Multi-hop LoRaWAN: including a forwarding node

Bruno Van de Velde

***Abstract*—In this paper a proof-of-concept implementation of a forwarder device was developed that allows using LoRaWAN for multi-hop communication. Experiments were performed with Class A and Class C devices to test the impact of adding an extra hop to the communication. It is shown that adding a forwarder can improve the signal strength and significantly decrease the energy consumption of the end-device, providing it with a longer battery life.**

Keywords*—*LoRa, LoRaWAN, forwarder, multi-hop.

I. Introduction

The LoRaWAN protocol was designed for low power Internet of Things (IoT) devices. LoraWAN allows for consuming little power and having a long range. Furthermore, security is built into the protocol, all data is end-to-end encrypted.

Although LoRaWAN is designed for long-range communication, the range may not be sufficient in all cases. The end-device could e.g. be a water meter that is placed (deep) underground. Being underground can seriously impact the maximum communication distance. The range may also be insufficient in places with poor network coverage.

In this paper we therefore examine the feasibility of inserting a forwarder-node in between the end-device and the gateway to improve the range and quality of LoraWAN communications. We also examine the effect on the energy consumption of the end-device, as in general the energy needed to transmit a packet decreases when the distance between the devices becomes smaller.

A. LoRa & LoRaWAN

LoRa is a physical layer technology designed for long-range low-power wireless communication. The modulation format from Semtech can be described as a “frequency modulated chirp” [1]. This chirped spread spectrum radio modulation format allows an inexpensive chip with a cheap crystal to achieve very high sensitivity. LoRa is also capable of demodulating multiple orthogonal signals at the same frequency when they are using different chirp rates.

The chirp rate in LoRa is referred to as spreading factors (SF) in the datasheets. Higher spreading factors denote lower bitrates.

LoRaWAN specifies a MAC layer to use on top of LoRa [2]. It uses unlicensed radio spectrum in the Industrial, Scientific and Medical (ISM) bands. In this paper we only focus at communication that takes place in the 868 MHz band.

1) Topology: LoRaWAN specifies a topology where end-devices communicate with gateways. No device-to-device communication is defined. These gateway relays messages between the end-devices and the network server in the backend. Fig. 1 shows such a topology.

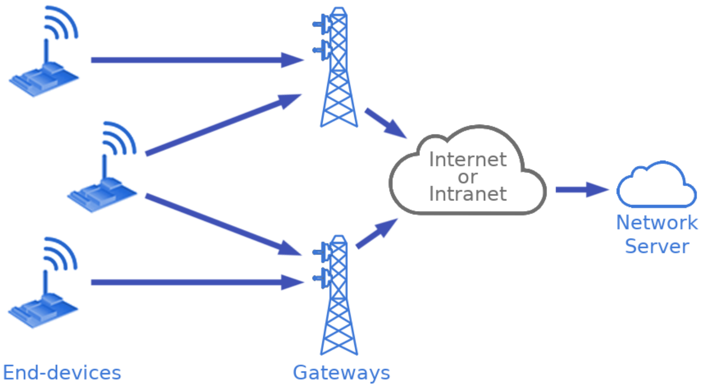

Fig. 1. LoRaWAN Topology, arrows in direction of uplink communication

The network server will in turn send the application data from the packet to the application server of the owner of the end-devices.

2) Data rates: LoRa supports different data rates, ranging from 0.3 kbps to 50 kbps. The higher the chosen data rate, the shorter the communication range will be. Selecting the data rate is thus a trade-off between the communication range and the transmission time. This data rate does not have to be static, LoRaWAN supports an adaptive data rate (ADR) scheme where the network server is able to change the data rate of each individual end-device.

The data rates for LoRa are given by the spreading factor. The lowest data rate corresponds to SF12 while the highest data rate corresponds to SF7. With every increased step (e.g. from SF8 to SF9), the transmission time approximately doubles.

Transmissions from devices using different data rates can occur simultaneously on the same frequency, as LoRa is able to demodulate simultaneous signals using different spreading factors [3].

With ADR, the data rate is changed on two different occasions:

- The network server can request the end-device to change its data rate. It can do this occasionally when it notices that the Signal to Noise Ratio (SNR) in the last several uplink messages was good enough or was too bad.
- When no acknowledgement (ACK) arrives at the end-device after sending a confirmed message (the type of message that requires an ACK), the end-device will retransmit the packet. After every 2 failed transmissions, the data rate will be decreased.

