COMMUNITY AWARENESS SYSTEM FOR ANDROID DEVICES

Final Report

Group : DEC1618 (dec1618@iastate.edu)

Clients : Dr Daji Qiao, Dr. George Amariucai

Advisors: Dr. Daji Qiao, Dr. George Amariucai

Team Members/Role:

Jason Wong: Team Leader

Erik Fetter: Communication Leader

Matt Gerst: Team Webmaster

Shikhar Vats: Key Concept Holder

Brad Anson: Key Concept Holder

Adit Kushare: Key Concept Holder

Contents

1 Project Design

1.1 System Specifications

1.1.1 Non-functional

- Periodic network updates should be quick without a major battery drain
- Network should be resistant to device loss by using the mesh topology
- Users should know in a reasonable amount of time if a device has left the mesh
- Remote data collection should be received in a reasonable amount of time (5-10 seconds)
- Collection of data should occur in the background, without UI interaction
- Android codebase should largely be modular and good/best practice (aside from the necessary rooting of the devices)
- The application does not need to scale very large, but it should be able to handle 10-20 devices on the network.

1.1.2 Functional

- A user should be able to find out who else is on the mesh network
- Users should be able to request information about other users on the network
- Network module should provide as many metrics, such as RSSI, if possible
- Users should be able to see a network topology, possibly organized to represent relative positions, which shows how many hops to another user
- The Application should expose both a network API usable by other applications and a sensor/inference API for the inference engine

1.2 Design Architecture

We divided the architecture of our application into 3 different layers. As follows:

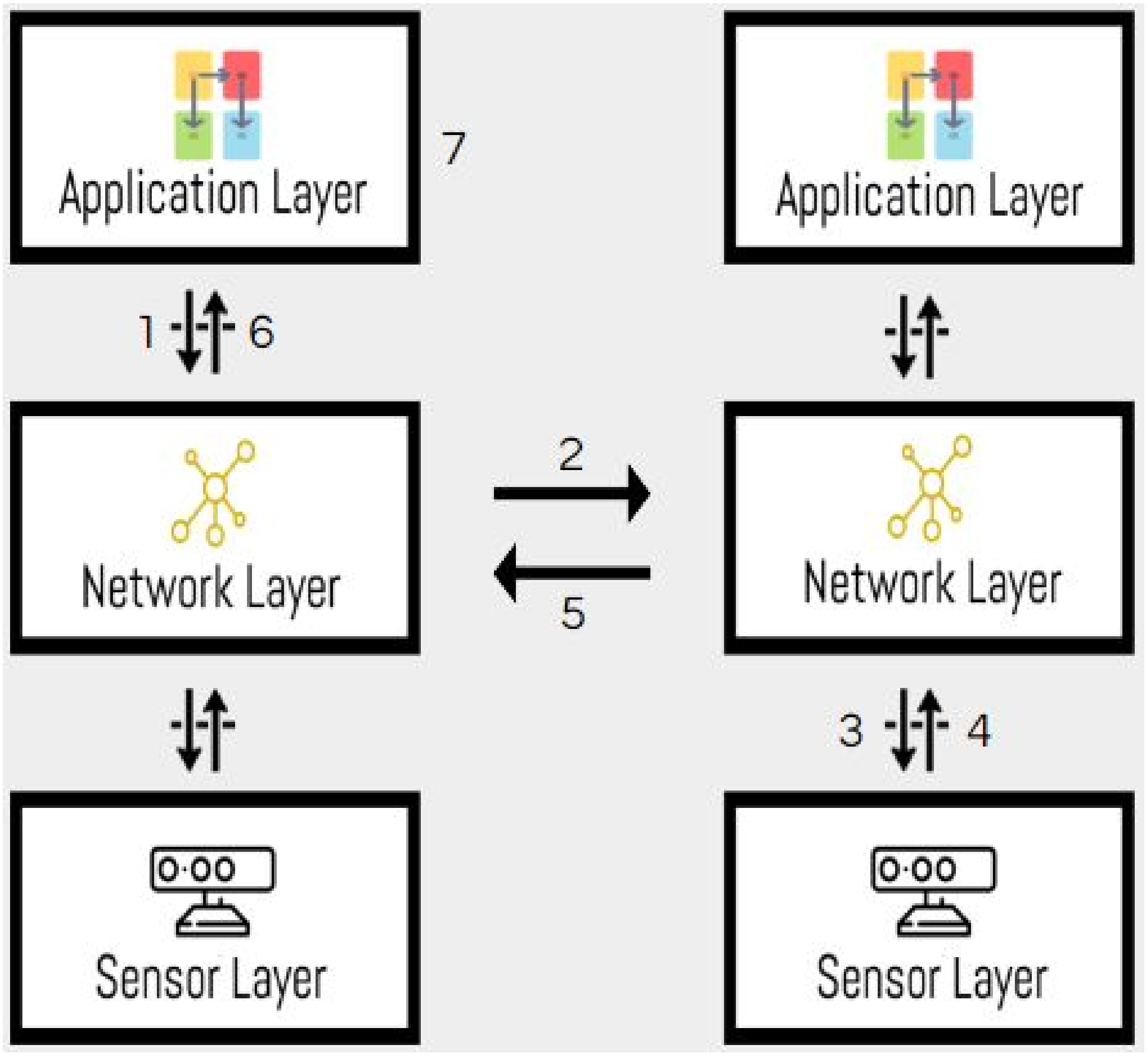


Figure: Flow of information between two devices on the network

1. The **Application Layer** from Device A queries the network layer to access data from Device B.
2 The **Network Layer** from Device A sends the query to Device B
3 The **Network Layer** from Device B receives the query, and queries the **Sensor Layer** in turn for the information.
4 The **Sensor Layer** receives the query, collects the sensor data, and replies to the **Network Layer** with the analyzed result
5 The **Network Layer** receives the information from the sensor layer, and transmits it to the **Network Layer** on Device A
6 The **Network Layer** on Device A receives the information from Device B and sends it to the **Application Layer**

7 The received information is displayed on the Device A

This project can be divided into three distinct interconnected system layers:

- Sensor Layer: The sensor layer essentially has functions to collect the raw data from the various sensors on the Android device. This information from various sensors include, but are not limited to, accelerometer, gyroscope, barometric sensors, is analyzed to generate a confidence number which would be used for communication with the other android devices. The results from the sensor layer are transmitted to the network layer which then uses the information to communicate with the other devices on the network.
- Network Layer: This layer encapsulates the core functionality of the project. The Android devices in our system would be interconnected using wireless ad-hoc network mesh topology. The network layer is responsible for establishing and maintaining a connection between the Android devices in the system. The higher level application layer queries information before transmitting the information, the network layer will query the sensor layer, which replies with the analyzed sensor data (see Sensor Layer). The network layer on the receiving device receives the information, and then transmits it to the higher application layer.
- Higher Level Application: This layer acts as the UI layer for the application. The flow of queries and information starts after the higher level application layer queries the network layer for information from any other device in the mesh network system.

2 Implementation Details

2.1 Application Layer

The application layer uses HTML/CSS and JavaScript to create the visualization of the network. This is made possible through Android's WebViews. The JavaScript graphics library we used is called CytoScape.js. It provides the ability to draw node-edge graphs. We constructed the graph by using Received Signal Strength Indicator (RSSI) values obtained from the networking layer. The RSSI values are used as the edge lengths in the visualization graph.

2.2 Networking Layer

We created a networking API based on a design called The Serval Project. The API provides two main features: sending commands to other devices and sending files. For file transfers we build on top of Serval's Mesh Stream Protocol (MSP), a connection-oriented protocol much like Transmission Control Protocol (TCP), to send files to individual devices on the mesh. For commands we have implemented a custom protocol (the Commands Protocol) which extends Serval's Mesh Datagram Protocol (MDP), a best-effort protocol much like User Datagram Protocol (UDP).

When a device receives a command, it uses Android's broadcast system to pass the data to any application that listens for it. This includes the fact that a file transfer is occurring, but not the data in the file itself. Many of the basic commands do not place restrictions on what the payload may contain, and expect that the receiving side be able to interpret the payload however it wants.

Obtaining RSSI values is also a responsibility of the networking layer. We created a **`RSSIService`** to periodically gather a list of all nearby devices and their associated RSSI value. We are able to access Bluetooth RSSI values through Android's Bluetooth library. Each device's Bluetooth name is set to the user's Serval Subscriber ID (SID) which is how we are able to distinguish the different devices.

