Jomo Kenyatta University of Agriculture and Technology (JKUAT)

Department of Computing

ICS2406: Computer Systems Project

System Analysis and Design

REF:JKU/2/83/022

Project Title: Cheaper Exchange of Information Via Wireless Technology

Author:
Name: Kairu Joshua Wambugu Reg. No: CS281-0720/2011
Submission Date: _____ Sign: _____
Course: Bsc. Computer Science

Supervisor 1:
Name: Professor Waweru Mwangi Sign: _____ Date: _____
Supervisor 2:
Name: Doctor Petronilla Muthoni Sign: _____ Date: _____
Supervisor 3:
Name: _____ Sign: _____ Date: _____

Period: June 2015

Contents

1 Background/Introduction

This project aspires to create a method through which two devices might communicate via wireless technology at a cheaper cost. The project aims to achieve this goal by using Wi-Fi communication between two mobile devices. The next few pages outline the system analysis and system design processes behind the system.

2 System Analysis

In this section, we will look at two requirements needed by any computer system:

a) Functional requirements.

b) Non-functional requirements.

These are discussed starting now.

2.1 Functional requirements

The system has two functional requirements.

First, it should allow **two smartphones** to **connect to each other over wireless**. This should be done **without** the use of **an intermediary device**. This will be achieved using relevant code on both phones, as well as a WiFi connection between the two.

Second, the system should allow **audio data** to be **transferred between the two connected phones**. This is also to be done using code.

2.2 Non-functional requirements

These are also two.

First, the system should be **secure**. This is important because audio can easily be recorded and misused by malicious individuals. Security over WiFi has always been a challenge. Since we cannot control the circumstances and motives of all who will end up using this system, we can only hope that they will set up their

WiFi connections over a protected wireless link. During the implementation phase of this project, we plan to use a connection protected by WPA-PSK2.

Second, the system should provide for **audio clarity**. The sound heard by individuals on both ends of the device should be clear enough for them to recognize. We hope that the inbuilt sound processing systems of the devices we will use have this facility in place.

3 System Design

To show the design of this system, we will use the following diagrams:

a) The Network Topology Diagram;

b) Class Diagrams;

c) Activity Diagrams;

d) The Sequence Diagram; and

e) The Navigation Diagram.

3.1 The Network Topology Diagram

Since the project is centered on creating a connection between two devices, we have to outline how the network will look. This is highlighted in the network topology diagram in Figure 1.

The figure shows two phones serving as server and client. The numbers close to the points where arrows touch the phones indicate the number of servers and clients allowed to communicate with each other at a moment. The 1's close to where the arrows touch the phones show that there can only be one server communicating with one client at any particular point in time.

This understanding is important as we design the system further since it ensures that we will not focus on broadcasts, multicasts, or any other one-to-many communication form.

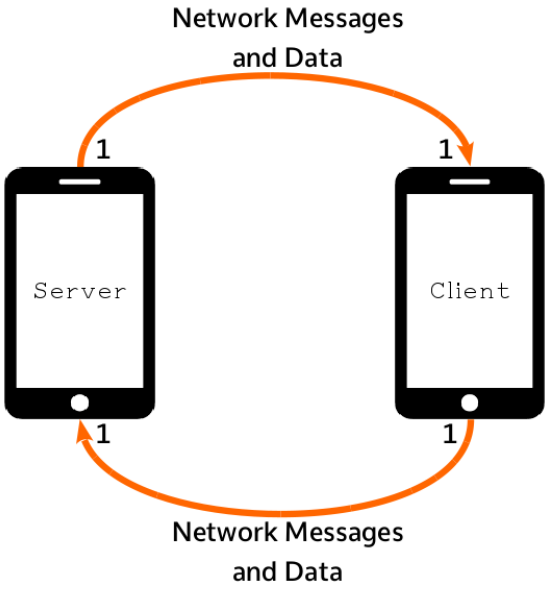

Figure 1: The Network Topology Diagram

3.2 Class Diagrams

Class diagrams show the various classes used in a software system as well as how these classes are related.

Our system has a total of 11 classes. These classes are explained below.

a) `HomeActivity` **class.** This class is the point at which the user lands when they fire up the system.

   The class diagram for the `HomeActivity` class is seen in Figure 2

b) `ReceiveCallActivity` **class.** This class waits for a call to arrive and processes the requests of the caller.

   The class diagram for the `ReceiveCallActivity` class is seen in Figure 2.

c) `IncomingCallActivity` **class.** This class alerts the user to an incoming call. It allows the user to their accept or reject the call.

   The class diagram for the `IncomingCallActivity` class is seen in Figure 2.

d) `CallInSessionActivity` **class.** This is the class where audio data is sent and received between the two connected devices.

   The class diagram for the `CallInSessionActivity` class is seen in Figure 2.

Figure 2: Class Diagrams for `HomeActivity`, `ReceiveCallActivity`, `IncomingCallActivity`, `CallInSessionActivity` and `MakeCallActivity`

Figure 3: Class Diagrams for `CallingActivity`, `Contact`, `SearchForContactsThread`, `WifiStateChangeBroadcastReceiver`, `MakeCallActivity`, `SendRecordedSoundTimerTask` and `SocketServerThread`

e) `MakeCallActivity` **class.** This class allows the user to select a contact to call.

The class diagram for the `MakeCallActivity` class is seen in Figure 2.

f) `CallingActivity` **class.** In this class, the user calls the contact selected in the `MakeCallActivity` class. If the contact is not a viable receiver or is busy, this class informs the user of the same. When the called individual picks up, this class directs the user to an instance of `CallInSessionActivity`.

The class diagram for the `CallingActivity` class is seen in Figure 3.

g) `Contact` **class**. This class represents a contact on the contact list found in

`MakeCallActivity`.

The class diagram for the `Contact` class is seen in Figure 3.

h) `SearchForContactsThread` **class**. This class extends the Java `Thread` class and is used to search for contacts from available hotspots. Searching for contacts is fairly expensive in terms of time. This means that doing it from the same thread as the UI thread would make the app look like it is hanging. To avoid this circumstance, in this class we search for contacts using a thread different from the UI thread.