3) Classes: The LoRaWAN specification defines 3 different classes:

- **Class A: Bi-directional end-devices.** All end-devices must support at least Class A communication. The end-

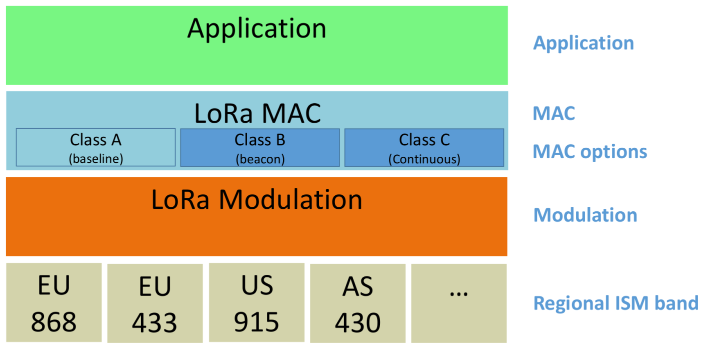

Fig. 2. LoRaWAN Classes

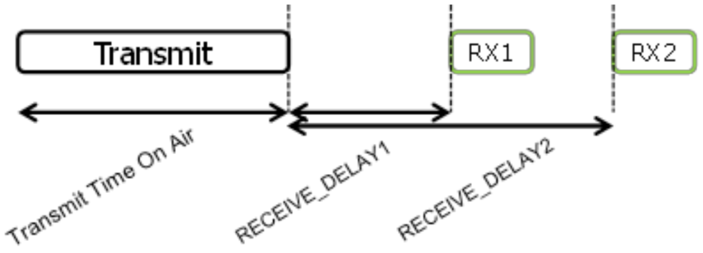

Fig. 3. End-device receive slot timing

device independently decides at which moment an uplink transmission will occur based on its own communication needs. The uplink transmission is followed by two receive slots during which downlink traffic can be sent. The network server can only send a downlink transmission shortly after an end-device has sent an uplink transmission. At any other time it will have to wait until the end-device performs another uplink transmission.

- **Class B: Bi-directional end-devices with scheduled receive slots.** The network server has more opportunities to send downlink transmissions to a Class B end-device. In addition to the Class A receive windows, extra receive windows are opened at scheduled times. This is made possible by time synchronized beacons that are transmitted from the gateway.
- **Class C: Bi-directional end-devices with maximal receive slots.** End-devices of Class C have nearly continuously open receive windows. They only stop listening on the radio when they perform an uplink transmission. The network server can send downlink messages at any time. Class C end-devices will obviously consume much more power than end-devices operating under Class A or Class B.

4) Receive slot timing: Class A communication occurs as shown in Fig. 3. After the packet has been transmitted, the end-device waits RECEIVE_DELAY1 seconds before opening the RX1 receive slot. The duration of RECEIVE_DELAY1 is configurable by the network server and is 1 second by default. If the gateway replies during RX1, the data rate will be based on the one that was used in the transmission by the end-device. By default the data rate used here will be identical to the data rate of the uplink transmission. The frequency used is always identical to the frequency in the uplink transmission. If a packet was received during RX1, no RX2 slot will be opened.

When no packet was sent during RX1, an RX2 receive slot will be opened one second later (the duration of RECEIVE_DELAY2 equals RECEIVE_DELAY1 plus one second). The frequency and data rate used during RX2 is preconfigured on both end-devices and gateways. The frequency and data rate do not have to be static, they can be changed with MAC commands from the network server.

The RX1 and RX2 receive slots remains open for approximately 20 microseconds each when no packet is received.

B. Forwarder

When devices have a bad connection, they have to use a lower data rate which increases the transmission time. Because the radio remains on longer, this leads to a higher energy-consumption and reduces the available airtime of other devices. A lower data rate also affects the amount of data that can be sent by the end-device. The maximum payload size is less for lower data rates and the duty cycle limitation increases the time before the end-device can send another packet on the same sub-band.

To address these issues, a “forwarder” node is added between the end-device and gateway. The end-device and gateway remain compatible with the LoRaWAN specification, but the communication between them is redirected through the forwarder.

Adding a forwarder to the topology differs from adding another gateway in two ways:

- A gateway typically forwards the packet to the network server via an ethernet connection. The forwarder will however use LoRa to communicate with the next hop in the topology. It can thus be placed on locations where infrastructure is missing.
- Adding a gateway is only an option if you own the network infrastructure. When you only own the end-devices and the gateways are a service which you use then adding a gateway close-by may not be an option.

II. Methodology

A. Components

1) End-device: We used an EFM32GG-STK3700 Giant Gecko [4] as end-device. To communicate using LoRa, a HopeRF RFM95W [5] was connected to the Expansion Header of the Gecko.

Since no LoRaWAN implementation existed yet for this hardware, we ported the reference implementation [6] to support the Gecko [7].

2) Gateway and network server: The gateway we used consists of an iC880A concentrator board [8] and a Raspberry Pi 3, connected with each other over SPI.

The concentrator board itself does not run any code. The gateway and the network server code both ran on the Raspberry Pi. In our setup we even ran the application server and application on the Raspberry Pi.

