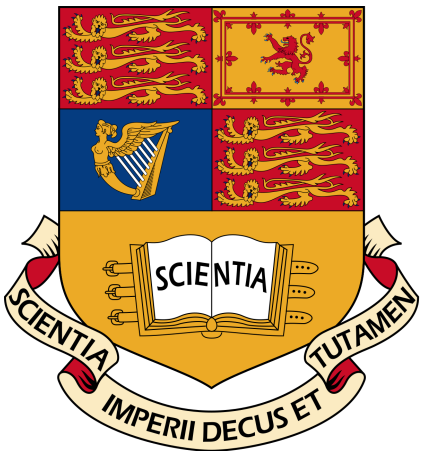


Imperial College London

Department of Electrical and Electronic Engineering

Giving Access to Reliable Communication for Outdoor Activities in Remote Areas

Authors' Names:	***Authors' CIDs:***
Andreas Floros	01539636
Arie Arya	01522674
Arnau Bonet Farres	01577271
Arunansu Patra	01517169
Gabriele Johannes Giuli	01514763
Matthew Cooper	01530402
Oliver Bihami	01547598
Vincent Borchers	01569387

Group Supervised by *Dr Javier Barria*

Final Report

27 March 2020

Contents

Abstract

The objective of the project is to give people lacking access to usual communication infrastructure the ability to communicate conveniently, reliably and with small financial expense within an off-the-grid network. The groups that are targeted by this solution are communities living in rural areas or more specifically, groups of hikers or groups taking part in exploring expeditions.

To achieve this kind of communication, a mesh network topology is implemented using portable, battery powered devices as the nodes. These devices are linked within the network by means of *XBee* radio modules. A major role in the process of communication has our own communication protocol that is used for encoding and decoding of messages.

The interface for the user happens through a custom phone *App* which is used to send and read text messages. A feature that lets all network participants know about the case of an emergency is implemented as well. A *GPS* module is used to automatically send the current location of a network participant in need for help to all other participants.

As a result, three physical nodes have been built to test the network, and it works as intended. The range of the system is in the hundreds of meters and can easily be raised to several kilometres by exchanging the existing radio modules with more capable ones. This would need to be done in order to achieve a decent range for using the system in real life applications.

Many aspects of the prototype of the system can be further improved. As a next feature, the system could be connected to the internet to allow communication to the rest of the world. From a constructional point of view, the circuit could be built on a *PCB* to increase reliability and to improve the small form factor of the devices.

1 Introduction

The aim of this project is to design an *off-grid* radio communication system targeted to small communities located in remote areas. The system can be used for ordinary communication, in the form of text messages. Furthermore, location services and a quick-access emergency button allows the system to be used for critical communication in a distress situation.

1.1 Problem Specification

Taking part in outdoor sports such as hiking or mountain climbing can be dangerous, so the people who do these sports need to be able to reliably send distress signals through long distances or plan rendezvous with other groups. However, in these remote environments, communication methods that rely on pre-built infrastructure can be less reliable or not even usable. From 2004-2014, about 20,000 search and rescue cases across all U.S. National Parks were for day hikers [7]. In 2007, two people froze to death in a cross-country skiing trip in Norway [21]. Current commercial communications rely on pre-built mobile networks and satellites which can be difficult to access in remote areas and, in the case of satellite communication, can be very expensive. Although traditional radio communication methods do not require external towers, their range is limited and the amount of information transmittable is very limited. In the event of an emergency or natural disaster, such methods of communication will be unreliable.

The aim of this project was to create a product that enabled communication through a network that did not rely on pre-existing ground infrastructure so that users can communicate on-the-go in remote areas. We focused on an audience of people who would be taking part in sports that have a significant element of danger in remote areas, and in the long term we want to expand our audience to be able to provide coverage for remote areas of residence.

We wanted to make a product that was as convenient as possible to the user, while also keeping its primary function of sending/receiving messages as reliable as possible. For user convenience, we aimed to make the product lightweight, compact (the device should ideally fit in a rucksack without being too heavy), user-friendly and relatively cheap. We aimed to minimize the cost of the product, as a dangerous situation could occur to anyone in a remote environment and such a product should be widely available to the public at a low price. To limit costs, we had to build the mechanical parts ourselves and use cheaper radio modules. We also were aware that the user may be stranded for a long period of time, so the battery life should be lengthy and the product should be easy and intuitive to use.

1.2 Competitor Analysis and Market Share

Although communication methods for remote and inaccessible areas exist, such as radio communication, applications of these methods require very specialised equipment and is very limited in range. Furthermore, the quality and amount of transmittable data information is limited. Satellite communication has now become the dominant method of communication, providing nearly global coverage without the limitations of radio communication. However, the increased cost of satellite communication results in many people not being able to afford such communication services. As such, mobile data access is limited in developing countries, and in 2014, 19 million people in the United States did not have broadband access [22].

There exist companies such as *Groundcontrol*, *Inmarsat*, and *Iridium* that provide satellite Wi-Fi solutions for prices ranging from $1,000 to $13,000 just for the device [5, 4, 6]. Other companies such as *Nokia*, *Omoco*, and *Ericsson* provide IoT networks for industrial applications, and plan to bring 5G to rural areas [3]. The main disadvantage in the services provided by these companies is the affordability, therefore accessibility, of their services to people especially in remote or less developed

areas. *Project Owl* offers an emergency network for people to connect via smartphones to ask for help in emergencies [11]. While their applications match closer to those of this project, the fact that this is only designed to send emergency signals and not routine messages limits its usability. Other solutions like *Firechat* connect mobile phones through an App using peer-to-peer wireless mesh networking via Bluetooth or various multipeer connectivity frameworks [10]. While this may be the closest currently existing solution to the outlined problem, it relies on people with smartphones being close in range with each other, thus cannot be extended to cover large distances. This project aims to deliver a reliable communication method for remote areas at a much reduced cost.

2 Design Criteria

This is a selection of the most important and updated product specifications. For a more complete analysis of the design criteria please refer to the *Product Design Specification* (9.4).

- It must be able to provide reliable off-grid communications as the product will be used in environments where there are no established means of communication. The reliability of this system is key, as the product must not fail when the safety of a human may be on the line.
- The product must be able to communicate long distances ($>$ 1km) so that the nodes that pass on the signal do not have to be right next to each other.
- The product must be able to transfer data in real time so that the user can get a response as quickly as possible but must also be capable of temporary data storage to account for the fact that users may be offline which should not be a reason to miss a message.
- The device should be capable of running on its own power for long periods and not rely on external infrastructure to send a message as in the intended environments, access to a power grid and other infrastructure may not be available.

These are additional design criteria that have been developed after the interim report:

- In the case of an emergency, the device should allow to easily broadcast the location of a person in need for help to all network participants. The capture of the location must not rely on the GPS functionality of a phone. The broadcasting of the emergency signal needs to be initiated by a simple button press. The button needs to be of reasonable size and positioned clearly in such a way that it is recognizable as an emergency button on the outside of the device.

3 Concept Designs and Selection

There are various designs that would satisfy the above specifications. In this section we present the designs that were considered and the reason why our chosen implementation is the best in terms of cost efficiency and utility. Note that for compactness, we present a summary of each idea. The reader is encouraged to refer to the preliminary report for a more in-depth analysis.

Our proposed solutions can be grouped in two categories: Satellite communication and communication via radio modules as shown below:

- Satellite communication
- Communication via radio modules:
  - Mesh network:

    - Centralised network
    - Decentralised network
    - Decentralised half duplex network
- Low power system

Satellite communication, while effective, is much more expensive, thus our product would not be very attractive for customers and surviving the competition in the market would be nearly impossible. Shown below is the cost analysis conducted for the satellite idea:

Component	**Individual Cost (£)**
Raspberry Pi 3 Model B	32.00
3.7 V 2600 mAh Lithium-Ion Batteries	7.11
TP4056 Li-Ion Charging Module	0.97
RockBLOCK 9603 Module	209.00
Additional Transmission Cost	1.60/kB
Total	249.08 + 1.60/kB

Table 1: Cost analysis of satellite idea [20, 15, 2, 16, 12]

We then considered implementing a communication system based on radio modules. We present two variations of this idea:

One approach would be to implement a mesh network structure and to use path finding algorithms to determine the optimal route for each message to reach its target destination. Based on this structure we can again take different approaches. The solutions we considered were a centralised system, a decentralised system and a half duplex network. The centralised approach is more robust compared to the other two since we would be implementing backup networks in case connection is lost. However, it is far more complicated and so we chose to reject this proposal. The differences between the half duplex system and the decentralised approach are software based, in terms of hardware the two approaches are identical. Overall, the half duplex approach is more restrictive as it assumes only one user can send a message at a time. Shown below is our cost analysis for these methods:

Component	**Individual Cost (£)**
Arduino MKR WiFi 1010	30.12
DIGI XKB2-Z7T-WZM	27.55
3.7 V - 1.8 Ah LiPo Batteries	9.26
BOB-08276 Breakout Board	2.29
Adafruit 2mm 10 pin Socket Headers	1
Total	70.22

Table 2: Cost analysis of mesh network idea [14, 9, 8, 17, 13]

The second option we considered is a naive approach to communication which relies on heuristics for successful reception of messages. The idea behind this method is that we would implement nodes with simple broadcasting algorithms which would broadcast the messages along with their target destination until they reached the intended recipient. The messages would essentially take a random trajectory which would depend on the users passing by broadcasting nodes. The key advantage of this method is that because of its simplistic approach, it would consume very little power compared to the other proposals. The obvious disadvantages are that we wouldn't be able to guarantee successful

communication and that this system requires a lot of users to be effective. Our cost analysis, as shown in Table 3, shows that this approach is just as cost effective as the first approach discussed.

Component	**Individual Cost (£)**
Adafruit Feather M0 with ATWINC1500 WiFi Module	32.5
3.7 V 2600 mAh Lithium-Ion Batteries	7.11
TP4056 Li-Ion Charging Module	0.97
3W 6V 600mA Mini Solar Panels	10.95
Adafruit RFM96W LoRa Module	18.5
Total	70.03

Table 3: Cost analysis of low power idea[18, 15, 2, 1, 19]

Having performed the above analysis we chose to implement the mesh network idea and more specifically, the decentralised system. Shown below is the matrix we used to make this decision:

Solution	**Cost**	**Reliability**	**Appropriate Complexity**	**Scalability**	**Potential**	**Score**
Satellite Communication	3	10	4	10	2	29
Centralized Network	5	8	4	7	8	32
Decentralized Network	7	7	8	7	8	37
Decentralized Half-Duplex Network	7	6	5	7	6	31
Low Power System	7	1	6	4	7	25

Table 4: Decision matrix

4 Concept Development

After a careful evaluation of the *product design specification* (9.4) the project's high level design was developed with the help of the considerations made in Section 3.

As we can see in the figure above, the project can be mainly divided into two major subsystems:

- **User Interface**: allows the user to easily access the network to see who is online, exchange messages with other users, see online user's locations and send a distress signal
- **Network Node**: creates the communication link between users. Manages the stream of information, by redirecting messages to the right recipients

The user interface allows the user to register and access the functionalities of the network: it acts as a link between the user itself and the node. The network nodes constitute the network itself, and each node has to correctly receive, interpret, and send data in order to propagate the messages through.

The communication system of the radio mesh network can be categorized into two main parts, the *client-server protocol* and the *internodal protocol* (radio-to-radio). The foundation of the internodal communication between nodes is the in-built *DigiMesh* protocol in the *XBee* radio modules, which

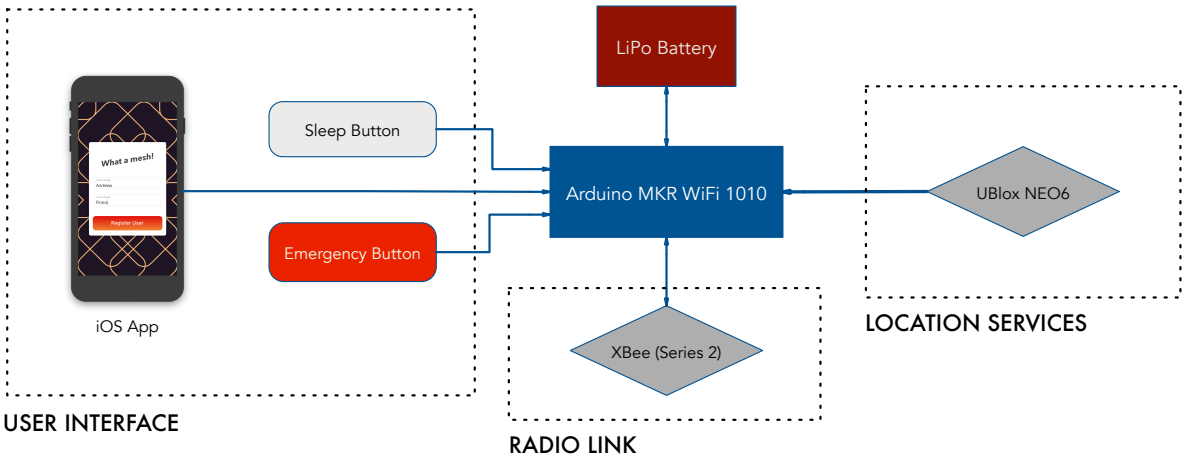


Figure 1: High level design of the project

is a peer-to-peer networking protocol that links the radio nodes in a meshed network topology. This allows for a decentralized communication infrastructure; or in other words, the autonomous relaying of messages throughout the network with no central node dependencies.

The primary objectives of the handwritten protocols introduced for our project are as follows:

- Reliable and collision-free
- Scalable and dynamic
- Fast transmission and reception

Although security may also be an important feature to consider, encryption and decryption systems are not introduced at this stage of the project as an off-grid communication will most likely not contain information that require concealment, nor would it be a chief target for cyber-attacks.

4.1 Client-Server Protocol

The client-side *iOS* application and the server run by the *Arduino* must share a common protocol for bi-directional communication between them. The four main protocols used in the client-server communication are the client connect protocol, message transmission protocol, and the network update and message reception protocol.

The main identification system for each client within the network is the use of a local client IP Address, the local Arduino MAC Address, as well as a unique *client ID* generated by the App. By knowing the ID of a client, a lookup table within the Arduino allows the client's *IP Address* and *MAC Address* to be identified, and hence the client can be located within the network and a directed message can be sent to it.

4.1.1 Client Connect Protocol

The client connect protocol is a *HTTP POST* request made by the App which tells the Arduino server that a new client has joined the local network. This requires both the name of the client, as well as

its unique ID. The *HTTP POST* request is made on the default server IP Address of 192.168.4.1 (set within the Arduino code):

$$192.168.4.1/client\_name : \{client\ \ name\}/client\_id : \{client\ \ id\}$$

Upon receiving the new client, the local server will then broadcast this information to the wide network to notify all other nodes of the existence of this new client, which enables other clients to begin sending messages to this new client via its unique ID, and vice versa. Additionally, upon connecting to the server, the client will be given a unique local IP Address which allows it to be distinguished from other clients in the node. This, along with the Arduino's MAC Address, is attached alongside the client's unique ID upon broadcasting to other nodes so that it can be located in the wide network. This broadcasting mechanism will be described in the internodal protocol section.

4.1.2 Message Transmission Protocol

The message transmission protocol is also HTTP POST request made by the App to the server. Attached to this request is the message to be sent, the source ID of the client sending the message, as well as the destination ID of the target client:

$$192.168.4.1/message : \{message\}/source\_id : \{source\ \ id\}/target\_id : \{target\ \ id\}$$

Upon receiving this request, the Arduino will parse the data and process it. Firstly, since the target ID alone is not sufficient to identify the target client's location within the network, the server must use its lookup table which contains information about all the clients connected in the whole network to find the corresponding clients' local IP Address and MAC Address. After retrieving this data, the message will then be placed in a message queue, which can hold multiple messages before the node broadcasts them out. This is necessary as the nodes cannot transmit the messages immediately; it uses a buffered token system to prevent network traffic and collision, which will be described in the *Internodal Protocol* section (4.2).

After the transmission of the message, the message queue will be cleared to allow it to take new messages from the client, and the process is repeated.

Index	Messages
1	"Hello"
2	"How are you?"
...	...
N	"Hi"

Index	Messages
1	-
2	-
...	-
N	-

Figure 2: Message queue lookup table before and after transmission

4.1.3 Network Update and Message Reception Protocol

This protocol is a *HTTP GET* request made by the App to the server and requests a *JSON* data containing all of the available clients in the whole network, as well as any client-specific messages. This allows the client to have an updated list of all clients in the network, whilst simultaneously receiving any incoming messages from other clients. This protocol combines both client information and message information into a single simple protocol to prevent the over-crowding of the local server by multiple protocols. The protocol is simply given by:

$$192.168.4.1/update\_data$$

The JSON data that will be sent to the client (App) will be in the form:

```
{"Type": "Users",
    "Data":[{"Name":"Name Here", "ID":"ID here", "Location":"Location Here"}],
    "Messages":[{"Source ID":"Source ID Here","Message":"Message Here"}],
    "Emergency":[{"Location":"Location Here}]
}
```

The 'Data' field contains the name and ID of all clients in the network, whilst the 'Messages' field contain all the messages to the client as well as the source ID of the clients who sent the messages. The emergency field contains the data of all users in the network that have called for an emergency assistance (this will be described in more detail in section 9.1.7). This JSON information will then be parsed by the App and both the list of clients and the messages will be displayed to the user.

4.2 Internodal (Radio-to-Radio) Protocol

The internodal protocol is a series of rules established for the communication between radio nodes. This, along with the *DigiMesh* protocol, forms the backbone of the whole communication process in the system.

The protocols can be divided into multiple sections: the broadcast / ping protocol, client and node entry protocol, client and node disconnect protocol, the message reception protocol, and the buffered token system. The most fundamental of these protocols that control the broadcasting of message and information are the buffered token system and the broadcast / ping protocol. These two sections will be described thoroughly in their individual parts below. As the other protocols have a more general and less sophisticated implementation, their code and description is put under section 9.1 of the appendix. These include the code for handling the GPS serial message and the operation of turning the device on and off using an internal digital interrupt.

4.2.1 Broadcast (Ping) Protocol

A key standalone protocol used for the communication between nodes is the use of "pings" to broadcast messages and client information throughout the whole network. This allows the nodes to obtain information about all of the clients connected in the wide network and any messages available for the clients in the node. It is the foundation of the whole communication process between nodes.