The class diagram for the `SearchForContactsThread` class is seen in Figure 3.

i) `WifiStateChangeBroadcastReceiver` **class**. This class extends the Android `BroadcastReceiver` class. Whenever the state of an Android device's WiFi changes, the new state is broadcast throughout the applications of the device. To receive that broadcast, an application needs to extends the `BroadcastReceiver` class. Since our system needs to be informed the moment WiFi is turned on so as to scan for contacts, we use the `WifiStateChangeBroadcastReceiver` class and configure it to receive the "WiFi on" state.

The class diagram for the `WifiStateChangeBroadcastReceiver` class is seen in Figure 3.

j) `SendRecordedSoundTimerTask` **class**. This class extends the Android `TimerTask` class. `TimerTask` classes are used to perform actions at specific intervals of time. We use the `SendRecordedSoundTimerTask` class to record audio every second and to send the recorded audio every second.

The class diagram for the `SendRecordedSoundTimerTask` class is seen in Figure 3.

k) `SocketServerThread` **class**. Like the `SearchForContactsThread` class above, this extends the Java `Thread` class. It is used by the `ReceiveCallActivity` class to listen for any connections coming in from the `CallingActivity` class.

The class diagram for the `SocketServerThread` class is seen in Figure 3.

A summarized class diagram showing relationships between classes is called an elided class diagram. Our elided class diagram is seen in Figure 4. The next few paragraphs explain the elided diagram.

To begin with, we see `HomeActivity` can start one of either `ReceiveCallActivity`

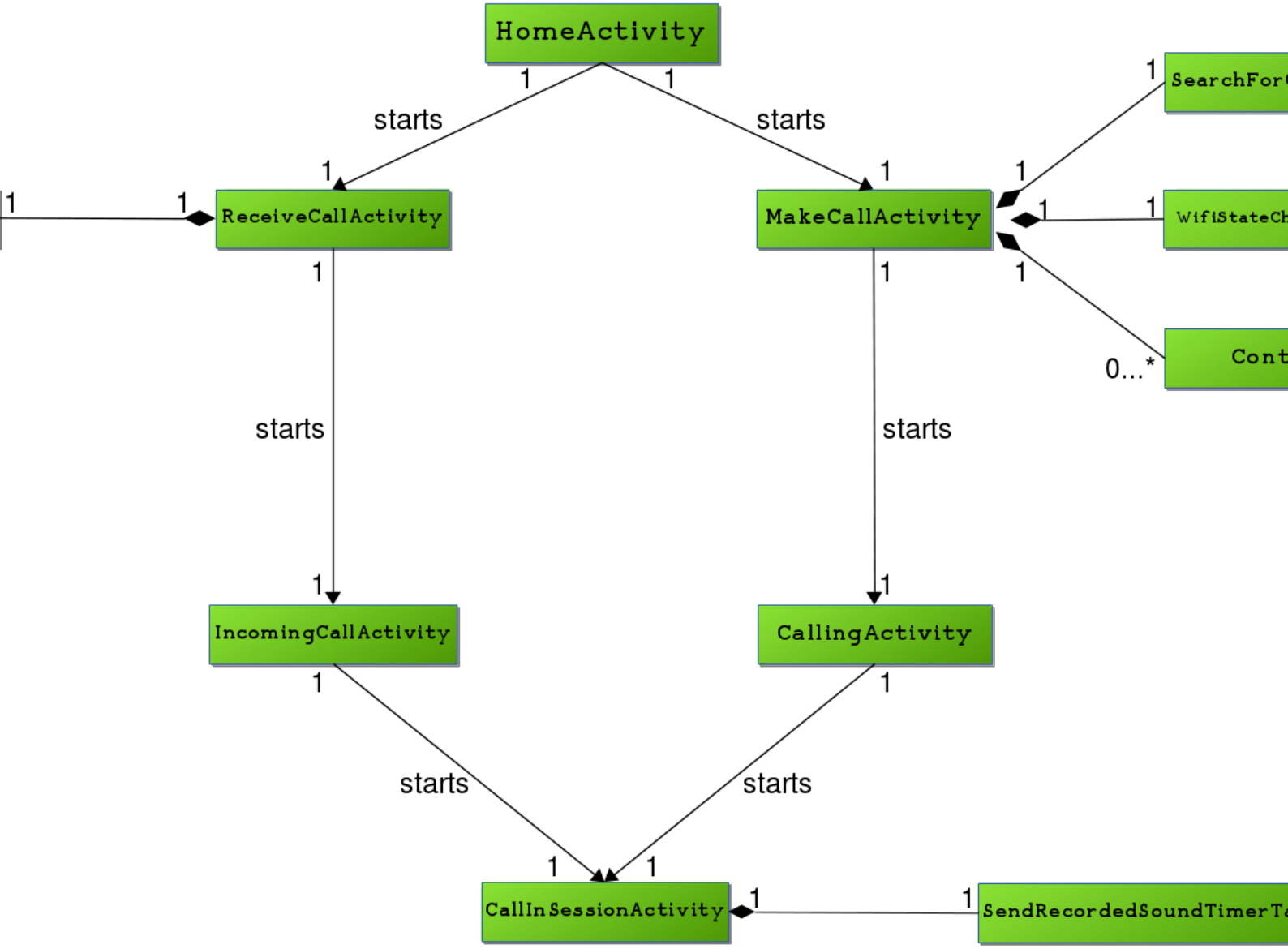

Figure 4: Elided Class Diagram

or MakeCallActivity. HomeActivity cannot start both these two classes at the same time.

The filled diamond on the line connecting ReceiveCallActivity to SocketServerThread tells us that ReceiveCallActivity has a SocketServerThread instance and that the SocketServerThread instance cannot exist without the ReceiveCallActivity instance. This is so because the SocketServerThread class exists to assist the ReceiveCallActivity class connect to clients. The ReceiveCallActivity can start only one instance of the IncomingCallActivity class.

IncomingCallActivity can start only one instance of CallInSessionActivity. There is not much else IncomingCallActivity can do in terms of inter-class interactions.