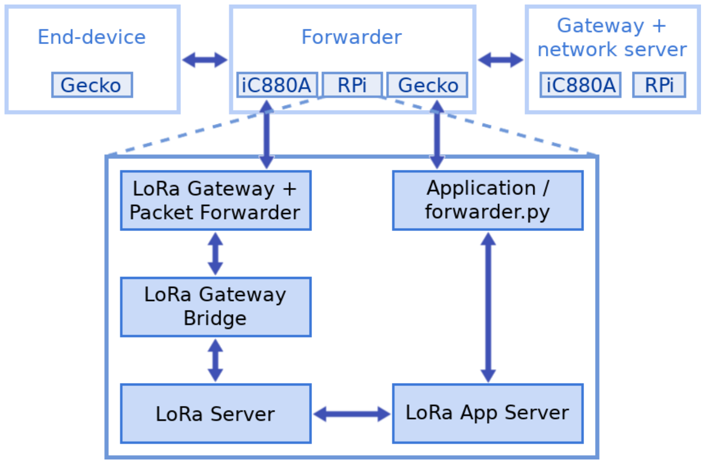

Fig. 4. Forwarder components

3) Forwarder: Since the forwarder is only a proof-of-concept, we build it to be as simple as possible. The hardware of the end-device and gateway where combined to form the forwarder.

The Gecko is connected to the Raspberry Pi by two USB cables, one to provide power and one to communicate. The latter USB was connected to a SparkFun FT232RL USB to Serial Breakout [9] to which the Gecko communicates using the UART.

The software components running on the Raspberry Pi are shown in Fig. 4. The "gateway" component consists of a LoRa Gateway library [10] for communicating with the iC880A concentrator board and a Packet Forwarder [11] that forwards the packets to the LoRa Server over UDP. Since the LoRa Server does not accept UDP packets as input, the LoRa Gateway Bridge [12] is placed in between to convert the packets into the right format. The LoRa Server [13] will process the packet and send the application data (if any) to the LoRa App server [14]. From there, the data is delivered to the "application", which is a python script that communicates with the Gecko.

B. Forwarder implementation

The forwarder runs everything that runs on the gateway and network server. This allows it to communicate with the end-device without needing any changes. The Gecko connected to the Raspberry Pi runs the same LoRaWAN implementation as the end-device to allow communication with the upstream gateway.

The code from the end-device and network server only supports Class A and Class C devices. This is because Class B is still experimental and will undergo changes in the LoRaWAN 1.1 specification. Our forwarder thus also does not support Class B devices.

To redirect all transmissions through the forwarder, the Application Key of the end-device is changed to the one configured on the forwarder. The Gecko of the forwarder will be using the Application Key corresponding to the upstream network server. These Application Keys are used to derive the network and application session keys during the join procedure. The Application ID also has to be updated when the forwarder does not use the same Application ID as the network server. Finally, the network server has to be configured to accept joins with the DevEUI of the forwarder instead of the DevEUI of the end-device.

For simplicity, we are only looking at the case where there is a single end-device connecting to the forwarder.

1) Naive implementation: We started with the most straightforward implementation. When the LoRa App Server received a packet, the application would be notified and send the packet to the Gecko over the serial connection. The Gecko then sends the packet to the gateway after a small delay (to avoid a collision with the reply that the forwarder may send to the end-device). When downlink data is received on the Gecko, the inverse path is followed. The Gecko send the data to the application over the UART and the application delivers the data to the LoRa App Server, which will make sure that the packet is send to the end-device at the next opportunity.

This setup has the advantage of being simple, no changes have to be made to the existing code, but it has several downsides:

- The downlink packets arrive too late for Class A devices. The RX receive slots on the end-device will have passed before the forwarder receives the downlink data from the gateway. The downlink message is only delivered to the end-device during the next uplink transmission.
- Packets could be lost without the end-device knowing. The forwarder will acknowledge confirmed packets before it has transmitted them to the gateway. If the packet is lost between the forwarder and gateway then the end-device will still believe that the transmission was successful.
- If the forwarder is allowed to retransmit packets, then it might exceed its allowed duty cycle faster than the end-device. It is possible that when the end-device sends a packet, the forwarder still has a timeout before it is allowed to transmit it to the gateway.
- The frame pending bit (FPending) will be incorrect. The queue in the forwarder can be empty while the gateway still has data. The forwarder will incorrectly inform the end-device during a downlink transmission that there are no further downlink packets waiting in the queue.

2) Improved implementation: To work around these downsides, we adapted the implementation in several places.

The general idea is that the forwarder should first transmit the packet to the gateway and wait long enough to get a reply back, before sending its answer to the end-device. This can only be achieved by increasing the RECEIVE_DELAY1 on the end-device which defines the amount of seconds between the uplink transmission and the RX1 receive slot. This is done by setting the RxDelay byte in the join-accept message. The LoRa Server component running on the forwarder also had to be changed to wait longer before asking the LoRa App Server whether there was downlink data. This DownlinkDataDelay has to be slightly smaller than the RxDelay but large enough such that there is enough time to transmit the uplink data to the gateway and receive a potential reply with downlink data. We currently hardcoded the DownlinkDataDelay to 9 seconds while the RxDelay is set to 10 seconds.

The next step is to not let the forwarder send anything to the end-device when nothing was received from the gateway. To do this, the downlink queue was removed from the Lora App Server and the way ACKs are handled was changed. An ACK is now only send to the end-device when one was received from the gateway.

