Computer Science Courses

SafeWalk System, Part 2

Due Fri Dec 05 23:59:59 EST 2014

Please read the entire page before you start coding.

Overview

In Part 1, you implemented a basic SafeWalk server program, and used the basic UNIX tool `telnet` along with your knowledge of the SafeWalk client-server protocol to test it. But a general SafeWalk user does not know the details of the protocol, and thus needs a convenient client which

- transforms a form to a request message,
- sends the message to the server, and
- parses the server response in human-understandable language.

In this project, you will design a socket client program running as a GUI on Kindle Fires (Android API), and this client conforms to an updated version of SafeWalk C/S protocol (discussed later).

Pre-requisites

Pre-requisites are:

- Knowledge of Java Socket API in Android [http://developer.android.com/reference/java/net/Socket.html]
- Knowledge of Android Callback handling mechanism [http://developer.android.com/reference/android/os/Handler.Callback.html]
- Knowledge of designing GUIs for Android apps [http://developer.android.com/guide/topics/ui/index.html]

In Lab 9 (Android Programming), you were guided to set up an Android project and design UI for a SafeWalk client. Don't hesitate to go back to the Lab 9 instructions.

Objectives

- Learn to use Android message handling mechanism
- Learn to use AsyncTask worker thread object
- Learn to design a socket client program

Protocol Updated

In this app we are using a protocol that has been extended from the one used in Part 1. You are given a jar file which contains the updated server program to test your client. Click on the link to download SafeWalkServer.jar and a sample command line Client.java file, save them on your disk and unzip them.

On command-line navigate to the folder where you saved the jar file and execute it using the following command:

```
java -jar SafeWalkServer.jar
```

The server returns the `port-number` on which it is listening. Connect your laptop/PC to the network and determine the `ip-address` using `ifconfig` command for MAC/LINUX OS and `ipconfig` command for Windows OS. This `ip-address` and `port-number` will be used in your app to connect to the server you started using the above command. (Note: Your kindle needs to be connected to the same network as your laptop/PC for the kindle-app to communicate with the server)

The Client can be executed by typing the following command:

```
java Client <ip> <port> <command>
```

Here is a summary of the new protocol to communicate with the server.

More User Types

In the updated protocol, the `TYPE` token is not always `0` anymore. Instead, we define two more values to denote two specific types of users:

Value of `TYPE`	Meaning
`0`	A user that has no preference of which types of users to match.
`1`	The user is a requester and wants to match with volunteers only.
`2`	The user is a volunteer and cannot match with other volunteers.

By *requester* we mean someone who is asking for help from SafeWalk, and by *volunteer* we mean someone who is offering help to SafeWalk.

Server Responses

From the client's perspective, sending a request message (note that *request message* is different from *command message*) will receive either a “Single Pairing” `RESPONSE:` if match is found or an `ERROR: connection reset` message.

Single Pairing responses

This type of message starts with string `RESPONSE:` and has format `RESPONSE: NAME,FROM,TO,TYPE.` where `NAME,FROM,TO,TYPE` is the information of the `paired` user.

In this case the client should tell the user to meet with `NAME` at `FROM` and go to `TO` together. Once your Client receives a match, it has to send back the command `:ACK` to the server in order to acknowledge. This command is used by the

server to ensure that the client is active and has received the message.

Error message

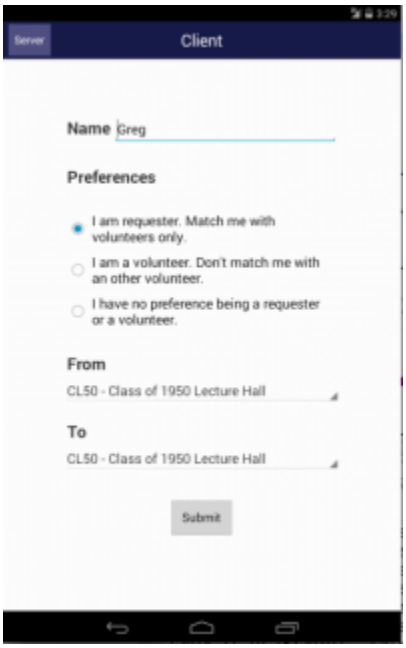
In this case the client should tell the user that the server has been reset.

