Final Report

7 May. 2020

Team Lora

Version 1.0

Community Aware Networks and Information Systems Lab

Dr. Morgan Vigil-Hayes (Sponsor)

Scooter Nowak (Mentor)

Ryan Wallace; Benjamin Couey; Mohammed Alfouzan; Brandon Salter

Table of Contents

1. Introduction

As the world becomes increasingly reliant on the internet, providing ubiquitous connectivity also becomes vital. Current technology that connects devices over a large area relies on very expensive cell towers or satellites. Due to the cost, these technologies are rarely set up to service rural communities, cutting these people off from the information and opportunities provided by the internet. A new technology, Long Range Wide Area Networks (LoRaWAN), has the potential to change this by providing connectivity that is both far reaching and inexpensive.

Our client, Dr. Vigil-Hayes and her research lab CANIS, have been working with this technology for about a year. Their intention is to take advantage of LoRaWAN's long range in order to increase connectivity in rural areas and support mobile crowdsensing endeavors. In these cases, LoRaWAN will be able to provide the services of a cell tower or satellite connection at a fraction of the cost and power consumption.

Mobile crowdsensing will require mobile devices, such as smartphones, to be able to connect over LoRaWAN. Traditional network technologies, such as WiFi and broadband transmissions, differ greatly from the underlying technology of LoRaWAN. Thus, there is presently no generic framework that would allow a smartphone or similar device to transmit messages over LoRaWAN, making it impossible to interact with any web applications. This project will make it possible for Android phones and, potentially, other devices to communicate over LoRaWAN.

To achieve this, we will be creating a framework for mobile developers that abstracts the process of transmitting a message over LoRaWAN. This framework will comprise a library for Android development and a proxy server. The library will encode messages and send them to the LoRaWAN network. These messages will then be received by the proxy server which will decode them and forward them to their intended destination. A configuration service will run on the proxy server which allows developers to define the encoding table used by both the library and server.

2. Process Overview

2.1 Team Roles and Standards

At the beginning of the Capstone year, our team assigned roles to each team member preliminarily, which have evolved and been reassigned as our individual strengths became more clear. The roles assigned were:

> **Ryan Wallace** - Team Leader and Customer Communicator: Ryan is responsible for coordinating tasks, monitoring task progress, running meetings, and resolving conflicts that arise. He is also responsible for maintaining communication with the team's sponsor Dr. Vigil-Hayes, the team's mentor 'Scooter' Nowak, and Dr. Doerry.
>
> **Benjamin Couey** - Architect and Recorder: Benjamin is responsible for maintaining documentation on project architecture and ensuring that implementation follows with architecture. He is also responsible for taking notes at all meetings.
>
> **Mohammed Alfouzan** - Release Manager: Mohammed is responsible for coordinating project branching, handling merge conflicts, and curating commit logs. He is also responsible for overseeing the issue board.
>
> **Brandon Salter** - Asset Coordinator: Brandon is responsible for obtaining and keeping track of any materials necessary for the project such as LoRa gateways, Android Phones, etc. He is also responsible for reserving meeting locations and is the team's liaison to Dr. Vigil-Hayes' student research lab.

2.1 Development Process

In the early development phase of this project we started off working in two pairs. Ryan and Moe started work on the Android Studio Library. Benjamin and Brandon worked on the Proxy Server. It was important for our team to split the development process because in order to "test" our project we needed every aspect to be working in some state or another. The two pairs had to work together for combining modules and working on common components, like managing version controls, and maintaining proper testing. The dynamic and roles within the pairs fluctuated throughout the

semesters as well based on an individual's strengths, other team responsibilities, or an individual's responsibilities at that time.

Of each development pair for the Android Studio Library and Proxy Server, one person was in charge of version control. The Android Studio Steam used the built in Version Control integrated with GitHub within Android Studio. This made keeping files up-to-date simple and caused very few merge conflicts. For each component within the Proxy Server, as someone would update or create a file, they would push their changes to their own dedicated branch within the Git repository. After which the version controller would be responsible for accurately merging the files. This is how our team managed the version control aspect of our project and kept our files up-to-date.

Software components for our project had to be developed in a specific order due to the complexity of "testing" our code with other components of our project. The Android Studio Library had to connect to our Proxy Server in order to properly test the Library. We could not emulate this connection without the Proxy Server working. This issue resulted in our team working on both components separately until both worked with one another. We typically started from the bottom and worked our way up. For the most part working on both components went smoothly. Occasionally one of us would get stuck on implementing a specific feature and extend it to the next week or switch tasks with another teammate. Halfway through our project we decided as a team to add a new component called the "configuration service". This would serve as a utility to help synchronize the front-end (mobile device) and the back-end (proxy server) of the framework. Our team had to adjust our schedule to accommodate this new change to our projects timeline. Our weekly meetings were targeted to make sure everyone was on task and no one felt lost with our project. We would use our weekly meeting times to delegate new work and make sure our team stayed on track. It was important for our team to prioritize tasks as development problems came up.

3. Requirements

In order to obtain the requirements for this project, we met with our client on a weekly basis to gather an understanding of what they wanted us to build. From this, we developed a rough requirements list which was passed back to the client for feedback, and then updated. We repeated this cycle a number of times to further refine our requirements list. Finally, we met with the CANIS lab to gain a deeper understanding of the LoRaWAN technology and the limitations it imposed upon our project. From this, we have determined the following domain level requirements:

3.2.1 An Android library which allows an application to submit data to LoRaWAN.

3.2.2 A flexible proxy server which abstracts the process of receiving data on the LoRa Gateway and forwarding it to its intended destination.