To support uplink ACKs we also needed to make a minor change to the LoRaWAN implementation on the Gecko. The ACK flag for uplink packets is internal to the LoraWAN stack while we need to set it manually from the application depending on how the flag was set in the uplink packet from the end-device. The LoraWAN stack was therefore modified to allow this.

3) Remaining challenge: While performing the final experiments we realized that there was one issue that we did not anticipate.

Imagine the end-device sending a packet of 29 bytes to the forwarder with SF9 and the forwarder sending it to the gateway with SF11. When the packet gets lost between the forwarder and gateway then the end-device will attempt to retransmit. It has to wait at least 22.7 seconds before being allowed to transmit on that frequency band again and the end-device will retransmit as soon as it is allowed to. The forwarder on the other hand was not allowed to transmit yet as it had to wait 90.6 seconds before transmitting again, as a result of having used SF11. Many retransmissions may be dropped before the forwarder will be able to attempt delivering the packet to the gateway again.

This issue could thus increase the amount of undelivered packets. Due to time constraints we did not conduct further research into this issue.

In our experiments the link between the forwarder and gateway used a higher data rate than the link between the end-device and forwarder which prevented the issue from showing up in our test results. The issue only occurs in the opposite case.

III. Experiment setup

We set up some tests to examine the impact of an extra hop on the energy consumption of the end-device, the message airtime and the throughput of the communication.

A. Payload

The payload size of each packet was chosen to be 16 bytes (large enough to contain statistics about the communication of the end-device). The header is usually 13 bytes but is 2 bytes larger when an ADR answer is added. The total packet size is thus 29 or 31 bytes, depending whether the network server had just requested the end-device to change the data rate or TX power. Most packets will be 29 bytes since such requests do not occur frequently.

B. Schedule

Each test ran for 12 hours, with the end-device programmed to send a confirmed packet every 3 minutes. The first hour of each test is ignored. The communication is always started using SF12 and a TX power of 14 dBm. By removing the first hour of the test we are only looking at the communication after the ADR has “stabilized” the data rate and TX power, which is the more interesting part of the measurement. The data presented later in the paper is thus only for a duration of 11 hours.

When no ACK is received, the packet will be retransmitted up to 8 times. These retransmissions follow each other as quickly as possible, depending on the duty cycle. When a packet is 29 bytes, transmission with SF12 would take 1647 milliseconds and the device would not be allowed to retransmit in the next 164.7 seconds. When transmitting with SF7, the device can however already retry after only 6.7 seconds.

C. Adaptive Data Rate

We enabled ADR on all devices. On the network servers (both on the upstream one and on the code running on the forwarder), there are two settings related to this that had to be configured. The ADR interval defines after how many frames the ideal data rate and TX power of the end-device should be recalculated. We set this value to 5, which means that the network server may request the end-device to change its data rate and TX power at most every 5 successful uplink transmissions. The second parameter is the installation margin which influences the relation between the SNR and changes in data rate and TX power. A higher margin will lower the data rate and therefore decrease packet-loss. A lower margin will increase the data-rate and therefore increase possible packet-loss. We set installation margin to the recommended value of 5 dB.

While the data rate was variable, the bandwidth was fixed to 125 kHz and the code rate was always 4/5.

Even when ADR is enabled, when the gateway replies during RX2, a fixed data rate would be used. By default SF12 is used. We decided to use the RX1 receive slot instead, so that the gateway uses the same data rate as used for the uplink transmission. This makes the difference between using different data rates even more visible.

D. Classes

We looked into two different types of devices:

- Class A device which only sends periodic uplink data but does not receive downlink data. An example of such a device is a simple sensor that does nothing more than measuring something and transmitting the measurements.
- Class C device which can also receive downlink data next to its periodic uplink transmission. Commands can be sent to the device to perform an action (e.g. change the color of the LED lights). Such devices typically don’t run on batteries because the device has to be constantly listening. In our tests, a downlink packet is generated every 10 minutes.

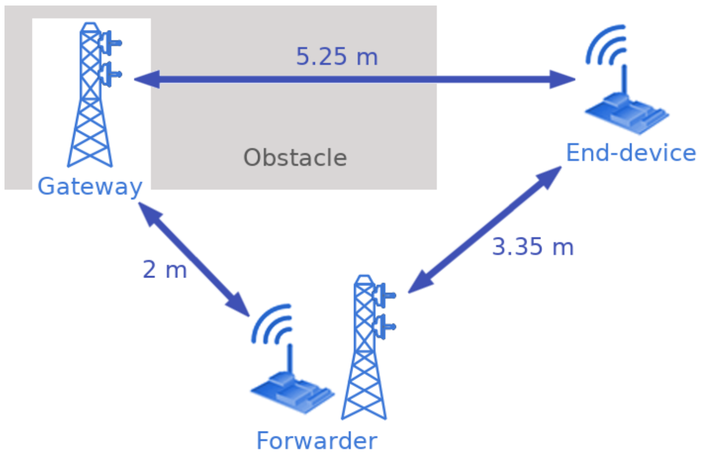

Fig. 5. Topology used in test setup

E. Placement

In order to simulate a bad connection, the antennas were removed from all devices. This made it much easier to position the devices at a place with the wanted signal quality. If the SNR is too low then we lose too many packets, but if it is too high then the connection is so good that placing a forwarder in between would not improve much.

