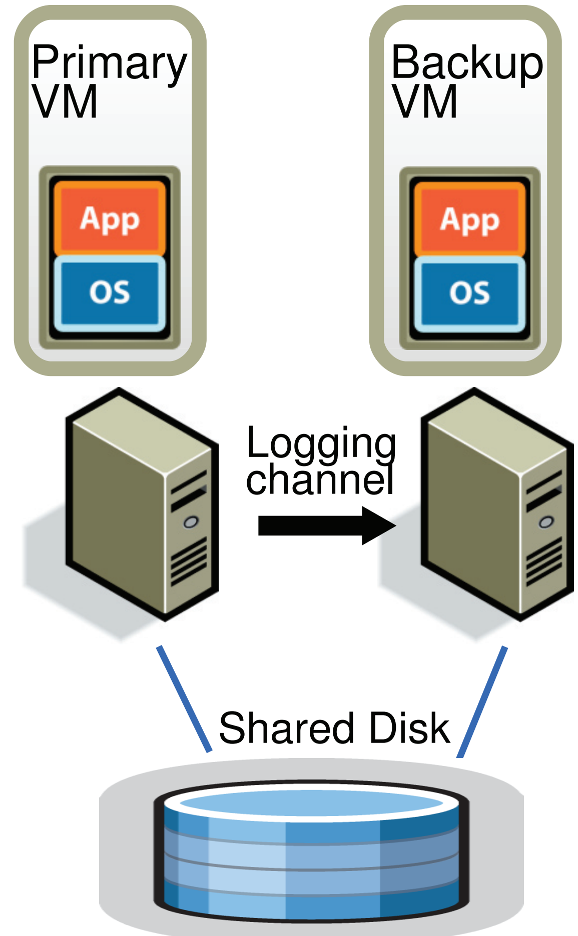
Virtual Machine I/O

- **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 the 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

Log-based VM replication

- **Step 2:** Primary hypervisor sends log entries to backup hypervisor

- 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

VM-FT Challenges

1. Making the backup an exact replica of primary

2. **Making the system behave like a single server**
   - **FT Protocol**

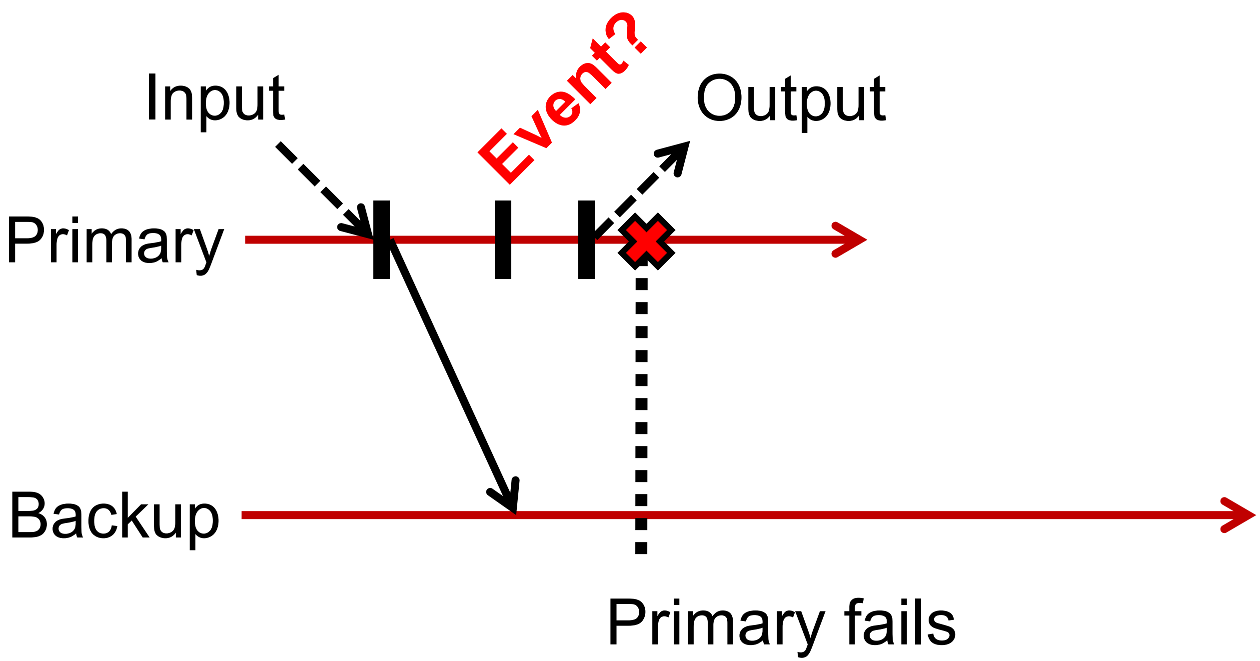
3. Avoiding two primaries (Split Brain)

Primary to backup failover

- When backup takes over, non-determinism makes it **execute differently** than primary would have
  - **This is okay!**