3.2.3 A proof of concept Android application which implements the aforementioned library and server to connect the iNaturalist and/or OpenCellID applications to LoRaWAN.

3.3 These aforementioned library and server will be easy for a developer to use or extend for their mobile projects.

3.1 Functional Requirements

Our functional requirements are based upon the requirements list we obtained from meeting with our client. This section is separated into the three main parts of our project: the Android library, the proxy server, and the proof-of-concept application which implements the library and server. These requirements are listed below:

Android Library

3.2.1 An Android library which abstracts the process of encoding data and sending it from the phone to the LoRa Node.

3.2.1.2 The library will provide functions which allow services and applications on the phone to submit data that will then be transferred over LoRaWAN.

The goal of these functions is to be used by applications on the Android device. Developers will implement these functions with their own API hooks for applications they wish to tie in to. These functions will be designed with generics to begin with which will make it easier for future developers to implement their own objects in future applications.

3.2.1.3 The library will provide functions which take this submitted data and encode it by mapping the message's information to bytes in a LoRaWAN packet.

The CANIS lab already has a method for encoding messages to fit onto the low-throughput connection of LoRaWAN. We will be using a similar method for how we encode our messages. The encoded message can only be 13 bytes.

3.2.1.3.1 The application which submitted the message is identified by the first byte in the encoded message.

The first byte will show which application is submitting the message. To begin with, we will only support the iNaturalist or OpenCellID applications. All other applications will be set to same byte ID.

3.2.1.3.2 The service which the message is performing is identified by the second byte in the encoded message.

The second byte will represent which service the application is trying to perform. These services will be API calls for functions supported by the application that sent the message which is represented by the first byte. If the sending application is not iNaturalist or OpenCellID, then this byte block will be ignored for the time being as we are only developing with the two aforementioned applications for this project but this can be expanded on by future developers.

3.2.1.3.3 Any arguments passed along with the message are encoded in the remaining bytes.

Depending on the service called by the second byte, some number of the remaining bytes can be used for arguments. This will give future developers plenty of room to expand as needed.

3.2.1.4 The above functions will be able to service the basic API calls made by the iNaturalist or OpenCellID application.

As mentioned above, we will be designing our encoded messages to work with iNaturalist or OpenCellID's APIs and all other applications will be ignored for the time being. Our encoded message will be able to use a few API hooks from the aforementioned applications when we are finished.

3.2.1.5 If a message is too large to be encoded into a single packet, our library will drop the message.

If the library finds that a message is too large, it will drop the message in instead will report an error to the node which will then be passed along to the gateway.

3.2.1.6 Functions which establish a WiFi connection to the LoRa Node and transmit the encoded messages.

Similar to how the first byte will represent the application that is sending the message, if the library finds that an unsupported application or an application that is supported is trying to send a message that is to large, it will drop the message in instead will report an error to the node which will then be passed along to the gateway.

Proxy Server

3.2.2 A configurable proxy server which abstracts the process of receiving data on the LoRa Gateway and forwarding it to its intended destination.

3.2.2.1 The proxy server will be able to connect to the LoRa Gateway over WiFi and receive incoming messages.

3.2.2.2 A configuration service will be running on the server that provides an interface that allows developers to connect to the proxy server and configure it.

3.2.2.2.1 The configuration service can accept a secure remote connection from a developer.

The proxy server will need to be configured by a developer to recognize the encoding pattern used on messages. To accomplish this, the developer will remotely connect to the proxy server and issue it commands.

3.2.2.2.2 The configuration service can accept a definition of a type of message to handle.

3.2.2.2.2.1 A definition of a message will include the application which submitted the message.

This information is purely to identify the message as well as its destination. The messages will be passed off to handlers based upon the application which submitted them.

3.2.2.2.2.2 A definition of a message will include the service which the message is performing.

This is to determine how the handler will decode the message. Since messages with different purposes will require different encoding methods, they also require different decoding methods.

3.2.2.2.2.3 A definition of a message will include necessary authentication tokens for the message.

Due to the limitations of LoRaWAN, it is currently infeasible to include an authentication token in the encoded packet sent from the LoRa Node to the LoRa Gateway. For this reason, authentication tokens for messages will be supplied by the developer during configuration of the proxy server.

3.2.2.3 The proxy server will generate a handler for an application based upon the definitions provided above.

3.2.2.3.1 The handler can receive messages intended for its application.

Each handler will be responsible for decoding and forwarding all messages associated with an application. It will thus must be able to receive all messages from the LoRa Gateway which were sent by the application the handler was created to serve.

3.2.2.3.2 The handler can decode the received message based upon the service the message is performing.

After the message has been received from the LoRa Gateway, the message will be decoded to get the message ready to be sent to its intended destination. Different services will require different encoding schemes, and so the handler will need to differentiate

between them by looking at the second byte of the encoded message [3.2.1.3.2].

3.2.2.3.3 The handler can handle and forward any message associated with its application.

Since the handler services all messages sent by its application, it must be able to service any type of message that application could send.

3.2.2.3.3.1 The handler must be able to service the basic API calls made by the iNaturalist or OpenCellID applications.

For now, our client only requires that we extend the iNaturalist or OpenCellID applications. Thus, for now, we need only make handlers for these specific applications.

3.2.2.3.4 The handler can manage and distribute the authentication tokens for the messages being handled.

Any message sent will require an authentication token which verifies to its recipient that the message was sent from a legitimate source. These authentication tokens will be provided by the developer during configuration, and the handler will manage the process of assigning these tokens to outgoing messages.