MakeCallActivity has one SearchForContactsThread object and one WifiStateChangeBroadcast object. However, it can have as many Contact objects as possible. MakeCallActivity uses a SearchForContactsThread object to search for contacts and display them. It uses a WifiStateChangeBroadcastReceiver instance to be alerted whenever the WiFi is turned on. And MakeCallActivity can have as many Contact objects as possible since we cannot determine beforehand how many hotspots there will be around the place of application execution. The last thing about MakeCallActivity is that it can start only one instance of the CallingActivity class.

Just like IncomingCallActivity, CallingActivity can start only one CallInSessionActivity object and can do nothing much else.

From the diagram, we see that CallInSessionActivity instances can be instantiated by either IncomingCallActivity instances or CallingActivity instances. Each of those last two mentioned classes can start only one CallInSessionActivity object each. CallInSessionActivity has a SendRecordedSoundTimerTask object which it uses to send recorded sound every second and to receive recorded sound and play it every second.

3.3 Activity Diagrams

Activity diagrams highlight the order in which instructions are executed in an instance of a class.

In our case, all classes which have a set of instructions to execute have their activity diagrams drawn. Explanations are as follows.

a) HomeActivity **class.**

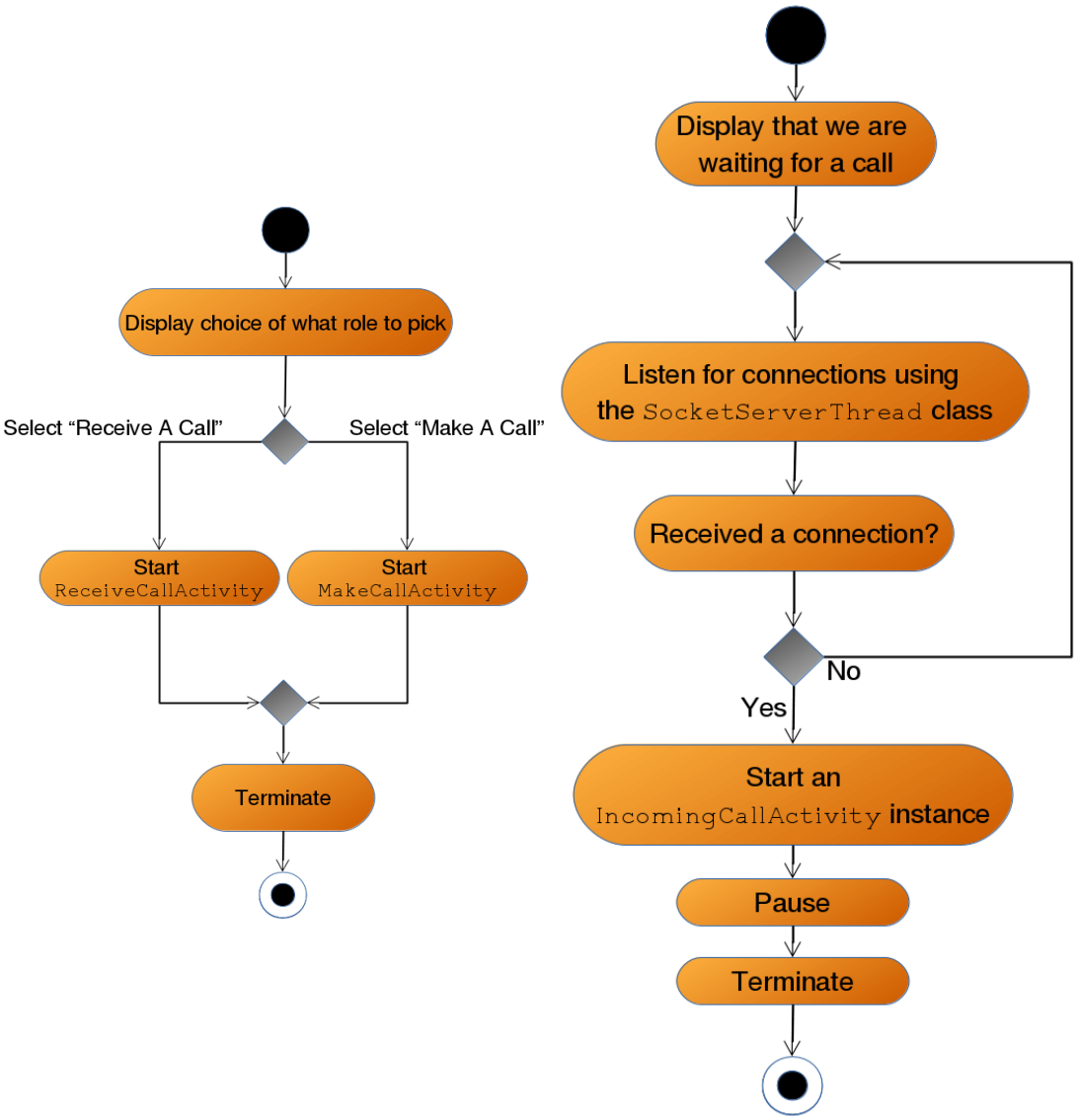

Figure 5: Activity Diagram for `HomeActivity` on the left and `ReceiveCallActivity` on the right

The activity diagram for the HomeActivity class is seen in Figure 5

The class starts off displaying to the user a UI that allows him/her to choose which role they want to be: caller or receiver. If the user chooses to make a call, the class starts the `MakeCallActivity` class. If the user chooses to receive a call, the class starts the `ReceiveCallActivity`. After processing user requests, `HomeActivity` terminates.

b) `ReceiveCallActivity` **class.**

The activity diagram for the `ReceiveCallActivity` class is seen in Figure 5.

