18-749 Fall 2017
Lab 1: Registration, Heartbeats, and Fail-over

Originally written by : Utsav Drolia, Nathan Mickulicz and Jiaqi Tan.
Modified by : Rohit Agrawal

Friday, 29 September 2017

1 Administrivia

This lab is due **Monday, 09 October, 2017, 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 `lab1-codehandout.zip` and can be downloaded via the #lab-1 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_1.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_1.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 @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: You will have 10 penalty free submissions per group. Beyond this, there is a 10% penalty for each submission

2 Lab Architecture

2.1 Overview

The goal of this lab is for you to build some basic components required for fault-tolerance. We will build on the basic Proxy architecture that you explored in Lab 0 for this lab. Recall the architecture in Figure 1. In this architecture, the 3 components, namely Client, Proxy and Server, are 3 distinct processes capable of communicating only over remote procedure calls. Just as in Lab 0, the communication details have been abstracted for you so that you can concentrate on the fault-tolerance aspects of the labs. You will be expanding on this basic architecture to implement new features to help you achieve fault-tolerance for our `BankAccount` distributed application.

In Lab 0, you completed the provided implementation of the Proxy, so that the Proxy selects a Server to forward the requests it receives to the Server that is connected to the Proxy. Figure 1 illustrates this operation.

Building on Lab 0, you will be implementing three features in this Lab 1:

1. You will develop a registration mechanism for each Server to register with the Proxy to inform the Proxy that it is available to service any requests. Note there may be several Servers registered simultaneously (i.e. more than two).

2. You will develop a heartbeat mechanism for each Server component to inform the Proxy that it is alive and ready to accept requests.

3. You will develop a fail-over mechanism to: (i) detect when a Server has failed; (ii) if the Active Server has failed, change the Server to which the Proxy is forwarding requests; (iii) ensure that failures are only visible to the client when all servers have failed.

As you will see from this exercise in Lab 1, the Proxy plays a crucial role in providing the fault-tolerance functionality of our simple distributed system. The Proxy keeps track of the Servers (which are replicas) in our system, and decides which Server to forward requests to, based on the Servers that are available.

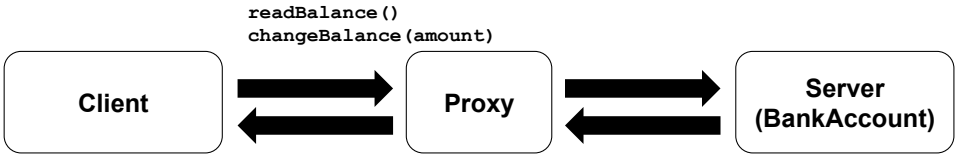

Figure 1: Recap: Basic `BankAccount` application architecture, with a proxy to provide fault-tolerance features.

The Proxy needs to select **one** Server to forward requests to, and we will call this the Active Server. Any server may be the Active Server. If the Proxy detects that the Active Server is unreachable, then the Proxy needs to select another Server to function as the Active Server. If no Servers are available (i.e. there is no Active Server and no other Servers can be promoted to the Active Server), the Proxy should respond to Client requests by throwing the `NoServersAvailable` exception, which will then be sent back to the Client and handled appropriately. Note that the client MUST NOT receive an exception in any other case.

2.2 Server Registration

Recall that each Server in our distributed system is an `BankAccount` object. In Lab 0, we manually configured the Proxy to tell it the hostnames and port numbers of where each Server could be contacted. In Lab 1, we require each Server to register with the Proxy, so that the Proxy can forward requests to it. Figure 2 illustrates this registration mechanism. Figure 2 shows the Server calling the `register` method, which is a remote procedure provided by the Proxy to allow Servers to register with it. This will be available to you on the Server to call when needed. **Your task is to implement the `register(serverHostname, serverPort)` method on the Proxy**, to handle registration requests from Servers. This method should record the hostname and port number for the server so that it can be used for future communication with

