DEVELOPMENT OF A LORA BASED MULTIPLAYER GAME USING AN ARDUINO APPLICATION

FINAL REPORT

COURSE: WIRELESS MOBILE NETWORKS
FACULTY: ANTÓNIO GRILO

MIGUEL RODRIGUES Nº76176
HENRIQUE PIEDADE Nº75546
GROUP 19

3/06/2018

1 Introduction

In this final report, we aim to expose the work done in order to create a shooting and mine deploying game with an Arduino Uno board as a gun, coupled with a Dragino LoRa GPS shield, with the communications being made using LoRaWAN, a network based in LoRa modulation technology, as described in the previous report. In short, a player is available to shoot and deploy mines. Also, the player is notified when he or she gets killed or kills someone by audible queues.

For this project, different technologies are used, such as the Arduino programming language (which is actually a mix of several languages that can be used in the Arduino IDE, as C and C++), for the board and JavaScript, PHP, HTML and mySQL for the server, database and communication with The Things Network.

As for the communication methods, the LoRa modulation is used, making use of the LoRaWan architecture with which the communication is made to the The Things Network, who forwards the needed information from the board to the server and from the server to the board. This way we use a somewhat new IoT technology that can be used reliably and might play an important role in IoT development.

2 Game Rules

In order for the game to be played, there is a set of rules that must be set to ensure the game is played correctly.

As the game starts, the player must wait for an available GPS transmission analyzed by the value of the available satellites. While there are not enough GPS satellites available, no communication is sent and no action can be performed. To notify the player, the buzzer produces a high pitched tone (2kHz) for two seconds, as described in Listing 1, where v is a control variable, used only to check if the program is already correctly running the the proper GPS connection and *smartdelay* is a delay function described further ahead in the Arduino Libraries section, as well as the GPS and functions and values.

Listing 1: Wait for a valid GPS signal to join the game.

```
if(v==0){
    if(gps.satellites()!=TinyGPS::GPS_INVALID_SATELLITES){
        tone(buzzer,2000);
        delay(2000);
        noTone(buzzer);
        v=1;
        do_send(&sendjob);
    }
smartdelay(1000);
}
```

After this initial moments, the player's position is updated every 45 seconds automatically. Since each payload has an estimated airtime of a little bit less than 45 ms, and according to the Fair Use policy described in the next section, the idle messages updating the player's position are sent every 45 seconds after the last one or after the last action performed by the player. That being said, the user can only perform an action (shoot or deploy a mine) every 30 seconds after the previous action. Using the *millis()* function and a variable to store the last action's time, a new action is only accepted if the result of the subtraction of that function for the stored variable is higher or equal than 30000 milliseconds. If the user tries to act on the forbidden time interval, a small beep is sent to notify the user that is was not accepted.

As for the actual gameplay, in order to shoot the gun, the button must be pressed and released in under three

seconds and to deploy a mine the player must press the button for at least threes seconds. According to the three different situations, the *function* value may differ between 0 (regular position update), 1 (player shoots), and 2 (player deploys a mine).

Listing 2: Player actions.

```
    if(v!=0){
        if(lastButtonStateGreen == LOW && ButtonStateGreen==HIGH && (millis()-lastAction >30000)){
            firstTime=millis();
        }

        if(lastButtonStateGreen == HIGH && ButtonStateGreen==LOW&& (millis()-lastAction >30000)){
            millis_held=millis()-firstTime;

        if(millis_held<3000){
            os_clearCallback(&sendjob); //clear scheduled job
            function[0] = '1';
            os_setCallback(&sendjob, do_send);
            lastAction=millis();

        }else{
            os_clearCallback(&sendjob); //clear scheduled job
            function[0] = '2';
            os_setCallback(&sendjob, do_send);
            lastAction=millis();
        }
    }
    lastButtonStateGreen = ButtonStateGreen;
    os_runloop_once();
}
```

A mine will kill a player in a radius of 10 metres from where it was deployed and a bullet will kill any player in a straight line for 50 metres, according to the shooters orientation, in an interval of 5 seconds, making it very difficult to kill someone, since positions are only updated every 30 seconds at the maximum rate (constant actions).