Only a single protocol is used for communication between the nodes to prevent the over-crowding of data and traffic in the channel; which improves the reliability of the network considerably.

This protocol has two main subcomponents, the client information component, and the message component. The client information component sends a list of all available clients connected to the local node. Denote "clientInfo_N" as the information about the N-th local client in the node, the protocol which describes this client completely is given by:

$$clientInfo\_N = client\_name : \{client\ \ name\}/client\_ip : \{client\ \ ip\ \ address\}/client\_mac\ \{client\ \ mac\ \ address\}$$

Then, the client information component of the ping protocol is given by:

$$ping\_mac : \{source\ \ mac\ \ address\}/client : \{clientInfo\_1\}/client : \{clientInfo\_2\}/\ .../client : \{clientInfo\_N\}$$

Attached behind this client information component is a list of all messages from all of the clients within the node. As mentioned previously, all messages sent by clients within the node is temporarily held

in a message queue and is sent out when it is the node's turn to broadcast (described under Buffered Token System below) to reduce network traffic. Denote "messageInfo_N" as the N-th message sent by clients within the node. Then, the protocol to describe this message is given by:

$$messageInfo\_N = /message : \{message\}/target\_ip : \{destination\ \ IP\ \ address\}/target\_mac : \{destination\ \ mac\ \ address\}/source\_id : \{source\ \ id\}$$

This message protocol contains a complete information of both the location of the target client within the network as well as the information of the client who sent the message. Since multiple messages are attached within this whole protocol, the whole protocol will then be:

$$messageInfo\_1/messageInfo\_2/.../messageInfo\_N$$

Combining both the client information component and the message component, the whole ping / broadcast protocol has the form:

$$ping\_mac : sourcemacaddress/client : clientInfo\_1/client : clientInfo\_2/.../client : clientInfo\_N /messageInfo\_1/messageInfo\_2/.../messageInfo\_N/token : token/finish\_protocol$$

The 'token' and 'finish_protocol' will be described in the *Buffered Token System* and *Overcoming the Arduino Serial Buffer Limit section* (9.1.6).

The code to implement this broadcast is shown below:

```
/* Broadcast Protocol */
// broadcast has general form ping_mac:(source mac)/client:(client info 1)/client:(client info 2)/...
void broadcastPing(){
  String broadcast = "ping_mac:" + currMacAddress;
  for(int i = 0; i < currClientIndex; i++){
    broadcast = broadcast + "/client:" + clientArray[i];
  }
  // refreshes the local network clients
  for(int i = 1; i < currClientIndex+1; i++){
    networkMacAddress[0][i] = clientArray[i-1] + "/client_mac:" + currMacAddress;
  }
  for(int i = currClientIndex+1; i < 11; i++){
    networkMacAddress[0][i] = "";
  }
  for(int i = 0; i < currMessageSendIndex; i++){
    broadcast = broadcast + "/" + messageSendList[i];
    messageSendList[i] = "";
  }
  // Tokens of form: 1-2-3-4-5-....-N-0
  currMessageSendIndex = 0;
  currToken = currToken.substring(2) + "-" + currToken.substring(0,1); // moves token forward
  broadcast = broadcast + "/token:" + currToken + "/finish_protocol";
  Serial1.println(broadcast);
}
```

This long string from the protocol will then have to be parsed during the reception of the pings to separate it into individual client or message information used by the server. This will be described by the individual sections where they are used below.

4.2.2 Buffered Token System

The radio mesh network is limited in that the nodes cannot transmit messages randomly as it wishes. If multiple nodes broadcast at any one time, the messages from these nodes may collide upon reception, causing corruption and loss of data. Though the likelihood of collision in small networks is small, as the network grows in size, the probability of collision increases considerably, making this notion of "transmit as you please" unscalable and unreliable.

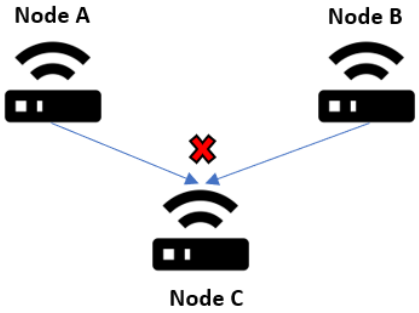


Figure 3: Collision of transmitted data

A solution to this is the buffered token system. It is a procedure used within the network which allows only a single node to transmit / broadcast messages at any one time whilst the others wait and listen. This allows for a collision-free and reliable communication channel with a precisely moderated traffic. The procedure is analogous to a token ring system, the main difference being the topology of the network. The radio network remains a meshed (and not a ring) network but utilizes tokens as a means of controlling the order of broadcasting nodes. As explained previously, the standalone protocol for communication between the radio modules is the ping protocol. This is of the form:

$$ping\_mac : \{sourcemacaddress\}/client : \{clientInfo\_1\}/client : \{clientInfo\_2\}/.../client : \{clientInfo\_N\}$$
$$/\{messageInfo\_1\}/\{messageInfo\_2\}/.../\{messageInfo\_N\}/token : \{token\}/finish\_protocol$$

At the end of every broadcast from a node, a token is sent in conjunction. The token is a string of numbers with dashes in between, and have the general form:

$$Token : n_1\text{–}n_2\text{–}n_3 - \ldots - n_N\text{–}0$$

Where n_i represents the node ID associated with the i-th node. Every node in the network will have a unique node ID associated with it. For example, let the network contain 5 nodes with ID 1, 2, 3, 4, and 5. The token will then have the form:

$$Token : 1\text{–}2\text{–}3\text{–}4\text{–}5\text{–}0$$

Note it does not have to be in an increasing order, it will depend entirely on the order of the nodes joining the network. Another example of token for the same network is:

$$Token : 5\text{–}3\text{–}2\text{–}4\text{–}1\text{–}0$$

The 0 token is a special token which is always present in the token system. Its purpose will be described later.

Upon receiving the pings, each node will parse this token and store it in a global variable called "currToken". The procedure then is as follows. The node will read the first node ID in currToken and compare it with its own ID. If it is equal to its own ID, it will immediately broadcast its own ping to

the network, with the token moved forward one step. For example, if the token received by all of the nodes is:

$$Token : 1\text{–}2\text{–}3\text{–}4\text{–}5\text{–}0 \quad (1st\ \ step)$$

All of the other nodes with ID 2, 3, 4, 5 will see that its own ID does not match the first ID in the token (which is 1), so they will wait and not broadcast. Node 1 will receive this token, see that the first ID matches its own, and broadcast its own ping with the token modified to move forward one step, and the 1 token placed at the back.

$$Token : 2\text{–}3\text{–}4\text{–}5\text{–}0\text{–}1 \quad (2nd\ \ step)$$

The procedure then continues, with Node 2 receiving the token and broadcasting its own ping with a modified token of:

$$Token : 3\text{–}4\text{–}5\text{–}0\text{–}1\text{–}2 \quad (3rd\ \ step)$$

This allows all of the nodes to take turn broadcasting its pings (which contain both network information and messages), which completely eliminates the possibility of collision whilst maintaining speed and scalability.

An important issue that arises from a token system is the possibility of the token information being lost. What happens if the token is 1–2–3–4–5–0 and Node 1 disconnects from the network? A mechanism used to deal with this situation is to have a timer. If all of the other nodes do not receive a broadcast from node 1 within a given time interval (5 seconds), the global variable currToken will be moved one step forward automatically, whilst simultaneously deleting the idle token ID (the token now becomes 2–3–4–5–0). This will allow the token system to continue forward, even when a node disconnects in the middle of transmitting data, hence eliminating the possibility of a lost token (which will freeze the system).

Lastly, there is the challenge of new nodes attempting to join the network. The naive way to handle new nodes is to immediately broadcast their information as they are powered on. However, this may cause collision of data with any nodes transmitting at that time instant (i.e. the node whose ID is the equal to the first token ID). To overcome this problem, we use the 0 token. This is a special token that does not describe any one node. If the token iteration is at the 0 token, a timer is used to wait and listen for any new nodes that want to join the network (over a time interval of 3 seconds). At the 0 token, no node in the network will be transmitting data (they will all be listening for broadcasts), and any new nodes wishing to join the network will be able to broadcast their information to the network.

For example, let node 6 be a node that wishes to join a network with 5 nodes. It will first wait until the 0 token is reached:

$$Token : 3\text{–}2\text{–}0\text{–}1\text{–}4\text{–}5 \quad (1st\ \ step)$$

$$Token : 2\text{–}0\text{–}1\text{–}4\text{–}5\text{–}3 \quad (2nd\ \ step)$$

$$Token : 0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2 \quad (3rd\ \ step)$$

At this point, all other nodes are quiet, and node 6 can broadcast a new token with its own ID in the list.

$$Token : 6\text{–}0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2 \quad (4th\ \ step)$$

Now, node 6 is part of the token list, and is has successfully joined the network.

Another issue that may arise is, what happens if multiple nodes wish to join the network at the same 0 token? To resolve this issue, all of these new nodes at the 0 token will be given a pseudo random number between 0 and 3000 (0 and 3 seconds), which is the time interval of the 0 token. This represents the time after the token reaches 0 at which these new nodes broadcasts. Through this system, there will be an extremely small probability that any two new nodes within this small time interval will collide.

For example, say node 6 and node 7 wishes to enter the network with 5 nodes at the same 0 token, and the pseudo random number attached to node 6 is 743 (so 0.743 seconds) whilst node 7 is 2300 (2.3 seconds). Then, the procedure follows similarly as before, they will wait for the 0 token, and broadcast after a certain pseudo random time interval.

$$Token: 2\text{–}0\text{–}1\text{–}4\text{–}5\text{–}3 \quad (1st\ \ step)$$

$$Token: 0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2 \quad (2nd\ \ step)$$

$$Token: 6\text{–}0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2 \quad (3rd\ \ step)$$

$$Token: 0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2\text{–}6 \quad (4th\ \ step)$$

$$Token: 7\text{–}0\text{–}1\text{–}4\text{–}5\text{–}3\text{–}2 - 6 \quad (5th\ \ step)$$

As can be seen, node 6 and node 7 has successfully joined the network using this procedure.

In conclusion, the buffered token system provides a powerful, robust, and reliable mechanism to transmit information throughout the network, whilst also maintaining its speed and scalability (the delay between each non 0 token is negligible, hence adding more nodes will not affect its speed considerably, which allows for scalability).

The code to implement the buffered token system can be observed in section 9.2.2 of the appendix. The segment which involves the token being pushed forward upon broadcast is given under the "broadcastPing()" function described under the *Ping Protocol* section:

```
currToken = currToken.substring(2) + "-" + currToken.substring(0,1); // moves token
    forward
```

4.3 iOS Application

Most of the user interface was implemented as a *iOS App*. This ensured that the interface is **user-friendly** and **flexible**. The *App* features the following functionalities:

- **Registering**: when the App is first launched, the user is asked to enter his or her credentials. A unique id is assigned to the user, that can now join any compatible network
- **Managing Users**: a table shows the users online, by tapping on one entry, the specific conversation is opened.
- **Chatting**: a text-message environment allows the user to exchange private messages with the selected recipient
- **Monitoring Locations**: a map shows the real-time location of the users connected to the network

The reader is encouraged to refer to Section 9.3 for the full *App* source code.

4.3.1 User Registration

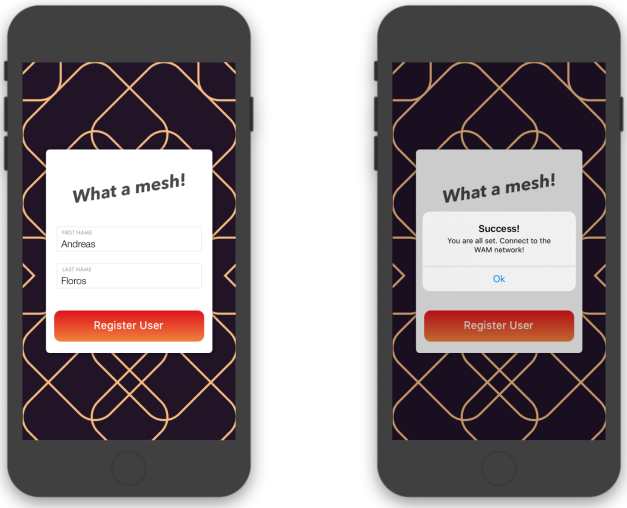
When the application is opened for the first time, the user is prompted to insert their credentials. The program checks for the validity of the name and then a timestamp *ID* is created: this code will later be used to uniquely identify each user.

The unique identifier has to be generated with no internet connection. To do so, a string composed of the date and time followed by a random sequence of numbers is used as *ID*. This makes the event of two identifiers being equal very low. The code used to generate the identifier is the following:

```
let randomNumber = Int.random(in: 0 ..< 10000) // Generate random number

let today_string = String(year!) + String(month!) + String(day!) + String(hour!) + String
    (minute!) + String(second!) + String(randomNumber) // Append random number to date-
    time
```

The program also saves the user's name, surname and *ID*. Therefore, this information can be retrieved automatically when the *App* is launched, and the *AppDelegate* automatically assigns the *ConversationViewController* (the view that holds the list of online users and conversations) as the entry point of the program:

```
UserDefaults.standard.set(newId, forKey: "USER_ID") // Save ID
UserDefaults.standard.set(firstName.text!, forKey: "USER_FIRSTNAME") // Save first name
UserDefaults.standard.set(lastName.text!, forKey: "USER_LASTNAME") // Save second name
```



(a) User inputting data in the form

(b) The user is registered

Figure 4: Screenshot of *FirstLaunchViewController.swift* view. This view controller is prompted during first launch

4.3.2 Connection to the Node

The phone connects to the node via *Wi-Fi* communication. Communication between the App and the node happens through a series of *HTTP GET* requests made by the application. The App then receives the information from the node in form of *JSON*, as described in Section 4.1.

The *HTTP* request is triggered at a constant interval of 3 seconds by a system timer:

```
Timer.scheduledTimer(timeInterval: 3, target: self, selector: #selector(refreshUsers),
    userInfo: nil, repeats: true)
```

Every time the timer is triggered, the function `refreshUsers()` is called:

```
@objc func refreshUsers() {
    print(self.available_users)

    if !connected {
```

```
        performHandshake()
    }

    let requestString = "http://\(self.address)/update_data"
    Alamofire.request(requestString).responseJSON(completionHandler: { response in
        if let json = try? JSON(data: response.data!) {
            print(json)
            self.parseUsers(json: json)
        } else {
            print("Error in JSON")
        }
    })
}
```

If the connection to the node was not established yet, the program performs an *handshake*: the user credentials are sent to the node, that links them with the IP address of the client performing the *handshake*.

After the handshake is performed, the App sends a request informing the node to transmit the data, which is processed by `parseUsers()`:

```
func parseUsers(json: JSON) {
    for message in json["Messages"].arrayValue {
        let sender_id = message["Source ID"].stringValue
        let message_text = message["Message"].stringValue

        self.addMessage(message_text: message_text, user_id: sender_id)
    }

    for user in json["Data"].arrayValue {
        let user_id = user["ID"].stringValue
        let user_name = user["Name"].stringValue
        let location = user["Location"].stringValue

        let latlon = location.split(separator: ";", maxSplits: 1)
        guard let lat = Float(latlon[0]) else { return }
        guard let lon = Float(latlon[1]) else { return }

        insertUser(user: ParsedUser(name: user_name, ID: user_id, lat: lat, lon: lon))
    }
}
```

The function parses the incoming *JSON* and exctracts: users credentials, any new incoming message, the locations coordinates and any distress signal. All the parsed information is held in an array of `Users`. The list of available users is then displayed in the form of a `UITableView`:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
    UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "StandardCell", for:
    indexPath)

    cell.textLabel?.text = available_users[indexPath.row].name
    cell.detailTextLabel?.text = available_users[indexPath.row].messages.last

    return cell
}
```

Each cell contains the name of the user and the latest received message from the considered contact.

4.3.3 Exchanging Messages

Once one of the available users is selected, the *ConversationViewController* is opened. This view utilizes the `MessengerKit` pod to display a *iMessage* like interface. The data displayed by this view is updated by the *ChatsViewController* that periodically parses the data coming in from the node:

```
if let vc = self.fwdVC, let id = self.selected_user_id {
    if user_id == id {
        vc.id += 1

        let body: MSGMessageBody = (message_text.containsOnlyEmoji && message_text.count
    < 5) ? .emoji(message_text) : .text(message_text)

        let message = MSGMessage(id: vc.id, body: body, user: vc.tim, sentAt: Date())
        vc.insert(message)
    }
}
```

When the user wants to send a message, another *HTTP* request is made, and the node is informed to send the message:

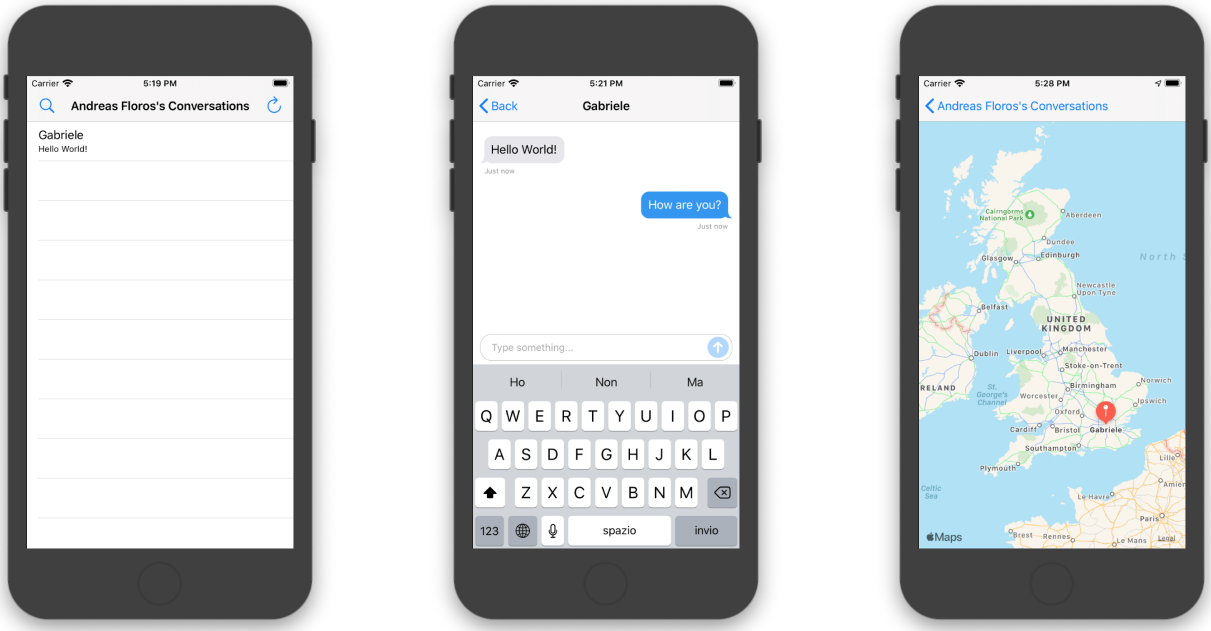
```
func sendMessage(message: MSGMessage) {
    if let text = message.body.rawValue as? String {
        var requestString = "http://\(self.address)/message:" + processMessage(inString:
    text) + "/source_id:" + self.this_user!.ID + "/target_id:"
        requestString = requestString + self.recipient!.ID

        print("Request: " + requestString)