Specifications

Review of the UI

The reference design of front-end UI has three screens: Server screen, Client screen, and the Matching result screen. Each screen is a Fragment [http://developer.android.com/guide/components/fragments.html] and the basic layout and navigation between fragments is provided to you in the skeleton code. As an example the Server Screen UI is already implemented for your reference. Design your UI for the remaining screens in this project according to the specifications given below by adding the specified objects to your layout. The interactions go as described below.

When the user enters the app, the Client screen is displayed:



There are four fields from top to bottom, corresponding to the four tokens in a request message: `NAME`, `TYPE`, `FROM`, and `TO`. Note that `NAME` is a EditText [http://developer.android.com/reference/android/widget/EditText.html] object, `TYPE` is a set of RadioButton [http://developer.android.com/guide/topics/ui/controls/radiobutton.html], and `FROM` and `TO` are dropdown lists(Spinner [http://developer.android.com/guide/topics/ui/controls/spinner.html]). This is an example implementation. However the developers are free to design their own UI for client interaction.

If the user hits the “Submit” button at the bottom, the app should validate the form, jump to the Matching Result pane, and send request messages to the server.

If the user wants to change the server to connect with, he or she should click the “Server” button on the top left corner and go to the Server screen. The Server screen should define default host and port which are used when the user does not specify his or her own server. This has been already implemented and you can modify the default

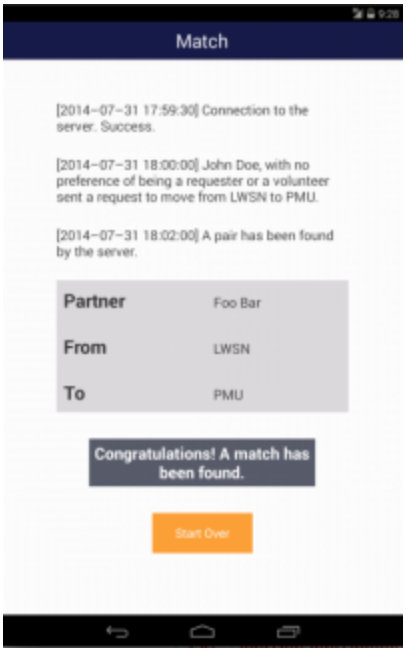
values as per your requirement. This helps for testing different servers. Note that in these fields you will be using the `ip-address` and the `port-number` you obtained earlier. The default IP address in the skeleton (10.0.2.2) is the IP of your computer from the emulator.

Server Screen

There are two "EditText" fields in the Server screen: host and port.

When the user hits "Client" button on the top right corner, he or she will jump back to the Client screen.

In the Matching Result screen, a brief log is shown to indicate the progress, and the details of the paired user will show up when receiving the result. (The format of the log is up to the app designer.) If the server responds with a pairing response, display the detailed table of the paired user.TextView [http://developer.android.com/reference/android/widget/TextView.html] can be used for this purpose.



When the user clicks "Start Over" button, the app should exit gracefully, restore the Matching Result screen to its default state, and then navigate back to the Client screen. The navigation has been implemented. You need to implement the other features. Also note that there is no need to restore the default values of the Server screen.

Thus there are two forms to parse: one for the server host and port, and one for the client information.

Form Parsing

When the user clicks the "Submit" button, the app should first check the validity of the forms before assembling and sending requests.

For the server form,

- the host should be a non-empty string and must have no space character;
- the port should be an integer value of range [1, 65535].

For the client form,

- the user `NAME` should not contain any comma (“`,`”) and should be non-empty.
- the user `TYPE` should be a number of `0`, `1`, or `2`.
- the user `FROM` location should be a defined location and should not be `*`.
- the user `TO` location should be a defined location and should be different from `FROM`.

And as a special-case note, let's specify that if the user uses `*`, his or her `TYPE` must be `2`; otherwise the form is invalid.

When the form is invalid, pop up a notification message (AlertDialog [http://developer.android.com/reference/android/app/AlertDialog.html]) telling the user what is wrong and take the user back to the Client screen when the user accepts the notification message.

