The University of Hong Kong

Department of Electrical and Electronic Engineering

ELEC 4848

Senior Design Project

Final Report

Channel Access Control Protocol for LoRaWAN

HO Yu Hin

BEng (ElecE)

UID: 3035276696

Supervisor: Prof. Lawrence K. Yeung

Second Examiner: Prof. Ricky Y.K. Kwok

Abstract

LoRaWAN is a low-power wide-area network (LPWAN) based on LoRa technology that uses Aloha-based protocol as its channel access control protocol, which limits its network performance. Several channel access control protocols are implemented on the software simulator extended from LoRaSim and Arduino/Raspberry Pi – based hardware. The results show that the increase in the number of frequency channel and the use of non-sequential channels improve the network performance in term of packet delivery ratio (PDR). Moreover, the uses of separated channels for the transmission of acknowledgement packets and critical packets reduces the number of retransmissions and improved the PDR of critical packets. By integrating the design in this project to the existing LoRaWAN standard, the reliability of the communication system can be enhanced.

Acknowledgement

I would like to express my gratitude to my supervisor - Prof. Lawrence K. Yeung, who gives me a lot of guidance. In addition, I would like to thank my second examiner – Prof. Ricky Y.K. Kwok, who also provide me feedback in the midterm presentation.

Table of Contents

List of Figures

List of Tables

Abbreviations

Below terms and abbreviations are used in this report

- ABP

  Activation by personalization

- ACK or ack

  Acknowledgement

- BS

  Base station

- BW

  Bandwidth

- CR

  Coding rate

- CRC

  Cyclic redundancy check

- CSMA/CA

  Carrier sense multiple access with collision avoidance

- DSSS

  Direct Sequence Spread Spectrum

- FSK

  Frequency-shift keying

- HAL

  Hardware Abstraction Layer

- LBT

  Listen before talk

- LoRa

  Long Range. A modulation method developed by Semtech that provides long range communication

- LMiC

LoRaMAC in C

- LoRaWAN

  A wide area network based on LoRa

- LPWAN

  Low power wide area network

- MAC

  Medium access control

- OTAA

  Over-The-Air-Activation

- PDR

  Packet delivery ratio

- RF

  Radio frequency

- RFH

  Random frequency hopping

- SF

  Spreading factor

- SNR

  Signal to noise ratio

- SPI

  serial peripheral interface

- TTN

  The Things Network

1. Introduction

With the rapid development of Internet of Things (IoT) technologies in these years, the number of IoT applications has been continuously increasing, such as industrial plant monitoring and smart metering. More devices will be connected in the future. Wireless technologies are needed to connect billions of devices as they provide larger flexibility, higher efficiency and lower cost, comparing to wired communication technology. Also, the energy consumption of devices become an important consideration in the communication system design. In some cases, it is unrealistic to replace the battery of many devices. There is a need to minimize the energy consumption in communication. Traditional wireless communication technologies do not simultaneously provide both characteristics of low power consumption and long transmission range. In these years, many new modulation techniques have been invented to fill in this gap. Those techniques are used to build Low Power Wide Area Networks (LPWAN) to connect the devices.

LoRa (Long Range) is a state-of-the-art wireless digital modulation scheme with the advantages of long transmission range, low power consumption and high robustness. It operates in the Industrial, Scientific and Medical (ISM) radio bands, 867-869 MHz for Europe, 902-928 MHz for North America and 470-510MHz for China. Using the lowest data rate, it has a transmission range of 2 km. However, it is shown that using the lowest data rate yields unacceptable packet delivery ratio (PDR), and hence 1.2 km coverage should be assumed [1]. Moreover, LoRa provides flexibility in varying the physical parameters to fulfill different communication requirements, and this leads to the popularity of the LoRa technology.

Based on LoRa modulation, LoRaWAN (LoRa Wide Area Network) is one of the LPWAN protocol that is commonly used. However, LoRaWAN only uses Aloha-based protocol and does not provide any channel access control mechanism, which limits its performance in packet delivery [1]. Since the transmission medium is shared in wireless communication, we need an efficient protocol to avoid collision to enhance the PDR. The use of random frequency hopping (RFH) and carrier sense multiple access with collision avoidance (CSMA/CA) schemes are proven to yield better PDR in simulation [2]. Another research suggests that it is possible to apply channel planning scheme such as reserving a set of channels for retransmission [3]. Currently, there are few researches focusing on the evaluation of channel access control protocols on both the software simulator and hardware implementation, and they are not adaptive to the LoRaWAN standard. However, it is important to evaluate the performance on both platforms. Software simulator development allows evaluation of network performance before implementation of the network. It also provides analysis on the scalability of the system. Hardware implementation demonstrates the feasibility of implementing channel access control protocols and channel planning schemes in LoRa and how the protocol be compatible with LoRaWAN standard.

The aims of this project are to develop a software simulator to evaluate the improvement in PDR by applying different frequency hopping schemes and channel planning schemes on LoRa and implement them on Arduino platform to show that the result matches with the simulated result and demonstrate the feasibility of implementation. The project shows that various frequency hopping schemes and channel planning schemes can be implemented compatibly with LoRaWAN standard and yield improvement in PDR.

The rest of the report is organized as follows. Section 2 analyzes the research problem including different channel access control protocols and the method of implementing frequency hopping schemes on LoRa. Also, the mechanism and procedure for setting up the channels and modifying the hopping sequence are discussed. Section 3 discusses the software consideration, hardware consideration, and the design of experiment. In section 4, the design and implementation of LoRa simulator and hardware are presented. After that, section 5 presents the experimental results. Finally, section 6 concludes the result in this final year project and provides various recommendation for future study.

2. Analysis of Problem

2.1 Physical Layer

This part introduces several important parameters in physical layer, including modulation technique, bandwidth, spreading factor and coding rate. It first introduces the relationship between Shannon – Hartley Theorem and spread spectrum modulation. Then some important physical parameters of LoRa are discussed. Specifically, how those parameters affect the transmission range and power consumption are discussed.

2.1.1 Shannon – Hartley Theorem and Spread Spectrum

Shannon – Hartley theorem states the maximum transmission rate of information over a noisy communication channel, mathematically,

$$C = B * \log_2(1 + \frac{S}{N}) \quad \textbf{(1)}$$

, where

C = channel capacity (bit/s)

B = channel bandwidth (Hz)

S/N = signal to noise ratio

The implication of the Shannon – Hartley theorem is as follows. To achieve large channel capacity, the channel bandwidth and/or signal to noise ratio (SNR) need to be increased. In other words, increasing channel bandwidth can compensate the degradation of SNR in a noisy radio channel. Therefore, spread spectrum technique is invented to spread the signal across a large bandwidth to increase the channel capacity.

The most traditional spread spectrum technique is Direct Sequence Spread Spectrum (DSSS). In DSSS, at the transmitter, each modulating signal is multiplied by a spreading code/chip sequence. The chip sequence has much higher rate comparing with the modulating signal. By multiplication, the modulating signal is spread across a larger bandwidth. At the receiver, the received signal is multiplied by the chip sequence again. By multiplication, it converts the spread signal back to un-spread signal which is the original modulating signal. This is the basic principle of spread spectrum modulation and demodulation.

The main problem of DSSS is that in order to demodulate the signal at the receiver, same chip sequence as the transmitter needs to be known. It requires an extremely accurate reference clock source, and thus increases the cost of the transceiver. Another problem is that, the demodulation involves correlation of whole code sequence, which leads to long demodulation time. These two problems make DSSS unsuitable in power-constrained low-cost devices used in the communication system of IoT applications. Therefore, chirp spread spectrum is invented [4].

2.1.2 Chirp Spread Spectrum

Chirp spread spectrum (CSS) was originally used in military application, such as radar, in the 1940s, and now CSS is commonly used in low power wireless communication. Chirp is a signal with continuously varying frequency. Chirp with increasing frequency is called an up chirp. Chirp with decreasing frequency is called a down chirp. By modulating the data signal with the chirp, the spreading of spectrum is achieved. In the modulation process, the modulating signal is first chipped at a chip rate, and then modulate with the chirp. Since the modulation and demodulation process does not rely on a chip sequence which needs to be known to both the transmitter and the receiver, it overcomes the main disadvantage of DSSS.

2.1.3 LoRa Physical Parameters

LoRa belongs to the CSS modulation scheme. Below define some important physical parameters and discuss their relationship with transmission range and power consumption for the understanding the experimental settings.

2.1.3.1 Bandwidth

Bandwidth (BW) is the difference between the minimum and the maximum frequency of a signal. Larger BW gives higher transmission rate as shown in equation 2 and 3 in section 2.1.3.4. A LoRa transceiver can select a BW between 7.8 to 500 kHz, but typically at 125, 250 or 500 kHz used in LoRaWAN [5].

2.1.3.2 Spreading Factor

Spreading factor (SF) indicates the number of chips per symbol. SF can be selected from 6 to 12. It is denoted by {SF6, SF7, SF8, SF9, SF10, SF11, SF12} in this report.

Higher the SF means lower symbol rate as shown in equation 3 in section 2.1.3.4. From the figure 1, it can be observed that double time is required to transmit a chirp for SF9 comparing to SF8, which shows that the symbol rate is lower.

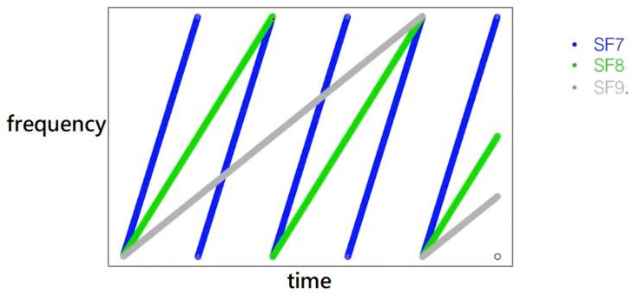


Figure 1 Spreading Factor in Time and Frequency Domain

Larger SF also means longer transmission range. This is because the spreading of signal makes the signal more resistant to noise.

2.1.3.3 Coding Rate

Coding rate (CR) is the ratio of useful information in a transmitted packet. LoRa provides cyclic error coding for data recovery by adding redundant bits in the packet. CR in LoRa can be selected from 4/5, 4/6, 4/7 and 4/8 [5]. For example, 4/8 CR means half of the bits are redundant, giving an overhead of 2. This reduces the transmission efficiency and consumes more energy.

2.1.3.4 Bit Rate, Symbol Rate and Chip Rate

Bit rate

Bit rate $(\mathrm{R_b})$ is the rate at which data is transferred.

Mathematically,

$$R_b = SF * \frac{1}{\left[\frac{2^{SF}}{BW}\right]} \tag{2}$$

, where

SF = spreading factor

BW = modulation bandwidth (Hz)

Symbol rate

The symbol rate $(\mathrm{R_s})$ is the rate at which the symbol is transmitted, which is proportional to the bandwidth of signal and related to the spreading factor. Mathematically,

$$R_s = \frac{BW}{2^{SF}} \tag{3}$$

Chip rate

The chip rate (R_c) is the number of chips transmitted per second, which is proportional to the symbol rate and related to the spreading factor. Mathematically,

$$R_c = R_s * 2^{SF} = BW \tag{4}$$

2.2 MAC Layer

The Medium Access Control (MAC) layer provides multiplexing and flow control for the communication medium. It controls the physical parameters used in transmission. A data unit in MAC layer is called a frame. MAC layer defines the mechanism to transmit and receive frames. It also provides functions for error checking and data recovery and thus gives protection against error.

For easy implementation of communication system based on LoRa, LoRaWAN standard is invented to provide and define functions in MAC layer and network layer to facilitate information exchange between devices. LoRaWAN is currently a popular standard that used by most of the developers and thus this project aims at developing a new transmission mechanism that is compatible with the LoRaWAN standard. To understand how the new design be compatible with LoRaWAN, the LoRaWAN architecture needs to be understand.

Section 2.2.1 discusses the architecture of LoRaWAN system. Section 2.2.2 introduces the classes of LoRaWAN end devices. Section 2.2.3 analyzes the frequency plan of LoRaWAN. Section 2.2.4 discusses the initialization of channels in the join process. Section 2.2.5 aims to illustrate the MAC command exchange mechanism which allows network server to send commands for modifying the network setting via downlink packet. Section 2.2.6 explains how the channels can be modified using MAC command exchange. Section 2.2.7 discusses the channel access control protocols.

2.2.1 Architecture of LoRaWAN System

The simplified architecture of LoRaWAN system is shown in figure 2

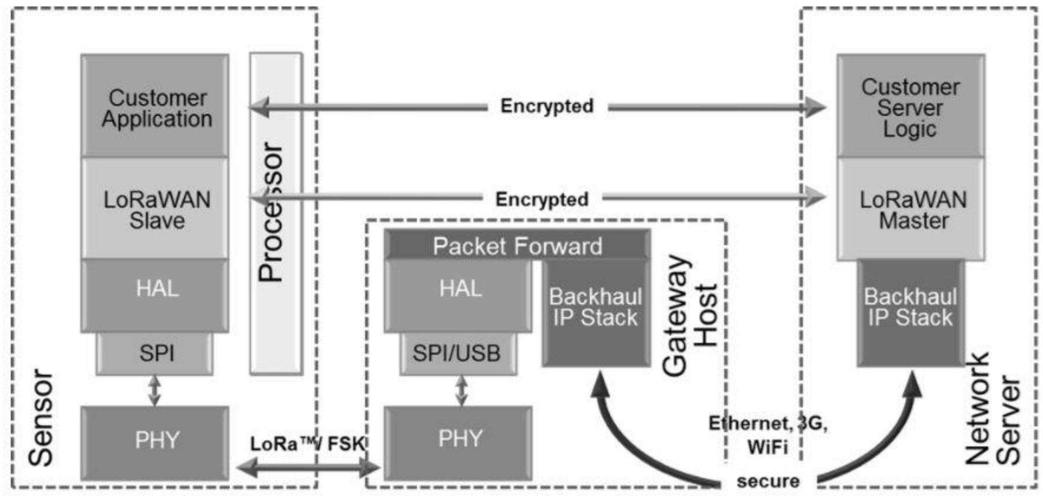


Figure 2 LoRaWAN Architecture

The architecture of LoRaWAN system is shown in figure 2. It basically consists of three components, namely sensor/end device, gateway/concentrator and network server.

End devices are devices that consist of a microcontroller and a LoRa transceiver. In the microcontroller, there is a microprocessor running customer application and controlling the LoRa transceiver via serial peripheral interface (SPI). In common IoT applications, the customer application collects data from the physical world using sensors and instructs the transceiver to transmit the data as a radio frequency (RF) packet using LoRa modulation or frequency-shift keying (FSK) modulation.

For interfacing the application and the transceiver, there is a hardware abstraction layer (HAL) that is implemented in software for developers to access the transceiver. HAL contains the information of pin mapping and register mapping. This separates the software implementation from the hardware implementation. Or in other words, the changes in hardware selected would not require modification in the application software. This encapsulates the complex hardware details.

The gateway receives the packet from end devices and passes upward to the MAC layer via SPI and HAL. It checks if the packet passes the cyclic redundancy check (CRC) which is an error detecting code. Under common implementation, the gateway only forwards the packet to the network server if the packet passes the CRC. Although it can be set to forward packets that not passing the CRC, it is not suggested as most of the network servers are not able to process it.

After the network server receives the packet, it passes the packet to the application as specified in the packet. The network server would handle the packets differently depending on the server logic.

The packets are encrypted using AES128 encryption algorithm to ensure security [5]. The keys used in encryption and decryption are decided when an end device joins the network. Since internet security is out of the scope of this project, section 2.2.4 only briefly discuss the keys defined in the join process.

There are some classifications of packets defined in LoRaWAN standard [5]. Packets that are sent from the end devices to the network server are called uplink packets. Packets that are sent from the network server to the end devices are called downlink packets. Packets that require an acknowledgement replies from the receiver are called confirmed packet. Packets that does not requires an acknowledgement replies from the receiver are called unconfirmed packet. Different types of packets are used in different experiments in this project.

2.2.2 Classes of LoRaWAN End Devices

Since there are different operation requirements of end devices, LoRaWAN provides three different classes of end devices [5].

Class A devices can only receive data via two receive windows after the transmission as illustrated in Figure 3. Hence, it is in sleep mode at most of the time, and thus it is the most energy efficient option. For this reason, it is commonly used in battery powered end devices.

The network server could determine which receive window is used for downlink transmission. Regarding the receive window, it is defined by an offset time, duration and data rate. Receive window 1 (RX1) has an offset of RxDelay1 from the transmit slot. Receive window 2 (RX2) has an offset of RxDelay2 from the transmit slot. The length of RX1 and RX2 must be larger than the time on air of the uplink packet.

The data rate of the first downlink transmission is calculated as a function of the uplink data rate and the receive window offset. The data rate of second window is fixed to be 0.3 kbps.

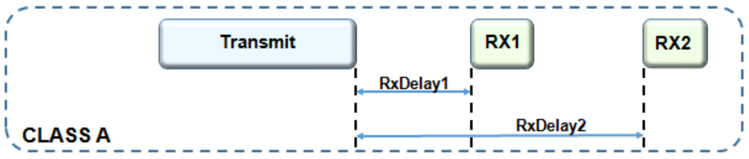


Figure 3 Class A Device Timing Diagram

Class B is extended from class A and used when the applications need to receive extra downlink data. Class B devices are synchronized by periodic beacons (BCN) sent by the gateway as illustrated in Figure 4. The gateway can schedule additional receive windows called ping slots for devices. Therefore, a class B device not only can receive downlink packets after transmission, it can also receive downlink packets periodically. Since its receive windows open more frequently than a Class A device, it is less energy efficient.

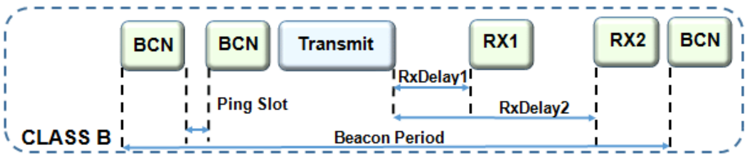


Figure 4 Class B Device Timing Diagram

Class C is also extended from class A and used in main powered actuators. Class C devices can listen to the channel continuously, except in the transmission, as illustrated in Figure 5. The latency for downlink communication can be minimized. It is less energy efficient than Class A and Class B devices.

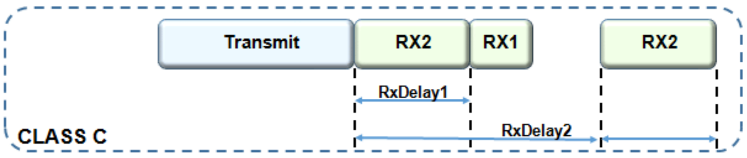


Figure 5 Class C Device Timing Diagram

Class A must be implemented in all end devices so that they provide the basic functionality of bi-direction communication as defined in LoRaWAN standard. Class B and Class C must be compatible with Class A.

2.2.3 LoRaWAN Frequency Plan

The frequency used by LoRa transceiver needs to adhere the regulation of the respective country. Different countries allocate different ISM band for communication using LoRa. According to the regulatory document from The Office of the Communications Authority in Hong Kong, the allocated frequency band is AS923-925 MHz [6]. However, due to hardware limitation, EU868 is used in this project for experimental purposes. Also, different development platforms have different frequency plans. Since The Things Network (TTN) is used in this project, the channel allocation plan follows the standard in TTN. The reasons of choosing EU868 and TTN as the development platform are justified in the section 3.2.3 and section 3.2.4.

Since EU863-870 and TTN are used, the frequency plan is shown as follows.

Uplink:

1. **868.1** - SF7BW125 to SF12BW125
2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
3. **868.5** - SF7BW125 to SF12BW125
4. **867.1** - SF7BW125 to SF12BW125
5. **867.3** - SF7BW125 to SF12BW125
6. **867.5** - SF7BW125 to SF12BW125
7. **867.7** - SF7BW125 to SF12BW125
8. **867.9** - SF7BW125 to SF12BW125
9. **868.8** - FSK

Downlink:

- Uplink channels 1-9 (RX1)
- **869.525** - SF9BW125 (RX2 downlink only)

2.2.4 Initialization of Channels in the Join Process

The end devices must be activated to join the LoRaWAN network. There are two ways of activation – Activation By Personalization (ABP) or Over-The-Air-Activation (OTAA) [5].

Before activation, some EUI and keys must be stored in the end device, including JoinEUI, DevEUI, AppKey and NwkKey, depending on whether ABP or OTAA is used. Below summarize some key components of join process.

EUI and keys [5]

- JoinEUI is an application identifier in IEEE EUI64 address space that uniquely identifies the join server that help the end devices in the join procedures. JoinEUI must be stored in OTAA devices before the join procedure.
- DevEUI is an end device identifier in IEEE EUI64 address space that uniquely identifies the end device. DevEUI must be stored in OTAA devices before the join procedure.
- NwkKey is an AES-128 keys assigned to an end device. It is used to derive other session keys. NwkKey must be stored in OTAA devices before the join procedure.
- AppKey is an AES-128 keys assigned to an end device. It is used to derive application session key. AppKey must be stored in OTAA devices before the join procedure.

After activation, the device would contain the following information.

1. The EUI and keys aforementioned

2. DevAddr

3. NwkSEncKey

4. SNwkSIntKey

5. FNwkSIntKey

6. AppSKey

- Devaddr is the 32-bits address of the end devices. It has a format as shown in table 1 [5].

Bit	31..32-N	31-N..0
DevAddr bits	AddrPrefix	NwkAddr

Table 1 Structure of Devaddr

, where N is an integer in the [7,24] range.

The AddrPrefix is derived from the Network Server's unique identifier NetID.

The NwkAddr is the network address of the end device.

- NwkSEncKey is a network session key of an end device, which is used to encrypt and decrypt MAC commands.
- SNwkSIntKey is a network session key of an end device, which is used to verify the MIC of downlink packet and calculate part of the MIC of uplink packet.
- FNwkSIntKey is a network session key of an end device, which is used to calculate part of the MIC of uplink packet.
- AppSKey is an application session key of an end device, which is used to encrypt and decrypt the payload of messages.