We begin by displaying to the user that the system is waiting for a call. We then begin listening for any incoming connections using an instance of the `SocketServerThread` class. We keep doing this listening until we receive a connection from the `CallingActivity` class. On receiving such a connection, we start an instance of the `IncomingCallActivity` class and pause the `ReceiveCallActivity` class. An option is there to terminate the `ReceiveCallActivity` if necessary.

c) `IncomingCallActivity` **class.**

The activity diagram for the `IncomingCallActivity` class is seen in Figure 6

In this class, we start by displaying the needed UI so that the user might know that a call is incoming. We then retrieve references to the connection established in the parent activity of this class. The parent activity is the `ReceiveCallActivity` since instances of the `IncomingCallActivity` class are started off by instances of the `ReceiveCallActivity` class. Part of the UI of the `IncomingCallActivity` class contains two buttons: one for accepting the incoming call and the other for rejecting it. If the user taps the "Accept" button, the `IncomingCallActivity` class starts a `CallInSessionActivity` activity then terminates. If the user selects "Reject" then `IncomingCallActivity` simply terminates. In the latter case, since `IncomingCallActivity` was started by `ReceiveCallActivity` and since `ReceiveCallActivity` had been paused after starting `IncomingCallActivity`, `ReceiveCallActivity` is resumed.

d) `CallInSessionActivity` **class.**

The activity diagram for the `CallInSessionActivity` class is seen in Figure 6.

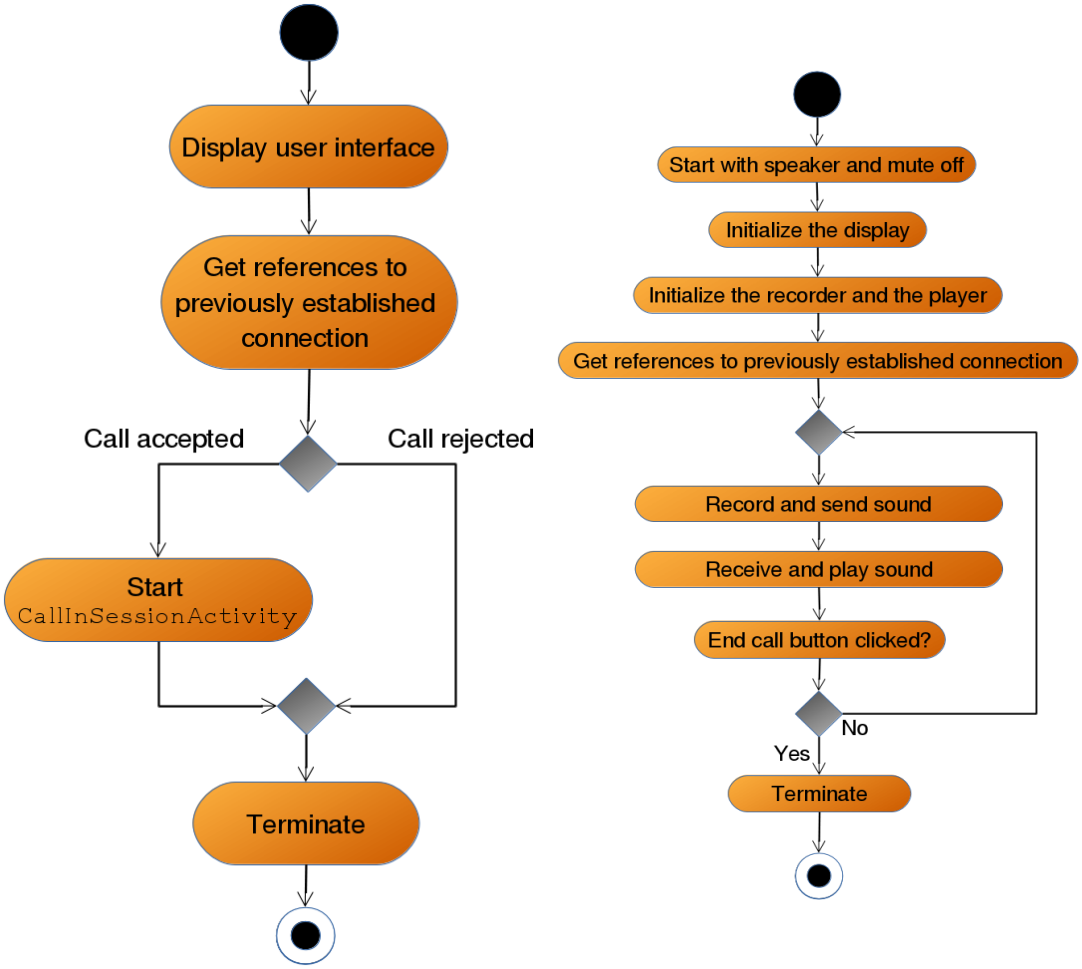

Figure 6: Activity Diagram for `IncomingCallActivity` on the left and `CallInSessionActivity` on the right

This class begins by initializing the speaker and mute to be off. It sets up the display, initializes the media recorder used for recording sound and initializes the media player used to play sound. It also gets references to the previously established connection. These references come from the class that started this class. As we saw in the class diagrams, `CallInSessionActivity` can be started by either `CallingActivity` (from the caller's side) or `IncomingCallActivity` (from the receiver's side). Either parent class has references to the established network connection and passes these to the `CallInSessionActivity` class. The UI in `CallInSessionActivity` has a button for ending the call. As long as this button is not clicked, the class records sound and sends it to the `CallInSessionActivity` instance on the other side of the network. The class also receives sent sound and plays it. As soon as the end button is clicked, the class terminates.

e) `MakeCallActivity` class. This class allows the user to select a contact to call.

The activity diagram for the `MakeCallActivity` class is seen in Figure 7.

We begin by displaying a user interface which will show the user the contacts available for him/her to call and allow the user to scan for contacts. The class also gets the SSID of the user's device. This will be used to identify the user when he/she calls another. After this, the class initializes a `SearchForContactsThread` object, which will be used to search for contacts when the user elects to scan for some. The final initialization step is to turn on the WiFi. If the user scans for contacts and none are found, they can keep scanning for them. Any contacts that the class finds will be displayed in a list. When the user selects a contact to call, `MakeCallActivity` establishes a network connection with the selected contact, starts `CallingActivity`, then pauses itself. The user has the option of eventually terminating this class if he/she sees fit.

f) `CallingActivity` **class.**

The activity diagram for the `CallingActivity` class is seen in Figure 7.