        Alamofire.request(requestString)
    }
}
```

4.3.4 Real-time Map

The parsed user locations are also passed to *MapViewController*. This view adds annotations (pins on the map) for each user online, according to their location:

```
func showLocations() {
    for user in self.users {
        let annotation = MKPointAnnotation()
        annotation.title = user.name
        annotation.coordinate = user.location
        self.map.addAnnotation(annotation)
    }
}
```

The map is saved on the phone's memory. Hence the App is fully functional without any internet connection. The resulting view is given below: Figure 5b and 5a show the conversation selection and the message exchange. Figure 5c shows the real time map.



(a) List of available users (b) Private conversation (c) User location on map

Figure 5: Screenshots showing data being parsed by the *App*

4.4 Hardware

After deciding on the implementation of a decentralized system we had to consider the most appropriate hardware to fit the budget and meet computing power specifications for the network. We utilized the *Arduino WiFi MKR 1010* as microcontroller since it consumes less power than a *Raspberry Pi* and the extra computational capabilities of the *Raspberry Pi* were not needed. Furthermore, the *Arduino* has both a WiFi module and a *Lipo* battery management system integrated on the board at its affordable price. We choose the *XBee* modules over *LoRa* technology for its built-in support for mesh network topology. We also chose a GPS module capable of *UART* transmission.

4.4.1 Assembly

Everything was first assembled on a breadboard as a prototype. Then, a PCB was considered to implement our circuit (Figure 6), but due to supply shortages and high manufacturing prices, we opted for a *strip board*. Virtual serial communication ports were utilized to interface the GPS module with the Arduino. We drived the button LEDs in the with *NPN BJTs*, since the *Arduino* can only supply 7mA and the LEDs need a minimum of 20 mA to be bright. We did not add current limiting resistors since the LEDs are rated for 5 V. We configured the buttons to be active high by attaching pull down resistors.
After building the circuit on strip boards we measured the dimensions and the enclosure was designed to accommodate the circuit.

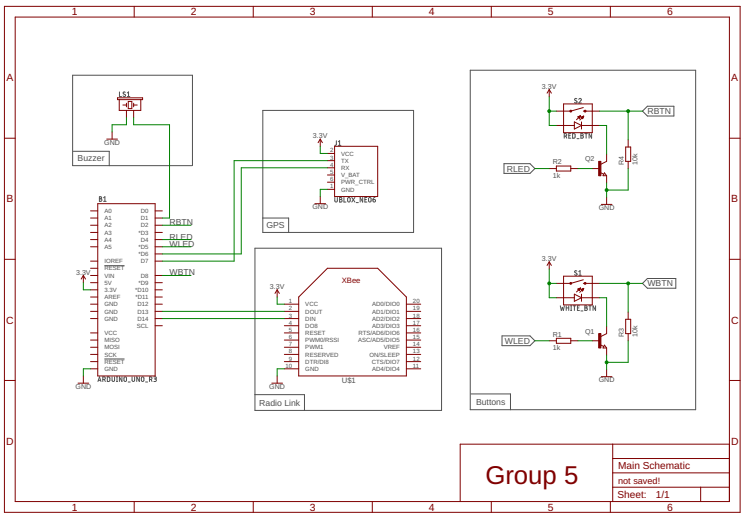


Figure 6: Node's circuit diagram. Note that the *LiPo* battery is not included in the diagram

4.4.2 Enclosure

The enclosure for our product was designed in *Autodesk Fusion 360* with the intention to be 3D printed using *PLA*. While designing the case the design criteria were to be kept in mind: the ability to charge the battery/upload new code without needing to remove the circuitry; a method of keeping the circuitry stable within the case so that it would not shake around and get damaged; sufficient space so that we could fit all the circuitry inside but still be small enough to be carried around in a large pocket or small bag; a lid that could be removed for occasional maintenance, but would stay firmly attached otherwise; holes of a reasonable size to allow the buttons to smoothly interface with the case and to allow the sound of the internal buzzer to be heard.

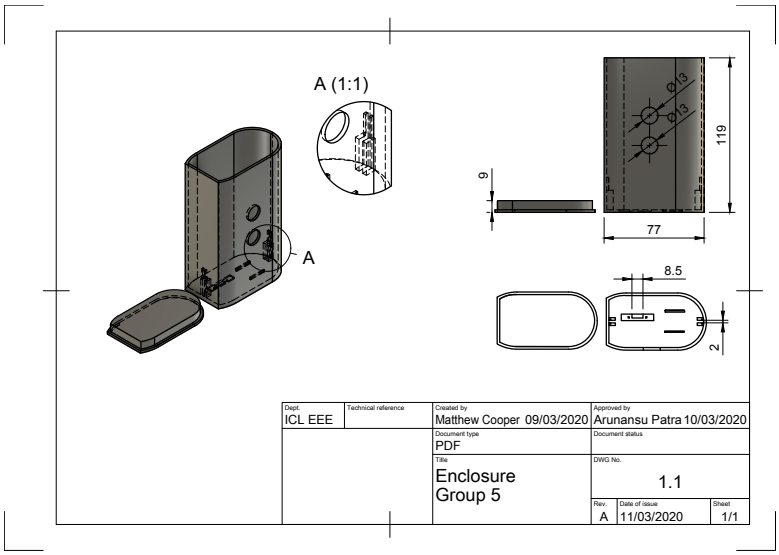


Figure 7: CAD drawing of the enclosure and lid. Generated with *Fusion 360*

To charge the battery/upload the code, a hole was added to the bottom of the case, where the micro USB cable could be inserted to line up correctly with the port on the other side of the case. To keep the circuit stable two slots were added on opposite sides of the case so that the board could easily

slide in and be supported within the case. These slots were positioned such that when the board was inserted there would be room for the other, off-board components such as the buzzer and the battery. The case ended up having approximate dimensions of 120x77x44mm, which we found could fit into large pockets and bags. The lid was a simple design that relied on the friction generated by being inserted into the box. This meant the lid could be removed with a little effort, but otherwise it would stay in position perfectly and would not be loose. The holes for the buttons were designed with a recess that allowed the back of the buttons to be flat against the side of the case and some small slits were cut at the base, near the USB hole, so that the sound from the buzzer would be able to get out of the case.

5 Project Management

As discussed in the preliminary report, as soon as we knew the problem we wanted to tackle, we started our research and came up with a work breakdown structure (Figure 8). From there, we produced a *Gantt Chart* (Section 9.5) that we have been updating throughout the project. We have been meeting once a week to discuss the advances in the project, the next tasks to do, and dividing them between members. We have mainly divided the group in technical stream and non-technical stream, as we will discuss further in the next part. We communicated through a *WhatsApp* group and mainly worked in the EEE labs.

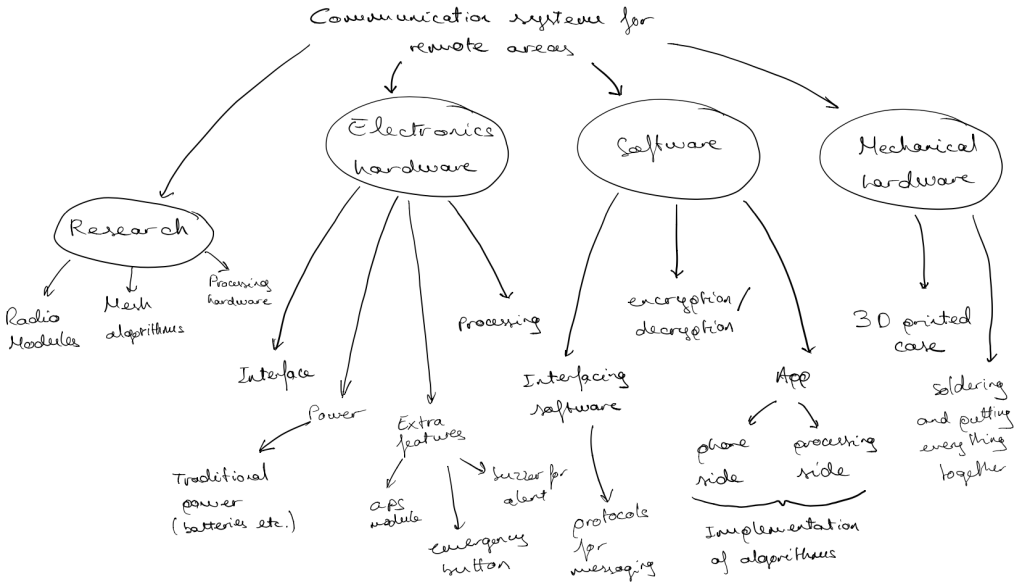


Figure 8: Work breakdown structure

5.1 Workflow and Division of Tasks

As discussed in the previous part, we had some members working only on the technical side (writing protocols, coding the App, and soldering), and other members in less technical tasks (research, project management, budgeting, and creating new ideas for the project). We participated to the *Imperial IC Hack* hackathon, where we made a lot of progress on the project and got a second prize for the *best hack for the community*. This was very helpful as it propelled our project forward and gave us a more thorough insight to the scope of our project and how much time we have to dedicate to it.

5.2 Cost of the Project

The total cost of the project was £305, with each device costing £81.65, well below the average of satellite phones, which go for an average of above £500. So we have met the objective of reducing costs compared to satellite phones. This is a breakdown of the cost of our project:

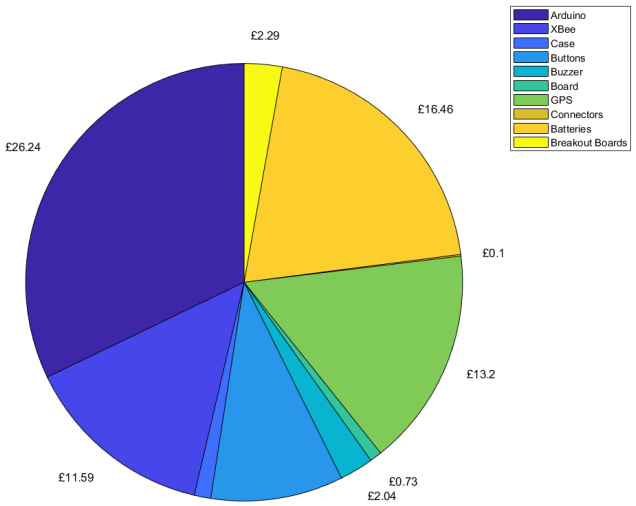


Figure 9: Pie chart showing cost of components

The current cost of the radio modules are just £11.59, so we could easily increase to a longer range radio for a price still lower than satellite phones.

6 Extra Features of Our Product

In addition to our core communication system, we decided to add some extra features to improve the capabilities of our product. Each of the subsections is divided into: how we implement it, what advantages it brings, its cost, and its weak points.

6.1 GPS Module

There are different ways to implement geolocation. First of all, it can be done using the phone's satellite location. Our radio module is connected to the phone via the interfacing App, so we could write a function that uses this location at all times, or at given moments. The problem with this implementation is that we are always depending on the phone, and the whole point of the GPS module is to be able to send a location in case of an emergency, and we don't want to depend both on the radio module and the phone. The second possible implementation uses an external GPS module. We found the *U-blox NEO 6 GPS* module, which uses *UART* serial communication, and we can declare an extra serial communication port in the *Arduino* to enable this communication. A big advantage is that it allows the users to be located in emergencies, and locations can be shared between users. There is no cost of using the phone's GPS, while the cost of the *U-blox NEO 6* is just below £15. The only disadvantage is the added weight and complexity, which does not outweigh the proposed benefits and low costs. We implemented this feature using the GPS module because it brings reassurance to the users who will now not only be dependent on their phones in emergencies.

6.2 Emergency Button

The implementation is very basic: we simply need a button connected to our circuit and an extra bit of code to send a the distress signal. One of the objectives of our product is to improve people's security in remote areas by providing access to a communication network where they can ask for help if needed. If the phone battery runs out, the node can still send the user's location and a distress signal to all connected nodes. Those who receive these emergency messages can then call authorities and provide help to this person. The downside of this solution is only the addition of a button in the case which could get damaged in case it falls or gets wet through the extra opening. We will implement this feature because the advantages outweigh the costs by much. In the future, the button may directly alert authorities instead of sending a message to all the other users, improving the response time for those in emergencies.

6.3 Buzzer for Alert

Some emergencies may be life-threatening, so instead of relying on people to check if they have any messages, a loud buzzer can indicate an emergency. The implementation is very straightforward and is explained in section 9.1.7. The advantage, as highlighted before, is a faster response to an emergency situation, and it costs only £1. In terms of disadvantages, it only slightly adds more weight. We implemented it because it is simple and very cheap.

6.4 On-Off Button

Lastly, an important functionality in the node is an on-off button. This is important as the node drains a large amount of current when it is on, especially from the radio module, the Wi-Fi module, and the GPS. For this reason. It is important to be able to switch the device off (a deep sleep mode) to minimize the power consumption and increase battery duration, which is essential in off-grid environments. Its implementation is thoroughly described in section 9.1.8.

7 Future Work

We are satisfied with the time management during the whole length of the project: we met all the deadlines and completed the scheduled tasks, including all the extra features and refining the protocol. If we continued with the project now, the next steps would be to continue implementing extra features and add some wearable features. As a result, we would be able to target our product to our audience more effectively.

7.1 Direct connection with emergency services

The idea is that the location of the user in distress can be sent directly to an emergency response team. This connection can only be done if a node in the mesh network has access to a normal telecommunication network. The implementation would not be simple: we would have to define another protocol and use the phone connected to that device to send a signal to the emergency response team. This would allow a faster response from a medical or rescue team. Furthermore, it provides privacy for the user as their location is not sent to all the other users. The downside of this is the difficulty behind this connection, and that we are using another user's phone to send the message to the emergency response team. Users should agree to this before using our modules.

7.2 LCD for Battery Status

Implementing this feature would require a battery display, which costs about £10. To measure the battery level, we would measure the voltages at full charge and no charge (ie. as we are using 3.7 V batteries, the threshold voltages would go from 4.2 V (100% battery) to 3.0 V (0% battery)) and study the rate of decline. Then we would connect the battery output to one of the analog ports of the Arduino and display a different percentage depending on the voltage sensed by the Arduino. Optimal battery management was a crucial part of our project. Access to a power source may be limited in remote areas, so we need a long battery life and an indicator to show the remaining battery. On the other hand, a display is quite difficult to incorporate into the casing and adds a significant weight.

7.3 Heart rate tracker

We could buy a heart rate tracking system, as implementing a heart rate tracker from circuit level would be too complex and is not the objective of our project. In terms of advantages, it can be connected to the emergency protocol, so that if the user has a cardiac problem, it would send an emergency message. It could also be used to calculate the physical activity of our hikers and climbers, and display such information in the App. The cost is very high: trackers with memory can cost up to £70, and most users of our product will not find it very useful. The implementation at systems level is also not simple, and it adds extra weight.

7.4 Route Calculator Using GPS

The idea behind this feature is for hikers to be able to see the route they have completed, the total distance and their current altitude. We already have the GPS module, so recording the position every five minutes and using downloaded maps would allow us to approximate the route taken. For the altitude some device like a barometer would have to be bought and interfaced with the Arduino and the App. In terms of advantages, a lot of useful information is provided to the users about their routes, and a predicted route can even be extrapolated. The routes, altitude, and the extrapolated information would also be displayed in the App. In terms of disadvantages, it is quite difficult to interface to the App with the maps and the code would be quite difficult to write. Furthermore the selected GPS module does not provide altitude, and either another module has to be chosen or a barometer has to be added.

8 Conclusion

Overall, this project was a true success for us. We transformed our core idea that we had from the beginning of the project into a physical, astonishingly reliably working prototype system. On top of that, we added useful features that we didn't originally think of. Of course, we have learnt most from facing new challenges in the course of the project. Furthermore, we have applied our theoretical knowledge from many of our courses during this project. From Communications Systems through Software Engineering to Analogue Electronics, our project required us to gather all of our knowledge. With our prototype we have laid a foundation to our solution to the problem of communication in remote areas.

9 Appendices

9.1 Appendix A - Additional Protocols

9.1.1 Node Entry Protocol

The node entry protocol is a simple protocol used when a new node wants to enter the mesh network. The actual entry procedure which allows the new node to be a part of the network and begin broadcasting is explained under the *Buffered Token System*. This section will simply explain what is being broadcasted by the new node as it first enters the network.

When a new node enters the network, it must provide details of its MAC Address, as well as the IP Addresses of any local clients connected to it. This is done in a single string (ping) that is broadcasted to the whole network, containing all of this information. As explained under the Ping Protocol above, the information of the client is stored with the protocol:

$$clientInfo\_N = client\_name : \{client\ \ name\}/client\_ip : \{client\ \ ip\ \ address\}/client\_mac\ \{client\ \ mac\ \ address\}$$

And, the client information component of this protocol, which contains the information of all of its local clients is given by:

$$ping\_mac : \{source\ \ mac\ \ address\}/client : \{clientInfo\_1\}/client : \{clientInfo\_2\}/\ .../client : \{clientInfo\_N\}$$

The source MAC Address represents the MAC Address of the node broadcasting the ping. After broadcasting this message, other nodes in the network will receive and parse this information, storing the information of the new node and all of its local clients in a 2-dimensional array (for more efficient lookup complexity). This information will then be passed on to the client, as described under *Network Update and Message Reception Protocol* in the *Client-Server Communication section.*

However, we must be careful that this new node does not broadcast at the same time as other nodes in the network, which may cause the collision and corruption of data. This is dealt with under the *Buffered Token System* section below.

9.1.2 Node Disconnect Protocol

The network must be able to identify nodes that have disconnected from the network and notify all other nodes of this occurrence. However, as nodes that have disconnected are unable to broadcast this event to the wider network, such process of two-way "handshake" or agreement between the nodes are not possible to identify that a node has left the network.

Instead, the protocol uses a timeout procedure. Each node, at every given time interval, will send a broadcast (or a ping) to the whole network to notify that it is still present in the network. The protocol to detect a node that have disconnected is then quite straightforward. Each node will keep a parallel lookup table of all the nodes that are present in the wide network as well as a timer associated to each node. Whenever a node pings that they are still in the network, the timer for that node in the lookup table is reset to 0. However, when any one node disconnects and stops sending pings, their associated timer in the other nodes will continue to increase. As it reaches above a certain value, the node will be terminated from the list of nodes in the network lookup table, indicating that the node has disconnected from the network.

The code required to implement this is given as:

```
1 if(macTimer%20000 == 0){
```

```
    for(int i = 1; i < currMacIndex; i++){ // don't check index 0, since it is current
    device
      networkMacAddressTime[i] = networkMacAddressTime[i] + 20000; // increments timer
    }
    // check if any mac address is beyond a certain time limit and removes it from
    network
    for(int i = 1; i < currMacIndex; i++){
      if(networkMacAddressTime[i] >= 50000){ // timeout set at around 30 seconds
        for(int j = i; j < currMacIndex - 1; j++){
          for(int k = 1; k < 11; k++){
            networkMacAddress[j][k] = networkMacAddress[j+1][k];
          }
          networkMacAddressTime[j] = networkMacAddressTime[j+1];
        }
        for(int k = 0; k < 11; k++){
          networkMacAddress[currMacIndex-1][k] = "";
        }
        networkMacAddressTime[currMacIndex-1] = NULL;
        currMacIndex--;
      }
    }
  }
```

9.1.3 Client Entry Protocol

As have been described previously, the unique identifier of a node is its MAC Address. However, the unique identifier of a client in the local node is its IP Address. Alongside the lookup table of all the clients and nodes within the mesh network, the node also stores information of all the local clients it is serving in an array.

When a new client joins a network, a unique IP Address is assigned to it. A simple client entry protocol then is to check whether this IP Address already exists in our list of available clients. If it is, then do nothing (in fact, a timer will be used to check for client disconnect, described in the *Client Disconnect Protocol* below). if it is not in the array, then the node will add this information into its list, along with the client's name and its unique client ID. Afterwards, this information will be broadcasted by the node ping to the wide network to notify of a new client.

Other nodes will then parse this information and store this new client information in its own lookup table. It will then serve the information about this new client to all of its own local clients, allowing messages to be sent between them.

9.1.4 Client Disconnect Protocol

The protocol to identify a client that has disconnected from the network is similar to that used in the Node Disconnect Protocol, in which a timeout procedure is used instead of a two-way "handshake" agreement.

A parallel array of timers is used alongside the array of clients (identified by its IP Address) in the local node. Whenever a client requests for information about the network, the timer for that particular client is reset (the client App is programmed to request network information once every 10 seconds).

If the client stops requesting this information (i.e. it has disconnected from the network), the timer for that client will continue to increase until it is above a certain threshold. At this point, the client information will be terminated from the local node. This information will then be broadcasted

by ping to the wider network, and the information of the disconnected client will be removed from every other node in the network.

The code to implement the client timeout in the local node is given by:

```
// --- Checks Local Clients for Timeout ---
  if(macTimer%5000 == 0){
    for(int i = 0; i < currClientIndex; i++){
      clientTime[i] = clientTime[i] + 5000; // increments client timer
      if(clientTime[i] >= 50000){ // timeout set at around 30 seconds
        for(int j = i; j < currClientIndex - 1; j++){
          clientTime[j] = clientTime[j + 1];
        }
        clientTime[currClientIndex - 1] = NULL;
        currClientIndex--;
      }
    }
  }
```

9.1.5 Message Reception Protocol

This section describes the parsing procedure of the received messages given under the *Ping Protocol*, and how individual messages are sent to its associated target client. As described previously, the general form of the message component of this broadcast is given by:

$$messageInfo\_1/messageInfo\_2/\ldots/messageInfo\_N$$

Where messageInfo_N describes the N-th message in the message queue sent by clients within the node. This in itself has the protocol:

$$messageInfo\_N = /message:\{message\}/target\_ip:\{destination\ IP\ address\}/target\_mac:\{destination\ mac\ address\}/source\_id:\{source\ id\}$$

As all nodes in the network will receive this ping, it is essential then to identify which messages belong to which node, and which messages belong to which clients. Firstly, a filter is used which checks if the message target MAC address is equal to the MAC address of the receiver node. If it is not, this message is simply discarded. If it is, it is stored in a second message queue (note, a message queue is used for both messages being transmitted, and messages received). This is necessary as the clients request this information once every 10s and does not want the message immediately.

After placing all of the messages in this queue, a second filter is used, which checks if the message target IP Address is equal to the IP Address of the client the server is currently serving. If it is not equal, the message is ignored (but not discarded). If it is equal, then this is the correct destination client, and the message is passed on in JSON format to the client.

Since the JSON format has the form:

{"Type": "Users",
"Data":[{"Name":"Name Here", "ID":"ID here", "Location":"Location Here"}],
"Messages":[{"Source ID":"Source ID Here","Message":"Message Here"}],
"Emergency":[{"Location":"Location Here}]
}

The client only requires information of the message itself, as well as the sender client's ID. Hence, after identifying the correct client, the MAC Address and IP Address of the client is no longer needed and is discarded; only the source ID and the message is sent to the client.

After sending the message to the correct client, the message is discarded from the message queue, so no duplicate messages are sent.

9.1.6 Overcoming the Arduino Serial Buffer Limit

One main drawback of using a standalone protocol described in the Ping Protocol is the length of each broadcast. As shown previously, the protocol is given by:

$$ping\_mac : \{sourcemacaddress\}/client : \{clientInfo\_1\}/client : \{clientInfo\_2\}/.../client : \{clientInfo\_N\}$$
$$/\{messageInfo\_1\}/\{messageInfo\_2\}/.../\{messageInfo\_N\}/token : \{token\}/finish\_protocol$$

The main issue with using this protocol is the limitation of the Arduino. The Arduino serial port has a buffer which can at max hold up to 60 bytes of information (equivalent to 60 characters in a string). If the buffer receives more information than this limit, the message will be cut off and the information will be lost. Since each ping broadcast will likely take more than 60 bytes of information, this is a clear issue for our standalone protocol.

The solution then is to read serial data faster than new data is being placed in the serial buffer. To do this, we must introduce the string "finish_protocol" at the end of every ping. This is crucial to indicate the end of a ping broadcast, so the node will continue reading more and more serial data until this end-of-ping indicator is reached. Previously, an asynchronous delay is used to wait for the buffer to be filled before beginning to read the serial data, but this method is extremely unreliable and often leads to corrupted data. Without this indicator, the system will endlessly attempt to read serial data without knowing the end of the ping broadcast.

The code function used to receive serial data using this protocol is given by:

```
void readSerialMessage(){
  int timer = 0;
  if(Serial1.available() > 0){ // continues reading serial message until /finish_protocol
     is detected
    while(incomingMessage.indexOf("/finish_protocol") == -1 && timer < 20000 &&
    incomingMessage.indexOf("\n") == -1){
      while(Serial1.available() > 0){
        char c = Serial1.read();
        incomingMessage.concat(c);
      }
      timer++;
    }
  }
}
```

The timer is present so that the system can break out of the while loop in the event of a glitch / data corruption that prevents the detection of "/finish_protocol" at the end of ping broadcasts.

9.1.7 GPS and Emergency Broadcast

An additional feature to the node is a GPS to track the location of users in the network. This is an important feature in emergency situations, as it allows other users in the network to track the location of the person in need of help. The coordinates of each user is updated and sent to the client, after which it will be displayed in a map to show all users connected in the network.

Additionally, the emergency functionality is added using an external button. This button, if held for three seconds, will begin transmitting emergency signal to all nodes in the network for a duration of 30 seconds. To indicate that the broadcasting process is working, the LED of the button will turn red. This is easily implemented, and the full code segment can be observed in section 9.2.3 of the appendix.

If the button is held for 3 seconds, the boolean variable *broadcastEmergency* will be set to true, which allows the device to begin transmitting emergency signals. This emergency signal will be added into the ping broadcast protocol before the token segment. This is shown below.

$$ping\_mac:\{sourcemacaddress\}/client:\{clientInfo\_1\}/client:\{clientInfo\_2\}/.../client:\{clientInfo\_N\}$$
$$/\{messageInfo\_1\}/\{messageInfo\_2\}/.../\{messageInfo\_N\}/emergency\_broadcast:\{Location\}$$
$$/token:\{token\}/finish\_protocol$$

This addition in the ping broadcast protocol is handled upon the broadcasting process by first checking if the boolean variable *broadcastEmergency* is true:

```
if(broadcastEmergency){
    broadcast = broadcast + "/emergency_broadcast:" + nodeLocation;
  }
```

Upon reception of this emergency broadcast from other nodes, the information about the emergency broadcast (including its location of origin) will be parsed and sent to the client, which in turn will be displayed in a map. In addition to this, if any emergency broadcast is detected by the node, a beeper will be triggered, which will produce a pulsating alarm noise to notify users throughout the network that someone is in need of assistance. This beeping process is easily implemented after detecting an emergency signal by alternating the digital output of the beeper between low and high every 0.25s. This can also be observed in section 9.2.3 of the appendix.

9.1.8 On-Off Device Handler

Lastly, an important functionality in the node is the on-off button. This is important as the node drains a large amount of current when it is on, including a significant portion from the radio module, the Wi-Fi module, as well as the GPS. If it is kept on constantly, the total current that is drained from the battery amounts to approximately 100mA. Given the LiPo battery capacity of 2000mAh, this will only last for 20h given the battery was initially fully charged. To improve upon this, it is important to be able to switch the arduino to low power mode when the node is unused.

For this, the library *ArduinoLowPower* is used, in which the arduino can be switched into deep sleep mode upon holding the on-off button for 3s. The device can be awaken by using a digital interrupt by clicking the on-off button again (the interrupt will be triggered on the falling edge of the on-off pulse). In addition to putting the module to sleep, the Wi-Fi module will also be turned off to further reduce power consumption. The code implementation of this is relatively straightforward, and can be observed below.

```
...
// this interrupt is declared on setup
LowPower.attachInterruptWakeup(onPin, intermediate, FALLING);

// --- Turns Arduino to Low Power Mode ---
  if(digitalRead(onPin) == LOW){
    currentMillis = millis();
    if(!onCounting){
      onCounting = true;
      onTimerMillis = millis();
```

```
    }
    if(currentMillis - onTimerMillis > 3000 && !sleepMode){
      digitalWrite(onLED, LOW);
      sleepMode = true;
      WiFi.end(); // turns Wi-Fi module off
      LowPower.deepSleep(); // switches arduino to deep sleep mode
    }
  }
  else{
    digitalWrite(onLED, HIGH);
    onCounting = false;
    if(sleepMode == true){
      initialStartup(); // restarts the Wi-Fi network after arduino wakes up
      sleepMode = false;
    }
  }
```

By switching the Arduino and Wi-Fi module off, the current consumed by the module is brought down to approximately 10mA, meaning the device can theoretically last 200h in the off state before it needs proper recharging.

9.2 Appendix B - Node Source Code

9.2.1 Full Backend Code

```
/*
  Project Name: Group 5 Client Communication

  Version: 16

  Summary: Handles local communication between different clients using a series of HTTP POST requests (from clients). Messages
           received by the Arduino is temporarily held in a message queue, which is then sent from the Arduino to the local client
           via JSON, whose response content can be detected and parsed by the app.

  Default Web Server IP Address: 192.168.4.1

  Client Protocols: There are three main protocols shared by the arduino and the client for bidirectional communication.
    1. Client Initialization Protocol: /client_name:{your name here}/client_id:{your id here}
    2. Message Sending Protocol: /message:{your message here}/source_id:{source id}/target_id:{target id}
    3. Network Update and Message Reception Protocol: returns JSON information of clients and available messages.
                                                      When receiving the message, it will be in the form:
                                                      message:{message here}/source_id:{source id here}

  Communication Protocol: The main protocol used for the communication system is a token buffer protocol.
                          These tokens are sent along with radio broadcasts and allows only the nodes
```

```
21                          with a valid token to transmit messages. This terminates the
    possibility
22                          of collision during data reception. Tokens have the form
    1-2-3-4-5-....-N-0,
23                          where each number represents a node ID within the network, and
    the 0 token
24                          represents a delay timer to allow new users into the network.
25 */
26
27 #include <SPI.h>
28 #include <WiFiNINA.h>
29 #include <TinyGPS.h>
30 #include <ArduinoLowPower.h>
31 #include <Arduino.h>
32 #include <wiring_private.h>
33 #include <variant.h>
34 #define PIN_SERIAL_RX       (7)                 // Pin description number for PIO_SERCOM
    on 7
35 #define PIN_SERIAL_TX       (6)                 // Pin description number for PIO_SERCOM
    on 6
36 #define PAD_SERIAL_TX       (UART_TX_PAD_2)       // SERCOM pad 2 TX
37 #define PAD_SERIAL_RX       (SERCOM_RX_PAD_3)     // SERCOM pad 3 RX
38
39
40 /* Network Variables */
41 char ssid[] = "TestNet2";         // your network SSID (name)
42 int keyIndex = 0;                 // your network key Index number (needed only for WEP)
43 byte mac[6];
44 int status = WL_IDLE_STATUS;
45 WiFiServer server(80);
46
47 /* Communication Protocol Variables */
48 String currToken = "";
49 String nodeID = "2";
50 unsigned long startMillis;
51 unsigned long currentMillis;
52 bool startTimer = true;
53 bool loneBroadcaster = true;
54 int randomInitialTimer = 0;
55
56 /* Client Protocol Variables */
57 String clientArray[10]={""}; // static array for client information (ip address and name)
     (currently handles 10 clients at max)
58 String messageQueue[50]={""}; // places all incoming messages in a queue, since Arduino
    operation is not asynchronous
59 String networkMacAddress[20][11];
60 String currMacAddress = "";
61 String messageSendList[30]={""}; // protocol is /message:(your message)/target_ip:(
    destination ip)/target_mac:(destination mac)/source_id:(source id)
62 String incomingMessage = "";
63 int networkMacAddressTime[20]; // this is in form client_name:(client name)/client_id:(
    client id)/client_ip:(client ip)/client_mac:(client mac)
64 int clientTime[10] = {NULL};
65 int currClientIndex = 0;
66 int currMessageIndex = 0;
67 int currMacIndex = 0;
```

```
68 int currMessageSendIndex = 0;
69 int macTimer = 0; // counts up and checks all mac addresses
70
71 /* GPS and Emergency Protocol Variables*/
72 Uart gpsSerial(&sercom3, PIN_SERIAL_RX, PIN_SERIAL_TX,PAD_SERIAL_RX , PAD_SERIAL_TX); //
       Create the new UART instance assigning it to pin 0 and 1
73 TinyGPS gps;
74 String nodeLocation = "0.000000;0.000000"; // MUST FILL THIS NEW VARIABLE WITH LOCATION
75 String emergencyLocations[20];
76 bool broadcastEmergency = false;
77 bool buttonCounting = false;
78 bool onCounting = false;
79 bool sleepMode = false;
80 float lat;
81 float lon;
82 int currEmergencyIndex = 0;
83 const int beeperPin = 1; // pin for beeper
84 const int emergencyPin = 2; // pin for emergency button
85 const int onPin = 8; // to switch to low power mode
86 const int emergencyLED = 4;
87 const int onLED = 5;
88 unsigned long startBeeperTimer;
89 unsigned long startEmergencyTimer;
90 unsigned long endEmergencyTimer;
91 unsigned long checkBeeperMillis;
92 unsigned long onTimerMillis;
93
94 /* Access Point Setup */
95 void setup() {
96   //Initialize serial and wait for port to open:
97   Serial.begin(9600);
98   Serial1.begin(9600);
99   gpsSerial.begin(9600);
100  pinMode(emergencyPin, INPUT_PULLUP);
101  pinMode(onPin, INPUT_PULLUP);
102  pinMode(beeperPin, OUTPUT);
103  pinMode(emergencyLED, OUTPUT);
104  pinMode(onLED, OUTPUT);
105  pinPeripheral(PIN_SERIAL_TX, PIO_SERCOM);
106  pinPeripheral(PIN_SERIAL_RX, PIO_SERCOM_ALT);
107  initialStartup();
108  LowPower.attachInterruptWakeup(onPin, intermediate, FALLING);
109 }
110
111 void intermediate(){
112   // do nothing
113 }
114
115 void initialStartup() {
116   // resets all the variables
117   digitalWrite(onLED, HIGH);
118   clientArray[10]={""};
119   messageQueue[50]={""};
120   currMacAddress = "";
121   messageSendList[30]={""};
122   incomingMessage = "";
```

```
  clientTime[10] = {NULL};
  currClientIndex = 0;
  currMessageIndex = 0;
  currMacIndex = 0;
  currMessageSendIndex = 0;
  macTimer = 0;
  startTimer = true;
  loneBroadcaster = true;
  randomInitialTimer = 0;

  checkBeeperMillis = millis();
  Serial.println("Access Point Web Server");
  // check for the WiFi module:
  if(WiFi.status() == WL_NO_MODULE) {
    Serial.println("Communication with WiFi module failed!");
    while (true);
  }
  String fv = WiFi.firmwareVersion();
  if (fv < WIFI_FIRMWARE_LATEST_VERSION) {
    Serial.println("Please upgrade the firmware");
  }
  Serial.print("Creating access point named: ");
  Serial.println(ssid);
  // Create open network. Change this line if you want to create an WEP network:
  status = WiFi.beginAP(ssid);
  if (status != WL_AP_LISTENING) {
    Serial.println("Creating access point failed");
    // don't continue
    while (true);
  }
  // wait 10 seconds for connection:
  delay(10000);
  // start the web server on port 80
  server.begin();
  // you're connected now, so print out the status
  printWiFiStatus();