2.2.1 Commands Protocol

The Commands Protocol defines the following command types: **`MESH_REQ`**, **`MESH_RESP`**, **`MESH_START`**, **`MESH_END`**, **`MESH_STATUS`**, **`MESH_SENSOR`**, and **`MESH_ERROR`**.

`MESH_REQ`: This command represents a generic request. It takes an arbitrary payload and expects the receiving end to differentiate between requests.

`MESH_RESP`: This command represents a generic response, usually after receiving a **`MESH_REQ`**. Like the request, it expects the receiving end to handle the interpretation of the payload.

`MESH_START`: This command indicates that a long-running process is about to, or has already started (i.e. File Transfers). The payload of this command is expected to be in the format of a Command (an Intent-like object). The command includes an action that is used to differentiate messages from different system. A special action (**`MESH_TRANSFER`**) indicates a file transfer, which is handled internally by the network API.

`MESH_END`: Indicates that a long-running process started in a **`MESH_START`** has finished. The payload for this command is simply the same Command object sent by the **`MESH_START`**.

`MESH_STATUS`: Used to send status updates on a long-running process. Contains a simple payload of an arbitrary string (called an action) to differentiate between different processes, and float representing the progress of the process.

`MESH_ERROR`: Indicates there was an error while handling a command. Includes an error message as well as an optional Command object that can contain additional data about the error.

2.3 Sensor Layer

2.3.1 Miscellaneous Sensors:

Periodically, all sensor data is recorded to a SQLite database. When a sensor request is received from the networking layer, the requested data is retrieved from the SQLite database and sent over the network.

2.3.2 Audio

There are three facets to the audio recording: on demand recording, periodic background recording, and searchable requests. All of the audio tasks are accomplished by using two services that are started as soon as the app is started. The **`MediaRequestService`** handles the processing of the incoming requests and the **`BackgroundAudioService`** handles the periodic background collection of audio.

For the periodic recording functionality in the application, all devices utilize the **`BackgroundAudioService`** to continuously recording audio. This service is based on the android **`JobService`** that was introduced in API 23 and that allows customization to run only when defined conditions are met. For example, to conserve battery you can specify this background service to only run when the battery is over the 15% level. Currently the application records 6 minute audio clips every 6 minutes and records 5 hours of total audio. With so many files, we intentionally encoded the audio files in a way such that they do not occupy much space on the device. To solve the problems of file management, we found it simplest to set the name of every recorded audio file as the timestamp and date of when the recording is finished. Once files are saved and encoded, we save a list of the names to a local database. One of the most significant challenges of the audio portion of our project was performing audio collection in the background even when the device was sleeping and/or locked.