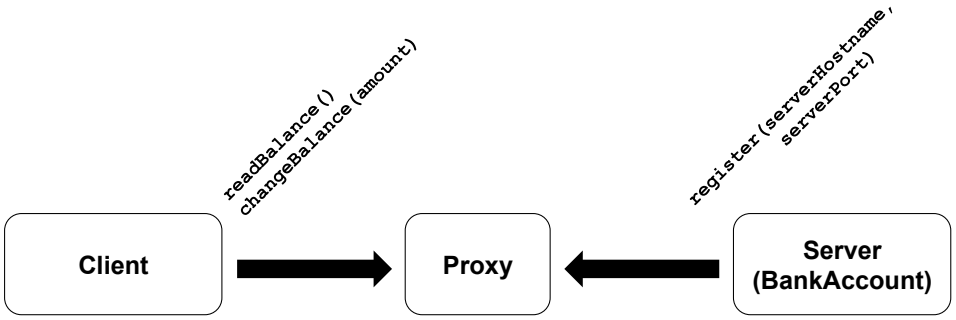

Figure 2: Registration of Server with Proxy.

that server. The method should also return a unique 64-bit (long) identifier, which the Server will use when making heartbeat calls to the Proxy.

Remember, the point of the registration procedure is to ensure that the Proxy knows where it can forward Client requests to. The Server should register with the proxy on startup (note that we assume that the Proxy never fails, so the server does not need to worry about re-registering). We have provided the necessary communications mechanisms in the `AbstractServer.ProxyControl` class in the Server (you are not responsible for the underlying communications), such that when the Server calls the `ProxyControl.register(serverHostname, serverPort)` method, the `ProxyControl` object remotely calls the `register(serverHostname, serverPort)` method of the proxy class.

We have also provided convenience classes to help you retrieve configuration options from a configuration file specified as a command-line argument. The `Configuration` object in the `BankAccountI` class supplied to you contains the parsed configuration options. The hostname and port of the Proxy are specified in the Configuration file as the `proxyHost` and `proxyPort` options, and the hostname and port of the Server, where it will listen for requests, as the `serverHost` and `serverPort` options. *(Be sure to specify the correct filename for your configuration file, and that you specify the correct hostname and port for the Proxy server you wish to connect to.)* An example of reading a configuration value is provided in the `Proxy.java` starter code.

2.3 Server Heartbeat

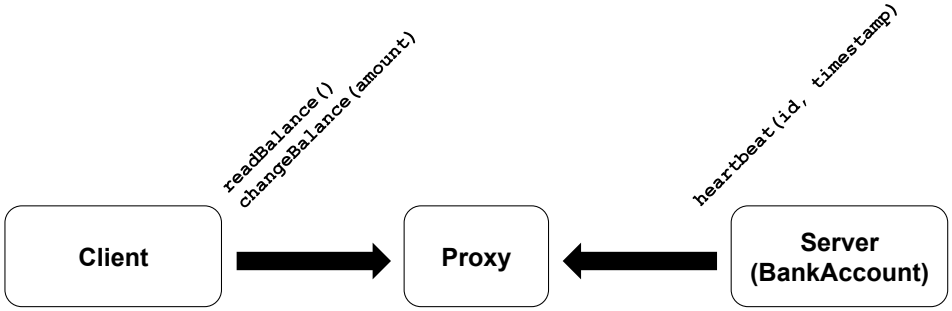

Figure 3: Heartbeat from Server to Proxy.

In our simple distributed system, each Server contacts the Proxy periodically to inform the Proxy that it is still alive. We call this mechanism a "heartbeat", where the Server sends a "heartbeat" message to the Proxy periodically at some pre-defined interval. Figure 3 illustrates this heartbeat mechanism. You must use the heartbeat interval specified in the configuration file as the option `heartbeatIntervalMillis`. This configuration file value can be obtained using the same `Configuration` object that you use to obtain the `proxyHost` and `proxyPort` options above.

