# BATA_RACE: Wireless Keypad
# Design Project: Group 16

Aliaksei Kouzel
Danylo Liashenko
Douwe Osinga
Matteo Schut
Ben Van Viegen

*The University of Twente*
*The Netherlands*

April 19, 2024

# Contents

**1 Introduction** **2**

**2 Requirement Analysis** **3**
2.1 Envisioned System . . . 3
2.2 Functional Requirements . . . 4
2.3 Non-Functional Requirements . . . 5

**3 System Design** **6**
3.1 Packet Structure and Reliable Transmission . . . 6
3.2 Software Design . . . 11
3.3 System Validation . . . 13
3.4 User Interaction . . . 14
3.5 Security . . . 15
3.6 Extensibility . . . 16
3.7 Concurrency . . . 16
3.8 Risk Analysis . . . 17

**4 Technical Specifications** **19**
4.1 Software Specifications . . . 19
4.2 Hardware Specifications . . . 20

**5 Testing** **22**
5.1 Testing Strategy . . . 22
5.2 Unit Testing . . . 23
5.3 Integration Testing . . . 23
5.4 System Testing . . . 23
5.5 Performance Testing . . . 24

**6 Evaluation** **27**
6.1 Planning . . . 27
6.2 Responsibilities . . . 27
6.3 End result . . . 28
6.4 Future Improvements . . . 29

**Appendices** **31**

**Bibliography** **34**

# Chapter 1

# Introduction

The Batavierenrace is the biggest relay race in the world. Around 8500 runners participate each year, 300 of whom compete simultaneously, running all the way from Nijmegen to Enschede. Teams pass through 24 relay points in total, where at each point, each runner's time is calculated and another runner of the team will continue the race.

To prevent runners from blocking the race track while waiting for their teammates at each relay point, a screen is placed on top of each point to display the team number of the runner currently approaching it. This way, the next runner can come onto the track only when their teammate is nearing the relay point.

Currently, the way runner numbers are put on the display requires two people, two walkie-talkies, a keypad and a piece of paper. One person stands 100m from the relay point, looking out for approaching runners. When a runner is spotted, this person writes their number down on a piece of paper and communicates the number via walkie-talkie to another person, sitting next to the relay point. This other person then enters the number using the keypad so it is displayed on the screen. Two people are needed because the keypad is currently physically connected to the control box of the relay point (the registratiekast, RK in short).

Ideally, there should only be one person standing 100m from the relay point, typing in the runner numbers directly. However, with the current wired solution, that would result in kilometres of wire to wire up all the keypads to all the relay points, each hundred-metre wire having to go through sometimes quite muddy and harsh terrain. Since part of the race is also held in the dark, these wires would not only become incredibly dirty but may also be tripped over, causing safety hazards and breakages.

For a few years, the Batavierenrace has been gradually moving over from wired to wireless solutions, and this system is one of the last ones left. Thus, the request from the eBART committee, the engineers keeping the Batavierenrace up and running, is to make this currently wired keypad wireless.

# Chapter 2

# Requirement Analysis

After the first meeting with the client, a list of requirements detailing the final functionality was presented. After some discussion and some small tweaks, the client and the team agreed on the requirements. In the end, quite little had to be changed, as the requirements as presented by the client were already quite clear and listed in MoSCoW format.

Below is a brief explanation of the envisioned system by the client, along with the final requirements.

## 2.1 Envisioned System

### 2.1.1 General Overview

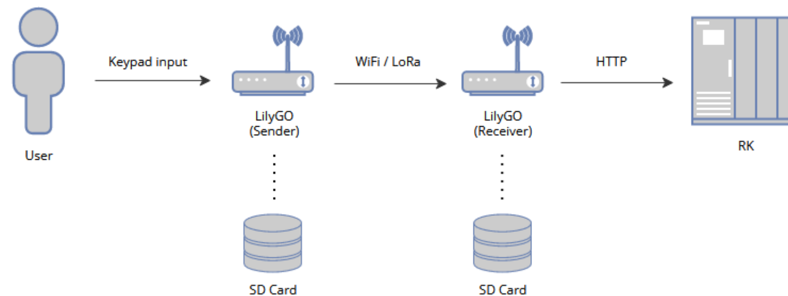


Figure 2.1: General data flow

The envisioned system by the client can be summarised into the following:

The solution should consist of two LilyGO nodes, equipped with both LoRa and Wi-Fi connectivity for long and short-range communication, respectively. One node acts as a sender and is operated by a human, standing approximately 100 meters from the relay point. The second node is used as a receiver, located near the relay point and the RK. The system must ensure a reliable transmission between sender and receiver throughout the race (at least for 12 hours straight, possibly for longer, if the keypad is not turned off immediately after the race). Both nodes should ideally be equipped with SD cards, which should be used for logging and synchronization purposes.

If the LoRa connection drops, the data should be stored as a backup on a sender's SD card. This data should get synced to the RK manually, by placing the sender in range of the Wi-Fi Access Point at the relay point and pressing a 'Sync' button. Additionally, if the LoRa link is severed completely, the nodes should be able to communicate over Wi-Fi instead, given that both nodes are in the range of the Wi-Fi Access Point.

Additionally, on the receiver's side, Wi-Fi should be used for communicating runner numbers with the RK via HTTP requests. The RK is used as the "single source of truth" and acts as a medium between the receiver and the display for runners.

## 2.2 Functional Requirements

### Must

- The system must be able to transmit numbers ranging from 1-999.
- The system must use a LoRa connection as a ‘base communication link’.
- The receiver must forward received numbers via the HTTP to the RK.
- The sender and receiver must have a hard-coded ID
  - each RK has a unique number already: $'A'$-$'I'$ with reserves $'R1'$-$'R3'$, which can be used for the link as well.

#### Reliability

- The system must be able to detect corrupted packets.
  - When a corrupted packet is detected, this packet does not need to be processed.
- The system must be able to recognise and retransmit lost packets.
- The system must be able to detect when a connection is established/lost
- The system must be able to communicate to an operator that a connection is established/lost

#### Backup

- The system must back up entered numbers by saving them on a local storage device on the sender.
- The numbers saved on the local storage medium must be stored in chronological order.
- The system should be able to synchronise the backed-up numbers from the sender to the receiver.
- The system must be able to deal with (temporary) connection loss.

### Should

- The received numbers should be backed up on the local storage medium on the receiver.
- The sender and receiver should be able to communicate with Wi-Fi as a backup link, instead of LoRa.
- The system should be able to mark any sent number as ‘deleted’.
- The system should be able to deal with duplicate numbers by storing them.
- The system should display the connection status of the current link.
- The system should display the state of synchronisation.
- The system should display if there were any changes after synchronisation.

#### Power Saving

- The system should turn off any power-consuming electronics (e.g. antennas) if they are not being used at a given moment.

### Could

- The system could save a timestamp next to each entered number.
- The system demo could include a keypad and a display.
- The system could display the reason behind why the synchronisation failed (e.g. connection loss).
- The system could display the last number sent.

## 2.3 Non-Functional Requirements

### Must

- The system must be able to transmit a number with LoRa correctly at least 99.9 percent of the time, including retransmissions, within at least 10 seconds [1]. The goal should be to not miss a single runner in a race.
- The system must work reliably at a distance of at least 100m.
- The system must remain operational for at least 12 hours once per year.
- The project must be "hand-over ready".
  - The project must be able to be understood within some time by an experienced Technical Computer Scientist.
  - The documentation must provide at least a general overview of how each component works and how it interfaces with the system.

#### Security

- The system must not be hackable or exploitable using "layman's" tools.
  - No open Wi-Fi access points, or other blatant security flaws.
  - No need to ensure safe Wi-Fi communication, as those packets are already secured by the Wi-Fi network's encryption.

### Should

- The design project deliverable should include performance metrics (data transfer rates, lost packets, percentage of retransmitted packets, effective range) about the LoRa connection.
  - With a particular focus on the target distance of 100m.
  - Including at least tests for (non-)obstructed line of sight and different weather conditions.
- The design project deliverable should include performance metrics about the Wi-Fi connection.
- The system should consume "as little power as possible", e.g.:
  - Turning off antennas when not in use.
  - Limiting the amount of power-consuming operations (e.g. connection checks).
- The system should work at ranges larger than 100m (101-200m).

### Could

- The system could work at ranges well over 100m (201-500m).

[1] Assuming the fastest runners run roughly at max. $22km/h \approx 6.11m/s$ ([1], figure 2). Assuming the operator is standing $100m$ from the relay point and the time for a runner to get ready is $\approx 5s$, the maximum permitted time to transmit is $\frac{100m}{6.11m/s} - 5s \approx 11s$, rounding it down to 10s to account for other time-consuming tasks such as the operator typing in the number.

# Chapter 3

# System Design

The following chapter is dedicated to showcasing the final design. This design aims to incorporate all requirements, while offering flexibility in the places that most need it, such as adding new input devices, or making new outputs to present the status of the system (e.g. connection status, syncing status) or to serve as an output for sent or received numbers.

Along with an explanation about the packet structure and general design, sections are dedicated verifying the working of the system, describing how the user interacts with the system, detailing the security measures, describing where and how to extend the current system, explaining where concurrency was chosen in the project and for what reasons, and analysing some risks.

## 3.1 Packet Structure and Reliable Transmission

### 3.1.1 Packet Structure

Due to the quite specific use case that this project demands, and the low throughput that LoRa offers, it was decided that a custom protocol needed to be created to ensure fast and reliable transmission. In fig. 3.1, the structure can be viewed.

This protocol only includes the bare minimum needed to transfer data from sender to receiver, such as only including one ID, which is shared by the sender and receiver. Furthermore, the size of all elements of the header has been shrunk as much as possible, such as only having 10 bits for the runner number (just barely enough to support the numbers 1-999), and only having 6 bits for the sender and receiver ID (to allow for as much space in the number of relay points (section 2.2) while still leaving two bits to decide the packet type.

Speaking of the different packet types, this protocol includes support for three (and up to four) different packet types. Currently, these types are the FlagPacket, NumberPacket and SyncPacket:

- The FlagPacket (fig. 3.1, top row) is used for communicating acknowledgements and PINGs (for checking the connection status when no packets are sent).
- The NumberPacket (fig. 3.1, central row) is used by the sender to send a number entry (sequence number, runner number, and purpose such as add/delete) to the receiver.
- The SyncPacket (fig. 3.1, bottom row) is used by the sender, exclusively in Wi-Fi mode, to communicate a large number of entries to the receiver, in the case the operator wants to synchronise the entries from the sender to the receiver.

