

Software Design

7 Feb. 2020

Team Lora

Version 1.1

Community Aware Networks and Information Systems Lab

 Dr. Morgan Vigil-Hayes (Sponsor)

Scooter Nowak (Mentor)

Ryan Wallace; Benjamin Couey; Mohammed Alfouzan; Brandon Salter

[**1 Introduction**](#_n0bwdk47l47e) **4**

[**2 Implementation Overview**](#_hpvshahqk5t8) **5**

[**3 Architectural Overview**](#_7vy9ja13siow) **7**

[**4 Module and Interface Descriptions**](#_a69q9s5557) **8**

[**4.1 LoRaMessenger**](#_cd3f3iq5n9ol) **8**

[**4.1.1 Diagrams**](#_ep10tjsgjis4) **9**

[**4.1.2 Public Interface**](#_af0d39ddkbrt) **10**

[**4.2 Proxy Server**](#_mwj2mnquzcf9) **12**

[**4.2.1 Diagrams**](#_ntv2t3hj5q7n) **13**

[**4.2.2 Public Interface**](#_og0str74oo1j) **14**

[**4.3 Configuration Service**](#_g00hntktw4hu) **15**

[**4.3.1 Developer specified JSON file**](#_2wv12mvjeyaj) **15**

[**4.3.2 Resulting Encoding Table**](#_g3c8j17tmtx2) **16**

[**5 Implementation Plan**](#_jg3ebfv0e992) **16**

[**6 Conclusion**](#_jtk1l38xc9d5) **17**

# 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 be comprised of 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.

In this document, we will be describing our implementation of this framework in increasingly greater detail. First, the implementation overview will reiterate the overall approach we are taking to implement the required functionality of the framework. Then, the architectural overview will discuss in greater detail the primary responsibilities of each component of the framework, along with how those components communicate with one another. The module and interface description will examine the specific functions which make up the components of the framework as well as go into detail on the interface these components present to developers. Finally, the document will lay out our scheduled plan for implementing the framework.

# 2 Implementation Overview

Our library will allow future developers to create Android applications that can send messages via LoRaWAN to the wider internet. For apps like OpenCellID and iNaturalist, this could make recording field-research data significantly easier and allow collaborators to view progress much sooner than before. Our library will be designed in two parts; a front-end and a back-end; this can be seen in the figure below, where the front and back-ends are separated by the “Project Libraries” arrow. The front-end will be the LoRaMessenger library that future developers will use during development of their apps. We will provide various functions and a LoRaMessage class that will make transmitting messages over LoRaWAN possible. The back-end will be a proxy server that will handle decoding and forwarding these messages. Additionally, the back-end will have a configuration service that will create a lookup table used for encoding on the front-end and decoding on the back-end.



#### **Figure 1**: A diagram depicting the journey of a message through our implementation. The LoRaMessenger library will be included in a developer’s application which will be installed on an Android device. Their app will send data via a message encoded by our library to a local LoRaNode. From there, the CANIS lab will handle the transport of the message from the LoRaNode to a LoRaGateway which will then forward the message to a proxy server. The proxy server will then decode the message and forward it along to the intended recipient service with given API calls and necessary parameters.

# 3 Architectural Overview

The project’s architecture consists of three components that are connected over the loRaWAN network. First, the LoRaMessenger component that will be responsible for encoding an outgoing message. Second, a proxy server that will be responsible to decode the message when it’s arrived from the LoRa Gateway. As it shown in Figure 2, both of these aforementioned components will reference an encoding table which will relate bytecode keys to data values. Finally, a configuration service that will allow future developers to generate their own customized encoding tables for their specific application.

####

#### **Figure 2**: An architecture diagram which shows our system’s high-level components and how they will interact with each other.

**3.1 LoRaMessenger**

The LoRaMessenger will be an Android library that provides a black box interface for an Android app to call on in order to send a message over LoRaWAN. As shown in Figure 2, this interface will be responsible for encoding the message by referencing the encoding tables that are generated on the proxy server’s configurable services. The developer is responsible for copying the generated encoding table files and including them with their app. After the message gets encoded, the message is forwarded to the LoRa node via a wifi connection that requires an IP address and a port number. Then, the message will be ready to be sent over the loRaWAN network by the CANIS lab’s protocols.

**3.2 Configuration Service**

Based on Figure 2, the configurable service will live on the proxy server component and serves as a utility to generate encoding tables. Future developers will submit a collection of the valid data values their app might send, and these values will be related to byte code keys. After it gets customized, the encoding tables will be written to a JSON file that will be copied to the LoRaMessenger’s package by future developers to synchronize the encoding scheme between the LoRaMessenger and proxy server. Later, both the LoRaMessenger and proxy server will refer to their copy of the encoding table to encode and decode messages respectively.

**3.3 Proxy server**

The proxy server will be responsible for receiving and handling messages from the LoRa Gateway and decoding it based on the encoding table. After the message gets decoded, the proxy server will simply forward the message to its destination.

# 4 Module and Interface Descriptions

## 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 3, 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 3**: Shows the workflow of the LoRaMessenger class inside the LoRaMessenger package and how the message will be transferred to the LoRa Node.



#### **Figure 4**: 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 5, the proxy server will have a main thread which initializes 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 5**: 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 6. 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 6**: 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 5. 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 libraries 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 7**: 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 how programs specified by the developer which are then mapped to byte strings.

|  |
| --- |
| { 0001 : "TempControl", 0002 : "LightControl"} |

 **Figure 8**: 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 9**: 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 Implementation Plan

The goal of this project is to complete the most critical parts by the end of February in order to produce a working prototype. In figure 10 you can see our Gantt chart depicting our implementation plan. Each team member is assigned to one or more tasks over the upcoming weeks in order to complete the project. In figure 10 you can see that each section has been divided into 3 main sections: developing the LoRaMessenger, developing the Proxy Server, and developing the Configuration Service.

Since testing of our project requires all components to be working and communicating, we are adopting a parallel development strategy. This will help speed the production of our project and will keep us on track to have a working prototype completed before the end of March. As shown in Figure 10, after spring break we will focus on refining time on testing our code while working with our client and her CANIS lab to implement it on their hardware.



#### **Figure 10**: A Gannt chart depicting which members of the team will be working on which components of the project as time goes on.

# 6 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 will supply this framework by creating an Android library and proxy server which, together, abstract the process of transmitting a message over LoRaWAN. The library will provide functions 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.

In this document, we detailed how we will implement this framework by breaking the project down into its three critical components: the LoRaMessenger library, the proxy server, and the configuration server. As we develop the framework, this document will serve as a blueprint from which to build. If it becomes apparent that parts of this plan need to be refactored as we work on the project, this document will be updated to reflect our changed implementation. Finally, we outlined a schedule for the development of these components.

Considering this project’s current and future development, we are confident in our ability to implement the framework as it is described in this document. We have already created a proof-of-concept for the configuration service and proxy server which have proven promising. Having this early version of the configuration service will be very valuable as it will help us develop a working prototype of the other components which rely on the configurations service. Our only outstanding concern is how we will overcome LoRaWAN’s limitation of extremely small packet size to transmit arbitrary primitive data types. 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.