Since OTAA is a more convenient and secure option for joining the network than ABP, only OTAA is discussed and used in this project [5].

OTAA

In OTAA, the join procedure is initiated by the end device. The end device sends a **join request** message as illustrated in table 2 [5].

Size (bytes)	8	8	2
Join request	JoinEUI	DevEUI	DevNonce

Table 2 Structure of Join Request Message

DevNonce is a counter that is incremented whenever the device sends a join request.

If the network server accepts the join, it replies with the **join accept** message as illustrated in table 3 [5].

Size (bytes)	3	3	4	1	1	16 optional
Join accept	JoinNonce	Home_NetID	DevAddr	DlSettings	RxDelay	CFList

Table 3 Structure of Join Accept Message

, where

- JoinNonce is a device specific counter value provided by the server. It is used to create the session keys, including NwkSEncKey, SNwkSIntKey, FNwkSIntKey and AppSKey.
- Home_NetID is the NetId of device's home network.
- DevAddr is the device address of the end device allocated by the network server.

All of the above parameters are not important in the design, so they are not explained in detail in this report.

The following are related to the communication setting.

- DlSettings allows changes of data rate of two receive windows.
- RxDelay is the delay for the receive window.

CFList is a 16 bytes field that can be used to set the frequency of the channels, as shown in table 4 [5].

Size (bytes)	3	3	3	3	3	1
CFList	Ch3	Ch4	Ch5	Ch6	Ch7	CFListType

Table 4 Structure of CFList

The optional 16 bytes CFList field can be used to set the frequency of 5 additional channels [5]. According to the LoRaWAN standard, if CFList present in the join accept message, the end device shall replace all its channel with the CFList, except for the default channels 1 to 3. In other words, after the join process, maximum 8 channels can be set up. Since only eight channels are needed in the experiment, all of the channels can be set up in the join process. To disable a specific channel, the value of that channel is set to zero.

The algorithm of setting up channels for communication in the join process is stated as below.

1. End device transmits a LoRaWAN join request

2. When the gateway receives the RF packet, it performs the cyclic redundancy check to check if the packet is valid. If the packet is valid, it relays the packet to the server

3-4. The server determines whether it should accept the join request. Reply the join accept message to the end device via the gateway. In this process, it defines the i) downlink data rate offset of receive window 1 ii) downlink data rate of receive window 2 iii) DevAddr of the end device iv) RxDelay v) additional channels for end device to transmit data

5. The end device updates its network information, including updating the uplink and downlink channels

Figure 6 illustrates the process of setting up channels for communication.

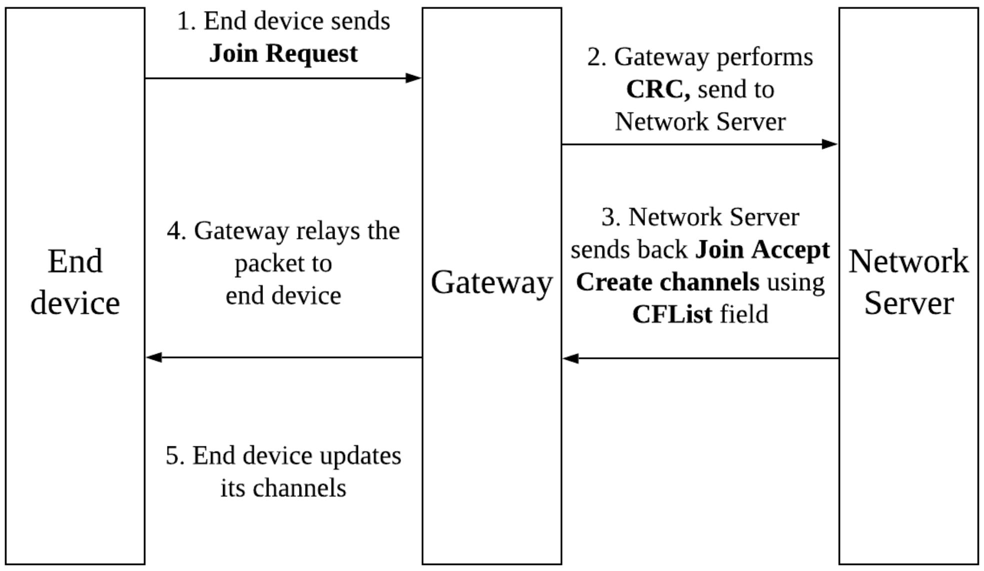

End device
Gateway
Network Server
1. End device sends **Join Request**
2. Gateway performs **CRC,** send to Network Server
3. Network Server sends back **Join Accept** **Create channels** using **CFList** field
4. Gateway relays the packet to end device
5. End device updates its channels

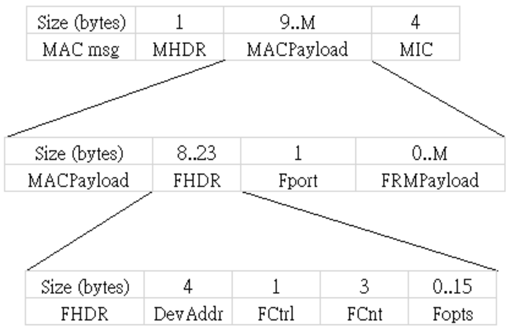
Figure 6 Procedure for Creating Channels during Joining

2.2.5 MAC Command Exchange

Since channel access control protocol is in the MAC layer, it is important to investigate what functions does LoRaWAN provide which enable the implementation of frequency hopping schemes on LoRaWAN. In the MAC layer, devices communicate with each other using MAC message.

The MAC message consists of a single-octet MAC header (MHDR), MAC payload (MACPayload) and a four-octet message integrity code (MIC) [5]. For the MACPayload, it consists of a four-octet frame header (FHDR), a single-octet frame port (FPort) and a frame payload (FRMPayload). For the FHDR, it consists of a four-octet device address (DevAddr), a single-octet frame control (FCtrl), a three-octet frame counter (FCnt) and a frame options field (FOpts).

The structure of MAC message is illustrated in figure 7.



Size (bytes)	1	9..M	4
MAC msg	MHDR	MACPayload	MIC

Size (bytes)	8..23	1	0..M
MACPayload	FHDR	Fport	FRMPayload

Size (bytes)	4	1	3	0..15
FHDR	DevAddr	FCtrl	FCnt	Fopts

Figure 7 MAC Message Structure

Therefore, the minimum size of MAC Message is 14 bytes (without FRMPayload and Fopts). The maximum size of MAC Message is 255 bytes as stated in the LoRaWAN specification [5]. The size of MAC message is important as it represents the airtime needed to transmit the packet.

The network parameters of devices are exchanged via MAC command. There are two ways of sending MAC commands. The first way is piggybacking the commands in FOpts field. The second way is sending as a separated data frame, in which FPort is set to 0 and the commands are placed in FRMPayload. The main constraint is that the FOpts can maximumly contains 15 octets of MAC command/answer.

The algorithm of sending MAC commands from the end devices and the network server is explained as follows.

For the end device, it first checks if the MAC commands exceed 15 octets. If it exceeds, then the commands must be sent in a separate data frame, otherwise it checks if the application layer has anything to send. If there is something to send, the commands are piggybacked in the FOpts field, otherwise the device send it as a separate data frame. A diagram illustrating the MAC command exchange algorithm for end devices is shown in figure 8.

For the network server, it first checks if the MAC commands exceed 15 octets. If it exceeds, then the commands must be sent in a separate data frame, otherwise it checks if the application layer has anything to send. If there is something to send, it checks if the MAC command is received in a single frame or otherwise send as a separate data frame. If the MAC command is received in a single frame, it is forced to send the MAC command separately. Otherwise, it piggybacks the command in FOpts field. A diagram illustrating the MAC command exchange algorithm for end devices is shown in figure 9.

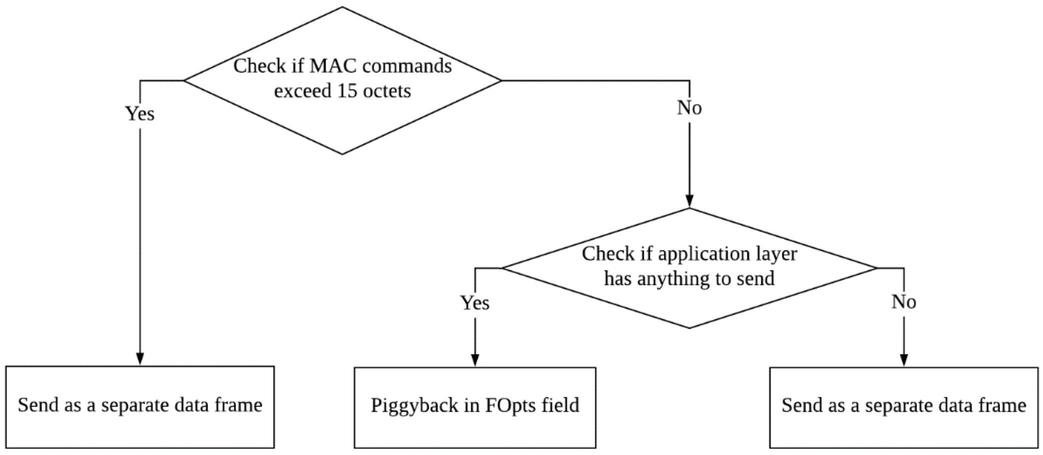

Check if MAC commands exceed 15 octets
Yes
No
Check if application layer has anything to send
Yes
No
Send as a separate data frame
Piggyback in FOpts field
Send as a separate data frame

Figure 8 MAC Command Exchange Algorithm for End Devices

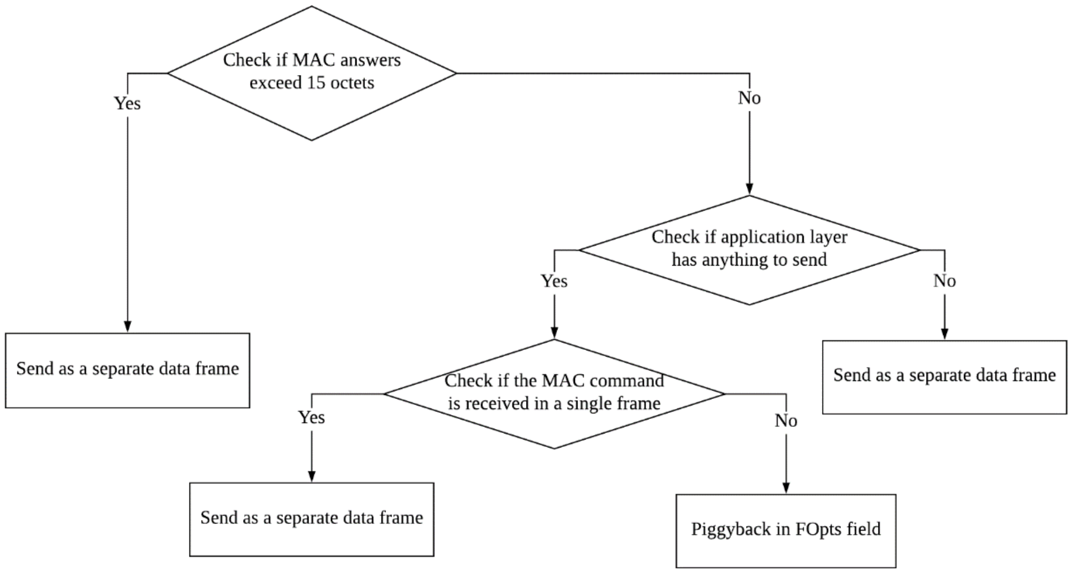

Check if MAC answers exceed 15 octets
Yes
No
Check if application layer has anything to send
Yes
No
Send as a separate data frame
Check if the MAC command is received in a single frame
Send as a separate data frame
Yes
No
Send as a separate data frame
Piggyback in FOpts field

Figure 9 MAC Command Exchange Algorithm for Network Server

This section concludes the mechanism of MAC command exchange.

2.2.6 Creation and Modification of Channel

Creating a channel is crucial in communication. This section discusses how the gateway create uplink and downlink channels for the end devices. Uplink channel is for the end device to send packet to the gateway and downlink channel is for the gateway to send packet to the end device. By establishing both the uplink and downlink channel, two-way communication can be achieved.

2.2.6.1 Creation/Modification of Uplink Channel

The uplink channels of end devices are created and modified by the gateway using the **NewChannelReq** command as shown in table 5. It defines the center frequency of the channel and the range of data rates that can be used on this channel.

Size (bytes)	1	3	1
NewChannelReq	ChIndex	Freq	DrRange

Table 5 Structure of NewChannelReq Command

ChIndex

ChIndex is the index of the channel to be created or modified. The maximum number of channels that an end device can handle is 16. Therefore, ChIndex is range from 0 to 15. However, the maximum number of channels that can be listened simultaneously by a gateway is 8. Although it is possible to combine multiple gateway to utilize more channels, it is out of the scope of the study. In this project, there are eight channels ranging from 0 to 7.

Freq

Freq represent the center frequency of the channel, in a format of 24 bits unsigned integer. The actual center frequency is $100 * Freq$ (Hz). However, frequencies below 100 MHz are reserved. In order words, the frequency range is [100, 1670] MHz. A Freq value of 0 disable the channel.

Recall that LoRa operates in 867-869 MHz for Europe, 902-928 MHz for North America and 470-510MHz for China. In the project, 868 MHz LoRa transceiver and gateway are selected. The common practice of choosing frequency as suggested by The Thing Network at 868 MHz is {868.1, 868.3, 868.5, 867.1, 867.3, 867.5, 867.7, 867.9}.

DrRange

DrRange specifies the range of data rate of the uplink of the channel.

Bits	7..4	3..0
DrRange	MaxDR	MinDR

Table 6 Structure of DrRange Field

As shown from the table 6, bits 7..4 specifies the maximum data rate and bits 3..0 specifies the minimum data rate.

After the end device receives the NewChannelReq, it replies a NewChannelAns command to acknowledge the reception.

2.2.6.2 Modification of Downlink Channel

The frequency of downlink channel is by default to be the same as the uplink channel as shown in the table 7. By utilizing the **DlChannelReq**, the gateway can modify the frequency of the downlink channel.

Size (bytes)	1	3
NewChannelReq	ChIndex	Freq

Table 7 Structure of DlChannelReq Command

The definition of ChIndex and Freq is stated in section 2.2.6.1.

After the end device receives the DlChannelReq, it replies a DlChannelAns command to acknowledge the reception.

2.2.6.3 Conclusion

In conclusion, channels can be modified using NewChannelReq and DlChannelReq MAC command following the procedures stated in section 4.1.2 and the figure 10 below.

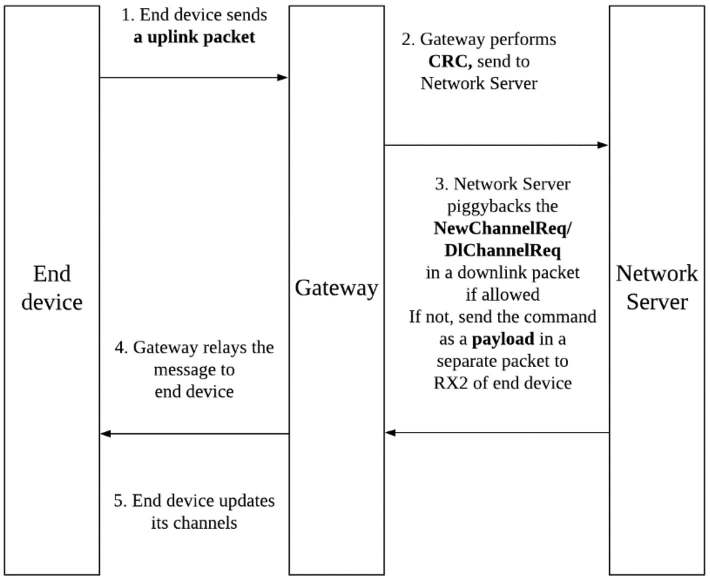

End device
Gateway
Network Server
1. End device sends a uplink packet
2. Gateway performs CRC, send to Network Server
3. Network Server piggybacks the NewChannelReq/ DlChannelReq in a downlink packet if allowed If not, send the command as a payload in a separate packet to RX2 of end device
4. Gateway relays the message to end device
5. End device updates its channels

Figure 10 Procedure for Modifying Channels

2.2.7 Channel Access Control Protocol

Since LoRa is a wireless communication technique and the transmission medium is shared in wireless communication, collision occurs if two or more devices simultaneously sending packet using the same frequency of electromagnetic wave with same spreading factor. Collision results in packet loss and the subsequent packet retransmission leads to high energy consumption. LoRaWAN uses Aloha protocol, in which if the sender has packet to send, just send it. The main reason of using Aloha protocol in LoRaWAN is the simplicity. The simple design provides the advantage of low power consumption of the transceiver. However, collisions can easily occur. To avoid collision, channel access control protocols can be used. In the following, different commonly used channel access control protocols for wireless communication are discussed.

2.2.7.1 Classification of Channel Access Control Protocols

Channel access control protocols can be classified into frequency division multiple access (FDMA), time division multiple access (TDMA), code division multiple access (CDMA), space division multiple access (SDMA) and power division multiple access (PDMA). Those channel access method multiplex the signal in frequency domain, time domain, coding, space domain and power domain respectively. Among these methods, FDMA, TDMA and CDMA are commonly used in communication technology in IoT.

Frequency hopping techniques belong to CDMA and usually called frequency hopping code division multiple access. It can be classified as channel-ignorant or channel-aware. For channel-ignorant frequency hopping, the hopping sequence is determined regardless of channel quality [5]. For channel-aware frequency hopping, the hopping sequence is determined based on channel quality. It can be further classified as reduced-hop-set and probabilistic-channel-usage technique. Reduced-hop-set means that the bad channels are eliminated from the frequency list. Probabilistic-channel-usage means that bad channels would have a lower probability in selection, but not eliminated. Figure 11 illustrates the classification of frequency hopping techniques.

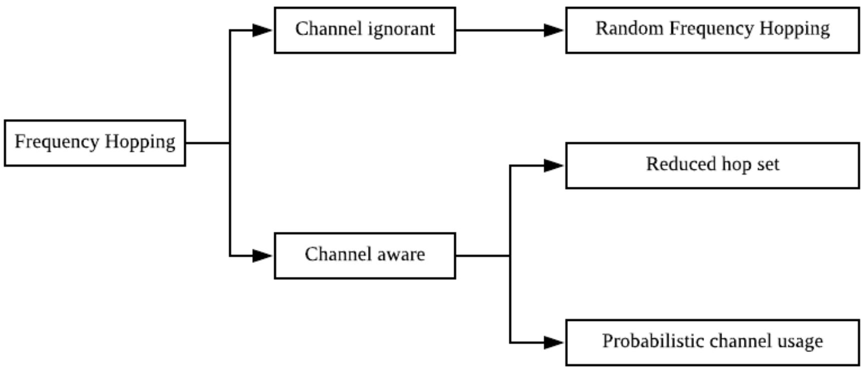


Figure 11 Classification of Frequency Hopping Techniques

A commonly used channel ignorant protocol is random frequency hopping (RFH). RFH is a method of transmitting signal with different frequency channels randomly. A device first randomly or pseudo randomly select a frequency channel. Then it transmits the packet using that channel. However, the hopping sequence of the channels can be designed to improve the reliability of the system. Also, by planning uplink channels and downlink channels, the packet delivery ratio of critical packet and downlink packet can be increased. This is the focus of the project.

Limited by the scope of the project, channel aware protocols are not investigated in this project.

Another channel access control protocol is carrier sense multiple access with collision avoidance (CSMA/CA), which is used in IEEE802.11 RTS/CTS mechanism and IEEE 802.15.4/Wireless PAN standard. A simplest way to describe CSMA/CA is Listen Before Talk (LBT). Before transmission, the sender listens to the channel to check if there is any other device sending packets. If so, it waits a random duration then listens again. It only sends packet if the channel is clear. However, there exists problem in this approach which makes it unsuitable to be implemented in LoRa and they are explained in the following section.

2.2.7.2 Problem of CSMA/CA in LoRa

There are three problems in implementing CSMA/CA in LoRa.

The first problem is that determining whether a packet can be successfully received by the gateway is depending on the receiver, not the sender. When an end device senses the channel to check if other end devices are also sending the packet to the gateway, if that two end devices are out of range of each other, they cannot ensure that another device is sending or not, and this is called the hidden terminal problem [8]. Therefore, listening to the channel before sending at the transmitter may not yield a satisfactory improvement, especially for the communication system with large time on air like LoRa.

The second problem is that LoRa chips developed by Semtech do not provide a reliable mechanism to ensure that the channel is not occupied. Since LoRa transceiver can receive packet below the noise floor, using RSSI to determine whether there is a packet in a channel is not reliable [9]. Therefore, LoRa chips provide a mechanism called channel activity detection (CAD) which is claimed to be able to detect the preamble of the packet [10]. Detection of preamble is obviously insufficient to implement a reliable LBT mechanism as failure of detecting payload of a packet does not guarantee that the channel is clear. Although a research performed an experiment and found that CAD not only can detect preamble, but the whole packet, the research also points out that the reliability of CAD decreases significantly with increasing transmission range, and this problem exacerbate if the network traffic is high [9]. Specifically, CAD becomes unreliable even only at a transmission distance of 400m in a dense environment. The failure in detecting channel activity using CAD leads to difficult implementation of reliable CSMA/CA mechanism in LoRa.

The third problem of CSMA/CA is that it requires opening of receive windows to detect the activity on the channel. This increases the energy consumption of the devices, which is contradictory to the design principle of a LPWAN, as the aim is to maximize the life time of battery powered device such that replacement of battery would not occur frequently.

Therefore, CSMA/CA is not implemented in this project. In contrast, the focus of the study is on the frequency hopping schemes and channel planning schemes.

