15-446: Distributed System
Project 1: Distributed File Updates with Eventual Consistency

TA: Dongsu Han

February 16, 2009

Assigned: January 22, 2009
Checkpoint 1 due: February 5, 2009
Design document due: February 19, 2009
Final version due: February 26, 2009

1 Introduction

The purpose of this project is to give you hands-on experience in developing applications for mobile and distributed systems. We will be using the Android platform. You will develop a file sharing application where each mobile user keeps a copy of a file and the updates get synchronized whenever users come across each other within wireless communication range. The application works as follows. The application advertises itself by sending periodic broadcast messages over the air, and listens for broadcast messages sent by others to detect their presence (Service discovery). When the application detects other device running a copy of the application within range, it establishes a TCP connection. The connection is then used to exchange meta data including version vectors and synchronize the files for eventual consistency.

The project will be divided into two parts. The first part is to implement the service discovery and establish a single TCP connection between every pair (Checkpoint 1). The second part is to implement a fully functioning system.

Part of the goal for checkpoint 1 is to get familiar with developing on Android. You will need to learn how to develop applications on Android. So start early!

2 Logistics

- The files for this project can be found here:
  `http://www.cs.cmu.edu/~srini/15-446/S09/project1/CS446Project1.`

`tar.gz`

- This is an individual class project. You must implement and submit your own code.
- Checkpoint 1 is due on Feb. 5.
- Design document is due on Feb. 19.
- Final is due on Feb. 26.

3 Project Details

A template application and a set of API will be provided for the project. The template application runs on Android and talks with a local server to emulate connectivity changes. An API similar to Java Socket API is also provided in order to make it easy to communicate between multiple emulators.

3.1 Developing Environment

Final version will be graded on linux.gp machines you may develop on any machine, but you should make sure it runs on linux.gp
To facilitate the testing, we installed eclipse in the class AFS directory: /afs/cs.cmu.edu/project/cmcl-srini-4/15-446/android.

- Eclipse.
  Applications for Android are written in Java. So we encourage you to use Eclipse IDE.

  A copy of Eclipse is installed in the following AFS directory : /afs/cs.cmu.edu/project/cmcl-srini-4/15-446/android. This will be the HOME directory for android related installations.

  To run execute ./eclipse/eclipse in the HOME dir on a linux machine. Refer to README for troubleshooting.

  Create a directory on your own AFS or disk space and use it for workspace. It prompts for a workspace when it starts up.

  You can copy the whole directory on a (local/personal) Linux machine, if you want.

  If you want to install everything yourself, follow the instructions on `http://code.google.com/android/intro/installing.html`. You will need to install Eclipse 3.4, Android SDK, and Android plugin. Also, Java SDK 1.5 or 1.6 is required by Android and Eclipse 3.3 or 3.4.

  If you want to use Windows or OS/X, make sure you can install Ruby and eventmachine library. Ruby, Ecplise and Android has Windows and OS/X versions, but I personally haven not tried.

- Android SDK/Emulator
  We will need to use the Android emulator instead of real devices. The emulator either can run stand-alone or can be run from a eclipse plug-in. The eclipse installed on AFS has the Android plug-in installed.

  Android emulator is installed on HOME/android-sdk-linux_x86-1.0_r2/tools. Include this directory in your PATH environment variable to run adb. 'adb shell' lets you connect to a shell on a running emulator. Specify the name of the android emulator ('adb -s {name} shell'), if you are running multiple androids.

  Android emulator binds to TCP port 5555 to 5585 when it starts up. So make sure you run the emulator on a machine that is not being shared at the time. Otherwise, you may run into problems.

- Android Debugger
  You can debug Android applications using Eclipse IDE very similar to how you would debug normal Java application.

  Also Android provides logging infrastructure. 'adb shell logcat' will show you the log. You can also see the log by executing an application in debug mode and switching to "Debug" perspective in Eclipse.

