18-749 Fall 2016
Lab 2: Active Replication

Utsav Drolia (@utsavdrolia), Nathan Mickulicz (@nmickuli),
Jiaqi Tan (@tanjiaqi)

Wednesday, 5 October 2016

1 Administrivia

This lab is due **Wednesday, 26 October, 2016, 23:59 hours Eastern Time (GMT -0400)**.

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 `lab2-codehandout.zip` and can be downloaded via the #labs 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_2.zip`, substituting `<Andrew ID>` for your actual Andrew ID.

For written answers, type up your answers in a plain text document, and name your file `<Andrew ID>`_lab_2.txt. This text file should be included in the root of your zip file.

Once you have created your zip file, please upload the zip file to the @749bot 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 ahs been accepted; if you do not receive an acknowledgement, please contact the TAs.**

2 Lab Architecture

2.1 Overview

The goal of this lab is for you to implement Active Replication in our simple distributed system architecture. This means that: (i) all Servers must have the same state at all points in time, and (ii) there will be no state inconsistencies between different Servers (unlike in Lab 1). Then, in the event of a Server being lost/killed, the system should continue functioning with no inconsistencies nor loss of state (unlike in Lab 1).

In this lab, we have made some minor changes to the ways in which the two basic methods provided by our `BankAccount` object, `changeBalance()` and `readBalance()`, are provided, so as to accommodate Active Replication. In short, the `changeBalance()` and

`readBalance()` methods are now implemented asynchronously, i.e., calls to the `changeBalance()` and `readBalance()` functions do not return immediately to the caller. Instead, the implementor of the `changeBalance()` and `readBalance()` methods (i.e., the `Proxy` and `BankAccount` classes) must now explicitly complete the servicing of the method by calling a "reply", or `end` method of the client to return a result.

The main tasks for you in this lab are:

1. To familiarize yourself with the new way in which the `BankAccount` and `Proxy` classes need to support the `changeBalance()` and `readBalance()` methods for callers,
2. To update the Server Registration functionality to support Active Replication,
3. To provide Active Replication amongst all the Servers connected to your Proxy, and
4. To handle Servers that register with the Proxy, but that has state that is different than the Servers already connected to the Proxy.

2.2 `changeBalance` and `readBalance`: Asynchronous Calls

2.2.1 Previously: Synchronous Remote Calls

As you learned in Labs 0 and 1, in the distributed system architecture of our labs, `readBalance()` and `changeBalance()` are remote methods that are invoked by a client (i.e., the Client and Proxy respectively), and serviced by a remote program (i.e., the Proxy and Server/BankAccount respectively). In Labs 0 and 1, to simplify the labs, these remote methods were called **synchronously**: this means that the methods behaved like regular methods in a local program in the following ways: (i) after being called, the remote program (i.e., the Proxy-end of a method servicing the Client-called method) could simply return its result to the caller by using the Java `return` statement, and (ii) the caller immediately received the result of the call from the remote program where it called the remote method.

For instance, consider the following code from Lab 1 in the Proxy:

```
BankAccountStub s;
public long register(String hostname, String port) {
    s = this.connectToServer(hostname, port);
}
public int readBalance() throws NoServersAvailable {
    return s.readBalance();
}
```

Notice that in the implementation of `readBalance()` in this code: (i) the Proxy is able to respond to the remote caller of its `readBalance()` method by returning a result directly, and (ii) when it calls the Server (in invoking `readBalance()` on object `s`), the Server immediately returns its result.

While this is convenient for programming, it does not allow us to perform Active Replication. Instead, the implementation of the `changeBalance()` and `readBalance()` remote methods will be asynchronous, as we will explain next.

2.2.2 Asynchronous Remote Calls

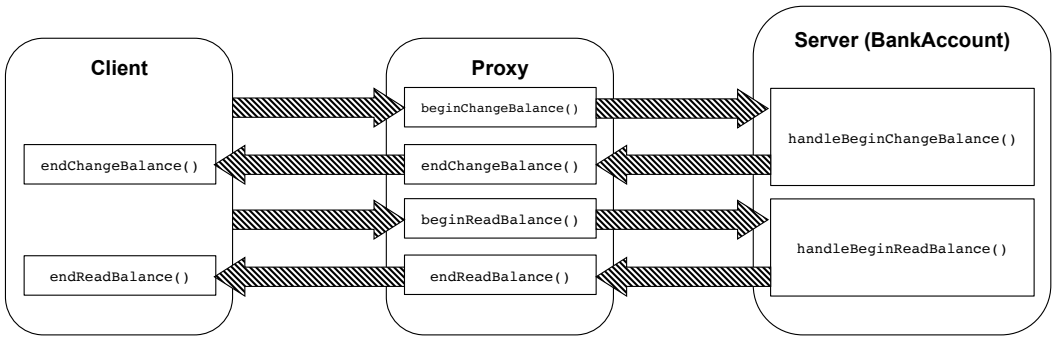

Figure 1: New `BankAccount` application architecture, with asynchronous remote calls.