3. Methodology

The methods of investigation can be split into three parts, software implementation, hardware implementation and experimental design. For the software implementation, a simulator is built for evaluating the performance of protocols. The purpose of developing a simulator is that the performance of a large-scale network can be evaluated without deploying the hardware at site. For hardware implementation, the protocols are implemented in hardware. The purpose of realizing the protocols in hardware is to demonstrate the compatibility of the protocols and the industrial standard. In the experimental design, the traffic model used and the selection experimental setting are justified.

3.1 Software Consideration

3.1.1 Programming Language Selection

In this part, the main target is to select a programming language and library package to develop a LoRa network simulator. For the programming language, the major consideration is the development speed as time is limited in the project. It is well known that C/C++/Java program has higher execution speed, but they have the drawback of very long development time, especially for new developers who are not familiar with the language syntax. Therefore, Python is usually used in scientific research for the following reasons. Firstly, the language syntax is simpler. It is much easier for researcher to acquire the skills needed to implement their thought. Secondly, there are more open-source library, such as SimPy, which is a discrete event simulator package, is often used in simulator development.

Another reason of choosing Python is that there exists an open-source LoRa simulator called LoRaSim which is developed in Python. T. Voigt and M. Bor in the University of Lancaster developed LoRaSim for simulating collisions and analyzing scalability in LoRa networks. Many researches have used this simulator to test their protocol [1][2]. In this project, instead of rebuilding everything from scratch, the source code of LoRaSim is modified. The details of the implementation are discussed in the section 4.2.

3.2 Hardware Consideration

3.2.1 Micro-controller and Micro-processor Selection

Selecting a micro-controller for end devices has the following considerations. Firstly, the cost should be low enough as the number of end devices in IoT application could be quite large. Secondly, the processing power and flash memory should be high enough to provide sufficient computational power and store the compiled program. Thirdly, the development platform should be popular and thus library could be easily found to reduce the development time.

Finally, the micro-controller should provide a communication interface such as SPI for connecting the LoRa transceiver.

Among different choices of platform, Arduino appears to be the most popular platform which provides sufficient support for development. Table 8 shows a comparison on the chip architecture, flash memory, SRAM, clock speed, SPI interface and the cost of different Arduino development board.

Development Board	Architecture	Flash (KB)	SRAM (KB)	Clock (MHz)	SPI interface	Cost (Official website)
Arduino Uno	AVR	32	2	16	Provided	$22 USD
Arduino Mega	AVR	256	8	16	Provided	$38.5 USD
Arduino Due	ARM	512	96	84	Provided	$38.5 USD

Table 8: Comparison of Arduino Development Boards

For the chip architecture, there are more libraries supporting AVR instead of ARM. However, both Arduino Uno and Arduino Mega does not provide enough memory for storing the compiled program. Although the cost of Arduino Due is inevitably higher than the UNO and ARM architecture is used, this is the only choice we have.

Selecting a micro-processor for gateway has a similar consideration as the end devices. However, since gateway requires more computational power while keeping the power consumption low enough, Raspberry Pi is the best choice we have. There are lots of support from community which makes the implementation easier as Raspberry Pi is popular.

3.2.2 LoRa Transceiver Selection

LoRa chips are built by the Semtech company. They can be divided into 2 types, namely SX12xx series and SX13xx series. SX12xx chips are capable to listen to only one channel. Therefore, they are used in the end devices. SX13xx chips are capable to listen to multiple channels. Therefore, they are used in the gateway.

Common selection of transceiver for end devices are SX1276/SX1277/SX1278/SX1279. The main consideration is whether the existing Arduino-LoRaWAN libraries supporting these chips. After researches, most of the libraries only support SX1276/SX1277. Hence, SX1276 is selected. To provide a better performance and simplify the hardware implementation, a LoRa Shield for Arduino developed by Dragino is selected. It is compatible with Arduino Due.

Common selection of transceiver for gateway is SX1301. The major drawback of SX1301 is the high cost. However, since simultaneously listening to multiple channels is the essential requirement in implementing the random frequency hopping, it must be purchased. Among different LoRaWAN gateway modules, RAK831 is a common and relatively low-cost choice, which is compatible with Raspberry Pi 2 Model B.

There are two ways to implement the gateway. In common practice, a gateway module based on SX1301 chip developed by Semtech is used, which is capable to listen to up to 8 channels simultaneously. Another approach is to build the gateway with multiple Ra-02 modules which are capable to listen to a single channel. Assuming the gateway need to listen to 8 channels, 9 Arduino Due and 8 Ra-02 Modules are needed. Each Ra-02 module is connected to an Arduino Due for receiving packet from end devices and sending acknowledgement to end devices for a particular channel. Those Arduino are further connected to a master Arduino, which gathers the information of those channels. Table 3 compares the cost of these two approaches.

Item	Cost per Item (HKD)	Quantity	Cost (HKD)
Approach 1			
Ra-02	30	8	240
Arduino Due	80	9	720
		Total Cost (HKD):	960
Approach 2			
RAK 831 with Raspberry Pi 2 Model B	1136	1	1136
		Total Cost (HKD):	1136

Table 9 Comparison of the Method for Building the Gateway

From table 9, it can be observed that the cost of approach 2 is higher than approach 1 by around 20%. However, for the easiness of implementation and ensuring the quality of measurement, approach 2 is selected. The left hardware of figure 11 is the Dragino LoRa shield with the Arduino Due and the right hardware of figure 11 is the RAK831 gateway module.

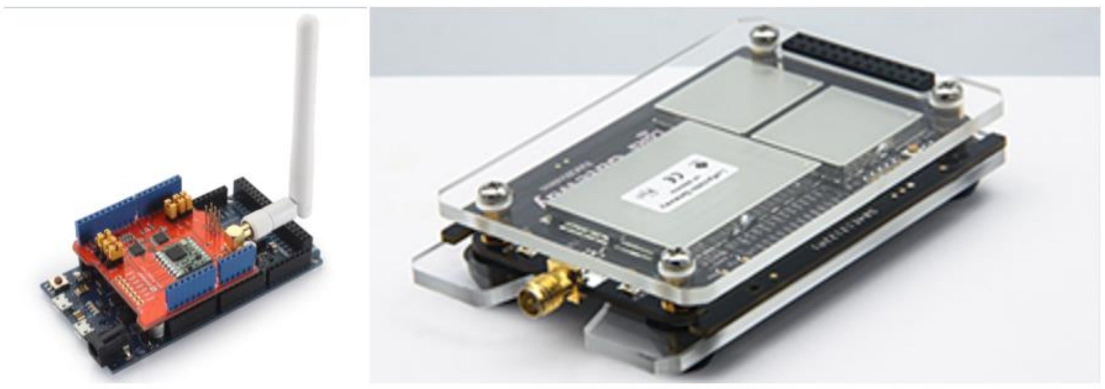


Figure 11 Dragino LoRa Shield and RAK831 Gateway Module

3.2.3 Frequency Band Selection

As mentioned in section 2.2.3, there are regulation on the frequency band that can be used for LoRa in different countries. AS 923-925 MHz should be used in Hong Kong. However, RAK831 module does not provide this option as it only has version of 433MHz/868MHz/915MHz. Therefore, the selection is limited to these three frequency bands. Since most of the active developers of LoRa application are in Europe, selecting EU868 allows me to receive better support from the community. Therefore, for both the Dragino LoRa shield and RAK831 gateway module, EU868 frequency band is selected just for the experimental purpose. AS 923-925 should be selected for the large-scale deployment of the system to comply with the regulation.

3.2.4 Development Platform Selection

To the best of my knowledge, there are two development platforms for LoRaWAN application development – The Things Network (TTN) and LoRaServer. TTN is the most popular development platform mainly because it is easy to use. Devices are simpler to set up comparing to LoRaServer. However, due to its simplicity, it is very difficult to modify the backend system in TTN. For example, it cannot handle MAC command exchange for every possible command. LoRaServer can be used to build a private LoRaWAN network which allows modification of backend system. Nevertheless, it is not necessary to implement MAC command exchange for evaluating the performance of frequency hopping schemes and channel planning schemes. Therefore, TTN is selected in this project for simplicity. For the future development, LoRaServer can be used to implement a complete system.

3.3 Design of Experiment

For testing the effectiveness of the protocols, we first need to understand what parameters to be tested on, then design the experiment with justification.

Recall that this project aims at evaluating the performance of frequency hopping schemes and channel planning schemes in LoRaWAN. The measure of performance is the packet delivery ratio, which defines as the ratio of data packets, not including the retransmitted packets, passing the CRC received by the receivers to the packets sent by the senders. The number of channels, the type of frequency hopping schemes and channel planning schemes are the independent variables in the experiments. The experiments are listed in table 10.

Experiment	Title	Software	Hardware
1	The Effect of Using Different Number of Channels on the PDR of Uplink Packets	✓	✓
2	The Effect of Round Robin Scheduling with Non-sequential Channel on the PDR of Uplink Packets	✓	✗
3	The Effect of Using Separated Channel for Acknowledgement on the PDR of Acknowledgement Packets	✓	✓
4	The Effect of Using Separated Channel for Acknowledgement with Different Number of Nodes on the PDR of Acknowledgement Packets	✓	✗
5	The Effect of Using Separated Channel for Acknowledgement with Different Number of Gateways on the PDR of Acknowledgement Packets	✓	✗
6	The Effect of Using Separated Channel for Critical Packet on the PDR of Critical Packets	✗	✓

Table 10 List of Experiments

Regarding the experimental settings, several data traffic models can be used, including periodic data reporting model and event driven data generation model [1]. The use of data traffic models simulating the real-life application is useful for deployment of the system.

Periodic data reporting model means that each end device transmits a single packet per day. The experiment in this project that use the periodic reporting model also send a packet once per day. The simulation period is 3 years such that each end devices would send around 1000 packets in the simulation. Research states that if all smart meters report the data simultaneously, the packet collision probability is high [1], and this is investigated in experiment 2.

Event driven data generation model means that the data is generated following a Poisson distribution with average λ. The experiments in this project that use the event driven data generation model also sends the data following a Poisson distribution. The λ specified in that paper is 5 minutes and 1 hour, which consider to be impractical for the hardware experiment. Instead, λ is set to be 5 seconds in this report. The experiment end when there are 400 packets received, unless otherwise specified.

These two models are considered to be sufficiently general which can be used to model most of the real-life application. Hence, if the experiments prove that there is improvement on network performance by applying those channel hopping schemes and channel plans, there should also be an improvement in a real-life deployment.

Regarding the experimental setting, unless otherwise specified, {SF7, BW125, CR4/5} is used as this is the default choice in LoRaWAN. The packet size of uplink packets and downlink packets are 18 bytes and 14 bytes respectively. Downlink packets have the minimal size containing no payload, thus they are 14 bytes. Uplink packets contains 4 bytes of payload as uplink packets are used to report data, thus they are 18 bytes. The hardware consists of four end devices and one gateway.

Experiment 1 aims at evaluating the effect of using different number of channels on the PDR of uplink packets. It is done on both the software and the hardware. Event driven data generation model is selected. The number of channels is selected from {1, 2, 4, 8} as there are maximum eight uplink channels of LoRa in EU868 band as defined in TTN. Unconfirmed packets are sent by the end devices. The PDR of the uplink packet is measured by counting the number of received and missed packets in TTN console, and the result is verified by the packet logger in the gateway introduced in section 4.3.3. Also, this experiment also aims at confirming the functionality of the simulator, so that it can be used for the scalability analysis in experiment 4 and 5. The functionality of the simulator is confirmed if the result generated from hardware experiment matches that from the simulator.

Experiment 2 aims at evaluating the effect of round robin scheduling with non-sequential channel on the PDR of uplink packets. It is done on software. The reason of not performing this experiment on hardware is that it is difficult to control the end devices to send packet simultaneously, and hence the worst-case scenario of simultaneous transmission cannot be generated effectively. Periodic data reporting model is selected. Unconfirmed packets are sent by the end devices. The number of channels is selected from {1, 2, 4, 8} as there are maximum eight uplink channels of LoRa in EU868 band as defined in TTN channel list. The PDR of the uplink packet is calculated in the LoRa simulator.

Experiment 3 aims at evaluating the effect of using separated channel for acknowledgement on the PDR of acknowledgement packets. It is done on both the software and the hardware. Event driven data generation model is selected. Confirmed packets are sent by the end devices. There are eight uplink channels as specified in TTN channel list. The downlink channel frequency is set to be 869.525 MHz as explained in section 4.1.5. For the software, the PDR of the acknowledgement packet is measured in the simulator. For the hardware, the PDR of the acknowledgement packet is measured by counting the number of received and "retry confirmed" packets in TTN channel list, and the result is verified by the packet logger in the gateway introduced in section 4.3.3. The "retry confirmed" indicates that the end device does not receive an acknowledgement from the network server and thus it retransmits the packet which recognized by the TTN as "retry confirmed" packet. Also, this experiment also aims at confirming the functionality of the simulator, such that it can be used for the scalability analysis in experiment 4 and 5. The functionality of the simulator is confirmed if the result generated from hardware experiment matches that from the simulator.

Experiment 4 aims at evaluating the effect of using separated channel for acknowledgement with different number of nodes on the PDR of acknowledgement packet. This is a scalability analysis of the LoRaWAN system. Due to the limitation on the quantity of hardware, the result can only be simulated in the simulator. Event driven data generation is selected. Confirmed packets are sent by the end devices. The channel list is the same as the experiment 3. The number of end devices is selected from {8, 16, 32, 64, 128, 256, 512, 1024, 2048}. There are four gateways in this simulation. The PDR of the acknowledgement packet is measured in the simulator.

Experiment 5 aims at evaluating the effect of using separated channel for acknowledgement with different number of gateways on the PDR of acknowledgement packet. This is also a scalability analysis of the LoRaWAN system. Due to the limitation on the quantity of hardware, the result can also only be simulated in the simulator. Event driven data generation is selected. Confirmed packets are sent by the end devices. The channel list is the same as the experiment 3. The number of base stations is selected from {8, 16, 32, 64, 128, 256}. The number of end devices is set to be 1024 because it is impractical if the number of end devices is smaller than the number of gateways in the real-life deployment. Therefore, it is set to be four times of the number of gateways, so as to be consistent with the hardware setup in this project. The PDR of the acknowledgement packet is measured in the simulator.

Experiment 6 aims at evaluating the effect of using separated channel for critical packet on the PDR of critical packets. It is done on the hardware only due to time limitation of the project. Event driven data generation model is selected. Confirmed packets are sent by the end devices. There are seven uplink channels, i.e. channel 0 - 6. The channel 7 of frequency 867.9 MHz is reserved for the transmission of critical packets. There are one gateway and four end devices. Device 1 transmit a critical packet every 5 packets. Device 2 transmit a critical packet every 6 packets. Device 3 transmit a critical packet every 7 packets. Device 4 transmit a critical packet every 8 packets. These transmission intervals are selected because we assume that there are four different applications where they have different frequency of transmission of critical packets. The PDR of the critical packet is measured by counting the number of received and missed critical packets in TTN console, and the result is verified by the packet logger in the gateway introduced in section 4.3.3.

3.4 Summary

In conclusion, Arduino Due is selected as the micro-controller for end devices. Dragino LoRa shield is selected as the LoRa transceiver for end devices. Raspberry Pi 2 Model B is selected as the micro-computer for the gateway. RAK831 is selected as the LoRaWAN module for the gateway. There are one gateway and four end devices for hardware setup. Six experiments are performed in this project.

4. Design and Implementation

This section is divided into three subsections. Section 4.1 discusses the protocol design, including frequency hopping schemes and the channel planning schemes. Section 4.2 discusses the design and implementation of the software simulator. Section 4.3 discusses the design and implementation of hardware.

4.1 Design of Protocol

This section discusses the techniques that can be applied to the LoRaWAN system to improves its network performance. Section 4.1.1 discusses the frequency hopping schemes. Section 4.1.2 discusses the channel planning schemes.

4.1.1 Frequency Hopping Schemes

There are several methods for end devices to hop between different channels.

1. Random Selection

2. Round Robin Scheduling with Sequential Channel

3. Round Robin Scheduling with Non-sequential Channel

For random selection, a random number in range [0, number of channels] is generated by a random number generator at each time of transmission. This gives high level of randomness, but extra processing is needed. It increases the latency and power consumption. The latency problem is not a major concentration for LoRa-based application. However, that extra processing is not necessary.

For round robin scheduling with sequential channel, the channel is selected sequentially in a round robin fashion. For example, there are 4 channels can be selected, namely, {C1, C2, C3, C4}. The channels are selected sequentially as C1, C2, C3, C4, C1, C2, and so on. This method introduces a forever-collision problem when devices in the network are synchronized and send packets at the same time. If their transmitted packets collide with channel C_j, since the channels in the channel list is sequential, they will transmit the next packet using same channel C_{j+1} if $j+1 \leq m$ or C_1 if $j+1 > m$, where m is the largest channel number. That means their next packet definitely collide again.

An example of happening forever-collision problem is the smart meters application in which end devices report data to the network server simultaneously at a particular moment periodically. Another scenario is sudden event, such as a detector detects that there is a fire or an earthquake. In this case, multiple end devices report the event simultaneously. This leads to high probability of packet collision. In real-life application, there often are other measures to alleviate this problem, such as introducing a random delay before transmission. However, this problem still exists in the worst-case scenario. Also, adding random delay is inappropriate for reporting sudden events as the network server needs to be alerted as fast as possible.

For round robin scheduling with non-sequential channel, the channel sequence in the channel list is randomly determined. For example, there are 4 channels can be selected, namely, {C1, C2, C3, C4}. A channel list is randomly generated, such as {C2, C1, C4, C3}. The channels are selected as C2, C1, C4, C3, C2, C1 and so on. Another end device may have a channel list of {C3, C4, C1, C2} and its channels are selected as C3, C4, C1, C2, C3, C4 and so on. Obviously, this method introduces two benefits. Firstly, extra processing for randomly selecting a channel is not needed. Secondly, this reduces the probability of occurring the forever-collision problem as this problem occurs only when their channel lists are identical. Assume there are N channels. The probability of two end devices having the same channel list is

$$P(same\ channel\ list) = \frac{1}{N!} \tag{5}$$

Therefore, using non-sequential channel can greatly reduce the chance of occurring the forever-collision problem.

4.1.2 Channel Planning Schemes

This section discusses the effect of allocating a specific channel for the transmission of specific types of packets. Two types of packets are discussed in this report, including acknowledgement packets and critical packets.

Since acknowledgement and critical packets are transmitted less frequently comparing to normal packets, it is beneficial to allocate a specific channel for the transmission of acknowledgement or critical packets to reduce the chance of collision by the normal packet.

4.1.2.1 Using Separated Channel for the Transmission of Acknowledgement Packets

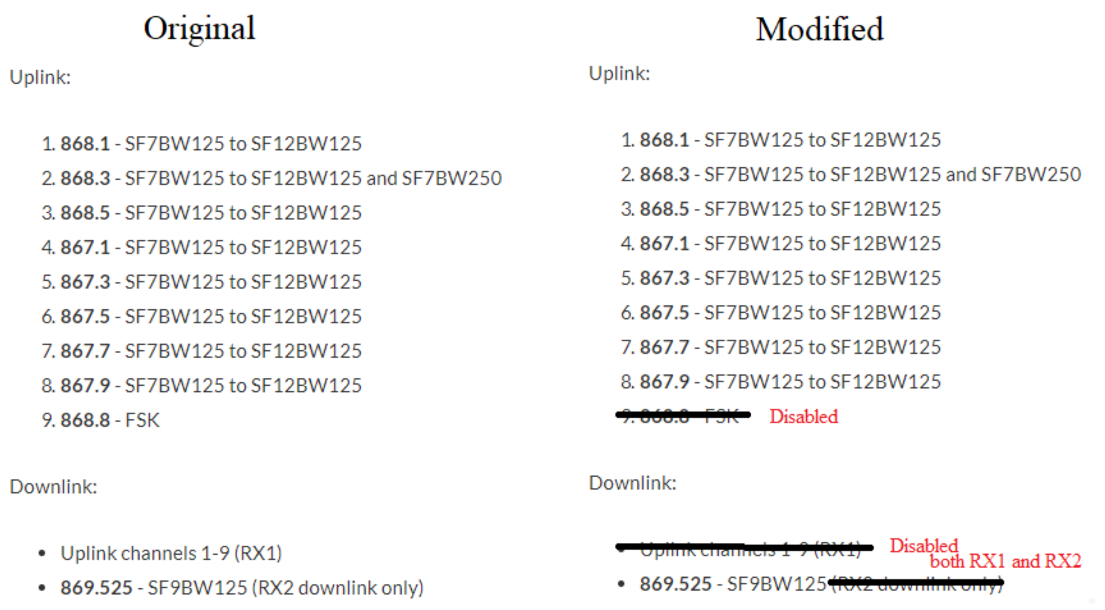
According to the LoRaWAN standard, there are two types of packet – confirmed and unconfirmed. If the receiver receives a confirmed packet, it must reply an acknowledgement packet to the sender. If the receiver receives an unconfirmed packet, it does not need to transmit any downlink packet. If the sender does not receive an acknowledgement, depending on the application layer implementation, the sender may retransmit the packet until it receives an acknowledgement or gives up after several times of retransmission. The retransmission is detrimental to the network performance as the network is more congested when there are more transmissions.

There are two main reasons of not receiving an acknowledgement packet. First, packet loss occurs in uplink. The network server does not receive the uplink packet from the end device. It can be due to the collision at the gateway, failed CRC or buffer overflow at the gateway. Second, packet loss occurs in downlink. The network server sends an acknowledgement to the end device via the gateway. However, since the receive window is of the same frequency of the uplink channel, collision occurs when other end devices or gateways send packet using that frequency. The acknowledgement packet collides with other packets at that end device thus it cannot receive the acknowledgement.

The first reason of packet loss can be mitigated with channel access control protocols or applying different frequency hopping schemes as discussed in section 4.1.1. The second reason can be mitigated with channel planning.