#### Reliable Transmission

One of the most important requirements of this project is the reliable transmission of numbers. To achieve this, a stripped-down version of TCP was chosen. A sliding window mechanism was considered and partially implemented, but due to the realisation that there is practically no propagation delay and because of the lack of medium access control for LoRa, this idea was dropped and replaced by a more simplistic approach.

This approach, while simpler, turned out to satisfy the requirements even better, and the simplicity of it makes it easier to maintain for the client. This protocol is a version of the stop-and-wait protocol, with a maximum time-to-live per packet, and a minimum number of transmissions per packet (explained in the following

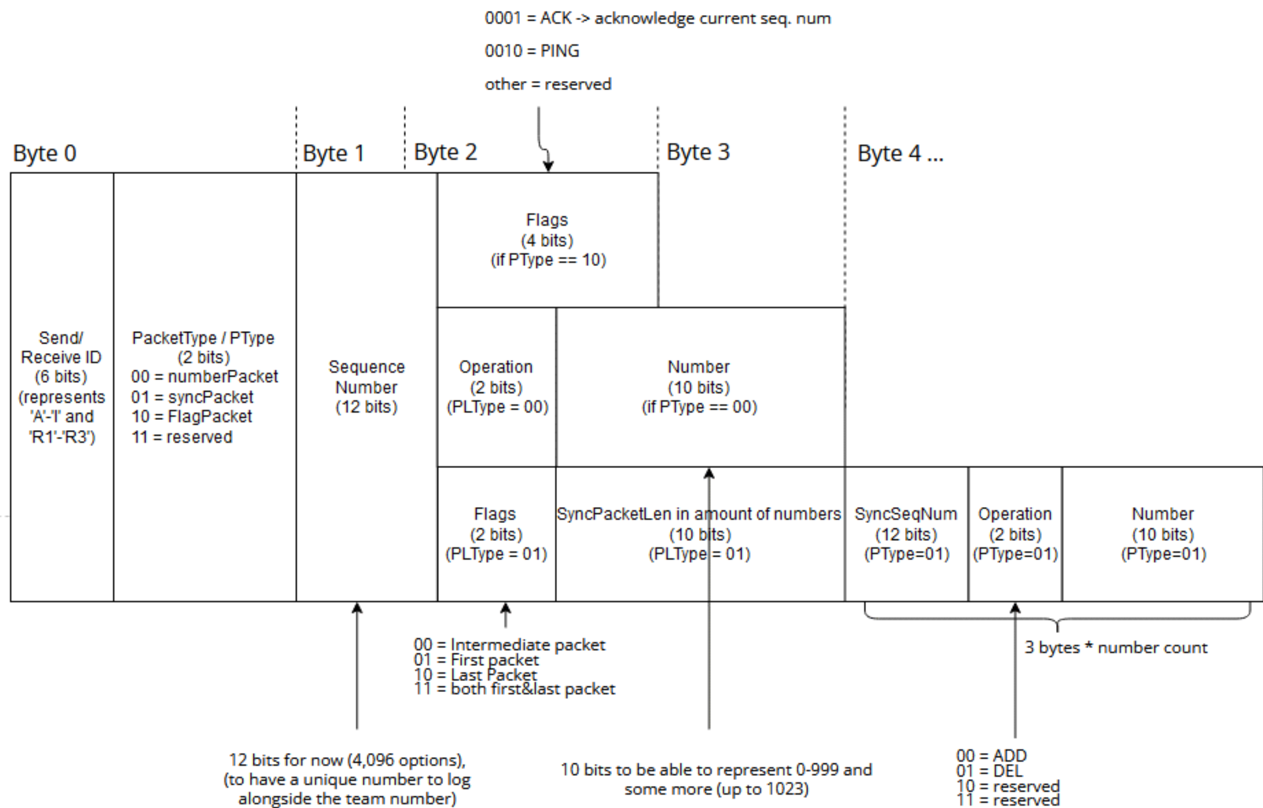


Figure 3.1: Packet Structure

paragraph).

In this protocol, the sender sends one message and waits for the receiver to send an acknowledgement. The sender is only allowed to continue to send the next packet after it receives the acknowledgement for the previous packet, or retransmit the previous packet when the retransmission timeout has passed and the receiver has not sent an acknowledgement ([2] section 2.5.1). The key deviation from the original stop-and-wait was to implement a maximum Time To Live (TTL, in the code called $MAX_PACKET_RETRANSMISSION_TIME$). This was decided because in the context of the Batavierenrace, numbers (and thus packets) should arrive within a given time, otherwise the point of displaying the number is defeated, as the number has already arrived at the relay point. This means that the longer a packet is delayed, the less valuable it becomes. If there would just be a set number of retransmissions per packet, the queue could get congested with old packets that do not matter anymore and could delay new packets that have more significance.

This means that if the TTL expires, the packet will not be retransmitted anymore, if it has already been transmitted a set number of times. Including the minimum transmissions makes sure that each number at least has a chance of getting received, even though the receiver's acknowledgement might never arrive.

This combination of a maximum TTL and minimum transmission count results in packets getting dropped if they are already in the queue for a long time and have been retransmitted already, therefore making space for new packets. This protocol is shown in fig. 3.3. A more general overview of the interaction between the sender, the receiver, the operator, and the RK can be seen in fig. 3.2

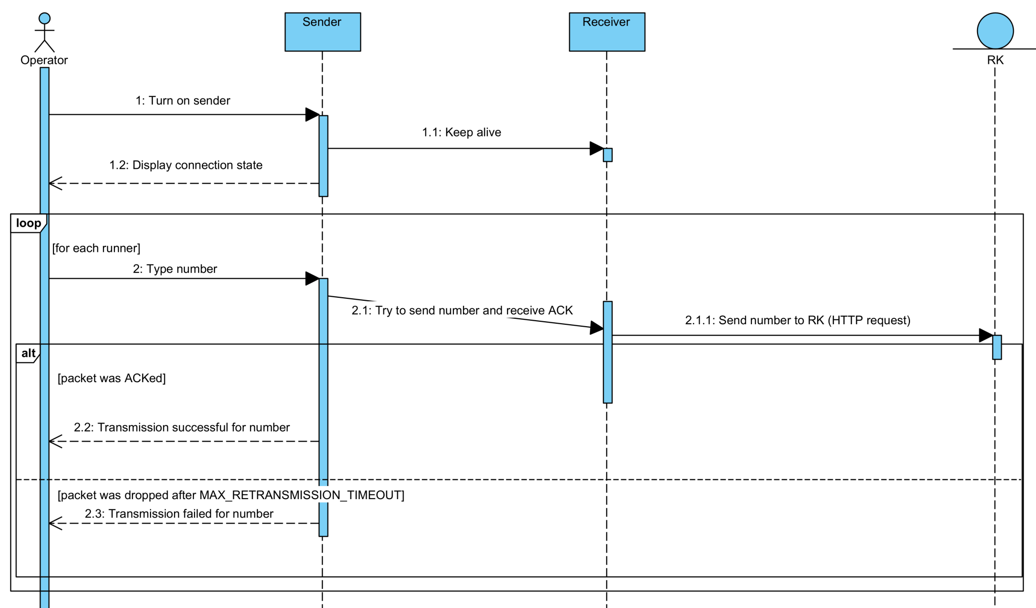


Figure 3.2: Outline of the main feedback loop between the operator, sender, receiver and RK.

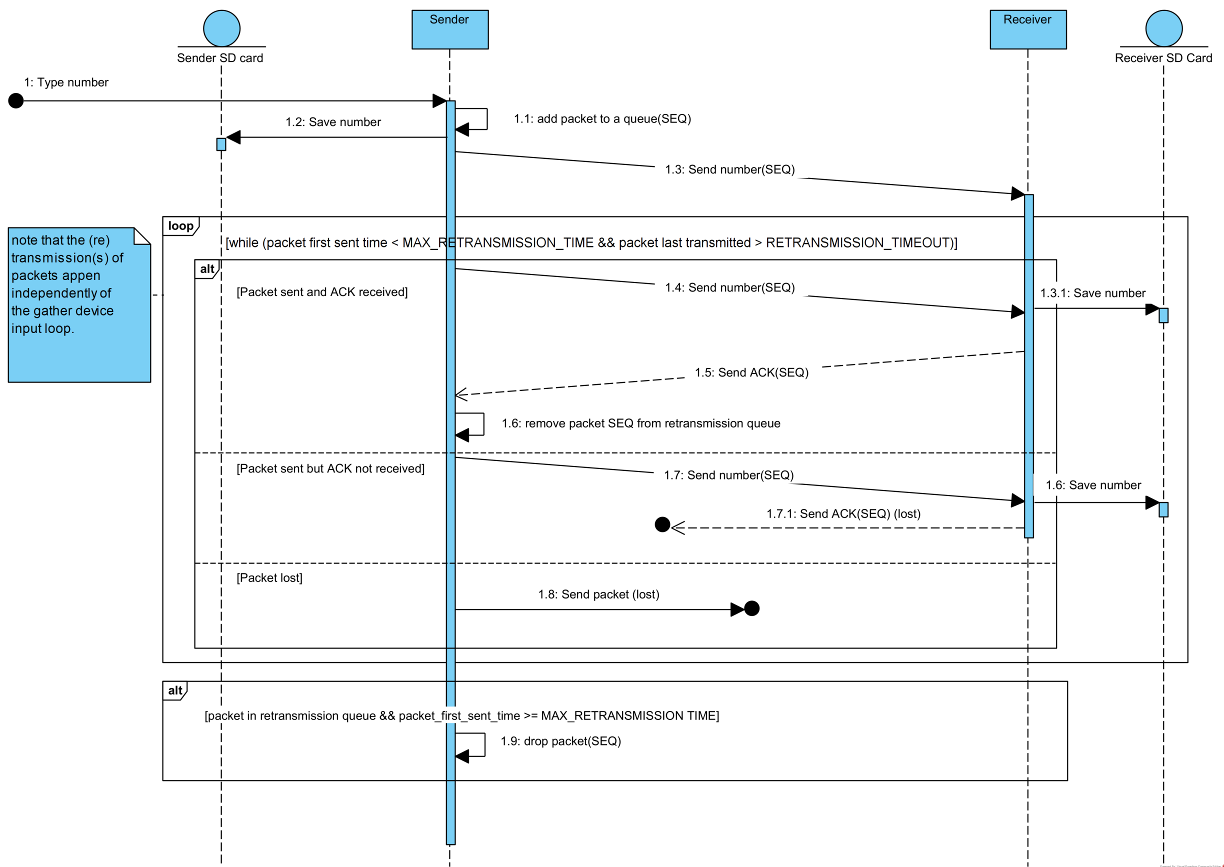


Figure 3.3: A detailed look into number transmission (point 'try to send number' in fig. 3.2).

