LoRa Failure Communication and Recovery with Silent Period

Aman Ganapathy Manvattira
amanvatt02@g.ucla.edu

Yuyang Chen
yuach@ucla.edu

Yinglei Fang
yf3575@g.ucla.edu

ABSTRACT

LoRa is a long range, low data rate, and low power wireless communication technology. Communications through LoRa take the form of uplink communications from client to the server and downlink communications from the server to the client. The downlink communications take place across two different time periods when the client is primed to receive data called "receive windows". Additionally, due to its simplistic design, it does not have a standardized failure handling mechanism.

Through calling the appropriate functions on the client end and ranging to identify the time intervals, we were able to enable the second receive window to receive data. Additionally, we developed the debug framework for a reset function to reset the client and server connection. We used this functionality to develop a "recovery mode" that we set up to be triggered in the two major failure cases we identified.

KEYWORDS

Receive window, client or device, server, gateway, uplink (ul), downlink (dl), message integrity code (MIC)

1 INTRODUCTION

1.1 Background

LoRa is a protocol standard that is meant for long range and low data rate communication. It is designed to be as simplistic as possible. This results in it not having standardized failure recovery mechanisms which means that if there is some data corruption on either the client or server end or timeout related issues with the transmitted packets, then this can cause the devices to crash and fail to recover.

LoRa controls the receival of downlink data through the use of two "receive windows". These are timings for the LoRa device to receive data from the server. In the traditional LoRa standard, the second receive window only opens if no data is received in the first receive window. As such, if no data is received in either receive window, then the device experiences a "silence period".

The architecture that we are working with is the Wingstack server, the sx1302_hal concentrator/gateway, and a Raspberry Pi which serves as the client utilizing the MCCI LoRaWAN library. The system as we first encountered it only had a working first receive window and no recovery mode mechanisms.

1.2 Goal

Here, we explore the development of a standardized failure recovery mechanism for LoRaWAN. To do so, we first had to contend with the lack of a second receive window in the architecture we used. Our goals are arranged below:

a) Enhance the architecture to accept data in multiple receive windows
b) Implement a "recovery mode" triggered by receive window under failure
c) Perform corresponding recovery action on the device side
d) Encode information about failure reasons on server side and client side.

2 ISSUES

2.1 Figuring out second receive window timing

Beyond just enabling it on the client end and making sure it only starts when it is supposed to, we also need to make sure the server is only sending the packets during the window. This involves accounting for the width of the receive window and making sure the entire packet can be transmitted within it. If we don't time it right, we will run into timeout issues. The receive window will time out on the client end.

2.2 Client side automatic decryption

The automatic decryption on the client end is called when processing a data frame. Because of this, the payload data will not be processed plainly. This means we need to figure out how to encrypt it appropriately on the server side in order to ensure that the payload data can be parsed on the client side. We could use this data to encode failure reasons.

Additionally, this issue expands out into making sure that after we use the appropriate cipher on the server end that we

compute the MIC correctly otherwise the packet will be rejected as an invalid downlink.

2.3 Recognizing when failure occurs

We need to signal to the client when there is a problem that necessitates entering recovery mode. We need to account for when the channel itself is faulty and cannot transmit the packet and when the error is due to something the server has parsed and figured out. The former requires identifying a way to communicate failure without the use of data packets.

2.4 What will the recovery mode be?

We need to identify a recovery mode that isn't too cumbersome for the client to trigger but will be able to deal with the majority of failures.

3 PROPOSED APPROACH

Firstly, we need to initialize the second receive window. We should, based on the LoRa specification, make sure it is only triggered when the first receive window doesn't receive any data.

We also need to give the server the ability to send not just ACKs, but also data. So we need to construct downlink data frames and send them back to the client. This would involve setting the appropriate headers and options and calculating the correct MIC. Noting that the LoRa documents specify that the client must perform decryption of the data it receives, we need to figure out how to appropriately encrypt the data on the server end and to handle it on the client end. Based on the aforementioned documents, we will use the AES_CCM cipher to encrypt data on the server end.

A good start to a "recovery mode" is to try a retry & reset function. If it is experiencing issues with the network, it should try resetting the device and rejoining afterwards. We can separate our resets into soft and hard resets. A soft reset is a scheduled task in which the LMIC library removes its information of the current connection and rejoins via a new join procedure. A hard reset involves the firmware program forcing a full reset of the LMIC status (including the queue) and optionally initializing first-time join procedure. We can therefore create a multi level retry and reset system to deal with errors of different severity.