As usual for an `Activity` class, this class begins by displaying the relevant UI. It then waits for five seconds to simulate the behavior users might expect during the making of a call. Without this delay, the app will execute very quickly and that might leave users a bit confused. After the five seconds, we get references to the previously set up connection. This connection is the one set up at `MakeCallActivity`. Then we process the messages coming from the `IncomingCallActivity`, which by this time should be running on

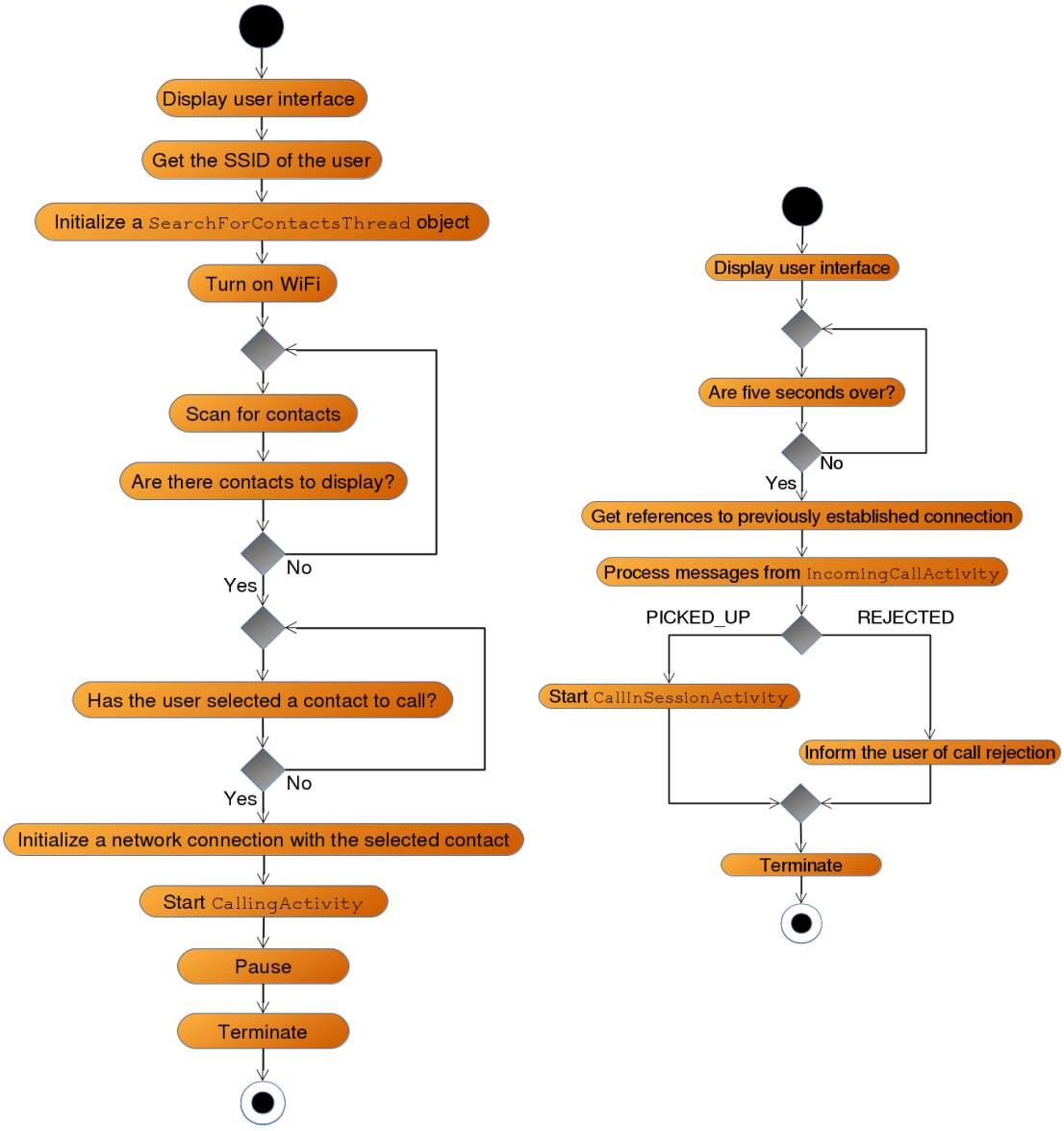

Figure 7: Activity Diagram for `MakeCallActivity` on the left and `CallingActivity` on the right

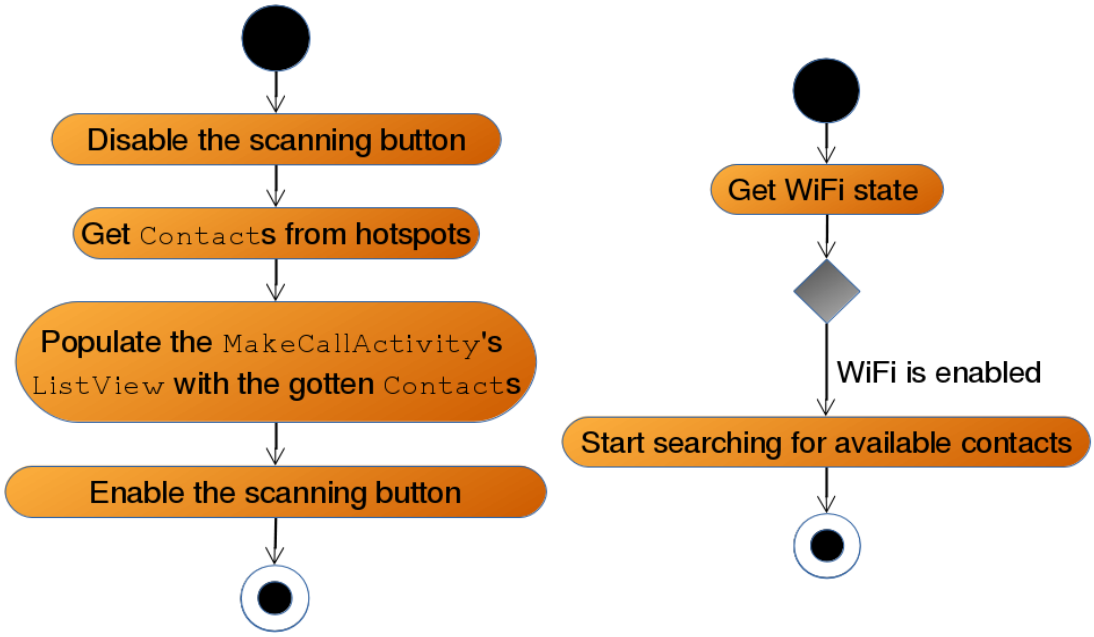

Figure 8: Activity Diagram for `SearchForContactsThread` on the left and `WifiStateChangeBroadcastReceiver` on the right

the device on the other side of the network. If the `IncomingCallActivity` sends a "PICKED_UP" message, this means that the other user has picked up the call and so we start `CallInSessionActivity`. If we receive a "REJECTED" instead, this means that the user does not desire to communicate now so we change the UI inform the caller of this. After processing the messages, we terminate the `CallingActivity` instance, leaving either the newly started `CallInSessionActivity` instance or the previously paused `MakeCallActivity` instance to continue interacting with the user.