3.2.2.3.5 The handler can forward the decoded messages to their intended destination.

The handler will be responsible for establishing a secure connection to the destination of a message and sending the message to its intended destination.

Proof of Concept Application

3.2.3 A proof-of-concept application which extends the OpenCellID app and demonstrates the use of the library and proxy server developed above.

We are primarily developing a framework for future developers to use and build upon. To prove this framework's efficacy, we will be creating a proof-of-concept application that uses our library [3.2.1] and proxy server [3.2.2] to show their functionality.

3.2.3.1 This application will take messages sent by the iNaturalist or OpenCellID apps, intended to be uploaded to the web server, and, using the library, send these messages to the LoRa Node.

The main goal here is to take messages and data sent from iNaturalist and or OpenCellID and send those messages as packets to the LoRa Node. It is important that these two applications work because our client specifically request these two apps. The messages do not need to be broken up but be sent as one big packet.

3.2.3.2 The proxy server will be configured to handle the messages sent by this application.

The proxy server will be configured with definitions for the API calls made by the supported application. Any necessary authentication tokens will also be put on the proxy server.

3.2.3.3 Assuming that the messages arrive at the proxy server, it will forward them to the appropriate server on the web.

Once the message arrives at the proxy server, it must be successfully decoded and forwarded to the web server we are extending with our framework.

3.3 Non-Functional Requirements

Our non-functional requirements are based on our client's desire for this project to serve as a tool for future development of mobile applications using LoRaWAN as a network. The section also includes the basic requirement of security common to all networked applications.

3.3.1 The framework will maintain the security of data entrusted to it

3.3.1.1 The wifi connection [3.2.1.6] between the Android application and the LoRa Node will be secured with encryption.

3.3.1.2 The wifi connection [3.2.2.1] between the LoRa gateway and the proxy server will be secured with encryption.

3.3.1.3 The proxy server will not allow developers to supply authentication tokens for messages [3.2.2.3.4] over an insecure connection.

Due to the limitations of LoRaWAN, we cannot include authentication tokens in the messages themselves. As such, the authentication tokens will be provided by the developer when they configure the proxy server. To maintain the security of this, developers will only be able to supply authentication tokens over a secure connection.

3.3.1.4 The connection between the proxy server and the wider internet [3.2.2.3.5] will be secured with encryption.

3.3.2 The framework will be easily usable by future developers.

3.3.2.1 The codebase will be open source.

Since the ultimate goal of this project is to enable further development of mobile applications which use LoRaWAN, we will make the codebase open source so that others may extend or copy our code for their own projects.

3.3.2.2 The codebase will use markdown to create a robust reference document hosted on the Github repository.

For the same reasons stated above, we will endeavor to make our work easy to understand and use. The reference document created with markdown will be a major component to this.

3.3.2.3 The CANIS lab will be able to implement the framework without outside assistance.

To verify that our framework is easily usable by future developers, we will be passing the finished project with documentation to the CANIS lab. The expectation is that they should be able to implement our framework to send a message over LoRaWAN. Being experienced with LoRaWAN technology, the

CANIS lab is representative of the developers who might want to use our library in the future.

3.3.3 The framework will be easily extensible by future developers.

3.3.3.1 The codebase will be heavily commented and documented to explain design decisions.

Many of our basic design decisions will be based upon the work and experience of the CANIS lab. These will be explained in the documentation to provide future developers with the context that led to the finished framework.

3.3.3.2 The codebase will avoid an Android-specific implementation wherever possible.

While initially the client only wants to prove the viability of this project on an Android platform, later on they want to expand the project to support other platforms. By avoiding an implementation which leans heavily on Android, we can make this future goal easier.

3.4 Environment Requirements

When formulating our functional and nonfunctional requirements, we also needed to take into consideration the CANIS Lab and their technology. Environmental requirements deal with the domain requirements and their interactions with the system and the final product will run on. They are as follows:

3.4.1 The project will be compatible with the CANIS lab LoRa Node.

The CANIS lab is working with the LoRa Node and will be the primary users of our project. With this in mind it is important that our Android library [3.2.1] is compatible with their technology.

3.4.2 The project will be compatible with CANIS lab LoRa Gateway.

Another compatibility requirement we need to keep in mind is having the CANIS lab's LoRa Gateway work with our proxy server [3.2.2]. We will endeavor to make the proxy server work with any LoRa Gateway but must make sure our system works with the CANIS lab's hardware.

3.4.3 The proof of concept application will extend the iNaturalist or OpenCellID application.

To demonstrate our working library we will be creating a proof-of-concept [3.2.3] application. Our client requested this test application would use data from iNaturalist or OpenCellID. This will be a great way to demonstrate our working library at the end of the project.

4. Architecture and Implementation

4.1 LoRaMessenger

The LoRaMessenger library will be an Android library that provides an interface for developers to send messages over LoRaWAN. When given a message, the library will lookup an appropriate encoding table for the specific API call of the message. As is shown in Figure 1, using this encoding table, the library will convert the data of the message into much smaller byte codes, including a byte code to identify which app sent the message and what API call the message is using. All of these bytes will be combined into an encoded message that will be stored in an output stream and make sure that the socket is still connecting to the LoRaNode. When the socket is still connected, the message will be ready to be sent off to the LoRaNode. If not, the socket needs to loop back into establishing connection and get it ready to be sent off to the LoRaNode.

