UNIVERSITÉ
SAVOIE
MONT BLANC

LoRaWAN®

for advanced users

Antoine AUGAGNEUR
Sylvain MONTAGNY

CLOCK SYNCHRONIZATION

MULTICAST

FRAGMENTATION

FUOTA

JOIN SERVER

ROAMING

About this Book

This book is the result of work carried out by the teaching staff of Savoie Mont Blanc University. It follows the first eBook called "LoRaWAN for beginners" also available for free on our website.

All remarks, modifications, improvements or corrections can be proposed on our website contact page: www.univ-smb.fr/lorawan/en/contact/.

Summary

1 Clock Synchronization

1.1 Clock Synchronization in LoRaWAN®: why?

Setting the time in an IoT end-device can be very useful. In some cases, it can be a mandatory feature if the application requires to timestamp data or events. There are several ways to set up the time remotely when using LoRaWAN®, and one of these methods is called **Clock Synchronization**.

1.1.1 Definition

Clock Synchronization is a LoRaWAN feature. It means synchronizing the time of an end-device with the time of the LoRaWAN server it is connected to. The end-device simply sends its time, and the server corrects it.

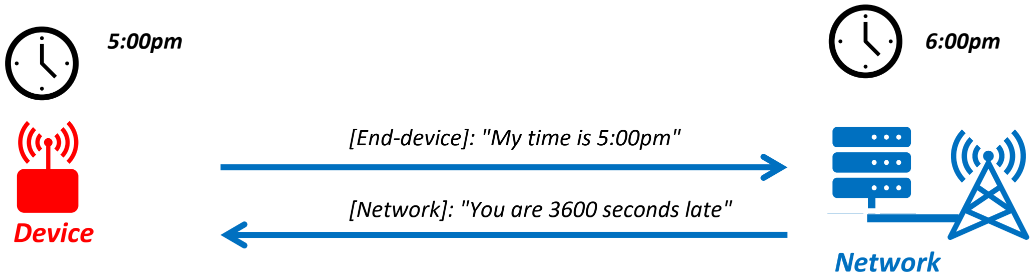

5:00pm
6:00pm
[End-device]: "My time is 5:00pm"
[Network]: "You are 3600 seconds late"
Device
Network

Figure 1: Time synchronization between an end-device and the network

Obtaining the right time requires a specific package called **Clock Synchronization** which is detailed in the following document: LoRaWAN® Application Layer Clock Synchronization Specification.

LoRaWAN Clock Synchronization allows an end-device to be synchronized. This feature is only interesting if the end-device doesn't have another time source available such as:

- An accurate time source (e.g., GPS, …)
- A class-B beacon synchronization
- A specific MAC command available for LoRaWAN version 1.1 and above (*DeviceTimeReq* MAC command).

- **In this course, we only focus on the Clock Synchronization Application Layer.**

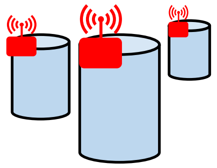
1.1.2 Use cases

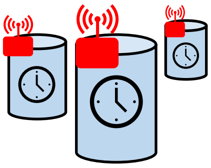
Let’s take a look at a few cases where a time-synchronized end-device can be useful.

Case 1: Time stamped metering

In this example, we assume that an operator owns water tanks and wants to assess their water level. The operator has hundreds of tanks and needs each water level to be measured **at the same time**.

The first idea is to measure all water levels at a specific time and send the value right away to the Network Server. In that case, we don't timestamp the data in the end-device, because it's the Network Server that will apply the time on each received packet. In many situations, this behaviour may not work because it is not recommended to send LoRaWAN data simultaneously as there is a high risk of collision. Indeed, if there is a retransmission due to packet loss, then the timestamp provided will be wrong. This situation shall be avoided.





The **Clock Synchronization** feature offers a better solution. When each end-device is clock synchronized, one device can process its water level measurement synchronously with the others and save the measurement as well as the corresponding timestamp in a transmission buffer. Each end-device sends its data asynchronously (with a random time) to avoid collision.

Case 2: multicast context:

LoRaWAN Clock Synchronization can be used as a single feature. However, it can also be used with other features such as **Multicast**. Multicast will be explained later on in this course.

Multicast allows sending a single frame to a group of end-devices. It requires that each end-device of the group be ready to listen to the incoming downlink frame. If the end-device works in class C, there is no issue. But if it works in class A, it must be switched to class C for a few seconds to receive the downlink packet. In order to make each end-device open their class C session at the same and right time, the devices must be time-synchronized.

The following figure illustrates this situation.

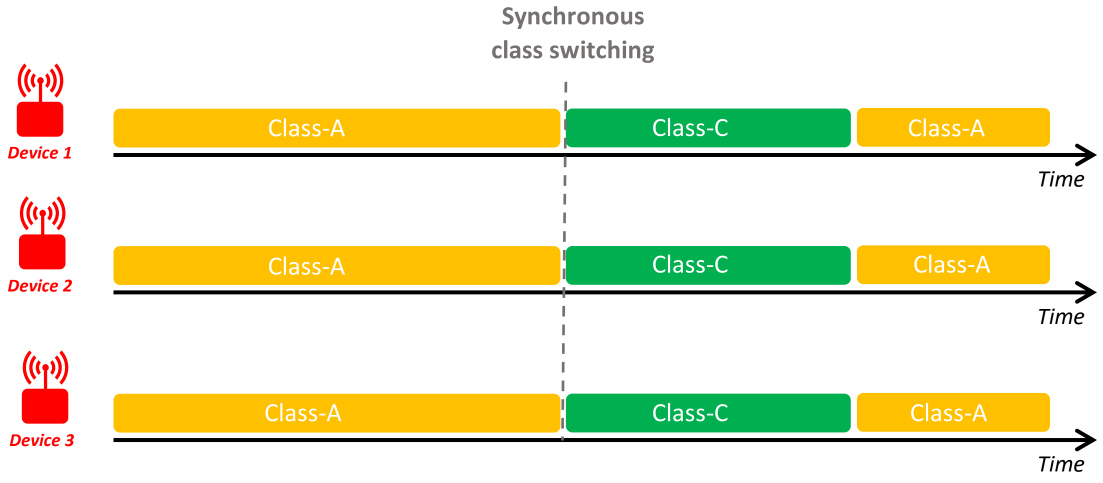


Figure 2: Synchronous class switching in Multicast context

Remember that a low-power end-device shall remain for as little time as possible in class-C.

1.2 How does it work?

As mentioned previously, this section will only deal with the Clock Synchronization messaging layer package.

1.2.1 Application layer and MAC layer

As a reminder, there are three LoRaWAN protocol layers:

- **The Physical Layer**: refers to the modulation method to send and receive data.
- **The MAC Layer:** the heart of the LoRaWAN protocol. It allows end-device authentication, data encryption, MAC commands, …
- **The Application Layer:** corresponds to the layer that the user can manage and edit.

Clock Synchronization is an **application layer** messaging package. The end-device must support this package in its LoRaWAN stack to allow the end-device to send and receive **clock commands**.

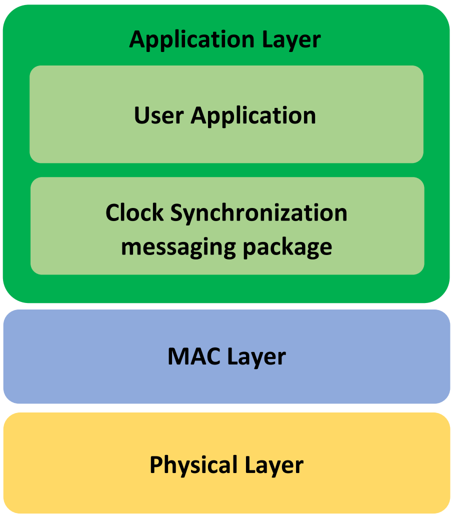


Figure 3: LoRaWAN protocol layers and the Clock Synchronization package

Because the Clock Synchronization package and the user application belong to the same layer, the end-device gets confused and mixes up data and application commands. To solve this issue, Clock Synchronization commands are received and sent on the dedicated port 202 (FPort). This means that Clock Synchronization commands must be sent on port 202, while all other non-dedicated ports can be used by the user application.

- **Only end-devices that own a LoRaWAN® stack with the Clock Synchronization Package can deal with Clock commands.**
- **Only LoRaWAN servers that respect the Clock Synchronization Specification can offer time features.**
- **Clock Synchronization commands are sent on port 202.**

1.2.2 Clock Synchronization

The Clock Synchronization package offers several commands, which are all sent on port 202. The full presentation of these commands is available in the LoRaWAN® Application Layer Clock Synchronization Specification. However, only the **AppTimeReq** command is used to synchronize the end-device's clock. The request is made by the end-device, and the clock correction is provided by the LoRaWAN server.

- Step 1: The end-device sends the **AppTimeReq** command that embeds the end-devices current time.
- Step 2: The LoRaWAN server analyses the current time of the end-device and sends a time correction. This is the **AppTimeAns** command.

> Time values are indicated in seconds. The time (date, hours, minutes, and seconds) is expressed by the number of seconds since 00:00:00, Sunday 6th of January 1980 modulo 232.

