18-749 Fall 2017
Lab 0: Warmup

Initially developed by : Utsav Drolia, Nathan Mickulicz and Jiaqi Tan
Modified by : Rohit Agrawal

Monday, 18 September 2017

1 Administrivia

This lab is due **Friday, 22 September, 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, see §1.2) that you will develop your lab in. This file will be named `lab0-codehandout.zip` and can be downloaded via the #labs channel on Slack (alongside this lab writeup).

To submit your completed lab, zip up the same directory with your completed source files. Name your zip file `<Andrew ID>_lab_0.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_0.txt. This text file should be included in the root of your zip file.

You may use any zip utility that you wish to create your zip file. Within the provided VM, we suggest using the command-line tool `zip` to create your zip file. An example command that you can use to create this file is "`zip -r <andrew ID>_lab_0 *`" from within your `lab0-codehandout` directory. This will produce a file named `<andrew ID>_lab_0.zip` which is ready for submission.

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.**

1.2 Developing in Java

To develop your code for the labs, you can either download the VirtualBox VM image which we provide, with the recommended IDE installed, or you can download and install the IDE on your own computer.

We recommend you use the IntelliJ IDEA Community Edition integrated development environment, which you can download from `https://www.jetbrains.com/idea/download/`.

We will provide you with a zip file containing an IntelliJ IDEA project that you can open and immediately begin development in. This zip file contains starter source files as well as libraries packaged as JAR files that contain additional support code needed to run the lab. This zip file can be downloaded from the class #labs channel on Slack.

1.3 Using the VM image

If you do not have VirtualBox installed, please download it from `https://www.virtualbox.org/wiki/Downloads` and install it. Download the image from `http://goo.gl/ry76mV` to a location of your choice. Once downloaded, double-clicking on the file should open up VirtualBox and show you the VM being imported. Click on "Import". Once the VM is imported, do a sanity check of the settings. Right click on the imported VM, select "Settings", select the "Network" icon on the settings dialog, expand "Advanced" under "Adapter 1". Finally, ensure that the "Cable Connected" checkbox is checked. You should now be ready to start the VM[1].

1.4 Opening and Working With Your Project

For your convenience, we have set up an IntelliJ IDEA project in the zip file that we have provided you. To work with your lab, start IntelliJ IDEA, then open the project by either: (i) selecting "Open" from the welcome screen, and selecting the directory where you unzipped your code, or (ii) going to the "File" menu, selecting "Open", and selecting the directory where you unzipped your code.

Then, to work on the code, go to the "`lab0-codehandout`" folder in the project pane in the left hand side of the window, and expand the `src` directory, which contains stub files that you need to modify to complete your lab.

To run each of the components (see below), go to the "Run" menu, and select "Run..." to select which of the components you wish to run. Go to "Run" > "Edit Configurations..." if you wish to change the command-line options to provide to each of the components in the lab that you wish to run.

2 Overview of Labs

In 18-749, we have designed a series of labs to walk you through the process of developing a reliable distributed application. We begin from a basic distributed application organized in a client/server model, which we provide you with. At the end of this series of labs, you will have implemented a number of different mechanisms to provide fault-tolerance and consistency for the distributed application.

We will have a total of 4 labs, Labs 0 through 3, throughout the semester. Each lab will build on the concepts that you implement in the previous labs. However, you are not required to reuse your code from previous labs: we will provide you with the appropriate starter code for each lab.

You will be required to program in Java for these labs. Java is popular with real-world distributed systems, and is a robust and mature language with a large feature set. Syntactically, some aspects of Java are similar to other programming languages such as C, which we believe

[1] username:student, password:749IsGreat

will lower the learning curve for most students. If you are not familiar with Java, we encourage you to review the large amount of available tutorials on the Internet, such as `https://docs.oracle.com/javase/tutorial/`.

3 Lab Architecture

3.1 Distributed Application: **`BankAccount`**

3.1.1 Organization of Application

The distributed application in this series of labs is the `BankAccount` application. A `BankAccount` application models a single bank account, which stores a single integer that represents the dollar balance of the bank account. This application models a bank account, and provides the following two functions:

1. **`readBalance():`** Returns the current balance of the account.
2. **`changeBalance(amount)`**: Changes the balance of the account by the specified amount; positive numbers represent a deposit, and negative numbers represent a withdrawal. Returns the balance of the account after the deposit/withdrawal has been made to the account.

The `BankAccount` server listens on an IP port for `readBalance()` and `changeBalance()` requests from its clients, and returns the result of each request.

Figure 1 shows the basic architecture of the `BankAccount` application, depicting the client and the `BankAccount` server, without fault-tolerance. If a `BankAccount` server goes down (e.g., crashes), the client will not be able to perform its desired requests.

For the purposes of this lab, you do not need to be concerned with how the Client object calls the `readBalance()` and `changeBalance()` functions on the Server object, as we have provided underlying implementations for these communication mechanisms. The remote calls from the Client to the Server object is not in the scope of this series of labs.

