Android Walkie Talkie - Final Project Report

ECE/CSC 575, Dr. Sichitiu
North Carolina State University
April 21, 2013

Authors - Team A3:

Andrew Davis (*ajdavis7*)
Travis Folsom (*twfolsom*)
Deenan Ravindra (*dravind2*)
Dustin Swisher (*dmswishe*)

Table of Contents

Abstract

New Android platform devices offer significant computing power which allows for easily accessible audio processing capabilities. The problem presented is that of creating a “walkie-talkie” system using the Android platform over WiFi in infrastructure mode. The system should allow for the specification of privileged conversation groups. A design for this system is presented and its implementation is realized. The implementation consists of four key screens which corresponding to the key features of the application: setup, joining, hosting, and conversing. Explanations of application behavior such as audio delay and hardware compatibility are then given. Finally, possibilities of further development, especially in the realm of security, are presented.

Introduction

The goal of the Android Walkie Talkie project was to have a group of users that are somehow specified to talk to each other in the following scenario: when one user activates its microphone, by pressing the talk button, the voice from that user is digitized, packetized and sent to the other users in the connection. While one user's microphone is activated, all other microphones should be disabled during this time.

Push-to-talk functionality is important to users who would like the ability to talk at a distance when they do not have line of sight. This includes scenarios (butler/maid, mechanic, technician, soldier, guard, etc) where a worker needs to communicate with another party that is unable to be co-located (for reasons of safety, convenience, or efficiency).

The contribution of this project is to explore the development of a push-to-talk application for the Android platform which could be conveniently installed on any of a phone, tablet, or computer (larger tablet).

Background

Needless to say, there are already innumerable devices that accomplish the goal of push-to-talk functionality. Motorola, Cobra, Uniden, Midland, and Lenmar are popular examples of companies who develop devices with this functionality exclusively. More specific to the problem presented (push-to-talk *on Android*), there are several projects (and therefore apps) which provide push-to-talk functionality. The first of these was **pttdroid** (https://code.google.com/p/pttdroid/), an open source project which utilizes the Speex codec in any of unicast, multicast, or broadcast modes. Another was the **sipdroid** (https://code.google.com/p/sipdroid/) project, which allows for VoIP calls in general. Additionally, a quick search on the Google Play store reveals several apps offering "walkie-talkie" functionality: Voxer, TiKL, AZT, touchPTT, Sprint Direct Connect Now, Coco Voice, and Zello, among others.

As has been illustrated, the "walkie talkie" Android market is rather saturated. This project aims to bring a fresh perspective to the problem by using modern APIs and a master/slave architecture which could allow for more flexibility than offered by others like **pttdroid**, it offers configuration for explicit use of only one of unicast, multicast, or broadcast.

Pre-Analysis

Formulation of a solution began with a consideration of the networking capabilities required. It was given that the solution used voice data over WiFi, so the first thought was to use datagrams in a uni/multi/broadcast model. Unicast could be used for peer-to-peer conversations, multicast

could be used for group conversations, and broadcast could be used for discovery services. Using previous networking and programming experience, UDP sockets were determined to be a perfect fit for the application's "backbone."

Before any code was written, however, the "vision" of the program's operation was created in mock-ups. Using the mock-ups as a guideline, the necessary networking commands were drafted to form a sort of protocol specification. The initial protocol specifications can be found as-is in Appendix A. The mock-ups, however, can be found below as Figure 1.

