Simulation of WiFi Direct

Lab Guide - Week 7

Mobile Computing - 2012/13

MEIC/MERC

Objectives

This document provides some guidance for simulating the WiFi Direct technology. It starts with an overview of the WiFi Direct Simulator (WDSim), the recommended framework for enabling P2P communication between Android emulator instances in the context of the CMov project. Then, the document provides instructions for exploring and extending the current implementation of WDSim. This presentation unfolds around the core components of the framework: console and API.

1. Overview of the WiFi Direct Simulator

The WiFi Direct Simulator (WDSim) is a framework for simulating the WiFi Direct API of the 4.0+ Android platform in the context of the CMov project. The WDSim aims to implement the subset of services of the WiFi Direct API that are required to implement the decentralized version of the project. WDSim mimics as close as possible the programming model of Android's API.

WDSim simulates three core services of WiFi Direct: peer discovery, P2P connection, and data transfer. Peer discovery enables an emulated device to detect the peers located in its vicinity and be notified of any changes to the set of available peers. After knowing which peers are available, an emulated device can initiate a P2P connection with one of them, and create a WiFi P2P group, if necessary. In this process, peers must agree who plays the client and who plays the group owner (GO) role. The GO manages the P2P group and assigns an IP to the client. Data transfer ensues the P2P connection setup. The client can now establish a TCP connection with any node of the group.

The WDSim architecture includes two main components: the API and the console. The WDSim API is located on the emulated devices. It simulates the programming interface of WiFi Direct and implements the protocols for interacting with other emulator devices and with the WDSim console. The WDSim console is a standalone Java application that simulates the physical movement of the emulated devices. The console offers its users the ability to define the *proximity topology* of the nodes, i.e. the set of peers that each device can reach at a given time. The console can then propagate this information to the emulated devices, information that the WDSim API uses to simulate and forward peer discovery events to applications.

The WDSim framework has been partially implemented and its code published. The current version includes the implementation of the console and a the implementation of the service discovery part of the API. However, in order to meet the basic requirements of WiFi Direct applications, it is necessary to implement the P2P connection and data transfer protocols. This guide helps to

understand the parts that are implemented and provides directions for implementing the remaining parts. Download the WDSim distribution from the website and import it into eclipse. We will use it in the remainder of this guide. We focus on the WDSim components separately: console and API.

2 The WDSim Console

Suppose we want to simulate a situation where three emulated devices A, B, and C are waiting on a line; A and B can talk, B and C can talk, but not A and C. Then, a fourth device D comes into the picture and goes past them, moving from the tail (A) to the head of the queue (C). WDSim allows us to do this by specifying how the proximity topology changes. This is how.

1. Launch the WDSim console: In eclipse, pick the WDSim console project, build it, and execute it. We can now issue commands to the WDSim console, which looks like this:

```
WiFi Direct Simulator

  Working Directory = /Users/nsantos/Documents/CM/workspace/CMov-SimWifiP2P-Console
  Type "help" or "h" for the full command list

>
```

Commands can be issued using their full names or abbreviations. Use the help command to check out the complete list of command names and respective abbreviations.

2. Register the devices: Emulated devices must be registered with a unique name and address. The address is a combination of IP and port of the emulator instance. For now, we use bogus addresses. To register devices A, B, C, and D, issue the *register* command sequence:

```
>register A 1.1.1.1 4000
>register B 2.2.2.2 4000
>register C 3.3.3.3 4000
>register D 4.4.4.4 4000
```

We can list the registered devices with *print devices*:

```
>print devices
A       1.1.1.1       4000
B       2.2.2.2       4000
C       3.3.3.3       4000
D       4.4.4.4       4000
```

At this point is already possible to check if the emulated devices are available on their respective *IP:port* addresses. This is done with the ping command:

```
>ping
```

```
A       1.1.1.1         4000    OFFLINE
B       2.2.2.2         4000    OFFLINE
C       3.3.3.3         4000    OFFLINE
D       4.4.4.4         4000    OFFLINE
```

The output shows that all devices are offline, as expected. (It is possible to remove nodes with the *unreg* command, and to clear all nodes with the *clear* command).