> The delta correction sent to the end-device is set with the current LoRaWAN server clock. Therefore, the correction depends on the UTC time zone of the LoRaWAN server.

1.2.3 How to clock-synchronize an end-device?

The end-device requests a clock correction (**AppTimeReq**). But, if the end-device is not programmed to send this request, we can force it by sending a **ForceDeviceResyncReq** to the end-device. The overall behaviour is as follows:

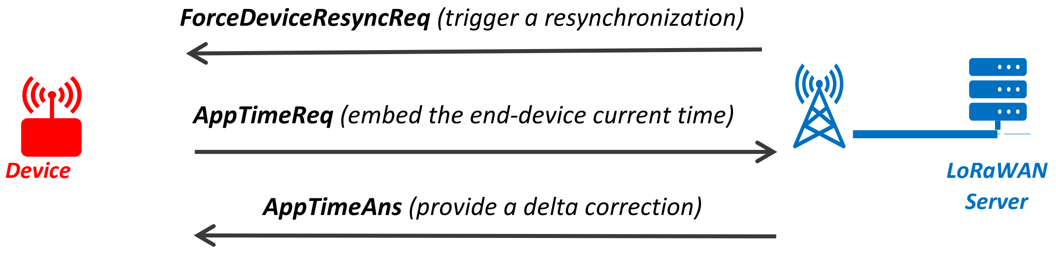

ForceDeviceResyncReq (trigger a resynchronization)
AppTimeReq (embed the end-device current time)
AppTimeAns (provide a delta correction)
Device
LoRaWAN Server

Figure 4: Clock Synchronization sequence

1.2.4 How to test the Clock Synchronization feature?

To test the Clock Synchronization, we need:

- An end-device with Clock Synchronization package enabled
- A LoRaWAN Gateway
- A LoRaWAN Network Server supporting Clock Synchronization
- A Clock Synchronization server

For this demonstration, we will use:

- A STM32WL end-device with LoRaWAN version 1.0.3 and Clock Synchronization enable
- A LoRaWAN Gateway
- ThingPark Community [https://community.thingpark.org]
- Our open-source Clock Synchronization server (Node-RED):
  [https://github.com/SylvainMontagny/fuota-server]

> Our open-source Clock Synchronization server works with any end-device, any gateway and any Network Server, as long as it provides the Clock Synchronization service.

Depending on your Network Server, you might need to set up your own MQTT Broker to interface the Network Server (MQTT client) and the Clock Synchronization server (MQTT client). Figure 5 presents the overall infrastructure for the Clock Synchronization demonstration.

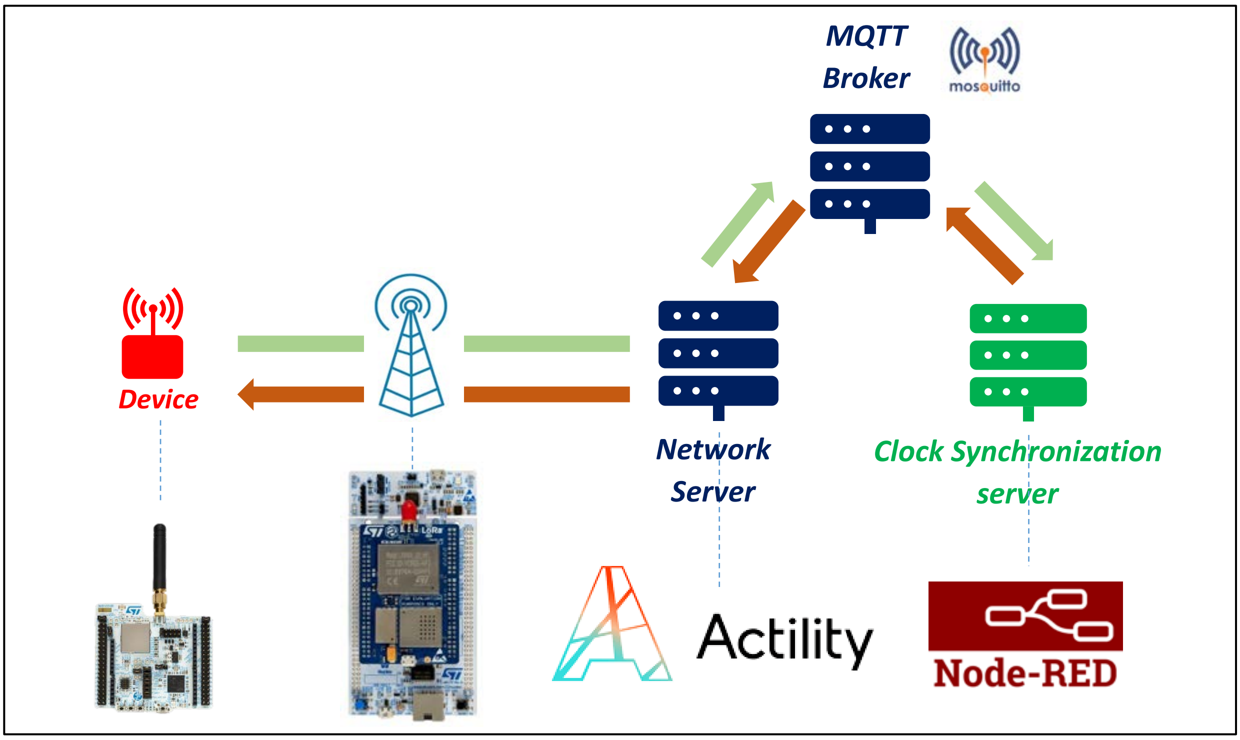


Figure 5: Demonstration infrastructure

Watch the video of this Clock Synchronization server: FUOTA videos playlist

2 Multicast

In some situations, we need to transmit the same packet to multiple end-devices. Sending a single frame targeting hundreds of assets is the main purpose of multicast. Multicast capabilities save bandwidth and reduce the transmission downlink delay that occurs when we reach a large fleet of end-devices.

2.1 Why use multicast?

2.1.1 Definition

Multicast means sending a single downlink frame to several end-devices at the same time. The end-device needs to be part of a group the server sends a unique frame for. An end-device can belong to up to four groups and the membership can be managed using LoRaWAN multicast commands. The figure below illustrates multicast in LoRaWAN with two groups.

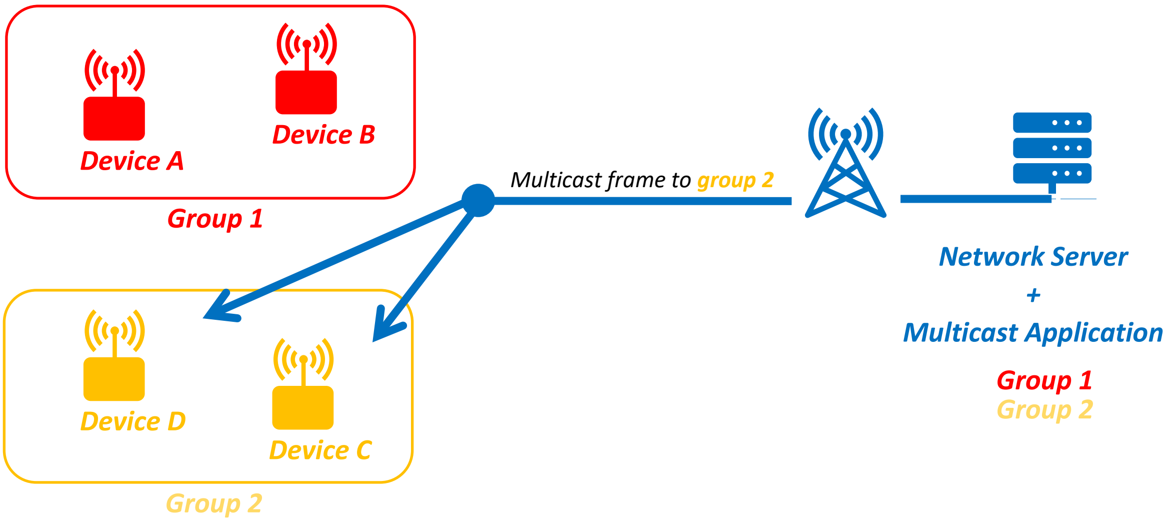


Figure 6: Multicast with two registered Multicast groups

Sending a multicast downlink frame requires that the end-device listens while the packet is transmitted. If end-devices are class C, this is not a problem. However, if end-devices are class A, they must be programmed to open a listening slot at a specific time. The server can achieve this by ordering a multicast distribution window to a group of end-devices. During this multicast window, end-devices are going to switch from class A to class C, or from class A to class B.

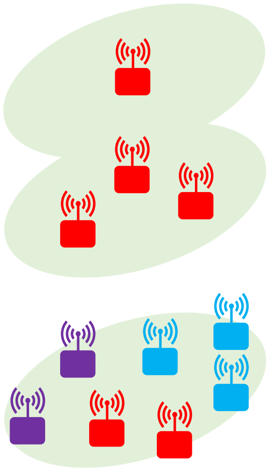
Multicast is a LoRaWAN native feature. All details are presented in the document: [LoRaWAN Remote Multicast Setup Specification].

Multicast is only possible if the end-device is class B or C ready.

