Implementing a simulated multi-stream LoRa gateway in GNU Radio

M. van Dijke
m.v.dijke@student.tue.nl

***Abstract*—Long-range low power wide area networks (LPWANs) are an important driving force for fueling the ever-increasing expansion of Internet of Things (IoT) applications. A particular technology that is dominating the LPWAN space is called LoRa which uses chirp spread spectrum (CSS) modulation. Although LoRa is a proprietary standard by Semtech, several reverse engineering attempts have been made opening up more research into the (inner) workings of LoRa. The aim of this paper is to build upon previous reverse engineering research into LoRa by implementing a LoRa gateway in GNU Radio. This LoRa gateway is capable of receiving multiple messages from different transmitters simultaneously, a so-called multi-stream LoRa gateway.**

***Index Terms*—GNU Radio, LoRa, LoRa gateway**

I. Introduction

LOW power wide area networks (LP-WANs) are gaining more and more (industry) interest and acceptance as large scale IoT appliances, especially in wide-area appliances such as for example smart cities [1]. The idea behind these IoT appliances is mostly in the form of covering an area with sensors to better monitor these areas. The most emerging and industry-accepted wide area network is LoRa, which is a proprietary standard made by Semtech [2]. LoRa uses a chirp spread spectrum (CSS) modulation and through this modulation technique is able to excel at low power, low data-rate & large coverage area. As compared to its technology competitors such as NB-IOT or Sigfox. Fig 1 shows a typical large area setup of Lora, where multiple sensors, so-called end nodes are connected to one or more LoRa gateways. An example of an end node is the temperature sensor or the luminosity sensor as depicted in the figure. These LoRa gateways are in turn connected to network servers handling the received sensor data on an IP level. Finally, the application server uses this data for visualization for the end-user.

Although details of the physical layer (PHY) are proprietary, and therefore not open source. The LoRaWAN (Long Range Wide Area Network) which builds on the PHY is not proprietary and is open source. This LoRaWAN is more geared towards the IP layer of the (data) transmission. Which is depicted as the network server and application server in Fig. 1. For instance a popular open-source LoRaWAN is ChirpStack [3]. Partly caused by this somewhat open-source nature of LoRa, several reverse engineering attempts have been made over the past couple of years to open-source the LoRa PHY and the inner workings of the LoRa PHY. In order to achieve an entire open-source implementation of all components of LoRa and LoRaWAN.

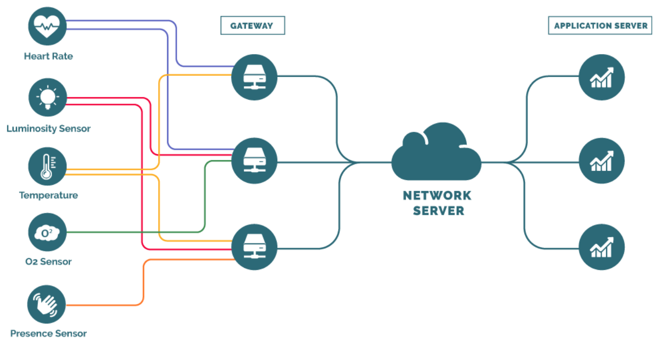

Fig. 1. General wide-area LoRa setup, figure taken from [4].

Most of these reverse engineering attempts have been made using the GNU Radio framework [5], which is an open-source software toolkit for signal processing. These reverse engineering attempts and scientific experimentation and verification of LoRa have led to a great understanding of how the LoRa PHY works. Although there is quite some literature on the topic, most literature has been focused on LoRa performance and capabilities.

Functionality that is still missing in open-source reverse engineering of LoRa is LoRa's multi-stream gateway capability. This multi-stream capability entails that a LoRa gateway can receive data from multiple end nodes simultaneously, as is depicted in Fig. 1 by having multiple sensors be connected to one gateway. This makes it possible for a LoRa gateway to service more the one LoRa end node in the same area of coverage further increasing their practical use in real-life large area network(s). This paper focuses on adapting existing reverse engineering attempts in order to achieve a multi-stream LoRa gateway. This is done in an effort to more closely reassemble the real-life behavior (and performance) of LoRa gateways. Along with laying the groundwork for an open-source version of a multi-stream gateway further accomplishing the goal to have an entire open-source implementation of all components of LoRa and LoRaWAN.

II. Related work

As stated before numerous reverse engineering attempts have been made to open source the LoRa PHY, these reverse engineering attempts have had varying scopes of the LoRa PHY. Some attempts focussed on the conceptual (system level) of LoRa such as [6], or real-life verification/testing of the coverage area of LoRa [7] while others focus on the verification of the properties of the LoRa standard [8]. One of the earliest works in this field of reverse engineering has been done by M. Knight and B. Seeber [9] with accompanying code [10]. In this paper, the authors dive into the different stages used in the LoRa standard and introduce an open-source GNU Radio implementation that can decode specific LoRa messages. The main shortcoming of this work is that not all spreading factors of the LoRa standard are supported. The authors of [11] also made a reverse engineering attempt on the LoRa PHY, with their GNU Radio based code [12]. While this implementation can decode message overall spreading factors that are present in the LoRa standard. The implementation is only able to decode LoRa messages, encoding of the messages to be LoRa compatible is not supported. Making this implementation only viable in combination with commercially available LoRa transmitters. Work by author O. Afisiadis [13] provides an extensive thesis on the working of LoRa and the reverse engineering of the LoRa PHY. The accompanying GNU Radio code [14], from author J. Tapparel provides a fully functional low-level open-source LoRa PHY. In this implementation, messages are encoded and decoded, and this implementation is almost compatible with the entire range of LoRa parameters, except for some specific usage of LoRa parameters. Next, to reverse engineering projects using the GNU Radio framework, attempts have been made using the Pothos framework [15]. While this framework is similar to GNU Radio, the framework is newer and not yet field-proven as of this moment. Making it thus undesirable to develop for. Yet, Github user myriadrf [16] has made a reverse engineering attempt using Pothos but unfortunately this attempt only works for a specific SDR receiver.

