Ad-hoc Social Networking for the Google Android Platform
Final Report

CSC 714 Real Time Systems
David Gurecki
dwgureck@ncsu.edu

http://www4.ncsu.edu/~dwgureck/csc714/

Summary

Configuration of the Android G1 phone in ad-hoc mode was achieved, and communications between 2 G1 phones was successfully tested. The resulting software application, adhocClient, was built by incorporating a script for kernel module configuration, some binaries, and modified code from the Android Wi-Fi Tethering open source project. A prototype social networking GUI was built on top of this base. The adhocClient can operate in either Ad-hoc mode or using a WiFi Access Point as configured via Settings. Ad-hoc mode includes a discovery mode, during which the application alternates between a DHCP Client and DHCP Server to allow for automatic negotiation of IP addresses with other Ad-hoc endpoints. Other clients are discovered using a UDP heartbeat message sent to the network broadcast address. The heartbeat is always being broadcasted, and as long as a heartbeat message is received from other clients the DHCP role is frozen. When heartbeats are no longer being received from any client, the application begins alternating DHCP modes again. When one or more clients are discovered, the various clients can exchange status updates, messages, and GPS coordinates from each other.

Since there was only limited access to a second G1 phone, a testing application was also developed which runs on any laptop with a WiFi radio.

Solution

The work to develop this application centered around several key tasks: adapting components from the open source Wi-Fi Tethering project to configure the radio in Ad-hoc mode, specifically to enhance the "tether" Linux script to act as a DHCP Server or Client; developing a UDP messaging protocol to discover other clients and transmit and receive text messages and GPS location; learning Android GUI fundamentals to develop various GUI screens containing lists, buttons, and text input fields; testing an Ad-hoc connection and UDP messages with two G1 phones; and writing a testing Java app that runs on a laptop. There was insufficient time to develop some of the fancier GUI capabilities such as transmitting multimedia content, or displaying GPS location using maps or a compass. However, this can be seen as a proof of concept that could be built upon in the future.

The solution will be described by its components in the following sections:

Graphical User Interface

The GUI for AdhocClient contains the following screens:

Screen	**Description**
Setup	Uses Android's built-in preferences screen for various settings, including: username, WiFi mode (Ad-hoc or Access Point), WiFi SSID, channel, and power mode.
Incoming Events	Displays a history of incoming messages and status updates from other clients, sorted in reverse timestamp order. Clicking on a message entry displays the message in the Message View screen.
Friend List	Displays the list of Friends ever discovered since the app was started. The list shows current status, including real-time GPS coordinates, and an indication of whether the Friend has stopped heart-beating (i.e. stale data). Clicking on a Friend brings up the Message View screen with an editable field for typing and sending a message to the Friend.

My Status	Allows the user to change their status, as well as start/stop the network control service (which also stops Ad-hoc mode and discovery of other devices).
Message View	Either displays messages received from Incoming Events list, or provides an interface for typing a new message to be sent to a Friend. When browsing received messages, a Prev and Next button allow the user to read other messages in the list.

When the application first loads, the Incoming Events screen is initially displayed. The MENU button is used to select from the first four screens. The Message View screen is only accessible by selecting a Friend from the Friend List (for sending messages), or selecting an Incoming Message from the Events list. The Back button is then used to return to one of the four screens.

Activities

Each interactive GUI screen in the Android framework is implemented as an Activity. This application has the following activities:

Activity	**Superclass**	**GUI Screen**
SetupActivity	PreferenceActivity	Setup
IncomingEventActivity	ListActivity	Incoming Events
MessageActivity	Activity	Message View
FriendListActivity	ListActivity	Friend List
MyStatusActivity	Activity	My Status

Other classes

In addition to GUI activities, there are other classes which implement functionality for this application.
The AdhocClient class implements the android Application superclass. This class is used by all the Activities for data storage of lists (messages, friends, etc.) and controlling the current state of the application. The storage behind the event list and friend list are ArrayList's of String objects. There is also an embedded class called FriendData for storing the name, IP address, last message time, and current status message from each Friend. FriendData also contains methods for building and parsing messages for the UDP messaging protocol. This consists of two types: *Heartbeat*, which is used to announce a client's presence, send current status message and GPS position, and *Message*, which is used to transmit a single text message. These messages consist of simple ASCII strings.