2.1.2 Use cases

Multicast can be very helpful in many situations. We are going to see the relevance of multicast through the following example.

One to many:



In this example, there is one streetlight which can be requested to turn on or off the light. The user can send a downlink frame to this class C end-device and the end-device processes the corresponding action. The application works well.

Later on, the user decides to add a few more streetlights in the city. That means that if they want to change the streetlight's status, they must send one downlink frame to each of them. We need one frame for each end-device. The application works well.

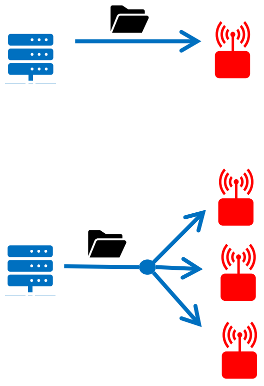
Now, the user wants to add 1.000 streetlights and create groups in order to obtain different behaviours depending on, for instance, the streetlight's geolocation. The user now needs to schedule one downlink for each end-device of the fleet. This is obviously a huge waste as most of the frames are redundant and LoRaWAN has very limited download capabilities.

A better solution for this kind of fleet management is **multicast**. With this LoRaWAN feature, the user manages their fleet in two steps:

1. Define the group to which the end-device belongs. That can be done before deployment with predefined group values, or remotely after deployment (add/delete/modify a group).
2. Send one downlink frame with the required value to a specific group.

FUOTA Context

LoRaWAN multicast can be used as a single feature. However, it can also be used with other features such as **FUOTA** (**F**irmware **U**pdate **O**ver **T**he **A**ir).



Launching a FUOTA campaign means updating the firmware of several devices already deployed in the field. Updating a firmware requires several kilobytes of data to be sent to each end-device. The new firmware is divided in as many blocks as necessary and sent to the end-device individually one after another. Of course, this requires a huge number of downlinks.

How can we manage thousands of end-devices?

Once again, sending messages with each end-device individually is not efficient, and multicast can be used to send the firmware update to a group of end-devices. This will optimize the FUOTA campaign process. The block fragmentation and the FUOTA application will be detailed later on in this course.

2.2 How to use multicast?

We are now going to present the conditions and steps to perform a successful multicast transmission.

2.2.1 End-device activation

There are two methods to activate a LoRaWAN end-device: OTAA or ABP. However, whatever the activation method, the result is the same: the end-device and the LoRaWAN server own a device profile composed of the DevAddr, the NwkSKey and the AppSkey. When a LoRaWAN frame reaches an end-device with the right information (DevAddr, NwkSKey and AppSkey) then it can authenticate and decrypt the payload.

However, when a LoRaWAN server sends a multicast frame to a group of devices, it does not use the device profile mentioned above. So, how can an end-device receive a multicast frame? The solution is to add a **multicast profile** in addition to the device profile. This multicast profile is composed of:

- a **McAddr** (Multicast group DevAddr)
- a **McNwkSKey** (Multicast group NwkSKey)
- a **McAppSKey** (Multicast group AppSKey)

Note: Mc stands for **M**ulti**c**ast.

In other words, a multicast group is like a classic ABP profile. When many end-devices own this same multicast profile, they represent a **multicast group**. If a LoRaWAN server transmits a downlink frame related to one multicast profile, then all end-devices in this group can receive, authenticate and decrypt the frame.

The multicast profile is only used for downlinks.

2.2.2 Multicast groups

Figure 7 represents an example of multicast profiles in end-devices and in the Network Server. The Network Server has three device profiles (X, Y and Z) and two multicast profiles (1 and 2). All end-devices have the multicast profile 1, but only X and Y have the multicast profile 2.

A single end-device can be involved in only four different multicast groups.

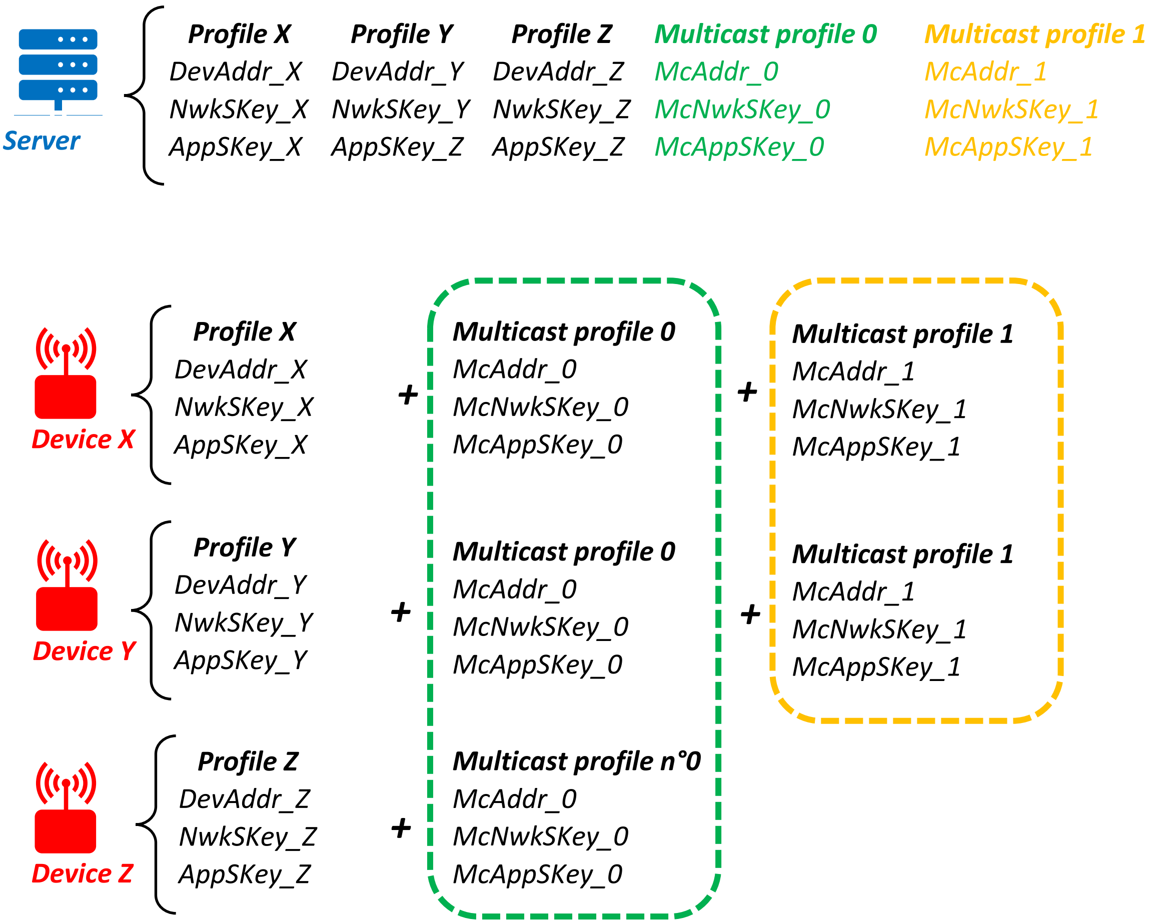


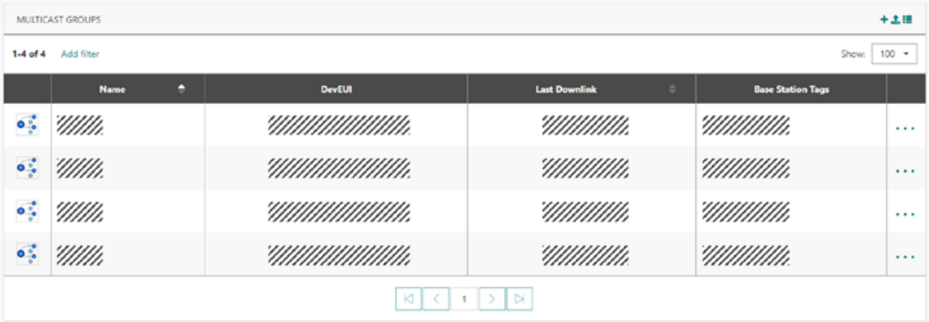
Figure 7: LoRaWAN multicast groups

The Network Server can send:

- Unicast frames to X, Y and Z.
- Multicast frames to the group X-Y-Z
- Multicast frames to the group X-Y

2.2.3 How to set up multicast groups?

To set up a successful multicast session, both the Network Server and the end-device must have the same multicast profile. On the server side, it is easy to add, modify or delete a multicast group. In Figure 8, we find the user interface management of multicast groups on Actility's Network Server (ThingPark).



MULTICAST GROUPS

1-4 of 4 Add filter Show: 100

	Name	DevEUI	Last Downlink	Base Station Tags	
	[redacted]	[redacted]	[redacted]	[redacted]	...
	[redacted]	[redacted]	[redacted]	[redacted]	...
	[redacted]	[redacted]	[redacted]	[redacted]	...
	[redacted]	[redacted]	[redacted]	[redacted]	...

Figure 8: Multicast management group with ThingPark Actility

We now need to set up the multicast groups on the end-device side.