III. Problem statement

The work of author O. Afisiadis [13] and J. Tapparel [14] is the most complete and low-level reverse-engineered implementation that is currently available. However, while this implementation has been verified to work on real hardware, the work is focused on only the LoRa PHY. This means that no additional layers on top of the LoRa PHY have been implemented. Therefore, the implementation exhibits two shortcomings which this paper hopes to address:

1) The first shortcoming of this implementation is that the implementation does not allow for receiving and sending multiple messages with different spreading factors simultaneously and can thus not be used as a multi-stream gateway.
2) The second shortcoming of the implementation is that the accompanying code is developed for use on USRP's (Universal Software Radio Peripheral) and hence no "simulation mode" is available where the software is executed on the computer without the use of USRP's.

While the second problem should be solvable using the inherent simulation capabilities of the GNU Radio framework it should be noted that this is not tested or verified and therefore is listed as a subsequent shortcoming of the work.

The major problem this paper tries to address is the first problem of implementing a multi-stream LoRa gateway. Since this is the most complete work currently available it has been chosen to build further on this work to implement the multi-stream gateway.

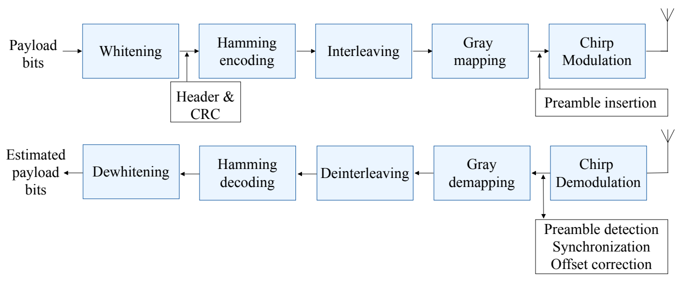

Fig. 2. Transceiver chain of LoRa, figure taken from [13].

IV. Concepts behind LoRa

In this section, the driving concepts behind the LoRa PHY will be explained, in order to understand how the LoRa PHY works. As well as how a multi-stream gateway can work. Internally the LoRa PHY uses a multi-stage approach for creating and decoding LoRa packets. Each stage performs its own specific computation on its input and has a different design reason it is used in LoRa. All these stages and underlying dependence between the stages are depicted in Fig. 2. Together all stages form the LoRa PHY which is able to encode and decode LoRa messages. Each stage is represented in the code as a separate GNU Radio block, and shall hence be referred to as a *block*.

A. Modulation & Demodulation

LoRa uses a CSS modulation scheme for its modulation of payload bits into transmissible frequency waves. This spread spectrum based modulation is also called a chirp based modulation [8] because of the analogy that a linear increase in the frequency leads to a chirping sound to the human ear. The used CSS modulation in LoRa uses bandwidth B and $N = 2^{SF}$ as the number of chips, where SF is the spreading factor with values $SF \in \{7, \dots, 12\}$. Each LoRa symbol maps to a specific frequency, this frequency determines the symbol and the corresponding value of that LoRa symbol. For example, the LoRa symbol for 0 is given by f_0 which maps to f_{min}, the minimum frequency LoRa uses. A graphical representation of a modulated LoRa symbol is given in Fig. 3, in this figure the spectrogram of a LoRa symbol that maps to f_1 is shown. In this figure t_s is the symbol duration which is equal to $\frac{N}{B}$ and $t_{fold} = \frac{N-s}{B}$ which described the frequency folding point in time and where f_{max} is the maximum frequency LoRa uses. The modulation can be written in the time domain as [13], [17]

$$x_s(t) = \begin{cases} e^{j2\pi\left(\frac{B}{2T_s}t^2+\left(s\cdot\frac{B}{N}-\frac{B}{2}\right)t\right)}, & 0 \leq t < t_{\text{fold}} \\ e^{j2\pi\left(\frac{B}{2T_s}t^2+\left(s\cdot\frac{B}{N}-\frac{3B}{2}\right)t\right)}, & t_{\text{fold}} \leq t < t_s \end{cases} \quad (1)$$

Analogously, the discrete time variant of a LoRa symbols can be written as (2) if the assumption is made that $f_s = B$, and where $\mathscr{S} = \{0, \dots, N-1\}$ [13]. After being transmitted

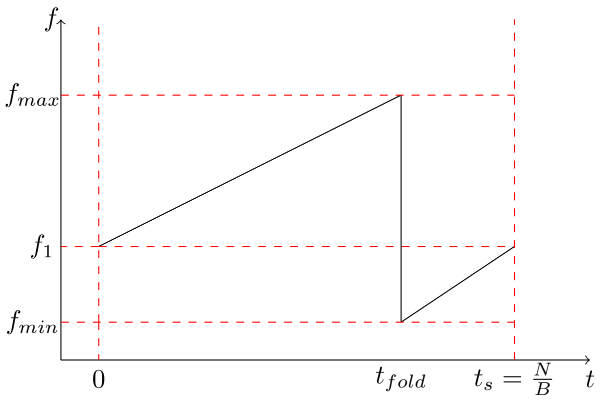

Fig. 3. Simplified spectrogram of modulating a LoRa symbol.