g) `SearchForContactsThread` **class.** The activity diagram for the `SearchForContactsThread` class is seen in Figure 8.

This class starts out by disabling the button used to scan for contacts. This is done to show the user that they cannot scan until the previous scan request has been fully processed. Thankfully, processing scan requests in a separate thread takes a rather short time. The class then initializes a set of `Contact` instances from the hotspots found after scanning. After this, the class populates the `ListView` of `MakeCallActivity` with this list of `Contacts`. The last thing done by this class is enabling the scanning button.

h) `WifiStateChangeBroadcastReceiver` **class.**

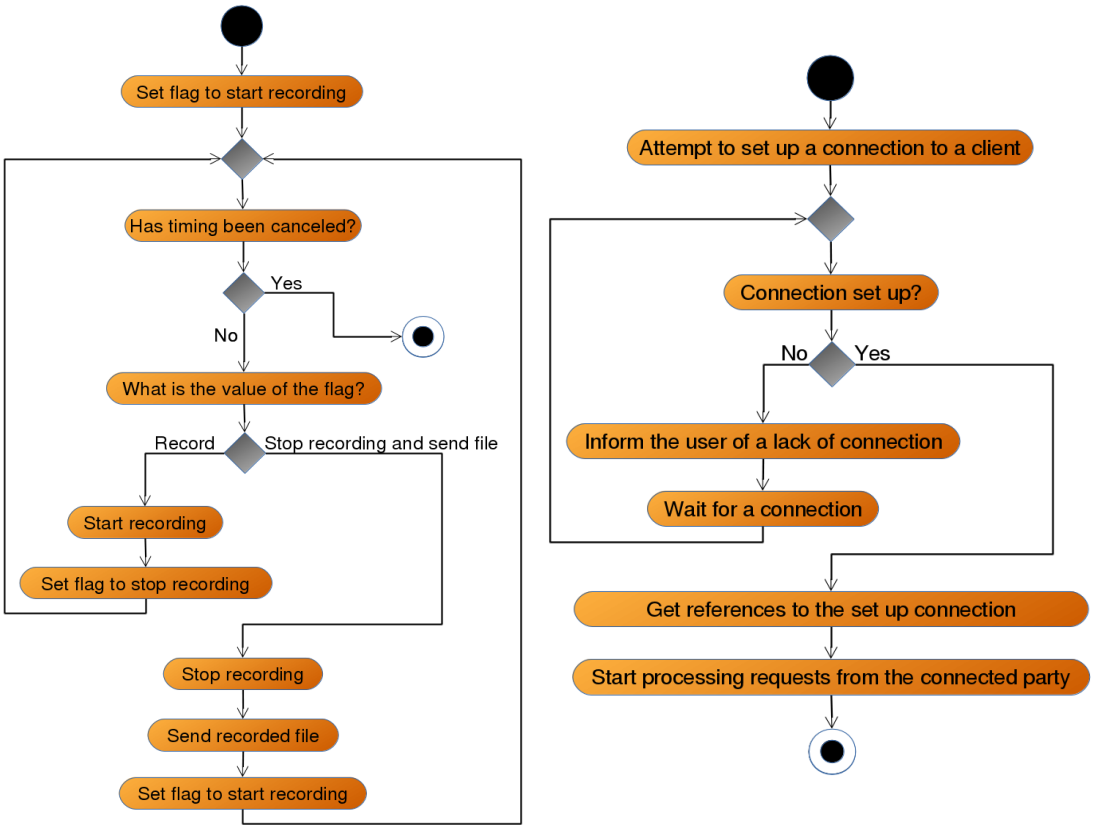

Figure 9: Activity Diagram for `SendRecordedSoundTimerTask` on the left and `SocketServerThread` on the right

The activity diagram for the `WifiStateChangeBroadcastReceiver` class is seen in Figure 8.

What happens in this class is fairly simple. We get the wait for the WiFi state to change. It changes to the state where it is enabled, we start searching for available contacts and then terminate after doing so.

i) **`SendRecordedSoundTimerTask` class.**

The activity diagram for the `SendRecordedSoundTimerTask` class is seen in Figure 9.

This class is very important for the sending and receiving of audio data between `CallInSessionActivity` instances. What happens in this class is generally controlled by a flag which oscillates between the "record" and the "stop recording and send recorded file" states. Since the class does its

job based on fixed intervals of time, if timing is canceled then the class terminates. The class starts off with the flag set to the "record" state. In this state, the class starts recording sound for a fixed amount of time. When that is finished, it sets the flag to the "stop recording and send file" state. As long as the timing is not canceled, processing continues. The value of the flag is read and found to be "stop recording and send file". As long as the timing is not canceled, processing continues. With timing still not canceled, the class checks the flag, sees that it reads "record", and goes about executing the "record" state. This cycle goes on until the timing gets canceled.

j) `SocketServerThread` **class.**

The activity diagram for the `SocketServerThread` class is seen in Figure 9.

This class is where most of the networking is done. We first attempt to connect to a client. If the attempt fails, we inform the user of a failure and then we keep waiting for a connection. If we are able to set up the connection, we then get references to that connection. These references are important since they help us send data to and receive data from device at the other end. With the references acquired, we can now start processing requests from the connected party.

3.4 The Sequence Diagram

A sequence diagram is used to show how different objects in a system interact with each other. The sequence diagram for this project is shown in Figure 10. The table in Figure 11 explains what each of the text styles in the sequence diagram represents.