### Connection / Keep Alive

Since numbers are not constantly being sent (on the contrary, there might be long periods where no runner might be approaching the relay point), the system includes PINGing functionality to keep track of the connection quality while the link would otherwise be idle. The pinging mechanism can be seen in more detail in fig. 3.4.

### Synchronising

In fig. 3.5, the synchronisation logic between sender and receiver can be observed. Note that while the sender and receiver are synchronising their backups, the sender is *not* allowed to transmit any number of entries. This ensures that the backup process does not mess with the ordering of the numbers (the sequence numbers will still remain ascending, instead of a new number being inserted in between two sets of old entries received by syncing).

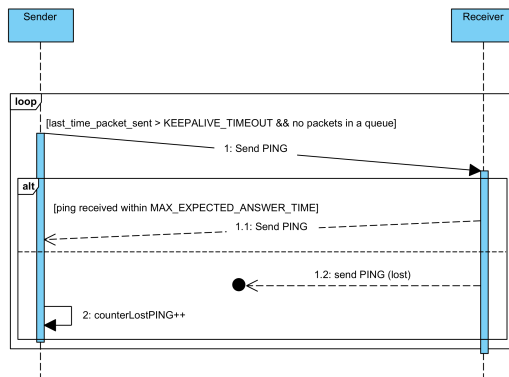


Figure 3.4: Keepalive logic for monitoring the connection between sender and receiver.

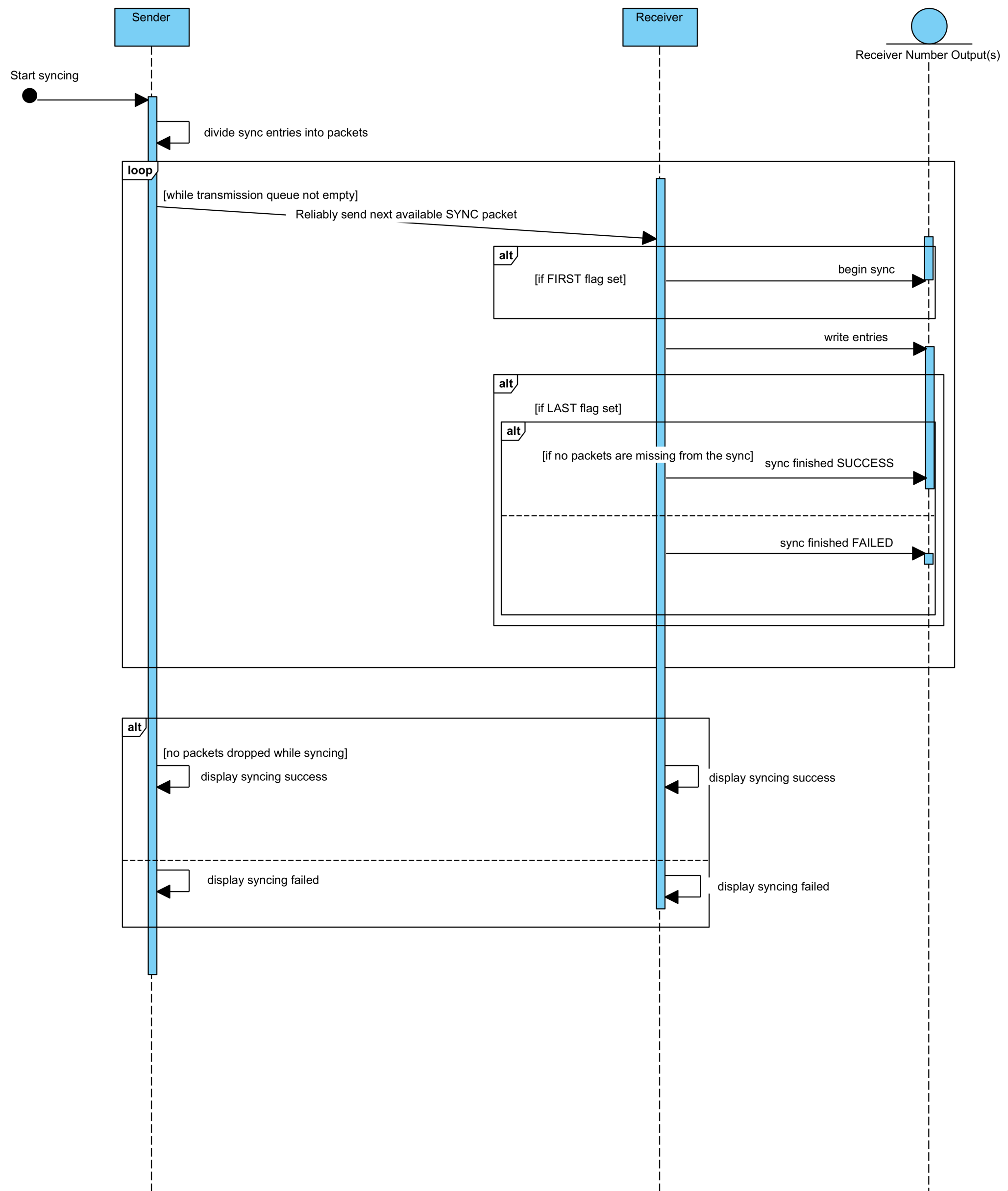


Figure 3.5: Synchronisation logic between sender and receiver.

## 3.2 Software Design

### 3.2.1 Class Diagram

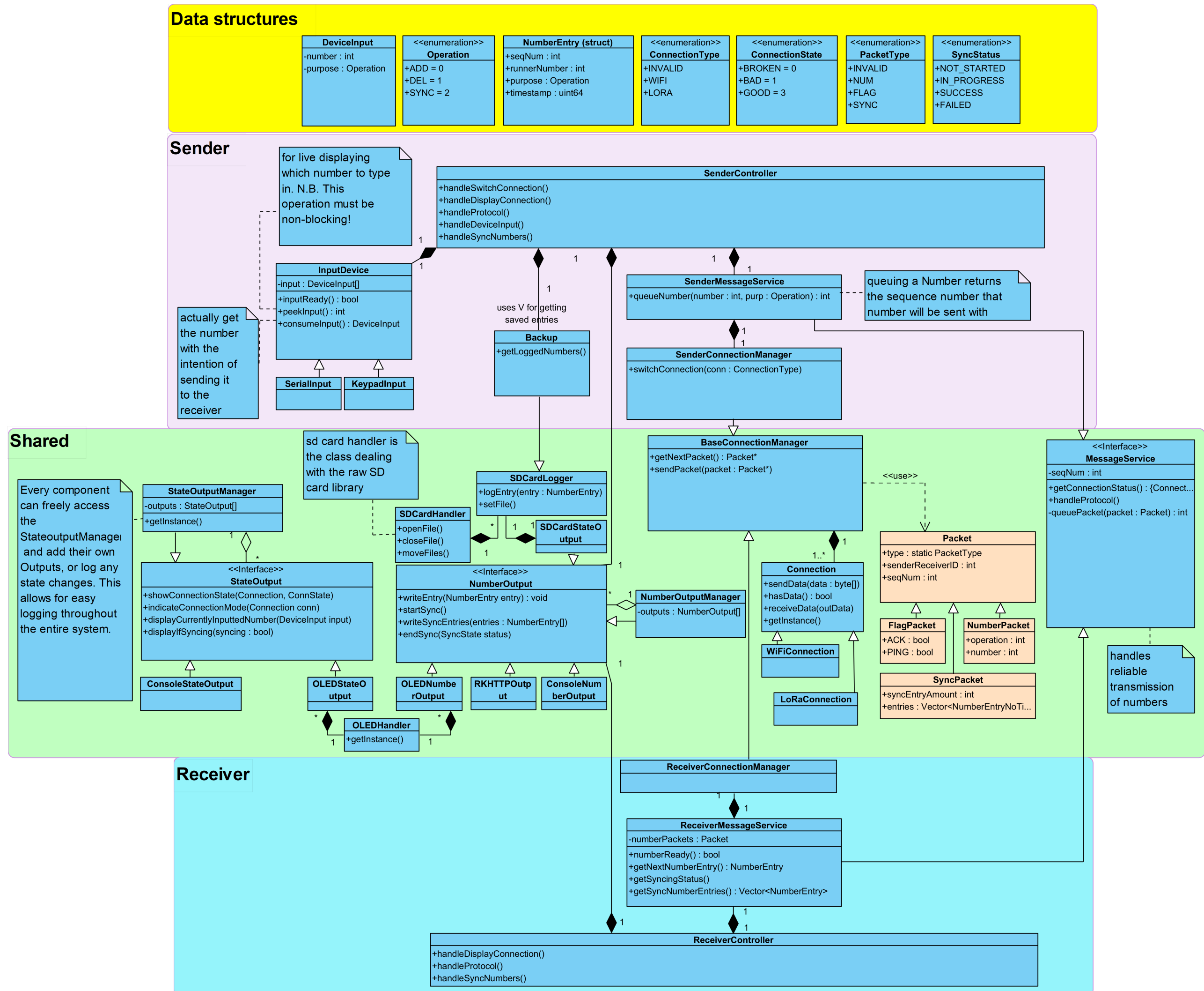


Figure 3.6: System Overview.

In fig. 3.6, the outline of the main class structure can be observed. This diagram is structured in the following 'blocks': Data structures, Sender, Receiver, and Shared. Firstly, a look will be taken at how an input by the operator traverses through the classes. Secondly, the purpose and contents of the 'blocks' are explained.

#### Data flow

Number entries inputted by the operator arrive at the *InputDevice* (fig. 3.6, Sender block). When *handleDeviceInput*() is called in *SenderController*, the *InputDevice* is polled for possible inputs. If the *InputDevice* contains an input, the *SenderController* will consume that input and will queue this number in the *SenderMessageService*. This message service ensures the reliable delivery of numbers and synchronisation messages. This function will also return the sequence number the entry gets for reliable transmission, which serves multiple functions, namely to later be able to identify the order of entries.

After the sequence number is added to the device input in the *SenderController*, the entry (a number, purpose (ADD/DELETE) and a sequence number) is passed to the *NumberOutput*, which can be either a single output, like an SD card output or a console output, or a Manager which can contain multiple separate outputs. Having *NumberOutputs* in the Sender allows us to easily add an SD card or a screen to print the sent numbers.

Once the *SenderConnectionManager* is ready to send the packet, it calls the *ConnectionManager*'s *sendPacket*() function. This will send the packet through the currently used medium. This abstraction allows for completely

switching out the underlying medium, while still keeping sequence numbers and packet structure consistent.