over a time invariant channel h the received LoRa symbols are given by (3) where $z[n]$ represents the additive white gaussian noise (AWGN) noise with $\sigma^2 = \frac{N_0}{2N}$ and N_0 is the single-sided noise power spectral density [13].

$$x_s[n] = e^{j2\pi\left(\frac{n^2}{2N}+\left(\frac{s}{N}-\frac{1}{2}\right)n\right)} \; n \in \mathscr{S} \tag{2}$$

$$y[n] = hx_s[n] + z[n] \; n \in \mathscr{S} \tag{3}$$

Once the samples have been received, they need to be demodulated in order to obtain the estimated transmitted symbol $\hat{s}$. This is done by multiplying the received signal by a complex conjugated of a reference signal x_{ref} as shown in (4). This is often the LoRa symbol for $s = 0$ (f_0) i.e. an upchirp.

$$x_{\text{ref}}[n] = e^{j2\pi\left(\frac{n^2}{2N}-\frac{n}{2}\right)} \; n \in \mathscr{S} \tag{4}$$

Using the non normalized discrete Fourier transform (DFT) on the dechirped received signal ones obtains $\mathbf{Y} = \text{DFT}(y \odot x^*_{\text{ref}})$, where $\odot$ is the Hadamard product and $\mathbf{y} = \begin{bmatrix} y[0] & \dots & y[N-1] \end{bmatrix}$ and $x_{\text{ref}} = \begin{bmatrix} x_{\text{ref}}[0] & \dots & x_{\text{ref}}[N-1] \end{bmatrix}$ [13]. Now, in order to obtain $\hat{s}$ the frequency bin index of the DFT computation with the maximum magnitude is our best bet for $\hat{s}$. After all this is the bin where the most power has been transmitted forming the most likely candidate for our transmitted symbol as (5) states.

$$\hat{s} = \arg\max_{k \in \mathscr{S}} (|Y_k|) \tag{5}$$

B. Blocks

1) Whitening: The first stage of the LoRa PHY is the whitening block, the main purpose of this block is to remove any DC offset from the data stream. The reason is to ensure that inside a period of one data stream no inter period of time has a higher energy leading to offset in the transceivers.
The computation done on the incoming data stream is XOR'ing the data stream with a predefined sequence of values called *whitening sequence*. By XOR'ing the data stream the output has on average a random power distribution inside one data stream period and thus exhibits no DC offset in the data stream.

TABLE I
POSSIBLE CODING RATES IN LORA PHY

CR = 4/8	d0	d1	d2	d3	p0	p1	p2	p3
CR = 4/7	d0	d1	d2	d3	p0	p1	p2	
CR = 4/6	d0	d1	d2	d3	p0	p1		
CR = 4/5	d0	d1	d2	d3	p0			

2) Hamming Coding: Because of the nature of (wireless) transmission signals and especially the designed usage of LoRa, long-range and low power. A large emphasis is made on making the entire LoRa PHY more bit error-resilient and in general being able to recover from bit error(s). LoRa uses Hamming coding [18] in order to detect and recover from bit errors. More specifically LoRa uses Hamming(k,n) codes where $n \in \{5, \dots 8\}$ and k is always equal to 4.
In this way the extent to which bit errors may be detected and a single bit error may be detected can be varied by changing the n parameter of the Hamming code. The Hamming code Hamming(k,n) is represented in LoRa notation as the coding rate (CR) and is $CR = \frac{k}{n}$. Table I shows the possible coding rates of the LoRa PHY where the parity bits are denoted as p0,p1,p2,p3, and the data bits of the incoming data stream is denoted by d0,d1,d2,d3.

3) Interleaving: After the Hamming encoding stage, the data streams moves to the interleaving block, where the data is interleaved. The underlying design reason behind this is to introduce more bit error resilience into the overall system, especially bit error resilience in the form of burst bit errors. Burst bit errors are bit errors that occur in many consecutive bits independently of each other and are common in wireless transmission. Eq. (6) shows the mathematical computation done in order to compute the interleaved matrix I from the original matrix D. The i index in (6) is given by $i \in \{0, \cdots, CRL\}$ where CRL is the Coding Rate Length (CRL) which is the n parameter of the CR, and the j index is given by $j \in \{0, \cdots, SF\}$.

$$I_{i,j} = D_{((i-j-1) \bmod SF),i)} \tag{6}$$

Fig. 4 shows a visualization of the mathematical computation done. On the left side of the image, the original matrix D is displayed which gets interleaved into matrix I which is displayed on the right side of the image, and where CL stands for Codeword Length. A codeword is the data to be transmitted is depicted as one colored row of data bits (d_n) and parity bits (p_n), which gets interleaved in the process. From Fig. 4 it is clear that by using interleaving the data and parity bits are spread out over multiple "codewords". This leads to a reduced effect of the burst bit error since it is now more likely to be able to be correct by the Hamming stage.