Using a separated channel for transmission of downlink packet can reduce the chance of collision. There are two explanations of the benefits provided from this method. Firstly, the acknowledgement packet at the end device will never collide with packets sent from other end devices. Secondly, although the acknowledgement packet may collide with other downlink packet sent from other gateways, the chance is low due to the assumption that the number of gateways is far less than the number of end devices. Recall that a LoRaWAN network has a star of stars topology which means multiple end devices are connected to a single gateway. Since the transmission range of the gateway is large and it can handle thousands of end devices, it is logical to assume that each device is only in the range of few numbers of gateways. Therefore, the probability of collision of downlink packet is low. In the extreme case when an end device is only in the range of a single gateway, it is impossible for the downlink packet collision to occur, since a gateway can only transmit a single packet at a time.

From section 2.2.3, it can be observed that there are eight channels for uplink and downlink transmission using LoRa. There is also one channel for downlink for RX2 only. Therefore, the simplest way is to extend the downlink for RX2 to both the RX1 and RX1, since the acknowledgement packet from TTN is sent to the RX1 of the end device. Below shows the modified frequency plan.


Original

Uplink:

1. **868.1** - SF7BW125 to SF12BW125
2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
3. **868.5** - SF7BW125 to SF12BW125
4. **867.1** - SF7BW125 to SF12BW125
5. **867.3** - SF7BW125 to SF12BW125
6. **867.5** - SF7BW125 to SF12BW125
7. **867.7** - SF7BW125 to SF12BW125
8. **867.9** - SF7BW125 to SF12BW125
9. **868.8** - FSK

Downlink:

- Uplink channels 1-9 (RX1)
- **869.525** - SF9BW125 (RX2 downlink only)

Modified

Uplink:

1. **868.1** - SF7BW125 to SF12BW125
2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
3. **868.5** - SF7BW125 to SF12BW125
4. **867.1** - SF7BW125 to SF12BW125
5. **867.3** - SF7BW125 to SF12BW125
6. **867.5** - SF7BW125 to SF12BW125
7. **867.7** - SF7BW125 to SF12BW125
8. **867.9** - SF7BW125 to SF12BW125

~~9. 868.8 - FSK~~ Disabled

Downlink:

~~Uplink channels 1-9 (RX1)~~ Disabled both RX1 and RX2

- **869.525** - SF9BW125 ~~(RX2 downlink only)~~

Figure 12 Channel Plan with Separated Downlink Channel

From the Figure 12, it can be observed that the downlink channels of receive window 1 (RX1) are the same as the uplink channels. In other words, if the gateway receives a packet from a particular channel, the network server replies using that channel. For receive window 2 (RX2), a specific channel with 869.525 MHz, SF9 and BW125 is used.

Since TTN sends downlink packets to the RX1 of end devices, the end devices receive downlink packet from the channel that used to transmit the corresponding uplink packet. To create a specific channel for downlink communication, we can modify the transmitting frequency of the gateway. Regardless of the frequency specified in downlink packets, the transmitter will transmit the packets using 869.525 MHz. The procedure of modification is illustrated in section 4.3.4.

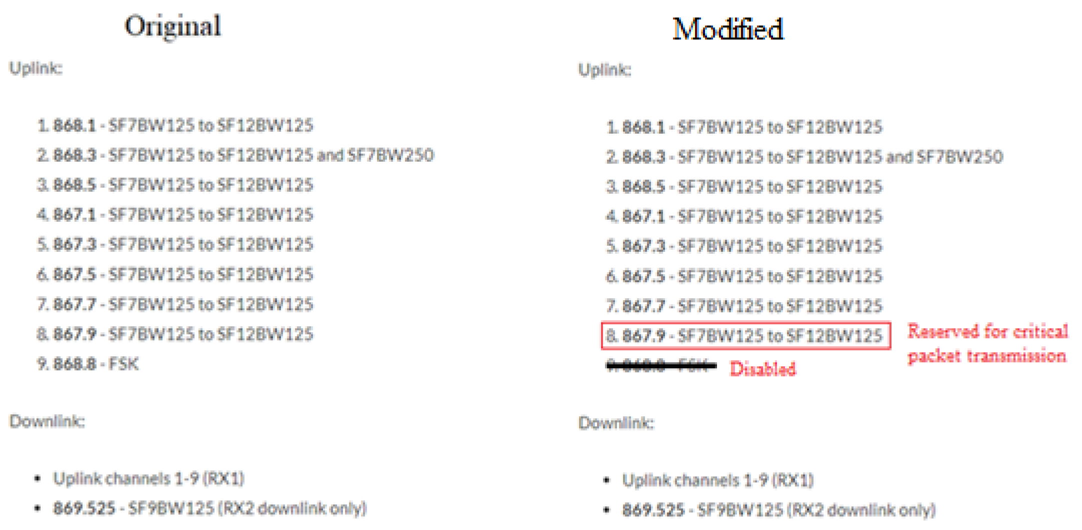
4.1.2.2 Using Separated Channel for the Transmission of Critical Packets

In some applications, there are critical packets of which the probability of collision needs to be minimized as the network server needs to be alerted as fast as possible.

To minimize the chance of collision of critical packets. A separated channel can be allocated for the transmission of critical packets. Critical packets are sent less frequently comparing to the non-critical packets by nature. Therefore, the packet sent in a separate channel is less likely to be collided.

The drawback of using a separated channel for transmission of critical packets is that the network bandwidth is not utilized efficiently. Since critical packets are sent less frequently, the separated channel is unutilized at most of the time. On the other hand, the non-critical packets share a smaller number of channels, from N to N-1, where N is the maximum number of channels and $N > 2$. The probability of collision increased by a factor of $\frac{N}{N-1}$. The increase is significant if there are few available channels.

From section 2.2.3, it can be observed that there are eight channels for uplink and downlink transmission using LoRa. In this project, channel 8 is reserved for critical packet transmission. In other words, non-critical packets can only be transmitted using channel 1 to 7. The modified channel plan is illustrated in figure 13.


Original

Uplink:

1. **868.1** - SF7BW125 to SF12BW125
2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
3. **868.5** - SF7BW125 to SF12BW125
4. **867.1** - SF7BW125 to SF12BW125
5. **867.3** - SF7BW125 to SF12BW125
6. **867.5** - SF7BW125 to SF12BW125
7. **867.7** - SF7BW125 to SF12BW125
8. **867.9** - SF7BW125 to SF12BW125
9. **868.8** - FSK

Downlink:

- Uplink channels 1-9 (RX1)
- **869.525** - SF9BW125 (RX2 downlink only)

Modified

Uplink:

1. **868.1** - SF7BW125 to SF12BW125
2. **868.3** - SF7BW125 to SF12BW125 and SF7BW250
3. **868.5** - SF7BW125 to SF12BW125
4. **867.1** - SF7BW125 to SF12BW125
5. **867.3** - SF7BW125 to SF12BW125
6. **867.5** - SF7BW125 to SF12BW125
7. **867.7** - SF7BW125 to SF12BW125
8. **867.9** - SF7BW125 to SF12BW125 Reserved for critical packet transmission
9. ~~**868.8** - FSK~~ Disabled

Downlink:

- Uplink channels 1-9 (RX1)
- **869.525** - SF9BW125 (RX2 downlink only)

Figure 13 Channel Plan with Separated Channel for Critcal Packets

4.2 LoRa Simulator

As mentioned in the methodology section, instead of rebuilding everything, developing the simulator upon the existing simulator (i.e. LoRaSim) gives a faster development process. LoRaSim simulates that the end devices send packet to the gateway. Note that node and base station are used instead of end device and gateway in the LoRaSim. In the rest of the project regarding the simulator, the terms node and base station (BS) are be used to follow the definition in the LoRaSim.

Since the aim of the project is to apply frequency hopping schemes and channel planning on LoRaWAN instead of merely LoRa, the code is modified to simulate the LoRaWAN transmission process. In the following, the modification needed is first discussed. After that, the classes of different objects are discussed. Then the algorithms of different functions are discussed.

4.2.1 Modification

This section lists out the problems of LoRaSim and the modifications needed.

1. LoRaSim does not provide acknowledgement function. In LoRaWAN standard, if a network server receives a confirmed packet, it needs to reply an acknowledgement packet to the sender of the confirmed packet to acknowledge the sender the success of receiving. As discussed, a failure in receiving the acknowledgement packet leads to retransmission of confirmed packet, which causes the network to be more congested. In this project, acknowledgement function is implemented. The PDR of acknowledgement packet is measured to evaluate the performance of network.

2. LoRaSim does not implement the transmission mechanism for different LoRaWAN classes. In this project, class A and class B modes are be implemented. The reason of not implementing class C is that class C devices are not commonly used, due to its high energy consumption, which is contradictory to IoT application requirement.

3. LoRaSim does not provide a functionality for simulating the worst-case scenario, which is simultaneously transmission of packets. It is added for the experiment 2.

4. The base station in LoRaSim does not contain a packet buffer. This function is added is added using the Queue library. According to the data sheet of SX1301 chip, the packet buffer is 1024 bytes. Since each downlink packet is 14 bytes in this project, the buffer can store at most 73 packets. When there are more than 73 packets send to the base station, buffer overflow occurs, and the data is lost.

5. The channel list is modified to follow the LoRaWAN standard. A function is provided for shuffling the channel list.

4.2.2 Classes

The program utilizes the object-oriented programming principle. There are several classes in the simulator, including myBS, myNode, myPacket and myACKPacket, providing initial values and implementation of behaviour for objects. The objects are created by instantiating the class. Below discusses the functionality of classes and the instance variable contained in different classes. The source code is provided in the appendix 1.

1. myBS

This is the class for base station. The functions in this class are basically used to place the base station at a particular position if there is a symmetry and the number of base station is small, or random position. This class contains an instance variable **id** to uniquely identify each BS object. It also contains an instance variable **position** for storing the position of the BS object.

2. myNode

This is the class for node. The functions in this class include placing the node, calculating the distance from each of the base station. class contains an instance variable **id** to uniquely identifying each node object. It also contains an instance variable **position** for storing the position of the node object, and **distance** for storing the distance to each base station. It contains **isReceiving** status which indicates if the receive window is opened/closed. It has **BSNode** for storing the BS that reply the ACK to that node. It also has a list of **packet** that are of type myPacket associated to each BS.

3. myPacket / myACKPacket

These are the classes for normal packet and acknowledgement packet. They contain the sequence number of packet **SeqNr**, which is a global variable. They also contain the physical layer setting such as packet size, spreading factor, coding rate, bandwidth, frequency and airtime **rectime** according to the experiment selected.

They have different packet size. ACK packets are set to have the minimum packet size, 14 Bytes. For normal packets, it is assumed to have size of 18 Bytes.

4.2.3 Functions

1. frequencyCollision, sfCollision, powerCollision, timingCollision

These are the functions for checking if two packets are collided. frequencyCollision function checks if two packets have the same frequency. sfCollision function checks if two packets have the same spreading factor. powerCollision function checks if two packets have similar power. timingCollision function checks if two packets arrive at similar time.

2. CheckCollisionAtBS, checkCollisionAtNode

These functions run the collision checking functions at BS and at Node, respectively.

3. Transmit

This function is used for transmitting a packet. Below shows the algorithm of the function.

1. For **Class A** devices **jump to step 6**. For **Class B** devices, **delay** for **BCN**.

2. **Open** the **ping slot**

3. Check if the beacon is collided.

4. **Close** the **ping slot**

5. **Delay** for **PNG**

6. **Delay** for each **transmission**

7. **Determine if the packet is confirmed packet.** Add it to isConfPac/notConfPac

N.B. Confirmed packet requires the BS to send a ACK. Unconfirmed packet does not require. User can change the ratio of confirmed packet using the command.

8. **Sending** process start.

 i) sent ++

 ii) packet sequence number ++

 iii) At each **BS**, **check** if there is **collision**. If not, **add the packet** to **packetsAtBS[i]**.

 iv) At each **Node**, **check** if the **receive window** of that node is opening. If yes, **check** if there is a **collision**. If not, **add the packet** to packetsAtNode[i].

9. **Delay** for **rectime** for the Node/BS to completely receive the packet

10. For received packet, **store** the **BS id** to **recvBSList[node.nodeid]** and remove the node from **packetsAtBS[i].**

11. Store the **received and confirmed** packet in confReceived[]

12. **Determine the BS that reply the ACK**

Logic:

Assuming the BS are connected, such that they know which BS receives the packet first. Only the nearest BS to the node will reply the ACK unless that BS does not receive the packet due to packet loss/collision. In this case, the second nearest BS reply the ACK.

Implementation:

The distance of each node to each BS needs to be known. It is sorted and stored in node.sortedDist [i]. Then start with **node.sortedDist [0]**, check if it is in the **recvBSList[node.nodeid].** If **yes**, it is the BS that reply the ACK, **set the node.BSNode** to be that BS. If not, check if **node.sortedDist [1]** is in the **recvBSList[node.nodeid].** Following this process, the BS that reply the ACK can be determined.

13. Put the received confirmed packet to the buffer of base station

14. **Delay** for **RxDelay1**

15. **Open** the **receive window for some time, here I set recvWindow = 5*rectime.**

N.B. In the specification, recvWindow must be at least rectime, the exact length can be set arbitrarily. Since the ack maybe queueing at the BS, if the recvWindow is too short, the receive window will be closed when the ack arrive the node. However, if it is too long, it reduces the network performance.

16. **Close** the **receive window**

17. **Delay** for **RxDelay2 + rectime.** If the node does not receive packets in second receive window, as a common practise for LoRaWAN.
18. For **Class A**, **jump** back to **step 6**.
For **Class B** devices, delay until the beacon period end if the transmission time is later than the next beacon period.

4. ReplyAck

It is for the base station to check if there is ACK need to be sent. If yes, it sends the ACK.
1. Delay for 100ms (checking interval)
2. Check if there is confirmed packet in the buffer. If it is, get that packet out from the queue.
3. Check if the buffer size is 0. If it is, that mean no other packet is in the queue. Wait for RxDelay1.
4. At each **Node**, **check** if the **receive window** of that node is opening. If yes, **check** if there is **collision**. If not, **add the bs** to packetsAtNode[i]. For the **node which sent the packet**, store it in **ackRecPackets[]**.
5. **Delay** for **rectime** for the Node/BS to completely receive the packet
6. Jump back to 1

5. main

Algorithm
1. **Get the arguments** and check if they are valid.
2. **Print out the setting information**.
3. **Initialize variables**
4. **Create myBS objects and myNode (class A/B) objects**
5. Set the **simulation environment process** of each node to be **transmit** function
6. **Start the simulation**
7. **Print out the simulation result**
8. **Store the result to the file**

4.3 Hardware Implementation

4.3.1 Hardware Setup

Figure 14 shows the hardware setup in this project.

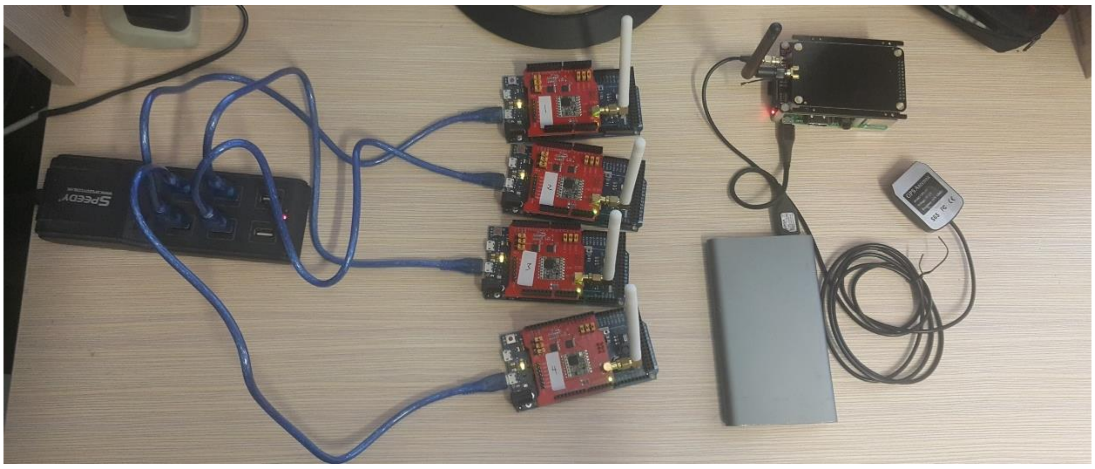


Figure 14 Hardware Setup

As discussed in section 3.2, there are one gateway and four end devices in the hardware setup. From the figure 14, it can be observed that four end devices on the left of the figure are connected to a USB hub for the power supply. The gateway on the right of the figure is separately connected to a USB battery pack as it requires larger and more stable power supply.

4.3.2 Setting up the RAK831-Raspberry Pi Gateway

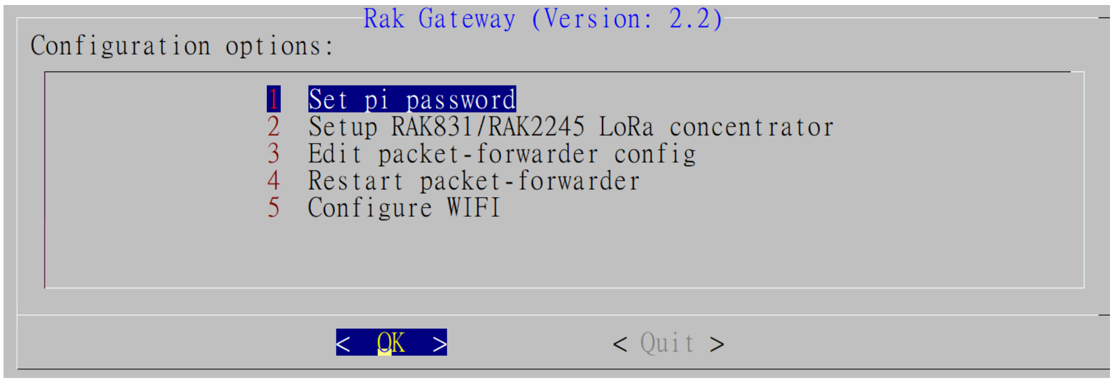
1. Install the operating system following the steps in https://github.com/RAKWireless/RAK2245-RAK831-LoRaGateway-RPi-Raspbian-OS

2. Enter the following command to login to the raspberry pi

```
$ ssh pi@192.168.0.189
```

3. Enter the following command to enter the configuration GUI

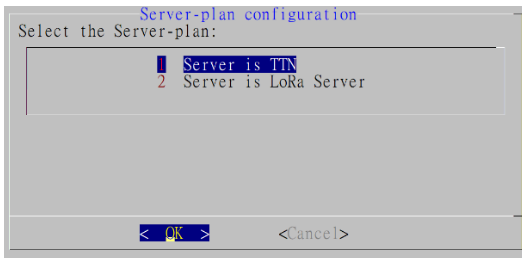
```
$ sudo gateway-config
```



```
Rak Gateway (Version: 2.2)
Configuration options:

    1  Set pi password
    2  Setup RAK831/RAK2245 LoRa concentrator
    3  Edit packet-forwarder config
    4  Restart packet-forwarder
    5  Configure WIFI

        <  OK  >           < Quit >
```

Figure 15 Gateway Configuration GUI

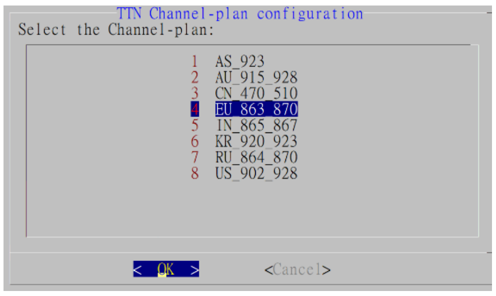
4. Select “2 Setup RAK831/RAK2245 LoRa concentrator”. The following appears.



```
Server-plan configuration
Select the Server-plan:

    1  Server is TTN
    2  Server is LoRa Server

        <  OK  >     <Cancel>
```

Figure 16 Server-plan Configuration

5. Select “1 Server is TTN” since we are using TTN as the server. The following appears.



```
TTN Channel-plan configuration
Select the Channel-plan:

    1  AS_923
    2  AU_915_928
    3  CN_470_510
    4  EU_863_870
    5  IN_865_867
    6  KR_920_923
    7  RU_864_870
    8  US_902_928

        <  OK  >     <Cancel>
```

Figure 17 TTN Channel-plan Configuration

6. Select “4 EU_863_870” since we are using EU868.

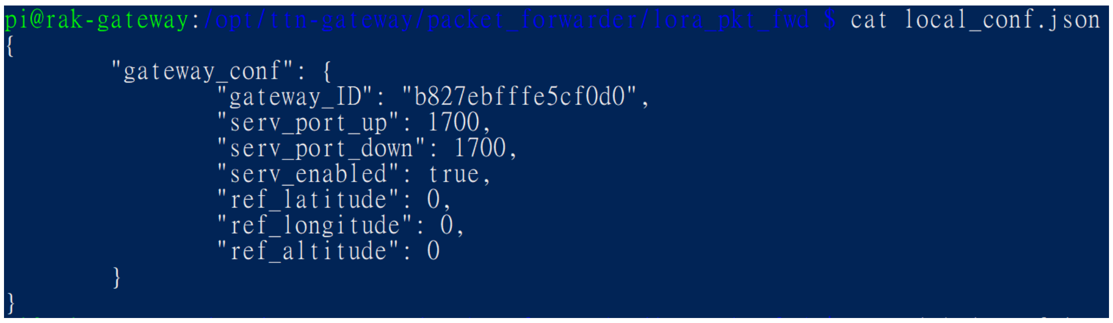
7. After that, the lora gateway and packet forwarder will restart, click enter if needed.

