ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Tanenbaum and Van Steen, Dist. Systems: Principles and Paradigms
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS

Today’s material: Tanenbaum & Van Steen, sec. 7.5.2

Optional reading:
Outline

1. Primary-backup replication

2. Case study:
   • VMWare fault tolerance
LIMITED FAULT TOLERANCE IN TOTALLY-ORDERED MULTICAST

• Stateful server replication for fault tolerance...
• But no story for server replacement upon a server failure → no replication

Today: Make stateful servers fault-tolerant?
PRIMARY-BACKUP: GOALS

- **Mechanism:** Replicate and separate servers

- **Goal #1:** Provide a highly reliable service
  - Despite some server and network failures
  - **Continue operation after failure**

- **Goal #2:** Servers should behave just like a single, more reliable server
STATE MACHINE REPLICATION

- **Any server** 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
- We will relax this assumption later today
PRIMARY-BACKUP (P-B) APPROACH

- Nominate one server the **primary**, call the other the **backup**
- Clients send all operations (get, put) to current primary
- The primary **orders** clients’ operations
- Should be only **one primary at a time**

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

- Network and server **failures**

- Network **partitions**
  - Within each network partition, near-perfect communication between servers

- Between network partitions, **no communication between servers**
1. Primary logs the operation locally

2. Primary sends operation to backup and waits for ack
   - Backup performs or just adds it to its log

3. Primary performs op and acks to the client
   - After backup has applied the operation and ack’ed
A **view server** decides who is primary, who is backup

- Clients and servers depend on view server
  - Don’t decide on their own (might not agree)

**Challenge in designing the view service:**

- Only want one primary at a time
- Careful protocol design needed

For now, **assume** view server **never fails**
MONITORING SERVER LIVENESS

• Each replica periodically **pings** the view server

• View server declares replica **dead** if it missed N pings in a row

• Considers the replica **alive** after a single ping

• **Can a replica be alive but declared “dead” by view server?**

• Yes, in the case of network failure or partition
THE VIEW SERVER DECIDES THE CURRENT VIEW

- **View** = (view #, primary server, backup server)

Challenge: All parties make their own local decision of the current view number
AGREEING ON THE CURRENT VIEW

- In general, any number of servers can ping view server
- Okay to have a view with a primary and no backup
- Want everyone to agree on the view number
- Include the view # in RPCs between all parties
TRANSITIONING BETWEEN VIEWS

- How to ensure new primary has up-to-date state?
  - Only promote a previous backup
    - *i.e.*, don’t make a previously-idle server primary
  - Set liveness detection timeout > state transfer time

- How does view server know whether backup is up to date?
  - View server sends `view-change` message to all
  - Primary **must ack new view** once backup is up-to-date
  - View server stays with current view until ack
    - Even if primary has or **appears** to have failed
SPLIT BRAIN

View Server

(1, S₁, S₂)
(2, S₂, −)

S₁

(1, S₁, S₂)

S₂

(2, S₂, −)

Client
SERVER $S_2$ IN THE OLD VIEW

View Server

(1, $S_1$, $S_2$)
(2, $S_2$, -)
SERVER $S_2$ IN THE NEW VIEW

**View Server**

$$(1, S_1, S_2)$$
$$(2, S_2, -)$$

**Client**

$$(1, S_1, S_2)$$
$$(2, S_2, -)$$
STATE TRANSFER VIA OPERATION LOG

- How does a new backup get the current state?
- If $S_2$ is backup in view $i$ but was not in view $i-1$
- $S_2$ asks primary to transfer the state
- One alternative: transfer the entire operation log

Simple, but inefficient (operation log is long)
STATE TRANSFER VIA SNAPSHOT

• Every op must be either before or after state transfer
  • If op before transfer, transfer must reflect op
  • If op after transfer, primary forwards the op to the backup after the state transfer finishes

• If each client has only one RPC outstanding at a time, state = map + result of the last RPC from each client
  • (Had to save this anyway for “at most once” RPC)
SUMMARY OF RULES

1. View i’s **primary** must have been primary/backup in view \( i-1 \)

2. A **non-backup** must reject forwarded requests
   - Backup accepts forwarded requests only if they are in its idea of the current view

3. A **non-primary** must reject direct client requests

4. Every operation must be **before or after** state transfer
• First step in our goal of making stateful replicas fault-tolerant

• Allows replicas to provide continuous service despite persistent net and machine failure

• Finds repeated application in practical systems (next)
Outline

1. Primary-backup replication

2. Case study:
   - VMWare fault tolerance
VMWARE VSPHERE FAULT TOLERANCE (VM-FT)

• **Goals:**

1. Replication of the whole virtual machine

2. Completely transparent to applications and clients

3. High availability for any existing software
OVERVIEW

- Two virtual machines *(primary, backup)* on different bare metal

- **Logging channel** runs over network

- Fiber channel-attached *shared disk*
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)
Step 1: Hypervisor at the 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 (RDTSC)
LOG-BASED VM REPLICATION

- **Step 2:** Primary hypervisor sends log entries to backup hypervisor over the logging channel.
- 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)
• When backup takes over, non-determinism will make it execute differently than primary would have done

• This is okay!

• Output requirement: When backup VM takes over, its execution is consistent with outputs the primary VM has already sent
THE PROBLEM OF INCONSISTENCY

Primary

Backup

Input

Event?

Output

Primary fails
FT PROTOCOL

- Primary **logs each output** operation
- **Delays** any output until Backup acknowledges it

Can 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: 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
UC San Diego