4) Gray mapping: After interleaving the data stream enters the Gray mapping block, which is used in the LoRa standard to further strengthen the bit error resilience of the overall system. This is done by modulating the data stream in such a manner that two subsequent symbols differ in one bit. This means that if the estimated decoded message denoted by $\hat{s}$ equals ± 1

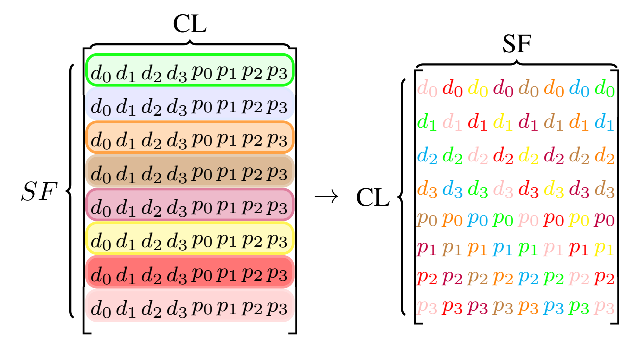

Fig. 4. Graphical representation of interleaving process.

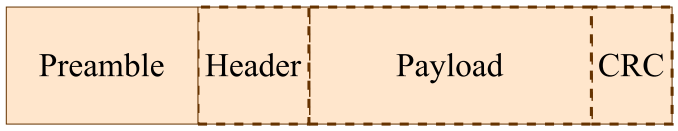

Fig. 5. LoRa packet structure, figure taken from [13]

there is a one-bit error. This one-bit error can be detected and repaired by the earlier discussed Hamming stage if $CR = \frac{4}{7}$ or $CR = \frac{4}{8}$ are used.

C. Data Packets

In order to explain the synchronization mechanism between transmitter and receiver, a deep dive into the data packet structure of LoRa must be made. This packet structure consists of the following parts:

1) Preamble, which is used for packet synchronization between transmitter and receiver.
2) Header, this is an optional part of the packet and houses information about the following:
   a) the packet length of the to be transmitted data packet.
   b) the coding rate used.
   c) the presence of CRC.
   d) the checksum of the header itself.
3) Payload, which is the actual payload bits that are being transmitted to the receiver.
4) Cyclic redundancy check (CRC), which is an optional extra measure to be able to detect bit errors.

For synchronization between transmitter and receiver the most important part of the LoRa data packet is the preamble. The preamble itself consists of three different parts namely:

1) Upchirps, which are used as a detection sequence for the receiver side. If a predefined number of upchirps denoted by N_{pr} are detected at the receiver side, the receiver side will synchronize these received samples in time to be the preamble and thus basic synchronization of the LoRa data packet is achieved.
2) Network identifiers, which consists of two symbols that are used for frame synchronization and to distinguish between devices from different networks.
3) Downchrips, which are used for further synchronization between transmitter and receiver. Most notably for estimation and correction of time and frequency offsets between the receiver and transmitter.

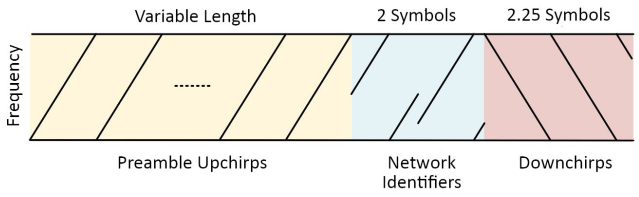

Fig. 6. Preamble structure, figure taken from [13]

The structure of the preamble is also depicted in Fig. 6.

V. Implementation

A. Gateway

The goal of the multi-stream gateway is as stated before to allow for receiving and sending multiple messages with different spreading factors simultaneously. This can fundamentally happen because of the two facts:

1) The estimate of the received signal is only dependent on the reference signal given by (4) which is only dependent on an upchrip at a specific spreading factor. A more graphical explanation of this fact is also possible by using Fig. 3, this figure shows that the energy of the modulated LoRa symbol is spread out over $\frac{2^{SF}}{B}$ samples. Since B is considered to be constant, and thus the only variable in the figure is SF.
2) The basic synchronizations step as detailed in Section IV-C is only dependent on a reference upchirp which is again only dependent on a specific spreading factor.

This in turns means that on a conceptual level, messages with different spreading factors are practically orthogonal from each other. It should therefore be possible to implement a multi-stream gateway by making sure all signals originating from all blocks are being processed in parallel.

The original work where this paper builds upon uses several blocks in order to transmit and receive messages. Although this allows for quick prototyping and experimentation of the code, for the use case of this implementation this is not convenient. After all the multi-stream gateway works on a higher (system) abstraction layer of LoRa. As implementation details of the underlying blocks as given in IV-B are therefore not of at most importance. Since this implementation is made on a higher abstraction level and uses and adapts these blocks where necessary. This abstraction is done in order to allow for easier experimentation. This is done by wrapping all the individual blocks needed for transmitting into a new block called *Tx*. Along with the individual blocks needed for receiving into a new block called *Rx*. These *Tx* and *Rx* classes are depicted in Fig. 7 and Fig. 8.

Each GNU Radio block is spun up in a separate thread [19]. And as *Tx* and *Rx* just wrap the individual blocks together, different *Tx* and *Rx* blocks are therefore executed in parallel. Fulfilling a large prerequisite for a multi-stream gateway. Fig. 9 propose a structure for the multi-stream gateway, which consists of one incoming (received) data stream that is split

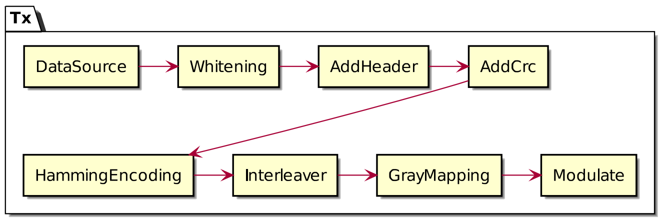

Fig. 7. Structure of Tx chain.