  /* Generate Mac Address for Arduino */
  WiFi.macAddress(mac);
  for(int i = 5; i >=0; i--){
    currMacAddress = currMacAddress + String(mac[i]);
    if(i !=0){
       currMacAddress = currMacAddress + ".";
    }
  }
  // immediately put current mac address as the first address in the array
  networkMacAddress[0][0] = currMacAddress;
  // Initially flush all Serial data to avoid error in reception
  while(Serial.available() > 0) {
    char t = Serial.read();
  }
  // check if any other nodes present in network
  int checkBroadcastTimer = 0;
  while(Serial.available() == 0 && checkBroadcastTimer >= 5000){
    checkBroadcastTimer++;
    delay(1);
```

```
  }
  readSerialMessage();
  if(incomingMessage.indexOf("/token:") != -1){
    loneBroadcaster = false; // if other tokens present in network, indicate this
    int tokenIndex = incomingMessage.indexOf("/token:");
    int finishIndex = incomingMessage.indexOf("/finish_protocol");
    currToken = incomingMessage.substring(tokenIndex + 7, finishIndex);
  }
  else{
    currToken = nodeID + "-0"; // initialize token for lone broadcaster: X-0
    Serial.println("I am alone");
  }
}

void loop() {
  // --- Turns Arduino to Low Power Mode ---
  if(digitalRead(onPin) == LOW){
    currentMillis = millis();
    if(!onCounting){
      onCounting = true;
      onTimerMillis = millis();
    }
    if(currentMillis - onTimerMillis > 3000 && !sleepMode){
      digitalWrite(onLED, LOW);
      sleepMode = true;
      WiFi.end();
      LowPower.deepSleep();
    }
  }
  else{
    digitalWrite(onLED, HIGH);
    onCounting = false;
    if(sleepMode == true){
      initialStartup();
      sleepMode = false;
    }
  }

  if (status != WiFi.status()) {
    // it has changed update the variable
    status = WiFi.status();
    if (status == WL_AP_CONNECTED) {
      // a device has connected to the AP
      Serial.println("Device connected to AP");
    } else {
      // a device has disconnected from the AP, and we are back in listening mode
      Serial.println("Device disconnected from AP");
    }
  }
  WiFiClient client = server.available();   // listen for incoming clients
  if (client) {
    // obtain client IP in String Format
    String clientIP = "";
    for(int i = 0; i < 4; i++){
      clientIP = clientIP + client.remoteIP()[i];
      if(i!=3){
```

```
235         clientIP = clientIP + ".";
236       }
237     }
238     Serial.println("new client");
239     String currentLine = "";
240     boolean currentLineIsBlank = true;
241     while (client.connected()) {
242       if (client.available()){
243         char c = client.read();
244         Serial.write(c);
245         if (c == '\n' && currentLineIsBlank) {
246           /* JSON Data and Parsing */
247           /*
248            {"Type": "Users",
249             "Data":[{"Name":"Name Here", "ID":"ID here", "Location":"Location here"}],
250             "Messages":[{"Source ID":"Source ID Here", "Message":"Message Here"}],
251             "Emergency":[{"Location":"Location here"}, {"Location":"Location here"}]
252            }
253           */
254           client.println("HTTP/1.1 200 OK");
255           client.println("Content-Type: application/json");
256           client.println("Access-Control-Allow-Origin: *");
257           client.println("Refresh: 2");
258           client.println("");
259
260           // --- Obtains Message From Client ---
261           // clientMessage has form: /message:(your message here)/source_id:(source id)/
    target_id:(target id)
262           int startIndex = currentLine.indexOf("GET /");
263           int endIndex = currentLine.indexOf("HTTP/1.1");
264           String clientMessage = currentLine.substring(startIndex + 5, endIndex);
265           clientMessage.replace("%20", " ");
266           Serial.println(clientMessage);
267
268           // --- Handles Client Request for Updated Data ---
269           if(clientMessage.indexOf("update_data")!= -1){
270             bool deleteComma = false;
271             // attaches all connected users in network to JSON
272             String finalJson = "{\"Type\": \"Users\",\"Data\":[";
273             for(int i = 0; i < currMacIndex; i++){
274               for(int k = 1; k < 11; k++){
275                 // networkMacAddress in form client_name:(client name)/client_id:(client
    id)/client_ip:(client ip)/client_loc:(client location)/client_mac:(client mac)/
276                 if(networkMacAddress[i][k] != ""){
277                   deleteComma = true;
278                   int indexName = networkMacAddress[i][k].indexOf("client_name:");
279                   int indexID = networkMacAddress[i][k].indexOf("client_id:");
280                   int indexIP = networkMacAddress[i][k].indexOf("client_ip:");
281                   int indexLoc = networkMacAddress[i][k].indexOf("client_loc:");
282                   int indexMac = networkMacAddress[i][k].indexOf("/client_mac:");
283                   String clientLocation = networkMacAddress[i][k].substring(indexLoc+11,
    indexMac);
284                   String clientName = networkMacAddress[i][k].substring(indexName+12,
    indexID-1);
285                   String clientID = networkMacAddress[i][k].substring(indexID+10,indexIP
    -1);
```

```
286                 finalJson = finalJson + "{\"Name\":\"" + clientName + "\", \"ID\":\"" +
    clientID + "\", \"Location\":\"" + clientLocation + "\"},";
287               }
288             }
289           }
290           if(deleteComma){
291             finalJson.remove(finalJson.length()-1);
292             deleteComma = false;
293           }
294           finalJson = finalJson + "], \"Messages\":[";
295           // Releases messageQueue to JSON requested by client
296           // messageQueue of form /message:(your message)/target_ip:(destination ip)/
    target_mac:(destination mac)/source_id:(source id)
297           // message output to app has general form /message:(your message)/target_id:(
    target id)
298           // Need to search in client table for this target id and obtain corresponding
     ip and mac address
299           for(int i = 0; i < currMessageIndex; i++){
300             int indexIP = messageQueue[i].indexOf("target_ip:");
301             int indexMac = messageQueue[i].indexOf("target_mac:");
302             String targetIP = messageQueue[i].substring(indexIP+10, indexMac-1);
303             targetIP.replace(" ", "");
304             targetIP.replace("\r", "");
305             targetIP.replace("/", "");
306             if(targetIP == clientIP){
307               deleteComma = true;
308               int indexMessage = messageQueue[i].indexOf("message:");
309               int indexID = messageQueue[i].indexOf("source_id:");
310               String message = messageQueue[i].substring(indexMessage + 8, indexIP - 1)
    ;
311               String sourceID = messageQueue[i].substring(indexID + 10);
312               sourceID.replace(" ", "");
313               sourceID.replace("\r", "");
314               finalJson = finalJson + "{\"Source ID\":\"" + sourceID + "\", \"Message
    \":\"" + message + "\"},";
315               for(int k = i; k < currMessageIndex; k++){
316                 messageQueue[k] = messageQueue[k + 1];
317               }
318               // moves message queue forwards once it is sent to JSON
319               messageQueue[currMessageIndex] = "";
320               currMessageIndex--;
321               i--;
322             }
323           }
324           if(deleteComma){
325             finalJson.remove(finalJson.length()-1);
326             deleteComma = false;
327           }
328           if(currEmergencyIndex != 0){
329             deleteComma = true; // if emergency list is not empty, delete last comma
330           }
331           finalJson = finalJson + "], \"Emergency\":[";
332           for(int i = 0; i < currEmergencyIndex; i++){
333             finalJson = finalJson + "{\"Location\":\"" + emergencyLocations[i] + "\"},"
    ;
334           }
```

```
335             if(deleteComma){
336               finalJson.remove(finalJson.length()-1);
337               deleteComma = false;
338             }
339             finalJson = finalJson + "]}";
340             finalJson.replace("\n", "");
341             Serial.println("final JSON: " + finalJson);
342             client.println(finalJson);
343           }
344           // --- Handles Client Message Transmission ---
345           // send message if it does not contain an automatic "favicon.ico" response from
       web server
346           // messageQueue has the general form /message:(your message)/target_ip:(
      destination ip)/target_mac:(destination mac)/source_id:(source id)
347           if(clientMessage.indexOf("favicon.ico") == -1 && clientMessage.indexOf("
      target_id:") != -1){
348             int indexTargetID = clientMessage.indexOf("target_id:");
349             String targetID = clientMessage.substring(indexTargetID + 10);
350             targetID.replace(" ", "");
351             String targetMac = "";
352             String targetIP = "";
353             // networkMacAddress has form /client_name:(client name)/client_id:(client id
      )/client_ip:(client ip)/client_loc:(client location)/client_mac:(client mac)
354             for(int i = 0; i < currMacIndex; i++){
355               for(int k = 1; k < 11; k++){
356                 if(networkMacAddress[i][k].indexOf(targetID) != -1){
357                   int indexMac = networkMacAddress[i][k].indexOf("client_mac:");
358                   int indexID = networkMacAddress[i][k].indexOf("client_id:");
359                   int indexIP = networkMacAddress[i][k].indexOf("client_ip:");
360                   int indexLoc = networkMacAddress[i][k].indexOf("client_loc:");
361                   targetMac = networkMacAddress[i][k].substring(indexMac+11);
362                   targetMac.replace(" ", "");
363                   targetIP = networkMacAddress[i][k].substring(indexIP+10,indexLoc-1);
364                   break;
365                 }
366               }
367             }
368             Serial.println("targetMac:" + targetMac);
369             Serial.println("targetIp:" + targetIP);
370             if(targetMac != "" && targetIP != ""){
371               // the input clientMessage is /message:(your message)/source_id:(source id)
      /target_id:(target id)
372               // clientMessage output here will now have the same protocol as in
      messageQueue
373               // which is /message:(your message)/target_ip:(destination ip)/target_mac:(
      destination mac)/source_id:(source id)
374               int indexSourceID = clientMessage.indexOf("source_id:");
375               int indexTargetID = clientMessage.indexOf("target_id:");
376               String sourceID = clientMessage.substring(indexSourceID+10,indexTargetID-1)
      ;
377               clientMessage = clientMessage.substring(0, indexSourceID - 1);
378               clientMessage = clientMessage + "/target_ip:" + targetIP + "/target_mac:" +
       targetMac + "/source_id:" + sourceID;
379               clientMessage.replace("\n", "");
380               // if the targetMac is equal to Current Mac Address, we store in local
      messageQueue
```

```
            if(targetMac == currMacAddress){
              messageQueue[currMessageIndex] = clientMessage;
              currMessageIndex++;
            }
            // if target mac is a different node, broadcast it out to other nodes
            else{
              messageSendList[currMessageSendIndex] = clientMessage;
              currMessageSendIndex++;
            }
          }
        }

        // --- Handles New Client to the Local Network ---
        // input is /client_name:(name)/client_id:(id)
        // clientArray  has form /client_name:(client name)/client_id:(client id)/client_ip:(client ip)/client_loc:(client location)
        // clientMessage has the form client_name:(client name)/client_id:(client id)
        if((clientMessage.indexOf("client_name:") != -1) && (clientMessage.indexOf("client_id:") != -1)){
          bool newClient = true;
          int indexID = clientMessage.indexOf("client_id:");
          String clientID = "client_id:" + clientMessage.substring(indexID + 10);
          String clientInfo = clientMessage + "/client_ip:" + clientIP;
          clientInfo.replace(" ", "");
          clientID.replace(" ", "");
          for(int k = 0; k < currClientIndex; k++){
            // if client already exists in list, do not add it again
            if(clientArray[k].indexOf(clientID) != -1){
              newClient = false;
            }
          }
          if(newClient){
            clientInfo = clientInfo + "/client_loc:" + nodeLocation;
            clientArray[currClientIndex] = clientInfo;
            currClientIndex++;
          }
        }

        // --- Refreshes Local Client Connection Status ---
        for(int i = 0; i < currClientIndex; i++){
          if(clientArray[i].indexOf(clientIP) != -1){
            clientTime[i] = 0;
            break;
          }
        }
        break;
      }
      if (c == '\n') {
        currentLineIsBlank = true;
      }
      else if (c != '\r') {
        currentLineIsBlank = false;
        currentLine += c;
      }
    }
  }
```

```
435     // close the connection:
436     client.stop();
437     Serial.println("client disonnected");
438   }
439   macTimer++;
440
441   // --- Handles Incoming Message from Other Nodes ---
442   incomingMessage = "";
443   readSerialMessage();
444
445   // --- Handles Incoming Mac Address Pings ---
446   if(incomingMessage.indexOf("ping_mac:") != -1){
447     int emergencyIndex = incomingMessage.indexOf("/emergency_broadcast:");
448     int tokenIndex = incomingMessage.indexOf("/token:");
449     int finishIndex = incomingMessage.indexOf("/finish_protocol");
450     if(emergencyIndex != -1){
451       endEmergencyTimer = millis();
452       String emergencyLoc = incomingMessage.substring(emergencyIndex+21, tokenIndex);
453       bool found = false;
454       for(int i = 0; i < currEmergencyIndex; i++){
455         if(emergencyLocations[i] == emergencyLoc){
456           found = true;
457         }
458       }
459       if(!found){
460         emergencyLocations[currEmergencyIndex] = emergencyLoc;
461         currEmergencyIndex++;
462       }
463     }
464     startTimer = true; // must reset this if multiple users try to join network
465     currToken = incomingMessage.substring(tokenIndex + 7, finishIndex);
466     Serial.println("Current Token: " + currToken);
467     incomingMessage = incomingMessage.substring(0, tokenIndex);
468     int indexEndMac = incomingMessage.indexOf("/");
469     int indexStartPingMac = incomingMessage.indexOf("ping_mac:");
470     String pingMac = incomingMessage.substring(indexStartPingMac + 9, indexEndMac);
471     bool newMac = true;
472     int indexRefreshMac;
473     for(int i = 0; i < currMacIndex; i++){
474       if(networkMacAddress[i][0] == pingMac){
475         networkMacAddressTime[i] = 0;
476         newMac = false;
477         indexRefreshMac = i;
478       }
479     }
480     if(newMac){
481       indexRefreshMac = currMacIndex;
482     }
483     networkMacAddress[indexRefreshMac][0] = pingMac;
484     if(incomingMessage.indexOf("/client:") != -1){
485       int clientIndexBottom = incomingMessage.indexOf("client:") + 7;
486       int clientIndexTop = incomingMessage.indexOf("client:", clientIndexBottom);
487       int i = 1;
488       // will store in networkMacAddress in the form client_name:(client name)/client_ip
    :(client ip)/client_loc:(client location)/client_mac(client mac)
489       while(clientIndexTop != -1){
```

```
490         String newClientInfo = incomingMessage.substring(clientIndexBottom,
    clientIndexTop)+ "/client_mac:" + pingMac;
491         newClientInfo.replace(" ", "");
492         networkMacAddress[indexRefreshMac][i] = newClientInfo;
493         clientIndexBottom = clientIndexTop + 7;
494         clientIndexTop = incomingMessage.indexOf("client:", clientIndexBottom);
495         i++;
496       }
497       for(int k = i; k < 11; k++){
498         networkMacAddress[indexRefreshMac][k] = "";
499       }
500       int messageIndex = incomingMessage.indexOf("/message:");
501       if(messageIndex == -1){
502         networkMacAddress[indexRefreshMac][i] = incomingMessage.substring(
    clientIndexBottom)+ "/client_mac:" + pingMac;
503       }
504       else{
505         networkMacAddress[indexRefreshMac][i] = incomingMessage.substring(
    clientIndexBottom, messageIndex)+ "/client_mac:" + pingMac;
506       }
507       networkMacAddressTime[indexRefreshMac] = 0;
508     }
509     if(newMac){
510       currMacIndex++;
511     }
512   }
513
514   // --- Handles Client Emergency Broadcast ---
515   if(digitalRead(emergencyPin) == LOW){
516     if(!buttonCounting){
517       buttonCounting = true;
518       startBeeperTimer = millis();
519     }
520     currentMillis = millis();
521     if(currentMillis - startBeeperTimer > 3000){ // must hold button for 3s
522       broadcastEmergency = true;
523       buttonCounting = false;
524       startEmergencyTimer = millis();
525     }
526   }
527   else{
528     buttonCounting = false;
529   }
530
531   // --- Broadcasts Emergency Signal for 30 seconds ---
532   if(broadcastEmergency){
533     digitalWrite(emergencyLED, HIGH);
534     // maybe make LED light up to indicate you are broadcasting emergency?
535     currentMillis = millis();
536     if(currentMillis - startEmergencyTimer > 20000){
537       broadcastEmergency = false;
538     }
539   }
540   else{
541     digitalWrite(emergencyLED, LOW);
542   }
```

```
543
544   // --- Triggers Beeper if Emergency Detected ---
545   if(currEmergencyIndex > 0){
546     currentMillis = millis();
547     if(currentMillis - checkBeeperMillis > 250){
548       checkBeeperMillis = millis();
549       if(digitalRead(beeperPin) == HIGH){
550         digitalWrite(beeperPin, LOW);
551       }
552       else{
553         digitalWrite(beeperPin, HIGH);
554       }
555     }
556   }
557   else{
558     digitalWrite(beeperPin, LOW);
559   }
560
561   // --- Handles Message Inputs from Other Nodes ---
562   // message input in form /message:(your message)/target_ip:(destination ip)/target_mac
       :(destination mac)/source_id:(source id)
563   int messageIndexBottom = incomingMessage.indexOf("/message:");
564   int messageIndexTop = incomingMessage.indexOf("/message:", messageIndexBottom + 7);
565   while(messageIndexBottom != -1){
566     int macIndexStart = incomingMessage.indexOf("target_mac:", messageIndexBottom);
567     int macIndexEnd = incomingMessage.indexOf("source_id:", messageIndexBottom);
568     String targetMac = incomingMessage.substring(macIndexStart + 11, macIndexEnd-1);
569     String newMessage;
570     if(messageIndexTop != -1){
571       newMessage = incomingMessage.substring(messageIndexBottom, messageIndexTop);
572     }
573     else{
574       newMessage = incomingMessage.substring(messageIndexBottom);
575     }
576     messageIndexBottom = messageIndexTop;
577     messageIndexTop = incomingMessage.indexOf("/message:", messageIndexBottom + 7);
578     Serial.println("NEW MESSAGE");
579     Serial.println(newMessage);
580     targetMac.replace(" ", "");
581     currMacAddress.replace(" ", "");
582     if(targetMac == currMacAddress){
583       messageQueue[currMessageIndex] = newMessage;
584       currMessageIndex++;
585     }
586   }
587
588   // --- Handles Token Push and Broadcasts ---
589   if(currToken.startsWith(nodeID)){
590     delay(100);
591     broadcastPing(); // broadcasts that device is still in network
592   }
593   else if(currToken.indexOf(nodeID) == -1){
594     if(currToken.startsWith("0")){
595       if(startTimer){
596         startTimer = false;
```