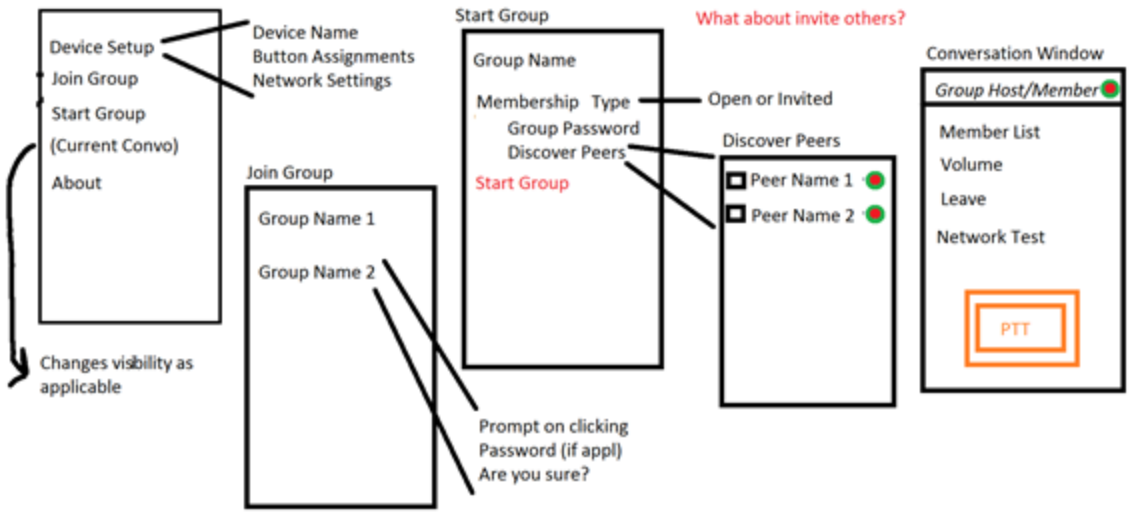

Figure 1: Mock-ups of the program's operation.

Armed with the rough protocol specification and mock-ups, application architecture began. The architecture was to take into consideration best practices for object oriented development such as MVC, inversion of control, dependency injection, and the like. Networking operations would be abstracted to avoid boilerplate code, and interfaces/abstract messages would be leveraged to make implementation of the core state machine and message parsing understandable and flexible.

Once the networking core was established, work could begin on determining the best way of handling group membership and management. This would then be followed by the the key item of the app: voice data. The method of encoding could be implemented using third party native libraries, or leverage APIs already present in the Android SDK. This choice would be dependent on choice of hardware compatibility.

After all the design and development choices are made, it would come time to test and debug the application; starting first with one device using network tools to verify messaging attempts. Two devices would then be used to test group management, and ultimately voice data. Three or more devices would then be used to test robustness of the application. All these configurations

would be in addition to testing with the devices in different physical locations to ensure consistent performance.

The expected output is an application that is able to allow two people, each with an Android device, to communicate with one another verbally through the action of holding down a button while speaking and having that audio signal sent to the interested parties.

Results

The final product is a very simple app whose operation involves four screens:

1) Main Menu - The screen that appears when one first runs the application. It provides an option for specifying your device name, as well as buttons to initiate joining or starting a group.

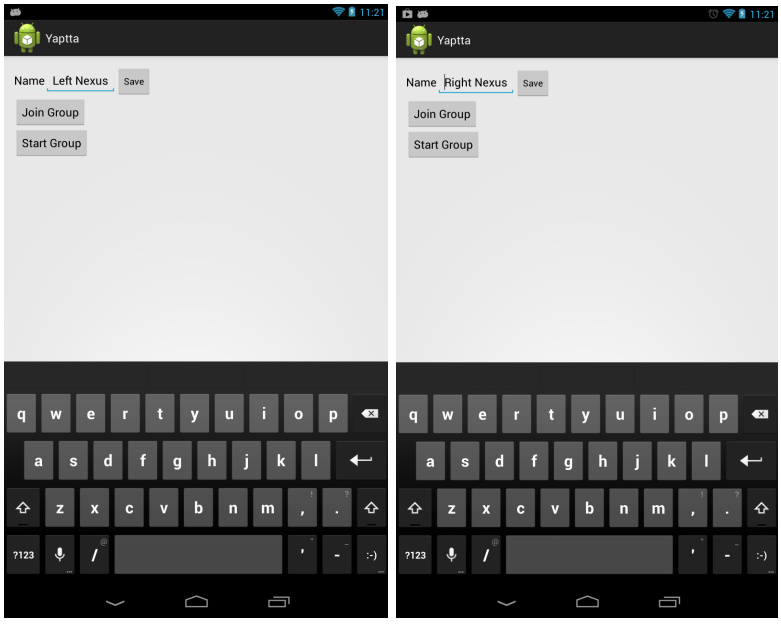

Figure 2: Main Menu

2) Join Group - This screen appears after one selects to join a group from the main menu. It polls the network to see what groups are currently active in the subnet. Choosing a group either connects right away or asks for a password, depending on the group preferences.