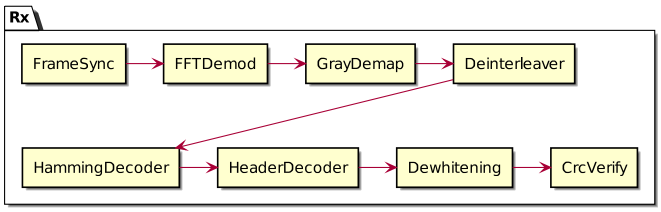

Fig. 8. Structure of Rx chain.

onto multiple *Rx*'s each having a different spreading factor. This incoming data stream is an addiction (in the time domain) of multiple transmitting *Tx* so that the output samples of each *Tx* have been added together.

B. Simulation

The above-described implementation implements a multi-stream gateway using an abstraction of the underlying blocks. This however does not describe or implement ways to simulate the multi-stream gateway. The main hurdles to overcome for simulating the gateway are:

1) Computational buffers and computations themselves are unrestricted in simulation in the sense that simulating the code will use all computational resources available since there is no limiting hardware factor that is slowing the system. This problem can be mitigated by using throttle functionality built into GNU Radio.
2) The number of threads needed to run the implementation is quite large since 9 threads per *Rx* block are used and 6 *Rx* blocks are used in parallel totaling to a maximum of 54 threads for the gateway only. While CPUs with such a large number of threads/cores are available it is far from common in consumer-grade hardware and may cause problems.
3) In order to measure performance characteristics such as throughput of the system the gateway needs to exit once it has completed the requested amount of packets. This requires some form of control synchronization in order to reach some consensus on when the gateway has decoded all messages that have been transmitted. This functionality is currently not provided by the original works and requires adding control flow capabilities into all blocks used.

VI. Validation & Results

Since we now have all the necessary details and components in order to simulate a multi-stream gateway. It's time to test

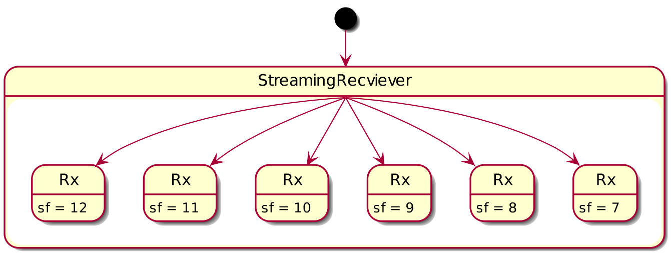

Fig. 9. General implementation structure of a multi-stream gateway.

and benchmark the system to see if the chosen implementation is correct and its performance characteristics. But in order to compare the results of the multi-stream gateway, a reference benchmark must be made on a single transmitter and receiver setup. All benchmarks have been executed on an Intel i7-4700MQ with 4 cores with hyperthreading turned on totaling the number of threads to 8 and version 3.8.1 of GNU Radio. Due to the GNU Radio block scheduler, blocks are executed in a non-deterministic fashion that prioritizes throughput of the blocks over deterministic behavior [19]. This means that no guarantees can be made on the size of input of each execution of the block and the order in which blocks are being executed. Due to this non-deterministic behavior, a small variance is expected in all benchmark results.

A. Single Tx & Rx

The used parameters for the single transmitter and receiver benchmark are given by Table II, where n_r describes the number of times the same benchmark has been executed in order to obtain the statistical behavior of the system, (B) denotes that the parameter is of a boolean type, BW represents the bandwidth and ML represents the message length, *Impl* denotes the use of implicit header mode, CRC denotes if the CRC check option is enabled, CR is the coding rate, and N_{fr} is the number of packets that must be sent where each packet contains the number of bytes defined by ML. During testing and verification, a large dependence on of the t_{wait} parameter on the behavior of the system has been noted. This t_{wait} parameter is the minimum time the system waits to create new packets. For example, a $t_{wait} = 200$ entails that there is a minimum time of 200 milliseconds between the first packet inside *Tx* being created and send to the second stage of *Tx* (the whitening). By varying this parameter its possible to distinguish two cases:

1) Low values of $_{wait} \approx 100 - 300ms$ in this region of t_{wait} the computational time needed in order to process all computations dominates the behavior of the system.
2) High values of $t_{wait} \gtrapprox 900ms$ in this region of t_{wait} the wait time between packets dominates the behavior of the system.

Since this parameter is of influence on the behavior of the system, all benchmark cases have been executed with both options. Roughly speaking the first $\pm 3 - 3.5$ seconds of execution time are spend initializing and loading all the code into RAM without an actual signal computation. This behavior has been inspected by using the scalene python code profiler

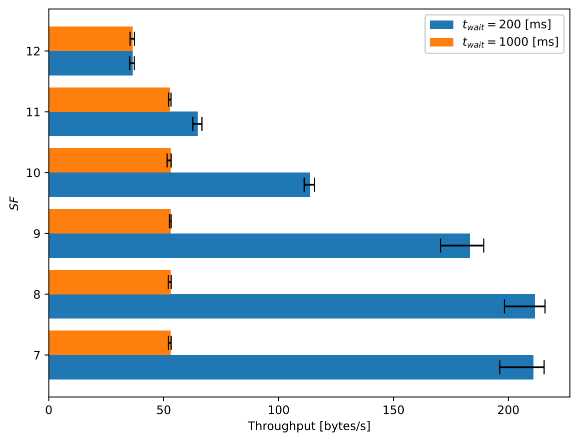

Fig. 10. Mean throughput for varying spreading factor, parameters Table II.Error bars indicate the minimum and maximum observed values.

