Lab 2: Primary–Backup

MSC5703/MCS4993: Intro to Distributed Computing Fall 2025
Duration: Two Weeks   Due Date: [Nov 5, 2025]

1. Introduction & Objectives

In this lab, you will build a simple, fault-tolerant service using the Primary–Backup replication scheme. This is a common and fundamental technique for building systems that can survive the failure of a single server.

Your goal is to implement a fault-tolerant Key-Value (KV) store. Clients should be able to `Put` a value associated with a key and `Get` the value for a key. The service must continue to operate correctly even if one of the servers (either the primary or the backup) crashes.

To keep this lab "simple," we will make a few key assumptions:

- We will not handle network partitions (i.e., you can assume the network is reliable).
- We will not handle a "split-brain" scenario.
- We will use a central `View Service` to act as the single source of truth for which server is the primary and which is the backup. This `View Service` is a single point of failure (SPOF), which is acceptable for this lab.

Learning Goals:

- Understand the design and trade-offs of the Primary–Backup replication model.
- Implement a fault-tolerant service that handles server failures.
- Gain practical experience with RPC (Remote Procedure Call).
- Manage distributed state and handle state transfer.

2. System Architecture

Your system will consist of three components:

The `View Service`: A single, central server that maintains the "view" of the system.

- It decides who is the `Primary` and who is the `Backup`.
- It detects server failures via heartbeating (`Ping`s).
- It manages the failover process by promoting the backup.

The KV Servers (`Primary` & `Backup`): These are the servers that store the actual data (the key-value map).

- Only one server is the `Primary` at any given time.
- Only the `Primary` can accept `Put` / `Get` requests from clients.
- The `Backup`'s job is to be a perfect replica of the `Primary`, ready to take over.
- Both servers will periodically `Ping` the `View Service` to announce they are alive.

The Client: A simple library (that you will also write) that talks to the service.

- The client first asks the `View Service`, "Who is the primary?"
- It then sends all `Get` and `Put` requests to that primary.
- If its requests fail (e.g., connection refused), it asks the `View Service` again for the new primary.

3. Implementation Details

You are free to use any language, but Go, Python, or Java are recommended. You must use an RPC framework for all communication (e.g., Go's `net/rpc`, gRPC, or Python's `xmlrpc`). Do not use simple sockets.

Part A: The `View Service`

The `View Service` is the "brains" of the operation. It doesn't store any key-value data, only the system's current configuration.

It must expose (at least) these RPCs:

- `Ping` **(server_name):**
  - Called by KV servers (e.g., "server1", "server2") every 0.5 seconds.
  - The `View Service` records the time of the last successful ping from `server_name`.
  - It returns the current `View` object (see below).
- `GetView` **():**
  - Called by clients to find the current primary.
  - It returns the current `View` object.

The `View` object should contain:

- `ViewNumber`**:** An integer that increments every time the view changes.
- `Primary`**:** The string name/address of the primary server.
- `Backup`**:** The string name/address of the backup server (can be empty).

Internal Logic: The `View Service` must run a "ticker" function (e.g., once every 0.5 seconds) that:

- Checks the last ping time for all servers.
- If the `Primary` hasn't pinged in (e.g.) 1.5 seconds, it is declared dead.
- If the `Backup` hasn't pinged in 1.5 seconds, it is declared dead.
- Handles promotions:
  - **Primary failure:** If `Primary` is dead and `Backup` is alive, promote the `Backup` to `Primary`. The view now has a `Primary` but no `Backup`.
  - **Backup failure:** If `Backup` is dead, the view now has a `Primary` but no `Backup`.
  - **New server:** If a new, "idle" server starts pinging and there is no `Primary`, promote it to `Primary`.
  - **New server (as backup):** If a new, "idle" server starts pinging, there is a `Primary`, but no `Backup`, assign the new server as the `Backup`.

This logic ensures the system "heals" itself by assigning roles to available servers.

Part B: The KV Server (`Primary` & `Backup`)

This is the core of your lab. You will write a single server program that can act as either a `Primary` or a `Backup`, depending on what the `View Service` tells it.

Server State:

- Its own name/address (e.g., "server1").
- The current `View` it got from the `View Service`.
- The key-value data, stored in a simple hash map (e.g., `map[string]string`).

Main Loop: In a background thread, the KV server must:

- Call `Ping`() on the `View Service` every 0.5 seconds.
- Receive the new `View` from the `Ping`() response.
- Compare the new `View` with its current `View`.
- If the `View` has changed, update its internal state and role.