If the end-device is already deployed without any multicast group registered, then this must be done remotely using **the Multicast commands** which are detailed in the LoRaWAN® Remote Multicast Setup Specification. Of course, the LoRaWAN server itself must be compliant with the LoRaWAN Remote Multicast Specification too.

These commands are special messages sent on LoRaWAN port 200 which is reserved for multicast group setup and management. This port must not be used for other purposes such as sending data frames (unicast or multicast).

- **Only devices that own a LoRaWAN stack with the Multicast Remote Package can receive multicast commands.**
- **Only LoRaWAN servers compliant with the Remote Multicast Specification can offer multicast features.**
- **Multicast commands (only commands, not messages) are sent on port 200.**

To set up a multicast group, the end-device needs three pieces of information: **McAddr**, **McNwkSKey**, and **McAppSKey**. To send this information to the end-device, the LoRaWAN server must use the **McGroupSetupReq** command. This command has a payload with two main parameters:

- **The McAddr**: This the DevAddr of the multicast group. Depending on your infrastructure, this address can be set up or randomly generated. See chapter 2.2.5 for more information.

- **The Multicast_Key:** This is a special key used by the end-device to generate the **McNwkSKey** and **McAppSKey.** The LoRaWAN server also uses the same Multicast_Key to generate the same McNwkSKey and McAppSKey. This process of key generation will be detailed later on.

Depending on your infrastructure, this **Multicast_Key** can be set up or randomly generated. See chapter 2.2.5 for more information.

Figure 9 recaps the multicast group creation.

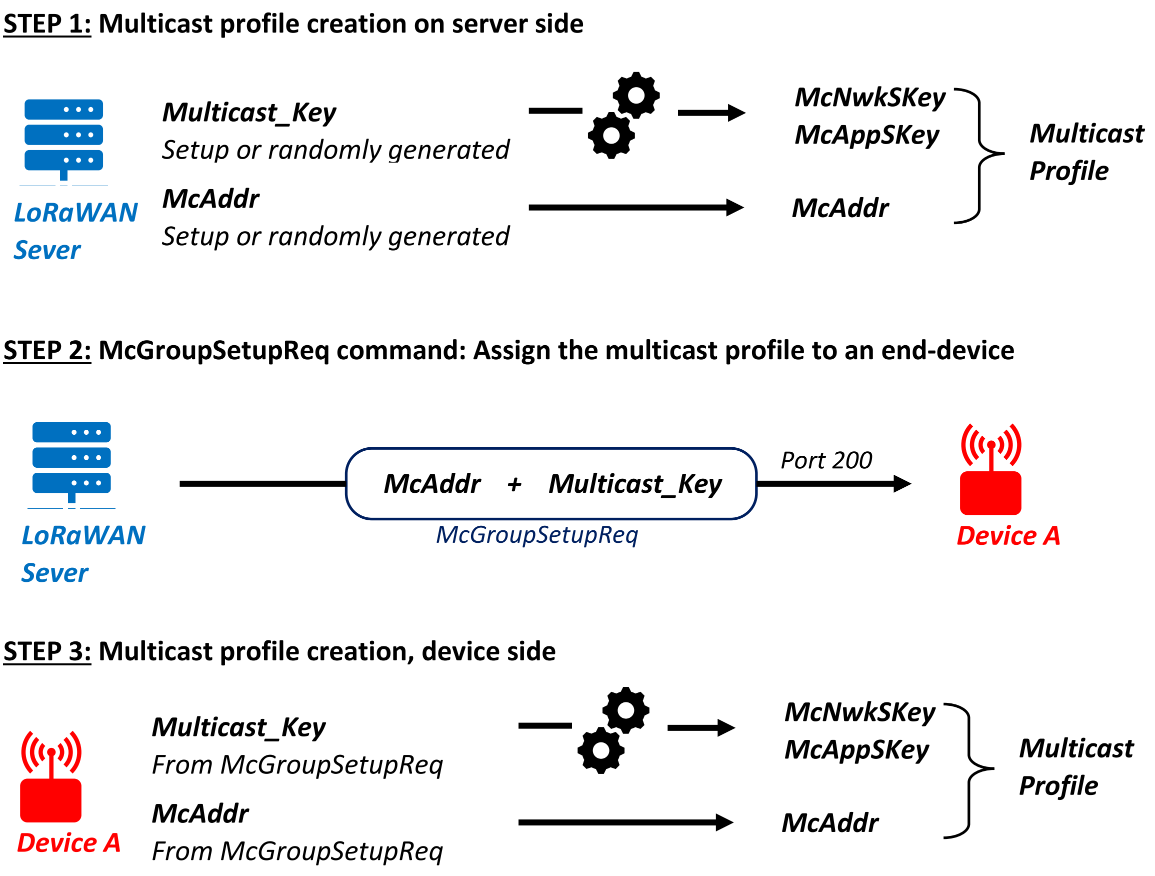


Figure 9: Main steps to set up a new multicast group

Now multicast groups are set up both on the end-device side and server side, and the LoRaWAN server can send multicast frames to the group.

- **The multicast keys generation and the multicast profile creation has been simplified. A more detailed presentation is available in chapter 2.2.5.**

2.2.4 How to send multicast frames?

The last step is to send a multicast frame to a group. However, before sending any multicast frames to a group of end-devices, the LoRaWAN server must be sure that end-devices are in class C (or B), ready to receive multicast downlinks.

There are different cases:

1. Class C end-devices: In this case, the LoRaWAN server can send a multicast frame whenever it wants to.
2. Class A end-devices: In this case, the LoRaWAN server must ask the end-device to toggle to another class (**McClassCSessionReq** for class C, **McClassBSessionReq** for class B). This command is sent to each member of the multicast group. The end-device will:
    - toggle to class C (or B) at a specific time. The end-device must be clock synchronized.
    - set its radio parameters in accordance with those chosen on the server side.
    - revert to class A after a specific timeout.

> Mind the radio parameters setup (frequency and Data Rate) of the multicast downlink frame. This must be the same in the end-device and on the server.

The very last step is to send a multicast frame to the group. To do so, it works like a classic unicast downlink where we can choose the port number and the payload. Figure 8 shows the user interface to send a multicast downlink frame to a group of end-devices in ThingPark.

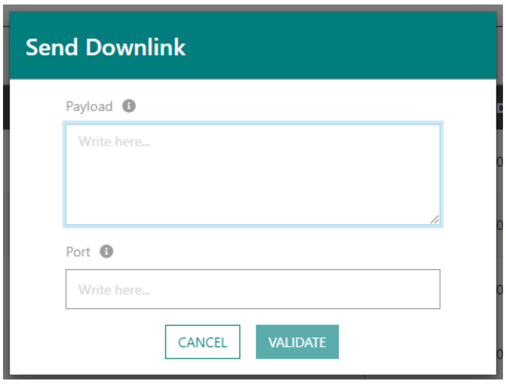


Figure 10: Multicast downlink with Actility's ThingPark

2.2.5 Multicast Keys

This section explains the multicast encryption process and key generation, though it is not important to understand the multicast process itself. We will also detail the multicast McGroupSetupReq command.

To create a multicast profile (on the server and end-device), a McAddr and a Multicast Key are needed. The McAddr is not confidential, so this information can be shared without encryption. However, the McKey used to generate the McNwkSKey and McAppSKey can't be shared without caution because it must not be exposed. This McKey is therefore encrypted. The overall process is presented in Figure 11 and is the same on the server and on the end-device.

The multicast keys derivation is detailed in the LoRaWAN® Remote Multicast Setup Specification.

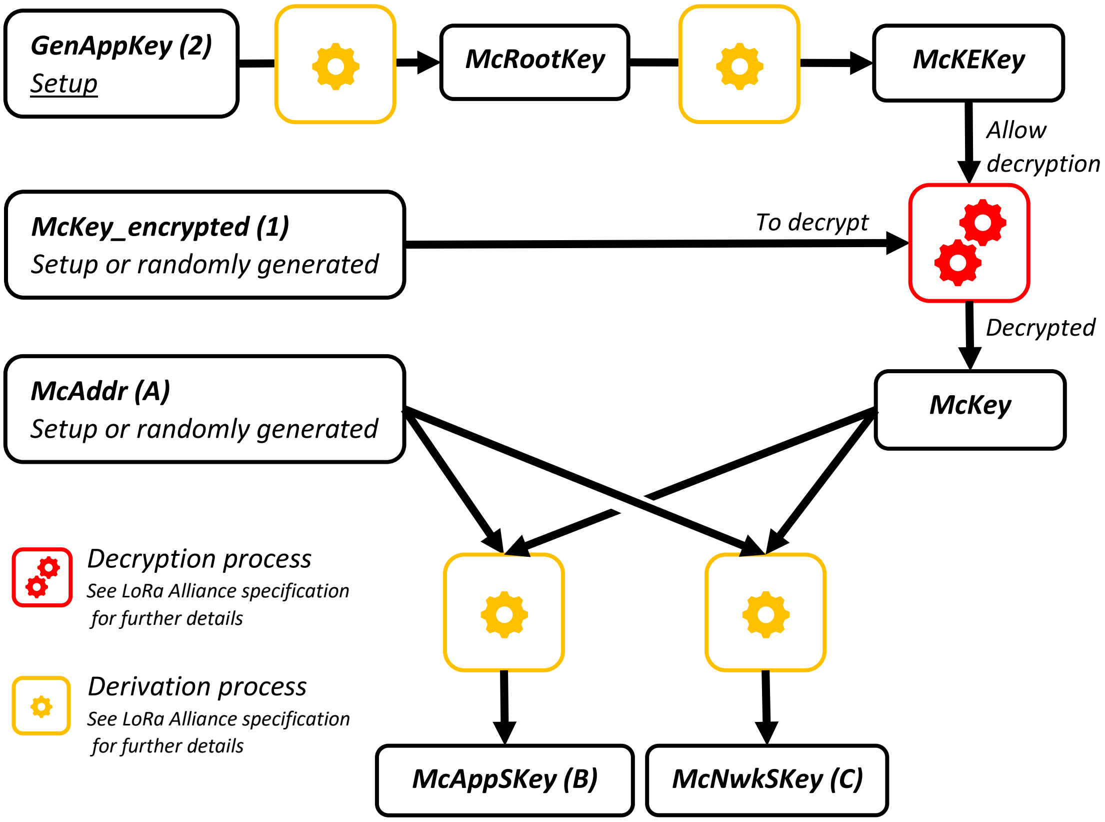


Figure 11: Schematic of the multicast keys generation process

How does it work on the server?