After the packet is received on the receiver side, the receiver's connection classes will convert the received data back to a packet, at which point it will arrive at the *ReceiverConnectionManager*. If the packet is a number or sync entry, it gets picked up by the *ReceiverMessageService*. This service is responsible for acknowledging the packets, and keeping track of the state of synchronisation if the receiver initiated it. Any numbers or synchronisation entries that the *ReceiverMessageService* receives are stored for the *ReceiverController* to pick up. The *ReceiverController* is responsible for the more high-level functionalities and employs the *ReceiverMessageService* for the reliable receiving of packets. Once this controller gets the number entry or sync entries, it outputs these to the *NumberOutput*, in a similar fashion to the *SenderController*.

**Explanation of 'blocks'**

The **Data Structures** section outlines the various data structures shared between multiple components of the system. For the sake of simplicity, associations have not been drawn between which components use which data structures, as this would make the diagram significantly less readable. Particularly interesting data structures are the *DeviceInput*, which contains a number along with an operation, which can either be ADD or DELETE. This way, an operator can indicate whether they wish to display a number, cancel a number from being displayed, or mark this number as an error.

The **Sender** and **Receiver** sections contain the particular classes solely used by either the sender controller or the receiver controller, the components responsible for managing the high-level functionality, such as sending/receiving numbers and displaying state.
One example of this high-level functionality for the Sender is the gathering of user input. This abstraction can be seen in the *InputDevice*, which the sender uses to get the inputs from. This allows for switching out the input to a dummy input for easily testing the sender controller, or easily switching out the input used in the final product for another keypad or serial input.

The **Shared** section contains all classes shared by the Sender and Receiver since the Sender and Receiver have some shared functionality. One example of this shared functionality is the Connection class structure that both the Sender and Receiver use. Since we require bidirectional communication over both LoRa and Wi-Fi, it makes sense to share these abstractions of the raw connections between both the sender and receiver. While these Connection classes function the same regardless of which controller is using it, the *ConnectionManager*s, which keep track of and use the connections, function quite a bit differently for the Sender and the Receiver. The *SenderConnectionManager* can only use one underlying *Connection* object at a time, and contains logic for retransmissions, while the *ReceiverConnectionManager* checks all connections, and further only contains acknowledgment logic.

## 3.3 System Validation

Two of the key parts of the system are data integrity and data safety. This means data has to be safely, consistently, and accurately sent and received. In order to ensure this, the system needs to be able to deal with corrupted packets, lost packets, and dropped connections, among a host of other issues. For this, several backup strategies have been put into place.

This section will focus on the data transfer between the sender and receiver, explaining various backup strategies that aim to account for different types of failures.

### 3.3.1 Data Integrity

#### Hardware Error Detection

The main communication happens via LoRa (see fig. 3.7.1). Since this is a wireless link, the channel is not perfect, meaning there is a nonzero chance of packet corruption. Luckily, the LoRa modules that come with the LilyGO boards provide some hardware functionality for this: the transceivers come equipped with a 16-bit hardware CRC [3], allowing for detection of "any contiguous burst of errors shorter than [16 bits], any odd number of errors (...), and every possible arrangement of 1, 2, or 3-bit errors" [4] [5].

Of course, error detection algorithms can only do so much, so packets can still arrive with a set of bit errors in such a way that the CRC checks pass while the packet is severely corrupted. This issue is partially mitigated by checking for valid sequence numbers, and runner numbers, along with the other fields in the packet, but in the end, there may be some packets that get through to the application layer with bit errors. However, we believe that this is a minor issue, partially because this type of error is quite uncommon, as the packet needs to be corrupted in a very particular way, and partially because *if* such as packet gets passed on to the application layer, the worst that could happen is that a person gets ready while their teammate is not yet nearing the relay point. *If* this were to happen, the log of the receiver would provide transparency of which number was corrupted, and corruption would be apparent by looking at the entries on the SD cards with the same sequence number, but a different runner number.

#### Sequence Numbers

Speaking of sequence numbers, sequence numbers are used to ensure the in-order delivery of packets. In addition, these sequence numbers are reused as a unique identifier for runner numbers. Storing the sequence numbers along with the sent numbers allows for identifying the order of operations if packets get reordered due to retransmissions or other causes.

Note that the ordering of packets can also be secured by means of a timestamp. However, this would require a hardware clock in order to provide a true chronological history of events regardless of potential reboots, and since a timestamp is an extra requirement, this is not implemented as of now.

### 3.3.2 Data Safety

#### Retransmissions

If the packets fail the CRC check, they are by default dropped by the LoRa modules [3]. When this happens, or when a packet is dropped due to some other reason, the receiver does not send an acknowledgment back to the receiver, which will cause the sender to retransmit the packet (fig. 3.7.2). The receiver will keep trying to retransmit the packet, indicating to the operator that the connection is getting worse, until the packet is finally acknowledged.

#### Wi-Fi

If, however, the operator decides it has taken too long for the packet to transmit, there is the option to switch to a backup link: Wi-Fi. This way, even if the LoRa connection is severed, the sender can still communicate with the receiver (fig. 3.7.3).

Here, checking corrupted packets is also handled in the lower layers, by means of a 32-bit CRC ([2], section 2.7 "Frame Format"), double the size of LoRa.

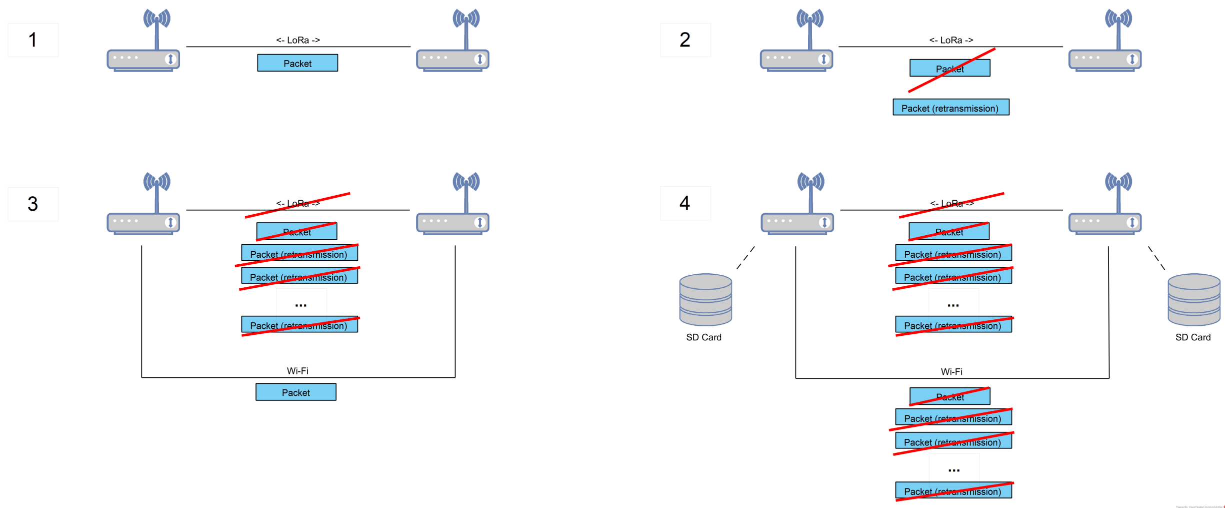


Figure 3.7: Backup strategies for communication between sender/receiver

#### Local storage

If both links, LoRa and Wi-Fi, are down, and no data can be transferred whatsoever by both links for a considerable amount of time, all is not lost. While the main function of the system may not work, i.e., displaying numbers on the screen, as numbers cannot be communicated to the receiver, the sender *does* save a backup on local storage, as does the receiver (fig. 3.7.4). This ensures that if the receiver dies while it has not communicated all the numbers to the RK, the sender still has all the numbers sent, and vice versa.

This backup can also be synchronised with the receiver, by turning on the 'Sync' mode on the sender, which will prompt it to send the entire log of runners to the receiver, which will in turn also send this backup to the RK. Note that the receiver will not *replace* their own contents with the sender's, as this would theoretically allow an attacker or a series of wrong packets to corrupt the receiver's SD card, but will only log that synchronisation packets have arrived.

#### The Inevitable

Of course, there are also scenarios in which all these backups combined cannot save the data. In the case that the links fail, a number or multiple numbers have not been sent to the receiver yet *and* the SD card fails on the sender, some data loss may occur. Data loss is also inevitable when both the SD cards of the sender *and* the receiver fail at the same time, and the connection is severed.

However, we believe these scenarios are so rare to occur that we are confident that, with these backup strategies, the system is about as robust as we can make it.

## 3.4 User Interaction

### 3.4.1 Instruction manual

An instruction manual can be viewed in Appendix B - Operator Manual. This manual is meant for providing a guide for the operator in how to use the end system. This includes topics such as how to start the system, how to send numbers, and how to deal with errors or undesired behaviour.

### 3.4.2 Keypad

The system includes an *Input* class, which is an abstraction for the user input, allowing users to indicate which number they would like to enter, and whether they would like to display these on the screen (ADD) or mark these as old or incorrect (DELETE). This *Input* class can be easily extended to feature, for example, a *KeypadInput*.

### 3.4.3 Switches

Next to being able to enter numbers, the *Input* class also contains a few 'indicate' methods (e.g. *indicateSync*, *indicateConnectionType*). These 'indicate' methods are meant as abstractions for the eventual switches on the physical sender keypad. These switches control important functionalities of the system, such as starting the synchronisation process between connected devices[1], or switching the connection from LoRa to Wi-Fi or back, allowing operators to use the Wi-Fi back-up link for entering numbers when the LoRa link is down.

### 3.4.4 Output Screen

For things such as displaying the strength of the currently used connection, indicating the reliability of the network, error messages, for alerting the operator of any issues, indicating the synchronization status, and displaying the currently inputted number, the system uses the *StateOutput* class. This class has been designed so that one can easily add a particular screen later which can display (part of) these state outputs. An example implementation can be viewed in fig. 3.8.

Figure 3.8: Overview of the OLED Status and Number output

## 3.5 Security

### 3.5.1 LoRa

Due to LoRa not being a commonly used technology, and due to the low impact of possible attacks, security was decided to be of low priority. Even if an attacker were standing in range of the sender/receiver with a compatible LoRa module, and would have figured out the correct packet structure, the most damage that could be done would be displaying runner numbers on the large screen and messing up the backup on the receiver side. This would only cause some inconvenience for the runners since incorrect runner numbers would then be displayed and it would make the backup on the receiver side littered with some incorrect data.