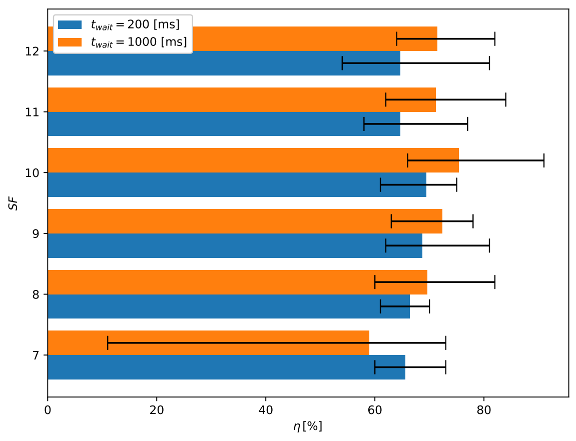

Fig. 11. Number of correctly decoded messages for varying spreading factor, parameters Table III. Error bars indicate minimum and maximum observed values.

[20] and this behavior is inherent to all benchmarked implementations and is considered constant among all benchmarks and therefore negligible in comparing the results from the benchmark.

Figure 10 shows the mean throughput of a single transmitter and receiver. From this figure, we can deduce that for low spreading factors $SF_{low} \in \{7, 8\}$ and low t_{wait} value there is almost no difference in throughput between spreading factors. However, the difference in throughput for larger spreading factors is more noticeable with a decreasing throughput as the SF is being increased. This relationship stems from the fact that although more bits can be sent using a higher SF, the symbol time as given by $t_s = \frac{N}{B}$ also increases leading to lower throughput. For high t_{wait} values the relation is different since more time is spent waiting on packets to be sent leading to lower throughput, which in turn means that there is almost no difference in throughput for $SF \in \{7, 8, \ldots, 11\}$ and only for $SF = 12$ there is a significant difference in a lower throughput since the symbol time needed to transmit is much higher.

Next to the throughput of the system, the percentage of correctly decoded messages is also of great interest as a benchmark result. In order to compute this, a static message that is to be transmitted is fed into the transmitter and on the receiver side, the number of times this message has been received is counted. The number is then divided by the total amount of messages it should have received leading to the percentage of correct messages. Since there is no simulated noise in the system the percentage of decoded messages should be 100 %. Yet, as Fig. 11 shows, the percentage of correctly decoded messages is not 100%. Overall looking at the figure we can say that on average high t_{wait} values perform better. The phenomenon of not being able to decode all messages is most likely due to a bug in the code of the project, despite the fact that efforts have been made to fix the bug, the bug in the code has not been found. The hypothesis is that since the implementation uses an asynchronous message type for control and data message timing errors inside the system occur resulting in unexpected behavior of the system. This hypothesis is especially likely to be the case in combination with the non-deterministic block scheduler in GNU Radio leading to the large variation of the observed values. It must be noted that the maximum percentage of decoded messages is very high, and thus underlying computational code is assumed to be functioning correctly. Nevertheless, more research and in-depth analyzes must be made to find the real cause of this problem.

TABLE II
PARAMETERS USED FOR BENCHMARKING

n_r	B[Hz]	ML[int]	Impl[B]	CRC[B]	CR[int]	N_{fr}[int]
25	250000	64	True	False	4	15

B. Multi-stream gateway

For the multi-stream gateway, the number of used *Tx* is denoted by N_{Tx} and is increased in order to benchmark the multi-stream gateway. Each additionally added *Tx* has a higher spreading factor starting from $SF = 7$. This means that with $N_{Tx} = 1$ there is one *Tx* sending at $SF = 7$ but with $N_{Tx} = 6$ there are 6 *Tx* that in total send messages at $SF \in \{7, \ldots, 12\}$. Fig. 12 shows the mean throughput for the multi-stream gateway from this figure shows that the throughput is in general much higher than for the single transmitter and receiver case. This is as expected since more data is being transmitted in roughly the same time period. Furthermore, this figure shows that for low t_{wait} values the throughput increased up to $N_{Tx} = 4$, since $SF = 11$ and

TABLE III
PARAMETERS USED FOR BENCHMARKING

n_r	B[Hz]	ML[int]	Impl[B]	CRC[B]	CR[int]	N_{fr}[int]
15	250000	64	True	False	4	100

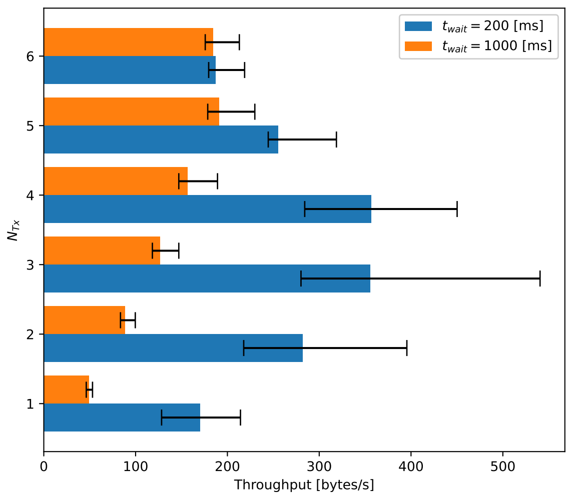

Fig. 12. Mean throughput for varying number of N_{Tx}, parameters Table II. Error bars indicate minimum and maximum observed values.