Role-Based Logic:

If I am the `PRIMARY`:

- Accept `Get`(key) RPCs: Look up the key in the local map and return the value.

- Accept `Put`(key, value) RPCs:
  - Check if there is a `Backup` in the current view.
  - If yes, first forward the `Put` request to the `Backup` (e.g., via an internal `ForwardUpdate`(key, value) RPC).
  - Wait for the `Backup` to acknowledge (return OK) that it has stored the data.
  - Only after the `Backup` acks, update the `Primary`'s own local map.
  - Reply OK to the client.
  - If no `Backup` exists, just update the local map and reply OK to the client.

This is synchronous replication and ensures the `Backup` is never behind the `Primary`.

If I am the `BACKUP`:

- Reject all `Get`(key) and `Put`(key, value) RPCs from clients (e.g., return an "I am not primary" error).
- Accept the internal `ForwardUpdate`(key, value) RPC from the `Primary`.
- On `ForwardUpdate`, update the local map and return OK to the `Primary`.

Part C: State Transfer

There is one critical challenge: What happens when a new `Backup` joins the system?

- The `View Service` will assign it as the `Backup`.
- The `Primary` will see this new `Backup` in the `View` it gets from its `Ping`().
- This new `Backup` is empty! It doesn't have the `Primary`'s data.

You must implement state transfer:

- When the `Primary` sees a new server has become the `Backup` (e.g., `View`.`Backup` was empty, and now it's "server2"), it must transfer its entire state to the new `Backup`.
- The `Primary` should send its entire key-value map to the `Backup` (e.g., via a new internal RPC like `SyncState`(`map[string]string`)).
- The `Backup` receives this map, overwrites its local map, and acks.
- Crucially: While the state transfer is happening, the `Primary` must queue (but not process) any new client `Put` requests to avoid inconsistency. It can process them after the transfer is complete.

4. The Client

Your client library should expose two functions:

- `Get` (key)
- `Put` (key, value)

Internally, the client must:

- Keep track of the last known `Primary`.
- On a `Put` or `Get`, try to send the RPC to that `Primary`.
- If the RPC fails (e.g., server is dead, or it replies "I am not primary"), the client must:
  - Call `GetView` () on the `View Service` to get the new `Primary`.
  - Retry the `Put` / `Get` request with the new `Primary`.
  - Loop until it succeeds.

5. Testing

You are responsible for testing your own code. We will run a similar test harness to grade your lab. A good test script would:

1. Start the `View Service`.
2. Start KV server "S1".
3. Check the view (S1 should be `Primary`).
4. Client: `Put` ("a", "1"), `Get` ("a") (should return "1").
5. Start KV server "S2".
6. Check the view (S1 `Primary`, S2 `Backup`).
7. Client: `Put` ("b", "2").
8. **Test Primary Failure:** Kill S1.
9. Wait 2–3 seconds.
10. Check the view (S2 should be `Primary`, no `Backup`).
11. Client: `Get` ("a") (should return "1"), `Get` ("b") (should return "2").
12. **Test State Transfer:** Start KV server "S3".
13. Check the view (S2 `Primary`, S3 `Backup`).

14. Wait for state transfer to complete.
15. **Test New Backup:** Kill S2.
16. Check the view (S3 `Primary`, no `Backup`).
17. Client: `Get`("a") (should return "1"), `Get`("b") (should return "2").

If your client gets the correct data after all these failures, your implementation is correct!

6. Deliverables

Source Code: A zip file containing your complete, commented source code for the View Service, the KV Server, and the Client library.

Report: A report includes:

- Your name and student ID.
- A description of your design, particularly any challenges you faced (e.g., how you handled state transfer).
- Results and analysis.
- Any known bugs or limitations.
- Instructions on how to compile and run your code.

7. Grading Policy (100 points total)

- **View Service correctness (20 pts)**: Accurate ping tracking; timely detection of dead servers; correct promotion/demotion; monotonic `ViewNumber` updates.
- **KV Primary/Backup logic (30 pts)**: Primary accepts client `Put`/`Get`; Backup rejects client calls; synchronous replication on `Put` with waiting for Backup ACK before commit; correct local updates and client replies.
- **State Transfer (25 pts)**: Full snapshot via `SyncState`; Backup state overwrite; Primary queues client `Put` requests during transfer and drains afterward; no inconsistencies.
- **Client & Report (25 pts)**: Client retries with `GetView` on role errors/failures until success; clear Report with build/run instructions and concise design notes.