Every time an action results in a kill, the player is notified according to Table 1 and the scores of the respective players are shown in the game's page.

Game Action	Board Action	Downlink Message
death by shooting	one second low-pitch tone (0.5kHz)	0
death by mine	two one second low-pitched tones (0.5kHz)	1
kill by shooting	one second medium-pitched tone (1kHz)	2
kill by mine	two one second medium-pitched tones (1kHz)	3

Table 1: Relation between game actions, board actions and downlink messages.

3 LoRa, LoRaWAN and The Things Network

LoRa stands for a patented **Lo**ng **Ra**nge wireless data communication IoT protocol, developed by Cycleo of Grenoble and currently owned by Semtech. It uses CHIRP (Compressed High Impulse Radio Pulse) modulation and allows for very long-range transmissions with a very low power consumption, making it very useful for IoT development, as the devices won't have their batteries drained in a short period of time while still being able to communicate. Communication can be established between LoRa nodes and servers, with no resource to gateways or auxiliary platforms. However, in order to centralize our application, LoRaWAN is used, through the The Things Network.

The upper layers of LoRa are defined in LoRaWAN, a Low Power Wide-Area Network, responsible for managing the frequencies to use and other technical aspects of the communication protocol.

In this network, communication is made asynchronously, through nodes and gateways. Gateways are public devices set up in order to receive and deliver data from and to the nodes (end devices that are used to send/receive data and perform the required actions for that system) to the network. When a node has available data to send, it sends it's data to LoRa's frequencies hoping that a gateway will capture that data and have it forwarded to the application defined in The Things Network. Both the destiny application and the device that send the data are identified by private keys (each app has its own and each node has its own key for a specified app), as pictured in Figure 1.

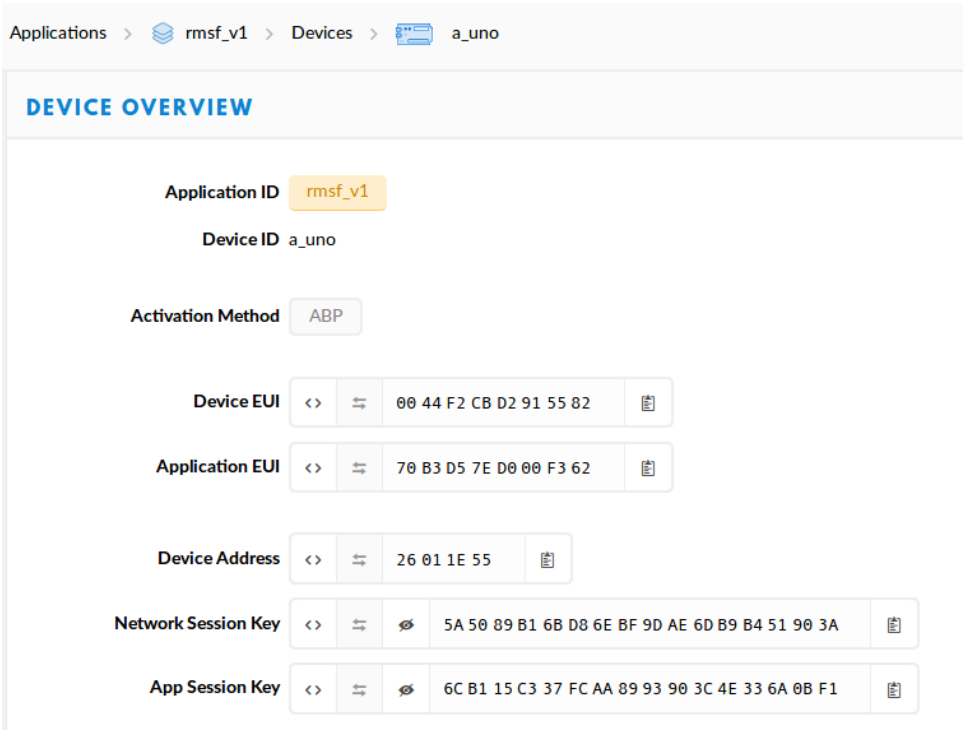

Figure 1: Application in the The Things Network

The are two types of activation methods in the TTN (The Things Network): Over The Air Activation (OTAA) and Activation By Personalizing (ABP). In OTAA, dynamic keys are generated for each session and in ABP, these are set in a static way. In this project, ABP activation is used and the keys are defined as described in Listing 3. These are set utilizing the Most Significant Bit order for the network key and the Least Significant Bit order for the application key. Due to problems with endianess, these must be set in this way for the LMiC library to work properly, although the available information isn't very clear about what formats to use.