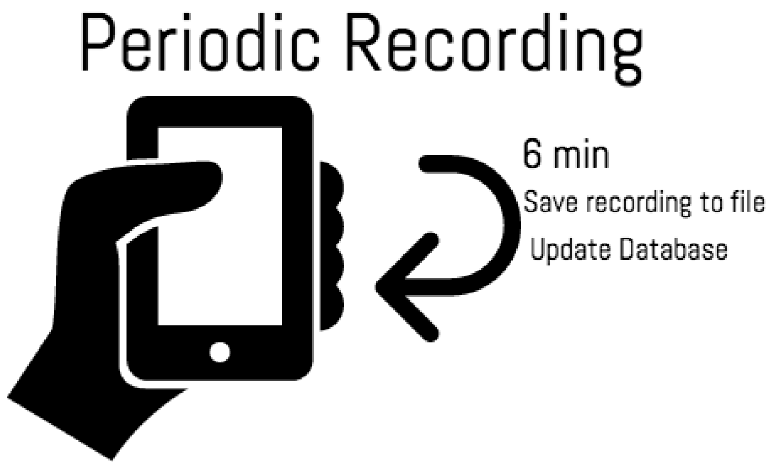


Figure: Periodic Recording

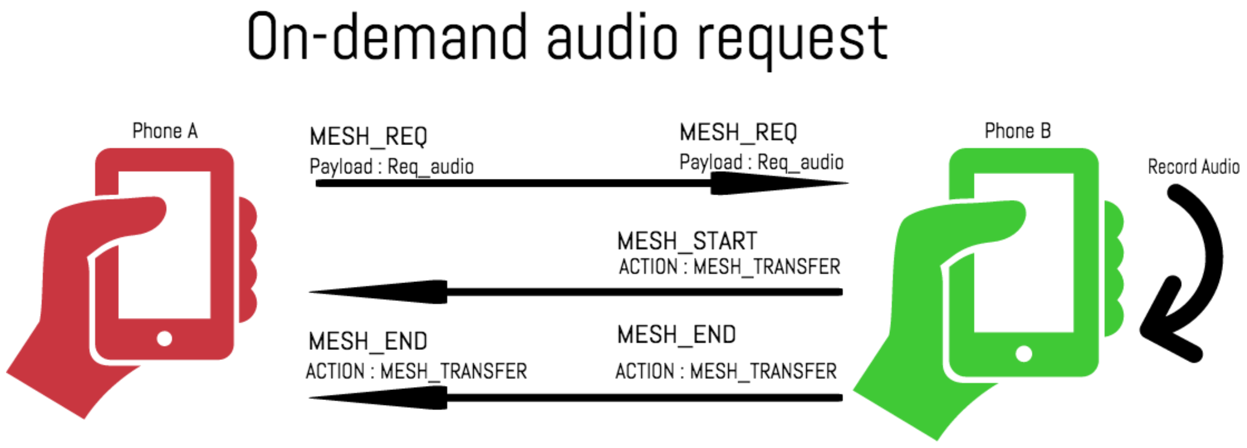


Figure: On-Demand Audio Request

On demand audio makes a request to the target device and creates and sends a fresh recording from the time of the request. The file is sent via MSP. A searchable request works in the same way as an on demand request but without recording fresh audio. Initially, Phone A sends a request to the target device with a timestamp as the payload of the request. The target device then searches this timestamp in its database and finds a file that is closest to the specified time. This file is then sent back to the requesting device using MSP. The flow of the commands used to accomplish is shown in the images above.

2.3.3 Video

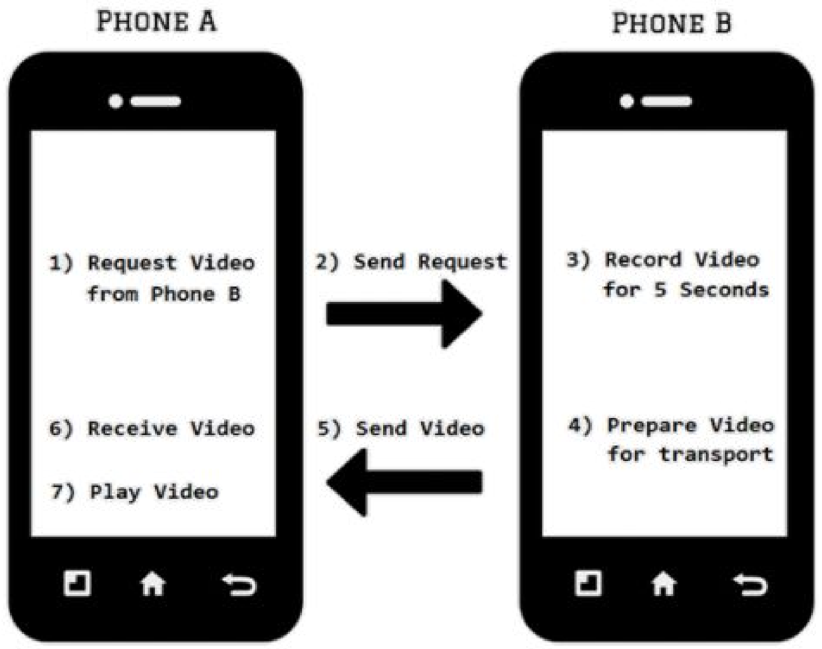
At first glance, Android doesn't support camera use that allows recording while the device is not displaying the video. This is to prevent malicious activity, spy apps, or embarrassing accidents. However, by making use of different parts of the framework and carefully supplying Android what it needs, we are able to achieve this with relative ease.

When a **`MESH_START`** command is received over the network, a service named **`MediaRequestService`** handles that request and launches a separate service named **`VideoRecordingService`**. It's inside **`VideoRecordingService`** where Android norms must be circumvented in order to accomplish the desired design of a remote device being able to capture video in any state.