  Read the following documents for details.
  `http://code.google.com/android/intro/develop-and-debug.html`
  `http://code.google.com/android/reference/adb.html`

3.2 Project Components.

- "CS 446 File Sharer" App
  This is the template application that you will need to fill in. The application extends Activity class of Android SDK and brings up "start" and "stop" buttons that starts and stops the file sharing service. The file sharing service extends the Service class of Android SDK, and spawns threads to connects to the local server, handle broadcast and run a FileSynchronizerThread which is what you have to fill in. It connects to a specified port (10001) to talk to local server. The TA will let you know how to change the port if you have problems using the port.

- Local Sever
  The local server controls the connectivity between multiple emulators. To change the connectivity at different times, we introduce "time slot". Each time slot represents a logical time in which connectivity is sustained. Connectivity does not change within a time slot. Connectivity is always mutual.

  To run the time server run './serv.rb'. The file is included in the tarball. The time advances on request basis. The File Sharer App can advance time to simulate changes in connectivity. You should call the function, advanceTime(), to advance time on every device in order to

advance the logical time. Check out the template code as it already implemented. Once you call the function all the connectivity will be lost until all the devices call it. So make sure you call it only after everything's been done at each time slot.

Local server uses port 10001 to 10001+ (number of emulators). The TA will let you know how to change the port if you have problems binding to the port.

You will need eventmachine library to run this. The library is installed in the AFS, and Ruby was installed by default on linux.gp machines so I am assuming all CS machines will have Ruby. To use the eventmachine library, you have to setup the right environment variable. Run 'source HOME/rubyenv' to set the environment variable. If you have copied the directory, change the path accordingly by editing the rubyenv file. As a test you can run, './echo.rb' provided in the HOME directory if it does not complain then you are good to go.

- Configuration file
  The configuration file that defines connectivity is written in XML. The number of emulators and number of time slots are important parameters that this file specifies. It defines the connectivity at each time slot. A few different configurations will be provided. You may edit it to test different configurations.

- Socket API
  This will be covered in class, and the documentation will be available through lecture notes.

- Broadcast API
  This will be covered in class, and the documentation will be available through lecture notes.

4 Getting Started

- **Start early!** It will take time to learn how to program on Androids. Although we provide installed programs through AFS, you may have to install some programs to start the project on you own machine. Take account for the time to set up the developing environment. Although Android itself is pretty stable, the emulator may have some bugs and problems. So expect hurdles and start early.

- **Start simple.** Start with a simple, working code. For example, start with two emulators for basic service discovery and build it up for test cases with multiple emulators.

- **Possible Bugs in the emulator.** There might be some bugs in the emulator. I've seen a case where a read to a socket does not return even if the other end has closed a socket. If you experience such problems embed a length on your protocol so that you don't wait for data that won't be sent. If you can't get around like this, then just make it so that the bug does not block the main code which should return.

- **Think ahead about the second project.** Your next project will be self defined. Think about what you want to do. There are many interesting application that are developed for Android. Check them out on Android homepage for ideas.

4.1 Running the Android Application

Download the tar file and untar the file in your local directory. Run eclipse and open the project by clicking New → Project → Android → Android Project. Click 'Next'. Select "Create project from existing source" and select the CS446Projec1/FileSharerProject directory from the untarred directory.

Then you will see the FileSharerActivity in the Package Explorer. You will need to adjust the path for a jar file. Right click on FileSharerActivity and click Properties. Go to Libraries tab and select FileSharingApp.jar, click 'Edit' button. Inside the CS446Project1 directory there will be the FileSharingApp.jar. Select it and click OK. Now you can run the application on Andriod.

Click on the project and go to Run → Run. Select Android Application. Android Emulator will come up and will start your activity.