Listing 3: ABP key definition.

```
// LoRaWAN NwkSKey, network session key
static const PROGMEM u1_t NWKSKEY[16] = { 0x5A, 0x50, 0x89, 0xB1, 0x6B, 0xD8, 0x6E, 0xBF,
    0x9D, 0xAE, 0x6D, 0xB9, 0xB4, 0x51, 0x90, 0x3A };
// LoRaWAN AppSKey, application session key
static const u1_t PROGMEM APPSKEY[16] = { 0x6C, 0xB1, 0x15, 0xC3, 0x37, 0xFC, 0xAA, 0x89,
    0x93, 0x90, 0x3C, 0x4E, 0x33, 0x6A, 0x0B, 0xF1 };
// LoRaWAN end-device address (DevAddr)
static const u4_t DEVADDR = 0x26011E55;
```

The messages exchanged between the device and the TTN (payloads) are comprised of hexadecimal format bytes that must be decoded in order to use the information sent by the device. Each payload transmitted is comprised of 10 bytes that are used to transmit the device's longitude,latitude, orientation and the action performed. The payload is decoded according with the following decoder function, returning a JSON object. Even though the board's orientation is given in a 16 positions compass, we only use 8 (N,S,E,W,NW,NE,SW, and SE), therefore if three letters are received, we only use the last two.

Listing 4: TTN payload decoder.

```
function Decoder(b, port) {
    var lat = (b[0] | b[1]<<8 | b[2]<<16 | (b[2] & 0x80 ? 0xFF<<24 : 0)) / 10000;
    var lng = (b[3] | b[4]<<8 | b[5]<<16 | (b[5] & 0x80 ? 0xFF<<24 : 0)) / 10000;
    var orient;
    if(0x41<b[6]&& b[6]<0x5A){
        if(0x41<b[7]&&b[7]<0x5A){
            if(0x41<b[8]&&b[8]<0x5A){
                orient=String.fromCharCode(b[7],b[8]);
            }else{
                orient=String.fromCharCode(b[6],b[7]);
            }
        }else{
            orient=String.fromCharCode(b[6]);
        }
    }
    if(b[9]==0x30 || b[9]==0x31 || b[9]==0x32)
        var f = parseInt(String.fromCharCode(b[9]));
    return {
        location: {
            latitude: lat,
            longitude: lng,
            orient: orient,
            func:f
        }
    };
}
```

After receiving the information from the device, TTN's HTTP integration is used in order to send information to the server using the POST method. This is done by setting an endpoint in the TTN, so that the JSON object can be forwarded to that URL. These objects also send an endpoint for downlink communication, so that the server can reply to the device, using the same integration, that also allows for downlink communication, by receiving a JSON object identifying the app and the device to which the message must be sent.

The The Things Network is a LoRaWAN network that provides several different integrations (as HTTP or Storage) and works as a network of gateways created by its user to which every end device can connect to. According to the keys exchanged in the activation (either OTAA or ABP), the payload is forwarded to the