A point could be made about making 'delete' packets, which is why it was decided to not actually delete the number from the backup, but instead add a new entry stating the deletion of a number. This way, the attacker cannot destroy anything from the receiver's backup. In the case that the receiver's backup would be littered with malicious entries, the sender's backup, which cannot be corrupted by malicious transmissions, can be used,

[1]Note that in order to start synchronising, the connection type must be set to Wi-Fi

and no harm would be done. This scenario would only become a problem when the attacker spams the receiver with entries and the sender's backup is corrupted in some way.

In conclusion, the harm a malicious actor can cause over the LoRa link is little, so little that security for the LoRa communication was deemed of low priority and thus not implemented due to the prioritisation of other features.

### 3.5.2 Wi-Fi

For communication over Wi-Fi the Wi-Fi network of the relay point is used. This network uses the standard WPA2 security protocol. According to the product owner, the assumption can be made that only members of the relay point team have access to this network, so there should be no malicious actors that have access. Even if a malicious actor had access, they would still have to figure out the packet structure, and the harm they could cause is again very little. Therefore, relying on the WPA2 security protocol of the Wi-Fi Access Point should be secure enough for this application.

## 3.6 Extensibility

A key design element in the system is horizontal extensibility, which is the act of building an object-oriented program with the intent to "help support the system quality attributes of modifiability, maintainability and scalability"[6]. In order to support extensibility, in this context, allowing developers of the eBART committee to easily add their own input and output devices, is to have interface classes for each extendable component, and to separate groups of components into folders.

The main components that have been designed to be added to the system in the future are inputs (*Input*), outputs (*NumberOutput*, *StateOutput*) and new connections (*Connection*). Adding new inputs and outputs is as easy as extending the particular interface and implementing the methods. The input/output can then be swapped or added to the sender or receiver controller on startup.
For each extendable component, a detailed explanation is given on how to create one in the ReadME file in its folder.

## 3.7 Concurrency

### 3.7.1 Threading strategy

In this project, several things sparked discussion about threading. Things such as sending data via a connection takes time, and requests that require an answer from another party (for example, HTTP requests) might time out. Hence, a concurrency strategy needed to be thought of. Since this project is microcontroller-based, and threading can become quite expensive to run on these small devices, the strategy used in this project is to try and use one thread as much as possible, and only letting tasks that take significant"[2] time have their own, individual thread. This makes reasoning about control flows easier as mostly everything is executed in sequence, with the exception of the most time-consuming tasks. Striving to keep most things on one thread also largely avoids dealing with concurrency issues, such as deadlocks and race conditions. Any multi-threading done on the LilyGOs has been carefully examined for shared variables, whose interactions have been wrapped in mutexes to ensure thread safety.

As mentioned before, not all processes were able to be put on a single thread. In particular, two critical operations are most likely to take a "significant"[2] amount of time, namely the code responsible for executing HTTP requests to the RK and the code for connecting and sending/receiving data via Wi-Fi.

#### HTTP thread

During testing, it was found that, when sending an HTTP request via the *HTTPClient* library, it will block the current thread while waiting for a response until it either receives a response or the request times out. If the LilyGO is connected to the network but the RK is not, the HTTP request will wait for its maximum duration, which can take up to two seconds. Since blocking all operation on the receiver side for two seconds would interfere with the reliable transmission, and significantly decrease the responsiveness of the system, it

[2]Tasks that take "significant" are tasks that would disrupt the control flow and timing of other parts of the system. For example, waiting one or two seconds for a timed-out HTTP request would severely cripple the message service's ability to reliably send packets

was decided to reserve a separate thread for sending these HTTP requests.

#### Wi-Fi thread

When switching to a Wi-Fi connection, connecting to the network may take more than a few seconds. Additionally, during this waiting time, ideally, one might still want to queue packets to be sent after the connection is established. Another reason for putting Wi-Fi on a separate thread is the transmission delay. As only small packets are sent via LoRa, it does not experience a large delay[3], but Wi-Fi also needs to handle synchronisation packets, which for a large entry count can take several hundred milliseconds, again messing with the timing of reliable transmission and such. Due to these reasons, it was decided to put the transmission and reception of Wi-Fi packets on its own thread as well.

To ensure that the Wi-Fi is disconnected fully upon calling $WiFiConnection :: disconnect()$, the main thread will wait to join with the send/receive thread before proceeding with ordinary execution, ensuring no message can be sent or received after disconnecting Wi-Fi.

## 3.8 Risk Analysis

As reliability is the main focus of the system, it is necessary to address potential errors that may occur during its operation.

The risks associated with the system itself and the designed solutions are:

1. A packet is damaged:
   Since the LilyGo's LoRa module implements a 16-bit hardware CRC, if a packet arrives damaged, it will most likely be dropped. Some severely damaged packets might still make it through transmission though if the damage is quite specific causing it to still pass the CRC. Though, even if this happens, it will likely be caught by the receiver when trying to construct a packet out of this data, since the data most likely is not correct. However, there is no guarantee that a damaged packet is caught.

2. A transmitting packet is lost:
   The implemented protocol will handle retransmissions.

3. The connection was weakened for a short period:
   This could lead to the transmission queue being too big, if inputs are also sent too fast during a weakened connection. Since the next packet in a queue could only be sent after the current one is either acknowledged or dropped, and the countdown for dropping a packet starts immediately after it was queued. In case of a bad connection, when the first packet in a queue is dropped, the second one will have fewer retransmission attempts before it is also dropped. That also depends on the time the second packet was queued. However, to avoid a situation where the packet is dropped before it is even sent once, each packet has to be transmitted at least once before the dropping. There is also some delay in between sending and dropping to account for possible acknowledgement, that might follow in case the connection is restored.

4. LoRa or Wi-Fi connection is lost between sender and receiver:
   In case one link failure it is possible to switch to the other type of connection. The current queue of numbers to be sent and the sequence number of the packets remain the same. However, in case both connections are broken, there is no way to sent any numbers from the sender to the receiver. The entered numbers will be saved on SD card and could be sent to the receiver at once via syncing once the connection is restored.

5. Connection is lost between a receiver and the RK.
   All entries are be stored until they are successfully transmitted to the RK. When a connection is restored after some time of being broken, these entries will be retransmitted. Note that requests for additions will be prioritised over other requests (syncs, deletions).

6. The SD Card on the sender and/or receiver fails:
   In case of SD card experiences physical or any other type of failure, the system should still be able to send and receive numbers. However, will not be able to synchronize anymore (it is impossible to mirror two SD cards if one has failed). There may also be a risk of losing entries history depending on which SD card has failed. If only the receiver's SD card fails the history could still be restored based on the sender.

[3]After some experimental tests, it was found out that the normal NumberPackets took about $30ms$ to transmit.

Whereas if the SD card of the sender fails there is no guarantee that 100% of packets are delivered. The communication should be very reliable so the chance of entries missing is very minimal, but due to the possibility of LoRa or WiFi failing completely, it cannot be guaranteed. In the case where both SD cards fail the history, should still be on the RK since all entries that are received are forwarded to the RK. But as mentioned before transition of entries cannot be completely guaranteed, so therefore there is still a risk of some packets being dropped due to the insufficient connection.

7. The sender, receiver or RK server is no longer operational:
   Replacement will be necessary. Though, if the receiver or RK is no longer operational, numbers will still be stored on the SD card(s).
8. The Wi-Fi or LoRa connection is no longer operational:
   The state of the connection is displayed to the operator, which means that, given that the operator notices that the connection has been severed, they can turn on the backup link. One problem can arise when both links are broken. In this case, the antenna or LilyGo will need to be replaced.
9. Wi-Fi connection becomes unavailable while packets are being sent or while waiting for acknowledgements:
   Packets are sent at almost the speed of light, so and the protocol handles packet loss and connection loss so no issues should occur between sender and receiver.
   Between the receiver and RK a major issue could occur. If the RK is not connected to Wi-Fi but the receiver is, and the operator then turns off their hotspot, due to a fault in the HTTP library the LilyGo will reboot once. After rebooting, the LilyGO will try connecting once again.
   Some numbers may be lost during this operation, namely all numbers queued to be sent to the RK. These numbers will however be stored on the SD card before being lost. Therefore they can be resent to the RK via syncing.

Also, the system design must be able to handle unexpected user behavior, while remaining responsive for new inputs.

The risks associated with human interaction and the designed solutions are:

1. A human operator enabled synchronization while outside the range of a hotspot:
   The Wi-Fi will be unable to connect and syncing will not happen. This will be communicated with the operator.
2. A human operator sends a number while devices are syncing:
   The number will be sent after all synchronization-related packets are either acknowledged or dropped. The sending number in between syncing packets is not possible by the design, since both number and syncing packets are using the same FIFO queue.
3. A human operator typed in a wrong number:
   When the number is sent by the operator it will be displayed, but when the operator realizes that he made a mistake he can send a delete command to delete the number from the screen and the RK.

# Chapter 4

# Technical Specifications

## 4.1 Software Specifications

| | |
|---|---|
| Framework | PlatformIO |
| Programming language | C++17 |
| External libraries | • sandeepmistry/LoRa @ 0.8.0<br>• olikraus/U8g2 @ 2.35.15<br>• Unity @ 2.5.2 |
| Arduino libraries | • Wire @ 2.0.0<br>• SPI @ 2.0.0<br>• SD @ 2.0.0<br>• FS @ 2.0.0<br>• HTTPClient @ 2.0.0<br>• WiFi @ 2.0.0 |

### 4.1.1 Why C++17?

As ESP32 micro-controllers have limited memory and performance capabilities, opting for a fast language with full control over memory was critical for the project. Besides that, the language should have a (relatively) large embedded community, and some publicly available libraries for embedded development to improve the efficiency of the development process. The languages that fit these criteria were Rust, C, and C++. Eventually, it was decided to use C++, as some of its members were already familiar with this language, it has more features than C and was more familiar to the team than the at the time relatively new Rust. As for the version, C++17 was chosen as it is the most widely supported version for ESP32 development as of the time of the project.

### 4.1.2 Why PlatformIO?

After choosing C++ as the language of choice, the two clear options for developments tools were the default Arduino IDE and PlatformIO. PlatformIO is a framework for developing embedded software, and was chosen due to its more advanced feature set compared to the default Arduino IDE, and its support for the specific LilyGO devices used in the project. This more advanced feature set includes support for unit testing, static code analysis, multiple environments and automatic compilation.

### 4.1.3 Why these specific external libraries?

In this project, a minimalistic approach towards external libraries was chosen due to the strict memory restrictions of the provided hardware. Included below is an explanation for each library that the project uses:

- "sandeepmistry/LoRa" for managing the LoRa module: turn it on/off, and send/receive data.
- "olikraus/U8g2" for displaying information on an OLED display.
- "Unity" for implementing and running tests.