AdhocClient implements the UDP sending and receiving threads. The UDP Receiving thread is implemented by the *ClientReceiver* embedded class. This thread binds a UDP DatagramSocket to port 8888, and accepts incoming UDP messages from other clients. The UDP sending thread is implemented by the *ClientSender* embedded class. This thread's main purpose is to send Heartbeat messages to the broadcast address on the phone's current network. The current network can change depending on which state the application is in: either DHCP Server, DHCP Client, or AccessPoint. The current IP address (and thus broadcast address) is obtained in a slightly different manner depending on the current state. At this time, it is assumed that the broadcast address is always obtained by replacing the last of the IP address octet with 255. This assumption is always true when Ad-hoc mode is active and, while not always true on an Access Point network, it was sufficient for testing basic functionality.

The *NetworkControl* thread performs the important task of alternating between configuring the WiFi radio in DHCP Server and DHCP Client modes by invoking the netcontrol script. The DHCP Client mode runs for a maximum of 60 seconds if an IP address is obtained. If no IP address is obtained, it runs until the "dhcpcd" program returns after a timeout period, which appears to be 30 seconds. The DHCP Server mode runs for 60 seconds. If any heartbeat messages are received in these modes, the thread stops alternating and leaves the phone in the current mode until heartbeats cease to be received. This thread also responds to the Start Service / Stop Service buttons being pressed, driving transitions between the various states and starting up or shutting down services along the way.

The CoreTask contains code borrowed (with only slight modification) from the WiFi Tethering project. This is not an actual task or thread as the name implies, but a set of utilities for checking root permission, installing binaries into the appropriate directory, setting file permissions, running root commands, writing config files for dnsmasq (the

DHCP Server) and tiwlan.ini, executing commands (primarily the netcontrol script) in the underlying Linux system, and reading the output of these commands. These primitive operations could be reused in a similar manner as used by the Tethering application. Code which is not being used by this application has been commented out.

Several modifications were made to the Tethering project's *tether* shell script, which has been renamed to *netcontrol* for this application. The original script installed the WiFi kernel module, configured the tiwlan0 interface with a static IP address, started the dnsmasq program as a DHCP Server, configured IP forwarding, and executed iptables rules to enable NAT functionality with the 3G network. It could also stop all these services and return control of the WiFi radio to the Android OS. The netcontrol script reuses the script functionality for installing the kernel module and starting dnsmasq, however no iptables or IP forwarding is necessary for this application. There are 4 commands that can be executed with the netcontrol script:

- start_server – install kernel module, configure static IP address, and start dnsmasq DHCP server
- stop_server – stop dnsmasq, unconfigure and stop tiwlan0
- start_client – install kernel module, start tiwlan0 with no IP address, run dhcpcd client
- stop_client – unconfigure and stop tiwlan0

These commands are executed by the AdhocApplication's *NetworkControl* thread when Ad-hoc mode is in use. Initially, DHCP Client mode is started to see if any other DHCP Servers are available within the given Ad-hoc SSID network. The dhcpcd client times out after 30 seconds if no DHCP Offers are received. If this operation fails, or if it succeeds but no heartbeat messages are received, the *NetworkControl* thread runs stop_client and start_server commands to start the dnsmasq DHCP Server. This runs for another 60 seconds. The alternating between these modes only stops if heartbeat messages are received from another client.

AdhocClientTester

This program is included in a separate package and is intended to run on some laptop with a wireless adapter. The program is intended to mimic another "Friend" present in the same network as the G1 phone. When the program starts, it reads the friend's username as an argument to the program. It starts a UDP Server thread which binds to port 8888 and listens for messages. When the first heartbeat message is received, it analyzes the source IP address and converts it to a broadcast address by replacing the last octet with 255. It then sends continuous UDP heartbeat messages at 1-second intervals announcing its presence on the network. The status message and GPS coordinates are echoed back from the heartbeat message received from G1 phone. Thus, if the G1 phone changes its status message, this "friend" will appear to change its status message to the same thing. In addition, this tester application also sends a test message saying "Hello" every 30 seconds directly to the IP address of the G1 phone. The tester program outputs all messages received to stdout so the proper formatting can be verified.