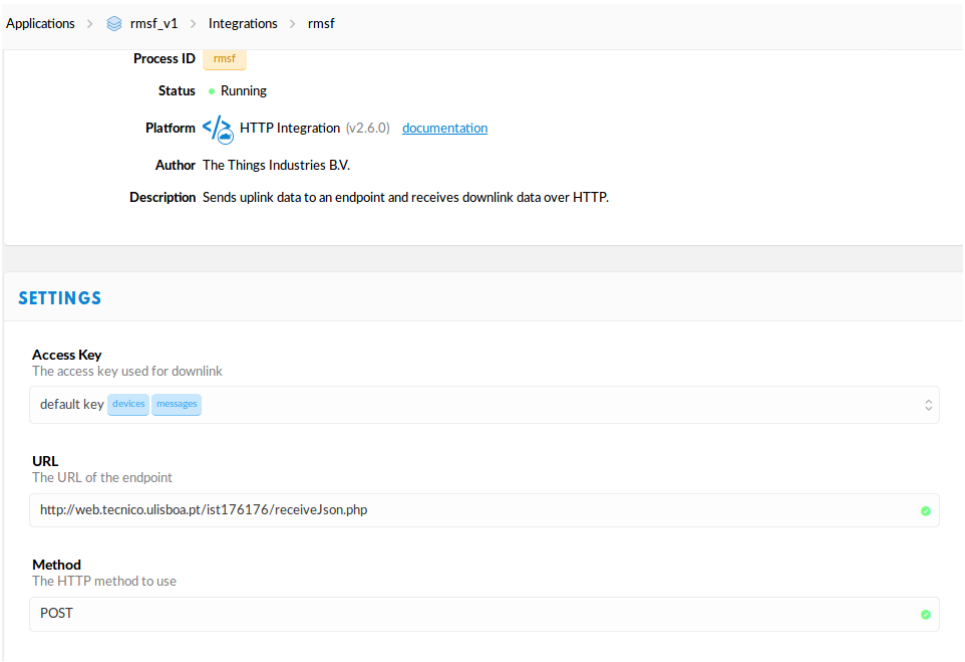

Figure 2: HTTP integration in the The Things Network

correct application. Being free and as resources are limited, the TTN has a set of guidelines (that aren't fully enforced, but users are advised to follow them) to ensure everyone gets to use the network properly. These include:

- A maximum uplink airtime of 30 seconds per node, which gives us around 666 uplink messages per day, or a message every 2 minutes and 10 seconds. It was assumed that a player would play at most 8 hours per day, so around 45 seconds per idle transmission;
- A maximum of 10 downlink messages (which, if enforced, would severely impact this project, since we rely on downlink messages to inform the player of actions that occurred).

4 Arduino

The Arduino board provides the gun to be used. In this section all of the hardware is described, as well as the libraries used and how the device works.

4.1 Components

As previously described in the intermediate report, our gun consists of an Arduino Uno board with Dragino LoRa/GPS shield providing the needed hardware to communicate through the LoRa protocol, being that the Shield is the LoRa transceiver, allowing for long-range communications at low-data rates, providing ultra-long range spread spectrum communication and robustness against interference. With a Quectel L80-R GPS module to receive GPS data and a LoRa Bee to send and receive data, the Shield has all we need to ensure communications are made and GPS data is collected.

For user interaction, a 5V DFRobot Digital Push Button is used (instead of two as mentioned in the intermediate report, due to unresolved issues with the LMiC library) as well as a 5V Buzzer from PTROBOTICS.

Figure 3: Arduino Uno

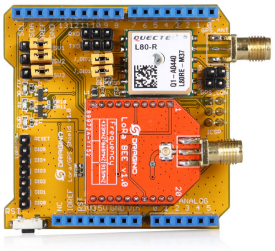

Figure 4: LoRa Shield

Figure 5: 5V Digital Push Button

Figure 6: 5V Buzzer

4.2 Configuration

The LMiC library defines a set of pins to be used, but some are also needed to connect the buzzer, the button and the TX and RX GPS functions. The Arduino board and components are set as described in the following Figure.