3. Defining device proximity: Now, we want to express that A and B are reachable to each other, and so do B and C. To represent the connectivity between devices, the console keeps a list of peers for each emulated device. Type *print neighbors* to display the neighbors list of each node:

```
>print neighbors
A =>
B =>
C =>
D =>
```

The list of neighbors of all node is initially empty, meaning that each node is isolated and cannot communicate with anyone. To change this situation, we have to use the *move* command that tells a node to move to the vicinity of a set of nodes, which will consist of its new neighbors. We can use this capability to express the proximity relationships A-B and B-C with two move commands:

```
>move A (B)
>move C (B)
>print neighbors
A => B
B => A,C
C => B
D =>
```

As we can see, A can talk to B, B can talk to A and C, and C can talk to B. Let's now represent D moving past this sequence of nodes. As D moves, its set of neighbors will change over time: first, it consists of A only, then of A and B, then of B only, then of B and C, and finally of C only.

```
>move D (A)
>print neighbors
A => B,D
B => A,C
C => B
D => A
>move D (A,B)
>print neighbors
A => B,D
B => A,C,D
C => B
D => A,B
>move D (B)
>print neighbors
```

```
A => B
B => A,C,D
C => B
D => B
>move D (B,C)
>print neighbors
A => B
B => A,C,D
C => B,D
D => B,C
>move D (C)
>print neighbors
A => B
B => A,C
C => B,D
D => C
>move D ()
>print neighbors
A => B
B => A,C
C => B
D =>
```

4. Adding delays: To be more realistic, the console allows the user to define a time that the console should wait before accepting the next command. This is the *wait* command. It accept a value in seconds, by default, but minutes is also supported. To wait 2 seconds we do:

```
>wait 2
```

5. Scripting support: Suppose that now, we want to wait 2 seconds between transitions of D. However, to prevent typing everything again, we write the sequence of commands in a file and tell the console to load it and execute it. So, first, create a file *cutline.txt* in the scripts directory of the WDSim console project, and add the following commands:

```
# register the nodes
clear
register A 1.1.1.1 4000
register B 2.2.2.2 4000
register C 3.3.3.3 4000
register D 4.4.4.4 4000

# represent the A-B-C line
move A (B)
move C (B)

# represent D's movement
wait 2
move D (A)
print neighbors
wait 2
move D (A,B)
print neighbors
wait 2
```

```
move D (B)
print neighbors
wait 2
move D (B,C)
print neighbors
wait 2
move D (C)
print neighbors
wait 2
move D ()
print neighbors
```

There are two relevant observations to make:

- The *clear* command was added in order to unregister any devices that have been previously registered in the console.
- The script supports comments by initiating lines with #

Next, we tell the console to execute the script by using the *load* command as follows (we don't display the output, which is similar to the output shown above):

```
>load cutline
```

As we can see, the console enforces the intended transition time of 2 seconds. Note that, normally, the *load* command takes as input the fully qualified name (name and path) of the scripting file. In this case, however, we only typed the name of the file and exclude the extension. To speed up typing, the consoles looks up for the file in the *scripts* subdirectory of the working directory (see the initial message displayed by the console), and assumes it is a *txt* file. (Files with txt extension can be edited in the eclipse editor.) Keep in mind, though, that this special filename resolution is only done if the console cannot locate the name of the file as provided in the input.

6. Propagating changes to devices: To mimic the device discovery of WiFi Direct, the console allows the user to push the current neighborhood sets to the emulated devices. At the emulated device's side, the WDSim API will notify the application of the peers available for communication. To push the peer information to the emulated devices, type the command *commit*:

```
>commit
B       2.2.2.2         4000    FAIL
A       1.1.1.1         4000    FAIL
D       4.4.4.4         4000    FAIL
C       3.3.3.3         4000    FAIL
```

In this case, the commit operation fails because devices are offline. To learn how to integrate the features of the console with the API, we have to study the next part.

4. The WDSim API

To demonstrate how the WDSim API works and how the console interacts with it, we use the SimpleChat app shipped with the WDSim package. Below, we can see two screenshots of this app.