The placement of the devices with the approximate distance between them is shown in Fig. 5. The gateway was placed in an open cupboard and the forwarder was placed such that it has line of sight to both the end-device and gateway.

This setup simulates the case where the devices with antennas attached are placed several hundreds of meters away from each other with a forwarder in between them.

F. Energy consumption

The Gecko contains an Advanced Energy Monitor (AEM) which can be used to measure its energy consumption [15]. There were however several issues that we found with using the Simplicity Studio IDE and its integrated energy tool that reads the values from the AEM:

- The measured consumption is limited to 50 mA while the energy consumption of the radio lies higher than this.
- Only the data of the last 45-120 minutes was available while our tests lasted 12 hours.
- We did not implement the Gecko code to be energy efficient as optimizing the energy consumption of the microcontroller was not the focus of this work. The consumption during sleep could still be 2 mA and the average consumption would thus lie much higher than with other hardware.

We therefore used a different method to compare the energy consumption of the end-device. For the RX mode of the radio, we simply kept track of the cumulated duration which we can directly compare with other tests as all tests have the same duration. The consumption from TX is harder to compare since not only the duration, but also the TX power is variable. For this we calculate the "on-air" energy consumption, which is the minimum energy needed to physically transmit the packet. We first converted the TX power from dBm to mW with the following formula:

$$P_{(mW)} = 1mW * 10^{(P_{(dBm)}/10)}$$

We then calculated the consumed energy E in Wh based on the transmission time Δt:

$$E_{Wh} = \frac{P_{(mW)}}{1000} * \frac{\Delta t}{(1000 * 60 * 60)}$$

We finally converted the energy to joule by multiplying with 3600.

The final formula to calculate the total energy E_J from TX power P and transmission time Δt for all transmissions is thus:

$$E_J = \sum_{i \in transmissions} \frac{10^{(P_i/10)} * \Delta t_i}{10^6}$$

This calculation does not take "real-world" factors into account, such as the energy consumption of the microcontroller, the radio chip inefficiencies and heat loss. For this reason, this metric does not provide an accurate representation of the real energy consumption. However, the metric does allow comparing the minimum energy required to transmit the packets.

G. Comparison

For each type of device (Class A and Class C), we performed 3 tests:

- The end-device and gateway communicate directly but have a relatively bad connection.
- There is a forwarder between the end-device and gateway. Both links have relatively good connections.
- We also tested the more ideal case as reference. The end-device and gateway communicate directly but with antennas attached. They thus have a very good connection with each other.

In these tests, we kept track of the following statistics to be able to compare the tests with and without forwarder. The same statistics were kept on the forwarder as on the end-device and gateway.

- The TX power of each uplink packet.
- The airtime of uplink packets.
- The duration the radio spend in RX mode.
- The amount of packets received at the end-device (only ACKs for a Class A device).
- The amount of application layer packets send by the end-device.
- The amount of (physical layer) packets send by the end-device (including retransmissions).
- The spreading factor used for each transmission.
- The spreading factor of each uplink packet arriving at the gateway.
- The RSSI and SNR of packets arriving at the gateway.
- The airtime of downlink packets.

IV. Results

A. Class A

1) Bad direct connection: The setup with a bad connection turned out to be really stable. All 222 packets were transmitted using SF11 at 14 dBm and only one of these packets was a retransmission. The reason why the connection was so stable and the ADR appeared to do nothing can be found in the SNR values. On average the SNR (which ranges from -20 to 20) was -12.55. At such a level and with an installation margin of 5 dB, the network server won't even attempt to increase the data rate beyond SF11.

The RX time was 130.265 seconds while the on-air consumption equaled 5.052 J.

2) With forwarder: A few packets were lost in the test with the forwarder in between. Out of the 208 application layer packets that were send, only 203 were acknowledged. In total 245 packets were transmitted.

In this test, there was a good and a bad link. On the link between the forwarder and gateway, all packets were transmitted with SF7 at 2 dBm, while on the link between the end-device and forwarder the packets were mostly transmitted with SF8 or SF9 at 14 dBm. The links had an average SNR of -2.21 and -5.83 respectively, where the first one is only negative because of the low TX power used.

The RX time of the end-device was 39.180 seconds while the on-air consumption equaled 1.676 J. To provide the full airtime we also looked at the Gecko of the forwarder. Here the RX time and on-air consumption were only 7.976 seconds and 0.022 J.