The heartbeat mechanism is provided as a method of the Proxy class, that is remotely called from the Server. We have provided the necessary communications mechanisms in the `AbstractServer.ProxyControl` class. You must implement the two ends of the heartbeat mechanism: (i) calling the heartbeat from the Server, where you need to periodically call `ProxyControl.heartbeat()` method, and (ii) handling the received heartbeat at the Proxy, by implementing the `Proxy.heartbeat()` method.

First, you must call the `AbstractServer.ProxyControl.heartbeat()` method on the Server, for the Server to send a heartbeat to the Proxy. Second, when you call the `AbstractServer.ProxyControl.heartbeat()` method, the corresponding `Proxy.heartbeat()` method is remotely invoked for you. `Proxy.heartbeat()` should invoke your logic for tracking and updating the state of the server sending the heartbeat (identified by the ID parameter). Note that the `Proxy.heartbeat()` method also has a timestamp parameter, which is the current Unix timestamp on the Server at the time that the heartbeat is sent. Because packets may be delayed or received out of order, this timestamp provides a way to ensure that you are receiving recent heartbeat messages from the server. If you receive a heartbeat message with a timestamp prior to the timestamp of the last heartbeat received from the same server, you should ignore that heartbeat.

Second, you must implement logic in the Server to periodically call the `ProxyControl.heartbeat()` method. The intervening remote call from the Server to the Proxy has already been implemented for you. You should make sure your implementation of the Server calls `ProxyControl.heartbeat()` periodically, with the delay between calls specified by `heartbeatIntervalMillis`, in milliseconds.

2.4 Fail-over

The final part of Lab 1 involves determining when a Server has failed and changing the Active Server if the failed server is the Active Server. In Lab 0, the Proxy forwarded requests to one Server. In Lab 1, one or more Servers may connect to the Proxy, and it is the responsibility of the Proxy to select one Server to forward requests to and return the response from the Server to the Client. We call this selected Server the Active Server.

In this lab, the Proxy may be connected to one or more Servers, as the registration mechanism that you implemented will allow additional Servers to register with the Proxy dynamically. In this lab, the Proxy must change the Active Server designation to a different Server that is reachable if the selected Active Server fails. Figure 4 illustrates the case where the first Server is the Active Server, but it becomes unavailable, so the Proxy needs to change its Active Server to the second Server.

Server failures should be detected using both heartbeat information and any errors or exceptions raised when making a remote call to a Server. For heartbeats, the Proxy should declare a Server failed if it has not received a heartbeat from that server within **2 proxy heartbeat**