In Lab 2, the `changeBalance()` and `readBalance()` remote methods will be asynchronously called. This implies the following:

1. Callers will not receive a response immediately (e.g., for a Client-to-Proxy connection, the Client, and for a Proxy-to-Server connection, the Proxy, will not receive a response immediately),

2. Callers must provide a way for Callees to return a response (e.g., for a Client-to-Proxy connection, the Proxy must have a way to return a response, and for a Proxy-to-Server connection, the Server must have a way to return a response), and

3. Callees must respond by calling the response method of the Callers (e.g., for a Client-to-Proxy connection, the Proxy must call the response method of the Client, and for a Proxy-to-Server connection, the Server must call the response method of the Proxy).

This is illustrated in Figure 1. Take for instance, the `changeBalance()` operation. The following steps occur for what we called the `changeBalance()` operation to occur:

1. The Client requests a change in balance by calling the `beginChangeBalance()` method (which gets sent to the Proxy).

2. The Proxy receives the Client's request to change the balance through its implementation of the `beginChangeBalance()` method, and it requests the Server to change the balance by calling the Server's `beginChangeBalance()` method (which in turn calls the `handleBeginChangeBalance()` method on the Server).

3. To reply to the Proxy, the Server's `handleBeginChangeBalance()` method calls the Proxy's `endChangeBalance()` method to send the Proxy the result.

4. On receiving the Server's response to the Proxy's `beginChangeBalance()` call (in Proxy's own `endChangeBalance()`), the Proxy in turn calls the Client's `endChangeBalance()` method to send the Client the result.

A similar process occurs for the `readBalance()` operation.
Note the following:

- The `begin*()` (and `handleBegin*()`) methods are implemented by providers of remote methods (i.e., the Proxy and Server, but not the Client).
- The `end*()` (and `handleEnd*()`) methods are implemented by the callers of remote methods (i.e., the Client and Proxy, but not the Server).

2.2.3 Unique Request Identifiers (UUID)

Next, consider what happens in the above workflow, when the Client sends a request (e.g., to change the balance) to the Proxy, which then forwards the request to the Server. The Server calls the Proxy's `end*()` method, which in turn calls the Client's `end*()` method. This sends a response to the Client. However, how does the Client know which request the response belongs to?

To simplify this lab, you can assume that the Client will supply a Universally Unique Identifier, more commonly known as an UUID, together with each `changeBalance()` and `readBalance()` request. This UUID effectively tags the request, and the Server can return the same UUID to inform the Client (at the end of the whole operation) which request the response being sent corresponds to. The UUID is indicated using the `reqid` parameter that is passed as the first argument of `beginChangeBalance()` and `beginReadBalance()`, and as the second argument of `endChangeBalance()` and `endReadBalance()`.

In practical systems, clients are typically not cooperative, and such UUIDs can be assigned by the Proxy as a Proxy would be the first point of entry of the request into the system. However, you can assume that UUIDs will be supplied to you for this lab.

2.2.4 Concurrent Processing of Requests in the Proxy

In Labs 0 and 1, you were allowed to assume that the Proxy processes only one request at a time, and that there are no concurrent requests being processed by the Proxy. This meant that there was only a single thread in the Proxy that called your `readBalance()` and `changeBalance()` methods.

In Lab 2, you may no longer make the above assumption. **There will be multiple threads** that receive requests from the Client, and that call the `beginReadBalance()` and `beginChangeBalance()` methods (previously `readBalance()` and `changeBalance()`) in your Proxy.

2.2.5 Error Handling

Note that, as in the previous lab, if you receive the `NoConnectionException` exception when calling the Server's `beginChangeBalance()` or `beginReadBalance()`, you need to handle the exception appropriately in the Proxy.

However, unlike Lab 1, you cannot throw a `NoServersAvailable` exception. Instead you will have to call `RequestUnsuccessfulException(int request_id)`. This function is implemented by the Client and is available on the Proxy to call.

Also, in the Server, you may assume, when calling `endChangeBalance()` and `endReadBalance()`, that the Proxy will never fail. Also, in the Proxy, you may assume, when calling `endChangeBalance()` and `endReadBalance()`, that the Client never fails.