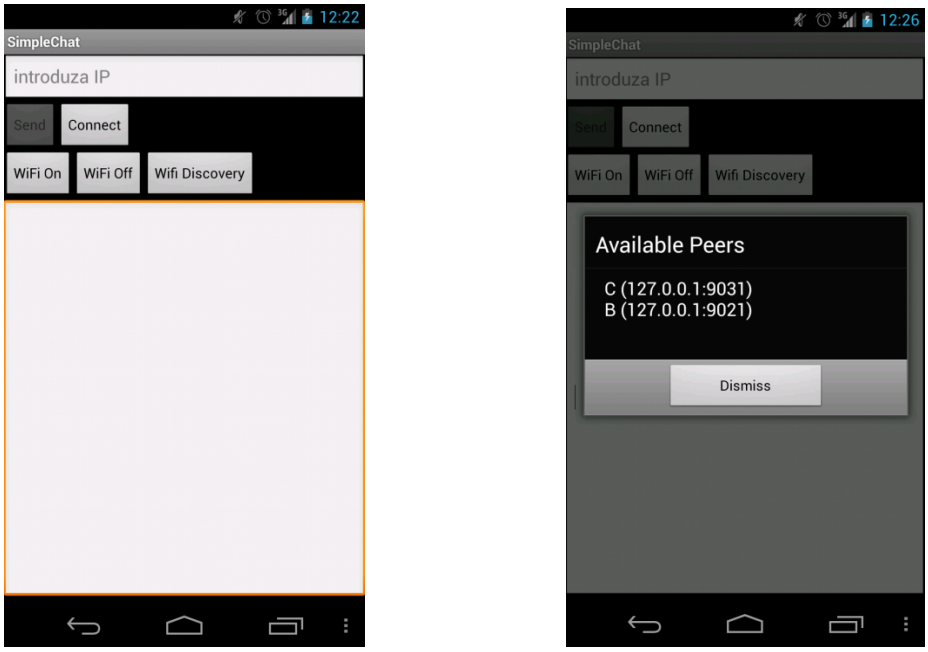

Figure 1: Screenshots of SimpleChat: the initial screen (left) and after receiving a peer notification (right)

The app's current functionality is to enable two devices to establish a TCP connection between them and exchange messages. The app subsumes both the server and client roles. As a server, it listens for connections on a particular port. As a client, the app enables the user to initiate a connection with a target device. The user only needs to input the IP of the target node.

Our ultimate goal is to modify this app so that all the communication takes place through the simulated WiFi Direct API. Hence, the communication must be intercepted and supported by the WDSim framework, and it must be restricted to only the devices available in the user's vicinity.

So far, the current SimpleChat implementation includes a few steps in the integration with WDSim. In particular, it implements the peer discovery service. By pressing the "WiFi On" button, this service is activated, and the user will be notified of peers presence 1) whenever a commit command is issued on the WDSim console, and 2) whenever the user hits the WiFi Discovery button. The peers shown in the dialog box reflect the neighborhood sets that have been submitted through the WDSim console. However, in addition to the peer discovery, it is necessary to add more features to WDSim and modify the app accordingly. Namely, P2P connection setup must be properly implemented and TCP connections must be restricted to peers that available.

In the rest of this section, we discuss in detail how to execute this app, and then provide some assignments aimed at exploring the current WDSim implementation and extending it so as to finish porting the SimpleChat application to a simulated Wifi Direct scenario.

4.1 Try Out the SimpleChat App

We now try the SimpleChat app using two emulator instances and the WDSim console. The latter controls the evolution of the wireless connectivity between both instances.

1. Launch two emulator instances A and B. Each of them will be listening to a different control port. Assume for example, that the control port numbers are 5554 and 5556, respectively.

2. In order for the apps running in the emulator instances to be reachable from the outside world, we have to use port redirection. The SimpleChat app has two open ports: 45678, which is used by the application to listen for connections from other nodes, and 9001, which is used by the WDSim API to listen for connections from the WDSim console. For now, we focus only on the latter. Therefore, let's just redirect port 9001 of emulator A to 9011 and port 9001 of emulator B to 9021. Open a terminal window and execute the following commands (in bold):

```
$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
redir add tcp:9011:9001
OK
exit
Connection closed by foreign host.
$ telnet localhost 5556
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
redir add tcp:9021:9001
OK
exit
Connection closed by foreign host.
```

3. On both emulators, hit the "WiFi On" button to launch the service of the WDSim API. This service spawns a thread that sits listening on the local port 9001, and will be waiting for instructions from the WDSim console.