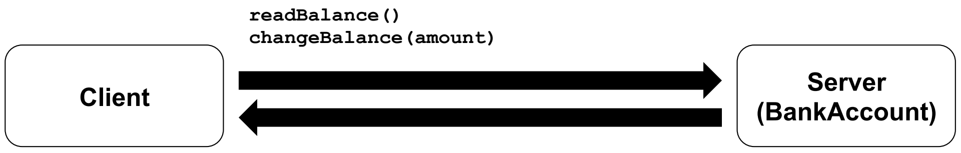

Figure 1: Basic `BankAccount` application architecture, without fault-tolerance.

3.1.2 Running the Server

The Server object, which implements the `BankAccount` application, provides two functions:

1. `int readBalance()`
2. `int changeBalance(int amount)`

The Server object is implemented in the `edu.cmu.rds749.lab0.BankAccountI` object, and this object is remotely callable by the Client.

The Server object can be executed by running the `edu.cmu.rds749.common.ServerMain` object in the provided starter code, which takes 1 command-line argument: a port number to listen for requests on.

The Client object provides a command-line interface for issuing requests to a specified Server object. The Client object can be executed by running the `edu.cmu.rds749.common.ClientMain` object provided in the starter code, which takes 2 command-line arguments: (i) the hostname/IP address of the Server, and (ii) the port of the Server.

The CLI interface for the Client can run the following commands.

- `check` returns the current balance
- `update <amount>` updates the balance with the given `<amount>` (`<amount>` can be negative to mimic withdrawals.)
- `!run-script filename` reads and executes commands from given file
- `exit` exits the CLI

To run the full `BankAccount` application, first start the Server (through `ServerMain`), and then start the Client (through `ClientMain`).

3.2 Warmup: Proxy

3.2.1 Overview of Proxy

The main goal of fault-tolerance for our `BankAccount` application is to allow the "Server" object in Figure 1 to fail. To protect against failure of the `BankAccount` object, we need to introduce an additional middle software layer between the client and the server. Then, in the event of a `BankAccount` server failure, this middle layer can redirect requests to another server.

We will call this middle software layer a "proxy", as it will act as an intermediary between the client and the server. Figure 2 shows where this middle proxy layer will reside in our software architecture.

In this lab, you will focus on implementing the proxy as a warmup exercise. In future labs, you will work on designing and implementing functionality at the proxy and other layers to provide fault-tolerance capabilities for our `BankAccount` application.

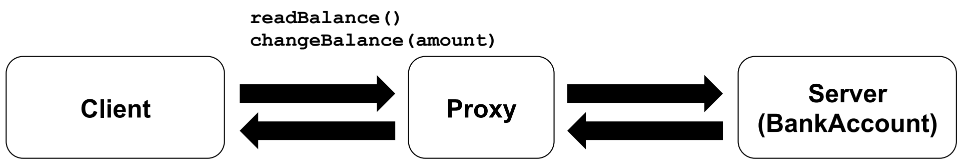

Figure 2: `BankAccount` application architecture, with a proxy to provide fault-tolerance features. You will implement the proxy in this lab as a warmup exercise, and you will implement fault-tolerance features in the proxy (amongst other components in the architecture) to provide fault-tolerance.

Hint: Look under `docs/index.html` in the code directory for documentation of the starter code to help you with the lab, particularly for the `AbstractProxy` class.

3.2.2 Running the Proxy

We have provided the `edu.cmu.rds749.common.ProxyMain` class, which provides the command-line setup code for setting up the Proxy. Currently, the `ProxyMain` class takes 3 arguments, in the following order: (i) the port number to listen on for requests, (ii) the hostname of the server to connect to, and (iii) the port number of the server to connect to.

We have also provided the `edu.cmu.rds749.lab0.Proxy` class, which you will modify as part of this lab. This Proxy object implements `edu.cmu.rds749.common.AbstractProxy` which has the same interface as the Server object (i.e., it is indistinguishable from the `BankAccountI` class to the Client). The `AbstractProxy` also provides a `actualBankAccounts` field, which is an array of connections to Server objects that the Proxy object has.

4 Goals of Lab 0

The goals of Lab 0 for you are to:

1. **[1 point]** Run the original `BankAccount` application by: (i) starting an instance of Server, and (ii) starting an instance of Client. Issue a series of requests from the Client to convince yourself that the Server is returning the correct results. **In your written answer, provide us with: (i) a list of the instructions that you issued on the command-line of the client, and (ii) the output of the server.**

   Structure your written answer as follows:

   ```
   Q1:
   =====INPUT=====
   .....
   =====SERVER OUTPUT=====
   .....
   =====END OUTPUT=====
   ```

2. **[3 points]** Complete the implementation of the `edu.cmu.rds749.lab0.Proxy` class, so that requests from the Client to the Proxy will be served transparently.

3. **[1 point]** Run the new `BankAccount` application with your proxy by: (i) starting an instance of Server, (ii) starting an instance of Proxy, and (iii) starting an instance of Client. Issue a series of requests from the Client to convince yourself that your Proxy is functioning correctly. **In your written answer, provide us with: (i) a list of the instructions that you issued on the command-line of the client, and (ii) the output of the server.**

   Structure your written answer as follows:

   ```
   Q3:
   =====INPUT=====
   ```

```
.....
=====PROXY OUTPUT=====
.....
=====SERVER OUTPUT=====
.....
=====END OUTPUT=====
```

Write your answers for (1) and (3) and include them in your submission zip in the appropriately-named text file.