4.1.1 Diagrams

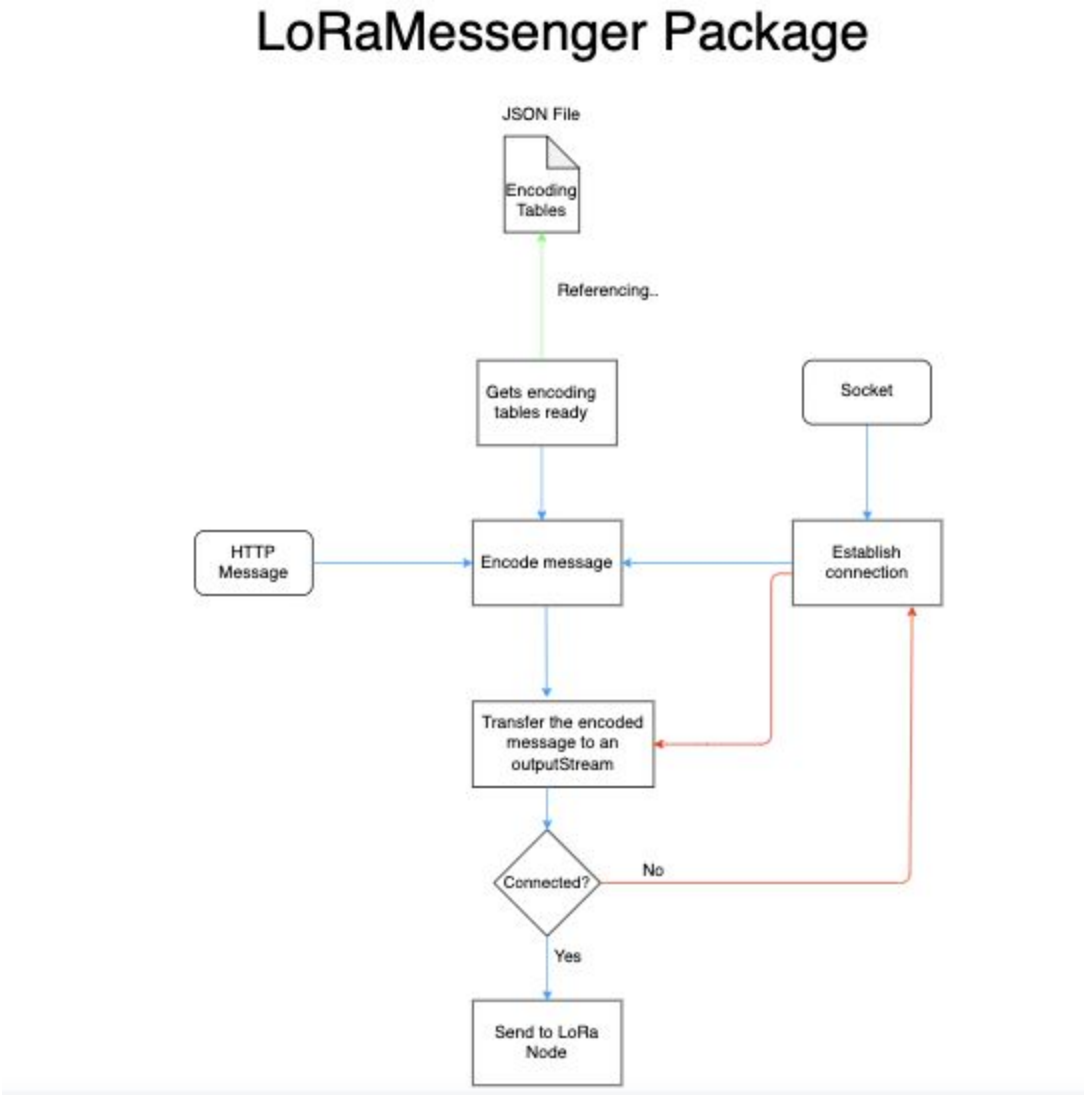

Figure 1: Shows the workflow of the LoRaMessenger class inside the LoRaMessenger package and how the message will be transferred to the LoRa Node.

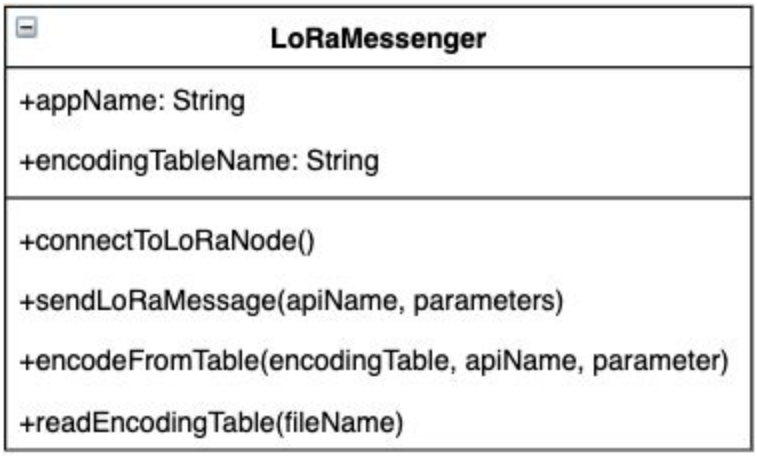

Figure 2: UML diagram of the LoRa Messenger class showing the methods and global variables that are needed to encode the message.