3) Good direct connection: Because the RSSI was still between -100 to -110 dBm (where -120 dBm is around the minimum RSSI for which the radio can distinguish data from noise) and the data rate varied a lot even when the devices were placed directly next to each other, we used antennas to show the situation with a great connection. As expected, SF7 was used almost everywhere with a TX power of 2 dBm.

With an average SNR of 9.33, the results should be much better than in the other tests. Although the RX time of 15.274 seconds and the on-air consumption of only 0.035 J are great, they are worse than on the link between the forwarder and gateway in the previous test.

The reason for this lies in the fact that out of the 218 application layer packets only 216 were acknowledged. In total 232 packets were send, which means that not a single packet was lost other than all retransmissions of those 2 packets. It is unclear what caused this behavior.

4) Comparison: Fig. 6 and Fig. 7 show the SNR and SF of each packet in the three tests. It is clear from Fig. 7 that in all cases where 2 packets in a row where lost, all subsequent retransmissions were lost as well, causing the SF to be increased by 3 each time.

Fig. 8 shows the RX time. It stands out that the RX time has nearly doubled in the test with the good direct connection compared to the communication between the forwarder and gateway. The loss of these 2 packets (and their retransmissions) thus clearly had a high impact on the energy consumption.

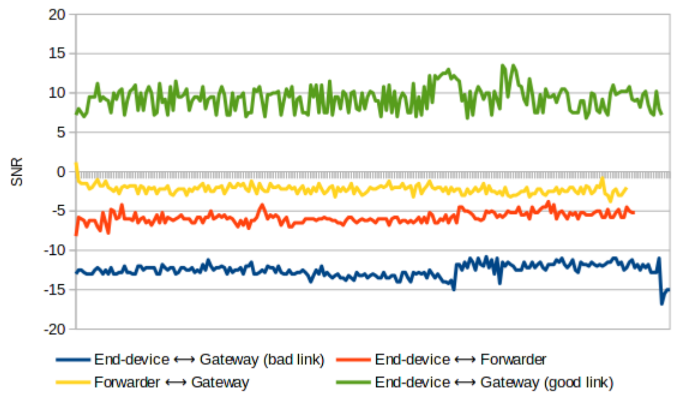

Fig. 6. SNR of each packet that arrived at the forwarder/gateway

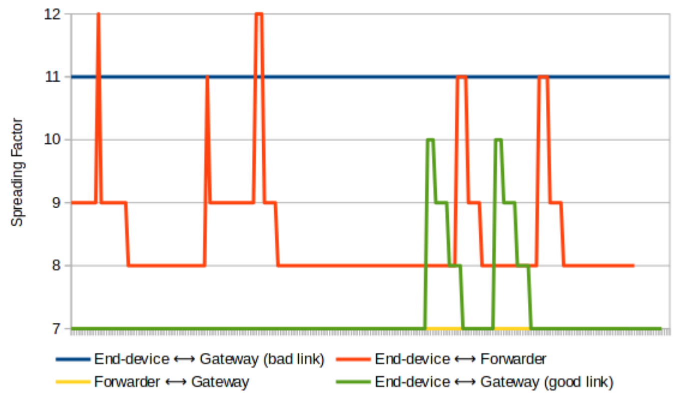

Fig. 7. SF used in each packet that arrived at the forwarder/gateway

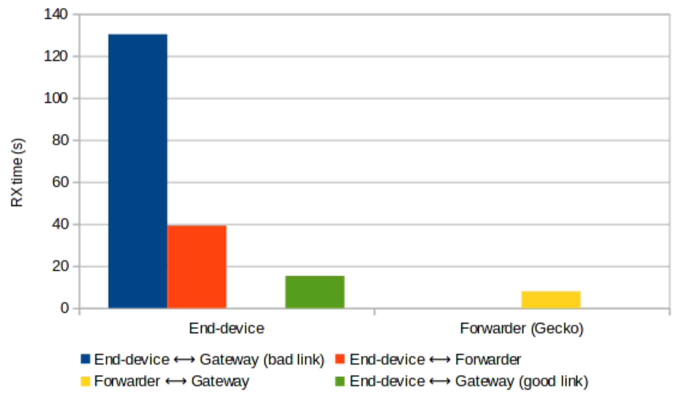

Fig. 8. Time radio spent in RX mode

In Fig. 9, the energy consumption caused by transmissions is shown. The relation between the bad connection and bad forwarder link is similar to the one in Fig. 8, but less prominent due to the logarithmic scale. The fact that a logarithmic scale is needed to visualize the difference with the good connections clearly shows the large impact of a higher data rate on the energy consumption.