In order for recording to occur in native Android, there must be a legitimate place to pipe the preview (the raw camera output) to. Throughout the process, we found a few different ways to accomplish this:

1. Display our video in a 1 x 1 pixel in the corner of the screen, utilizing the system's **`WindowManager`** and a **`SurfaceView`**.
2. Create a **`Surface`** from the **`VideoRecordingService`** itself; while remembering that services in Android do not have UI elements. Thus, the display was essentially being displayed in a background thread.
3. Dump the video stream to a non-existent OpenGL ES texture. Rather than sending the raw camera output stream to an existing Surface, it can be sent to an unestablished **`SurfaceTexture`**. **`SurfaceTexture`** objects are generally pre-defined **`View`** holders built for complex video streams.

It was the third option above that is the implementation that we went forward with in our final application. It required the least code and had virtually no performance impact. Since there is no buffer to store the preview whatsoever, it seemed to be the most efficient solution. For a clear picture of the whole process, see below.


PHONE A
PHONE B
1) Request Video from Phone B
2) Send Request
3) Record Video for 5 Seconds
4) Prepare Video for transport
5) Send Video
6) Receive Video
7) Play Video

Step 3 in Detail

1. Create a new MediaRecorder
2. Give it a profile (1080p)
3. Create a SurfaceTexture with an unidentified OpenGL ES Texture
4. On the MediaRecorder, call setPreviewDisplay(yourFakeSurface)
5. Run the MediaRecorder in a timed thread for 5 seconds.
6. Encode video file as MP4
7. Release MediaRecorder for system use

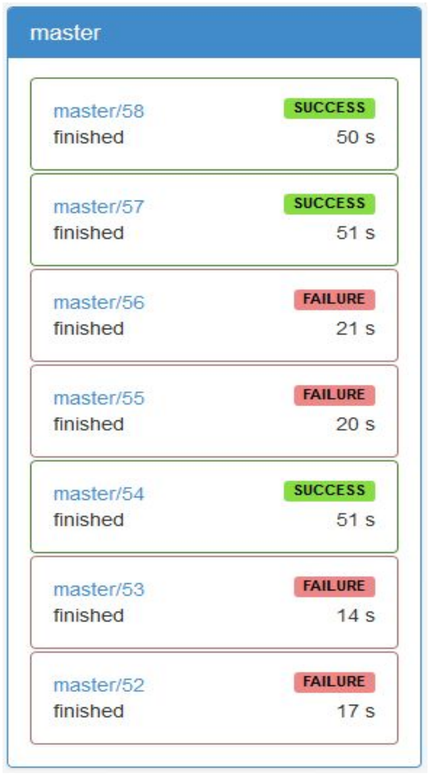
3 Testing

3.1 Theoretical Proof

File transfers using the MSP protocol guarantee data delivery; this is a characteristic of MSP and its connection-oriented design. As long as devices remain within communication distance the transfer should be able to complete.

3.2 Continuous Integration and Testing

As we were developing our application, we followed an Agile-like development process. For example, as we were developing and committing code, every GIT commit automatically triggered a build bot to build our application and test whether the build was successful.

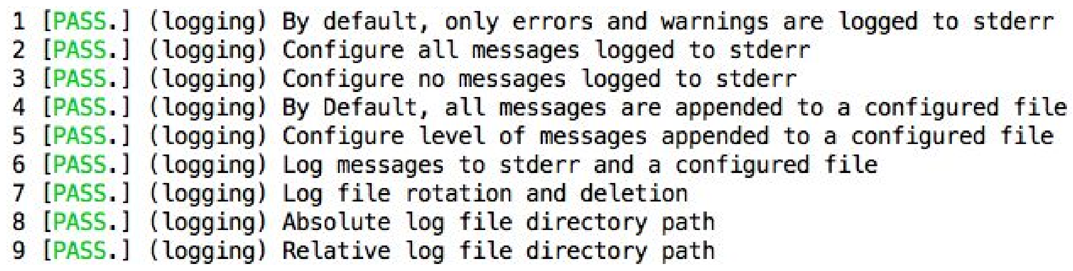


master

master/58 finished	SUCCESS 50 s
master/57 finished	SUCCESS 51 s
master/56 finished	FAILURE 21 s
master/55 finished	FAILURE 20 s
master/54 finished	SUCCESS 51 s
master/53 finished	FAILURE 14 s
master/52 finished	FAILURE 17 s