4.1.2 Public Interface

The public interface of the LoRaMessenger is the collection of public facing functions which developers can call in their application. The most important of these functions is sendLoRaMessage() which encapsulates much of the LoRaMessenger's behavior. A developer should be able to treat sendLoRaMessage() as a black box and avoid worrying about the details of how the package encodes and decodes messages. The other methods included in this interface are called by sendLoRaMessage() and perform the actual encoding and networking. These methods are made public so a developer that wants to tinker with their own implementation of sendLoRaMessage() is free to do so.

sendLoRaMessage(String : apiName, var : parameters)

The main method of LoRaMessenger which a developer calls, passing it the name of the API they wish to send and a collection of parameters. These parameters and return behavior are described below in table 1. This method will iterate through the parameters, use encodeFromTable() to convert these parameters into byte codes, concatenate these byte codes into a message, and then send that message.

apiName	The name of the API to be encoded by the library. This name should be given as it appears in the encoding table.
parameters	A collection of the parameters for the API call. Any parameter given must be included in the encoding table.
returns	If the apiName or a parameter could not be found in the encoding table, return

	UNKNOWN_ENCODING_PARAMETER_ERROR. If the combined byte codes of the parameters would exceed the allowable size of a packet on LoRaWAN return EXCEEDED_PACKET_SIZE_ERROR. Otherwise, return nothing.

Table 1: sendLoRaMessage Function(String : apiName, var : parameters)

connectToLoraNode()

The connectToLoraNode function will check to see if a connection has been established to the LoRa Node. If this is not the case, it will establish a connection between the Android application and the LoRa Node. Otherwise, it will do nothing. The return behavior of this function is described below in table 2.

returns	True if either a connection with the LoRa Node has already been established, or no such connection existed but the function was able to establish one. False otherwise.

Table 2: connectToLoraNode()

encodeFromTable(var : encodingTable, String : apiName, var : parameter)

Given an encodingTable file handle and an API name, this function will look up the passed parameter in the encoding table and return the corresponding byte code. The parameters and return behavior of this function are described below in table 3.

encodingTable	An encoding table parsed and formatted as a dictionary, as returned from readEncodingTable() to be used to encode the parameter.
apiName	The name of the API this parameter is for. This name must be given as it appears in the encoding table.
parameters	The parameter to be encoded. Any parameter given must be included in the encoding table.
returns	The parameter's corresponding byte code.

Table 3: encodeFromTable(var : encodingTable, String : apiName, var : parameter)

readEncodingTable(String : filename)
TheJSON encoding table is read and parsed into a dictionary to be used by encodeFromTable(). Only the part of the encoding table needed for the app will be read. The parameters and return behavior of this function are described below in table 4.

filename	The filename of a valid encoding table stored as a JSON file.
returns	A dictionary which contains all the information of the encoding table needed for the app.

Table 4: readEncodingTable(String : filename)

4.2 Proxy Server

The proxy server will sit between the LoRaGateway and the wider internet. Its purpose is to receive the encoded messages sent by the LoRaMessenger and convert them back into readable, useful data. This will entail reading the app and API byte codes of the encoded message and using these to select the appropriate encoding table. Using the encoding table, the proxy server will convert from byte codes back to data and then finally forward the reconstructed message off to its destination.

The proxy server uses multithreading to distribute its various tasks. As depicted in Figure 3, the proxy server will have a main thread which initializes the server and listens for incoming messages from the LoRa Gateway. Once a message is received, the main thread will create a new thread and pass the message off to this child. Multiple different child threads working on different messages can be running concurrently. Meanwhile, the main thread will continue to listen for new incoming messages. The child threads will each be responsible for decoding a single message and forwarding it to its destination. Once a child thread has finished this task, it will terminate.

4.2.1 Diagrams

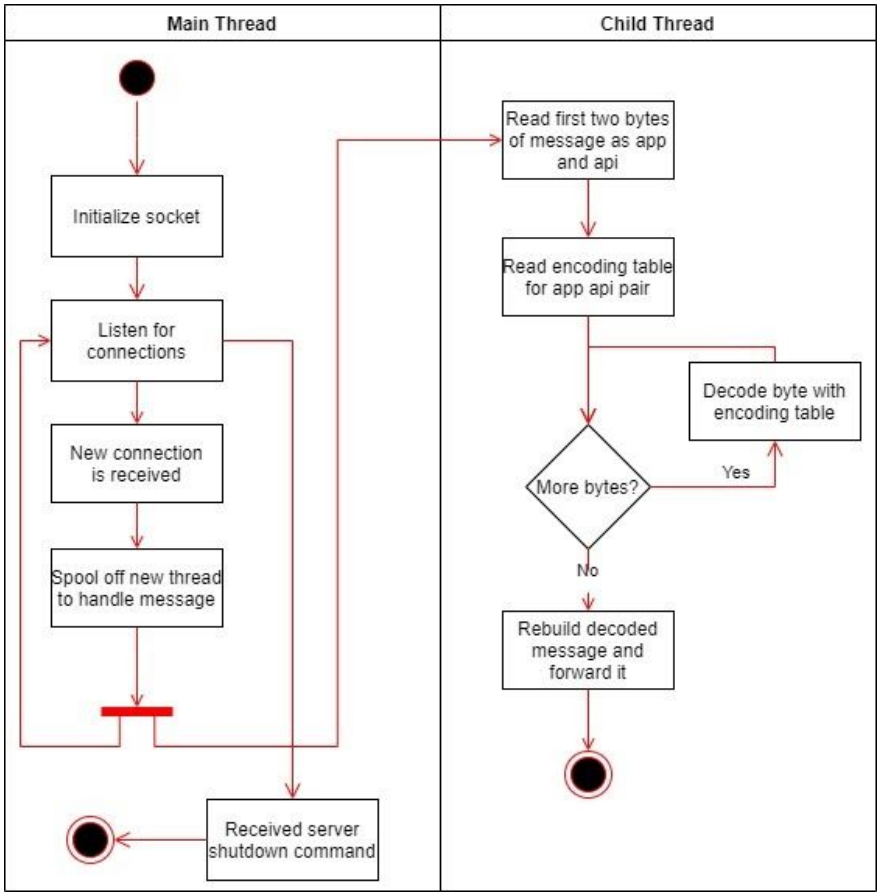

Figure 3: A UML activity diagram depicting the control flow of the proxy server.