Creating a Socket

When the forms are valid, jump to matching result screen, create a socket and try to connect to the given host and port (an example of using client sockets in Android may be found here [http://examples.javacodegeeks.com/android/core/socket-core/android-socket-example/]).

If the server cannot be reached, log an error alert telling the user that `the server is not available`.

If the socket gets established successfully without exception, send the request message in a worker thread (see next) to the server and wait for response. Remember to log about the request sent to the server.

Your socket should work on a separate background worker thread, using an AsyncTask, not on the UI thread; otherwise your UI may refuse to respond when the socket performs some blocking operations. A tutorial on how to use worker threads and thread handlers in Android is available here [http://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html] and the API of the AsyncTask here [http://developer.android.com/reference/android/os/AsyncTask.html].

Sending Messages

Assemble the client form to a request message, and send it to the server via the socket.

For example, if the user gives

- `NAME` is `John Doe`
- `TYPE` is `2`
- `FROM` is `LWSN`
- `TO` is `PMU`

then `println(“John Doe,LWSN,PMU,2”)` on the socket write-end.

Response Handling

After your app sends the request message (and incurs no `Exceptions`), wait and read the response from the server using BufferedReader [http://developer.android.com/reference/java/io/BufferedReader.html]. When the app gets such pairing response, send `:ACK` message to the server and then close the socket and free the resources.

When it gets the pairing response, update your matching result screen to display the information in a human-understandable format.

In your Matching Result screen, there is a “Start Over” button provided, that when clicked should do the following:

- if the client is waiting for a response from the server, close the socket to end current request
- stop the AsyncTask
- and then jump back to Client screen.

Logging to the UI

You should to use the `onPreExecute()`, `onProgressUpdate()` and `onPostExecute()` methods in the AsyncTask object to update the UI. However these functions mustn't be called directly:

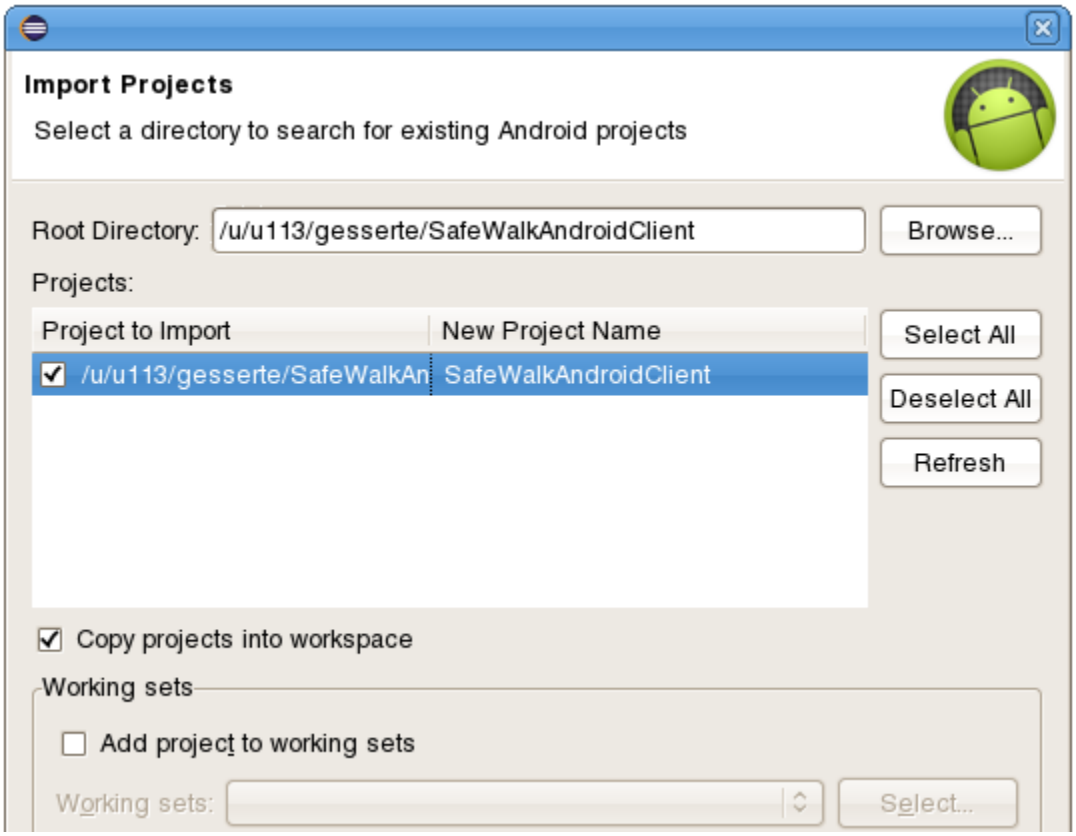
- `onPreExecute()`: automatically called before running the worker
- `onProgressUpdate()`: called through `publishProgress()` method
- `onPostExecute()`: automatically called after exiting the worker properly.