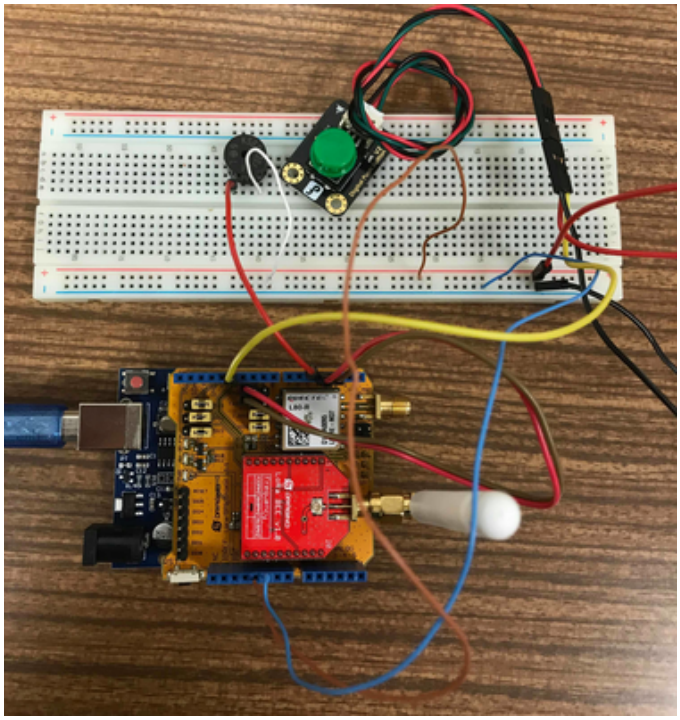

Figure 7: Hardware configuration.

Besides the pins listed in Listing 5, pin 3 is used to connect to GPS TX and 4 to GPS RX. Pin 5 is used for the buzzer and pin 12 for the digital push button.

Listing 5: Pin definition for the LMiC library on the Arduino Uno.

```
// Pin mapping
const LMiC_pinmap LMiC_pins = {
    .nss = 10,
    .rxtx = LMiC_UNUSED_PIN,
    .rst = 9,
    .dio = {2, 6, 7},
};
```

4.3 Libraries

For this project, three external libraries are used: the LMiC library, the TinyGPS library and the SoftwareSerial library.

The SoftwareSerial library is used to create a software serial connection between pin 3 and 4, so that data from the GPS module can be received. This data is then analyzed using the TinyGPS library functions. The values read from longitude and latitude are saved with six decimals, in order to ensure accuracy. The orientation given is not as precise as needed, alternating between several different values if the player is still or moving slowly. With the increase of the movement's speed, the accuracy is slightly better. Other data given by the GPS, as altitude and date, is not used.

The IBM LoRaWAN C-library (LMiC) is a ortable implementation of the LoRa MAC specification for the C programming language. It provides us with the necessary functions to communicate with the LoRaWAN network, working with jobs and events, as described in its documentation. It is important to take note that the available LMiC library in the Arduino IDE is not the correct one, it should be downloaded directly from https://github.com/matthijskooijman/arduino-LMiC, as the version from the IDE is with errors and outdated. This library is responsible for setting the needed parameters for the communication, as the frequency to use, the RX window and the spread factor.

4.4 Functioning

To begin, in the *Setup* section, the buzzer pin and the button pin must be initialized as OUTPUT. The LMiC library has built-in run-time environment to care for timer queues and job management that must be initialized with *os_init()*, followed by a *LMIC_reset()* call to rest the MAC session, discarding all pending data transfers. As a solution for downlink problems related with the timing of the transmissions, the clock error is set using *LMIC_setCLockError(MAX_CLOCK_ERROR *1/100)*, where MAX_CLOCK_ERROR is a pre-defined value from the LMiC library.

The session is then set in a static way, using the keys provided from the TTN and the call to *LMIC_setSession(0x01,DEVADDR,nwkkey,appskey)*. Link check validation is then disabled (*LMIC_setLinkCheckMode(0)*) and spread factor 9 is used for the second window of LoRa RX (where the downlink is received with these library definitions), since this spread factor is the one decided by the TTN for RX2 (*LMIC.dn2Dr = DR_SF9)*. Also, the data rate and the transmit power are set with *LMIC_setDrTxpow(DR_SF7, 14)*.

For the loop section of the program, one must wait for the GPS to be ready, as mentioned before. Then, a transmission can be sent with a press or hold of the button or by doing nothing, since after each transmission a new one is scheduled with *os_setTimedCallback()*. Then, *os_runloop_once()* runs all scheduled jobs, which will be only one.

When a player action is performed, the scheduled jobs are cleared with *os_clearCallback()* and a call to the send job is done (*os_setCallback()*), so that data can be sent to the server at the moment of the action and with the required function value.