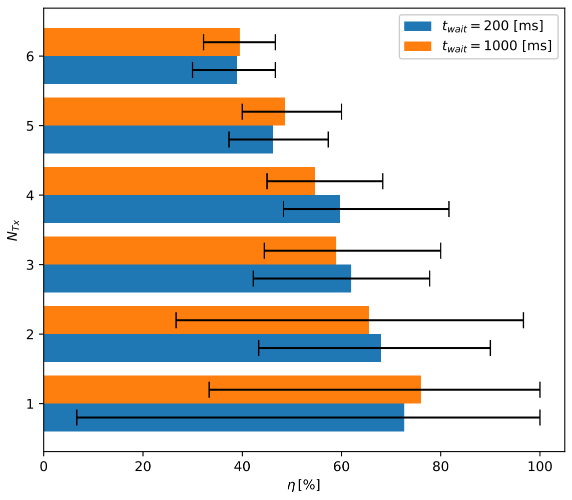

Fig. 14. Number of correctly decoded messages for varying number of N_{Tx}, parameters Table II. Error bars indicate minimum and maximum observed values.

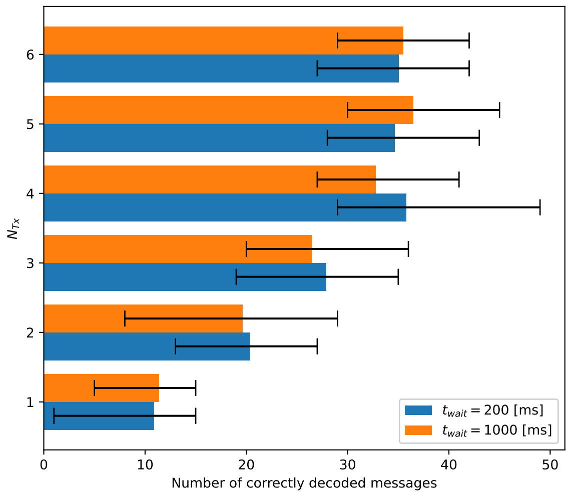

Fig. 13. Number of correctly decoded messages for varying number of N_{Tx}, parameters Table II. Error bars indicate minimum and maximum observed values.

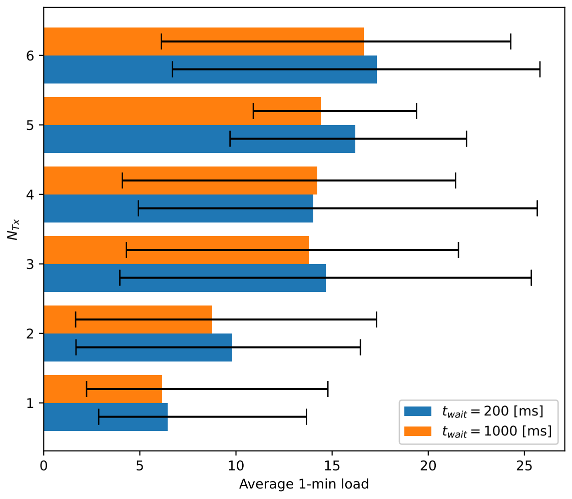

Fig. 15. Linux load number for varying number of N_{Tx}, parameters Table II. Error bars indicate minimum and maximum observed values.

$SF = 12$ need a (much) longer symbol time leading to a decrease in throughput. For high t_{wait} values the throughput is linear increasing simply because more data is transmitted in roughly the same time period. Moreover, this figure also shows that fundamentally the proposed implementation of the multi-stream gateway works since the throughput is higher than in the single transmitter and receiver case. This may also be seen by looking at Fig. 13 where the number of correctly decoded messages surpasses N_{fr} meaning that more messages from different SF have thus been decoded and thus the proposed implementation works.

The percentage of correctly decoded messages is shown in Fig. 14. From this figure, it shows that there is little distinguishing between a high and a low t_{wait} value. This is most likely due to the number of messages that need to be decoded in parallel. Resulting in the high value of mean to also be computation limited in the multi-stream gateway. It also shows that the percentage is decreasing for an increasing N_{Tx}. This is unexpected since one would expect this percentage to remain constant. It is hypothesized that this unintended behavior stems from CPU limits being reached. A likely candidate for this limit and unwanted behavior is the sheer amount of threads needed to be executed. This computational phenomenon can be seen by looking at Fig. 15 where the 1-min average Linux load is depicted. The average load number is used in Linux to tell how many CPU threads are needed, a load of 1 entails that only one CPU core is needed at 100 % capacity. Since the system running the benchmark has 8 threads it can sustain a load number of 8, any load higher than 8 would lead to a CPU bottleneck in the system leading to (not all) computations being done in parallel. Being able to run all computations in parallel is an important prerequisite for the multi-stream gateway and can in those conditions of high Linux load no longer be fulfilled, leading to a decrease in throughput. Currently, it is believed that this computational phenomenon in combination with the use of asynchronous messages is causing the system to behave unexpectedly and incorrectly. More research into the cause of this problem needs to be done in order to fully understand the problem and solve it.

