CSE 486/586: Distributed Systems

Programming Assignment 2, Part B[1]
Group Messenger 2

Introduction

This assignment is an extension of Programming Assignment 2, Part A. You will extend your group messenger to add ordering guarantees, implementing a total ordering across messenger instances that preserves FIFO ordering. Messages will be stored in your content provider as implemented in Part A. As you know by now, *you must follow these instructions exactly.* Failure to do so may result in no credit for this assignment.

Please read this document in its entirety before you begin. It is long, but that is only because it is complicated and precise. Revisit the instructions regularly as you implement to be sure you're implementing what is required. There are notes at the end that may assist you if you encounter common problems; make sure to review them as you go, as well.

This assignment will require you to:

- Implement a FIFO total ordering algorithm.
- Detect and handle failure of a single messenger instance.

1 Getting Started

We are providing a project template for you to get started. As in the previous assignments, the template configures a number of things on your behalf. It is important that you do not change settings that you do not have to change to complete your project.

2 The Group Messenger App

The graded portion of this project is an implementation of the group messenger app with ordering guarantees and message storage in a content provider with the same interface as Part A (but a different URI; see below). You will need to download the project from GitHub Classroom and open it in Android Studio. You should have received the GitHub Classroom invitation for this project via Piazza.

Instructions for cloning the project and opening it in Android Studio are the same as before, except that the base repository name is `GroupMessenger2`. You can find YouTube tutorials detailing the process here.

For this portion of the project, you will need to run five emulators. We will be using the multi-port configuration as described in project 1. Your app will open one server socket that

[1]This assignment is based, with permission, on an assignment developed and used by Steve Ko in his version of CSE 486/586. Large portions of text are used verbatim from that assignment.

listens on port 10000, but it will connect to a different port number on the IP address 10.0.2.2 for each emulator, as follows:

emulator serial	port
emulator-5554	11108
emulator-5556	11112
emulator-5558	11116
emulator-5560	11120
emulator-5562	11124

2.1 Total Ordering

Implementing the ordering functionality is the bulk of this assignment, and to get full credit you will have to successfully provide ordered messages in the face of the failure of a single app instance during execution. Your app should provide a total ordering for incoming messages *among all app instances*, and that total ordering should preserve FIFO ordering (hereafter, total-FIFO ordering). Total and FIFO ordering are as defined in class. The results of this ordering will be stored in your content provider, as described below.

2.1.1 Requirements

1. The URI of your content provider must be:
   `content://edu.buffalo.cse.cse486586.groupmessenger2.provider`
   Note that this is different from Project 2, Part A!

   The other requirements for the content provider are as in Project 2, Part A.

2. Each app should multicast every user-entered message to all app instances, *including the one that is sending the message*. In the rest of this description, "multicast" always means sending a message to *all app instances*. It must be able to do this when all five emulators are running simultaneously.

3. Your app should use B-multicast. It should not implement R-multicast.

4. You will need to find or design and implement an algorithm that provides total-FIFO ordering *under a single process failure*.

5. There will be *at most one failure of an app instance during execution*. This failure will be emulated by force closing an application instance. We will emulate a *crash-stop* failure: the failed app will not return during the execution run.

   (a) Every message should be used in detection of node failure.

   (b) You should use a timeout on socket read to detect failure. Choose a reasonable timeout (*e.g.*, 500 ms), and consider a node failed if it does not respond within the timeout.

   (c) This will require handling socket timeout exceptions explicitly (and differentiating them from other socket exceptions, such as at creation, connection, or end of file).

(d) *Do not rely only on socket creation or connection status to determine if a node has failed.* It is not a safe or sufficient solution! Please also use read timeout exceptions.

(e) You cannot assume which app instance will fail, and different app instances will be killed randomly on different testing runs. Therefore your algorithm should not rely on, *e.g.*, a single centralized sequencer, as it will then be unable to handle the failure of that sequencer.

(f) You should implement a decentralized algorithm (*e.g.*, something based on ISIS) for ordering, so that your messenger can order messages in the face of app failure.

(g) It is important to make sure that your app does not stall following failure. You should clean up any state related to the detected failure, then move on.

6. When there is a node failure, the grader will not check the ordering of messages *sent by the failed node*. See Testing, below, for more details.

7. Every message should be stored in the content provider on the local device individually by each app instance. Each message should be stored as a key-value pair, with the key being the final delivery sequence number for the message (as a Java string) and the value being the actual message (as a Java string). The delivery sequence number for this purpose should start with 0 and increase by 1 for each message. *Note that this may not, and need not, be the same as the priorities agreed upon by your total ordering algorithm.*

2.2 Testing

We have testing programs to help you see how your code does with our grading criteria. If you find rough edges in the testing programs, please report it so the teaching staff can fix it. The instructions for using the testing programs are the following:

- Download a testing program for your platform. If your platform does not run it, please report it.
    1. Windows: Tested on 64-bit Windows 8.
    2. Linux: Tested on 64-bit Debian 9 and 64-bit Ubuntu 17.10 (see below for important information about 64-bit systems).
    3. Mac OS: Tested on 64-bit Mac OS 10.9 Mavericks.
- Before you run the program, please make sure that you are running all five AVDs. You can use `python run_avd.py 5` to start them.
- Remember to start the emulator network by running `set_redir.py 10000`.
- Unlike previous testing programs, this test will require you to provide your APK as a command line argument, and will take care of installing and uninstalling it on the emulators.