2.3 Server Registration

In Lab 2, like in Lab 1, there will be a Server Registration process by which new Servers register themselves with the Proxy to provide services. However, note that each Server that registers with the Proxy begins with clean state (i.e., $balance = 0$), and if there are other existing Servers registered with the Proxy that contain state, then your Proxy must do something about this new Server with clean state, before this new Server can be handed requests, and considered to be one of the actively replicated Servers.

In Lab 1, you performed a fail-over to switch the Active Server to a different Server. In Lab 2, due to our use of Active Replication, there is no longer a notion of an "Active Server", nor the notion of failover.

Note that there may be any number of Servers connected to/registering with the Proxy.

2.4 Active Replication

Next, we describe the actual processes of Active Replication that you need to implement in your lab.

2.4.1 Replicating Requests

Incoming requests that are received by the Proxy (at `beginChangeBalance()` and `beginReadBalance()`) must be sent to all Servers that have registered successfully with the Proxy (that have also received the required state).

Recall that each request is accompanied by an UUID that is provided to you by the Client, as stored in the *reqid* argument supplied to each function.

Hint: *Note that each Server must see requests (i.e., calls to `handleBeginChangeBalance()` and `handleBeginReadBalance()`) in the same order as the Proxy received the requests from the Client.*

Hint: *What does the `synchronized` keyword mean in a method signature? You may want to refer to: `https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`.*

2.4.2 Duplicate Suppression

Next, all Servers that have received `beginChangeBalance()` and `beginReadBalance()` calls must respond to the call by calling the corresponding `endChangeBalance()` and `endReadBalance()` methods of its caller (i.e., the Proxy).

Then, if there are N Servers connected to the Proxy, the Proxy will receive N responses–one from each Server. However, the Client should receive only one reply in response to its call to `beginChangeBalance()` (or `beginReadBalance()`). Hence, the Proxy must perform Duplicate Suppression to hide repeated messages after the operation has been completed.

2.5 Server Failure

In an actively replicated distributed system, server (or replica) failures are counter-intuitive: servers are actively replicated anyway, so if there are N servers, where $N > 1$, and 1 server

becomes unreachable, then there are still $N - 1$ actively running Servers. So where does a Server Failure manifest itself?

As mentioned in Section 2.3, a Server Failure manifests during the Server Registration stage, when a new Server wishes to register with the Proxy to begin serving requests.

When there are existing Servers that contain state and that have been serving requests, the new Server is no longer consistent with the other Servers (that are actively replicated). In this case, actions must be taken to restore the consistency of the state of the new Server as compared to the other running Servers.

To do so, the Proxy needs to obtain state from one of the existing Servers that is alive, and transfer this state to the new Server.

2.5.1 Quiescence of System

Before retrieving the state from an existing Server, there must be no outstanding or in-flight requests (i.e., changing or reading the balance). We call this period a quiescent period of time, during which no state is being changed on all of the Servers.

As noted in Section 2.3, you need to do something about new Servers that register with the Proxy when there are existing Servers already connected to the Proxy. When performing any state manipulations, the entire system (i.e., all Servers) must be in a quiescent state before the state manipulations (e.g., state transfers) can be performed.

2.5.2 State Transfer

Next, the state needs to be retrieved from some Active Server, and appropriately sent to the new Server that is attempting to join the system.

To help with this, we have provided `getState()` and `setState()` methods that have to be implemented by the Server and can be called by the Proxy.

3 Goals (10 points)

1. **[1 pt]** Implement `beginChangeBalance()` and `beginReadBalance()` on the Proxy.

2. **[1 pt]** Implement `endChangeBalance()` and `endReadBalance()` on the Proxy.

3. **[1 pt]** Implement `beginChangeBalance()` and `beginReadBalance()` on the Server.

4. **[1 pt]** Implement `getState()` and `setState()` on the Server to enable the Proxy to retrieve the current state of a Server, and to push new state to a Server.

5. **[1 pt]** Implement `register()` on Proxy to bring a newly registered Server up-to-date with the other replica Servers that are already registered with the Proxy.

6. **[2 pt]** Implement Active Replication of incoming requests in `begin*Balance()` methods on the Proxy to replicate the request to all connected Servers. *Hint: You may want to implement a `synchronized sendToAllServers()` method.*

7. **[2 pt]** Implement Duplicate Suppression of responses in the `end*Balance()` methods on the Proxy to ensure that the Client receives only one result in response to each issued request.

8. **[1 pt]** What happens when there is a Server failure? How does your system's behavior in response to a Server failure compare to the fail-over in Lab 1? *Provide your written answer in a text file in your solution zip.*