### 4.1.4 Why Arduino libraries?

- "Wire" for communicating with I2C devices like the OLED display.
- "SPI" for communicating with SPI devices like the SD card.
- "SD" for writing to/from SD cards.
- "FS" for operating files within the SD card.
- "HTTPClient" for making HTTP requests to a web server.
- "WiFi" for instantiating servers, clients and sending packets via WiFi.

## 4.2 Hardware Specifications

Both the sender and the receiver consist of a TTGO LoRa32 V2.1 microcontroller. This board contains an ESP32, which is a 32-bit dual-core microcontroller with on-board Wi-Fi and Bluetooth. Additionally, the board contains a LoRa module and an SD card slot, both using the SPI bus. The LoRa module features a 16-bit CRC baked into the hardware of the module, practically eliminating the need for additional error detection. The board also features a 0.96" OLED screen which is controlled using the I2C bus. This board was chosen beforehand by the product owner, and a couple units were graciously provided to us for testing during the development of the software.

During requirement elicitation, the client expressed their desire to be able to add and/or swap out a keypad and the display connected via I2C. I2C was chosen for this to limit the number of pins needed. Additionally, I2C eliminates the need for constant polling of pins to check for key inputs. Due to the modular design of the input and output classes the client will be able to easily implement his own version and swap it out with one of the existing ones.

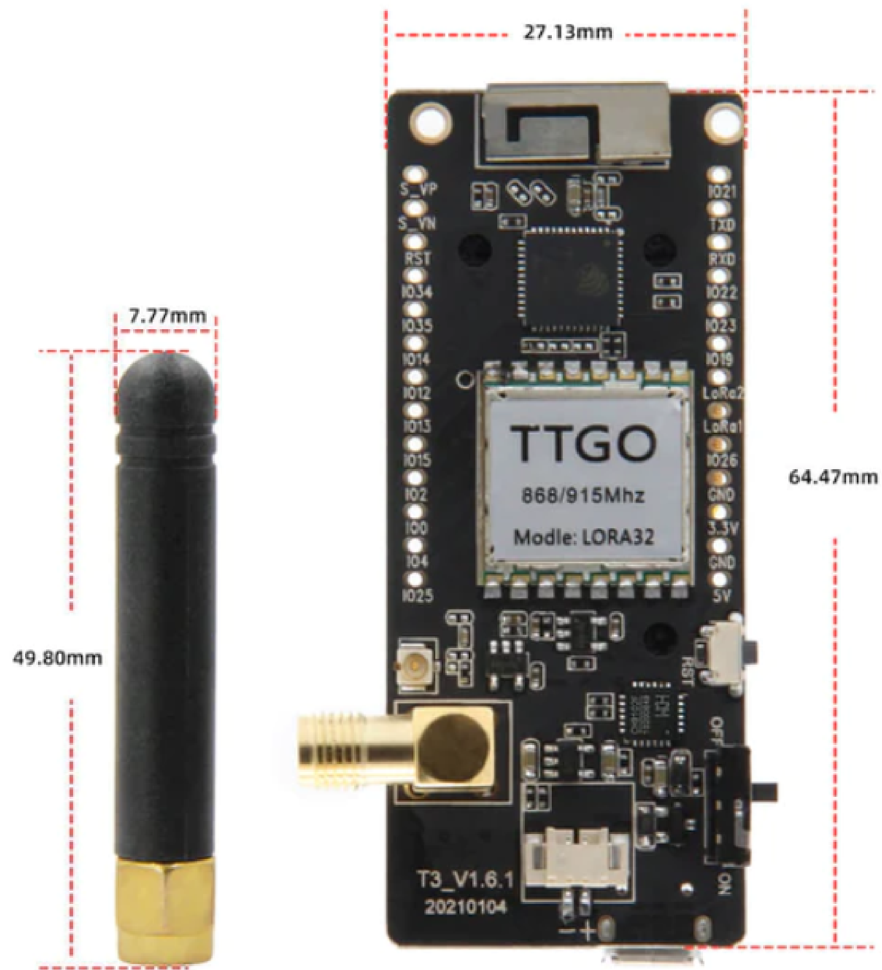


Figure 4.1: ESP32 microcontroller size

# Chapter 5

# Testing

## 5.1 Testing Strategy

Since the product will be deployed in a real-life setting, where it will need to work for the duration of the Batavierenrace, it is required that the product is reliable and has as few defects as possible. Therefore, it is very important to test the system thoroughly: from testing the individual methods up to verifying the overall system.

The types of tests that will be used for this are unit tests, integration tests, system tests, and performance tests.

Unit tests are used to verify single components, such as checking the encoding of a packet or showing error messages on a display screen. Integration tests verify the integration between separate parts of the system, such as handling connection loss between components or verifying that a number from an input device is correctly packed into a packet.

System tests are used to test the entire system, which includes testing if a number correctly makes it from the input device to the HTTP request sent to the RK, and onto the SD card of the receiver. These will be tested manually, as it is quite hard to verify all components of a system test automatically, and because it provides an opportunity to use the system as an operator, which will provide valuable information on any blatant user experience flaws of the system.

In addition to the aforementioned tests, performance tests are also included, to test the performance and physical limits of the system. Performance is in part dependent on the physical distance between sender and receiver, which is hard to simulate accurately with only a computer, making it necessary to do these tests by hand.

The goal of these tests is to create a robust and reliable system, which means that ideally all components of the system should be fully tested. To ensure every class is unit tested properly, the initial plan was to have minimum code coverage set to 95% of the entire code base. This metric was not set to 100% because some of the code can not reasonably be expected to be automatically tested, such as ESP32-specific methods, boilerplate, or dependency-reliant code. When developing it was discovered that, while running on ESP32 with code coverage metrics enabled, the ESP32 ran out of memory and failed. Therefore no code coverage could be added.

Some parts of the system do not need to be tested as much, particularly the physical link and its related hardware. The Wi-Fi and LoRa hardware and antennas are expected to work before starting to code.

Tests are created after coding, because using Test Driven Development is not a viable option for micro controller development. It is impossible to know if a way of implementing something is possible on ESP, and to run tests automatically several dummy files need to be made to extend the project. Therefore the strategy is to first create an implementation that might work, then create a test which accounts for each edge case, then iron out the bugs.

The unit tests will be run as part of the GitHub Continuous Integration routine, GitHub Actions. As soon as a pull request has been made to merge into the main branch, unit tests will be run to ensure that components have not broken with any changes. Integration tests are performed once the components of the system that are required by a particular integration test are finished, and whenever a component's interface changes. System tests are performed at a later stage in the project when an MVP has been accomplished, and performance tests will be done with the final version of the message services.

## 5.2 Unit Testing

Unit tests aim to test specific elements of the system. This is done to verify the individual functionality of components to eliminate errors on a component basis. A brief overview is given below for each unit test:

1. **connection**
   (a) **manager**
      i. **test_receiver_connection_manager -** Verifies whether the receiver connection manager can be instantiated with default or arbitrary connection objects, checks all the connections for packets, whether it replies on the last used link and whether it correctly connects and disconnects the connections.
      ii. **test_sender_connection_manager -** Verifies the functionality of the Sender Connection Manager: checks whether an instantiation of the Sender Connection Manager works with default arguments or specific connections, whether it sends via the correct connections and whether it correctly connects and disconnects the connections.
   (b) **test_lora_connection -** Tests the raw functionality of the LoRa link by letting the sender send some packets of varying sizes over the link and letting the receiver echo back that data.
   (c) **test_wifi_connection -** Tests the raw functionality of the Wi-Fi link by letting the sender send some packets of varying sizes over the link and letting the receiver echo back that data. It also verifies the maximum transfer unit of the Wi-Fi connection.
2. **message_service**
   (a) **test_receiver_message_service -** Tests the reliable transmission protocol on the receiver side, which involves testing acknowledgment, synchronization, and the handling of duplicates under different circumstances such as sequence numbers' overflow.
   (b) **test_sender_message_service -** tests the reliable transmission protocol on the sender side. Verifies packet re-transmissions and dropping. Additionally, tests synchronization and connection statuses.
3. **test_controller -** Natively simulates the sender & receiver microcontrollers that are connected by a "virtual" connection. It tests transmission and synchronization mechanisms.
4. **test_millis -** Tests if time measurements are correct on ESP32 devices.
5. **test_packets -** Tests packet encoding and decoding, verifies if packet sizes are correct and if the encoder/decoders handle invalid input appropriately.
6. **test_sd_card -** Checks whether entries are written and read correctly to and from the SD card. Additionally, verifies the creation of backups, and checks if the file structure is fixed upon initialization of the sd_card_handler.

## 5.3 Integration Testing

To test the connection between two elements of a system, integration tests have been created. A brief description is given below for each integration test:

1. **connection**
   (a) **lora -** tests sending a lot of packets via LoRa and checking whether they all arrive.
   (b) **wifi -** tests sending a lot of packets via WiFi and checking whether they all arrive.
2. **test_http_to_rk -** manually connects a microcontroller with a flask server (to emulate the RK) via Wi-Fi, and then tests sending a lot of packets and one big sync packet using HTTP.

## 5.4 System Testing

The system test is located inside /test/manual/system_test and consists of two runnable files: test_receiver.cpp and test_sender.cpp, corresponding to the receiver's and sender's source codes respectively. They simulate the full working of the system, including number transmission, synchronization, etc.

The tester can also utilize the FaultyLoRaConnection and the FaultyWiFiConnection classes (instead of LoRaConnection and WiFiConnection) to simulate packet loss for testing the retransmission mechanism.

Besides that, on the sender's side, the tester can choose an "input type", which can be one of the following:

1. "continuous": the sender will quickly send random numbers with a given timeout in between.
2. "dummy": the sender will send a given number of numbers at the start.
3. "serial: the sender will send numbers from the serial monitor. The protocol for writing numbers is located in the dummy_serial_input.h file.

## 5.5 Performance Testing

### 5.5.1 Transmission Reliability

Some tests were done to test reliable transmissions for the separate links, namely WiFi and LoRa. For each link, some distances were chosen, shown below in the results table. It is expected that not every packet will be received at longer distances on the first try, thus testing will provide clearer answers on how well the links function at range.
The parameters set for the tests were that the tests were all done with clear weather conditions, with the antenna facing upwards. Each distance was tested 3 times with 200 packets each time.

For realism, LoRa was tested with light obstacles, as it is nearly impossible to get a clear line of sight at higher distances. For testing sake, these will be compared against lower ranges with also light obstacles. Another measure taken was that the speed of inputs was adjusted to 2 inputs per second, as a layman will probably not input faster than that.