To create a multicast profile, we need to set up three elements: **McAddr (A)**, **McKey_encrypted (1)**, and **GenAppKey (2)**. **McKey_encrypted (1)**, and **GenAppKey (2)** are random values. GenAppKey (2) derivation will build the McEKey used for decrypting ***McKey_encrypted (1)*** and we will end up with the McKey. Thanks to this McKey, **McNwkSkey (B)** and **McAppSKey (C)** are generated. The server now has its full multicast profile: **McAddr (A)**, **McNwkSkey (B)** and **McAppSKey (C).**

How does it work on the end-device?

To create a multicast profile, we need to set up one element: **GenAppKey (2). GenAppKey (2)** must be the same random value used on the server. Then, the end-device waits for the **McGroupSetupReq** command. In this command, the end-device will find the McAddr (A) and the ***McKey_encrypted (1).*** GenAppKey (2) derivation will build the McEKey and will be used for decrypting ***McKey_encrypted (1).*** *We* will end up with the McKey. Thanks to this McKey, **McNwkSkey (B)** and **McAppSKey (C)** are generated. The server now has its full multicast profile: **McAddr (A)**, **McNwkSkey (B)** and **McAppSKey (C).**

- **GenAppKey must be set up in the end-device before deployment.**

2.2.6 The multicast server

The previous part dealing with multicast keys shows that the multicast group creation process is complex. Let's imagine a user wants to manage fleets of end-devices with various multicast groups. Managing them by hand is not efficient. A multicast server can help.

This tool is plugged into the Network Server and embeds features such as:

- Creation of multicast commands to send to the end-device
- Interpretation of multicast commands received from the end-device
- Multicast group management
- Multicast keys creation
- …

Figure 12 illustrates the position of the multicast server in the LoRaWAN ecosystem.

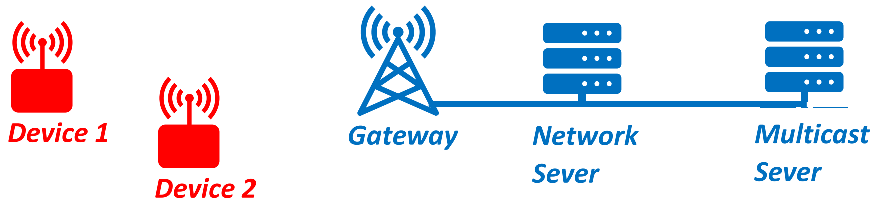


Figure 12: Multicast server

2.2.7 Example scenario

In this chapter, we are going to present an example with the following scenario:

There are two class-A end-devices (device 1 and 2) in the field. These end-devices:

- are already activated on their LoRaWAN network
- can switch to class C
- are multicast ready (they embed the Remote Multicast Package)

The device owner wants to send a <u>downlink frame to these devices at a specific time</u>.

- He uses a multicast server to manage its two end-devices fleet.
- Before end-device deployment, he set up the GenAppKey in the end-devices.

The infrastructure of this example is presented in Figure 13.

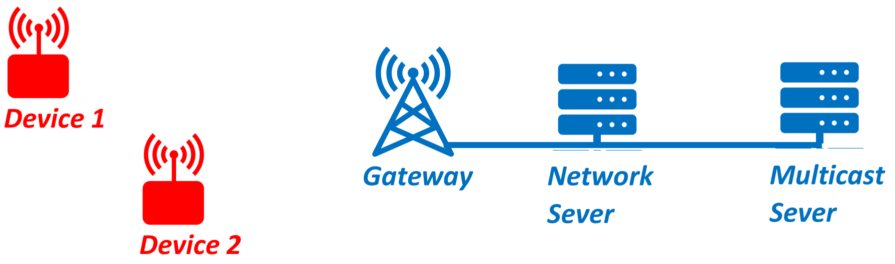


Figure 13: Example of multicast transmission

The following steps need to be followed to set up this scenario:

1. **The devices owner obtains the multicast keys thanks to the multicast server:**
    a. The device owner sets up the GenAppKey (the same as the one in the end-devices) in the multicast server.
    b. The multicast server randomly generates a McAddr and a McKey_encrypted.
    c. The multicast server generates McAppSKey and McNwkSKey (see Figure 11).

2. **The device owner creates a multicast profile in the Network Server:** He must use the McAddr, the McAppSKey, and McNwkSKey that the multicast server has created.

3. **The device owner sets up the multicast session thanks to the multicast server:**
    a. The device owner associates its two devices to the multicast group.
    b. The device owner sets the data (payload and port) he wants to send to the multicast group.
    c. The device owner sets the exact time he wants to send his data.

4. **The multicast server sets the multicast session:**
    a. The multicast server sends (through the LoRaWAN server) *McGroupSetupReq* command to device 1. Thanks to this command (and the GenAppKey the device already has), device 1 creates and sets up a multicast profile.
    b. The same command is sent to device 2.
    c. The multicast server checks if devices 1 or 2 are time-synchronized. If they are not, it synchronizes them. For more details about this topic, see chapter 1.
    d. The multicast server sends (through the Network Server) *McClassCSessionReq* command to device 1. Thanks to this command, the device knows when it must switch to class C to be ready to receive multicast downlink frames.
    e. The same command is sent to device 2.

5. **When the exact time for multicast session is reached:**
    a. Device 1 and 2 toggle themselves to class C.
    b. The multicast server sends (through the LoRaWAN server) the data the device's owner has set to the multicast group.
    c. Device 1 and 2 toggle back to class A after a timeout.

2.2.8 How to test the Remote Multicast feature?

To test the Remote Multicast feature, we need:

- An end-device with Remote Multicast package enabled
- A LoRaWAN® Gateway
- A LoRaWAN Network Server supporting multicast group management
- A Multicast server

For this demonstration, we will use:

- A STM32WL end-device with LoRaWAN® 1.0.3 and Multicast package enabled
- A LoRaWAN® Gateway
- ThingPark Community [https://community.thingpark.org]
- Our open-source Remote Multicast server (Node-RED):
[https://github.com/SylvainMontagny/fuota-server]

Our open-source multicast server works with any end-device, any gateway and any Network Server, as long as it provides the multicast group management feature.

- **Timestamped actions such as scheduled multicast sessions need the end-device to be on time. See part 1 to learn more about that.**

Depending on your Network Server, you might need to set up your own MQTT Broker to interface the Network Server (MQTT client) and the Clock Synchronization server (MQTT client). The figure below presents the overall infrastructure for the multicast demonstration.