The function *GPSRead* takes care of reading the data from the GPS string sent from the module and the function *GPSWrite* writes the data to be sent to the TTN. The *smartdelay* function provides a delay that is only accounted for when there's no information to read from the GPS data.

Every job returns an event, but due to the space available, we chose to ignore all the events related with warnings and unused information for us. This way, the only events used are EV_TXCOMPLETE, where we know the transfer of data to the network has ended, so the program waits for a downlink message to be received and a new transfer is scheduled, and EV_RXCOMPLETE when the data is received.

5 PHP Server & SQL Database

Both the PHP server and the database are stored in Técnico's sigma cluster, that provides free hosting and storage services for students.

the database consists on the storage of three types of objects, as described on Figure 8:

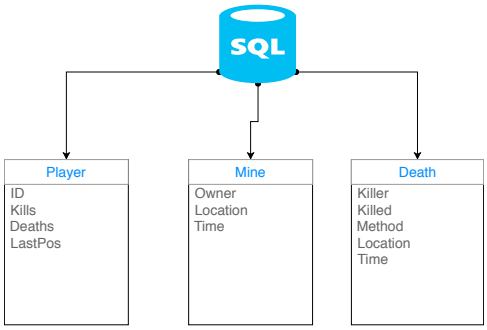

Figure 8: Database scheme.

- Player - Structure with the information relative to each player
  - Reference (ID) of the Arduino corresponding to each player, taking into account that each Arduino is directly related to one and only one player;
  - Record of how many kills the player has;
  - Record of how many deaths the player has;
  - Location, which corresponds to the last known location (latitude and longitude) of each player and that is updated throughout the game.
- Mine - Structure with information relative to each deployed mine
  - Reference of the Arduino corresponding to the player who deployed the mine in order to correctly identify who was the player who deployed it;
  - Location where the mine was deployed (latitude and longitude) which will help on calculation in terms of whether the mine is activated or not;
  - Date and time at which mine was deployed.

- DeathLog - Structure with the information relative to deaths that have occurred, serving as history of the game
  - Reference of the Arduino corresponding to the player who killed another player;
  - Reference of the Arduino corresponding to the player who was killed by another player;
  - Method that was used to kill, it can either be by a “mine” or a “shoot”;
  - Location of where the death occurred (latitude and longitude);
  - Date and time at which death occurred.

As for the PHP server, it will interact with the data stored for different purposes: extract information and create new entries on database or update data already inserted in database. Two pages are used: *receiveJson.php* - to where that data is sent -, and *UserArea.php* - to show some game data.

In *receiveJson.php*, a JSON object, forwarded from the TTN, is received. This object contains the needed information to update a player’s location and its action. An example of a received object can be seen in Figure 9. It its metadata is all needed info to send a downlink reply, simply use that downlink url to send a payload that the TTN will forward to the identified board. To send POST requests, the *curl* function is used. In this page, several actions over the database may occur, depending on the input.

[{"created_at": 1528061683.511259, "data": {"dev_id": "a_uno", "downlink_url": "https://integrations.thethingsnetwork.org/ttn-eu/api/v2/down/rmsf_v1/rmsf?key=ttn-account-v2.-eWlm8pWIfpQLOh70iRQuwSXjFLYRpDN_-fkhBmGJDY", "counter": 0, "app_id": "rmsf_v1", "payload_fields": {"location": {"latitude": 677.6679, "func": 0, "longitude": 776.3574, "orient": "N"}}, "payload_raw": "Z2dndnZ2TgAAMA==", "hardware_serial": "0044F2CBD2915582", "port": 1, "metadata": {"time": "2018-06-03T21:34:43.449859389Z"}}}]

Figure 9: Example JSON string used.

In order for the PHP file to correctly read this data we had to decode this information and store is in variables for future use, including the donwlink URL. After this, the connection to the database is made and check if the user playing with the Arduino is already inserted in the Database Player (using Arduino’s reference), if it is not, it is inserted in the database. We then get to the point where we have to understand which command the Arduino sent and call the function corresponding to each command, as described in the next Listing.