Lora was also tested more extensively at a range of 100 meters, as that is the range it will be used at. Therefore additional testing was done by adding a test for a clear line of sight and a test for heavily obstructed buildings.

Similarly, to ensure realism, WiFi was tested with a clear line of sight, as that is how it will be used during operation.

Sending 600 packets total per distance (200*3) is a very small sample size, thus outcomes could be influenced by random chance, however, these numbers were chosen because even with these restrictions range testing took upward of 2.5 hours.

RESULTS:

'Retransmission %' is the percentage of total traffic occupied by retransmissions. For example: if there were 200 retransmissions total, 'Retransmission %' is 25%, as 200 / (200 + 600) = 0.25
'Dropped %' is the percentage of the 600 packets sent that were dropped. If 300 were dropped, 'Dropped %' is 50%.
'Corruption %' is the percentage of corrupted packets that successfully passed CRC checking and were received by the receiver as valid packets. If 300 packets were received incorrectly, 'Corruption %' is 50%.

LoRa:

| Range (meters) | Retransmission % | Dropped % | Corruption % |
|---|---|---|---|
| 1 | 0% | 0% | 0% |
| 100 | 0.2% | 0% | 0% |
| 200 | 0.3% | 0% | 0% |
| 500 | 9.1% | 0% | 0.2% |
| 1000 | 52.2% | 27.5% | 3.8% |

| 100 meters | Retransmission % | Dropped % | Corruption % |
|---|---|---|---|
| Clear line of sight | 0% | 0% | 0% |
| Light obstructions (forest) | 0.2% | 0% | 0% |
| Heavy obstruction (building) | 7.6% | 0% | 0% |

From these tests, we can conclude that LoRa works extremely reliably for lower ranges, regardless of obstructions.
The last reliable range was 500 meters, as no packets were dropped and only one corrupted.
One thing of note was that the position of the antenna made a huge difference. When facing the wrong direction or when moving, the antenna did not pick up on some packets, even at 100 meters.

WiFi:

| Range (meters) | Retransmission % | Dropped % |
|---|---|---|
| 1 | 46.4% | 0% |
| 50 | 50.5% | 0% |
| 75 | 50.7% | 0% |
| 100 | 52.7% | 0.5% |
| 125 | 59.2% | 11.2% |

150 meters was also tested for 200 packets, but the test went so badly (186 dropped) that the distance was aborted.
From the above tests, we can conclude that Wi-Fi works more reliably than expected, especially because a phone hotspot was used as an access point. Wi-Fi does have much more retransmissions than LoRa, due to a higher amount of interference for Wi-Fi's wavelength.

The table above does not include packet corruption, because this was never constituted in any of the tests ever done using Wi-Fi.

Something to note was that Wi-Fi simply does not work with obstructions. Wi-Fi waves have a hard time moving around moderately sized structures, like a column of a building.

Wi-Fi will mainly be used for syncing, and that was not tested at range. The reason for this is that the operator should not be syncing at range. They could, but the sync entries are split over several packets and if any one of these packets were to be dropped syncing would fail. The system will simply display this to the operator.

During typical use, one entry would be sent every 5 seconds. Since this would take too much time to test for a large number of transmissions the decision was made to have a number inputted (automatically) every 0.5 seconds. In a scenario with a weak link, this does cause a lot more packets to be dropped than when it would only have one input per 5 seconds, and is, therefore, the worst-case scenario. This is the case since the packets have a time-to-live of 10 seconds from the moment they are added to the queue, to prevent unlimited congestion in the queue. Due to the weak link a lot of packets have to be retransmitted which congests the queue, causing the the time-to-live of many packets in the queue to already pass almost entirely. This leads to a lot of packets only getting around 2 transmissions before being dropped. If the 5-second interval had been a packet would have had (5 retransmissions per second * 5 seconds = ) 25 transmissions before causing any congestion. So in a typical use case, the system will perform a lot better in a lossy environment than the tests show, but testing with a typical input rate would have taken an unreasonable amount of time and by testing the worst-case scenario it can be guaranteed that the system will have the same or better performance during typical use.

### 5.5.2 Robustness against interference

A test was done with the receiver not configured to the sender. The receiver did log any activity and did not send any acknowledgments.

A test was also done with 2 senders and one receiver, the receiver being configured to only one of the senders. The signals from the non-configured sender did not interfere in any way with the working of the system, though sometimes an extra retransmission was needed.

A test was also be done over one meter with a jammer to forcefully test the robustness of the retransmission system. The system performed extremely well, though it could drop some packets if inputs were transmitted at too fast a rate and jamming was prolonged for a while. This was discussed in the previous section.

### 5.5.3 SD card longevity

SD cards have a limited amount of write-and-erase cycles. SD cards typically have a lifetime of 10,000 write-and-erase cycles, however, this does not guarantee that the cards that the client will be using satisfy this. To be completely certain that the SD cards would not fail during operation, a performance test was made to write a large number of number entries over and over again. The test has two modes: one where it writes the entries to a file, then clears the file, and then writes new entries to the same fill, and one where entries are written to a file and then moved to the "old_files" directory after which new entries are written to a new file. For both tests, every entry is checked to see whether it was written correctly. Both tests were run multiple times with 300 entries per cycle for 100 cycles without any issues, showing that the SD card should survive for at least a couple hundred races, which is more than enough to satisfy the client.

### 5.5.4 Power usage

Since the sender will be battery-powered in the final system, the power usage should be as low as possible. To achieve this the system only has the active connection turned on (LoRa and Wi-Fi are responsible for drawing the most power, the power usage of code execution very little in comparison). The amount that the wireless connection is used also influences the power draw, therefore the power consumption was done in a multitude of scenarios: Using the system at max speed using Wi-Fi, using the system at max speed with LoRa, and using the system at a realistic number input speed (one input per 5 sec). All measurements were made over a period of more than an hour per measurement using 5v USB power.

| usage type | power usage (mAh) |
|---|---|
| max throughput Wi-Fi | 400 mAh |
| max throughput LoRa | 358 mAh |
| typical throughput LoRa | 245 mAh |

# Chapter 6

# Evaluation

## 6.1 Planning

The planning can be found in Appendix A.

This was made at the end of week 2, just after making the draft of the design report. It was originally used for task division in week 3, thus the names attached to most of the tasks, but eventually a shift was made to cards on Trello for task division.

The planning was still used throughout, mostly as a broader list of tasks to be completed. This was used up until week 6, because as predicted the MVP was complete in week 6. After the MVP was shown to the client weeks 7-10 were rewritten, because priorities changed after receiving feedback from the client, and the original planning for week 7-10 assumed there were no major problems to be fixed in the MVP.

It is impossible in programming to predict how long any given task will take in advance. The implementations of systems did not take much time, but the integration of each members work took way longer than expected. A lot of time was also spent on refactoring and discussing code, something the team did not anticipate.

The biggest planning headache in the project ended up being range testing. The problem was that performance testing was planned to be done as early as possible, to find errors and have time to fix them. However, to do range testing, the system needed to fully work for at least one of the connection types (preferably LoRa). The message service ended up taking a long time to complete, so no range testing could be done for LoRa until the middle of week 9. Completely testing the system only ended up happening in week 10.

## 6.2 Responsibilities

Task division was first based on the planning, then on Trello cards. Below is an overview of what each member contributed:

- **Matteo:**
  - Input and Outputs
  - HTTP Communication to RK
  - WiFi Connection
  - Report Management
- **Douwe:**
  - Packets
  - Header Design
  - Reliable Transmission Protocols
  - General Management
  - C++ Tech Support
- **Ben:**

  - SD Card Backups
  - Number / State Outputs

- **Danylo:**
  - Message Services
  - Reliable Transmission Protocols

- **Alexey:**
  - Synchronisation
  - Controllers
  - Professional Refactorer

## 6.3 End result

At the end of the 10-week development cycle, most requirements were completed, and the client expressed satisfaction with the result.

However, not all is perfect, as there are some functional & non-functional requirements that we could not (or not fully) complete:

The most important one was packet corruption. At the start of the project, it was discovered that LoRa implemented 16-bit CRC and Wi-Fi implemented 32-bit CRC. This gave the team enough confidence that no further CRC needed to be added. Later during development, however, it was found that packets could get through LoRa's 16-bit CRC. However, since this packet corruption only occurred in scenarios when transmitting through multiple buildings, or at distances $>= 500m$, it was decided to focus on the more pressing issues at the time (such as continuous reboots). This small chance of corruption could be brought even further down, see section 6.4

In addition, some aspects of the project were not tested properly. For example, due to the inconsistency of weather conditions in the Netherlands, the only weather condition tested was 'sunny'. Continuous use over 12 hours was also not tested, though the sender and receiver have been running for periods of up to 2 hours. Unless there is an unaccounted memory leak somewhere, the system should be able to handle 12 hours given an appropriate power supply.

There were also some things the client did not deem deeply necessary, thus we decided not to implement them, due to limited development time. The requirements where this was the case were:

- The system could have saved a timestamp next to each entered number.
- The system demo could have included a keypad.
- The system could have included a casing around the keypad.
- The system could have displayed whether numbers were updated after synchronization

Though what is stated above all sounds negative, most requirements are completed and extensively tested!

Most non-functional requirements can be said to be completed with a reasonable amount of certainty. Using the performance tests we can say that LoRa will work with at least 99.9% accuracy, and we can state that the system works well at ranges up to at least 500 meters. The system is also documented well in code with code, and files are separated by functionality, so we can say a Technical Computer Scientist should be able to understand or extend the code relatively easily.

Even some of the requirements that did not have heavy priority were implemented! The system displays the current number being inputted and the number previously sent and does turn off power-consuming electronics to save power.

## 6.4 Future Improvements

### 6.4.1 Future Improvements

1. **Encryption**:
   Currently, no encryption is used in the communication between the sender and receiver. This can, as described in the risk analysis, lead to an attacker inputting malicious numbers, given that they know the packet format, and either have a matching LoRa unit or manage to break into the Wi-Fi Access Point. A potential improvement for this could be to add an encryption layer to the communication, for example in the connection managers, in order to ensure that an attacker has the smallest chance to send valid packets. Since the sender and receiver are already statically linked, a symmetrical encryption scheme might fit best here, as both the sender and receiver can keep the same key, just like they share the same sender/receiver ID (section 3.1.1).

2. **Additional checksum**:
   Unfortunately, in the performance tests, damaged packets sometimes managed to pass the hardware CRC check in the LilyGO, which could in practice lead to random values being interpreted by the receiver as a valid entry.

   While the likelihood in theory and in practice is quite low, and was only observed when trying to receive through multiple buildings or at distances greater than 500m, to make the chances of these packets getting through even lower one could add an additional checksum to the existing packet format.