Figure: Build Bot Dashboard

We are also able to run automated unit tests on our networking application to ensure our application works as expected. Overall we have 378 unit tests that can be run on the application. The issue is that the tests do take a while to run. Overall, it can take over 20 minutes to run all tests.



```
1 [PASS.] (logging) By default, only errors and warnings are logged to stderr
2 [PASS.] (logging) Configure all messages logged to stderr
3 [PASS.] (logging) Configure no messages logged to stderr
4 [PASS.] (logging) By Default, all messages are appended to a configured file
5 [PASS.] (logging) Configure level of messages appended to a configured file
6 [PASS.] (logging) Log messages to stderr and a configured file
7 [PASS.] (logging) Log file rotation and deletion
8 [PASS.] (logging) Absolute log file directory path
9 [PASS.] (logging) Relative log file directory path
```

Figure: Unit Testing Example

3.3 Field Testing

Beyond the theory and software tests, we also tested our application in the field. We were successfully able to connect about 6 devices to the network. We were limited by the amount of devices we have and are confident that the network can support many more devices. The network was approximated by our visualization using RSSI values, but we did see that RSSI values fluctuated even though the devices were not moving. This could be a result of signal interference. Additionally, we were able to send sensor requests and send sensor files across our network.

Multi-hop scenarios were also tested. We were able to successfully send data between devices that were not directly connected. The time delay of the file transfers increased, but overall the network functioned as it was intended.

We were able to test some greater levels of distance and verified that, while using the TP-LINK TL-MR3020 routers, we were able to achieve distances around 200 feet. This confirmed our suspicions that the routers were likely the most stable solution as Android devices cannot typically broadcast Wifi signals for this great of a distance.

Using PowerTutor, we were able to measure the power consumption of our application at around 21.5J/min. This is comparable to an app like Youtube which uses about

19J/min. We were extremely happy about this considering the amount of features and capabilities our application has.

We were able to test our final products transfer rate and found it to be 0.28 MegaBytes per second. This is not as fast as we would have liked but this translated to transferring audio files in less than 1 second and larger 1080p video in 70 seconds.

4 Standards

Given that our project contained multiple layers, there were several standards that we made use of our final implementation. Here are some of them:

1. 802.11 wireless standards and Bluetooth protocol. (Mobile Networking)
2. Android application development standards (Software Development)
3. Java standards, we decided on Java 7 (Software Development)
4. Program modularity and versatility (Software Development)
5. Good programming practices such as good commenting and structuring of the project (Software Development)
6. Clean code commiting using Git and branch management (Software Development)

Most of the standards above were more-or-less guiding principles rather than regularly referenced sources. We required knowledge of each of them in order to come up with a comprehensive and successful design but most of us had experiences with these areas before beginning the project.

Using these standards was significant noting that our project is a component of a larger system. To ease future use, we developed a API for other applications in the system to use. This put a non-trivial amount of emphasis on points 4 and 5 knowing that at some point it’s likely an independent team will be looking at our work and trying to incorporate it into a larger system. Making our work as clear and standardized as possible not only helped with troubleshooting, but also helped us to avoid common pitfalls that large software systems often come up against.

Within our project you can find many modules that impose clear boundaries based on their functionality such as networking, sensor collection etc. Recognizing the importance of modularity ensured that we didn’t conflagrate sections of code and functions of our program that had nothing to do with each other. We accomplished some of this by having strict package management and constant discussions of feature placement.

Recognizing the importance of version control systems, we chose to take time to agree on a set of standard git conventions. Keeping our commit history readable and free of unnecessary obfuscation made debugging and code research simple. We also benefitted from taking time to restring code commits to branches and utilizing merge requests to combine our sections.

Appendix I: Operation Manual

Setting up and Connecting to a Router

1. Plug in the router to a power source
2. Wait 2-3 minutes for the router to start up and initialize
3. Connect your device's Wifi to public.servalproject.org
4. Done!