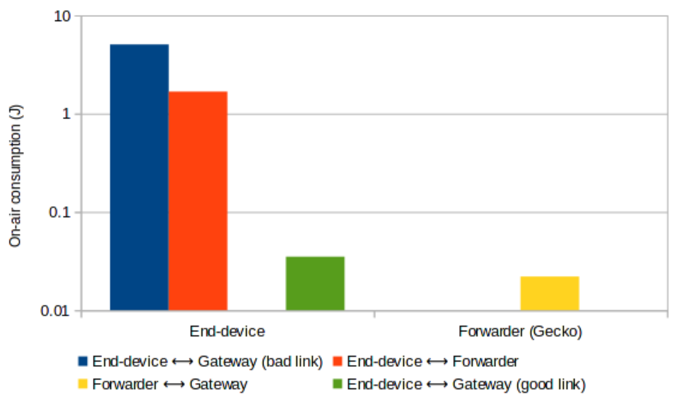

Fig. 9. On-air consumption (logarithmic scale)

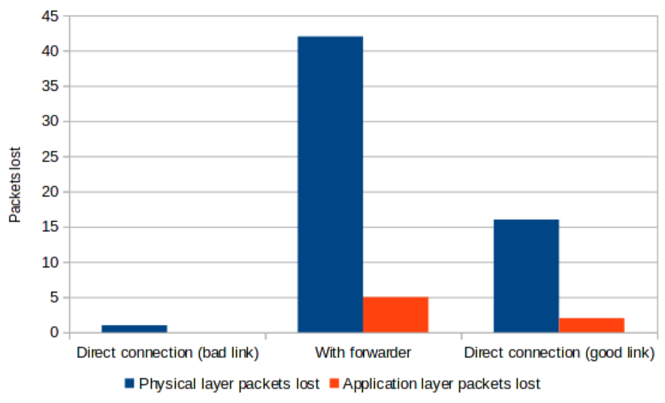

Fig. 10. Packets lost during tests

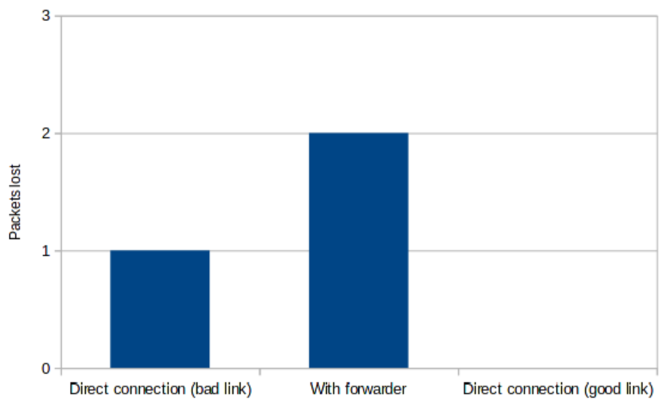

Fig. 11. Physical layer packets lost during tests, excluding cases where all retransmissions were lost

With these results we have shown that adding a forwarder has a positive influence on the energy consumption of the end-device. The lower spreading factor lead to the total airtime being more than 3 times smaller than without a forwarder, even with more packets being sent in the case where the forwarder was present. Because the duration during which the radio is active on the end-device decreases, this results in a lower energy consumption and thus a longer battery life.

From our results it appears that having a forwarder leads to increased packet loss. This is especially visible in Fig. 10, which shows how many packets were lost in each test. There are however several observations that should be taken into account. First of all, the test for a "bad connection" has very good results. As Fig. 10 shows, less packets were lost compared to the setup with a good connection. This "bad connection" setup was tested multiple time before the experiment discussed in this paper was conducted. We found that by moving the end-device a few centimeters, the signal quality varied a lot. In one test (which lasted longer than 11 hours) we transmitted 272 packets (21 with SF10, 114 with SF11 and 146 with SF12) of which only 232 arrived at the gateway and for only 229 the ACK reached the end-device. The first test thus could have been a lot worse than the good results where all packets were send with SF11. Secondly, most packet loss seems to occur on the retransmissions of the same packet. This could indicate that there is a bug somewhere in the end-device, gateway or network server that causes the transmission to be consistently unsuccessful even when the data rate is lowered 3 times. It is thus not clear how much the forwarder truly impacted the packet loss.

If we however assume that the cases where application layer packets were lost are caused by a bug, the results look more like we expected. The ADR algorithm chooses a data rate where the communication is expected to be stable, so only few packets are supposed to be lost. Fig. 11 shows the packet loss if we exclude the packets where all retransmissions were lost.

B. *Class C*

1) Bad direct connection: The end-device did not move compared to the Class A test and we again found that all packets were transmitted with SF11 with a TX power of 14 dBm. Again only one packet out of the 222 was a retransmission. The average SNR was -12.62 in this test. This time the network server generated downlink packets every 10 minutes as well and we found that all 66 packets have successfully been received and acknowledged by the end-device.

The on-air consumption remains unchanged at 5.052 J. The RX time on the other hand is far greater than with the Class A test because here the radio stayed in RX mode the whole time as the downlink packets could arrive at any time.

2) With forwarder: Unfortunately we were not able to test Class C devices with the forwarder. Although the code on the forwarder is based on the network server code which supports Class C as shown in the test with a direct connection, the forwarder failed to deliver the packets correctly. We believe that this is a mistake in our implementation that was made between the start of the project and the moment we performed these final tests. There should be no technical limitation that prevents the forwarder from supporting Class C devices.