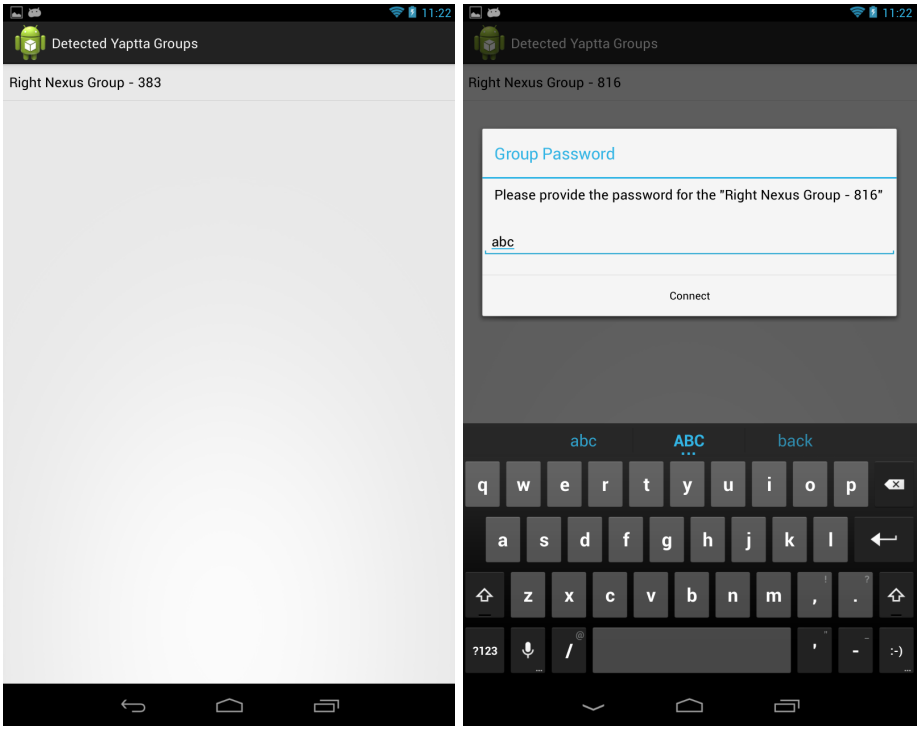

Figure 3: Join Group

3) Start Group - This screen appears after one selects to start a group from the main menu. It provides options for the group name, as well as the privacy settings. Selecting the Start Group button once more initiates the push-to-talk conversation.

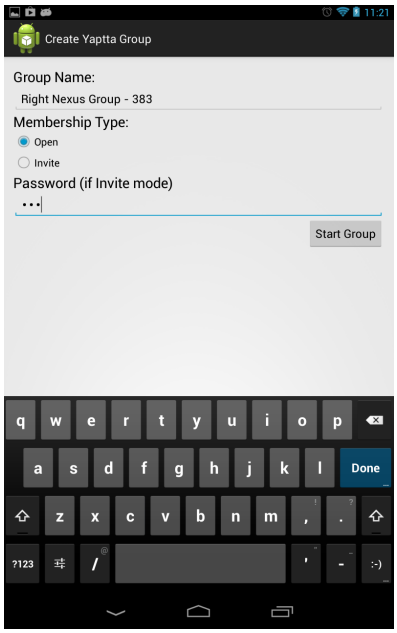

Figure 4: Start Group

4) Conversation Window - This screen appears after successfully joining a push-to-talk conversation or starting one. To end the conversation, the user can either press the Back or Leave buttons. To transmit voice, the user should press and hold the Push To Talk button.

Figure 5: Conversation Window

The overall operation of the application is straightforward. The first time running the application, one should set the device name to something unique (as it defaults to “Yaptta Device”). On the first device (which will act as the master), the user selects Start Group. This is followed by configuring the group settings as desired (though for initial operation, the defaults are sufficient). This will then bring the user to the conversation window, where it is now ready to host. The second device (and subsequent devices) selects Join Group, where it finds the master group. The user then selects the group, prompting for a password if applicable. Upon successful connection, it presents the conversation window, where the push to talk functionality will provide the desired “walkie talkie” functionality.

Post-Analysis

Unfortunately, the final product seems to have fallen short of the original vision (but does meet the requirements). While it does allow for the creation and membership of arbitrary groups in both an open or password-protected manner, it does not allow direct communication between two users (such as UDP unicast). Of course, one may set up a group that has only the two users in the group, but the communications are still multicast on the network.

For the audio encoding/decoding mechanism. in lieu of creating a custom mechanism and choosing an available third-party codec, the built-in RTP (real-time protocol) functionality of the Android platform was used (android.net.rtp package). Indeed, the Android Javadoc mentions that “apps that provide VOIP, push-to-talk, conferencing, and audio streaming can use these APIs to initiate sessions and transmit or receive data streams over any available network.” This allowed the team to focus more on the group membership mechanism instead of the channel locking mechanism, at the expense of some flexibility in when audio is sent, but at the benefit of not having to worry about buffering and sequencing.