We can implement a "failure detection" algorithm using the receive windows instead of passing any data through it. This would entail us taking advantage of the fact that the second receive window only opens when the first ends. As such, we will create expectations for the client to receive at least an ACK in one of the receive windows. That way, if it gets nothing, that is one indication of a failure. This also deals with communication breakdowns between the two pieces.

We also have to account for the fact that there can be other reasons for failure. Suppose the client sends data which is received by the server but is invalid. This can be detected by evaluating the mic of the dataframe. We need a way for the server to tell the client that there is an issue with the communications. So, we have to develop a way for the server to tell the client to initiate recovery mode. We can do this by having the server send a specially configured packet over downlink to indicate to the client that it is time to enter recovery mode.

Another consequence is that that there can be multiple reasons for failures. To resolve this, we can create a parsing tree to parse the reset packet from the server to figure out what kind of mechanism will need to be executed on the client side.

4 ENCOUNTERED ROADBLOCKS

4.1 Device and server environment

The server was not expected to send data back in the downlink. As such, we had to develop the capacity for it to send downlink packets. This involved constructing downlink dataframes, calculating the headers, and building functionality to model the MICs. We based this off the ACK code and adjusted it accordingly.

As for the client, the LMIC C library provided contains a set of basic debug print-out infrastructure. It provides basic functionality such as print-out of information and variables. In order for us to implement advanced functionalities such as Reset, we concluded that an interactive debug infrastructure is needed. We have implemented such debug infrastructure which is described in section 5.

Additionally, the server and the client kept track of time using different units and clocks. We had to figure out the units for both of them and figure out how to translate across them. This we did by performing tests with smaller increments on the server and client and comparing the timestamps across them.

4.2 Communication timing

LoRaWan Class-A communication uses timer-based receive windows to facilitate client-server communication. This is problematic because the client and server run their own clocks for communication - and the clocks can be out of sync.

Combining both the LoRaWAN specification and the default configuration of both client and server libraries, we have concluded that, after 1 second of the uplink packet, the client opens receive window 1 (Rx1) and listens for a downlink packet. If it does not receive a packet during that time, it would open a second receive window (Rx2) and attempt to receive another downstream packet.

The original implementation includes a standard uplink transmission procedure followed by a Rx1 ACK downlink packet transmission. We were able to verify the function of this implementation, but encountered the roadblock when we tried to enable Rx2 downlink packet transmission. The server sends the Rx1 packet at exactly 1 second of delay without problem, but if we tried to send the Rx2 packet at the 2nd second, the client wasn't able to receive it.

In order to solve this problem, we test various amounts of delay timing on the Rx1 downlink packet to determine the time-domain position, size and resolution (of changes allowed) of Rx1 as observed by the server. By testing the lower-bound and upper-bound of successful transmissions of Rx1, we have determined that Rx1 happens at 1000 milliseconds after receiving the uplink packet in server time, and the window is approximately 3.6 milliseconds.

To enable the second receive window, we used the client's debug printout of its radio-open time, and found out that if the first window is 3.6 milliseconds, the second window is approximately 2.1 milliseconds. We used this to test delay timing on the Rx2 downlink packet with a resolution of 1 milliseconds. Due to the fact that Rx1's window opens earlier than the expected 1000 ms, we tested to see if the Rx2 window opened and closed earlier than expected. We have found that the Rx2 window is at about 1996 to 1998 milliseconds as observed by the server, where the Rx2 downlink packet sent during that time-window can be successfully received by the client.

4.3 Inability to parse payload of downlink packets

We need to send data along with the downlink packet. For the purpose of this project, we wanted to use this payload data to specify encoding of failure reasons. The issue is that the client performs a decryption of the frame payload (frmPayload) of the downlink packet. Consequently, just sending the payload plain from the server end would not work; the payload would be rendered as an encrypted message on the client end. By default, LoRaWAN's data transmission uses AES encryption using application-unique identifiers. In its default configuration, the client was able to encrypt the packet in a way that is interpretable by the server, but the opposite wasn't true as the server is not programmed to send data to the client whatsoever, so there was no existing encryption functionality to work with.