Request a New Video Snippet to Be Sent Back to You Privately with Encryption

1. On the main screen, select the "Mesh Visualization" button
2. Select a node
3. Click on the "Video" button
4. On the prompt that comes up, click "MSP"
5. On the second prompt that comes up, click "New"
6. A progress notification will appear: it will take about 30 seconds to retrieve the video
7. Once the progress notification reaches 100%, the video will begin to play
8. If you want to view it again, click the notification

Request a New Video Snippet to Be Shared on the Entire Network

1. On the main screen, select the "Mesh Visualization" button
2. Select a node
3. Click on the "Video" button
4. On the prompt that comes up, click Rhizome
5. On the second prompt that comes up, click New

6. Messages will appear in the center of the screen giving you updates on the process
7. Once the video has been shared to every device on the network, it will begin to play on your screen

Replay the Last Shared Video Snippet by Device

1. On the main screen, select the “Mesh Visualization” button
2. Select a node
3. Click on the “Video” button
4. On the prompt that comes up, click “Rhizome”
5. On the second prompt that comes up, click “Old”
6. If the device has a recent clip, the video will begin to play

Request a New 30-Second Audio Snippet from Target Device

1. On the main screen, select the “Mesh Visualization” button
2. Select a node
3. Click on the “Audio” button
4. A new activity will pop up with a list of all available audio clips
5. Click the “Request” button to request a new audio
6. Wait approximately 40 seconds and the audio clip will play automatically
7. The transfer of the file should show up in the notification tray

Replay/Search an Old 6-Min Audio Snippet from Target Device

1. On the main screen, select the “Mesh Visualization” button
2. Select a node
3. Click on the “Audio” button
4. A new activity will pop up with a list of all available audio clips

5. You can either click on one of the items in the list OR rotate the clock to pick a time closest to the audio clip you desire
6. Once the time is chosen with the clock, click the search icon
7. Wait approximately 20 seconds and the audio clip will play automatically
8. The transfer of the file should show up in the notification tray

Request New Sensor Data

1. On the main screen, select the “Mesh Visualization” button
2. Select a node
3. Click on the “Sensor” button
4. A new activity will pop up with available sensors to be displayed
5. Hit the refresh button
6. Now select the sensor you want to see graphed

Review Old Sensor Data

1. On the main screen, select the “Sensors” button
2. From the dropdown on the top, select a device whose history you’d like to see
3. Select a sensor you want to see graphed

(Note: Only the most recent 500 points of data will appear on each graph, even though the device has stored as many as have been sent.)

Appendix II: Initial Designs

Design 1

Initially, we planned to implement a software only solution. In other words, the Android devices would use their wireless capabilities to communicate directly. To accommodate this, we had to root the Android device to gain admin privileges in order to install a custom operating system called Cyanogenmod. Theoretically, doing this would enable ad-hoc mesh networking on the Android devices. With the phones enabled to connect to a mesh network, we planned to use Serval Mesh.

The issues we found with this design are that even after rooting the devices, which voids a device's warranty, and installing a custom operating system, the network connectivity did not work consistently. Additionally, when it did work, the communication distance of the devices were extremely limited.

Design 2

In order to address the issues with design 1, we planned to integrate wireless routers in the mesh network. Basically, each phone would be paired wirelessly with its own router. This allowed us to offload the mesh networking to the wireless routers.
With the networking finally working, we tested out Serval Mesh's rhizome file sharing and Mesh MS. Our initial design involved utilizing these to meet our projects functional requirements. Rhizome would be used to transfer data between devices, and Mesh MS would be used to create sensor and command requests.

The issues we found with this design are that file sharing was extremely inefficient. Basically, when a file is shared it propagates to all devices rather than an intended receiver. Another issue is that we wanted texting to work in addition to our project's functional requirements, which overriding Mesh MS to do sensor requests would not allow us to do this. Thus, we felt we needed to create a custom protocol in order to keep the Mesh MS functionality for texting.

Current Design

Like design 1, we planned to integrate wireless routers in the mesh network. For file sharing, we used Serval Mesh's MSP protocol. For sensor request commands we used Serval Mesh's MDP protocol. This is our project's current design.

Appendix III: Other Considerations

Lessons Learned

The unique thing about senior design is that we are solving real world problems, and consequently don’t have solutions that are clearly lined out. In our case, creating an ad-hoc mesh network of Android devices was extremely difficult due to it not being natively supported by Android. Additionally, even if it was supported by Android, the hardware on the devices aren’t really designed to handle receiving and relaying data as part of a mesh network. In result, we learned that we had to off-load the mesh networking to routers that are better designed to operate in such an environment. Apart from the technical knowledge we gained during the project, our senior design project also gave us an opportunity to learn to work and collaborate as a team in solving a real world problem. We faced a lot of roadblocks, including at a time thinking that certain aspects of the project were not possible. But with dedicated teamwork and proper guidance from our advisors we overcame the challenges and were able to produce a polished product that would meet the industry standards.