8. Use the following command to go to the folder that contains the configuration file

```
$ cd ../../opt/ttn-gateway/packet_forwarder/lora_pkt_fwd
```

9. Enter the following command to see the local configuration file.

```
$ cat local_conf.json
```

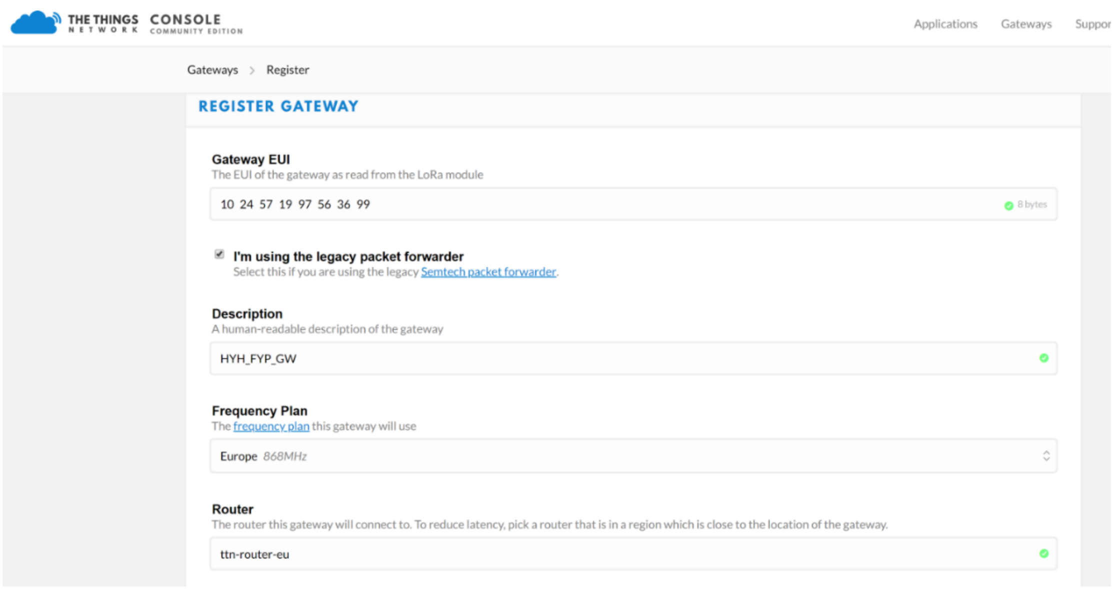


```
pi@rak-gateway:/opt/ttn-gateway/packet_forwarder/lora_pkt_fwd $ cat local_conf.json
{
        "gateway_conf": {
                "gateway_ID": "b827ebfffe5cf0d0",
                "serv_port_up": 1700,
                "serv_port_down": 1700,
                "serv_enabled": true,
                "ref_latitude": 0,
                "ref_longitude": 0,
                "ref_altitude": 0
        }
}
```

Figure 18 Local Configuration File in the Gateway

Mark down the gateway_ID. This is needed to register the gateway in TTN.

Go to TTN website to register an account and then register a gateway.



REGISTER GATEWAY

Gateway EUI
The EUI of the gateway as read from the LoRa module
10 24 57 19 97 56 36 99 — 8 bytes

✓ **I'm using the legacy packet forwarder**
Select this if you are using the legacy Semtech packet forwarder.

Description
A human-readable description of the gateway
HYH_FYP_GW

Frequency Plan
The frequency plan this gateway will use
Europe 868MHz

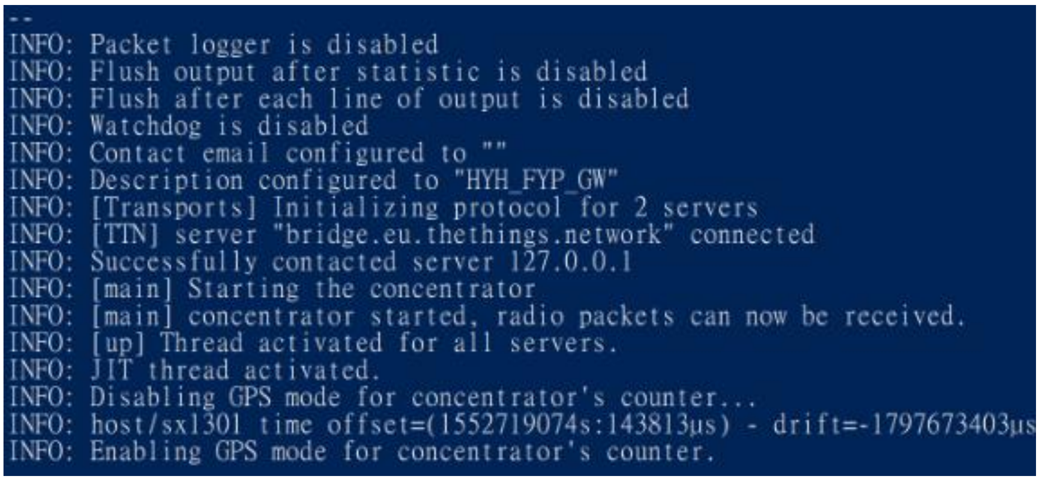
Router
The router this gateway will connect to. To reduce latency, pick a router that is in a region which is close to the location of the gateway.
ttn-router-eu

Figure 19 Registering Gateway in TTN

In the Gateway EUI, enter the gateway_ID, i.e. b827ebfffe5cf0d0. Click "I'm using the legacy packet forwarder". Select EU868 frequency plan.

10. Enter the following command to see the system log

```
$ sudo cat journalctl -f
```

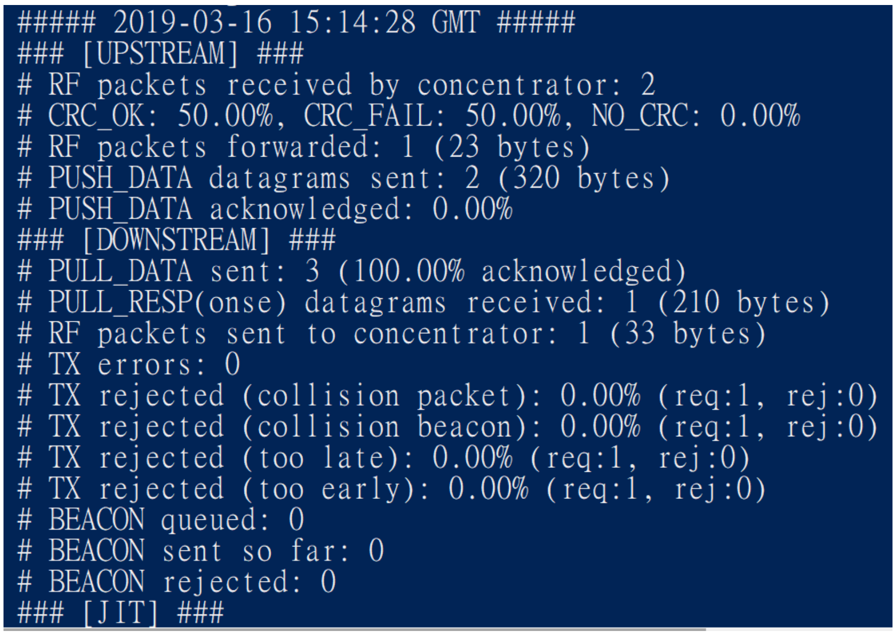


```
--
INFO: Packet logger is disabled
INFO: Flush output after statistic is disabled
INFO: Flush after each line of output is disabled
INFO: Watchdog is disabled
INFO: Contact email configured to ""
INFO: Description configured to "HYH_FYP_GW"
INFO: [Transports] Initializing protocol for 2 servers
INFO: [TTN] server "bridge.eu.thethings.network" connected
INFO: Successfully contacted server 127.0.0.1
INFO: [main] Starting the concentrator
INFO: [main] concentrator started, radio packets can now be received.
INFO: [up] Thread activated for all servers.
INFO: JIT thread activated.
INFO: Disabling GPS mode for concentrator's counter...
INFO: host/sx1301 time offset=(1552719074s:143813µs) - drift=-1797673403µs
INFO: Enabling GPS mode for concentrator's counter.
```

Figure 20 Verification of Gateway-TTN Network Server Connection

It can be observed that the gateway is successfully connected to TTN server. The concentrator is successfully started and thus it can receive RF packet.

11. Verify the functionality of packet forwarder

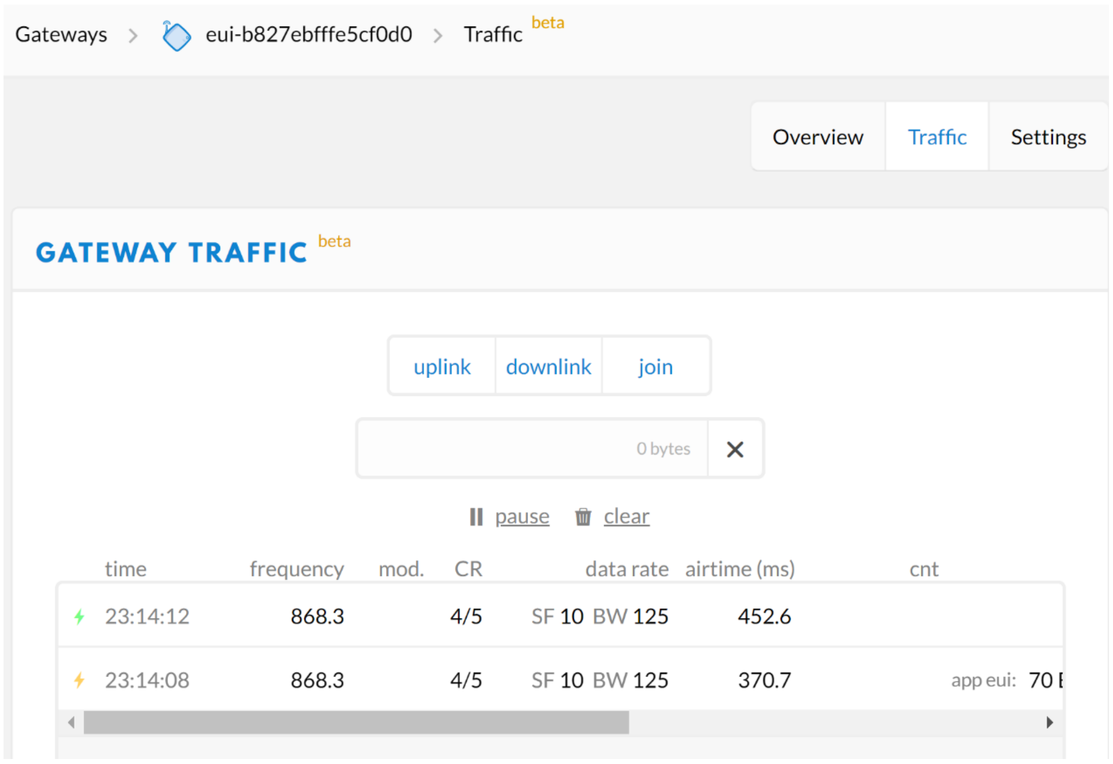


```
##### 2019-03-16 15:14:28 GMT #####
### [UPSTREAM] ###
# RF packets received by concentrator: 2
# CRC_OK: 50.00%, CRC_FAIL: 50.00%, NO_CRC: 0.00%
# RF packets forwarded: 1 (23 bytes)
# PUSH_DATA datagrams sent: 2 (320 bytes)
# PUSH_DATA acknowledged: 0.00%
### [DOWNSTREAM] ###
# PULL_DATA sent: 3 (100.00% acknowledged)
# PULL_RESP(onse) datagrams received: 1 (210 bytes)
# RF packets sent to concentrator: 1 (33 bytes)
# TX errors: 0
# TX rejected (collision packet): 0.00% (req:1, rej:0)
# TX rejected (collision beacon): 0.00% (req:1, rej:0)
# TX rejected (too late): 0.00% (req:1, rej:0)
# TX rejected (too early): 0.00% (req:1, rej:0)
# BEACON queued: 0
# BEACON sent so far: 0
# BEACON rejected: 0
### [JIT] ###
```

Figure 21 Verification of the Functionality of Packet Forwarder in Gateway

From the upstream traffic log, it can be observed that the gateway received two RF packets. However, there is only 1 packet successfully passed the CRC check. From the step 9 in the gateway configuration, we can see that the gateway only forward packet that pass the CRC check. Even though the gateway can be configured to send packets, it is not recommended since it will finally be rejected by the network server. Therefore, only one RF packet is forwarded.

From the downstream traffic log, it can be observed that there is one RF packet sent to concentrator. This is the reply sent from the network server.



time	frequency	mod.	CR	data rate	airtime (ms)	cnt
23:14:12	868.3		4/5	SF 10 BW 125	452.6	
23:14:08	868.3		4/5	SF 10 BW 125	370.7	app eui: 70

Figure 22 Verification of the Functionality of Packet Forwarder in TTN

From the gateway traffic log from the TTN website, two packets can be observed.

The bottom one with an orange flash logo is the join request packet received from the end device. The top one with a green flash logo is the join accept packet sent from the network server. Other information such as the frequency, coding rate, data rate, airtime and EUI information can be observed after the TTN server successfully decode the packet.

4.3.3 Setting Up the Packet Logger in the Gateway

1. Enter the following command to upgrade the system to the newest version and install pandas for reading and extracting information from csv files.

```
$ sudo apt-get update && sudo apt-get upgrade&& sudo apt-get install python-pandas
```

2. Enter the following command to modify the packet logger program

```
$ cd ../../opt/ttn-gateway/lora_gateway/util_pkt_logger/src && sudo nano util_pkt_logger.c
```

3. Enter the following command to recompile the packet logger program.

```
$ cd ..&& sudo make util_pkt_logger
```

4. Enter the following command to modify the gateway ID in the config file

```
$ sudo nano global_config
```

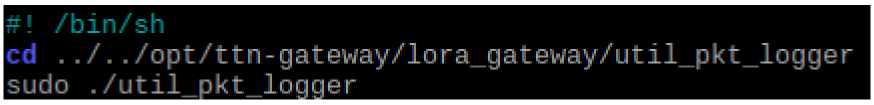
```
$ sudo nano local_config
```

5. Back to the default directory.

```
$ cd
```

Create scripts for running the packet logger and removing the csv files.

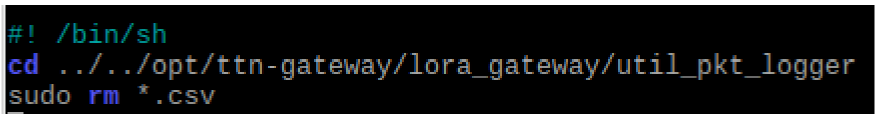
```
$ sudo nano run.sh
```



```
#! /bin/sh
cd ../../opt/ttn-gateway/lora_gateway/util_pkt_logger
sudo ./util_pkt_logger
```

Figure 23 Script for Running the Packet Logger

```
$ sudo nano del.sh
```

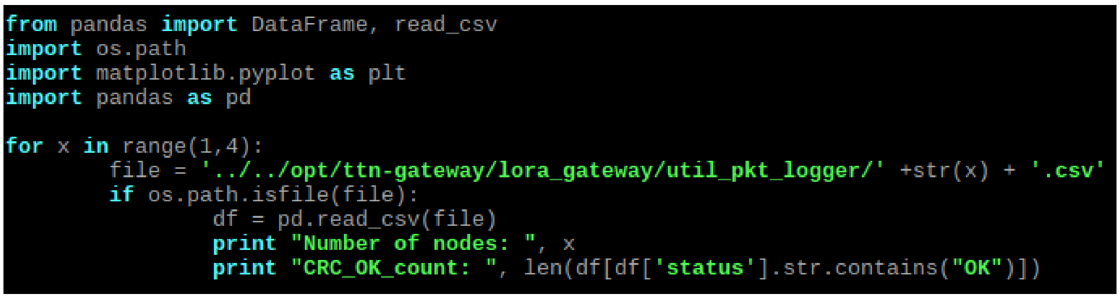


```
#! /bin/sh
cd ../../opt/ttn-gateway/lora_gateway/util_pkt_logger
sudo rm *.csv
```

Figure 24 Script for Deleting the Log File

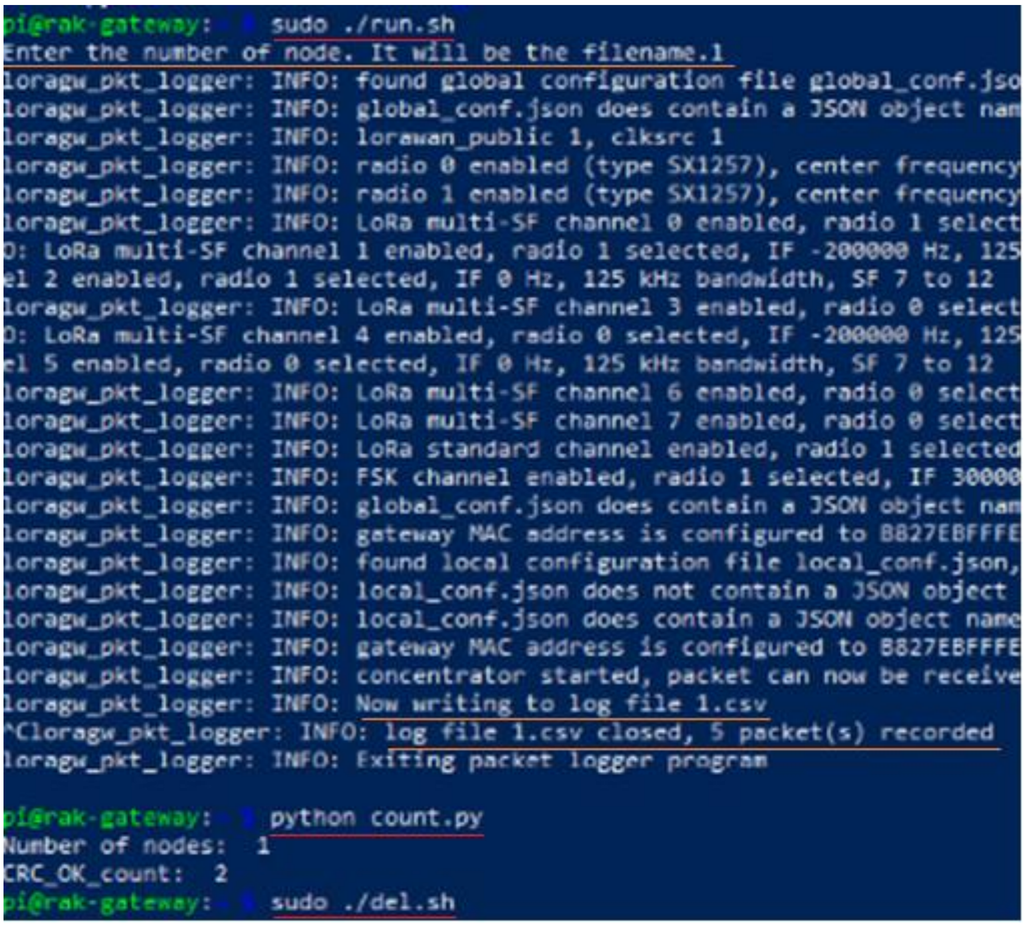
6. Create a python program to show the number of received packets passing the CRC for each number of nodes.

```
$ sudo nano count.py
```



```
from pandas import DataFrame, read_csv
import os.path
import matplotlib.pyplot as plt
import pandas as pd

for x in range(1,4):
        file = '../../opt/ttn-gateway/lora_gateway/util_pkt_logger/' +str(x) + '.csv'
        if os.path.isfile(file):
                df = pd.read_csv(file)
                print "Number of nodes: ", x
                print "CRC_OK_count: ", len(df[df['status'].str.contains("OK")])
```

Figure 25 Python Program for Counting the Number of CRC_OK

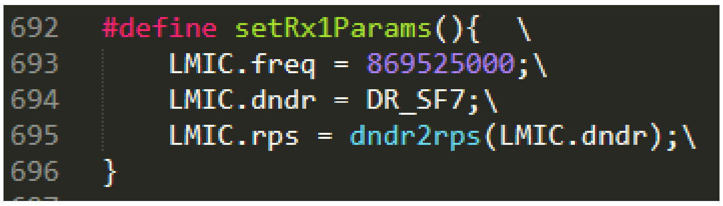


```
pi@rak-gateway:   $ sudo ./run.sh
Enter the number of node. It will be the filename.1
loragw_pkt_logger: INFO: found global configuration file global_conf.jso
loragw_pkt_logger: INFO: global_conf.json does contain a JSON object nam
loragw_pkt_logger: INFO: lorawan_public 1, clksrc 1
loragw_pkt_logger: INFO: radio 0 enabled (type SX1257), center frequency
loragw_pkt_logger: INFO: radio 1 enabled (type SX1257), center frequency
loragw_pkt_logger: INFO: LoRa multi-SF channel 0 enabled, radio 1 select
O: LoRa multi-SF channel 1 enabled, radio 1 selected, IF -200000 Hz, 125
el 2 enabled, radio 1 selected, IF 0 Hz, 125 kHz bandwidth, SF 7 to 12
loragw_pkt_logger: INFO: LoRa multi-SF channel 3 enabled, radio 0 select
O: LoRa multi-SF channel 4 enabled, radio 0 selected, IF -200000 Hz, 125
el 5 enabled, radio 0 selected, IF 0 Hz, 125 kHz bandwidth, SF 7 to 12
loragw_pkt_logger: INFO: LoRa multi-SF channel 6 enabled, radio 0 select
loragw_pkt_logger: INFO: LoRa multi-SF channel 7 enabled, radio 0 select
loragw_pkt_logger: INFO: LoRa standard channel enabled, radio 1 selected
loragw_pkt_logger: INFO: FSK channel enabled, radio 1 selected, IF 30000
loragw_pkt_logger: INFO: global_conf.json does contain a JSON object nam
loragw_pkt_logger: INFO: gateway MAC address is configured to B827EBFFFE
loragw_pkt_logger: INFO: found local configuration file local_conf.json,
loragw_pkt_logger: INFO: local_conf.json does not contain a JSON object
loragw_pkt_logger: INFO: local_conf.json does contain a JSON object name
loragw_pkt_logger: INFO: gateway MAC address is configured to B827EBFFFE
loragw_pkt_logger: INFO: concentrator started, packet can now be receive
loragw_pkt_logger: INFO: Now writing to log file 1.csv
^Cloragw_pkt_logger: INFO: log file 1.csv closed, 5 packet(s) recorded
loragw_pkt_logger: INFO: Exiting packet logger program

pi@rak-gateway:   $ python count.py
Number of nodes:  1
CRC_OK_count:  2
pi@rak-gateway:   $ sudo ./del.sh
```