We attempted to extend the DataFrame class on the server end to add encryption functionality. We successfully managed to encrypt the payload using AES CCM. But, due to a lack of documentation and various nuances that surround AES encryption, we were not fully able to decrypt the data on the client end. Consequently, we have chosen to use the decryption-stable contents of the packet to trigger client-side actions instead. Based on the different circumstances under which these functions are called, the reason for the failure is decoded. While this is not a perfect way of encoding failure reasons, it is at least able to separate it into two categories. This is explained further in section 5.

4.4 Switching dl packet to MAC command mode triggers erroneous parsing on client end

Based on the LoRaWAN specifications, if the fPort field of the dl packet header is set to 0x00, the dl packet is interpreted as a MAC command. Our initial idea was to exploit this when creating our server side reset packet. We could use a MAC command to indicate to the client that there were failures and to enter recovery mode.

However, we found that when the client received a MAC command, it triggered a MAC Command Parser on the client end. Since we struggled with the encryption and decryption of payload as mentioned in the previous roadblock, we decided to work around this and create a different way to indicate failures and for the server to tell the client to enter recovery mode.

We decided to use the fPort field, but to reserve different appdata ports (i.e port 1 onwards) for recovery notification. In our current implementation, port 0x01 is used to indicate intent to enter recovery mode. We then rebuilt the reset packet sending mechanism to use this port, and we engineered the parser on the client side to check to see the value of the fPort field and if it is 0x01 to enter recovery mode. As such, we worked around the roadblock and managed to get a working server side recovery mode initiation mechanism.

5 CURRENT STATUS OF IMPLEMENTATION

The current state of our implementation consists of a server and a client. On the server side, there is the wingstack server and the LoRa gateway, and on the client side, the firmware program and the LMIC library.

5.1 Server Status

We made adjustments to the WingStack server. Firstly, we implemented sending downlink data in the handle_data_up() function. This way, we can echo data we get initially from the client and send that as downlink data. We implemented parameters RWND1_START and RWND2_START which mark the start of the first and second receive windows. These parameters are now usable across any kind of downlink data which allows the server to send data across any receive window.To actually send a downlink packet, our modified handle_data_up has the downlink data call within it.

Additionally, we modified the DataFrame class on the server to give it encryption and mic calculation functionality. We wrote the encrypted() and encrypted_without_mic() functions which encrypt the payload of the dataframe and return it encrypted in AES_CCM. It uses the AppSessionKey to encrypt it but this is configurable if the user wants to use the network

session key instead. We also handled two ways of calculating the mic of the encrypted data: we wrapped a call to the Device Sessions mic calculation with the encrypted data but also gave the dataframe the ability to calculate its own MIC given the right parameters.

We also gave the server the ability to trigger recovery mode on the client end. We wrote functions such as gen_reset_dl() which will generate a reset packet. When this packet is received on the client end it will trigger recovery mode which as of right now triggers a reset and rejoin. This is implemented using the fPort part of the LoRa packet.

In summary, with the server, we gave it the ability to send downlink data on both receive windows using the echo_frame functionality we built and the receive window parameters we configured. Additionally, we gave it the ability to trigger recovery mode on the client with the reset_packet functionality. We also gave it the ability to encrypt data on its end and send a valid dl packet with the encrypted data.

5.2 Client side - Firmware Program

The firmware program is an arduino program (.ino) that serves as the main logic of the client - initializing LoRa library, initializing transmission, and handling received data. It consists of an initialization sequence, event handler, main logic, and various helper programs. We have implemented a debug console with a command line interface via Serial COM, and have made various changes to the existing parts as well.

The original initialization sequence initializes the LoRa LMIC library, and immediately starts attempting transmission to the server. Since we enabled a console interface, we transferred the duty of initializing transmission to the console instead.

In addition, we have also enhanced the event handler to handle different kinds of data from the server. Since we were able to access the header, we have implemented handling server-side requests via parsing for ACK bit, size of the payload, and the port number. Specifically, if the packet we have received is on port 1 (port 0 is used by the LMIC library for MAC commands), contains 0 bytes of data, and is an ACK, the client will initialize the server-requested-recovery mechanism. Additionally, if the server fails to receive messages twice, it will automatically enter recovery mode.