Test Case Descriptions

Test 1: Laptop DHCP Server, G1 DHCP Client
Setup: Configure a laptop to run as an Ad-hoc network with the same SSID as the phone. Specify a static IP address, e.g. 192.168.3.1 with netmask 255.255.255.0. Disable Windows Firewall (very important!). Obtain a DHCP Server program, e.g. http://download.cnet.com/Tiny-DHCP-Server/3000-2085_4-10796649.html . Configure the DHCP Server to respond to the MAC address of the phone (can be seen after a Discover message is received).
Execution: Start Adhoc Client on the Phone, and AdhocClientTester on the laptop. The Phone should start in DHCP Client mode. Verify DHCP Connection is made from Laptop. The AdhocClientTester should display

```
Received packet from: 192.168.3.2, port: 44760, msg: Heartbeat:~david~Available~249.0, 249.0
```

On the G1 Phone, the Friend List should display the "Laptop" friend with the same status as the phone application has. The GPS coordinates should either be actual coordinates, or if GPS satellites are not in range, sample numbers steadily counting up by 1 per second. The Event List should show an entry that the Laptop friend has updated its status.

Every 30 seconds, the AdhocClientTester will send a message to the G1 phone. This message should appear in the Event List. Selecting the event should bring up a Message View GUI displaying the message text: Hello. The Prev and Next buttons will traverse through all messages received.

Test 2: Change status message
Setup: continue from last test case
Execution: Change the status message on the G1 phone. The AdhocClientTester will begin echoing back the new status message. Verify that the status is updated on the Friend screen and on the Incoming Events screen.

Test 3: Send a text message
Setup: continue from last test case
Execution: On the Friends List, click on the “Laptop” friend. This will bring up a Send Message GUI screen. Type a message and click “Send”. Verify that the message is received by the AdhocClientTester, such as the following.

```
Received packet from: 192.168.3.2, port: 42847, msg: Message:~Laptop~test message~
```

Test 4: Stale Friends
Setup: continue from last test case
Execution: Stop the AdhocClientTester on the Laptop. After 15 seconds of no heartbeats, verify that the Friend status changes to “Unknown [stale]” on the Friend List. Verify that the count of Active friends decrements by 1 on the My Status page. If the count of Active friends is now 0, verify that the program begins alternating between DHCP Client and DHCP Server on 60-second intervals. After a while, start the AdhocClientTester program again. Verify that the Active Friend count goes up by 1, and the status for the Laptop friend is now updated in real-time on the Friend List.

Test 5: Multiple Friends
Setup: continue from last test case. Start a second instance of AdhocClientTester on the laptop, with a different username than the first instance. Note that the second instance will be unable to bind a UDP Server because the first instance is already bound to the same UDP port.
Execution: Verify that 2 friends are shown on the Friend List, and the My Status screen shows 2 active friends. Change the status message on the My Status screen. Verify both friends echo back the status change. Verify messages are received from both friends. After a while, shut down the second instance of AdhocClientTester. After 15 seconds of shutting down a tester, verify that the Active Friend count decrements by 1.

Test 6: Test Stop Service / Start Service button
Setup: continue from last test case
Execution: On the My Status screen, click the Stop Service button. Verify that the status changes to “Stopped” within 30 seconds or less.

Test 7: Laptop DHCP Client, G1 DHCP Server
Setup: Stop the AdhocClientTester program on the Laptop. Click “Stop Service” on the My Status page of the phone. Configure the laptop to run as an Ad-hoc network with the same SSID as the phone. Specify the wireless interface to act as a DHCP Client. Click “Start Service” on the Phone. Verify the phone initially tries to act as a DHCP Client, but gives up after 30 seconds approx. Verify the phone switches to DHCP Server mode. Verify the laptop receives a DHCP address. Now, start the AdhocTesterClient. (This must be done after the IP address is obtained so that the UDP Server binds to the correct IP address). Verify that the Laptop Friend appears on the G1 phone.

Repeat Tests 2-6 in this mode.