Figure 26 Illustration of Packet Logger

4.3.4 Modification of Downlink Channel Frequency

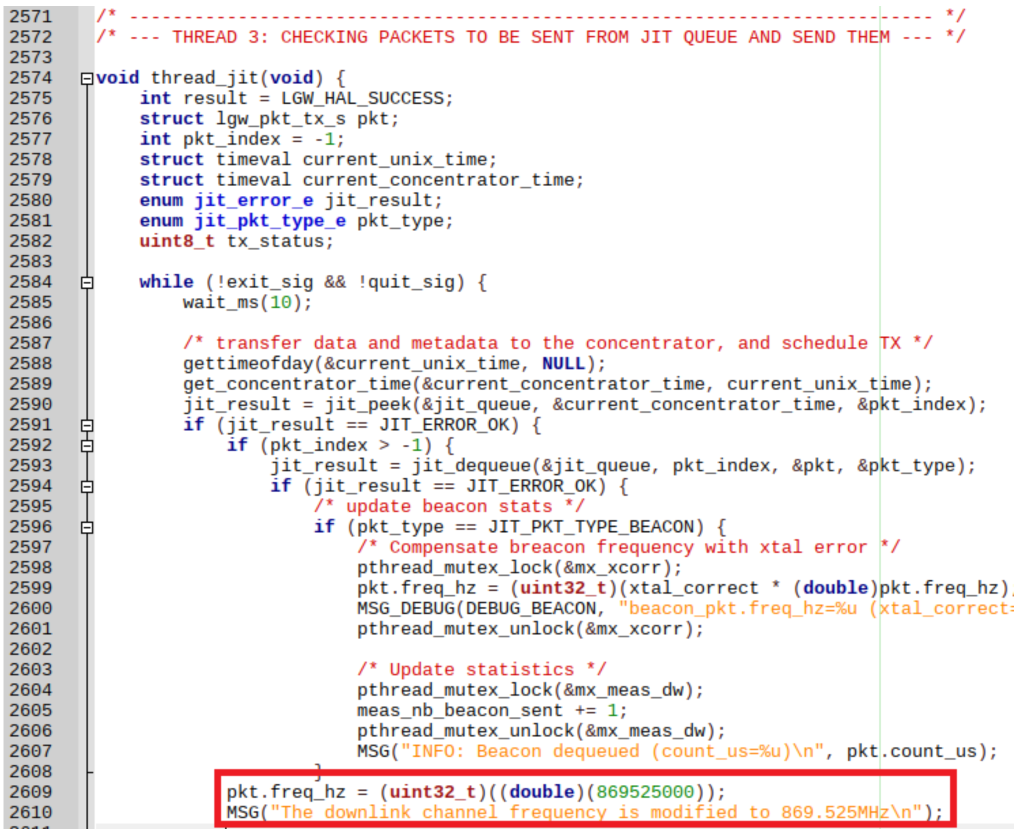
To receive downlink packets from the RX1 of nodes with a specific channel, the setRx1Params() function for EU868 nodes in lmic.c is modified as shown in figure 27. The frequency and the down link data rate of the downlink channel is set to be 869.525 MHz and DR_SF7 respectively.



```
#define setRx1Params(){  \
    LMIC.freq = 869525000;\
    LMIC.dndr = DR_SF7;\
    LMIC.rps = dndr2rps(LMIC.dndr);\
}
```

Figure 27 Modification of RX1 Frequency of End Devices

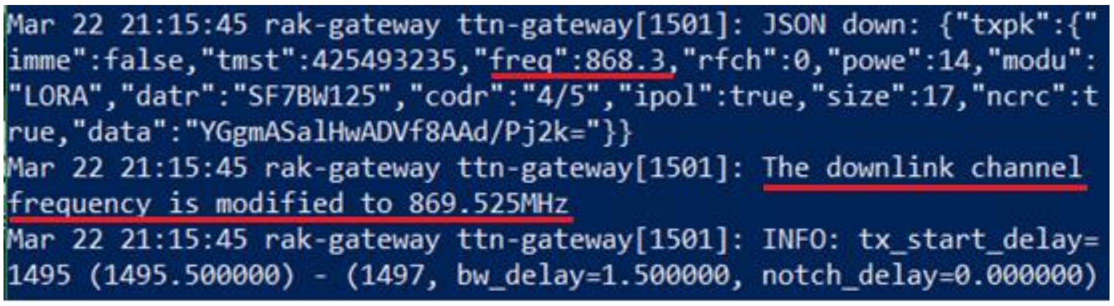
The source file of LoRa packet forwarder shows that there is a thread checking packets to be sent from jit queue and sending those queueing packets, as shown in figure 28. In line 2590, the program peeks into the jit queue to check if there is a queueing packet using the peek() function. If there is, the packet is dequeued and retrieved using the dequeue() function, as shown in line 2593. After retrieving the packet, under normal condition, the program forwards the packet to the concentrator for transmission. Before that, the packet frequency is modified to 869.525 MHz.



```c
/* -------------------------------------------------------------------------- */
/* --- THREAD 3: CHECKING PACKETS TO BE SENT FROM JIT QUEUE AND SEND THEM --- */

void thread_jit(void) {
    int result = LGW_HAL_SUCCESS;
    struct lgw_pkt_tx_s pkt;
    int pkt_index = -1;
    struct timeval current_unix_time;
    struct timeval current_concentrator_time;
    enum jit_error_e jit_result;
    enum jit_pkt_type_e pkt_type;
    uint8_t tx_status;

    while (!exit_sig && !quit_sig) {
        wait_ms(10);

        /* transfer data and metadata to the concentrator, and schedule TX */
        gettimeofday(&current_unix_time, NULL);
        get_concentrator_time(&current_concentrator_time, current_unix_time);
        jit_result = jit_peek(&jit_queue, &current_concentrator_time, &pkt_index);
        if (jit_result == JIT_ERROR_OK) {
            if (pkt_index > -1) {
                jit_result = jit_dequeue(&jit_queue, pkt_index, &pkt, &pkt_type);
                if (jit_result == JIT_ERROR_OK) {
                    /* update beacon stats */
                    if (pkt_type == JIT_PKT_TYPE_BEACON) {
                        /* Compensate breacon frequency with xtal error */
                        pthread_mutex_lock(&mx_xcorr);
                        pkt.freq_hz = (uint32_t)(xtal_correct * (double)pkt.freq_hz),
                        MSG_DEBUG(DEBUG_BEACON, "beacon_pkt.freq_hz=%u (xtal_correct=
                        pthread_mutex_unlock(&mx_xcorr);

                        /* Update statistics */
                        pthread_mutex_lock(&mx_meas_dw);
                        meas_nb_beacon_sent += 1;
                        pthread_mutex_unlock(&mx_meas_dw);
                        MSG("INFO: Beacon dequeued (count_us=%u)\n", pkt.count_us);
                    }
            pkt.freq_hz = (uint32_t)((double)(869525000));
            MSG("The downlink channel frequency is modified to 869.525MHz\n");
```

Figure 28 Modification of Downlink Frequency at Gateway

In the figure 29, it can be observed that the downlink packet received from the TTN that to be transmitted using 868.3MHz is modified to 869.525MHz.



```
Mar 22 21:15:45 rak-gateway ttn-gateway[1501]: JSON down: {"txpk":{"
imme":false,"tmst":425493235,"freq":868.3,"rfch":0,"powe":14,"modu":
"LORA","datr":"SF7BW125","codr":"4/5","ipol":true,"size":17,"ncrc":t
rue,"data":"YGgmASalHwADVf8AAd/Pj2k="}}
Mar 22 21:15:45 rak-gateway ttn-gateway[1501]: The downlink channel
frequency is modified to 869.525MHz
Mar 22 21:15:45 rak-gateway ttn-gateway[1501]: INFO: tx_start_delay=
1495 (1495.500000) - (1497, bw_delay=1.500000, notch_delay=0.000000)
```

Figure 29 Illustration of Modifying Downlink Channel Frequency

5. Results

5.1 The Effect of Using Different Number of Channels on the PDR of Uplink Packets

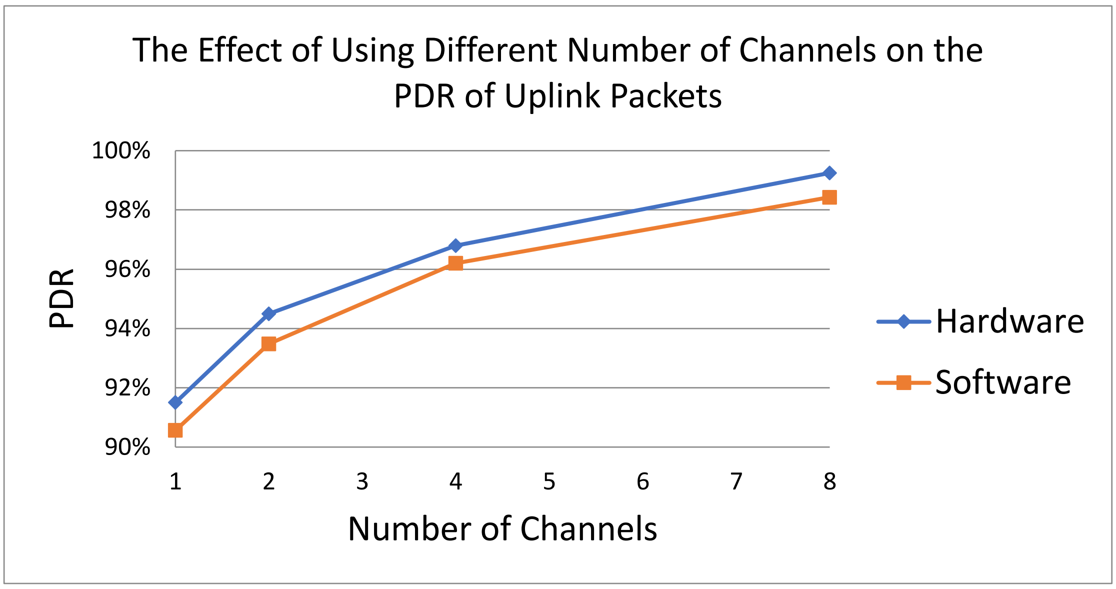


Figure 30 The Effect of Using Different Number of Channels on the PDR of Uplink Packets

Firstly, it can be observed that the increase in number of channels improves the PDR of uplink packets significantly. Specifically, from around 91% for a single channel to around 99% for eight channels.

Also, there exists 1% of discrepancy between the result collected from the hardware and the software. This can be due to the difference in the relative distance between the gateway and end devices in the software and in the hardware. End devices are placed randomly and may be as far as 100 meters from the gateway according to the default setting in LoRaSim, whereas end devices are placed at around 0.1 meter to the gateway in the hardware implementation. Therefore, the packets are more easily received by the gateway.

Despite the existence of discrepancy, the result demonstrates that the simulator can correctly simulate the network performance in a real LoRa network.

5.2 The Effect of Round Robin Scheduling with Non-sequential Channel on the PDR of Uplink Packets

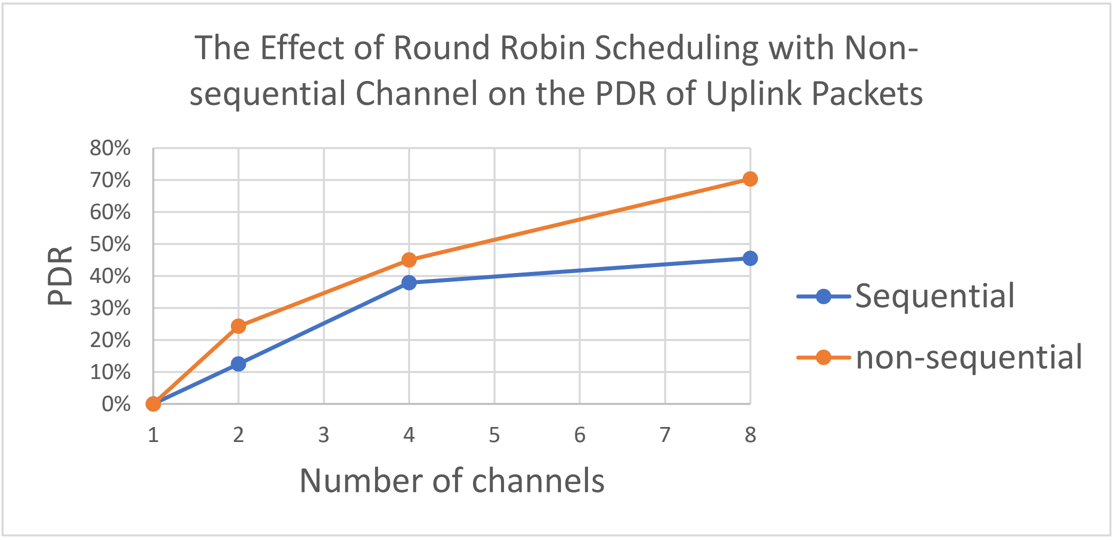


Figure 31 The Effect of Round Robin Scheduling with Non-sequential Channel on the PDR of Uplink Packets

It is obvious that when there is a single channel, the PDR of uplink packets is 0%. This is because a single channel guarantees that the collision happens if the end devices transmit simultaneously. When the number of channels increases, the PDR of uplink packets increases, regardless of whether sequential or non-sequential channel is used. This result is logical as larger number of channels reduce the probability of collision. It also confirms the result in experiment 1.

However, the PDR of uplink packets of using non-sequential channel is higher than that of sequential channel. This is as expected as using non-sequential channel reduces the probability of occurring forever-collision problem as explained in section 4.1.4. Therefore, the use of non-sequential channels improves the PDR of uplink packets in some applications like reporting sudden events.

5.3 The Effect of Using Separated Channel for Acknowledgement on the PDR of Acknowledgement Packets

Trial / Separated	1	2	3	4	5	Average packet loss	PDR in hardware	PDR in simulator
yes	3	8	11	11	5	7.6	98.1%	97.2%
no	23	21	21	20	24	21.8	94.6%	93.5%

Table 11 The Effect of Using Separated Channel for Acknowledgement on the PDR of Acknowledgement Packets

The use of separated channel for acknowledgement packets yields 3.5% and 3.7% improvement in the PDR of acknowledgement packets for the hardware experiment and the simulation respectively. The result is as expected since if separated acknowledgement channel is not used, the downlink packet arrives at the end device maybe collided with other uplink packet sent from other end devices.

Again, the result demonstrates that the simulator can correctly simulate the network performance in a real LoRa network.

5.4 The Effect of Using Separated Channel for Acknowledgement with Different Number of Nodes on the PDR of Acknowledgement Packets

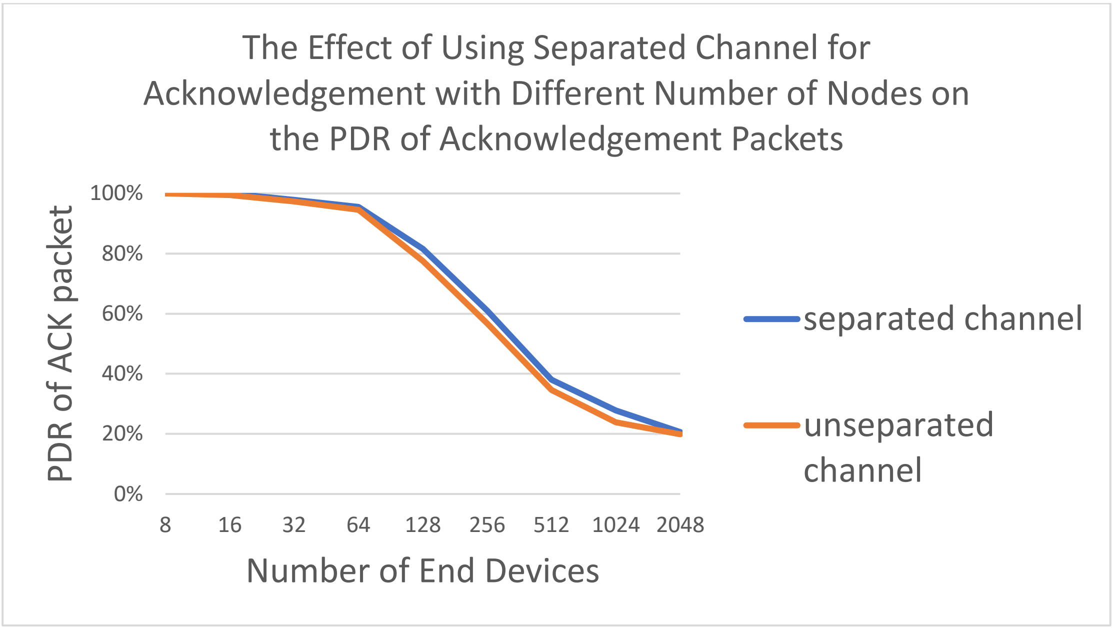


Figure 32 The Effect of Using Separated Channel for Acknowledgement with Different Number of Nodes on the PDR of Acknowledgement Packets

For both schemes, it can be observed that there is a drop in PDR with increasing number of end devices, from 99.5% for eight end devices to around 20% for 2048 end devices. This is because more end devices mean that there is higher probability for the acknowledgement packet to collide with packet sent from the end devices. Also, there will be a packet buffer overflow in the gateway when it receives too many packets from the network server.

Separated channel scheme improves the PDR of acknowledgement packets than unseparated channel scheme by around 5%. This is because allocating a specific channel for transmitting acknowledgement, the acknowledgement packets could not be collided with packets sent from other end devices. Therefore, the probability of collision is lowered and thus the PDR increases.

5.5 The Effect of Using Separated Channel for Acknowledgement with Different Number of Gateways on the PDR of Acknowledgement Packets

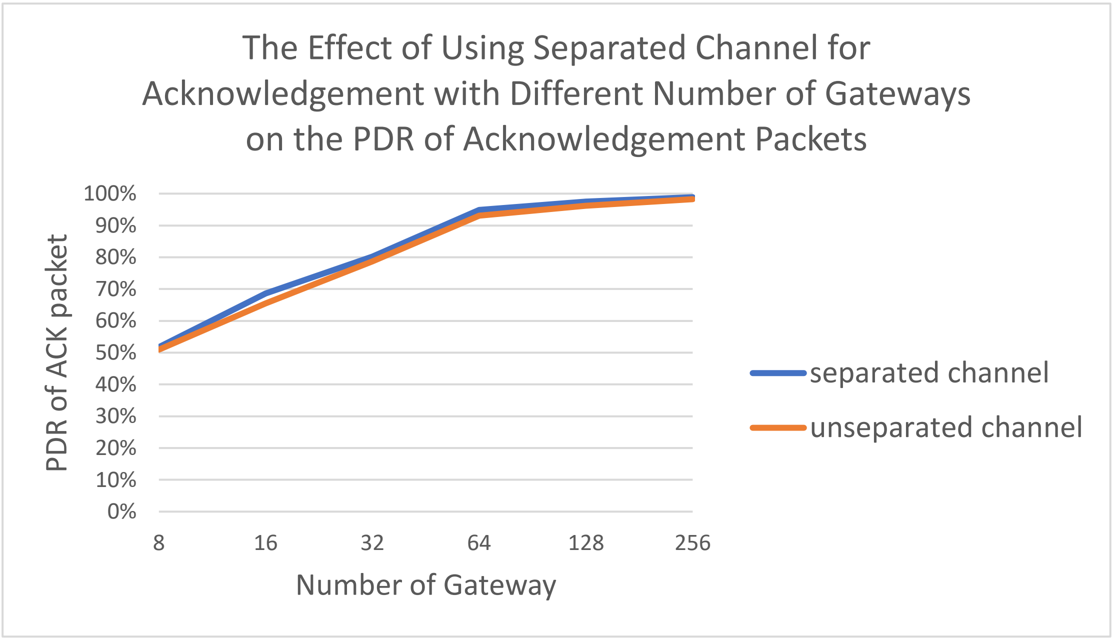


Figure 33 The Effect of Using Separated Channel for Acknowledgement with Different Number of Gateways on the PDR of Acknowledgement Packets

For both schemes, it can be observed that there is a rise in the PDR of acknowledgement packets with increasing number of gateways, from around 50% for 8 gateways to around 99% for 256 gateways. Given that the number of nodes is constant. This is because more base stations mean the packets are distributed to more base stations. The buffer overflow is less likely to occur, and this reduces the probability of acknowledgement packet loss.

Separated frequency scheme improves the PDR of acknowledgement packets than unseparated frequency scheme by around 3%. This is because allocating a specific channel for transmitting acknowledgement, the acknowledgement packets could not be collided with other packets sent from other end devices. Therefore, the probability of collision is lowered and thus the PDR increases.

5.6 The Effect of Using Separated Channel for Critical Packet on the PDR of Critical Packets and Non-critical Packets

Trial / Separated	1	2	3	4	5	Average critical packet loss	Number of critical packets	PDR of critical packet
yes	3	2	2	4	3	2.8	62	95.5%
no	8	8	7	8	7	7.6	62	87.7%

Table 12 The Effect of Using Separated Channel for Critical Packet on the PDR of Critical Packets

The use of separated channel for critical packet yields 7.8% improvement in the PDR of critical packets. The result is as expected since allocating a specific channel for transmitting critical packets, the critical packets cannot be collided with non-critical packets. The probability of collision is lowered and thus PDR increases.

Trial / Separated	1	2	3	4	5	Average non-critical packet loss	Number of non-critical packets	PDR of non-critical packet
yes	21	23	24	17	20	21	338	93.8%
no	13	13	11	14	12	12.6	338	96.3%

Table 13 The Effect of Using Separated Channel for Critical Packet on the PDR of Non-critical Packets

The use of separated channel for critical packet yields 2.5% decreases in the PDR of non-critical packets. The result is as expected since allocating a specific channel for transmitting critical packets would reduce the number of channels that can be used for transmitting non-critical packets. Non-critical packets are more likely to be collided with each other.