For the main logic, we have implemented the Serial COM debug interface alongside the call to LMIC library's main logic. The Serial COM debug interface allows a user to send various commands to the device such as "reset", "pause", "resume" and "dump". These commands' logic are pre-defined in our debug parse tree. The commands' names and functions are arbitrary, and can be easily changed for a variety of debug tasks.

Lastly, we have added various helper functions to facilitate both console commands and server-side requests. With these enhancements, a debug framework is built to work with both client-side console commands and server-side special packets.

5.3 Client - LMIC library

The LMIC library consists of various underlying functions that handle lower-level logic related to the physical LoRa radio. In order to enable Rx2 and the proper handling of downstream packets, we have modified how the server handles sending packets and receiving packets.

In terms of sending packets, the server needs to keep track of how many ACKs they expect, and that is handled during sending of the upstream packet. We have implemented a way to handle ACKs, so that the server can correctly handle ACKs during receiving.

On the other hand, for receiving, we have enabled the device to open Rx2 if no data was received during Rx1, as per the LoRa specification. We have ensured the data is parsed and returned correctly via the decodeFrame function.

Additionally, to facilitate debugging and feeding information back to the firmware logic, we have created a way to set error codes to notify the firmware about it. This function exists as a framework and is not actively used, because our current firmware program implementation has access to enough library-level information to make decisions and act, without the need of error-code logic.

5.4 Does our system work?

Our system is indeed capable of handling data across both receive windows. Additionally, we have implemented a recovery mode, maintained by the "triggerRecovery()" function. As of right now, the recovery mode performs a hard reset as we have found that it generally handles a wide variety of errors. The hard reset is a router level reset. We have implemented two main cases in which recovery mode is triggered: when no data is received across both receive windows and when the server itself initiates it. The idea is that if no data is received across both receive windows, then there must be an issue with the channel because otherwise it should receive at least an ACK.

Where the system struggles is in parsing the data in the payload. Whilst the server side encryption works according to specifications, we were not able to fully fix the client side parsing of it. Additionally, while the hard reset we implemented does indeed deal with a wide variety of failures, it is also an expensive operation. Some of our attempts to build lower levels of reset-and-rejoin ended up shutting down the Raspberry Pi.

6 CONCLUSIONS

6.1 How we would continue our work

There are several directions we would like to take our project in.

6.1.1 Recovery Mode

Our current recovery mode as mentioned before is too expensive. We would like to fix the issues with lower level reset-and-retry's. Primarily, we would like to try gracefully soft-reset first, and if it isn't enough, try the hard reset after a certain time period has elapsed.

6.1.2 Client side decryption

We would like to continue working on the client side decryption. Doing so would let us actually parse data in the payload of the messages, thus allowing the LoRa system to send actual decrypted packets to applications built on top of it. This would enhance its functionality and allow us to start using it as a proper communication technology. With proper communication, it would enable us to enhance its debug-via-server-message capabilities such as dumping statuses as an uplink packet.

6.1.3 MAC Commands

Whilst this was outside the scope of our regular project, we would like to refine the MAC command functionality. We would like to have seamless MAC command instructions to be conveyed from the server to the client.

6.2 Insights gained

This project gave us the opportunity to work directly with the LoRa framework. We were able to understand how downlink and uplink messages were structured, how clients and servers tell apart faulty uplinks and downlinks, and how the receive windows operated.

Beyond that, we were able to foray into dealing with communication errors. While implementing our recovery mode, we tested with different failure cases like timeouts, packet collisions, invalid downlink data. We wanted our recovery mode to be generic enough to handle all of these cases. As such, this made us gain insights on how to tailor recovery to deal with a variety of failures.

7 CODE

The server code was handled with modifications to wingstack. This can be seen on the wingstack branch 211_grp 11 as seen below.
https://github.com/dboyan/wingstack/tree/211_grp11

The client was handled with modifications to the lora-mcci repo. We maintained our changes on another branch.
https://github.com/dboyan/lora-mcci

8 REFERENCES

[1] The LoRa Alliance. 2018. *LoRaWAN™ 1.0.3 Specification.*

[2] The Things Network. (n.d.). *LoRaWAN Security*. LoRaWAN. https://www.thethingsnetwork.org/docs/lorawan/the-things-certified-security/