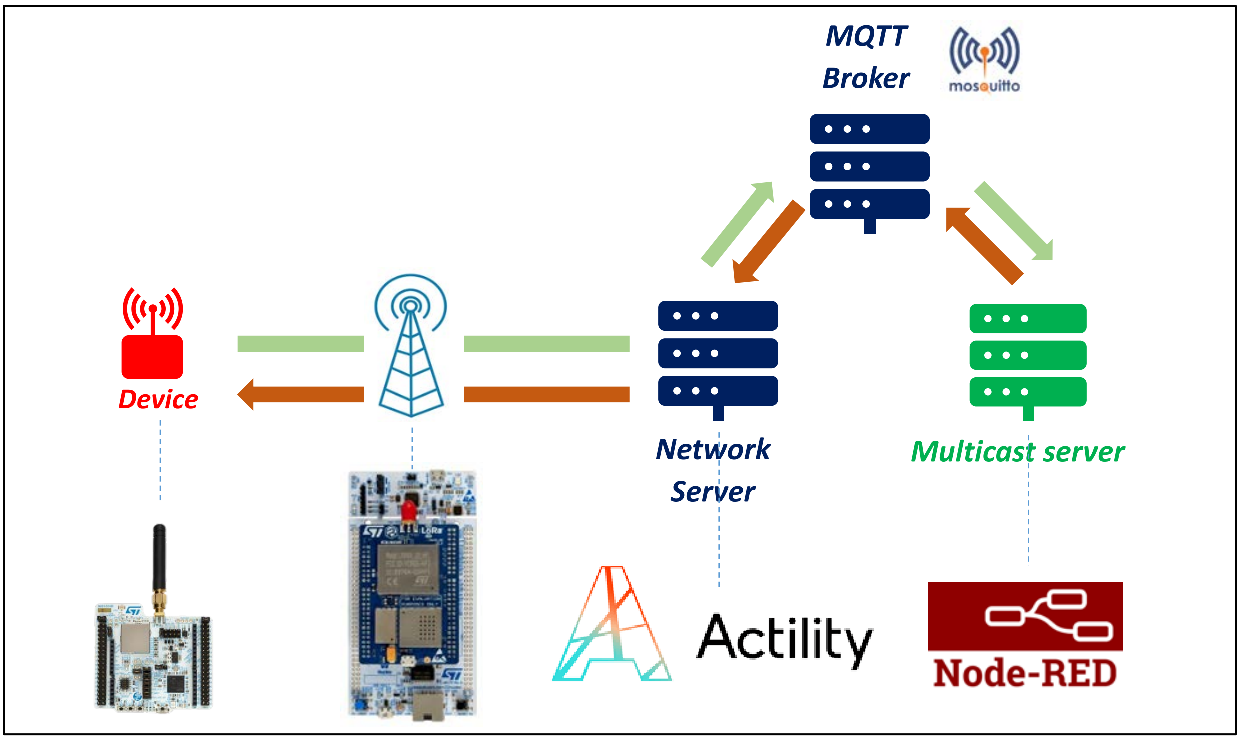


Figure 14: Demonstration infrastructure

Watch the video of this Remote Multicast server: FUOTA videos playlist

3 Data fragmentation

With LoRaWAN, the size of the data to be transmitted is small. However, a user might want to send a large data file. To do so, they can use a messaging package allowing them to send a data file whatever its size. That package is the **LoRaWAN Fragmented Data Block Transport**.

3.1 Sending a fragmented block of data: why?

3.1.1 Definition

Fragmentation in LoRaWAN means sending a large file over the LoRaWAN protocol. To do so, there is a 2-step process:

1- The large file is processed by a Fragmentation tool. The tool turns it into parts. The size of each part must respects the maximum payload size of the LoRaWAN frame.

2- Each part is sent over LoRaWAN protocol as a "classic" payload. Each part received is immediately reassembled with all other parts that already arrived. At the end of the transmission, the file is fully re-built.

3.1.2 Use cases

The most meaningful use case of data Fragmentation is **FUOTA** (**F**irmware **U**pdate **O**ver **T**he **A**ir). Launching a FUOTA campaign means updating the firmware of one or several end-devices already deployed in the field. Updating firmware requires several kilobytes of data to be sent to each end-device. In that case, the LoRaWAN Fragmented Data Block Transport process is a solution.

To illustrate the relationship between the fragmentation process and the FUOTA campaign, here is a diagram.

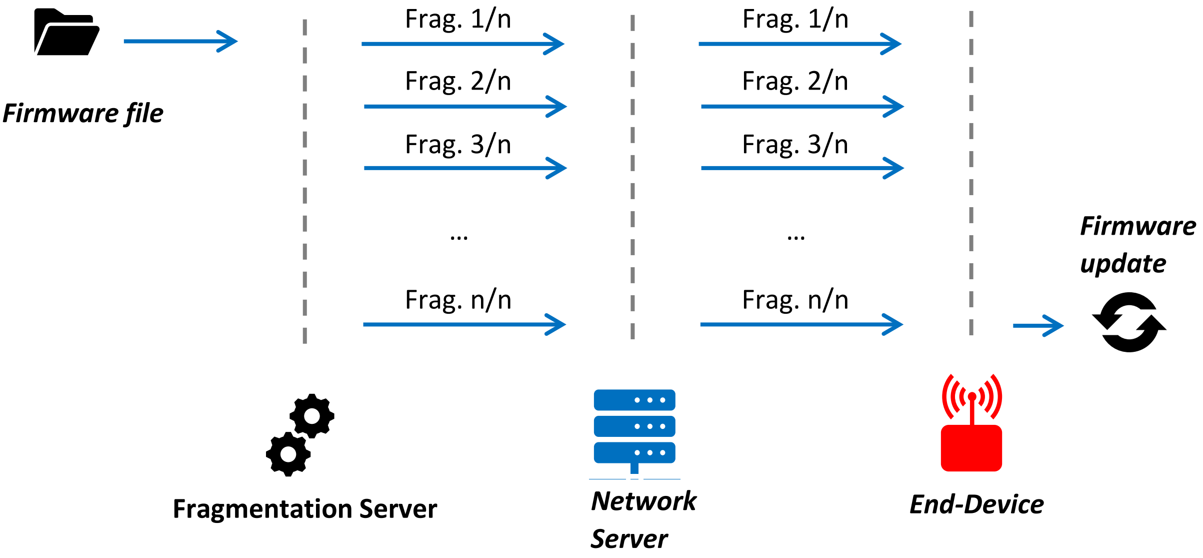


Figure 15: Fragmentation session during a firmware update (FUOTA)

3.2 How to send data via fragmented data block transport process?

3.2.1 The process

Like the other packages (Clock Synchronization, Remote Multicast) presented earlier, the data fragmentation works with commands. There are two main steps in the process:

- Step 1: Creation of fragmentation sessions
- Step 2: Sending fragments.

These commands are received and sent on the dedicated port 201.

- **Only devices that own a LoRaWAN stack with the Data Fragmentation Package can receive Fragmentation commands.**
- **Data Fragmentation commands (commands and fragments) are sent on port 201.**

3.2.1.1 Step 1: From data to fragments

The first one aims to split the binary file into fragments (also called "uncoded fragments"). With these fragments, there are redundancy fragments (also called "coded fragments"). Those special ones are sent in addition to the "normal" fragments such that the end-device can reconstruct the whole file even if it missed one (or more) fragment(s). See chapter 3.2.2 for further explanations.
The number of redundancy fragments added to the session depends on the end-device’s memory setup. Most of the time, the number of redundancy fragments allowed is a tiny percentage of the number of normal fragments (e.g.: 10%).

Another important parameter of the fragmentation process is the fragment size. It directly impacts the session duration and the network load. The fragment size depends on two things:

- The **radio parameters** used to send the downlinks.
  e.g.: with SF9 and BW 125kHz the maximum payload size is 115 bytes.
- The **composition of the *DataFragment* command**. That is the command that embeds the fragment. It is built as follows (see LoRaWAN Fragmented Data Block Transport Specification for further details):

CID (0x08)	Command header	Fragment
Size: 1 byte	Size: 2 bytes	Size: maximum payload size – 3 bytes

e.g.: with a maximum payload size of 115 bytes (SF9, BW 125kHz), the maximum size of a fragment is 112-bytes long.

The fragmentation session duration depends on fragment parameters (size and number), the radio settings, and the downlink policy of the Network (duty-cycle, fair policy).

3.2.1.2 Step 2: The Fragmentation session

The second step aims to send the fragments to the end-device. The session is divided into 3 parts: the session setup, sending fragments, and the end of the session.

- **The session setup**

  Before sending any fragment (with *DataFragment* command), the session shall be set up. To do so, the *FragSessionSetupReq* command is sent to the end-device. It sets up several parameters such as: the fragment size, the number of fragments, or the session number. To be noted: an end-device can handle up to 4 fragmentation sessions at a time.

  A Fragmentation session can be run through a multicast context. To do so, the *FragSessionSetupReq* command specifies which multicast group is allowed to receive the fragments of the session.

- **Sending fragments**

  When the session has been successfully set up, each fragment and redundancy fragment is packed into *DataFragment* commands. Then these commands are sent one by one to the end-device.

- **The end of the session**

  If no fragment has been lost during the session, the end-device can reconstruct the data block as soon as it receives the nth fragment over n.
  However, if some fragments have been lost, the end-device shall wait for redundancy fragments. As soon as it has received enough redundancy fragments to compensate for fragments losses, the end-device can reconstruct the data block.

  To be noted: in version 2 of the Data Fragmentation package, there is a dedicated command to signal the Fragmentation server that the data block has been fully reconstructed. However, version 1 of the package doesn’t implement it. To stop the fragmentation session, a proprietary protocol shall be added (e.g.: an uplink on a user port).

An end-device can handle 4 different fragmentation sessions at a time.

3.2.2 Fragments and redundancy fragments

Fragments are pieces of a data block. When they are all received by the end-device, the latter is able to rebuild the initial data block. But what if one packet or more is lost? As said earlier, the end-device can still reconstruct the data block thanks to redundancy fragments. How does it work?

To create fragments and redundancy fragments from an initial data block, an algorithm can be used: the **Forward Error Correction (FEC)** algorithm. It also shall be used on the end-device side to allow it to autonomously recover a file even if fragments were lost.

This document will not explain in detail the FEC algorithm. To get more information about that, see LoRaWAN Fragmented Data Block Transport Specification.

- To build normal fragments, the file is simply split into parts according to the desired fragment size.

- To build the redundancy fragment and, at the end, use them to recover a lost fragment, the FEC algorithm executes **XOR operation between different "normal" fragments**. Let's take an example to illustrate this.