```
597         randomInitialTimer = random(0, 3000); // random number between 0 and 3000 (0s to
    3s);
598         startMillis = millis();
599       }
600       currentMillis = millis();
601       if(currentMillis - startMillis > randomInitialTimer){
602         currToken = nodeID + "-" + currToken; // places in front of 0, so broadcast
    resets at 0, e.g. X-0-1-2-3-4-5-...
603         broadcastPing();
604       }
605     }
606     if(currToken.indexOf("0") == -1){
607       currToken = "0-" + nodeID;
608     }
609   }
610   // Pushes token forward and deletes idle node if it does not broadcast within 5s
611   else{
612     if(startTimer){
613       startTimer = false;
614       startMillis = millis();
615     }
616     currentMillis = millis();
617     if(currentMillis - startMillis > 5000){ // gives 5 seconds delay before moving token
    forward
618       // Tokens of form: 1-2-3-4-5-....-N-0
619       if(currToken.startsWith("0")){
620         currToken = currToken.substring(2) + "-" + currToken.substring(0,1); // moves
    token forward
621       }
622       else{
623         currToken = currToken.substring(2); // Removes the Idle Token as well
624       }
625       startTimer = true;
626     }
627   }
628
629   // --- Checks Network Nodes for Timeout ---
630   if(macTimer%20000 == 0){
631     for(int i = 1; i < currMacIndex; i++){ // don't check index 0, since it is current
    device
632       networkMacAddressTime[i] = networkMacAddressTime[i] + 20000;
633     }
634     // check if any mac address is beyond a certain time limit and removes it from
    network
635     for(int i = 1; i < currMacIndex; i++){
636       if(networkMacAddressTime[i] >= 50000){ // timeout set at around 30 seconds
637         for(int j = i; j < currMacIndex - 1; j++){
638           for(int k = 1; k < 11; k++){
639             networkMacAddress[j][k] = networkMacAddress[j+1][k];
640           }
641           networkMacAddressTime[j] = networkMacAddressTime[j+1];
642         }
643         for(int k = 0; k < 11; k++){
644           networkMacAddress[currMacIndex-1][k] = "";
645         }
646         networkMacAddressTime[currMacIndex-1] = NULL;
```

```
647         currMacIndex--;
648       }
649     }
650   }
651
652   // --- Checks Local Clients for Timeout ---
653   if(macTimer%5000 == 0){
654     for(int i = 0; i < currClientIndex; i++){
655       clientTime[i] = clientTime[i] + 5000;
656       if(clientTime[i] >= 50000){
657         for(int j = i; j < currClientIndex - 1; j++){
658           clientTime[j] = clientTime[j + 1];
659         }
660         clientTime[currClientIndex - 1] = NULL;
661         currClientIndex--;
662       }
663     }
664   }
665
666   // --- Measures new GPS location --- //
667   // clientArray  has form /client_name:(client name)/client_id:(client id)/client_ip:(
      client ip)/client_loc:(client location)
668   if(macTimer%10000 == 0){
669     findLocation();
670     for(int i = 0; i < currClientIndex; i++){
671       int indexLoc = clientArray[i].indexOf("/client_loc:");
672       String tempClientInfo = clientArray[i].substring(0, indexLoc);
673       clientArray[i] = tempClientInfo + "/client_loc:" + nodeLocation;
674     }
675   }
676
677   // --- Resets Emergency Location Array ---
678   currentMillis = millis();
679   if(currentMillis - endEmergencyTimer > 10000){
680     currEmergencyIndex = 0;
681   }
682 }
683
684
685 /* ADDITIONAL IMPLEMENTED FUNCTIONS */
686 /* Interrupt Handler */
687 void SERCOM3_Handler()
688 {
689   gpsSerial.IrqHandler();
690 }
691
692 /* Broadcast Protocol */
693 // broadcast has general form ping_mac:(source mac)/client:(client info 1)/client:(client
       info 2)/...
694 void broadcastPing(){
695   String broadcast = "ping_mac:" + currMacAddress;
696   for(int i = 0; i < currClientIndex; i++){
697     broadcast = broadcast + "/client:" + clientArray[i];
698   }
699   // refreshes the local network clients
700   for(int i = 1; i < currClientIndex+1; i++){
```

```
701     networkMacAddress[0][i] = clientArray[i-1] + "/client_mac:" + currMacAddress;
702   }
703   for(int i = currClientIndex+1; i < 11; i++){
704     networkMacAddress[0][i] = "";
705   }
706   for(int i = 0; i < currMessageSendIndex; i++){
707     broadcast = broadcast + "/" + messageSendList[i];
708     messageSendList[i] = "";
709   }
710   if(broadcastEmergency){
711     broadcast = broadcast + "/emergency_broadcast:" + nodeLocation;
712   }
713   // Tokens of form: 1-2-3-4-5-....-N-0
714   currMessageSendIndex = 0;
715   currToken = currToken.substring(2) + "-" + currToken.substring(0,1); // moves token
        forward
716   broadcast = broadcast + "/token:" + currToken + "/finish_protocol";
717   Serial.println("PING broadcast: " + broadcast);
718   Serial1.println(broadcast);
719 }
720
721
722 /* Find Current GPS Location */
723 void findLocation(){
724     while(gpsSerial.available()){ // check for gps data
725        if(gps.encode(gpsSerial.read())){ // encode gps data
726          gps.f_get_position(&lat,&lon); // get latitude and longitude
727        }
728     }
729     String latitude = String(lat,6);
730     String longitude = String(lon,6);
731     nodeLocation = latitude + ";" + longitude;
732     Serial.println("nodeLocation: " + nodeLocation);
733 }
734
735 /* Prints WiFi Status */
736 void printWiFiStatus() {
737   // print the SSID of the network you're attached to:
738   Serial.print("SSID: ");
739   Serial.println(WiFi.SSID());
740
741   // print your WiFi shield's IP address:
742   IPAddress ip = WiFi.localIP();
743   Serial.print("IP Address: ");
744   Serial.println(ip);
745
746   // print where to go in a browser:
747   Serial.print("To see this page in action, open a browser to http://");
748   Serial.println(ip);
749 }
750
751 /* Reads Incoming Serial Messages */
752 void readSerialMessage(){
753   int timer = 0;
754   if(Serial1.available() > 0){ // continues reading serial message until /finish_protocol
        is detected
```

```
    while(incomingMessage.indexOf("/finish_protocol") == -1 && timer < 20000 &&
    incomingMessage.indexOf("\n") == -1){
      while(Serial1.available() > 0){
        char c = Serial1.read();
        incomingMessage.concat(c);
        Serial.println(incomingMessage);
      }
      timer++;
    }
  }
}
```

9.2.2 Buffered Token Implementation

```
// --- Handles Token Push and Broadcasts ---
  if(currToken.startsWith(nodeID)){
    delay(100);
    broadcastPing(); // broadcasts that device is still in network
  }
  else if(currToken.indexOf(nodeID) == -1){
    if(currToken.startsWith("0")){
      if(startTimer){
        startTimer = false;
        randomInitialTimer = random(0, 3000); // random number between 0 and 3000 (0s to
    3s);
        startMillis = millis();
      }
      currentMillis = millis();
      if(currentMillis - startMillis > randomInitialTimer){
        currToken = nodeID + "-" + currToken; // places in front of 0, so broadcast
    resets at 0, e.g. X-0-1-2-3-4-5-...
        broadcastPing();
      }
    }
    if(currToken.indexOf("0") == -1){
      currToken = "0-" + nodeID;
    }
  }
  // Pushes token forward and deletes idle node if it does not broadcast within 5s
  else{
    if(startTimer){
      startTimer = false;
      startMillis = millis();
    }
    currentMillis = millis();
    if(currentMillis - startMillis > 5000){ // gives 5 seconds delay before moving token
    forward
      // Tokens of form: 1-2-3-4-5-....-N-0
      if(currToken.startsWith("0")){
        currToken = currToken.substring(2) + "-" + currToken.substring(0,1); // moves
    token forward
      }
      else{
        currToken = currToken.substring(2); // Removes the Idle Token as well
      }
```

```
      startTimer = true;
    }
  }
```

9.2.3 GPS Emergency Implementation

```
  // --- Handles Client Emergency Broadcast ---
  if(digitalRead(emergencyPin) == LOW){
    if(!buttonCounting){
      buttonCounting = true;
      startBeeperTimer = millis();
    }
    currentMillis = millis();
    if(currentMillis - startBeeperTimer > 3000){ // must hold button for 3s
      broadcastEmergency = true;
      buttonCounting = false;
      startEmergencyTimer = millis();
    }
  }
  else{
    buttonCounting = false;
  }

// --- Triggers Beeper if Emergency Detected ---
  if(currEmergencyIndex > 0){
    currentMillis = millis();
    if(currentMillis - checkBeeperMillis > 250){
      checkBeeperMillis = millis();
      if(digitalRead(beeperPin) == HIGH){
        digitalWrite(beeperPin, LOW);
      }
      else{
        digitalWrite(beeperPin, HIGH);
      }
    }
  }
  else{
    digitalWrite(beeperPin, LOW);
  }

/* Find Current GPS Location */
void findLocation(){
    while(gpsSerial.available()){ // check for gps data
       if(gps.encode(gpsSerial.read())){ // encode gps data
         gps.f_get_position(&lat,&lon); // get latitude and longitude
       }
    }
    String latitude = String(lat,6);
    String longitude = String(lon,6);
    nodeLocation = latitude + ";" + longitude;
    Serial.println("nodeLocation: " + nodeLocation);
}
```

9.3 Appendix C - iOS App Source Code

9.3.1 *AppDelegate.swift*

```
//
//  AppDelegate.swift
//  What a Mesh!
//
//  Created by Gabriele Giuli on 2020-02-08.
//  Copyright 2020 GabrieleGiuli. All rights reserved.
//

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?


    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

        let storyboard = UIStoryboard(name: "Main", bundle: nil)
        self.window = UIWindow(frame: UIScreen.main.bounds)

        if UserDefaults.standard.string(forKey: "USER_ID") != nil {
            let destinationViewController = storyboard.instantiateViewController(withIdentifier: "NavView") as! UINavigationController
            self.window?.rootViewController = destinationViewController
            self.window?.makeKeyAndVisible()
        } else {
            let destinationViewController = storyboard.instantiateViewController(withIdentifier: "RegView") as! FirstLaunchViewController
            self.window?.rootViewController = destinationViewController
            self.window?.makeKeyAndVisible()
        }

        return true

    }

    func applicationWillResignActive(_ application: UIApplication) {
        // Sent when the application is about to move from active to inactive state. This can occur for certain types of temporary interruptions (such as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.
        // Use this method to pause ongoing tasks, disable timers, and invalidate graphics rendering callbacks. Games should use this method to pause the game.
    }

    func applicationDidEnterBackground(_ application: UIApplication) {
        // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is terminated later.
```

```
43          // If your application supports background execution, this method is called
    instead of applicationWillTerminate: when the user quits.
44      }
45
46      func applicationWillEnterForeground(_ application: UIApplication) {
47          // Called as part of the transition from the background to the active state; here
     you can undo many of the changes made on entering the background.
48      }
49
50      func applicationDidBecomeActive(_ application: UIApplication) {
51          // Restart any tasks that were paused (or not yet started) while the application
    was inactive. If the application was previously in the background, optionally refresh
     the user interface.
52      }
53
54      func applicationWillTerminate(_ application: UIApplication) {
55          // Called when the application is about to terminate. Save data if appropriate.
    See also applicationDidEnterBackground:.
56      }
57
58
59  }
```

9.3.2 *ChatsViewController.swift*

```
1   //
2   //  ChatsViewController.swift
3   //  What a Mesh!
4   //
5   //  Created by Gabriele Giuli on 2020-02-08.
6   //  Copyright 2020 GabrieleGiuli. All rights reserved.
7   //
8
9   import UIKit
10  import Alamofire
11  import SwiftyJSON
12  import MessengerKit
13
14
15  class ChatsViewController: UITableViewController {
16
17      var connected: Bool = false;
18      var address = "192.168.4.1"
19      var available_users: [ParsedUser] = []
20      var this_user: ParsedUser?
21      var selected_user_id: String?
22
23      var fwdVC: ConversationViewController?
24
25      override func viewDidLoad() {
26          super.viewDidLoad()
27
28          print("VIEWDIDLOAD")
29
30          let name = UserDefaults.standard.string(forKey: "USER_FIRSTNAME")!
```

```
31          let surname = UserDefaults.standard.string(forKey: "USER_LASTNAME")!
32          let id = UserDefaults.standard.string(forKey: "USER_ID")!
33          this_user = ParsedUser(name: name, ID: id, lat: 0, lon: 0)
34          title = name + " " + surname + "'s Conversations"
35
36          Timer.scheduledTimer(timeInterval: 3, target: self, selector: #selector(
    refreshUsers), userInfo: nil, repeats: true)
37
38      }
39
40      // MARK: - Table view data source
41
42      override func numberOfSections(in tableView: UITableView) -> Int {
43          // #warning Incomplete implementation, return the number of sections
44          return 1
45      }
46
47      override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int)
         -> Int {
48          // #warning Incomplete implementation, return the number of rows
49          return self.available_users.count
50      }
51
52
53      override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
    -> UITableViewCell {
54          let cell = tableView.dequeueReusableCell(withIdentifier: "StandardCell", for:
    indexPath)
55
56          cell.textLabel?.text = available_users[indexPath.row].name
57          cell.detailTextLabel?.text = available_users[indexPath.row].messages.last
58
59          return cell
60      }
61
62      override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath
    ) {
63          self.selected_user_id = self.available_users[indexPath.row].ID
64          self.performSegue(withIdentifier: "SegueID", sender: self.available_users[
    indexPath.row])
65      }
66
67      @objc func refreshUsers() {
68          print(self.available_users)
69
70          if !connected {
71              performHandshake()
72          }
73
74          let requestString = "http://\(self.address)/update_data"
75          Alamofire.request(requestString).responseJSON(completionHandler: { response in
76              if let json = try? JSON(data: response.data!) {
77                  print(json)
78                  self.parseUsers(json: json)
79              } else {
80                  print("Error in JSON")
```

```
81              }
82
83          })
84      }
85
86      @objc func performHandshake() {
87
88          let name = UserDefaults.standard.string(forKey: "USER_FIRSTNAME")!
89
90          let id = UserDefaults.standard.string(forKey: "USER_ID")!
91
92          let requestString = "http://\(self.address)/client_name:" + name + "/client_id:"
    + id;
93          Alamofire.request(requestString)
94      }
95
96      func parseUsers(json: JSON) {
97          for message in json["Messages"].arrayValue {
98              let sender_id = message["Source ID"].stringValue
99              let message_text = message["Message"].stringValue
100
101             self.addMessage(message_text: message_text, user_id: sender_id)
102         }
103
104         for user in json["Data"].arrayValue {
105             let user_id = user["ID"].stringValue
106             let user_name = user["Name"].stringValue
107             let location = user["Location"].stringValue
108
109             let latlon = location.split(separator: ";", maxSplits: 1)
110             guard let lat = Float(latlon[0]) else { return }
111             guard let lon = Float(latlon[1]) else { return }
112
113             insertUser(user: ParsedUser(name: user_name, ID: user_id, lat: lat, lon: lon)
    )
114         }
115     }
116
117     func insertUser(user: ParsedUser) {
118         if !isUserPresent(input_user: user) {
119             if user.ID != self.this_user!.ID {
120                 self.available_users.append(user)
121                 self.tableView.reloadData()
122
123             } else {
124                 self.connected = true
125             }
126         }
127
128     }
129
130     func isUserPresent(input_user: ParsedUser) -> Bool {
131         for user in self.available_users {
132             if user.ID == input_user.ID {
133                 return true;
134             }
```

```
135         }
136
137         return false;
138     }
139
140     func addMessage(message_text: String, user_id: String) {
141         print("adding " + message_text)
142         for user in self.available_users {
143             if user_id == user.ID {
144                 user.messages.append(message_text)
145                 self.tableView.reloadData()
146
147                 if let vc = self.fwdVC, let id = self.selected_user_id {
148                     if user_id == id {
149                     vc.id += 1
150
151                     let body: MSGMessageBody = (message_text.containsOnlyEmoji &&
    message_text.count < 5) ? .emoji(message_text) : .text(message_text)
152
153                     let message = MSGMessage(id: vc.id, body: body, user: vc.tim, sentAt:
     Date())
154                     vc.insert(message)
155                 }
156                 }
157             }
158         }
159     }
160
161     override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
162         if let vc = segue.destination as? ConversationViewController {
163             vc.recipient = sender as? ParsedUser
164             vc.addMessagesAtBeginning()
165             self.fwdVC = vc
166         } else if let vc = segue.destination as? MapViewController {
167             vc.users = self.available_users
168         }
169     }
170
171
172     @IBAction func refresh(_ sender: Any) {
173         self.available_users = []
174         self.tableView.reloadData()
175         connected = false;
176     }
177
178     @IBAction func openMap(_ sender: Any) {
179         self.performSegue(withIdentifier: "MapSegueID", sender: nil)
180     }
181
182
183     /*
184     // Override to support conditional editing of the table view.
185     override func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath)
    -> Bool {
186         // Return false if you do not want the specified item to be editable.
187         return true
```

