18-749 Fall 2017
Lab 3: Passive Replication

TAs: Malhar Chaudhari, Hari Guduru

Wednesday, 1 November 2017

1 Administrivia

This lab is due **Sunday, November 19, 2017, 23:59 hours Eastern Time (GMT -0500)**.

1.1 Submission Instructions

We will provide you with a zip file containing the source-tree and IDE project (IntelliJ IDEA Community Edition) that you will develop your lab in. This file will be named `lab3-codehandout.zip` and can be downloaded via the lab-3 channel on Slack (alongside this lab write-up).

To submit your completed lab, zip up the same directory with your completed source files. Name your zip file `<Andrew ID>_lab_3.zip`, substituting `<Andrew ID>` for your actual Andrew ID. Also, all submissions should be by only one of the teammates name. Finally delete the files after grading is completed.

Once you have created your zip file, please upload the zip file to the `@rds749bot` user on Slack. This is a special user account managed by the course administrators that will retrieve your submission from Slack and store it for grading.

We will grade the last zip file submitted for your Andrew ID via Slack, prior to the deadline. Note also that you **must** receive an acknowledgement from the bot indicating that your submission has been accepted; if you do not receive an acknowledgement, please contact the TAs.

The total number of submissions for this lab is **20**. Any submissions beyond these will incur *10% penalty on every additional submission*. Note that these submissions do not include submissions that did not compile.

2 Lab Architecture

2.1 Overview

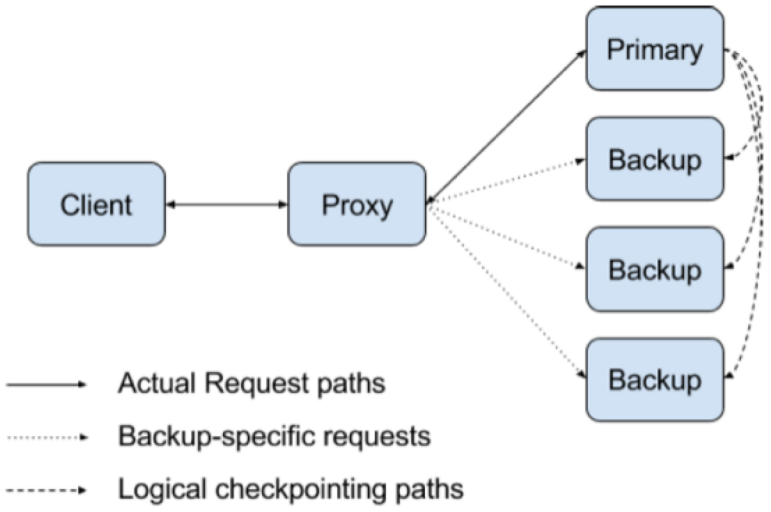

Figure 1: Lab 3 Architecture

The goal of this lab is for you to implement passive replication in our simple distributed system architecture. Like in Lab 2, this means that:

- all servers must have the same state at all points in time
- there will be no state inconsistencies between different servers

Then, in the event of a server being lost/killed, the system should continue functioning with no inconsistencies nor loss of state.

To accomplish this, you will need to implement *passive replication* (rather than active replication as was done in Lab 2). In passive replication, only one server is active, called the primary server. All other servers are backup servers. Only the primary server processes and responds to requests from clients. The backup servers receive requests from clients, but instead of processing them, write them to a local log (which is used later during failover and recovery).

2.2 Logical Time

Lab 3 makes heavy use of the concept of logical time, implemented by the sequence numbers attached to requests. The sequence numbers on requests establish an ordering in time of those requests. These sequence numbers can be used to ensure that requests are processed (and logged) in the correct order, and they are also used to identify the logical point in time at which a checkpoint is taken. The combination of a state value (the value of the bank account) along with the sequence number of the last request processed leading up to that state value is known as a checkpoint.

2.3 Logging and Checkpointing

The local log maintained by the backup servers will continually grow over time as requests are serviced. To prevent unbounded log growth, we introduce a mechanism called checkpointing. In checkpointing, the primary server periodically sends a checkpoint of its state (as defined above) to all the backup servers. The backup servers can use the checkpointed state to trim their log files to only those requests with sequence number greater than the checkpoint. This works, because the backups can reproduce the state of the primary by starting with the checkpoint state and replaying the log from the checkpoints sequence number forward. This is all that is needed for failover and recovery.

2.4 Failover and Recovery

When the primary server fails, the proxy must initiate failover and recovery procedures. Failover is the process of selecting one of the backup servers to be the new primary server. For this lab, you may choose any of the backup servers to become the new primary. Recovery is the process of synchronizing the state of the backup server so that it is identical to the state of the primary prior to failure. To do this, the backup should restore the last checkpointed state, and then replay, in order, all messages stored in its local log from the point of that checkpoint onward (which should be all the messages in the log, if the log is being trimmed appropriately).

Note that during recovery, the backup server may send duplicate messages while replaying its local log. Thus, the proxy needs to apply duplicate suppression to responses received from servers before forwarding them to the client. Duplicate suppression should be implemented by tracking the sequence numbers of responses sent to the client and verifying, prior to sending a new response, that the corresponding sequence number has not been sent to the client previously. See the Implementation Plan section below for additional guidance on implementing duplicate suppression.

Note: In Lab 2, we had required the system to handle a new replica being brought online while the system was actively handling requests. This meant that the system needed to achieve quiescence before transferring state to a new replica that is coming online. However, for Lab 3, you may assume that no additional servers are brought online after the client makes its first request.