Let's assume 2 fragments of 1-byte size:

```
      Fragment 1 =    0xAE =  0b10101110
      Fragment 2 =    0x57 =  0b01010111
```

A redundancy fragment is built:

```
      Redundancy 1  =        Fragment 1 XOR Fragment 2
                             0b10101110
                        XOR  0b01010111
                             -----------------
      Redundancy 1  =        0b11111001
      Redundancy 1  =        0xF9
```

First situation: we consider that **Fragment 1** is lost. Let's recover it thanks to **Redundancy 1**.

```
      Rebuilt Fragment 1 =   Redundancy 1 XOR Fragment 2
      Rebuilt Fragment 1 =   0b11111001
                      XOR    0b01010111
                             -----------------
                             0b10101110
      Rebuilt Fragment 1 =   0xAE
      Rebuilt Fragment 1 =   Fragment 1
```

Second situation: we consider that **Fragment 2** is lost. Let's recover it thanks to **Redundancy 1**

```
      Rebuilt Fragment 2 =   Redundancy 1 XOR Fragment 1
      Rebuilt Fragment 2 =   0b11111001
                      XOR    0b10101110
                             -----------------
                             0b01010111
      Rebuilt Fragment 2 =   0x57
      Rebuilt Fragment 2 =   Fragment 2
```

To create fragments and redundancy fragments, a fragmentation tool is needed. Further explanations about this are proposed in the chapter 3.2.3.

3.2.3 Fragmentation server

Preparing a fragmentation session and running it are complex tasks. To manage a successful fragmentation session, a tool is needed: the fragmentation server.

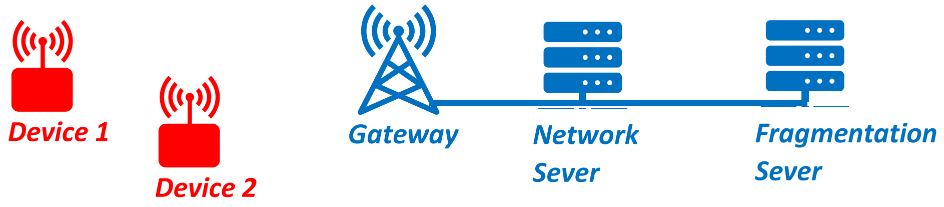


Figure 16: Fragmentation Server

This one can embed several features such as:

- FEC algorithm to create fragments and needed redundancy
- Session setup
- Frames sending
- Management of the session (progression, remaining time, end-of-session handling)

3.2.4 Example scenario

To sum up the fragmentation process, let's deal with a simple scenario:

A device owner wants to send a 9-bytes data block. He wants to split it into 3 fragments with 33% redundancy. Its end-device is activated and works in class-C. We assume that the end-device doesn't lose any fragment as it is in perfect network coverage conditions. The end-device implements version 1 of the Data Fragmentation package.

Here are the steps of the scenario:

1. **The device owner sets up the session.**
   a. He uses its Fragmentation server. He uploads its data file. Then, the tool gives him 3 fragments and 1 redundancy fragment.
   b. Thanks to the Fragmentation Server, he sends the *FragSessionSetupReq* command to the end-device. When the end-device receives the command, it prepares the session.
2. **The device owner runs the session.**
   a. He launches the session. The Fragmentation server sends the *DataFragment* commands that embeds the fragment one by one.
   b. When the 3rd *DataFragment* command is received, the whole data block is rebuilt on the end-device side. The end-device sends an applicative uplink to signal the Fragmentation Server that it can stop sending fragments.

c. The Fragmentation server receives the "end-of-session" uplink. Then, it doesn't send the redundancy fragment.

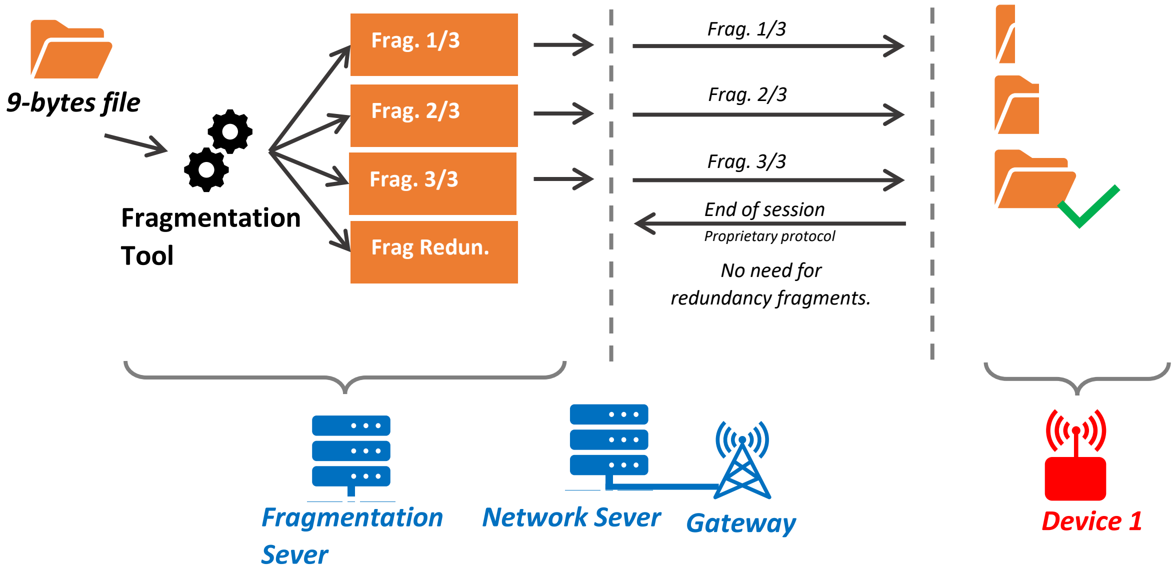


Figure 17: illustration of the scenario

3.2.5 How to test the Data fragmentation feature?

To test the Data Fragmentation feature, we need:

- An end-device with Fragmentation package enabled
- A LoRaWAN Gateway
- A LoRaWAN Network Server
- A Fragmentation tool (to turn files into fragments)
- A Fragmentation server

For this demonstration, we will use:

- A STM32WL end-device with LoRaWAN® 1.0.3 and Fragmentation package enabled
- A LoRaWAN® Gateway
- ThingPark Community [https://community.thingpark.org]
- A Python tool to get the fragments
- Our open-source Remote Multicast server (Node-RED): [https://github.com/SylvainMontagny/fuota-server]

> Our open-source fragmentation server works with any end-device, any gateway and any Network Server.

Depending on your Network Server, you might need to set up your own MQTT Broker to interface the Network Server (MQTT client) and the Fragmentation server (MQTT client). The figure below presents the overall infrastructure for the fragmentation demonstration.

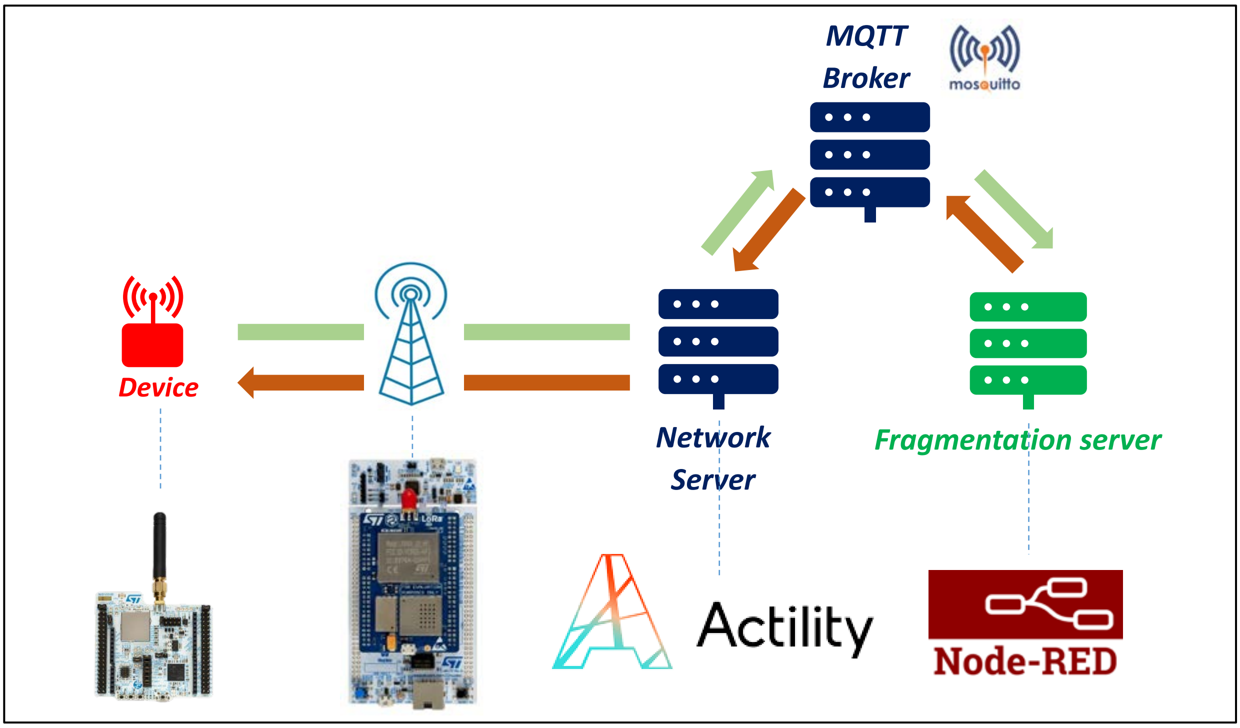


Figure 18: Demonstration infrastructure

Watch the video of this Fragmentation server: FUOTA videos playlist

4 FUOTA

FUOTA stands for **F**irmware **U**pdate **O**ver **T**he **A**ir. This process allows the firmware of an end-device to be updated without any physical intervention. Let's dive into this feature.