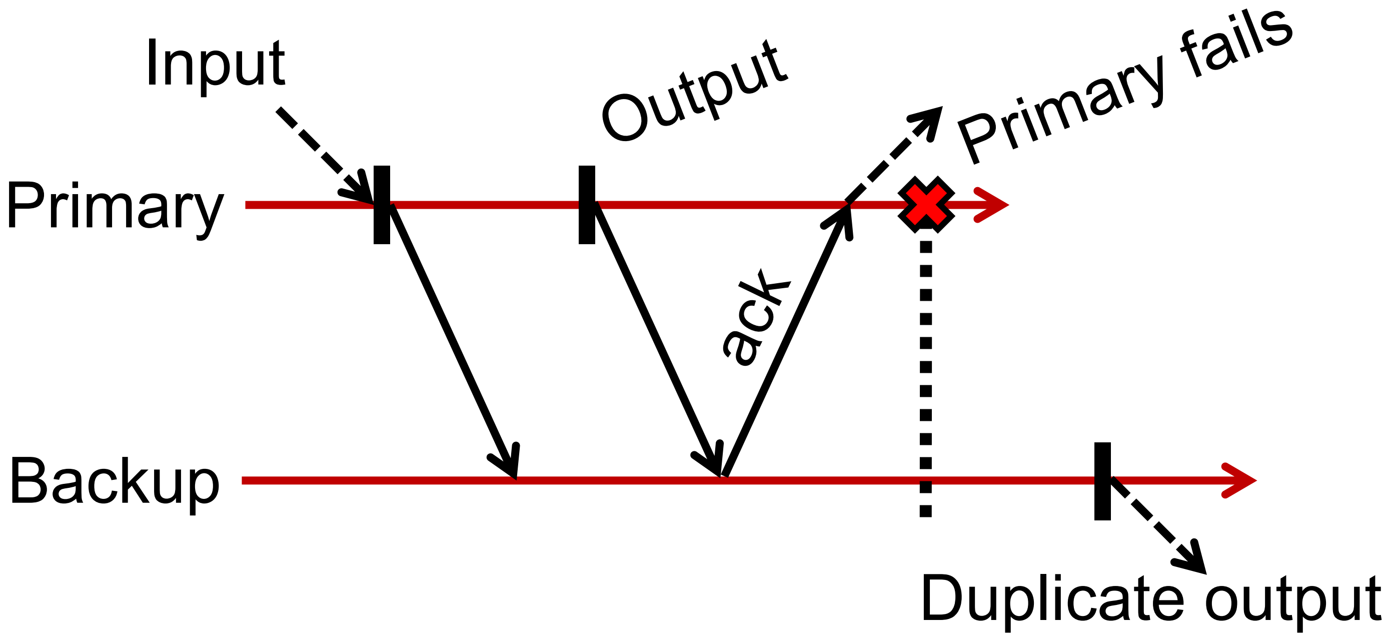
- **Output requirement**
  - When backup takes over, execution is **consistent** with **outputs** the primary has already sent

The problem of inconsistency



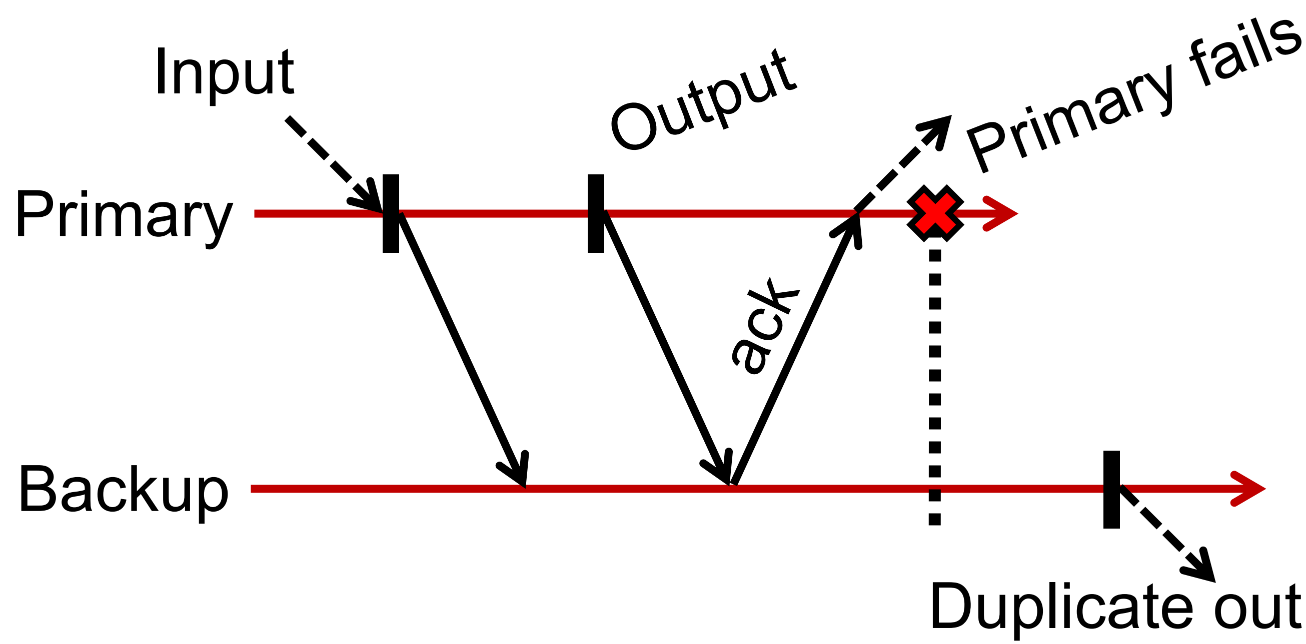
VM-FT protocol

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



VM-FT protocol

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



Restart execution at an output event

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**

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 an **atomic test-and-set** on a variable in shared storage
- If the replica finds variable already set, it **aborts**

VM-FT: Summary

- Primary-backup replication with whole machines!
- Design for correctness and consistency of replicated VM outputs despite failures
  - Determinism tricky, but doable
  - Primary delays output until acked by backup
  - Use atomic test-and-set on shared disk to avoid split brain

Primary-Backup: Take-away ideas

- All replicas receive and process the same sequence of (deterministic) operations

  - Clients send all operations to current primary
  - Primary **orders** clients' ops into a **log**
  - Primary **replicates** the log to backup
  - Backup executes ops

  - Async vs Sync replication: when the primary executes

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