However, since the improvement of the PDR for critical packets outweighs the decrease in the PDR for non-critical packets. Using separated channel for critical packets is overall beneficial to the network performance.

6. Conclusion

In this project, the main objective is to evaluate the network performance of the LoRaWAN communication system when different channel access control protocols are applied. The project has fulfilled the two sub-objectives. The first part is the creation of LoRa simulator which used in the scalability analysis. The second part is to implement different frequency hopping schemes and channel planning schemes on Arduino platform. Both platforms were used in the evaluation of network performance.

Experimental results indicate that the objectives are fulfilled. The simulator developed can successfully simulate the LoRaWAN transmission process in the real world. The results showed that the increase in the number of frequency channel and the use of non-sequential channels improve network performance in term of packet delivery ratio (PDR). Moreover, the uses of separated channels for acknowledgement packet and critical packet transmission reduces the number of retransmissions and improves the PDR of critical packets, respectively.

The main contribution of this project is that it proves that applying frequency hopping schemes and allocating separated channels for acknowledgement packets and critical packets are beneficial to the overall network performance. As the LoRa technology is becoming more popular, more devices will be connected to the LoRaWAN network in the future. With the increasing number of end devices, the probability of collision also increases. Therefore, the channel access control protocols stated in this project is crucial to allow the LoRaWAN communication system to be more reliable and efficient.

However, there are some limitations in this project and exists room for improvement. Due to the limited time, MAC commands are not used for setting up the channels. More channel access control protocols can be tested in LoRaWAN, such as channel-aware frequency hopping or CSMA/CA. Sophisticated channel allocation scheme can be investigated such as distributing the channels to end devices evenly rather than randomly in this project. Also, the extended LoRa simulator based on LoRaSim can further be developed such as implementing class C device or sophisticated traffic model for testing. Due to the limited budget, the scale of the hardware setup is not sufficiently large. If budget allows, experiments can be performed with multiple gateways.

References

[1] M.O. Farooq and D. Pesch, “Evaluation of Multi-Gateway LoRaWAN with Different Data Traffic Models” in *IEEE 43rd Conference on Local Computer Networks*, 2018.

[2] M.O. Farooq and D. Pesch, “A Search into a Suitable Channel Access Control Protocol for LoRa-Based Networks” in *IEEE 43rd Conference on Local Computer Networks*, 2018.

[3] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui and T. Watteyne, "Understanding the Limits of LoRaWAN," in *IEEE Communications Magazine*, vol. 55, no. 9, pp. 34-40, Sept. 2017.

[4] Semtech, LoRa Modulation Basics (revision 2), 2015.

[5] LoRa Alliance, LoRaWAN 1.1 Specification, 2017.

[6] LoRa Alliance, https://lora-alliance.org/.

[7] E.F.Gorostiza, J. Berzosa, J. Mabe and R. Cortinas, “A Method for Dynamically Selecting the Best Frequency Hopping Technique in Industrial Wireless Sensor Network Applications” *Sensors*, Feb. 2018.

[8] A. Tsertou and D. I. Laurenson, "Revisiting the Hidden Terminal Problem in a CSMA/CA Wireless Network," in *IEEE Transactions on Mobile Computing*, vol. 7, no. 7, pp. 817-831, July 2008.

[9] C. Pham, "Investigating and experimenting CSMA channel access mechanisms for LoRa IoT networks," *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, Barcelona, 2018, pp. 1-6.

[10] Semtech, LoRa SX1276/77/78/79 data sheet.

Appendices

Appendix 1 Source Code of the Simulator

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
 LoRaSim 0.2.1: simulate collisions in LoRa
 Copyright © 2016 Thiemo Voigt <thiemo@sics.se> and Martin Bor
<m.bor@lancaster.ac.uk>

 This work is licensed under the Creative Commons Attribution
4.0
 International License. To view a copy of this license,
 visit http://creativecommons.org/licenses/by/4.0/.

 Do LoRa Low-Power Wide-Area Networks Scale? Martin Bor, Utz
Roedig, Thiemo Voigt
 and Juan Alonso, MSWiM '16,
http://dx.doi.org/10.1145/2988287.2989163

 $Date: 2017-05-12 19:16:16 +0100 (Fri, 12 May 2017) $
 $Revision: 334 $
"""
"""
 SYNOPSIS:
   ./loraDir.py <nodes> <avgsend> <experiment> <simtime>
[collision]
 DESCRIPTION:
     nodes
         number of nodes to simulate
     avgsend
         average sending interval in milliseconds
     experiment
         experiment is an integer that determines with what radio
settings the
         simulation is run. All nodes are configured with a fixed
transmit power
```

```
        and a single transmit frequency, unless stated otherwise.
        0   use the settings with the the slowest datarate (SF12, BW125, CR4/8).
        1   similair to experiment 0, but use a random choice of 3 transmit
            frequencies.
        2   use the settings with the fastest data rate (SF6, BW500, CR4/5).
        3   optimise the setting per node based on the distance to the gateway.
        4   use the settings as defined in LoRaWAN (SF12, BW125, CR4/5).
        5   similair to experiment 3, but also optimises the transmit power.
    simtime
        total running time in milliseconds
    collision
        set to 1 to enable the full collision check, 0 to use a simplified check.
        With the simplified check, two messages collide when they arrive at the
        same time, on the same frequency and spreading factor. The full collision
        check considers the 'capture effect', whereby a collision of one or the
 OUTPUT
    The result of every simulation run will be appended to a file named expX.dat,
    whereby X is the experiment number. The file contains a space separated table
    of values for nodes, collisions, transmissions and total energy spent. The
    data file can be easily plotted using e.g. gnuplot.
"""

import simpy
import random
import numpy as np
import math
import sys
import matplotlib.pyplot as plt
import os
import Queue

# turn on/off graphics
graphics = 0

# do the full collision check
full_collision = True

# show detail information
showDetail = True

# use separate frequency for ACK transmission
sepFreq = True

# shuffle the channel list
shuffleList = True

# set the smart meter model where the nodes send simultaneously
isSmartMeterModel = True

# list of timing constants
RxDelay1 = 1000
RxDelay2 = 1000
bcn = 2000
png = 2000
beaconPeriod = 128000

# packet buffer
maxBuffSize = 73 # Maximum number of packets in the buffer

# sensitivity related to spreading factor
```

```
sf7 = np.array([7, -126.5, -124.25, -120.75])
sf8 = np.array([8, -127.25, -126.75, -124.0])
sf9 = np.array([9, -131.25, -128.25, -127.5])
sf10 = np.array([10, -132.75, -130.25, -128.75])
sf11 = np.array([11, -134.5, -132.75, -128.75])
sf12 = np.array([12, -133.25, -132.25, -132.25])

# check for collisions at base station
# Note: called before a packet (or rather node) is inserted into the list


# Function for checking collision at Base Station
def checkCollisionAtBS(packet):
    col = 0
    if packet.lost:
        return 0
    if packetsAtBS[packet.bs]:
        for other in packetsAtBS[packet.bs]:
            if other.nodeid != packet.nodeid:
                if packet.freq == other.packet[packet.bs].freq \
                    and sfCollision(packet, other.packet[packet.bs]):
                    if full_collision:
                        if timingCollision(packet, other.packet[packet.bs]):
                            c = powerCollision(packet, other.packet[packet.bs])
                            for p in c:
                                p.collided = 1
                                if p == packet:
                                    col = 1
                        else:
                            pass
                    else:
                        packet.collided = 1
                        other.packet[
                                packet.
                                bs].collided = 1
                            col = 1
        return col
    return 0


def checkCollisionAtNode(packet, packetsAtNode):
    col = 0
    if packetsAtNode:
        for other in packetsAtNode:
            if other.nodeid != packet.nodeid:
                global beacon
                if (other != beacon):
                    packet2 = other.packet[0]
                else:
                    packet2 = beacon
                if frequencyCollision(packet, packet2) \
                    and sfCollision(packet, packet2):
                    if full_collision:
                        if timingCollision(packet, packet2):
                            c = powerCollision(packet, packet2)
                            for p in c:
                                p.collided = 1
                                if p == packet:
                                    col = 1
                        else:
                            pass
                    else:
                        packet.collided = 1
                        packet2.collided = 1
                        col = 1
        return col
    return 0

# frequencyCollision, conditions
#
```

```
#         |f1-f2| <= 120 kHz if f1 or f2 has bw 500
#         |f1-f2| <= 60 kHz if f1 or f2 has bw 250
#         |f1-f2| <= 30 kHz if f1 or f2 has bw 125
def frequencyCollision(p1, p2):
    if (abs(p1.freq - p2.freq) <= 120 and (p1.bw == 500 or
p2.freq == 500)):
        return True
    elif (abs(p1.freq - p2.freq) <= 60 and (p1.bw == 250 or
p2.freq == 250)):
        return True
    else:
        if (abs(p1.freq - p2.freq) <= 30):
            return True
    return False


def sfCollision(p1, p2):
    if p1.sf == p2.sf:
        return True
    return False


def powerCollision(p1, p2):
    powerThreshold = 6  # dB
    print "pwr: node {0.nodeid} {0.rssi:3.2f} dBm node
{1.nodeid} {1.rssi:3.2f} dBm; diff {2:3.2f} dBm".format(
        p1, p2, round(p1.rssi - p2.rssi, 2))
    if abs(p1.rssi - p2.rssi) < powerThreshold:
        if showDetail == True:
            print "collision pwr both node {} and node
{}".format(
                p1.nodeid, p2.nodeid)
        # packets are too close to each other, both collide
        # return both packets as casualties
        return (p1, p2)
    elif p1.rssi - p2.rssi < powerThreshold:
        # p2 overpowered p1, return p1 as casualty
```

```
            print "collision pwr node {} overpowered node 
{}".format(
                p2.nodeid, p1.nodeid)
            return (p1, )
        print "p1 wins, p2 lost"
        # p2 was the weaker packet, return it as a casualty
        return (p2, )

def timingCollision(p1, p2):
        # assuming p1 is the freshly arrived packet and this is the 
last check
        # we've already determined that p1 is a weak packet, so the 
only
        # way we can win is by being late enough (only the first n - 
5 preamble symbols overlap)

        # assuming 8 preamble symbols
        Npream = 8

        # we can lose at most (Npream - 5) * Tsym of our preamble
        Tpreamb = 2**p1.sf / (1.0 * p1.bw) * (Npream - 5)

        # check whether p2 ends in p1's critical section
        p2_end = p2.addTime + p2.rectime
        p1_cs = env.now + Tpreamb
        print "collision timing node {} ({},{},{}) node {} 
({},{})".format(
            p1.nodeid, env.now - env.now, p1_cs - env.now, 
p1.rectime, p2.nodeid,
            p2.addTime - env.now, p2_end - env.now)
        if p1_cs < p2_end:
            # p1 collided with p2 and lost
            print "not late enough"
            return True
        print "saved by the preamble"
        return False

```

```

# this function computes the airtime of a packet
# according to LoraDesignGuide_STD.pdf
#
def airtime(sf, cr, pl, bw):
    H = 0  # implicit header disabled (H=0) or not (H=1)
    DE = 0  # low data rate optimization enabled (=1) or not (=0)
    Npream = 8  # number of preamble symbol (12.25  from Utz paper)

    if bw == 125 and sf in [11, 12]:
        # low data rate optimization mandated for BW125 with SF11 and SF12
        DE = 1
    if sf == 6:
        # can only have implicit header with SF6
        H = 1

    Tsym = (2.0**sf) / bw
    Tpream = (Npream + 4.25) * Tsym
    if showDetail == True:
        print "sf", sf, " cr", cr, "pl", pl, "bw", bw
    payloadSymbNB = 8 + max(
        math.ceil((8.0 * pl - 4.0 * sf + 28 + 16 - 20 * H) /
                  (4.0 * (sf - 2 * DE))) * (cr + 4), 0)
    Tpayload = payloadSymbNB * Tsym
    return Tpream + Tpayload


# this function creates a BS
class myBS():
    def __init__(self, nodeid, packetlen):
        self.nodeid = nodeid
        self.x = 0
        self.y = 0
        self.packet = []
        # initialize the packet buffer
        global maxBuffSize
        self.packetBuffer = Queue.Queue(maxsize = maxBuffSize)


        # if the number of BS is equal to 1, 2, 3, 4, 6, 8, 24, the BS can ge located symmetrically to maximize the coverage of each BS
        # otherwise, randomly place the BS
        if(nrBS == 1 or nrBS == 2 or nrBS == 3 or nrBS == 4 or nrBS == 6 or nrBS == 8 or nrBS == 24):
            if (nrBS == 1 and self.id == 0):
                self.x = maxX/2.0
                self.y = maxY/2.0

            if (nrBS == 3 or nrBS == 2):
                self.x = (self.id+1)*maxX/float(nrBS+1)
                self.y = maxY/2.0

            if (nrBS == 4):
                if (self.id < 2):
                    self.x = (self.id+1)*maxX/3.0
                    self.y = maxY/3.0
                else:
                    self.x = (self.id+1-2)*maxX/3.0
                    self.y = 2*maxY/3.0

            if (nrBS == 6):
                if (self.id < 3):
                    self.x = (self.id+1)*maxX/4.0
                    self.y = maxY/3.0
                else:
                    self.x = (self.id+1-3)*maxX/4.0
                    self.y = 2*maxY/3.0

            if (nrBS == 8):
                if (self.id < 4):
                    self.x = (self.id+1)*maxX/5.0
                    self.y = maxY/3.0
                else:
                    self.x = (self.id+1-4)*maxX/5.0
                    self.y = 2*maxY/3.0

            if (nrBS == 24):
                if (self.id < 8):
                    self.x = (self.id+1)*maxX/9.0
                    self.y = maxY/4.0
                elif (self.id < 16):
                    self.x = (self.id+1-8)*maxX/9.0
                    self.y = 2*maxY/4.0
                else:
                    self.x = (self.id+1-16)*maxX/9.0
                    self.y = 3*maxY/4.0
        else:

            # this is very complex procedure for placing BS
            # and ensure minimum distance between each pair of BS
            found = 0
            rounds = 0
            global bs
            while (found == 0 and rounds < 100):
                a = random.random()
                b = random.random()
                if b < a:
                    a, b = b, a
                posx = b * maxDist * math.cos(2 * math.pi * a / b) + bsx
                posy = b * maxDist * math.sin(2 * math.pi * a / b) + bsy
                if len(bs) > 0:
                    for index, n in enumerate(bs):
                        dist = np.sqrt(((abs(n.x - posx))**2) + (
                            (abs(n.y - posy))**2))
                        if dist >= 10:
                            found = 1
                            self.x = posx
                            self.y = posy
                        else:
                            rounds = rounds + 1
                            if rounds == 100:
                                print "could not place new node, giving up"
                                exit(-1)
                else:
                    if showDetail == True:
                        print "first node"
                    self.x = posx
                    self.y = posy
                    found = 1

        if showDetail == True:
            print "BSx:", self.x, "BSy:", self.y

        self.packet.append(myACKPacket(self.nodeid, packetlen))
        global graphics
        if (graphics):
            global ax
            ax.add_artist(
                plt.Circle((self.x, self.y), 3, fill=True, color='green'))



# this function creates a node
class myNode():
    def __init__(self, nodeid, period, packetlen):
        global bs
        self.nodeid = nodeid
        self.period = period
```

```python
        self.x = 0
        self.y = 0
        self.packet = []
        self.dist = []
        self.BSNode = None

        # this is very complex prodecure for placing nodes
        # and ensure minimum distance between each pair of nodes
        found = 0
        rounds = 0
        global nodes
        while (found == 0 and rounds < 100):
            a = random.random()
            b = random.random()
            if b < a:
                a, b = b, a
            posx = b * maxDist * math.cos(2 * math.pi * a / b) + bsx
            posy = b * maxDist * math.sin(2 * math.pi * a / b) + bsy
            if len(nodes) > 0:
                for index, n in enumerate(nodes):
                    dist = np.sqrt(((abs(n.x - posx))**2) + (
                        (abs(n.y - posy))**2))
                    if dist >= 10:
                        found = 1
                        self.x = posx
                        self.y = posy
                    else:
                        rounds = rounds + 1
                        if rounds == 100:
                            print "could not place new node, giving up"
                            exit(-1)
            else:
                if showDetail == True:
                    print "first node"
                self.x = posx
                self.y = posy
                found = 1

        # create "virtual" packet for EACH BS, storing distance information
        global nrBS
        for i in range(0, nrBS):
            d = np.sqrt(
                (self.x - bs[i].x) * (self.x - bs[i].x) +
                (self.y - bs[i].y) * (self.y - bs[i].y))
            self.dist.append(d)
            self.packet.append(
                myPacket(self.nodeid, packetlen, self.dist[i], i))
        if showDetail == True:
            print('node %d' % nodeid, "x", self.x, "y", self.y, "dist: ",
                  self.dist)

        self.sortedDist = sorted(
            range(len(self.dist)), key=lambda k: self.dist[k])
        if showDetail == True:
            print self.sortedDist
        self.sent = 0

        # status = 0 -> not receving
        self.isReceiving = 0

        # graphics for node
        global graphics
        if (graphics == 1):
            global ax
            ax.add_artist(
                plt.Circle((self.x, self.y), 2, fill=True,
                           color='blue'))

```

```

# this function creates a ack packet (associated with a BS)


class myACKPacket():
    def __init__(self, nodeid, plen):
        global experiment
        global Ptx
        global gamma
        global d0
        global var
        global Lpld0
        global GL
        global minsensi

        self.nodeid = nodeid

        # randomize configuration values
        self.sf = random.randint(6, 12)
        self.cr = random.randint(1, 4)
        self.bw = random.choice([125, 250, 500])

        # for certain experiments override these
        if experiment == 1 or experiment == 0:
            self.sf = 12
            self.cr = 4
            self.bw = 125

        # for certain experiments override these
        if experiment == 2:
            self.sf = 6
            self.cr = 1
            self.bw = 500
        # lorawan
        if experiment == 4:
            self.sf = 12
            self.cr = 1
            self.bw = 125

        # for experiment 3 find the best setting
        # OBS, some hardcoded values
        Prx = Ptx  ## zero path loss by default

        if (experiment == 3) or (experiment == 5):
            minairtime = 9999
            minsf = 0
            minbw = 0

            print "Prx:", Prx

            for i in range(0, 6):
                for j in range(1, 4):
                    if (sensi[i, j] < Prx):
                        self.sf = int(sensi[i, 0])
                        if j == 1:
                            self.bw = 125
                        elif j == 2:
                            self.bw = 250
                        else:
                            self.bw = 500
                        at = airtime(self.sf, 1, plen, self.bw)
                        if at < minairtime:
                            minairtime = at
                            minsf = self.sf
                            minbw = self.bw
                            minsensi = sensi[i, j]
            if (minairtime == 9999):
                print "does not reach base station"
                exit(-1)
            print "best sf:", minsf, " best bw: ", minbw, "best airtime:", minairtime
            self.rectime = minairtime
            self.sf = minsf
            self.bw = minbw
```

```python
            self.cr = 1

            if experiment == 5:
                # reduce the txpower if there's room left
                Ptx.txpow = max(2, Ptx - math.floor(Prx - minsensi))
                Prx = Ptx - GL - Lpl
                if showDetail == True:
                    print 'minsesi {} best Ptx {}'.format(minsensi, Ptx)

        # transmission range, needs update XXX
        self.transRange = 150
        self.pl = plen
        self.symTime = (2.0**self.sf) / self.bw
        self.arriveTime = 0
        self.rssi = Prx
        # frequencies: lower bound + number of 61 Hz steps
        self.freq = 860000000 + random.randint(0, 2622950)

        # for certain experiments override these and
        # choose some random frequences
        if experiment == 1:
            global sepFreq
            if (sepFreq == True):
                self.freq = 869525000 # if a separated channel for ack is used, set it to be 869.525 MHz
            else:
                global freqList
                self.freq = random.choice(freqList)

        else:
            self.freq = 860000000


# this function creates a packet (associated with a node)
# it also sets all parameters, currently random
```

```
#
class myPacket():
    def __init__(self, nodeid, plen, distance, bs):
        global experiment
        global Ptx
        global gamma
        global d0
        global var
        global Lpld0
        global GL
        global minsensi

        # new: base station ID
        self.bs = bs
        self.nodeid = nodeid
        self.seqNr = 0

        # randomize configuration values
        self.sf = random.randint(6, 12)
        self.cr = random.randint(1, 4)
        self.bw = random.choice([125, 250, 500])

        # for certain experiments override these
        if experiment == 1 or experiment == 0:
            self.sf = 7
            self.cr = 4
            self.bw = 125

        # for certain experiments override these
        if experiment == 2:
            self.sf = 6
            self.cr = 1
            self.bw = 500
        # lorawan
        if experiment == 4:
            self.sf = 12
            self.cr = 1
            self.bw = 125

        # for experiment 3 find the best setting
        # OBS, some hardcoded values
        Prx = Ptx  ## zero path loss by default

        # log-shadow
        Lpl = Lpld0 + 10 * gamma * math.log10(distance / d0)
        if showDetail == True:
            print "Lpl:", Lpl
        Prx = Ptx - GL - Lpl

        if (experiment == 3) or (experiment == 5):
            minairtime = 9999
            minsf = 0
            minbw = 0

            print "Prx:", Prx

            for i in range(0, 6):
                for j in range(1, 4):
                    if (sensi[i, j] < Prx):
                        self.sf = int(sensi[i, 0])
                        if j == 1:
                            self.bw = 125
                        elif j == 2:
                            self.bw = 250
                        else:
                            self.bw = 500
                        at = airtime(self.sf, 1, plen, self.bw)
                        if at < minairtime:
                            minairtime = at
                            minsf = self.sf
                            minbw = self.bw
                            minsensi = sensi[i, j]
            if (minairtime == 9999):
                print "does not reach base station"
                exit(-1)
            if showDetail == True:
                print "best sf:", minsf, " best bw: ", minbw, "best airtime:", minairtime
            self.rectime = minairtime
            self.sf = minsf
            self.bw = minbw
            self.cr = 1

            if experiment == 5:
                # reduce the txpower if there's room left
                Ptx.txpow = max(2, Ptx - math.floor(Prx - minsensi))
                Prx = Ptx - GL - Lpl
                if showDetail == True:
                    print 'minsesi {} best Ptx {}'.format(minsensi, Ptx)