3. **Reliable transmission protocol**:
   Currently, the project uses a stop-and-wait-based protocol with a First In First Out queue, meaning that any new numbers are put on the end of the queue and have to wait for other numbers to transmit first. Since each packet has a maximum time to live and a minimum transmission count, and this time to live starts right after it is queued, the packets that were added to the queue earlier are prioritised over those that were added later. The stop-and-wait protocol was chosen in particular to keep custom Medium Access Control logic to a minimum and improve throughput, since LoRa does not offer any MAC on its own, and the link (unlike most networks) is almost instant, meaning that there is almost no time in-between a sender/receiver sending a packet and that packet arriving on the other side.

   The FIFO queue was chosen to maintain simplicity in packet format and decrease overall congestion when sending multiple packets (since each packet has to have a separate acknowledgement).

   Ideas were also passed around to implement FILO queues, for prioritising newer packets, implementing a Round-Robin-based protocol ([2] 6.2.2), to increase fairness in case of longer queues, or to implement a sliding window protocol ([2] 2.5.2).

   However, the stop-and-wait protocol turned out to be more than adequate for a realistic workload that would typically be exerted on the network (say, an operator typing in less than one number per second), with some additional headroom[1].

   If, however, there would be any future demands for improved throughput with longer queues, cumulative ACKs, and a protocol more akin to TCP's Additive Increase / Multiplicative Decrease ([2] 6.3.1) can be chosen, to allow for sending multiple numbers without acknowledgement to reduce the total transmission time and, hence, throughput. Such functionality would, however, require the receiver to be able to acknowledge multiple packets with a single transmission, which would require a refactor for the receiver message service. It is important to restate, however, that in a real-life scenario, the aforementioned problem of delayed arrival of packets is very unlikely to occur due to the low workload, as an operator is typically not able to physically enter more than a couple of numbers per second.

4. **Channel activity detection (Medium Access Control)**: As discussed in item 3, the project ended up containing a stop-and-wait protocol, partially to help with Medium Access Control, as the sender is not allowed to send a packet before the receiver answers, or some time has passed. In the end, some (configurable) randomness to this timeout was added, to fix repeated collisions in the LoRa link by multiple sender/receiver pairs, if they were to be in range[2].

[1] The practical throughput for LoRa turned out to be around 3 to 4 numbers per second (regardless of distance), with Wi-Fi needing a bit more retransmissions and arriving at a steady 2

[2] One sender would transmit a packet at the exact same time as the other, or the acknowledgements to those packets would collide, after which both senders would wait the exact same time before retransmitting the same packet which would collide in the

A possibly more elegant way that would have fixed this would be Collision Avoidance, similar to CSMA/CA, used in Wi-Fi. However, while support for Channel Activity Detection is present for LoRa, the library used for LoRa did not allow us to easily add it to the project. Since we realised and fixed this problem quite late in the project, as it was only discovered when running tests for multiple pairs in parallel, there was unfortunately not enough time to look more thoroughly into this issue.

5. **Distinction between Backup and NumberOutput** :
   In the current state of the project, the distinction between NumberOutput and Backup is not ideal. A NumberOutput can write numbers, but a Backup can write *and* read numbers. We would have liked to improve this distinction, or decide on another solution, if there had been any time left.

6. **Timestamps**:
   Since the product owner did not put a lot of focus on including timestamps in the product, going as far as to even discourage the use of timestamps[3], there was not a great deal of focus placed on including timestamps in the final product. This means that, while there is support for it in the NumberEntry, and local timestamps are saved in the sender (the no. of milliseconds since the sender booted up), timestamps are not transmitted over the network, meaning that the timestamp for number entries at the receiver is different from the sender, and that there are *no* timestamps present in synchronisation packets, meaning that the receiver has no way of knowing when the backed up entries were typed in by the operator.

   Timestamp support for this project future can be achieved by either adding a hardware timer or keeping the local timestamps, adjusting the NumberEntry packet to feature a timestamp, letting the receiver fill in this timestamp in the NumberEntries it makes of the packets, and adding these timestamps to the HTTP requests for the RK.

7. **Testing (coverage)**: While the project is tested in the most crucial parts, particularly in terms of reliable transmission and the SD card logic, and even has some Continuous Integration tests, there could have been more thoroughly tested. This was partially due to that testing on the ESP32 is noticeably slower than on desktop, as the program has to slowly upload via a serial connection, and partially because the team did not place enough focus on ensuring all parts of the code were fully tested. This resulted in some cases where tests were passing, but a component would not behave exactly as expected in some situations.

   Attempts were made at including testing coverage metrics, however, the LilyGOs did not have enough storage for storing the additional code required to generate coverage reports.

same way.

[3]It was reasoned that the project did not need timestamps, as the current solution with walkie-talkies also does not

# Appendices

## Appendix A - Planning

### Week 1

- Requirement elicitation
- Project proposal

### Week 2

- Meeting with Pieter-Tjerk - design report layout
- Draft project design
- Develop a testing strategy
- Design diagrams
- Working LoRa link ("Hello world")

### Week 3

- HOLIDAYS

### Week 3 (Continued)

- Dividing requirements into cards
- Assigning people to cards
- Working Wi-Fi connection between two LilyGOs via an external network - Alexey (fixed by Matteo with HTTP protocol)
- Simple range testing (200m works fine even with UDP-like protocol)
- Start making reliable LoRa protocol (based on how much bandwidth/packet loss gathered from simple testing) - Danylo (done - sequence diagram?)
- Develop packet format - Douwe
- Make test environment to be able to run ESP code on laptop - Douwe
- Have working dummy input - Matteo
- Have working dummy output - Matteo

## Week 4

- Encode/decode packet - Alexey
- Store collected data on SD card (requires SD card) - Ben
- HTTP communication with RK
- Make up protocol, verify with Sarah - Matteo / Douwe
- Implement protocol - Matteo / Douwe
- Reliable LoRa protocol - Danylo / Douwe
  - (re)Send messages & receive ACKs to messages
  - Resend failed messages
  - CRC (handled by hardware in LoRa and WiFi)

## Week 5

- Sync SD cards with WiFi - Ben
- Implement wifi communication link - Douwe
- Switch between LoRa/Wi-Fi with button/switch

## Week 6

- PLAN INTEGRATION DAY!
- Test and debug MVP
- Take a good look at the documentation of the code!
- MVP done!

## Week 7-10

- Focus on:
  - Improving message service - Performance/range testing (+ charts for report!)
  - Real-world testing of syncing logic
  - Report
  - Integration tests
  - Making a nice end product (I2C stuff (LED display, keypad?), feedback to the operator (state output), etc.)

# Appendix B - Operator Manual

## Introduction

BATA_WAVE is a wireless keypad system for reliably transmitting team numbers to a screen of a relay point. The system consists of two parts: a sender and a receiver. The sender is the device with the keypad, while the receiver does not feature a keypad. This sender is meant to be used by the operator, standing in front of the relay point, for entering incoming runner numbers. The receiver is meant to be placed at the relay point, and will forward the received numbers to the screen so runners can see if their teammate is approaching the relay point.
This manual focuses on the operator's perspective. It explains how to set up the device, how to enter numbers, how to switch connection types, how to synchronise numbers, and what to do when certain things go wrong.

# Operation

## Setting up the system

In order to set up the system, first ensure that the receiver is connected to power (indicated by a glowing power button) and has connected successfully to the access point of the RK.
After confirming that the receiver is up and running, turn on the sender and wait until it is fully booted up. When the receiver has booted, it periodically checks the connection between the sender and receiver, which should reflect on the display with an indication that the connection is working.

## Sending numbers

Before trying to send any numbers, please make sure that the sender and receiver are operational.
BATA_WAVE allows for sending the numbers 1 to 999. In order to send a number in this range, simply enter the incoming runner number into the keypad and press the 'add' key. The number will be sent to the receiver and it will be displayed on the large screen.
As an operator, you can also mark a number as deleted, for instance if a wrong number was entered. To do this, enter the number into the keypad and press the 'delete' key.

## Switching connections

During operation, it might happen that the LoRa connection becomes unusable. When the sender shows a broken connection state when connected via LoRa, please walk a bit closer to the relay point (within roughly 50m) and switch to Wi-Fi mode by using the "LoRa / Wi-Fi switch". The device will repeatedly try to connect with the Wi-Fi network until it succeeds. If it succeeds in connecting to the Access Point and can send and receive data correctly, the device will show a good connection status. At this point, normal operation can be resumed. If the Wi-Fi connection appears to be broken, numbers can still be typed into the keypad and will be saved, however they might not arrive at the receiver and might require syncing. If the Wi-Fi connection issue does not resolve itself within a few seconds, please contact support.

## Synchronising numbers

It may happen that the connection drops out, or due to some other reason, not all numbers are sent successfully. To ensure that the backups on the sender and receiver side do not deviate, BATA_WAVE allows for synchronising all entries saved to the sender with the receiver.
If you want to synchronise numbers, please move into the range of the Wi-Fi network (>= 50m) and turn the device to Wi-Fi mode. After this, press the sync button. This will send all the stored entries in the sender device to the receiver.
Note that while synchronising, you are not allowed to enter any additional numbers. Please wait until the synchronisation process is finished.
The synchronisation process might take some time, depending on the amount of entries. In a typical scenario with <= 300 entries it should be done almost instantaneously, but with larger sets of entries it might take up to a couple of seconds. When syncing is finished, the sender device will indicate whether it synced successfully or whether the sync was not successful.
If the device shows that it failed to sync, try again after approximately 10 seconds. If the problem persists, contact support.

# Bibliography


[1] T. N. Kruse, R. E. Carter, J. K. Rosedahl, and M. J. Joyner, *Speed trends in male distance running*, https://doi.org/10.1371/journal.pone.0112978, 2014.

[2] L. Peterson and B. Davie, *Computer networks: A systems approach*, https://github.com/SystemsApproach/book, https://book.systemsapproach.org/, 2012.

[3] SemTech, *Sx1276/77/78/79 - 137 mhz to 1020 mhz low power long range transceiver datasheet*, https://cdn-shop.adafruit.com/product-files/3179/sx1276_77_78_79.pdf, 2016.

[4] T. Ritter, *The great crc mystery*, http://www.ciphersbyritter.com/ARTS/CRCMYST.HTM.

[5] W. W. Peterson and D. T. Brown, *Cyclic codes for error detection*, https://ieeexplore.ieee.org/abstract/document/4066263/, 1960.

[6] N. J. and A. Löfgren, *Designing for extensibility: An action research study of maximising extensibility by means of design principles*, https://gupea.ub.gu.se/bitstream/2077/20561/1/gupea_2077_20561_1.pdf, 2009.