The child threads of the proxy server are defined by the messageHandlerThread class, shown in Figure 4. Objects of this class contain the instance variable message and the function run() which define the majority of the object's behavior. When run() is called, it will make use of the classes' other functions to decode and forward the message.

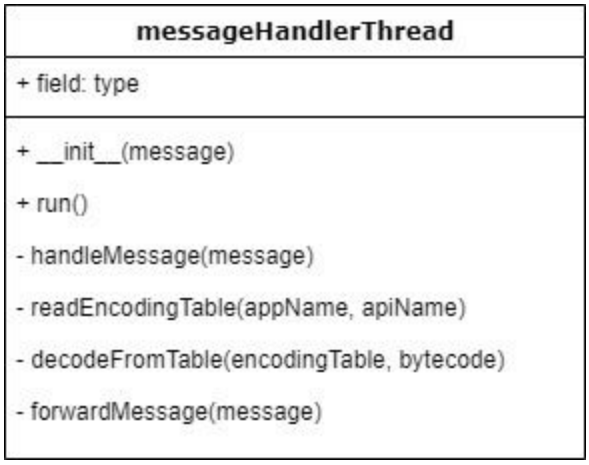

Figure 4: A UML class diagram depicting the class which performs the essential decoding of received messages. Objects of this class are created by the main thread and run on child threads.

4.2.2 Public Interface

The public interface of the proxy server is the collection of public facing functions for running the server. It is composed of the main function which runs the server loop and the messageHandlerThread object which decodes and forwards a single message. The purpose of the proxy server is to be a black-box that encapsulates the process of decoding messages. As such the rest of the functions of the messageHandlerThread are not part of this public-facing interface.

main()

The main method which runs the main thread as described above, in the left half of figure 3. This is the primary way that the proxy server is started and the primary way the server is shut down is by terminating this function.

__init__(var : message)

The constructor for a messageHandlerThread() object. Its parameters and return behavior are described in table 5. When a messageHandlerThread is created, it must be given a message to handle. The resulting messageHandlerThread object is responsible for decoding that message and only that message.

message	The message to be decoded, in the form of a string. Each messageHandlerThread object is concerned with decoding and forwarding a single message.
returns	Nothing.

Table 5: __init__(var : message)

run()

The main function of the messageHandlerThread object which performs the decoding and forwarding of the object's message. Its parameters and return behavior are described in table 5. When a messageHandlerThread is called with threading.start(), a new thread will be created and that thread will automatically call the messageHandlerThread's run function. As such, this function is essentially a wrapper for handleMessage().

returns	Nothing. There are two reasons for this. The first is that this framework is presently not concerned with implementing downlink. That is to say, the framework does not support

	the proxy server communicating back to the device which originally sent the message. Should something go wrong while handling the message, there is no way for the proxy server to request a retransmission. The second is that there is less overhead if the main thread of the proxy server does not worry about receiving returns from the handler threads it spawns.

Table 6: run()

4.3 Configuration Service

The configuration service serves as a utility to help synchronize the front-end (mobile device) and the back-end (proxy server) of the framework. The developer will create a JSON file that contains a definition of any API hooks and their parameter types they would like to support transmitting over LoRaWAN. The configuration service will then create an encoding table which keys the parameters the developer provided to byte codes. This encoding table will be copied to the library's resources before compilation so that the LoRaMessenger and proxy server. In this way, the encoding scheme of the front-end and back-end will be synchronized.

4.3.1 Developer specified JSON file

This is an example of the JSON file that a developer would submit to the configuration server to produce an encoding table. The file contains the programs “TempControl” and “LightControl” that will be accessed, as well as their API hooks “tempUp”, “tempDown”, “on”, and “off”. The name and type of parameters for these API hooks are also included.

```
{
  "TempControl" : {
    "tempUp" : { "increaseAmount" : "int-param" },
    "tempDown" : { "decreaseAmount" : "int-param" }
  },
  "LightControl" : {
    "on" : { "lightLocation" : "int-param",
          "intensity" : "int-param"},
    "off" : { "lightLocation" : "int-param",
           "intensity" : "int-param"}
  }
}
```

Figure 5: Example JSON file for configuration

4.3.2 Resulting Encoding Table

This is an example of one of the encoding tables produced by the configuration service. It demonstrates programs specified by the developer which are then mapped to byte strings.

```
{
        0001 : "TempControl",
        0002 : "LightControl"
}
```

Figure 6: Example program table

This is another one of the encoding tables produced by the configuration service. For each line in the ProgTable, an API table is created where all specified APIs are mapped to a bytestring.

```
{
        0001 : "increaseAmount",
        0002 : "decreaseAmount"
}
{
        0001: “on”
        0002: “off”
}
```

Figure 7: Example API tables

With the above encoding tables, an encoded message can be created. If the program TempControl wanted to call the function increaseAmount(), passing it the parameter 2, this would be encoded as (0001, 0001, 0002).