Using Wireshark as an investigation and debugging tool, the custom-developed multicast protocol is not “chatty” at all; the datagrams sent and received are minimal and necessary. The content of the packet, however, seems to have considerable overhead because of the Java serialization methods leveraged for strongly typing the messages in code. Furthermore, the TM-G5240 (802.11g) wireless router used during testing does not bog down the LAN with unnecessary traffic while voice communications are occurring. The team did not have access to a machine which properly implements WiFi “promiscuous mode,” so the actual traffic could not be determined.

Overall voice quality was very good, and while there was a noticeable delay of approximately 0.6 seconds, it is expected due to buffering. Additionally, running a continuous ping between two devices on the tested network revealed a high amount of jitter, with a minimum ping of 2 milliseconds and maximum ping of 117 milliseconds, with an average of 50 milliseconds at zero loss. This latency, coupled with the audio processing occurring on both ends of the conversation, are proportional to the delay encountered. Traditional walkie-talkies possess dedicated circuitry (modulators, demodulators, ASICs) which allows for low-latency processing of audio, so it is not surprising that such devices exhibit seemingly instantaneous transmission of voice.

The most difficult aspect of this project, however, was coping with the vendor support of multicast on the portable devices. While there are many anecdotal reports of multicast “not working” around the Internet, at least one official issue has been logged (https://code.google.com/p/android/issues/detail?id=51195) against the problem. The Android Javadoc (http://developer.android.com/reference/android/net/wifi/WifiManager.MulticastLock.html) mentions that multicast is soft-disabled to avoid battery drain, but that does not account for why it has been completely disabled by vendors. Indeed, the team possessed such a device (HTC Sensation 4G) and tried flashing the ROM between three different versions of Android (CyanogenMod 9, CyanogenMod 10, and HTC ICS Stock). The result was that if the ROM supported multicast, it didn’t have the appropriate codecs for RTP (and vice versa). All told, trouble with multicast resulted in the team having to procure additional devices with the capability to use multicast, update system software, as well as re-configure routers (for options like enabling IGMP, disabling snooping, etc). More details on the tested hardware can be found in the hardware documentation accompanying this report.

Conclusion

There are a couple of considerations should development continue further. The first of these would be to create a more secure authentication mechanism, since one notable shortcoming of the application is the lack of security. Passwords are transmitted in plaintext, and authentication for a group could easily be spoofed. Additional work could be done in the areas of communication reliability, as it is not uncommon for UDP datagrams to get dropped. This is important when discovering or joining groups, as a dropped packet may place the application in limbo as it waits for a response. Too, more customization of the network and audio parameters such as the codec used would be nice. Finally, another consideration would be to the application more “friendly” to use in general, such as more feedback to the user as well as better graphics and layout.

Overall, a solution to the Android-based “walkie-talkie” problem was proposed, designed, implemented, and tested. The application is able to broker group membership, in either an authenticated or unauthenticated manner, whereupon only group members are privy to the group’s communications. There was some difficulty encountered with device vendor multicast support which could only be surmounted by using different hardware or upgrading system software. However, past these obstacles, the application functions as an acceptable “walkie-talkie” alternative with very good voice quality.

Appendix A

Initial Protocol Specification Documentation

Actions and scenarios:

- ~~When the app is installed, generate a GUID for the device~~ Use MAC address
- When the device name is changed, broadcast the name change
- When IP address is changed, broadcast IP address change
- When Join Group is pressed, broadcast group name discovery, and wait for responses
- When Discover Peers is pressed, broadcast peer name discovery, and wait for responses
- When Start Group is pressed, send invites to all selected peers
- When invite is accepted, update group membership list
- When PTT button is pressed, broadcast to get lock on channel
- While PTT button is pressed, stream voice data
- When PTT button is released, release lock on channel
- When Leave Group is clicked, broadcast to leave group

Consideration: membership coherence protocol

Resulting broadcast/multicast signals:

- REQUEST_GROUPS (group discovery)
    - Replies to requestor with group currently hosted, if applicable
- REQUEST_PEERS (peer discovery)
    - Replies to requestor with device name
- UPDATE_PEER (either pushed, or respond to request_peers)
    - Broadcasts updated device name
- VOICE_DATA (multicast)
    - Provides group, codec info, and codec data
- GROUP_LIST
- REQUEST_CHANNEL_LOCK
    - brokered by group host
- RELEASE_CHANNEL_LOCK
    - brokered by group host
- PING_REQUEST (number crunching on peer discovery)

Resulting targeted (single recipient) signals:

- JOIN_GROUP
    - Sent from invitee to group host
- INVITE

    - Provides group name to initiate a group join
- BOOT
    - Forces user disconnect
- PREEMPT (nice to have functionality)