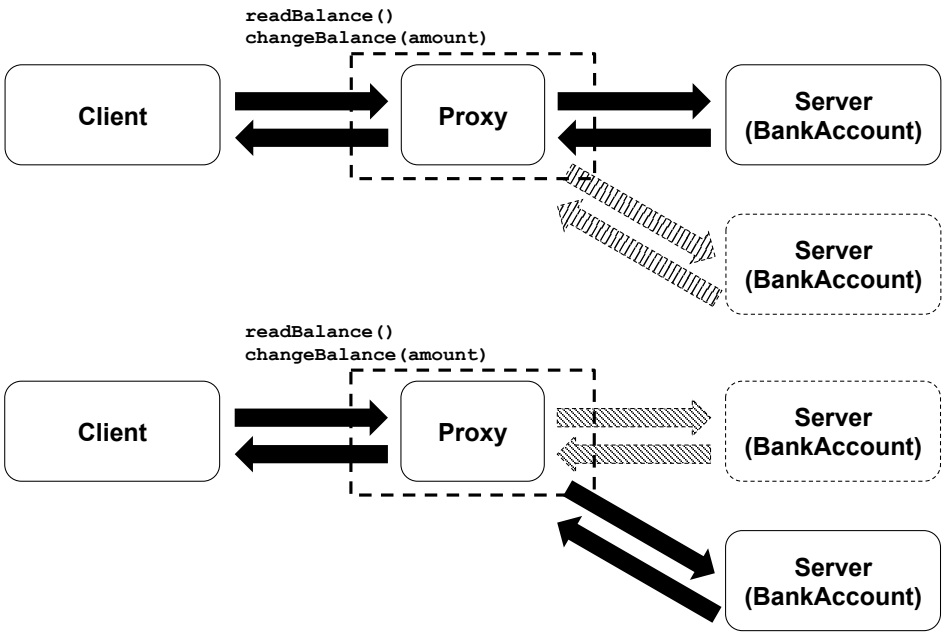

Figure 4: Fail-over at Proxy when a Server fails.

intervals of the last heartbeat. Similarly, if an exception is thrown while making a remote call to a Server, that Server should be declared failed. If the Active Server is declared failed, you should invoke logic to select another Server as the Active Server. Once a server is selected as the Active Server, another Server may only be chosen as the Active Server after the current Active Server has failed.

The Proxy must handle the case where a Server fails and then later recovers, which is indicated by receiving heartbeats from that Server after it has previously failed. Note that Server recovery should not trigger a fail-over action, and the current Active Server should remain the Active Server. However, once a server has recovered, it may be considered for selection as the Active Server should the Active Server fail, just like any other Server.

Hint: You are now responsible for the registration in the Proxy, as well as the usual `readBalance()` *and* `changeBalance()` *methods in the Proxy. The* `register()` *method tells you which Servers there are, the* `heartbeat()` *method tells you which Servers are alive, and the* `readBalance()` *and* `changeBalance()` *methods perform the actual forwarding of requests. Think about how to represent/store the available Servers in the Proxy so that you can easily identify which Servers are registered, and which ones are alive. You should detect Server failures by using both heartbeat information as well as any errors or exceptions thrown when making a remote call to a Server).*

Hint: The `BankAccountStub` *(previously named* `BankAccountPrx` *in Lab 0) class in the Proxy, represents the remotely-callable version of the Server. Similar to Lab 0, calling the* `BankAccountStub.readBalance()` *and* `BankAccountStub.changeBalance()` *methods will result in the method call being relayed to the remote Server, and its results relayed back to you.*

2.5 Run Configurations

Given that the Servers are now registering with the Proxy, the **Proxy should start-up first**. Then the Servers and Clients can come up in any order. There are 4 run configurations provided to you to test in the IDE. RunProxy starts up the Proxy, RunClient starts the Client, RunServer1 starts one Server, and RunServer2 starts a second server. There are 2 example configuration files provided - `config-server.xml` and `config-server2.xml` that are used in the run configurations. Following are some helpful Client CLI commands.

- `!run-script filename` reads and executes commands from given file.
- `!set-display-time true/false` toggles displaying of command execution time. Time is shown in milliseconds.
- `!enable-logging filename` and `!disable-logging` control logging, i.e. duplication of all the input and output in a file.

3 Goals

1. **[0.5 points]** Call the `AbstractServer.ProxyControl.register()` method on the Server to register with the Proxy.
2. **[0.5 points]** Implement the `register()` method on the Proxy to handle received registrations.
3. **[0.5 points]** Periodically call the `AbstractServer.ProxyControl.heartbeat()` method on the Server to send a heartbeat to the Proxy.
4. **[0.5 points]** Implement the `heartbeat()` method on the Proxy to handle received heartbeats.
5. **[2 points]** Provide fail-over on the Proxy to ensure requests from the Client are forwarded to the Active Server as far as possible.
6. **[1 point]** Run the Client, start 2 Servers, issue a series of at least 5 requests to change the balance in the BankAccount, kill the Active Server, then issue a series of another 5 requests. Provide us with: (i) your requests, indicating when the server was killed, and (ii) the output from the Server (as seen from the client). *Note: After the failover, the client should see the "wrong" output, since the 2 BankAccount Server replicas are not consistent and are operating independently. Later labs will address this inconsistency and you are not responsible for consistency.*