4. Next, define a proximity topology as shown in the script below. This topology contains nodes A and B, both online, and C, which is offline. The script is stored in file *scripts/simplechat.txt*.

```
# SimpleChat experiment
```

```
clear
register A 127.0.0.1 9011
register B 127.0.0.1 9021
register C 127.0.0.1 9031
move A (B,C)
print neighbors
commit
```

Push this topology to the emulator instances. On the WDSim console, type the following command:

```
>load simplechat
# SimpleChat experiment
clear
register A 127.0.0.1 9011
register B 127.0.0.1 9021
register C 127.0.0.1 9031
move A (B,C)
print neighbors
A => B,C
B => A
C => A
commit
A      127.0.0.1     9011   SUCCESS
B      127.0.0.1     9021   SUCCESS
C      127.0.0.1     9031   FAIL
>
```

The emulators should reflect the committed update as shown in the figure below:

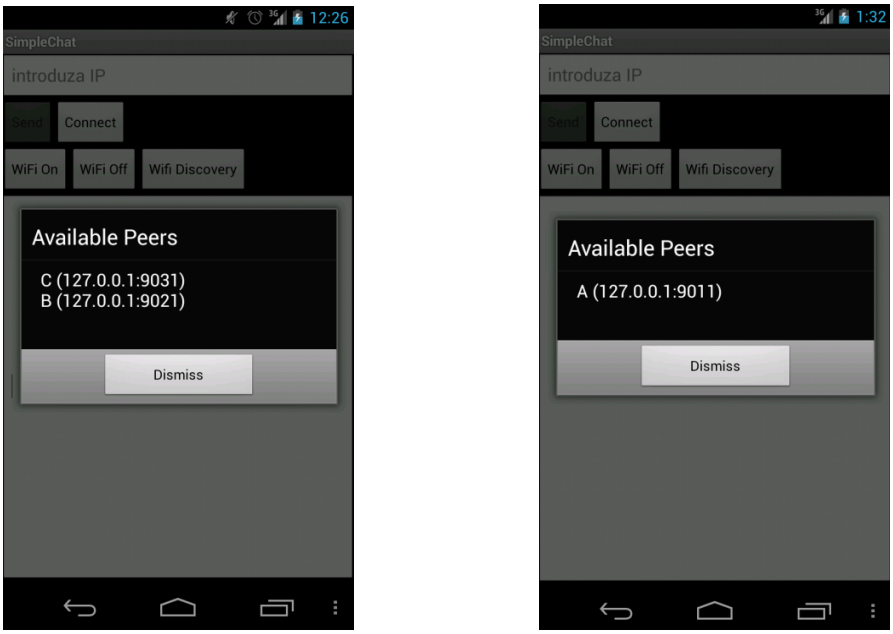

Figure 2: Screenshots of A (left) and B (right) after receiving the WDSim Console notification.

Switching off the WiFi service and switch it back on will result in the loss of peer information at the

emulators' side. As a result, the commit command must be reissued from the WDSim console.

4.2 Assignment

We terminate this guide with three assignments:

1. Compare the WDSim API with WiFi Direct API: Identify the similarities and differences between both APIs. Do this task by inspecting the source code of the WDSim projects and the reference documentation of the WiFi Direct API. This documentation can be found on the Android developers website [1,2].

2. Check out how the service discovery works. Inspect the source code of the WDSim and of the SimpleChat app in detail in order to learn how the service discovery is being implemented by the simulator.

3. Implement the WDSim connect service. Once the devices receive the peer list from the console, they should be able to connect to another peer and setup a WiFi P2P group. The connect service of the WDSim should implement this functionality.

References

[1] android.net.wifi.p2p
http://developer.android.com/reference/android/net/wifi/p2p/package-summary.html

[2] Connecting with Wi-Fi Direct
http://developer.android.com/training/connect-devices-wirelessly/wifi-direct.html