We however did not see much impact of having a forwarder in between the end-device and gateway. When a downlink packet was scheduled, a packet arrived at the end-device only moments later (as the bug only affected the contents of the packet, not the presence of them).

3) Good direct connection: Here the results were even better than for the Class A test. All 221 application layer packets were transmitted with SF7 with TX power 2 dBm and were acknowledged. Only one packet had to be retransmitted.

The average SNR was 9.05 while the on-air consumption was only 0.0236 J because there were no retransmissions that caused a lower data rate to be used.

4) Comparison: Without a full test with the forwarder in between, not many conclusions can be drawn. We can however see that our results are very similar to the Class A tests. We don't expect much extra information to be visible in the Class C test with forwarder, except about the latency of downlink packets.

Since downlink traffic for Class C devices is always sent using the settings of an RX2 receive slot, adding the forwarder doubles the latency. The forwarder provides no improvement on the used data rate and the downlink message has to be transmitted twice instead of once when there is a forwarder, the total airtime thus doubles.

V. Conclusion

We have shown that it is possible to add an extra hop between the end-device and gateway without having to change the code on these devices.

Having such a forwarder provided several advantages:

- The communication range can be increased and communication can become possible where the end-device and gateway could not reach each other.
- The signal strength can be improved which should result in less packet loss when a fixed data rate is used.
- The energy consumption of the end-device can go down if ADR is used and a better data rate becomes possible. In our experiments the time the radio was active was reduced with more than 3 times.

There are however some potential downsides as well:

- Latency of uplink packets increases. The time between sending a packet and receiving a response becomes longer even when the airtime itself reduces. This is because every hop increases the delay with at least one second (the minimum RECEIVE_DELAY1 value) and the delay has to be taken larger than needed when ADR is used (as the airtime can suddenly become much larger when the data rate decreases).
- For Class C devices, the latency of downlink traffic doubles (when assuming the same settings for RX2 are being used on the forwarder as on the gateway).

A. Future work

1) Data rate: The issue that can cause packet loss when the data rate between the end-device and forwarder is higher than the data rate from forwarder to gateway should be looked at. The same issue has to be solved when you would want to connect multiple end-devices to the same forwarder.

2) Class C: There are issues in our forwarder implementation that caused Class C devices to be unsupported. We however believe that there is no technical limitation here, after the bugs are fixed, the forwarder should support Class C devices as well without further issues.

References

[1] B. Ray, Link Labs (2015, Feb. 14). *What is LoRa?* [Blog]. Available: https://www.link-labs.com/blog/what-is-lora

[2] LoRa Alliance, *LoRaWAN Specification*, 2015.

[3] A. Augustin et al., "A Study of LoRa: Long Range & Low Power Networks for the Internet of Things", Sensors, vol. 16, no. 9, p. 1466, 2016.

[4] Silicon Labs. *EFM32 Giant Gecko 32-bit Microcontroller* [Online]. Available: http://www.silabs.com/products/mcu/32-bit/efm32-giant-gecko

[5] HopeRF. *RFM95W 868/915Mhz RF Transceiver Module* [Online]. Available: http://www.hoperf.com/rf_transceiver/lora/RFM95W.html

[6] Semtech. (2013). *LoRaWAN endpoint stack implementation and example projects* [Online]. Available: https://github.com/Lora-net/LoRaMac-node

[7] Bruno Van de Velde. (2017). *Fork of the LoRa-net reference LoRaMAC-node implementation that adds support for the EFM32GG-STK3700 / RFM95W hardware platform* [Online]. Available: https://github.com/imec-idlab/LoRaMac-node

[8] Wireless Solutions. *iC880A - LoRaWAN Concentrator 868MHz* [Online]. Available: https://wireless-solutions.de/products/radiomodules/ic880a.html

[9] SparkFun. *SparkFun USB to Serial Breakout - FT232RL* [Online]. Available: https://www.sparkfun.com/products/12731

[10] Semtech and TheThingsNetwork. (2013). *Driver/HAL to build a gateway using a concentrator board based on Semtech SX1301 multi-channel modem and SX1257/SX1255 RF transceivers.* [Online]. Available: https://github.com/TheThingsNetwork/lora_gateway

[11] Semtech and TheThingsNetwork. (2013). *Packet forwarder for Linux based gateways* [Online]. Available: https://github.com/TheThingsNetwork/packet_forwarder

[12] O. Brocaar. (2016). *LoRa Gateway Bridge abstracts the packet_forwarder protocol into JSON over MQTT* [Online]. Available: https://github.com/brocaar/lora-gateway-bridge

[13] O. Brocaar. (2016). *LoRa Server is an open-source LoRaWAN network-server* [Online]. Available: https://github.com/brocaar/loraserver

[14] O. Brocaar. (2016). *LoRa App Server is an open-source application-server for LoRa Server* [Online]. Available: https://github.com/brocaar/lora-app-server

[15] "User Manual Starter Kit EFM32GG-STK3700," Silicon Labs, Austin, TX, Okt. 10, 2013.