Listing 6: PHP function alternatives.

```
if(\$func == 0){ //PERIODIC UPDATE
   killCheckByMine(\$deviceRef, \$playerLat, \$playerLon, \$conn);
}elseif(\$func==2){ // CLICK ON MINE BUTTON
   addMine(\$deviceRef, \$playerLat, \$playerLon, \$conn);
}elseif(\$func==1){ //CLICK ON SHOOT BUTTON
   killCheckByShoot(\$deviceRef, \$playerLat, \$playerLon,\$orient, \$conn);
}
```

Consequently, according to the command sent by Arduino, a function is called to perform those actions. The functions we have in receiveJson.php are described as follows:

- *killCheckByMine($reference, $playerLat, $playerLon, $conn)* – this function is called periodically according to a timer in the Arduino. Its purpose is to receive a location of a player and check if there are any mines within a radius of his position, if that mine was not deployed by him it will explode. Lastly it sends the information of what happened back to the Arduino encoding the information back to a JSON string using function *sendJson()*;
- *addMine($reference, $mineLat, $mineLon, $conn)* – this function is called when the player performs the command to deploy a mine. It just adds a new entry to the Mine Table;
- *killCheckByShoot($reference, $playerLat, $playerLon,$orient, $conn)* – this function is called when the player performs the command to shoot. According to the location location, this function calculates the

orientation that every other player has related to the player playing the game. This means, it calculates if the other player is NORTH, SOUTH, etc... in relation to the shooters current location. After this it checks if there is any player with an orientation equal to the one sent in the JSON object sent by Arduino, in the last 5 seconds. Lastly it sends the information of what happened back to the Arduino encoding the information back to a JSON string using function sendJson();

- *haversineGreatCircleDistance($playerLat, $playerLon, $latitudeTo, $longitudeTo, $earthRadius)* – this function calculates the distance between two GPS coordinates using the Haversine formula;
- *getCompassDirection($bearing)* - this function calculates related the bearing of each player to an orientation, here we used an 8 direction compass rose;
- *sendJson($reference,$funcaoStr)* – this function encodes the data that we have to send to the board in order to inform the player of what happened, in JSON object to be decoded in the TTN, and redirects it to the correct URL for donwlink in the HTTP integration of the TTN, as extracted from the metadata.

This *userArea.php* file receives the reference of the player playing the game and prints out all the information corresponding to the player. It prints 4 tables, the first contains information relative to the database's Player structure, with its kills, deaths, and location. The second has the mines that the player deployed, their location and date and time of deployment. The third contains information relative to the kills made by that player and the last one contains information relative to the deaths that that player had;

6 Final Remarks

The project was concluded with some success, although changes had to be made from the initial planning until the conclusion of the game.

Project development was hindered due to the fact that LoRa and LoRaWAN are relatively new, therefore there isn't that much information available, even more when applied to our specific hardware.

TTN's limitations are also something to consider for projects like this, as LoRaWAN isn't made for real time data (as per the information provided in TTN's website), be it due to the uplink, airtime, or downlink limitations. An alternative that might be suited would be using LoRa only (having end devices communicating with fixed receiving devices using LoRa modulation), but that would imply a loss of centralization. For both cases, using only LoRa or LoRaWAN, it is needed that the area is covered by other devices to receive the data and that may not always be the case. TTN's approach to the public gateways where any person can have its own gateway and capture LoRa modulated messages, forwarding them to the respective application, is a good approach, making it so that there might be a gateway in each house, supported by its owner, to which everyone can send its messages.

As for hardware improvements, if the TTN is used, using their specific boards is more suited as the needed functions and library are well defined and work properly with small effort while occupying a small portion of the available programming space.

Finally, even though this project may not be suited for a LoRaWAN application, it was an interesting approach to the IoT world, and TTN which proves itself to be very useful with the development of different integrations, tutorials and its own boards, pushing for a more IoT developed world. Ensuring everything is connected at the lowest cost and in the most robust way is something of utmost importance for the next years, since IoT devices and technologies are growing with no signs of stopping. Having an opportunity to look into this technology is of value, as it gave us a small but interesting insight over what might be an IoT future.