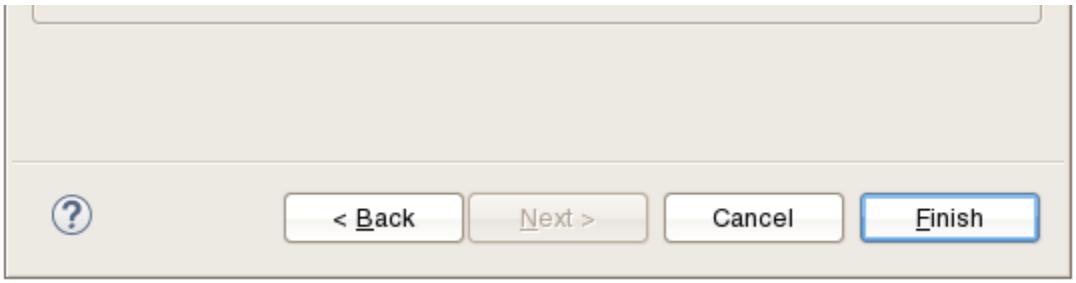
TA App example

The TA will let you have access to our solution on their Kindle. It may help you to have ideas and have a better understanding of the behavior.

Coding

The skeleton code for this part of the lab can be downloaded from SafeWalkAndroidClientSkeleton.zip. Download the file and unzip it (not in your workspace). In Eclipse do: File > import, > Android > Existing Android Code Into Workspace. Browse to the folder where you unzip the Skeleton and selected it. You should have this window:





Select “Copy project into workspace” and choose a name for your project (instead of MainActivity, here I choose SafeWalkAndroidClient).

Read carefully the code and comments which are adding more information. Some `TODO`s are there to help you and put you in the good direction, however they don't have to be followed if you have your own approach.

The package name contains 'YL' which should be replaced by your username. Here an easy way to do it in Eclipse:

- First open the file `AndroidManifest.xml`, and modify the line 3 (Only the line 3)
- Then click on the package and press F2, modify the name, leave the two first checkboxes checked and press OK
- Delete the bad import 'import edu.purdue.YL.R' in the MainActivity and Fragments classes.

Submission

You will need to turn in your android app via `turnin` before the deadline:

```
$ turnin -c cs180 -p project6 safewalk-app
```

where `safewalk-app` should be the directory that contains your android app project files.

And then, in the lab sessions that follow, you need to bring your Kindle Fire, with your submitted app installed, to your lab session, and demo your app to your lab TA. TAs will grade it manually according to the grading rubric form below.

Grading Rubric

The grade of your project is distributed below:

Item	**Requirement**	**Maximum Grade**	**Actual Grade**
Interface	No need to polish the UI to a professional level, but it should not be anti-human.	20%	
Host form	The app will connect to the server specified in the form, not one hard-coded.	5%	
Client form	All four fields are configurable as specified. Clear layout. Proper widgets used.	5%	

Item	Requirement	Maximum Grade	Actual Grade
Form parsing	Able to detect invalid data in host (5%) and client forms (2.5% for each widget).	15%	
Socket	Can report failures to connect to a server.	5%	
Socket	Can send the proper message to the server.	10%	
Socket	Acknowledge.	5%	
Socket	Can receive `RESPONSE:` responses from the server.	10%	
Matching Result	Can display the result of single pairing responses correctly.	10%	
Matching Result	Hitting “Start Over” will end client request, close socket and jump to Client screen.	10%	
Message Handler	No operations above can block the UI or crash the app.	5%	