4.2 Running multiple Androids

XML configuration file tells you the total number of androids (nodes attribute). With that you can specifiy or figure out how many emulators you should run. You have to bring up the specified number of emulators to make it running. The FileSynchronizer thread does not start until all of the applications on the emulators are started.
On starting multiple emulators on Eclipse, refer to the lecture note.

4.2.1 About the final version

Officiall, we only will support linux environment. You will need to make sure you can run your code on the linux.andrew machines for grading. You may choose to develop on other plaforms at your own risk. We will provide minimal trouble shooting support for those users. The ruby server can be run remotely for Windows users, but you need to use linux and run the ruby server locally to generate the final ouput (SD card image).

Getting Started

- The tarball for the final contains a new jar file, a new ruby server, , a SD-card image, and a set of new config files.
  Replace the old jar file with the new one (FileSharingApp_project1.jar). Fix the link to the jar file on Eclipse (Section 4.1).

- If the Eclipse indicates errors on java files, replace FileSharerActivity.java, FileShareService.java, and strings.xml with the new version. (People who have the latest patch for the jar may not have to do this.)

- FileSynchronizerThread.java have three (function) updates that you will need to copy to your own file from check point 1. The three updates are updateLocalFile(), FileSynchronizerThread::run(), and outputFiles().

- **Eclipse setup**
  Under tab Target in Run Configurations, mark Wipe User Data.

Under Additional Emulator Command Line Options, put -sdcard <path_to_the_untarred>/1.img. This will mount the sdcard image in /sdcard. Only the first emulator will be able to mount this, and all others will output warnings. Ignore this. The SD card image will be used to permanantly store the output files you generate. You need to run the ruby server locally to do this. The SD card image is to be turned in with the code.

- **Running the ruby server:** ./serv_final.rb <xml config> <base port>
  If you don't provide any argument, it will use config.xml and 10001 by default. The base port is the lowest port number that the server uses to communicate with Android emulators. The ruby server now does not output debug messages. To enable just give anything as a third arugment. Not recommended unless you have a problem you want to report for debugging.

Note: the old ruby server does not work with the new jar. Use the new ruby server.

Inputs and outputs

- We now have actual file updates from users. You will receive the update from updateLocalFile() at the beginning of each time step. Files have globally unique IDs and operation on the file is always append.

- You need to output all files after the synchronization at the end of each time step. Each Android, writes the content of the file in a file named <NodeID>-<fileID>-<time_slot>.

  To output the content in a file, we use the openFileOutput() Android API. Given a filename, this function opens the file in a application private path and returns a FileOutputStream of the file. The name cannot contain a path.
  Context.openFileOutput (`http://code.google.com/android/reference/android/content/Context.html#openFileOutput(java.lang.String,int)`)

  An example usage is given in FileSynchronizerThread.java. The file you open goes to /data/data/edu.cmu.cs446.ta/files in the emulator. To access the files use adb shell.

  At the very end (end of all time slots), the ruby server copies all the files in /data/data/edu.cmu.cs446.ta/files in all emulators and copies it to the /sdcard directory of the emulator that has the SD card mounted. It also generates a copy of the files in the directory that the ruby server is running. result-<timestamp> will contain all the outputs in your local directory.

What happens in each time step

- First, you get the input which is a set of updates from a user.
- Then you need to do synchronization (including service dicovery and connection setup).

- Finally, you will need to output the final content. Assume you're presenting it to the user, but the output will be used for grading as well.

What you need to implement

- You will need to implement a version vector system to keep track of distributed file updates.
- To maintain eventual consistency, you need to synchronize the content of files after you establish TCP connections and output the synchronized content at the end of each time step.
- You will need to implement all the internal structure. Input is given by the API and but you need to output files according to the format given.

Running the ruby server remotely. (Recommened for Windows developers.)
You can run the ruby server remotely. You need to give the name of the server by editing strings.xml. The xml file has place holders for local_server_name and local_server_port. Update them accordingly.