5. Testing

The LoRaMessenger library is a framework that is used to build Android applications, as such the only way to test a library is to try and build an application that utilizes the LoRaMessenger library. A single application was built which interfaced with OpenCellID and to verify that the library was usable. The app was created in the same manner that we assume a developer would go about creating an android app that would utilize our

library. This app shows that our library is able to send a message off to OpenCellID and takes the minimum amount of requirements that a post to OpenCellID requires.

To go along with this, a round-trip-test was built to verify that a message could be encoded, received by the proxy server, and decoded. After the message has been received and decoded, it was compared to the original, unencoded message, to verify that the proxy server is functioning properly. This test was done without any network activity.

Finally, we tested the OpenCellID app with the proxy server on both an actual android device on the same network as the proxy server as well as using an emulated android device.

6. Project Timeline

- **September 13, 2019**: We began by determining our team name and creating the team website. This website would become our team repository containing capstone reports and other required documents for the entirety of this project.

- **October 1, 2019**: We started research on the client's project description looking into its feasibility. At this point we were determining the client's needs and the solutions we could offer them to implement their ideas.

- **September 1, 2019**: Once our feasibility research was completed, our team determined functional, non-functional, and performance requirements for the project. We created a requirements document that the clients had to read and sign. These are the requirements that our team needed to implement for our final product.

- **November 5, 2019**: Work began on our prototype to be demonstrated at the end of the Fall 2019 semester.

- **November 23, 2019**: Our team presented the first Design Review for the project to our team mentor.

- **December 15, 2019**: Our team continued to work on the project over Winter break.

- **January 15, 2020**: Start of Spring 2020 Semester. With knowledge gained from the prototype, our team changed technologies to Android Studio and Firebase. Development on both the Android application and Administrative website begins. 20

- **February 25, 2020**: Our second Design Review was given at this time.

- **April 3, 2020**: Our team presented the third Design Review at this time via video (Due to not being able to meet in person.)

- **April 26, 2020**: Our team participated in the UGRADS virtual symposium.

- **May 6, 2020**: Our team demonstrated the final build of the project to our client. Our client acknowledged and signed off on the final build checklist. The client received the GitHub repo including all code needed to "hand-off" the project.

- **May 7, 2020**: The final report has been completed and turned in to our team mentor. This concludes our team's Capstone experience.

Figure 8: This is a Gantt chart showing some "basic" tasks that we needed to do over the entire capstone year.

Task Name	Feb 2 - 8	Feb 9 - 15	Feb 16 - 22	Feb 23 - 29	March 1 - 7	March 8 - 14	March 15 - 21	March 22 - 28	March 29 - 4	April 5 - 11	April 12 - 18	April 19 - 25	April 26 - 2	May 3 - 8
LoRaMessenger Library:														Today
connectToLoraNode()	Ryan & Moe													
sendLoRaMessage()			Moe & Ryan											
readEncodingTable()			Ryan & Brandon & Ben											
encodeFromTable()			Ryan & Brandon & Ben											
documentation				Moe & Ryan				Moe & Ryan						
testing & finalizing						Everyone		Everyone						
Proxy Server:														
messageLoop()	Ben													
handleMessage()	Ben													
readEncodingTable()	Ben & Brandon													
decodeFromTable()	Ben & Brandon													
forwardMessage()														
documentation				Ben & Brandon				Ben & Brandon						
testing & finalizing						Everyone		Everyone						
Configuration Service:														
lookupTableBuilder()	Brandon													
documentation				Brandon & Ben				Brandon & Ben						
testing & finalizing						Everyone		Everyone						

SPRING BREAK

Figure 9: This is a Gantt chart showing a more in-depth look at a capstone semester. In this Gantt chart you can see each person is assigned a task and a "rough" due date.

Task Name	DUE DATE	April 13th - 17th	April 20th - 24th	April 27th - 1st	May 4th - 8th
					Today
UGRADS:					
Poster Finished and Turned In		Ben			
Capstone Presentation Turned In FINAL	April 23rd (5pm)	Ryan			
Team Stuff:					
Team Website FINAL	April 24th	Ryan & Moe			
Peer Eval #4	April 24th (5pm)		Everyone		
Acceptance Test Demo (scooter)	May 4th (2 pm)		Everyone		
Team Reflections Doc	May 7th (3pm)			Everyone	
Final (as-built) report	May 7th (3pm)			Ryan - Everyone	
User manual and completed checkoff sheet, signed by client	May 6th (11pm)			Everyone	
Peer Eval #5	May 7th (5pm)			Everyone	
LoRaMessenger Library:					
fragmentation		Brandon			
documentation		Ryan & Moe			
testing & finalizing		Ben			
Proxy Server:					
fragmentation		Brandon			
forwardMessage()		Ben			
documentation		Brandon & Ben			
testing & finalizing		Ben			
Configuration Service:					
documentation		Brandon & Ben			
testing & finalizing					
OpenCellID Demo App:					
implementation		Moe			
implementation with connection			Moe		

Figure 10: This is a Gantt chart showing an even more in-depth look at our teams schedule. This would be a close up look over a few weeks. You can see that there are a lot more tasks and can see the "exact" due dates. Each task was assigned to one or more people.

7. Future Work

7.1 Receiving Multiple Messages Concurrently

Currently, our proxy server handles messages one at a time. This means that our proxy server needs to be implemented using the parallel principles. This will let the proxy server handle a huge amount of network traffic.

7.2 Receiving Messages On The Android Phone

Our project was designed for up-links which means that we are only sending messages from the Android Phone to the proxy server. A future work can be implementing down-link that sends messages from the proxy server to Android Phone across the LoRaWAN network. This will allow messages to be sent and received from both ends of the framework.