- Run the testing program from the command line. The testing program takes a number of command-line arguments to control its execution, which you can view with the `-h` argument. You may find them helpful for debugging.
- It may issue some warnings or errors during execution. Some of them are normal, some may indicate errors in your program. Examine them to find out!
- The testing program will give you partial and final scores.

The testing will proceed in two phases, where the first phase tests your application in the absence of failures, and the second tests it with a single app failure. As described above, you must implement a decentralized failure detection and handling algorithm that deterministically identifies failures. If you use a centralized or nondeterministic (*e.g.*, randomized) failure detection algorithm, or do not uniformly handle failures by cleaning up and moving on, the score you get for the second part is not guaranteed.

The ordering of messages sent by the failed app, in the event of app failure, will not be considered by the grader. In particular, it is possible that some app instances will have successfully delivered messages from the failed app, while others will not, and that is OK. The grader will examine the total-FIFO ordering guarantees only for the messages sent by live nodes. Note that, in phase 2 of the testing, this means that the grader may indicate that a particular key is missing; this is the message sequence number that the grader is verifying. It may not be the exact key due to failure.

The grader is multi-threaded, and error and diagnostic messages from the various threads are uncoordinated. This means that an error message may appear early in the execution, or in with other log messages, rather than cleanly at the end of execution. You will have to examine the output to determine where and when errors may have occurred.

Remember during your own testing that you may see strange contents in your content provider if you do not uninstall your application between tests. In particular, updating the app from Android Studio does not uninstall the existing app first, so the content provider's state will not be cleared. You can uninstall your app by hand with the following command:

```
adb -s <emulator> uninstall edu.buffalo.cse.cse486586.groupmessenger2
```

Notes for 64-bit Linux: The testing program is compiled 32-bit. If you get an error like the following, or the shell reports command not found when you run the executable, install the 32-bit `libz` for your system:

```
./simplemessenger-grading.linux: error while loading shared libraries:
    libz.so.1: cannot open shared object file: No such file or directory
```

On Debian-based distributions, you can accomplish this with the command `apt-get install zlib1g:i386` as root (you may need to use `sudo` or `su`). If `apt-get` reports an error about the architecture or says the package is not found, you may need to enable multiarch. To do this, run `dpkg --add-architecture i386` as root, then update your APT repositories with `apt-get update` as root. Once this is done, you should be able to install the 32-bit `libz`.

For other distributions you will need to consult your distribution documentation.

3 Submission

We use UB CSE autograder for submission. You can find autograder at https://autograder.cse.buffalo.edu/, and log in with your UBITName and password.

Once again, *it is critical that you follow everything below exactly.* Failure to do so **will lead to no credit for this assignment**.

Zip up your entire Android Studio project source tree in a single zip file. Ensure that *all* of the following are true:

1. You *did not* create your zip file from *inside* the `GroupMessenger1` directory.
2. The top-level directory in your zip file is `GroupMessenger1` or `GroupMessenger1-<something>`, and it contains `build.gradle` and all of your sources.
3. You used a zip utility and *not any other compression or archive tool:* this means no 7-Zip, no RAR, no tar, *etc.*

4 Deadline

This project is due 2018-03-09 11:59:00 AM. This is one hour before our class. This is a firm deadline. If the timestamp on your submission is 11:59:01, it is a late submission. You are expected to attend class on this day!

5 Grading

This assignment is 10% of your final grade. Credit for this assignment will be apportioned as follows:

- 4%: Your group messenger provides total-FIFO ordering guarantees with messages stored in the content provider.
- 6%: Your group messenger provides total-FIFO ordering guarantees with message stored in the content provider for all correct app instances under a single app failure.

Thus, an application that provides correct total-FIFO ordering both with and without a single app failure will receive a total of 10%: 4% for correct operation without any failures, and 6% for correct operation in the face of a single app failure.

Notes

- Please do not use a separate timer to handle failures, as this will make debugging very difficult and is likely to introduce race conditions. Use socket timeouts and handle all possible exceptions that may be thrown when there is a failure. They are: `SocketTimeoutException`, `StreamCorruptedException`, `EOFException`, and the parent `IOException`.

- Please reuse your TCP connections with a single remote host, instead of creating a new socket every time you send a message. You can use the same socket for both sending to and receiving from a remote AVD. Think about how you will coordinate this.
- Please do not use Java object serialization (*i.e.*, implementing `Serializable`). This will cause very large messages to be sent and received, with unnecessarily large message overhead.
- Please do not assume that a fixed number of messages will be sent in your system. Your implementation should be able to handle an arbitrary number of messages, and you should not hardcode such a value in any way.
- Remember that there is a cap on the number of AsyncTasks that you can execute simultaneously. Plan accordingly, to keep the total number of threads that you require under this limit (which is about five).