```
    }
    */

    /*
    // Override to support editing the table view.
    override func tableView(_ tableView: UITableView, commit editingStyle:
    UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
        if editingStyle == .delete {
            // Delete the row from the data source
            tableView.deleteRows(at: [indexPath], with: .fade)
        } else if editingStyle == .insert {
            // Create a new instance of the appropriate class, insert it into the array,
    and add a new row to the table view
        }
    }
    */

    /*
    // Override to support rearranging the table view.
    override func tableView(_ tableView: UITableView, moveRowAt fromIndexPath: IndexPath,
     to: IndexPath) {

    }
    */

    /*
    // Override to support conditional rearranging of the table view.
    override func tableView(_ tableView: UITableView, canMoveRowAt indexPath: IndexPath)
    -> Bool {
        // Return false if you do not want the item to be re-orderable.
        return true
    }
    */

    /*
    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation
    before navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destination.
        // Pass the selected object to the new view controller.
    }
    */

}
```

9.3.3 *ConversationViewController.swift*

```
//
//  ConversationViewController.swift
//  What a Mesh!
//
//  Created by Gabriele Giuli on 2020-02-08.
//  Copyright 2020 GabrieleGiuli. All rights reserved.
```

```
//

import UIKit
import MessengerKit
import Alamofire

class ConversationViewController: MSGMessengerViewController {

    let steve = User(displayName: "Steve", avatar: nil, avatarUrl: nil, isSender: true)

    let tim = User(displayName: "Tim", avatar: nil, avatarUrl: nil, isSender: false)

    var id = 100

    var address = "192.168.4.1"
    var this_user: ParsedUser?
    var recipient: ParsedUser?

    override var style: MSGMessengerStyle {
        var style = MessengerKit.Styles.iMessage
        style.headerHeight = 0
        return style
    }


    var messages: [[MSGMessage]] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        title = "iMessage"

        let name = UserDefaults.standard.string(forKey: "USER_FIRSTNAME")!
        let surname = UserDefaults.standard.string(forKey: "USER_LASTNAME")!
        let id = UserDefaults.standard.string(forKey: "USER_ID")!
        this_user = ParsedUser(name: name, ID: id, lat: 0, lon: 0)

        dataSource = self
        delegate = self
    }

    func addMessagesAtBeginning() {
        for message in self.recipient!.messages {
            id += 1
            print("Current Message: " + message)
            let body: MSGMessageBody = (message.containsOnlyEmoji && message.count < 5) ?
 .emoji(message) : .text(message)
            let message_b = MSGMessage(id: id, body: body, user: tim, sentAt: Date())

            self.messages.append([message_b])
        }
    }

    override func viewDidAppear(_ animated: Bool) {
        super.viewDidAppear(animated)

        collectionView.scrollToBottom(animated: false)
```

```
62      }
63
64      override func inputViewPrimaryActionTriggered(inputView: MSGInputView) {
65          id += 1
66
67          let body: MSGMessageBody = (inputView.message.containsOnlyEmoji && inputView.
    message.count < 5) ? .emoji(inputView.message) : .text(inputView.message)
68
69          let message = MSGMessage(id: id, body: body, user: steve, sentAt: Date())
70          sendMessage(message: message)
71          insert(message)
72      }
73
74      override func insert(_ message: MSGMessage) {
75
76          collectionView.performBatchUpdates({
77              if let lastSection = self.messages.last, let lastMessage = lastSection.last,
    lastMessage.user.displayName == message.user.displayName {
78                  self.messages[self.messages.count - 1].append(message)
79
80                  let sectionIndex = self.messages.count - 1
81                  let itemIndex = self.messages[sectionIndex].count - 1
82                  self.collectionView.insertItems(at: [IndexPath(item: itemIndex, section:
    sectionIndex)])
83
84              } else {
85                  self.messages.append([message])
86                  let sectionIndex = self.messages.count - 1
87                  self.collectionView.insertSections([sectionIndex])
88              }
89          }, completion: { (_) in
90              self.collectionView.scrollToBottom(animated: true)
91              self.collectionView.layoutTypingLabelIfNeeded()
92          })
93
94      }
95
96      func sendMessage(message: MSGMessage) {
97          if let text = message.body.rawValue as? String {
98              var requestString = "http://\(self.address)/message:" + processMessage(
    inString: text) + "/source_id:" + self.this_user!.ID + "/target_id:"
99              requestString = requestString + self.recipient!.ID
100
101             print("Request: " + requestString)
102
103             Alamofire.request(requestString)
104         }
105     }
106
107     func processMessage(inString: String) -> String {
108         var newString = inString.replacingOccurrences(of: " ", with: "%20")
109         newString = newString.replacingOccurrences(of: "\n", with: "%20")
110         return newString
111     }
112
113     override func insert(_ messages: [MSGMessage], callback: (() -> Void)? = nil) {
```

```
114
115         collectionView.performBatchUpdates({
116             for message in messages {
117                 if let lastSection = self.messages.last, let lastMessage = lastSection.
    last, lastMessage.user.displayName == message.user.displayName {
118                     self.messages[self.messages.count - 1].append(message)
119
120                     let sectionIndex = self.messages.count - 1
121                     let itemIndex = self.messages[sectionIndex].count - 1
122                     self.collectionView.insertItems(at: [IndexPath(item: itemIndex,
    section: sectionIndex)])
123
124                 } else {
125                     self.messages.append([message])
126                     let sectionIndex = self.messages.count - 1
127                     self.collectionView.insertSections([sectionIndex])
128                 }
129             }
130         }, completion: { (_) in
131             self.collectionView.scrollToBottom(animated: false)
132             self.collectionView.layoutTypingLabelIfNeeded()
133             DispatchQueue.main.asyncAfter(deadline: .now() + 0.3) {
134                 callback?()
135             }
136         })
137
138     }
139
140 }
141
142 // MARK: - Overrides
143
144 extension ConversationViewController {
145
146 }
147
148 // MARK: - MSGDataSource
149
150 extension ConversationViewController: MSGDataSource {
151
152     func numberOfSections() -> Int {
153         return messages.count
154     }
155
156     func numberOfMessages(in section: Int) -> Int {
157         return messages[section].count
158     }
159
160     func message(for indexPath: IndexPath) -> MSGMessage {
161         return messages[indexPath.section][indexPath.item]
162     }
163
164     func footerTitle(for section: Int) -> String? {
165         return "Just now"
166     }
167
```

```
168     func headerTitle(for section: Int) -> String? {
169         return messages[section].first?.user.displayName
170     }
171 
172 }
173 
174 // MARK: - MSGDelegate
175 
176 extension ConversationViewController: MSGDelegate {
177 
178     func linkTapped(url: URL) {
179         print("Link tapped:", url)
180     }
181 
182     func avatarTapped(for user: MSGUser) {
183         print("Avatar tapped:", user)
184     }
185 
186     func tapReceived(for message: MSGMessage) {
187         print("Tapped: ", message)
188     }
189 
190     func longPressReceieved(for message: MSGMessage) {
191         print("Long press:", message)
192     }
193 
194     func shouldDisplaySafari(for url: URL) -> Bool {
195         return true
196     }
197 
198     func shouldOpen(url: URL) -> Bool {
199         return true
200     }
201 
202 }
```

9.3.4 *FirstLaunchViewController.swift*

```
1  //
2  //  ViewController.swift
3  //  What a Mesh!
4  //
5  //  Created by Gabriele Giuli on 2020-02-08.
6  //  Copyright 2020 GabrieleGiuli. All rights reserved.
7  //
8 
9  import UIKit
10 
11 class FirstLaunchViewController: UIViewController, UITextFieldDelegate {
12 
13     @IBOutlet weak var firstName: LoginTextField!
14     @IBOutlet weak var lastName: LoginTextField!
15 
16     override func viewDidLoad() {
17         super.viewDidLoad()
```

```
18
19          self.firstName.delegate = self
20          self.lastName.delegate = self
21          // Do any additional setup after loading the view.
22      }
23
24      override var prefersStatusBarHidden: Bool {
25          return true
26      }
27
28      @IBAction func UserTapped(_ sender: Any) {
29          self.processData()
30      }
31
32      func processData() {
33          if !isValidName(firstName.text!) || !isValidName(lastName.text!) {
34              let alert = UIAlertController(title: "Check your Name", message: "The names
    you have just entered are not valid, re-enter them and retry", preferredStyle:
    UIAlertController.Style.alert)
35              alert.addAction(UIAlertAction(title: "Mhh... Sure!", style: UIAlertAction.
    Style.default, handler: nil))
36              self.present(alert, animated: true, completion: nil)
37          } else {
38              let newId = getNewID();
39              UserDefaults.standard.set(newId, forKey: "USER_ID")
40              UserDefaults.standard.set(firstName.text!, forKey: "USER_FIRSTNAME")
41              UserDefaults.standard.set(lastName.text!, forKey: "USER_LASTNAME")
42
43              let alert = UIAlertController(title: "Success!", message: "You are all set.
    Connect to the WAM network!", preferredStyle: UIAlertController.Style.alert)
44              alert.addAction(UIAlertAction(title: "Ok", style: UIAlertAction.Style.default
    , handler: { UIAlertAction in
45
46                  let storyboard = UIStoryboard(name: "Main", bundle: nil)
47                  let messagesViewController = storyboard.instantiateViewController(
    withIdentifier: "NavView") as! UINavigationController
48                  messagesViewController.modalPresentationStyle = .fullScreen
49                  self.present(messagesViewController, animated: true, completion: nil)
50
51              }))
52
53              self.present(alert, animated: true, completion: nil)
54          }
55      }
56
57      func isValidName(_ name: String) -> Bool {
58          let nameRegEx = "(?<! )[-a-zA-Z' ]{2,26}"
59
60          let namePred = NSPredicate(format:"SELF MATCHES %@", nameRegEx)
61          return namePred.evaluate(with: name)
62      }
63
64      func getNewID() -> String {
65          let date = Date()
66          let calender = Calendar.current
```

```
67          let components = calender.dateComponents([.year,.month,.day,.hour,.minute,.second
    ], from: date)
68
69          let year = components.year
70          let month = components.month
71          let day = components.day
72          let hour = components.hour
73          let minute = components.minute
74          let second = components.second
75
76          let randomNumber = Int.random(in: 0 ..< 10000)
77
78          let today_string = String(year!) + String(month!) + String(day!) + String(hour!)
    + String(minute!) + String(second!) + String(randomNumber)
79
80          return today_string
81      }
82
83      func textFieldShouldReturn(_ textField: UITextField) -> Bool {
84
85          //textField code
86
87          textField.resignFirstResponder()  //if desired
88          self.processData()
89          return true
90      }
91
92
93  }
```

9.3.5 *MapViewController.swift*

```
1   //
2   //  MapViewController.swift
3   //  What a Mesh!
4   //
5   //  Created by Gabriele Giuli on 2020-03-11.
6   //  Copyright 2020 GabrieleGiuli. All rights reserved.
7   //
8
9   import UIKit
10  import MapKit
11
12  class MapViewController: UIViewController, MKMapViewDelegate {
13
14      @IBOutlet weak var map: MKMapView!
15      let locationManager = CLLocationManager()
16      var users: [ParsedUser]!
17
18      override func viewDidLoad() {
19          super.viewDidLoad()
20
21          map.delegate = self
22          checkLocationServices()
23          showLocations()
```

```
    }

    func checkLocationServices() {
        if CLLocationManager.locationServicesEnabled() {
            checkLocationAuthorization()
        } else {
            // Show alert letting the user know they have to turn this on.
        }
    }

    func checkLocationAuthorization() {
        switch CLLocationManager.authorizationStatus() {
            case .authorizedWhenInUse:
                map.showsUserLocation = true
                case .denied: // Show alert telling users how to turn on permissions
                break
            case .notDetermined:
                locationManager.requestWhenInUseAuthorization()
                map.showsUserLocation = true
                case .restricted:
                break
            case .authorizedAlways:
            break
        }
    }

    func showLocations() {
        for user in self.users {
            let annotation = MKPointAnnotation()
            annotation.title = user.name
            annotation.coordinate = user.location
            print("\(user.name) ANNOTATION ADDED")
            self.map.addAnnotation(annotation)
        }
    }

}
```

9.3.6 *LoginTextField.swift*

```
//
//  LoginTextField.swift
//  What a Mesh!
//
//  Created by Gabriele Giuli on 2020-02-08.
//  Copyright 2020 GabrieleGiuli. All rights reserved.
//

import Foundation
import UIKit

@IBDesignable
class LoginTextField: UITextField {

    override func layoutSubviews() {
```

```
        super.layoutSubviews()

        self.layer.borderColor = UIColor(white: 231/255, alpha: 1).cgColor
        self.layer.borderWidth = 1
    }

    override func textRect(forBounds bounds: CGRect) -> CGRect {
        return bounds.insetBy(dx: 8, dy: 3)
    }

    override func editingRect(forBounds bounds: CGRect) -> CGRect {
        return textRect(forBounds: bounds)
    }

}
```

9.3.7 *ParsedUser.swift*

```
//
//  ParsedUser.swift
//  What a Mesh!
//
//  Created by Gabriele Giuli on 2020-02-08.
//  Copyright 2020 GabrieleGiuli. All rights reserved.
//

import Foundation
import MapKit

class ParsedUser {
    var name: String
    var ID: String
    var location: CLLocationCoordinate2D
    var messages: [String] = []

    init(name: String, ID: String, lat: Float, lon: Float) {
        self.name = name
        self.ID = ID
        self.location = CLLocationCoordinate2D(latitude: CLLocationDegrees(lat),
    longitude: CLLocationDegrees(lon))
    }
}
```

9.3.8 *User.swift*

```
//
//  User.swift
//  What a Mesh!
//
//  Created by Gabriele Giuli on 2020-02-08.
//  Copyright 2020 GabrieleGiuli. All rights reserved.
//

```