7.3 Configurable Wifi Connection in the Lora Library

Currently, our project handles a connection over a network by hard-coding the actual IP address. In future work, configurable wifi could be handy in connecting to different lora nodes or servers rather than hard-coding the receiver IP address.

8. Conclusion

As our world becomes increasingly networked, lacking access to the internet becomes an increasingly debilitating position. Many rural communities are in this position, lacking expensive cell towers to connect them to the world. Dr. Vigil-Hayes seeks to solve this problem with the new technology LoRaWAN by providing a cheap and power-efficient option which could connect rural areas and enable mobile crowdsensing. A barrier to this is the lack of an easy-to-use framework that allows a mobile application to communicate over LoRaWAN.

We supplied this framework by creating the LoRaMessenger library and proxy server which, together, abstract the process of transmitting a message over LoRaWAN. The library provides an easy to use blackbox interface to encode and transmit a message to the LoRa Node. Meanwhile, the proxy server will be able to receive messages from the LoRa Gateway, decode them, and forward them to their intended destination.

We believe this framework will serve as a valuable piece of our client's research and serve as a stepping stone for future LoRaWAN development and innovation. There is a great deal of potential for LoRaWAN to expand internet access and connect a great many people to the whole world.

This capstone project was a difficult, but valuable experience that taught us much about the difficulties of software development. No major project is a solo endeavor and so communication skills are paramount to being an effective engineer. Moreover, this project in particular was a good exercise in the difficulties of building backend software and tools. The gritty details are at their grittiest and there is rarely a clean flashy web page that can be shown to wow an audience. However, there is also something satisfying in creating a tool that future developers might use as a building block for future projects and that is what we will take away with us as we leave LoRaWAN behind.

9. Glossary

API : Application Programming Interface
API Hook : A function/method to be used from a given API
Git : Version control system https://git-scm.com/
GitHub : A site used for cloud based, Git version control
iNaturalist : Software used for reporting nature related data
https://www.inaturalist.org/
JSON : JavaScript Object Notation
LoRaWAN : Long range Wide Area Network
OpenCellID : Database of Cell Towers https://opencellid.org
UML : Unified Modeling Language

10. Appendix (absolutely required)

10.1 Hardware

Ryan, our team lead, developed for the project on an iMac (3.7 GHz 6-Core Intel Core i5) with 40 GB of RAM. Ryan also developed an HP laptop (2.0 GHz CPU) with 8 GB of RAM.

Benjamin, our architect, developed for the project on a 64 bit Windows 10 machine with 8 GB of RAM and a 1.60 GHz CPU. By preference, he also had the Windows Subsystem

for Linux (WSL) installed on his machine, which allowed for some things to be done in an Ubuntu environment.

Brandon, the asset coordinator, developed on a 64 bit Windows 10 machine with 16 GB of RAM, an Intel i7-7700K (4-Core) CPU at 4.20GHz. Additionally, Brandon used a VirtualBox virtual machine running Debian 10 with 4GB of RAM, to allowed for testing in a Linux based environment.

Moe, the release manager, developed on a 64 MacOS machine with 4 GB of RAM, an Intel i5 (dual-core) CPU at 2.5GHz.

Android Studio, a necessary IDE for the development of Android applications, has the following minimum requirements:

- Windows 7/8/10 (64-bit) OR Mac OS X 10.10 to 10.14 OR 64-bit Linux distribution capable of running 32-bit applications with a GNOME or DE desktop
- For Linux machines, GNU C Library 2.19+
- 4 GB RAM, though 8 GB RAM is recommended
- 2 GB available disk space, though 4 GB is recommended
- 1280 X 800 minimum screen resolution

As running Android Studio and its built in Android emulator were the most intensive operations necessary for development, the minimum requirements of Android Studio can be thought of as the minimum requirements for developing this project.

10.2 Toolchain

Android Studio
Android Studio is an IDE for developing and deploying Android applications. It comes with a wide array of tools, including an Android device emulator that allowed for easy testing. This was necessary for the development of the project as it was simply the only real tool for developing native Android apps.

Python3
Python3 is the most up to date version of the Python language which is notable for its ease of development and integration with other systems. With Python3 installed, any high quality text editor such as Atom, VSCode, or Idle, would be sufficient to develop the Python modules of the project.

Git

Git is a version control tool used by a huge number of software projects of all different kinds. It would also allow us to combine our individual work on the project in a clean and sensible way on GitHub. Some method of version control would be necessary for a project of this size, and Git is simply the most widely used tool available.

10.3 Setup

To set up a development environment for this project, you would need to:

- Install Android Studio
- Install Python3
- Install git
- Clone the LoRa-Package git repository into a folder on the local machine
- If the local machine will act as the proxy server, view the *How to run the Proxy Server* and the *How to run the configuration Service* files located inside the LoRa-Package/Proxy/res folder
    - If the local machine will not be the proxy server, clone the repository on the machine that will act as the proxy server to continue
- Import the Loralibrary into android studio
    - A guide on how to import the library can be found in the Android Framework ReadMe.md
- Begin development on your android app
    - The interworking of the Loralibrary can be found in the ReadMe.md inside the Lora-Package/Android Framework folder

10.4 Production Cycle

To edit a page layout, locate the XML file in the Android Studio Project tab by going to app > res > layouts

To edit a Kotlin file, locate the Kotlin files in the Android Studio Project tab in app > kotlin > yourProjectName

After making edits, the code changes will save after clicking Run or by amnualing saving the project. Ensure that an android device is plugged in or choose an emulator after clicking run. The application will load on the device of your choice.