Below is a brief explanation of what happens.

- `HomeActivity`: The system starts at this activity. The user chooses if they want to make a call or receive a call. Depending on the user's choice, the system calls a `startActivity()` for an instance of either the `MakeCallActivity` class or the `ReceiveCallActivity` class. After doing this, the `HomeActivity` activity finishes and closes down.
- `MakeCallActivity`: This activity is started when the user decides to make a call. Here, the user can select which individual to call. When the individual to be called is chosen, the `MakeCallActivity` activity calls the `CallingActivity` activity which will attempt to call the chosen individual. The sequence diagram shows that two extras are passed to the `CallingActivity` activity. An extra is a piece of data passed between Android activities. These two extras are the name of the called individual as well as the name of the individual calling. After starting the `CallingActivity` activity, the `MakeCallActivity` activity suspends itself but does not close down.
- `CallingActivity`: This activity sends a "DISCOVER" network message to the `ReceiveCallActivity` activity of the individual being called. Along with the "DISCOVER", the `CallingActivity` activity sends the name of the caller to the `ReceiveCallActivity` activity. The `ReceiveCallActivity` activity responds with an "OFFER" which confirms that it is active and has received the name of the called. As the diagram shows, the `ReceiveCallActivity` instance starts an `IncomingCallActivity` activity as soon as it sends the"OFFER". The `CallingActivity` activity then sends a "CALLING YOU" network message, which will be received by an instance of the `IncomingCallActivity`. When the `IncomingCallActivity` activity sends the `CallingActivity` activity a "PICKED UP" in response to the "CALLING YOU", the `CallingActivity` activity starts an instance of the `CallInSessionActivity` activity. The `CallingActivity` passes extras to the `CallInSessionActivity` containing the name of the called individual as well as the name of the parent activity. Since the user got to the `CallInSessionActivity` activity via the `MakeCallActivity` activity, the parent activity in this case is the `MakeCallActivity`. After passing these extras, the `CallingActivity` activity finishes and closes

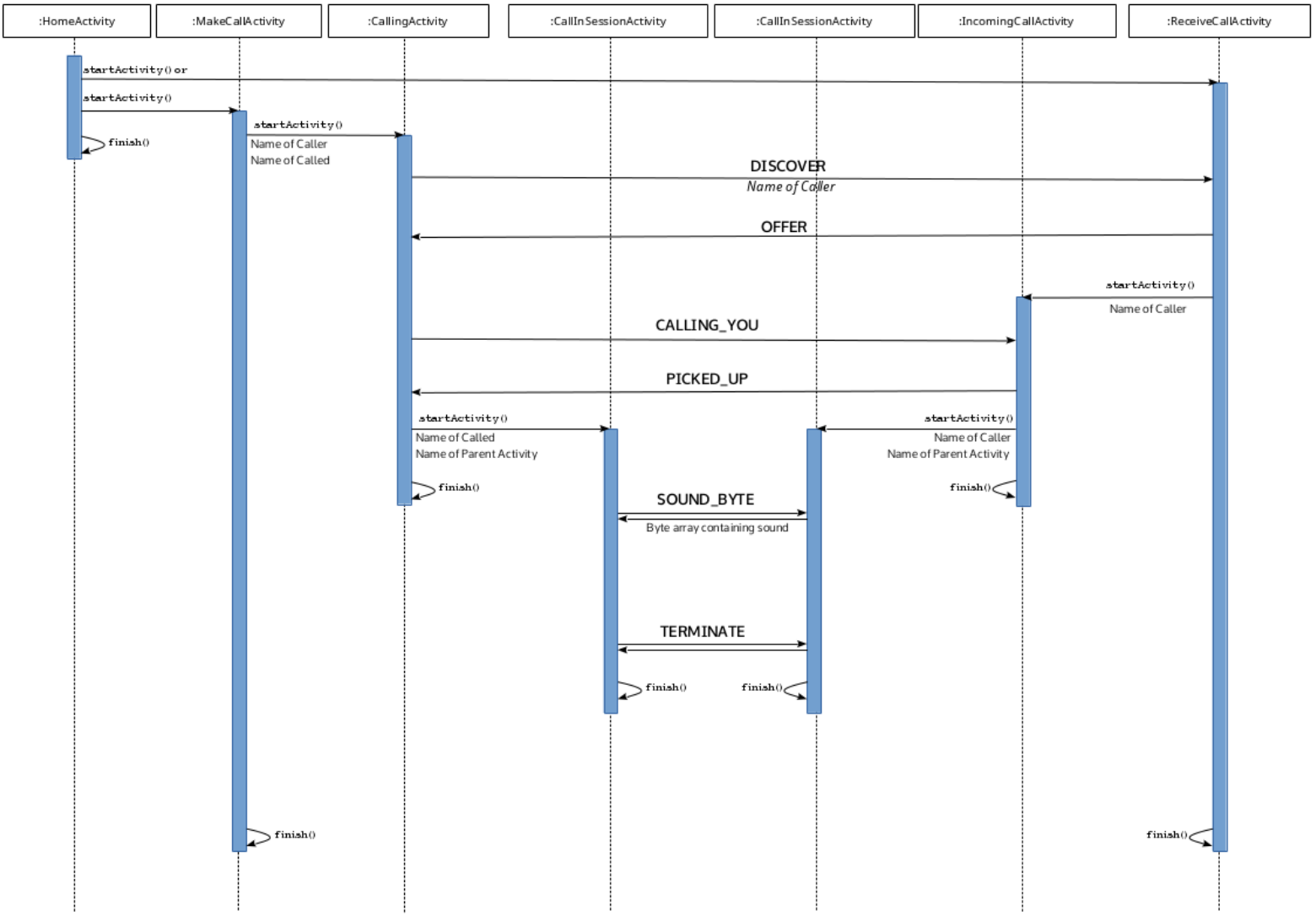

Figure 10: The Sequence Diagram

Component	*Text Style*
Activity object	:HomeActivity
Method	`startActivity()`
Extra passed to activity object	Name of Caller
Network message	**DISCOVER**
Data sent together with network message	*Name of Caller*

Figure 11: The Sequence Diagram Legend

down. Remember that the `MakeCallActivity` instance is still suspended in the background.