```
9
10 import MessengerKit
11
12 struct User: MSGUser {
13
14     var displayName: String
15
16     var avatar: UIImage?
17
18     var avatarUrl: URL?
19
20     var isSender: Bool
21
22 }
```

9.4 Appendix C - Product Design Specification

Performance The solution should be capable of providing a reliable *off-grid* communication network. The device should be capable of delivering the following main features:

- Establish a long range and secure communication (e.g. encrypted) channel between users willing to communicate to each other
- Feature a versatile and *user-friendly* interface that allows the user to easily access the communication link

The system must be capable of exchanging all types of digital data (i.e. text messages, emails, pictures...) over a long range while also being capable of both real-time data exchange and temporary data storage to account for offline users. Our device should be capable of running autonomously for long periods of time without access to ordinary powers sources such as grid power and must not rely on any external infrastructure to deliver the messages - it has to be completely autonomous.

Environment In order for our product to meet its specification it should be robust against harsh environments. To ensure this, we are looking into the following areas:

- Should be water-resistant (circuitry should be protected from water) and protected against small insects and debris, as the device may be subject to any remote environment where unexpected rain, dust, or presence of insects is to be accounted for
- Should be able to withstand extreme temperatures, humidity, and dirty conditions, so that the device may be used under any emergency
- Should not cause a negative impact to the environment (i.e. not pollute, not be too loud, not disrupt ecosystems)
- Should be shockproof as the device is mainly intended to be used for outdoor activities
- The presence and/or absence of signal obstacles (e.g. trees, mountains) should not hinder performance to an unusable extent (slight decrease in performance may be allowed)

Life in Service (Performance) Ideally, the product should function for long periods of time so that minimum maintenance is required. It is important that we maximize the product's life in service as the customers are likely to use this product in circumstances where communication is essential for survival (i.e. to communicate with a *search and rescue* team). Therefore, the final product's battery should last at least a few days (after which, the device would need to be recharged). In the context of emergency communication, life in service should be one of our priorities.

Maintenance We will be working at a systems level for our project. This means we will not be putting together each individual component and rather putting together systems that already have a purpose to make our overall project. This allows the user to easily buy spare parts from third party brands, hence increasing repairability. The main issue for maintenance will be battery life. The device will have an accessible rechargeable battery so that the user can recharge it when required and easily change battery when worn out: the customer does not have to buy a new product every time the battery wears out.

In conclusion, the only expected regular maintenance is charging the battery and substituting it when worn out. In addition to that, a modular design will allow for easy fixes even in remote areas where official resellers might not be available.

Target Product Cost As we require multiple pieces of hardware for our product to work, we will split the budget to be able to make them. This means we would have about £150 to make each product so we should aim to market them between £200 - £300. This is so our product would be much cheaper than the competitors, thus achieving our main goal while also being able to make profit on our product.

Competition Competition is limited since there is only one other company which already provides a similar device: *goTenna* (`https://gotenna.com/`). They sell similar devices, but they are not trying to implement in Wi-Fi or similar features for the whole network. Also, their main aim is for general usage of these mesh networks and not targeted to emergency situations or bringing communication access to rural areas. Other similar Open-Source projects are present but they do not represent a significant competitor.

Shipping The main practical methods of international delivery of the product are sea and air freights. However, due to the comparably lengthy delivery time of sea freights, as well as the marginally lower costs of air transportation (due to the small dimension and weight of our product), it is concluded that international shipping of single products will principally be done through air. This is because air freights are charged predominantly by the weight of the product; and since our product should optimally weigh less than 1kg, the overall delivery cost of air transportation is typically less than its seaborne counterpart. However, this option should still remain flexible, as for bulk delivery - where large amounts of the product can be sent at one time - sea freight can prove to be markedly cheaper than air transport. Domestic shipping within the UK, on the other hand, can mainly be done through *LTL* (Less Than Truckload shipping) or other available domestic courier services such as *DHL* and *Parcelforce*.

Packing At this stage, packing is not a priority for this project. However, depending on how fragile the final product is we may have to consider this parameter. Ideally, our final product should be robust enough (see *Environment* section) so that we do not require any elaborate packaging.

Quantity The approach we are planning to take for this project is to produce nodes capable of transmitting information from one end to the other, thus forming a network. With this approach

in mind, the final product should be capable of handling an arbitrary number of nodes and so the quantity will depend on the distances we need to cover. For demonstration purposes, two nodes will suffice (One stationary and the other one portable).

Manufacturing Facilities Initially a *buy-out* strategy will be followed, allowing the team to work on a system level: little or no specialized tools and facilities will be required. Furthermore, series production will not be required for the initial stage of the project (i.e. demonstration), hence the following standard and readily available facilities can be used to manufacture the product:

- **Imperial College Robotics Lab**: Based in the EEE building which has 3D printers and laser cutters and technical help to use these devices.
- **First and Second year labs**: Also based in EEE building, provides tools for prototyping/building simple circuits. Lab technicians can also be a useful resource.

For later stages of the product, specialized equipment and manufacturing facilities will be necessary for mass production and for a reduction in manufacturing costs. At this stage a *make-in* strategy might be chosen to ensure a higher degree of flexibility in design and overall lower production costs.

Size There are different situations in which our product could be used. The size of our product is mainly important for people who go hiking, climbing or cycling into remote areas without network connection. They normally carry a lot of items in terms of food, clothes, safety and health equipment, but they have a limited storage space in their backpacks. This is why we need to keep the size of the product to a minimum: we forecast that our product's size will be around $10 \times 10 \times 10$ cm, but we will try to make it as small as possible.

Weight Though there is no weight limit, the device should be made to be as light as possible so as to not cause an inconvenience to the user carrying it around in whatever environment they may be subject to. Ideally, the device structure should be made of less and lighter material, allowing for more of the budget to be invested in better quality components. However, the weight clearly depends on *Size*, and given current signal transducing and networking technology the absolute maximum mass should be around 1 Kg to ensure portability.

Aesthetic, Appearance and Finish The aesthetic and design of the product should be modern (perhaps even colorful), minimalistic, and simple to both appeal to the market's taste and to ensure the ease of use. For the outer finishing of the product, rubber or materials with similar properties in terms of shock-absorbance, water-resistance, and durability, is ideal, especially since the environment in which the product is used can be moderately hazardous (e.g. when used for hiking), hence giving it an industrial look. However, because the primary objective of the product remains to provide means of communication in rural areas or isolated communities, no important device functionality should be compromised for better design or appearance.

Materials We require the final product to be waterproof and robust because of the harsh environments it is intended to be used in. Because of our facilities (3D printers and laser cutters) we will have access to lots of waterproof plastics such as acrylic and PLA which are strong and waterproof – these would be used to provide a casing for the circuitry. We would also need material to provide shock absorption and to make the product stable on uneven ground. We suggest lining the edges and bottom of the case with rubber which would provide a solution to both problems as rubber has a high coefficient of friction and it is shock absorbent.

Product Life-Span In terms of overall life-span of our product, we will try to make it as durable as possible, especially for villages in remote areas where the device will be used daily during a long period of time. We want to find the balance between cost, life-span and performance, so we will keep in mind that it is not sustainable to double the life-span of the product if we have to triple its cost to achieve it.

There is a very critical point to tackle in this subsection, which is the battery life of our product. We need a good battery life for long excursions, because people may not be able to access a power source for more than a week. This problem scales quickly because the performance of a product does not only depend on that product itself but also on the other devices it has to connect to in order to send messages (we build our own infrastructure). There are various solutions possible to this problem:

- We could run our product on normal batteries (9V or Alkaline batteries), so our users could bring spare batteries with them in case.
- A battery saving mode, where if the user is not trying to send any message, it does not perform some of its tasks and simply plays the role of infrastructure for other devices.

No planned obsolescence is taken into account since we aim to build the device as reliably as possible: the system will be used in life-threatening situations and reliability is essential.

Standards and Specifications The device will potentially exploit radio-frequency technology to establish a communication link, hence it has to comply with *EMC* (ElectroMagnetic Compatibility) regulating standards such as the *FCC* and/or *CE* depending on the target market. In particular the device needs to operate in the *ISM* (Industrial Scientific Medical) frequency band to be used without a specific licence. Furthermore, the device's output power must be below the legal threshold. However, working at a system level (i.e. already available RF modules) will significantly reduce the effort needed to certify the final product. In addition, an initial prototype for demonstration purposes only will not require strict certifications, as long as the local frequency and output power requirements are met.

Ergonomics The device should be of reasonable size and weight such that it fits into a usual sized rucksack and can be carried around comfortably. The device's case needs to be smooth without any sharp edges or corners that could cause injuries.

Customer We intend to make this product for people going to potentially harsh and remote areas (e.g. rangers, search teams, climbers, trekkers) and for authorities who need communication channels during natural disasters. The target customer's age might vary significantly, however it can be concluded that the user's age should be 25+ years old. Physical characteristic (e.g. height, weight, physical structure, etc) should not affect the usability of the product; however, the user will need to be capable of carrying the device with them.

Quality and Reliability Although we need a very good performance and reliability, quality is not one of our main preoccupations: we don't need the best materials or the best technology as long as our product works reliably. Cost-efficiency is the real priority in this aspect to be competitive in the market. We aim for a 95% of success in our tests, which would prove good reliability.

In the end, *MTBF* (Mean Time Before Failure) and *MTBR* (Mean Time Before Repair) should both be maximized keeping in mind the economical constraints.

Shelf Life (Storage) Ideally, we would like to maximize shelf life by making our product robust and durable. As an estimate, we expect the final product to have a shelf life of at least a few years. By maximizing shelf life, we would be providing a cost-effective solution in the long run which is one of our priorities for this project.

Processes The outer shell will be 3D printed in PLA plastic with other small mounting components might be laser-cut in acrylic. Our main PCB will be manufactured either using internal resources or ordering it from third party PCB builders such as *JLCPCB*. Different modules will be interconnected using either soldering or fast-connectors, the latter is preferred since it enhances modularity and repairability.

Time-Scale The demo of the project will be on 17th March 2020. That means, from now on we have four months to complete the project. The following is an approximate timeline for the project:

1. Detailed definition of solutions [**week 6 - week 7**]
2. Detailed technical mapping and role allocation between group members [**week 7 - 8**]
3. Prototype hardware development (includes testing) [**week 8 - 11**]
4. Prototype software development (includes testing) [**week 8 - 11**]
5. Assembly of components [**week 11 - 13**]
6. Prototype testing [**week 13 - 14**]
7. Fine modifications [**week 14 - 17**]
8. Testing final product [**week 17 - 18**]

Testing The testing of the product will be divided into two phases. The first phase concerns the testing of the hardware that must be safe to handle must not overheat and currents and voltages need to be in the specified boundaries to guarantee a safe and reliable operation. The initial testing can be conducted on each subsystem before integrating it with the other modules. This will make the whole system less prone to issues. The testing tools required are oscilloscopes and multimeters. The device will be run for an extended period of time and important parameters will be checked. Any arising issue will be addressed.

After the hardware testing is finished, the device's software capabilities will be checked. This can be done quicker than hardware testing: the user interface can be tested instantly at each development stage. The total system behaviour is going to be more difficult to test, but it can be done by checking each functionality one by one hence debugging can be done quickly. Software testbenches can be also utilized for checking reliability and finding bugs.

Safety There are not a lot of safety issues regarding the use or the manufacturing of our product. The main safety concern would is represented by the potential overheating of the device that could cause some components to get damaged, burn and start a fire. But with the voltages we are dealing with there is no realistic risk of fire or any threat to human health. The only thing that could be damaged by overheating is the product itself. The radio frequencies and output powers we are planning on using are not considered harmful to the human body.

Company Constraints We are working with a budget of £450 for every expense in our project. Our main goal was to make this technology attainable by more people to increase communication capabilities. To do this we must limit our price point to much less than the competitors. We need to make multiple modules for our final product so that we can demonstrate the communication process, meaning we must account for making multiple products in our budget. We are also limited by our manufacturing facilities being mainly 3D printers and laser cutters in the robotics laboratory which limits the materials we can use.

Market Constraints As this device targets a niche audience, demand for this product overall may not be very high, therefore implying that the cost of production should be minimised as much as possible.Within the target audience, there may be a few customers who will not see the benefit of using out product rather than an alternative. To counteract this we would have multiple features (unique to our product) in our device, so that even if a new product optimises a certain function of our device, our product has the advantage of having other convenient features

Patents, Literature and Product Data There are no known patents that would limit the development of the product. The main *mesh networks* already have a few standards which are *IEEE* standards like *IEEE 802.11s* (`https://ieeexplore.ieee.org/document/5416357`). Hence there are no patent issues with the communication protocol. Looking at the unlicensed communication channels allowed in the UK: (863-870 MHz) or 2.4 GHz frequencies are off license hence these frequency bands can be used without acquiring a licence. Overall there are no patent or legislation issues that would interfere with the development of the product. `http://static.ofcom.org.uk/static/spectrum/fat.html`

Political and Social Implications The political and social implications of the product should be overwhelmingly positive. It will provide cellphone owners, who require cellular towers for wireless communication (or mobile coverage), a much cheaper alternative to satellite phones when travelling in regions where no such communication towers are nearby. This will also reduce the need and burden to governments to build cell towers in many rural parts of the country, which on average costs $ 150,000 per tower, an investment that could instead be directed elsewhere. Lastly, the product can connect isolated communities / villages to the rest of the country, and has the potential to greatly reduce the number of worldwide casualties due to being stranded with no means of communication (due to lack of mobile coverage). These factors contribute to the political and socioeconomical favourability of the product to countries around the world which utilizes it.

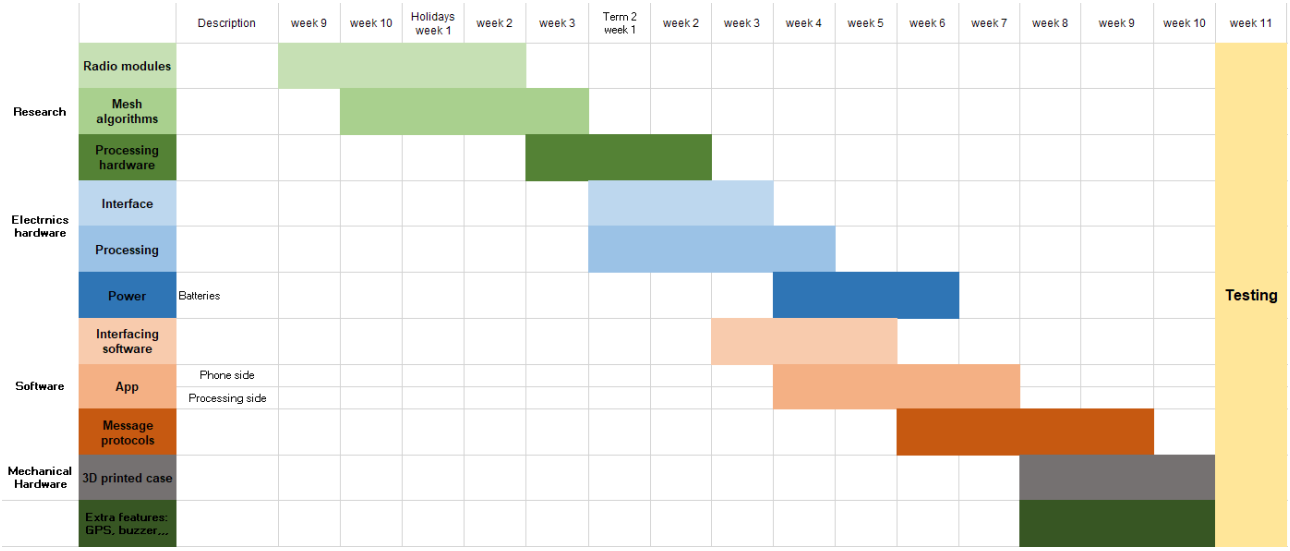
Legal The only legal problem that our product would have is how we deal and store data (locations, messages...). Our device is used as a part of the infrastructure, which means that all the messages and data go through different devices before reaching a network connection. This data should not be available to users of the devices who are neither the emitter nor the receiver of the message. We can either encrypt the message or build a firewall for the messages traveling across different devices. In the end *GDPR* like documentation will need to be consulted to ensure that we protect the users' privacy.

Installation The device is going to be transportable, user-friendly: no specialized training is required. In terms of software the device will come with previously installed firmware and the user is not required to install any additional piece of software in order to use the device. This shows that installation would not need technical expertise, hence the device will be easy to use by the costumer and even software updates would not be an issue.

Documentation Although the overall use of the device should be simple and self-explanatory, a detailed documentation and/or manual is required to both explain in high-level how the device works and the principles behind it, as well as to describe all the functionalities present. This may include methods of navigating to the desired frequency channel, instructions on how to set up the device and connect it with your own personal device (e.g. phone or laptop), and explanations on all the features of the App/software and how to use it to send bitstreams of information through the channel to target nodes.

Disposal The modular design of the device makes it possible to recycle and/or reuse most of the components. Furthermore, *RoHS* compliant components will be used to further increase environment compatibility. The outer PLA shell can be fully recycled, as well as the rubber used for shock protection.

9.5 Appendix D - Gantt Chart



References

[1] Amazon. NUZAMAS 3.5W 6V 600ma Mini Solar Panel. *https://www.amazon.co.uk/NUZAMAS-Outdoor-Camping-Battery-Charger/dp/B073CDW1VZ/ref=asc_df_B073CDW1VZ/?tag=googshopuk-21&linkCode=df0&hvadid=309964075885&hvpos=1o3&hvnetw=g&hvrand=15021748749304376447&hvpone=&hvptwo=&hvqmt=&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9072501&hvtargid=pla-469553834775&psc=1*.

[2] Banggood. TP4056 5V 1A Lipo Battery Mini USB Charging Board. *https://uk.banggood.com/TP4056-1A-Lipo-Battery-Charging-Board-Charger-Module-Mini-USB-Interface-p-1027027.html?gmcCountry=GB&currency=GBP&createTmp=1&utm_source=googleshopping&utm_medium=cpc_bgcs&utm_content=zouzou&utm_campaign=pla-gbg-all-pc-1010&ad_id=388626607191&gclid=CjwKCAiAob3vBRAUEiwAIbs5TmPW0WbOF_RQhewG3R3n_QUEbf6r82hlU-JvXGDKBuEyKKJZCVG4ohoCQ50QAvD_BwE&cur_warehouse=CN*.

[3] Ericsson. Population Coverage. *https://ericsson.com/en/mobility-report/population-coverage*.

[4] Ground Control. MCD-4800 - A Portable, Glocal, Satellite Internet Hotspot - Land & Sea. *http://www.groundcontrol.com/MCD-4800_BGAN_Terminal.htm*.

[5] Inmarsat. Inmarsat Services. *https://www.inmarsat.com/services/*.

[6] Iridium. Iridium Satellite Phones. *https://www.iridium.com/phones/*.

[7] Jayme Moye, NATIONAL GEOGRAPHIC. Day hikers are the most vulnerable in survival situations. Here's why. *https://www.nationalgeographic.com/adventure/2019/04/hikers-survival-tips/*.

[8] Mouser. BOB-08276 Breakout Board. *https://www.mouser.co.uk/ProductDetail/SparkFun/BOB-08276?qs=%2Fha2pyFaduiJQpwKf1JYNlCbhN0cQH8tpul8KPP47%252BhhZ8%2F67M8aag%3D%3D*.

[9] Mouser. DIGI XKB2-Z7T-WZM. *https://www.mouser.co.uk/ProductDetail/DIGI/XKB2-Z7T-WZM?qs=0lHBc9ha0jd03LtSa9HD5A==&vip=1&gclid=CjwKCAiAmNbwBRBOEiwAqcwwpfKTugAMlw49jV8jWPFVypoS_7301JQtVKD_5uxUvhef-Xa0mO6avBoCmq0QAvD_BwE*.

[10] Open Garden. Firechart App Presentation. *https://www.opengarden.com/firechat/*.

[11] Project OWL. Project OWL Presentation. *https://project-owl.com*.

[12] Rock Seven. About RockBLOCK Mk2. *https://www.rock7.com/products-rockblock*.

[13] RS Components. 3.7 V - 1.8 Ah LiPo Batteries. *https://uk.rs-online.com/web/p/speciality-size-rechargeable-batteries/1449405/*.

[14] RS Components. Arduino MKR WiFi 1010. *https://uk.rs-online.com/web/p/processor-microcontroller-development-kits/1763647/?relevancy-data=636F3D3126696E3D4931384E536561726847656E65726963266C753D656E266D6D3D6D617463*
searchHistory=%7B%22enabled%22%3Atrue%7D.

[15] RS Components. Samsung ICR18650-26F. *https://uk.rs-online.com/web/p/speciality-size-rechargeable-batteries/7887261/*.

[16] SparkFun. RockBLOCK 9603 - Iridium SatComm Module. *https://www.sparkfun.com/products/14498*.

[17] The Pi Hut. Adafruit 2mm 10 pin Socket Headers. *https://thepihut.com/products/adafruit-2mm-10-pin-socket-headers-for-xbee-pack-of-2?variant=27740416337&currency=GBP&gclid=CjwKCAiAx_DwBRAfEiwA3vwZYnL7ed8gy_Ts_5ltUgY9r88PILZJW9_3vc8y8yszsPWgg6fKetzc3hoC7X8QAvD_BwE*.

[18] The Pi Hut. Adafruit Feather M0 WiFi. *https://thepihut.com/products/adafruit-feather-m0-wifi-atsamd21-atwinc1500?variant=27740276881&currency=GBP&gclid=CjwKCAiAob3vBRAUEiwAIbs5TscN92XYGzY7aTMDzzjNZUbVHCWTQWfaKmbKEJt3Y0bebyGidZnN9Bo
BwE*.

[19] The Pi Hut. NUZAMAS 3.5W 6V 600ma Mini Solar Panel. *https://thepihut.com/products/adafruit-rfm96w-lora-radio-transceiver-breakout-433-mhz?variant=27740305809&currency=GBP&gclid=CjwKCAiAob3vBRAUEiwAIbs5Tg7ijHWkt2yrwgNDIqEwobwwp-GAkSFB4_s-qOfI4dTLo7XXL2XyMhoCKR8QAvD_BwE*.

[20] The Pi Hut. Raspberry Pi 3 Model B. *https://thepihut.com/products/raspberry-pi-3-model-b*.

[21] The Telegraph. Father saw son and friend freeze to death on Norwegian cross-country skiing trip. *https://www.telegraph.co.uk/news/worldnews/1545404/Father-saw-son-and-friend-freeze-to-death-on-Norwegian-cross-country-skiing-trip.html*.

[22] University of Hawaii at Manoa. New wireless communications system to serve remote and rural areas. *https://phys.org/news/2014-10-wireless-remote-rural-areas.html*.