Test 8: Laptop and G1 phone in Access Point mode.
Setup: Stop the AdhocClientTester program and click “Stop Service” on the phone. Bring up the Setup screen and change the WiFi mode to “Access Point”. Configure the Laptop to use the same Access point. Ensure that the Access point is using a network with 255.255.255.0 network mask. Go back to the My Status screen, and click “Start Service”. Verify that the status changes to Access Point. Start the AdhocClientTester on the laptop. Verify that the Laptop Friend appears on the G1 phone.

Repeat Tests 2-6 in this mode.

Test 9: Two G1 phones in Ad-hoc mode
Setup: Install AdhocClient application on 2 G1 phones. Start the first phone, and allow the service to run for enough time to transition from DHCP Client mode into DHCP Server mode. Start the second phone. Verify that both phones show “1 Active Friend” within about 30 seconds. Continue testing by sending messages, changing status, and verifying the Incoming Events and Friend List pages. Press “Stop Service” on the phone running a DHCP Server. After 15 seconds, the other phone will switch to DHCP Server mode. Press “Start Service” on the first phone, and allow it to become a DHCP Client. Verify the connection is re-established.

Open Issues, Limitations, Future Improvements

1. The assumption was made that the 3G data network is off during operation. It is uncertain how the program will act if the 3G data network is active. Since we do not enable IP forwarding, and are generally using private address ranges like 192.168.x.y, it may work fine.
2. The assumption was made that the network mask for Access Point mode is always 255.255.255.0. This is not always true, and ideally could be made into a setting or obtained some other way.
3. When switching back and forth between DHCP Client and DHCP Server, the script currently shuts down the entire tiwlan0 interface, reloads the kernel module, and restarts the tiwlan0 interface again. This adds some time delay that could be optimized out by keeping the interface up over the transition.
4. One major limitation is that there is no clear way to know when two WiFi radios have made a connection in Ad-hoc mode. The native WiFi manager available through the Android platform is disabled, so it cannot report when the radio is connected or what SSID networks are available. This leads to the requirement that all phones must use the same SSID, and limits the responsiveness of the application.
5. It sometimes takes time for 2 Adhoc WiFi radios to establish a connection. If the phone is acting as a DHCP Client, there is no way to know with certainty whether a WiFi connection exists before starting the “dhcpcd” client program to send Discover requests. Thus, the dhcpcd program may send Discover requests while the radio is disconnected, and may not resend them after the radio becomes connected. If the dhcpcd program times out, we switch back to DHCP Server mode. A hard delay of 5 seconds is used to increase the odds of success. Ideally, it would be better to research ways to determine if the WiFi connection exists before starting this client.
6.The application occasionally experiences some Java exceptions if a WiFi operation fails for some reason. While it is reasonably robust, it is not very obvious when something serious has happened (such as a thread exiting) and there is no current way to stop & restart the application, aside from using the “adb” tool or restarting the phone.
7. A Friend is uniquely identified in the Friend List by Name + IP Address. If a connection drops and resumes later, or DHCP roles are reversed between 2 phones, it is possible that the same friend now uses a different IP address. Yet, it will appear as a new friend in the Friend List. This could be fixed by some logic changes.
8. It is not clear how useful the DHCP handshake mechanism would be if one tried to scale up to many phones. It should be possible to communicate using an Ad-hoc network with many phones. However, one phone must act as a DHCP Server for the group. If two phones initially establish their roles, and other phones later enter as DHCP Clients, they can communicate. But if there were two pairs of phones, each paired to each other with Server/Client roles, they would not be able to communicate. Since the phone alternates between modes, it is likely that more than one DHCP Server would be assigned in a group of many phones. This shows the limitation of such an algorithm, and the only improvement would be using static IP addresses.
9. Randomized timeouts for the different DHCP states was not implemented. As a result, it is possible for 2 phones to become “synchronized” and never connect without manual intervention. This could be improved by randomizing the time spent as a Server, however it still may take some time for the phones to establish a connection.
10. The usefulness of this application is limited to a “proof of concept”. It could be expanded to include other multimedia content such as sending pictures, or showing a compass to find other friends given their GPS locatoin. All this could be built on the current base, the project as is just shows that it is possible to send the required information over an Ad-hoc connection.

Conclusion

Ad-hoc networking, while not available through the native Android platform, is possible using open source tools which manipulate the underlying Linux operating system. The feasibility of an Android application which can manage Ad-hoc mode and send simple messages between G1 phones has been demonstrated.