5 Hand-in and Grading

5.1 Hand-in

Individual hand-in directory will be set up in : `/afs/cs.cmu.edu/project/cmcl-srini-4/15-446/handin/S09_project1_final`

5.2 Testing

Make sure you can run it on linux.andrew machines. The program will be tested on linux.andrew machines for grading. It is your responsibility to make sure it runs on the machine.

5.2.1 Requirements for checkpoint 1

- Implement service discovery. Use broadcast to advertise yourself. If you receive broadcast messages from other devices, establish TCP connections.
- At the end of each time slot, make sure there is only one connection per a pair of connected nodes. You should be able to detect whether there are more than once connections and eliminate redundant connections.
- Implement the above in FilesynchronizerThread. Read the comments and follow the

5.2.2 Deliverables for the final version

You need to run the test with four different configuration files given. Run them locally in order to get the ouput files copied to the SD card.
Turn in your code, final SD card image containing all output files from four test cases, and a design document.

You can develop on other platforms and use your own machine to generate the SD card image with output files, but we only provide support for andrew linux clusters. Make sure you can reproduce them in andrew.linux machines because it will be the grading platform.

Refer to Section 5.3 and Section 4.2.1 for the requirments and other details.

5.3 Grading Scheme

If you cannot implement the full functionality up to the requirements, you may implement what you can based on this grading scheme. Prioritize the heavy ones.

- **Checkpoint:** 20 points
  10 points for setting up the connection and service discovery.
  10 points for setting up a single connection at the end of each time slot.

- **Design** 20 points
  Submit a brief design document. This should illustrate how you implemented the synchronization mechanism. Submit a design document (design.txt or design.pdf) in your hand-in directory.

  Briefly state the algorithm you will use to maintain eventual consistency.
  You also have to consider a chained topology with three nodes, (O — O — O), where a node at one end has a update and the other two (including the one that is connected through another node).

  We expect a consice document on the role of each modules, and how they perform the synchronization task together. Also state how would you detect newer version of files and version conflicts, and how you would update files (determining which node will send what to which node).

  The document should help you think about the design before you actually write a code.
  This document will also serve as a grading point on your understanding of version vectors and the eventual consistency model.

  This part will be the HW2.

- **Synchronization correctness:** 50 points
  20 points for working under simple topology with 3 nodes.
  10 points for working under simple topology with 4 or more nodes.
  10 points for working under chained topology (maximum chain of 3 nodes).
  10 points for correctly detecting collision.

  The grading will be largely based on the SD card image file you submit. So make sure you run all four test cases (xml config files).

- **Robustness:** 10 points
  Packets may get dropped or delayed. For example, broadcast packets may get dropped or delayed. Connection establishment may also get delayed or fail after a long delay. (Currently timeouts on CS446 socket is not implemented, but assume that it can generate IOException for timeouts.)

  Your program should work correctly under a resonable conditions of wireless connectivity. We will grade based on your design and implementation. Include a robustness section in the design document. This part is not part of HW2.

  You should make sure you don't call blocking operations inside threads that should not block. Also when establishing the TCP connection consider a case where initial connection attempt times out and you have to connect from the other node. This design is more robust than just failing or retrying after a long timeout.

- **Extra credit:** 20 points
  Implement synchronization method that uses less bandwidth. When you synchronize the files, send only the incremental update instead of the whole file.

6 Resources

- Lecture notes - Android Overview, Time, Consistency, Replication
- Android Development Portal `http://code.google.com/android/`
- Android Documentation `http://code.google.com/android/documentation.html`
  Android documentation has almost everything from "Hello World" to sample code and java doc. Documentation on Android emulator and debugging methods will be very useful.
- Android Emulator Documentation. `http://code.google.com/android/reference/emulator.html`

- Android Discussion Groups `http://code.google.com/android/groups.html` Q&A if you have problems.
- Ruby download `http://www.ruby-lang.org/en/downloads/`
- Ruby eventmachine library `http://rubyeventmachine.com/`