- `ReceiveCallActivity`: This activity is started when the user decides to wait to receive a call. When the `ReceiveCallActivity` activity receives a "DISCOVER" from a `CallingActivity` instance, it responds with a "OFFER" and starts an `IncomingCallActivity` instance – passing an extra containing the name of the caller. After starting the `IncomingCallActivity` activity, the `ReceiveCallActivity` activity suspends itself but does not close down.
- `IncomingCallActivity`: This activity waits for a "CALLING YOU" from the `CallingActivity` activity and responds to it with a "PICKED UP" as soon as the called individual picks up the call. After sending the "PICKED UP" the `IncomingCallActivity` instance starts the `CallInSessionActivity` activity, passing extras containing the name of the caller and the name of the parent activity. Since the user got to the `CallInSessionActivity` activity via the `ReceiveCallActivity` activity, the parent activity in this case is the `ReceiveCallActivity`. After passing these extras, the `IncomingCallActivity` activity finishes and closes down. The `ReceiveCallActivity` instance still remains suspended in the background.
- `CallInSessionActivity`: Here is where the actual voice transfer happens. `CallInSessionActivity` instances on the side of the caller and the called alternate in sending sound snippets. They do this by sending to each other a "SOUND BYTE" network message and a byte array containing a sound snippet immediately after that. This happens in fixed intervals so that sound is heard continuously. When the users decide to end the call, both `CallInSessionActivity` instances send each other "TERMINATE" network messages and close down by calling `finish()`. The system then goes back to the corresponding background activity, either `MakeCallActivity` for the caller or `ReceiveCallActivity` for the person being called.

3.5 The Navigation Diagram

An application's navigation diagram shows how a user can maneuver through the said application. Figure 12 shows the navigation options available to the user of the application being created in this project.

The user starts off at the `HomeActivity` activity and selects whether to make a call or to receive a call.

If, on the one hand, the user elects to make a call, they are directed to the

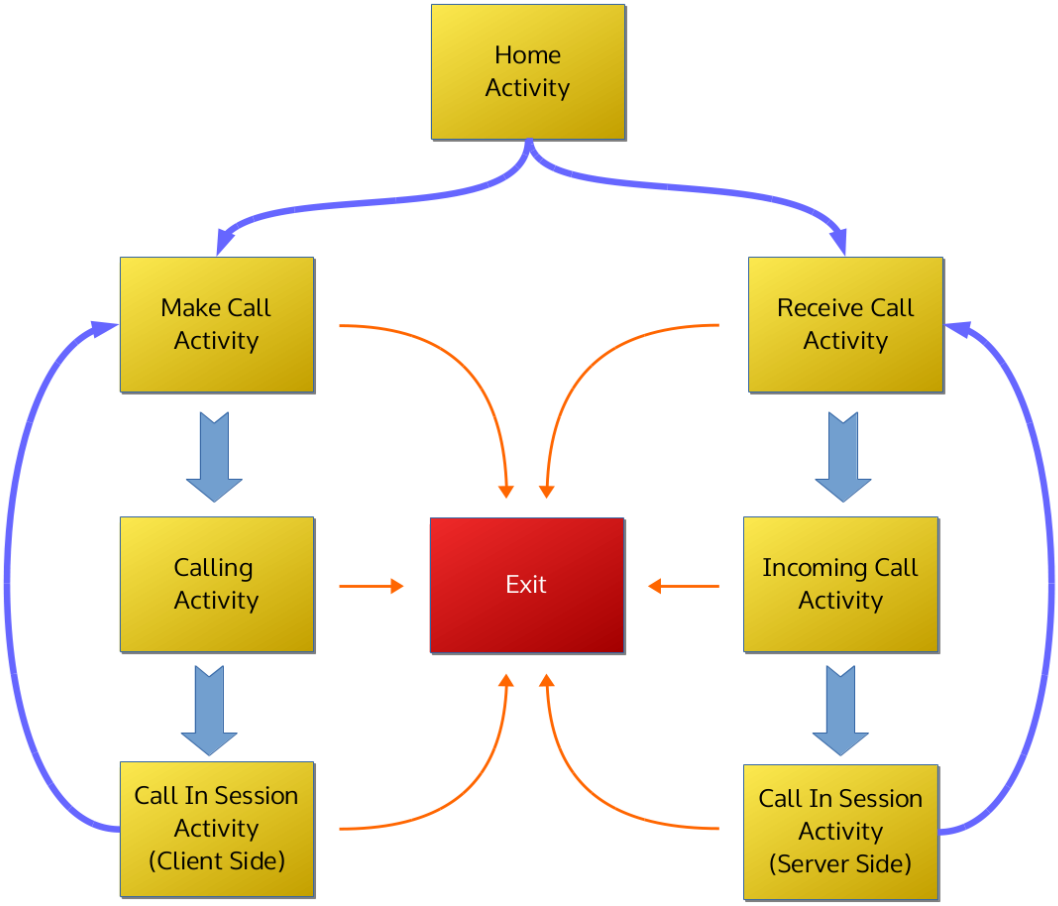

Figure 12: The Navigation Diagram

`MakeCallActivity` activity where they choose a contact to call. Choosing a person to call leads them to the `CallingActivity` activity where the system attempts to connect to the phone being called. Successful connection leads to the user getting to the `CallInSessionActivity` activity where they can talk with the other individual. As soon as the call ends, the user is returned to the `MakeCallActivity` activity. Unsuccessful connection returns the user to the `MakeCallActivity` activity.

If, on the other hand, the user decides to receive a call, they are taken to the `ReceiveCallActivity` activity where they wait for an individual to call them. When a call comes in, the user is taken to the `IncomingCallActivity` activity where they can decide to reject or accept the call. Should the user reject the call, they are returned to the `ReceiveCallActivity` activity. Should the user accept the call, they are presented with the `CallInSessionActivity` activity where they can talk with the person on the other side of the line. At the end of the call, the user is taken back to the `ReceiveCallActivity` activity to wait for another call.

The orange arrows show that it is – at least currently – possible for the user to exit the system at any time.