        # transmission range, needs update XXX
        self.transRange = 150
        self.pl = plen
        self.symTime = (2.0**self.sf) / self.bw
        self.arriveTime = 0
        self.rssi = Prx
        # frequencies: lower bound + number of 61 Hz steps
        self.freq = 860000000 + random.randint(0, 2622950)

        # for certain experiments override these and
        # choose some random frequences
        if experiment == 1:
            global freqList
            self.freq = random.choice(freqList)

        else:
            self.freq = 860000000

        self.rectime = airtime(self.sf, self.cr, self.pl, self.bw)
        if showDetail == True:
            print "frequency", self.freq, "symTime ", self.symTime
            print "bw", self.bw, "sf", self.sf, "cr", self.cr, "rssi", self.rssi
            print "rectime node ", self.nodeid, "  ", self.rectime
        # denote if packet is collided
        self.collided = 0
        self.processed = 0
        if experiment != 3:
            self.lost = self.rssi < minsensi
            if showDetail == True:
                print "node {} bs {} lost {}".format(self.nodeid, self.bs,
                                                                      self.lost)


def transmit(env, node):
    global nrNodes
    global confReceived
    global packetSeq
    global nrBS
    global freqList
    global isSmartMeterModel
    global freqAtBS
    if isSmartMeterModel == True:
        transmitDelay = node.period
    else:
        transmitDelay = random.expovariate(1.0 / float(node.period))

    if (classList[node.nodeid] == "B"):
        global bcn
        global png
        global beacon
        global beaconPeriod
        global beaconRec
        global beaconCol

        skipBCN = False
        reset = False
        if isSmartMeterModel == True:
            transmitDelay = node.period
        else:
            transmitDelay = random.expovariate(1.0 / float(node.period))

        timeDiff = 0

    while True:

        # randomize Freq of Packets
        for i in range(0, nrBS):
            global freqList
            global nodeFreqList
            global nodeChannelCounter
            newfreq = nodeFreqList[node.nodeid][nodeChannelCounter[node.nodeid]-1]
            if nodeChannelCounter[node.nodeid] < (len(nodeFreqList[node.nodeid])-1):
                nodeChannelCounter[node.nodeid] = nodeChannelCounter[node.nodeid] + 1
            else:
                nodeChannelCounter[node.nodeid] = 0
            node.packet[i].freq = newfreq

        localTimeBeforeBCN = env.now
        # Beacon window
        if (classList[node.nodeid] == "B" and skipBCN == False):

            yield env.timeout(bcn)

            # ping slot open
            node.isReceiving = 1
            if showDetail == True:
                print "Time: ", env.now, "node ", node.nodeid, "'s ping slot opening"

            pingSlot = node.packet[0].rectime
            yield env.timeout(pingSlot)
            # BS sending BEACON
            # Add the BS Node to Receiving nodes with collision checking

            for i in range(0, nrNodes):
                if (classList[node.nodeid] == "A"
                        and beacon not in packetsAtNode[i]):
                    packetsAtNode[i].append(beacon)

            if (beacon in packetsAtNode[node.nodeid]):
                print "ERROR: beacon already in"
            else:
                # adding packet if no collision

                if showDetail == True:
                    print "Time: ", env.now, "BS Sending Beacon ", "node ", node.nodeid, "is receiving"

                if (checkCollisionAtNode(beacon,
packetsAtNode[node.nodeid]) == 1):

                    if showDetail == True:
                        print "Time: ", env.now, "At node ", nodes[
                            node.nodeid].nodeid, ", BEACON Collided"
                    beaconCol = beaconCol + 1
                else:

                    packetsAtNode[node.nodeid].append(beacon)

                    if showDetail == True:
                        print "Node ", node.nodeid, " receives the BEACON"
                    beaconRec = beaconRec + 1

            if (beacon in packetsAtNode[node.nodeid]):
                if showDetail == True:
                    print "removed beacon from", node.nodeid
                packetsAtNode[node.nodeid].remove(beacon)

            for i in range(0, nrNodes):
                if (classList[node.nodeid] == "A"
                        and beacon in packetsAtNode[i]):
                    packetsAtNode[i].append(beacon)

            # close receive window
            node.isReceiving = 0
            if showDetail == True:
                print "Time: ", env.now, "node ", node.nodeid, "'s ping slot closed"

            yield env.timeout(png)

        classBcond = (classList[node.nodeid] == "B") and (timeDiff >=
                                                          transmitDelay)

        if (classList[node.nodeid] != "B"):
            yield env.timeout(transmitDelay)
        # transmit packet
        if (classList[node.nodeid] != "B" or classBcond):
            # determine if the packet is confirmed packet
            isConfirmedList[node.nodeid] = (random.random() < ratioConfirmed)
            global isConfPac
            global unConfPac
            if (isConfirmedList[node.nodeid] == True):
                isConfPac.append(packetSeq)
            elif (isConfirmedList[node.nodeid] == False):
                unConfPac.append(packetSeq)

            skipBCN = False
            reset = True
            # time sending and receiving
            # packet arrives -> add to base station
            if showDetail == True:
                print "Time: ",env.now, " Node ", node.nodeid, "Sending packet with number", packetSeq + 1, " with frequency ", newfreq
            node.sent = node.sent + 1

            packetSeq = packetSeq + 1

            # add to EACH BS
            for i in range(0, nrBS):
                freqAtBS[i].append(newfreq)

            yield env.timeout(10)


            for i in range(0, nrBS):
                if (node in packetsAtBS[i]):
                    if showDetail == True:
                        print "ERROR: packet already in BS ", i
                else:
                    # adding packet if no collision
                    if (freqAtBS[i].count(newfreq)>1 or checkCollisionAtBS(node.packet[i]) == 1):
                        node.packet[i].collided = 1
```

```python

                    else:
                        node.packet[i].collided = 0
                        packetsAtBS[i].append(node)
                        node.packet[i].addTime = env.now
                        node.packet[i].seqNr = packetSeq
            yield env.timeout(10)
            for i in range(0, nrBS):
                del freqAtBS[i][:]

# add to node with collision checking IF receive window is opening
            for nodeItem in range(0, nrNodes):
                if (nodes[nodeItem].nodeid !=
                        node.nodeid):  # receive by other nodes
                    if (node in packetsAtNode[nodeItem]):
                        if showDetail == True:
                            print "ERROR: packet already in Node", nodeItem
                    elif (nodes[nodeItem].isReceiving == 1):
                        # adding packet if no collision
                        if showDetail == True:
                            print "Time: ", env.now, "Node ", node.nodeid, "Sending packets  ", "node ", nodes[
                                nodeItem].nodeid, "is receiving"
                        if (checkCollisionAtNode(
                                node.packet[0], packetsAtNode[nodeItem]) == 1):
                            node.packet[0].collided = 1
                            if showDetail == True:
                                print "Time: ", env.now, "At node ", nodes[
                                    nodeItem].nodeid, ", Collided by  ", "node ", node.nodeid
                        else:
                            node.packet[0].collided = 0
                            packetsAtNode[nodeItem].append(node)
                            node.packet[0].addTime = env.now


            yield env.timeout(node.packet[0].rectime)

            for i in range(0, nrBS):
                if node.packet[i].lost:
                    lostPackets.append(node.packet[bs].seqNr)
                else:
                    if node.packet[i].collided == 0:
                        packetsRecBS[i].append(node.packet[i].seqNr)

                    else:
                        # XXX only for debugging
                        if (node.packet[i].seqNr not in collidedPackets):
                            collidedPackets.append(node.packet[i].seqNr)

            # complete packet has been received by base station
            # can remove it

            global recvBSList
            # reset the list
            recvBSList[node.nodeid] = []

            # decide a BS that sending reply
            node.BSNode = None
            for i in range(0, nrBS):
                if (node in packetsAtBS[i]):
                    # add the BS that received the packet from node
                    recvBSList[node.nodeid].append(i)
                    packetsAtBS[i].remove(node)
                    # reset the packet
                    node.packet[i].collided = 0
                    node.packet[i].processed = 0
            #global bs
            # if some BS received the packet from node, selecting a BS according to dist for transmission
            if (recvBSList[node.nodeid] != []):
                found = False
                i = 0
                while (found == False):
                    if (node.sortedDist[i] in recvBSList[node.nodeid]):
                        node.BSNode = bs[node.sortedDist[i]]
                        found = True
                        recPackets.append(node.packet[i].seqNr)

                        if showDetail == True:
                            print "Packet from node", node.nodeid, "with packet number ", packetSeq, "received by BS ", node.BSNode.nodeid
                        if (isConfirmedList[node.nodeid] == True):
                            confReceived.append(node.packet[i].seqNr)
                            if(node.BSNode.packetBuffer.qsize() < maxBuffSize):
                                node.BSNode.packetBuffer.put(node.packet[i], block=False)

                    else:
                        i = i + 1
            else:
                if showDetail == True:
                    print "Packet from node ", node.nodeid, " dropped"
            # complete packet has been received by node
            # can remove it
            for nodeItem in range(0, nrNodes):
                if (node in packetsAtNode[nodeItem]):
                    packetsAtNode[nodeItem].remove(node)

# Delay for RxDelay1, open receive window for rectime sec
            yield env.timeout(RxDelay1)
            # open receive window
            node.isReceiving = 1
            # reset the packetsAtNode
            packetsAtNode[node.nodeid] = []
            if showDetail == True:
                print "Time: ", env.now, "node ", node.nodeid, "'s receive window opening"
            yield env.timeout(node.packet[0].rectime*5)
            # close receive window
            node.isReceiving = 0
            if showDetail == True:
                print "Time: ", env.now, "node ", node.nodeid, "'s receive window closed"

            yield env.timeout(RxDelay2 + node.packet[0].rectime)


            if (classList[node.nodeid] == "B"):
                timeDiff = 0

                if isSmartMeterModel == True:
                    transmitDelay = node.period
                else:
                    transmitDelay = random.expovariate(1.0 / float(node.period))



        if (classList[node.nodeid] == "B"):
            localTimeAfterBCN = env.now
            beaconTimeDiff = localTimeAfterBCN - localTimeBeforeBCN
            timeDiff = timeDiff + beaconTimeDiff

            # this occur when transmitDelay<beaconTimeDiff after the transmission, reset the
            if (reset == True):
                timeDiff = 0
                reset = False

            if ((transmitDelay - timeDiff) < 0):
                timeDiff = transmitDelay

            if (((transmitDelay - timeDiff) <
                 (2 * beaconPeriod - beaconTimeDiff))
                    or ((timeDiff + beaconPeriod) > transmitDelay)):

                skipBCN = True
                yield env.timeout(transmitDelay - timeDiff)
                timeDiff = transmitDelay

            elif (beaconTimeDiff < beaconPeriod):
                timeDiff = timeDiff + beaconPeriod - beaconTimeDiff
                yield env.timeout(beaconPeriod - beaconTimeDiff)


def replyAck (env, bs):
    global nodes
    global packetsAtNode
    global showDetail
    global sepFreq
    global confReceived

    while True:
        # check buffer period
        yield env.timeout(100)
        buf = bs.packetBuffer
        if(buf.qsize() >0):
```

```python
            packet = buf.get(block=False)

            if(buf.qsize() == 0):
                yield env.timeout(RxDelay1)

            if (sepFreq == False):
                bs.packet[0].freq = packet.freq
            else:
                bs.packet[0].freq = 869525000
            print ("Frequency of this ACK packet is ", bs.packet[0].freq)
            for nodeItem in range(0, nrNodes):
                if(nodes[nodeItem].isReceiving == 1):
                    if (checkCollisionAtNode(packet,packetsAtNode[nodeItem]) != 1):
                        packetsAtNode[nodeItem].append(bs)
                        bs.packet[0].addTime = env.now
                        if showDetail == True:
                            print "Time: ", env.now, "Node ", nodeItem, "receives a ACK of node ", packet.nodeid
                        if(nodeItem==packet.nodeid):#
                            ackRecPackets.append(packetSeq)
                    else:
                        "ACKcollided"
            yield env.timeout(1000)
            for nodeItem in range(0, nrNodes):
                if (bs in packetsAtNode[nodeItem]):
                    packetsAtNode[nodeItem].remove(bs)


#
# "main" program
#

# get arguments
if len(sys.argv) >= 9:
    nrNodes = int(sys.argv[1])
    avgSendTime = int(sys.argv[2])
    experiment = int(sys.argv[3])
    simtime = int(sys.argv[4])
    nrBS = int(sys.argv[5])
    ratioConfirmed = float(sys.argv[6])
    nrClassA = int(sys.argv[7])
    nrFreq = int(sys.argv[8])
    if len(sys.argv) > 9:
        full_collision = bool(int(sys.argv[9]))
    print "================= Setting ================="
    print "Nodes:", nrNodes
    print "number of Class A nodes: ", nrClassA
    print "number of Class B nodes: ", nrNodes - nrClassA
    print "AvgSendTime (exp. distributed):", avgSendTime, "ms"
    print "Experiment: ", experiment
    print "Simtime: ", simtime, "ms"
    print "Number of BaseStation: ", nrBS
    print "Expected ratio of Confirmed Packet (require ACK from BaseStation): ", ratioConfirmed * 100, "%"
    print "Number of Frequency: ", nrFreq

    #print "Full Collision: ", full_collision
else:
    print "usage: ./loraDir <nodes> <avgsend> <experiment> <simtime> <basestation> <ratio of confirmed packet> <number of class A> <nRFreq>[collision]"
    #print "experiment 0 and 1 use 1 frequency only"
    exit(-1)
if (nrClassA > nrNodes):
    print "ERROR: nrClassA more than nrNodes"
    exit(-1)
elif (nrFreq < 1 or nrNodes < 1 or nrBS < 1):
    print "ERROR: nrNodes, nrBS, nrFreq must be larger or equal to 1"
    exit(-1)
elif (ratioConfirmed < 0 or ratioConfirmed > 1):
    print "ERROR: ratio of confirmed packet must be within 0 and 1"
    exit(-1)

# global stuff
nodes = []
packetsAtBS = []
packetsAtNode = []

# storing class info of nodes
classList = []

# Determine if the packet is confirmed or unconfirmed
isConfirmedList = []
isConfPac = []
unConfPac = []

# list of received packets
recPackets = []
collidedPackets = []
lostPackets = []

# storing the list of BS that receives the packet from a particular node
recvBSList = []

# storing received confirmed packet
confReceived = []

# storing list of ack received
ackRecPackets = []

# variable for Class B devices - beacon
beaconRec = 0
beaconCol = 0

# storing list of frequency in the frequency set
```

```
freqList = [868.1, 868.3, 868.5, 867.1, 867.3, 867.5, 867.7,
867.9]
tempFreqList = freqList[:nrFreq]
nodeFreqList = []
nodeChannelCounter = []

if shuffleList == True:
    for i in range (0, nrNodes):
        random.shuffle(tempFreqList)
        listn=tempFreqList[:]
        nodeFreqList.append(listn)
    for i in range (0, nrNodes):
        print(nodeFreqList[i])
else:
    for i in range (0, nrNodes):
        nodeFreqList.append(tempFreqList)
    for i in range (0, nrNodes):
        print(nodeFreqList[i])

for i in range (0, nrNodes):
    nodeChannelCounter.append(random.randint(0, nrFreq))


distInfoNode = []

env = simpy.Environment()


# maximum number of packets the BS can receive at the same time
maxBSReceives = 8

# max distance: 300m in city, 3000 m outside (5 km Utz
experiment)
# also more unit-disc like according to Utz
bsId = 1
nrCollisions = 0
nrReceived = 0
```

```
nrProcessed = 0
nrLost = 0

# global value of packet sequence numbers
packetSeq = 0

Ptx = 14
gamma = 2.08
d0 = 40.0
var = 0  # variance ignored for now
Lpld0 = 127.41
GL = 0

sensi = np.array([sf7, sf8, sf9, sf10, sf11, sf12])
## figure out the minimal sensitivity for the given experiment
minsensi = -200.0
if experiment in [0, 1, 4]:
    minsensi = sensi[5, 2]  # 5th row is SF12, 2nd column is BW125
elif experiment == 2:
    minsensi = -112.0  # no experiments, so value from datasheet
elif experiment == [3, 5]:
    minsensi = np.amin(
        sensi)  ## Experiment 3 can use any setting, so take minimum

Lpl = Ptx - minsensi
maxDist = d0 * (math.e**((Lpl - Lpld0) / (10.0 * gamma)))
if showDetail == True:
    print "amin", minsensi, "Lpl", Lpl
    print "maxDist:", maxDist

# base station placement
bsx = maxDist + 10
bsy = maxDist + 10
xmax = bsx + maxDist + 20
ymax = bsy + maxDist + 20
```

```

# maximum number of packets the BS can receive at the same time
maxBSReceives = 8

maxX = 2 * maxDist * math.sin(60 * (math.pi / 180))  # ==
sqrt(3) * maxDist

maxY = 2 * maxDist * math.sin(30 * (math.pi / 180))  # ==
maxdist

if showDetail == True:
    print "maxX ", maxX
    print "maxY", maxY

# prepare graphics and add sink
if (graphics == 1):
    plt.ion()
    plt.figure()
    ax = plt.gcf().gca()
    # XXX should be base station position
    #ax.add_artist(plt.Circle((bsx, bsy), 3, fill=True,
color='green'))
    ax.add_artist(plt.Circle((bsx, bsy), maxDist, fill=False,
color='green'))

    from matplotlib.patches import Patch
    from matplotlib.lines import Line2D

    legend_elements = [
        Line2D([0], [0],
               marker='o',
               color='w',
               label='BaseStation',
               markerfacecolor='g',
               markersize=15),
        Line2D([0], [0],
               marker='o',
```

```python
                     color='w',
                     label='Node',
                     markerfacecolor='b',
                     markersize=15)
    ]
    ax.legend(handles=legend_elements, loc='upper right')
# list of base stations
bs = []

# list of packets at each base station, init with 0 packets
packetsAtBS = []
packetsRecBS = []
freqAtBS = []
for i in range(0, nrBS):
    b = myBS(i, 14)
    bs.append(b)
    packetsAtBS.append([])
    packetsRecBS.append([])
    freqAtBS.append([])
    env.process(replyAck(env, b))
beacon = myACKPacket(999, 14)

for i in range(0, nrNodes):
    # myNode takes period (in ms), base station id packetlen (in
Bytes)
    # 1000000 = 16 min
    node = myNode(i, avgSendTime, 18)
    nodes.append(node)
    packetsAtNode.append([])
    distInfoNode.append(np.zeros((nrNodes, 2)))
    recvBSList.append([])
    isConfirmedList.append(True)

    if (i < nrClassA):
        classList.append("A")
    else:
        classList.append("B")
    env.process(transmit(env, node))


#prepare show
if (graphics == 1):
    plt.xlim([0, xmax])
    plt.ylim([0, ymax])
    plt.draw()
    plt.show()

# start simulation
env.run(until=simtime)


# compute energy
# Transmit consumption in mA from -2 to +17 dBm
TX = [
    22,
    22,
    22,
    23,  # RFO/PA0: -2..1
    24,
    24,
    24,
    25,
    25,
    25,
    25,
    26,
    31,
    32,
    34,
    35,
    44,  # PA_BOOST/PA1: 2..14
    82,
    85,
    90,  # PA_BOOST/PA1: 15..17
    105,
    115,
    125
]  # PA_BOOST/PA1+PA2: 18..20
# mA = 90    # current draw for TX = 17 dBm
energy = 0.0
mA = 90  # current draw for TX = 17 dBm
V = 3  # voltage XXX
sent = 0
for i in range(0, nrNodes):
    #    print "sent ", nodes[i].sent
    sent = sent + nodes[i].sent
    energy = (
        energy + nodes[i].packet[0].rectime * mA * V * nodes[i].sent) / 1000.0
print
print "================= Result ================="
print
print "number of packets sent: ", packetSeq
print "number of packets received by a BS: ", len(recPackets)
print "number of packets loss at BS: ", packetSeq - len(recPackets)
# data extraction rate
pdr = len(recPackets) / float(packetSeq)
print "PDR:", pdr
print
print "ratio of Confirmed Packet (require ACK from BaseStation): ", len(
    isConfPac) / float(packetSeq) * 100, "%"
print
print "number of confirmed packets sent: ", len(isConfPac)
print "number of confirmed packets received by a BS: ", len(confReceived)
print "number of confirmed packets loss: ", len(isConfPac) - len(confReceived)
print
print "number of unconfirmed packets sent: ", len(unConfPac)
print "number of unconfirmed packets received by a BS: ", len(
    recPackets) - len(confReceived)
print "number of unconfirmed packets loss: ", (packetSeq -
len(recPackets)) - (
    len(isConfPac) - len(confReceived))
print
print "number of beacon sent: ", beaconRec + beaconCol
print "number of beacon received at a classB node: ", beaconRec
print "number of beacon collided at a classB node: ", beaconCol
if ((beaconRec + beaconCol) != 0):
    beaconpdr = beaconRec / float(beaconRec + beaconCol)
else:
    beaconpdr = 0
print "Beacon PDR: ", beaconpdr
print
print "ACK sent: ", len(confReceived)
print "ACK received: ", len(ackRecPackets)
print "ACK collided/lost: ", len(confReceived) -
len(ackRecPackets)
if (len(confReceived) != 0):
    ackpdr = len(ackRecPackets) / float(len(confReceived))
else:
    ackpdr = 0
print "ACK PDR: ", ackpdr
print
print "energy (in mJ): ", energy
print

# this can be done to keep graphics visible
if (graphics == 1):
    raw_input('Press Enter to continue ...')

# store data in file
fname = "myData2.dat"
print fname
if os.path.isfile(fname):
    res = "\n" +  "{:.12f}".format(pdr)
```

```python

with open(fname, "a") as myfile:
    myfile.write(res)
myfile.close()
```