VII. Future work

Although the system is not perfect, the goal of this paper has been achieved. Therefore, there are two large topics for the future to be investigated, namely:

1) Investigating the experienced problems with asynchronous control types in the system and see if these control types are at the root of the unexpected and unwanted behavior of the system.
2) Optimizing the performance of the system. This can be done in a number of ways:
   a) Reducing the number of threads needed, by implementing a single block that performs all computations instead of wrapping individual blocks into one block as this paper does.
   b) Code optimizations, the implementation has the potential to be faster with certain code optimizations. Especially the *frame_sync* block can be potentially a lot faster using certain for example loop unrolling or other memory optimizations techniques.
   c) FPGA in order to better understand the parallel nature of the multi-stream gateway implementation may be made using FPGA accelerator, whereby using the inherently parallel nature of the FPGA much higher performance may be extracted out of the benchmark system. GNU Radio has support for such an FPGA computation offloading and may offer the easiest solution in the experienced CPU limit.

VIII. Conclusion

This paper has proposed and implemented a simulated multi-stream LoRa gateway, which is a novel addition to existing work in the LoRa reverse engineering space & open-source space. Both parts of the scope of the paper 1) implementing a multi-stream gateway 2) simulating such a gateway, have been implemented and have been shown to work. However, the implemented work is far from operating perfectly and there are thus several possibilities for future work as detailed in the previous section. The goal of the paper has been achieved and forms an important first step into more open-source research into this unexplored functionality of LoRa and brings a far more practical and real-life usage to existing open-source LoRa projects.

References

[1] A. Zanella and M. Zorzi, "L ong -R ange C ommunications in U nlicensed B ands : T he R ising S tars in the I o T and S mart C ity S cenarios," *IEEE Wirel. Commun.*, vol. 23, no. 5, October, pp. 60–67, 2016.

[2] Semtech, "LoRa Modulation Basics AN1200.22," *App Note*, no. May, pp. 1–26, 2015. [Online]. Available: http://www.semtech.com/images/datasheet/an1200.22.pdf

[3] "ChirpStack open-source LoRaWAN® Network Server." [Online]. Available: https://www.chirpstack.io/

[4] N. Santos, M. Cunha, B. Faria, R. Vieira, and P. Carvalho, "Performance of a LoRa Network in a Hybrid Environment - Indoor/Outdoor," no. June 2020, 2019.

[5] "GNU Radio - The Free & Open Source Radio Ecosystem · GNU Radio." [Online]. Available: https://www.gnuradio.org/

[6] M. Chiani and A. Elzanaty, "On the LoRa Modulation for IoT: Waveform Properties and Spectral Analysis," *arXiv*, vol. 6, no. 5, pp. 8463–8470, 2019.

[7] K. Hill, K. K. Gagneja, and N. Singh, "LoRa PHY Range Tests and Software Decoding - Physical Layer Security," *2019 6th Int. Conf. Signal Process. Integr. Networks, SPIN 2019*, pp. 805–810, 2019.

[8] B. Reynders and S. Pollin, "Chirp spread spectrum as a modulation technique for long range communication," *2016 IEEE Symp. Commun. Veh. Technol. Benelux, SCVT 2016*, no. 2, pp. 0–4, 2016.

[9] M. Knight and B. Seeber, "Decoding LoRa: Realizing a Modern LPWAN with SDR," *Proc. GNU Radio Conf.*, vol. 1, no. 1, 2016. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/8

[10] "BastilleResearch/gr-lora: GNU Radio OOT module implementing the LoRa PHY, based on https://github.com/matt-knight/research/tree/master/2016_05_20_jailbreak." [Online]. Available: https://github.com/BastilleResearch/gr-lora

[11] P. Robyns, P. Quax, W. Lamotte, and W. Thenaers, "gr-lora: An efficient LoRa decoder for GNU Radio," *Zenodo*, 2017.

[12] "rpp0/gr-lora: GNU Radio blocks for receiving LoRa modulated radio messages using SDR." [Online]. Available: https://github.com/rpp0/gr-lora

[13] O. Afisiadis, "Physical Layer Aspects of LoRa and Full-Duplex Wireless Transceivers," 2020.

[14] "tapparelj/gr-lora_sdr: This is the fully-functional GNU Radio software-defined radio (SDR) implementation of a LoRa transceiver with all the necessary receiver components to operate correctly even at very low SNRs. This work has been conducted at the Tele." [Online]. Available: https://github.com/tapparelj/gr-lora_sdr

[15] "pothosware/PothosCore: The Pothos data-flow framework." [Online]. Available: https://github.com/pothosware/PothosCore

[16] "LoRa modem with LimeSDR – MyriadRF." [Online]. Available: https://myriadrf.org/news/lora-modem-limesdr/

[17] C. Bernier, F. Dehmas, and N. Deparis, "Low Complexity LoRa Frame Synchronization for Ultra-Low Power Software-Defined Radios," *IEEE Trans. Commun.*, vol. 68, no. 5, pp. 3140–3152, 2020.

[18] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, apr 1950.

[19] B. Bloessl, M. Müller, and M. Hollick, "Benchmarking and Profiling the GNU Radio Scheduler," *Proc. GNURadio Conf. 2019*, 2019.

[20] E. D. Berger, "SCALENE: Scripting-Language Aware Profiling for Python," *arXiv*, 2020.