3 New/Modified RPCs

Lab 3 introduces a few new RPCs that you will need to use to implement passive replication.

1. `setBackup()`: Called to indicate that a server is to be a backup server. After this function is called, all `begin*Balance` requests should log the request rather than actively handling it.

2. `setPrimary()`: Called to indicate that a server is to be a primary server. If `setPrimary()` is called after a `setBackup()` call, this should trigger the restoration of the latest checkpoint and replay of the log (recovery).

3. `getState()` and `setState()`: Previously used for the setup of a new replica, these functions are now used for transferring checkpoints between the primary and the backups. `setState()` now returns an integer indicating the number of messages in the log, after any log pruning is applied. This is returned to the proxy for grading purposes (to verify that you have implemented logging and log pruning correctly). You do not need to use the return value from `setState()`.

This lab also includes a new class whose definition has not been included in the Javadoc. This class is used for convenience in handling the state of the servers, so do take note:

- **Class:** `Checkpoint`
- **Constructor Parameters:** `int reqid, int state`
- **Fields (Public):** `int reqid, int state`
- **Usage:**
  - `Checkpoint c = new Checkpoint(reqid, state);`
  - `int reqid = c.reqid;`
  - `int state = c.state;`

4 Implementation Plan

Below is a suggested game plan for completing this lab. There is no requirement that you follow this plan – it is provided only as a guide.

1. Review the Lab 2 solution that will be provided along with this lab. This should help you avoid implementation mistakes (such as race conditions) while working on Lab 3.

2. Implement management of primary and backup servers along with simple failover. The first server that registers should be the primary (call `setPrimary()`), and the remainder should be backups (call `setBackup()`). When a primary server fails, one of the backups should be chosen as the new primary (call `setPrimary()` on the backup). If a backup server fails, you should simply remove it from the list of backups.

3. Implement checkpointing and state transfer. The proxy should periodically request checkpoints from the primary at the interval specified in the config file and send this to the backups. The backups should receive new checkpoints from the proxy and update their local copy of the checkpoint. Ensure that the checkpoint is obtained in a consistent state!

4. Implement logging on the backup servers. When a server is in backup mode, calls to `begin*Balance()` should log the request and not call `end*Balance()`.

5. Implement log pruning on the backup servers when checkpoints are received. Backups should remove log entries prior to the logical time contained in the checkpoint when a checkpoint is received. Calls to `setState()` should now return the length of the log following pruning.

6. Extend your simple failover mechanism with recovery . When `setPrimary()` is called on a backup, the backup should pause handling requests, set the bank account state to the checkpointed state, and restore full state by replaying the log of requests and then continue handling requests as the new primary.

7. Implement duplicate suppression in the proxy. If the proxy has already received a certain request from a server (e.g. a call to `end*Balance()`), subsequent duplicate `end*Balance()` calls must be suppressed.

8. Write tests according to the test strategies section below. Make sure your code passes these tests locally.

9. Submit your solution to `749bot`.

5 Tips and Strategies

- Ensure that the server switches modes correctly when `setPrimary()` and `setBackup()` are called. In Primary mode, the server should update the bank account value and call the `end*Balance()` functions. In backup mode, the server should simply log the request instead. Think about how to handle `setPrimary()` or `setBackup()` when its the first time either of those calls are made (e.g. just after server startup) and then how to handle the promotion of a backup into a primary.

- Think about what data you need to store in checkpoints and in logs. Checkpoints need to contain both the state and a sequence number. The logs need to store all information needed to replay the request at a later time, including the sequence number of the request.

- Synchronize the reading of a checkpoint from the primary with other requests being made to the primary. You may need to use locks to handle this.

- When implementing checkpointing, youll need to use a timer or some other way of executing code periodically at fixed intervals.

- A good guide to Java multithreading (threads, mutexes, condition variables, etc.) can be found at
`https://docs.oracle.com/javase/tutorial/essential/concurrency/`

`index.html`
Note that the left side of the page contains links to many topics around synchronization and locking.

6 Testing Hints

- Ensure that basic `begin*Balance()`/`end*Balance()` functionality works with a single (primary) server. The behavior should be identical to the fault-free case in previous labs.
- To test that your checkpointing/logging mechanism works:
  - Print a log message in your server when:
    - Checkpoints are received
    - Messages are logged
    - Messages are pruned from the log
  - Configure a checkpoint interval of 1 second.
  - Spawn three servers and a proxy.
  - Use the client to send requests twice per second.
  - Check your log messages to determine if checkpointing is working correctly.
- To test that your failover/recovery mechanism works:
  - Print a log message in your server when:
    - Checkpoints are received
    - Messages are logged
    - Messages are pruned from the log
    - The primary fails
    - `setPrimary()` is called on a backup
    - Checkpoint restoration and log replay is complete
  - Configure a checkpoint interval of 1 second.
  - Spawn three servers and a proxy.
  - Use the client to send requests twice per second.
  - Kill the primary server.
  - Wait for failover/recovery to complete (you should see log messages telling you when this happens)
  - Verify the following:
    - Failover to a new primary occurred
    - The state was restored correctly (the sequence of requests and responses from the system is correct/consistent)
    - No errors were visible to the client during the failover/recovery process
    - Duplicate messages are not received while the proxy recovers

  - Repeat this test several times with different checkpoint intervals and request rates (change the values in b and d above) to test the robustness of your code. This should help to find race conditions and other hard-to-identify bugs.

- To help with testing, we are leaving in the print statements in the library if you use to log actions taken. You will see them when you run your code.

Note: We strongly recommend writing Java Unit Tests for checking the above suggested test cases. These will help you cover corner cases.