4.1 FUOTA: what does that mean?

4.1.1 Definition

The FUOTA process allows a user to send a firmware image to an end-device (or a fleet) and update the current firmware remotely.

To do so, there are some prerequisites:

- Communication features
  - Dedicated layer messaging packages are needed to perform FUOTA. See part 4.2.1.
  - The end-device shall implement the class-C or B (for FUOTA with Multicast)
  - Sending a firmware over the air is a critical operation. Authentication, confidentiality, and integrity are needed.
- End-device update capabilities
  - The end-device shall implement firmware update capabilities **before deployment**.
  - The end-device shall have enough memory to save one or more firmware images.

Here are the main steps in a FUOTA process:

1. The end-device owner gives a firmware with FUOTA capabilities to its end-device.
2. The owner deploys the device in the field.
3. For some reason, the owner wants to update their end-device firmware. He creates the new firmware image and sends it through the FUOTA campaign.

> Limitation: size of the firmware to be sent cannot exceed the **end-device memory slot size** that will host the firmware image.

4.1.2 Use cases

A FUOTA use case could be the following scenario: A beekeeper owns dozens of connected beehives. He retrieves data from its beehives such as the weight and the temperature. Now, he wishes to have the humidity of his beehives. The humidity sensor is already embedded on the hardware board, but the software doesn't use it. As its end-device firmware is a FUOTA ready firmware, the beekeeper doesn't need to change the firmware of each connected beehive manually. He can create the new firmware that suits him. Then, he can update all its connected beehives at the same time with a FUOTA campaign.

4.2 How to perform a FUOTA session?

4.2.1 The need of other packages

As said earlier, a FUOTA campaign needs special communication features. Those are enabled by the layer messaging packages already presented in this book:

- Clock Synchronization: It allows a user to clock-synchronize an end-device. An end-device shall be on time to perform scheduled actions such as opening a multicast session.
- Remote Multicast: It allows a user to send one frame to a fleet of end-device at the same time. To update over the air a whole end-devices fleet, the new firmware image shall be sent to all end-devices at a time.
- Data Fragmentation: It allows a user to send a large data file to an end-device. To send a new firmware image (that is a large data file) to an end-device fleet, a data fragmentation process is used.

Moreover, to perform a successful FUOTA campaign, a last package is needed:

- Firmware management: This package offers commands to manage the images the end-device owns. Thanks to this package, a user can check if an image is ready to be installed, delete an image, or program a reboot to trigger the firmware update with the new image. See Firmware Management Protocol Specification for further details.

- **Only devices that own a LoRaWAN stack with the Firmware Management Package can receive Firmware Management commands.**
- **Firmware Management commands are sent on port 203.**

4.2.2 End-device bootloader capability

In addition to all the needed packages previously listed, the end-device shall implement firmware update capabilities. It shall be able to get a firmware image sent through the LoRaWAN layer messaging package and store it in a dedicated memory slot.

When a reboot is initiated (e.g.: by a reboot command from the firmware management package), the end-device shall check if a new firmware image is available. If a new one is present, it shall reboot with the new firmware version.

4.2.3 The process

Let's deal with the following situation: a user wants to update the firmware of its class-A end-devices fleet. Here is the schematic of the exchanged information with only one end-device.

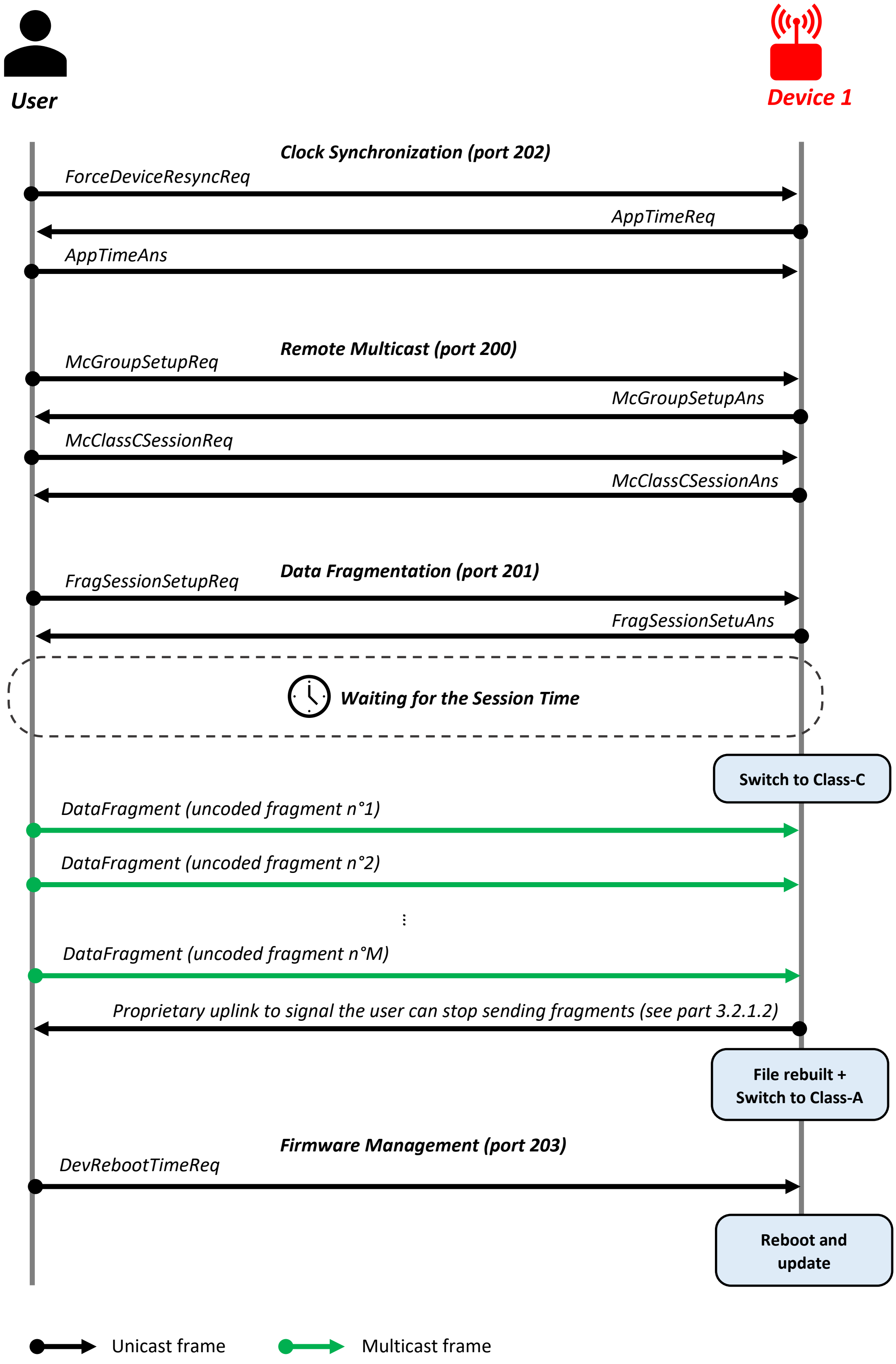

User
Device 1
Clock Synchronization (port 202)
ForceDeviceResyncReq
AppTimeReq
AppTimeAns
Remote Multicast (port 200)
McGroupSetupReq
McGroupSetupAns
McClassCSessionReq
McClassCSessionAns
Data Fragmentation (port 201)
FragSessionSetupReq
FragSessionSetuAns
Waiting for the Session Time
Switch to Class-C
DataFragment (uncoded fragment n°1)
DataFragment (uncoded fragment n°2)
⋮
DataFragment (uncoded fragment n°M)
Proprietary uplink to signal the user can stop sending fragments (see part 3.2.1.2)
File rebuilt + Switch to Class-A
Firmware Management (port 203)
DevRebootTimeReq
Reboot and update
Unicast frame
Multicast frame

Figure 19: FUOTA Campaign

Additional information about the process presented above:

- The Clock Synchronization part is not necessary. Indeed, the end-device may already be on time (because it uses another time source, or it already has been synchronized).
- In the Firmware Management part, *DevRebootTimeReq* command is used. We could have used *DevRebootCountdownReq* command.
- In this process, we assume that all frames are received: there are no coded fragments (redundancy).

4.2.4 The FUOTA Server

Running a FUOTA campaign, is a complex task. As we saw previously, different features are needed. To manage all of them at once, we need is a tool: the FUOTA server.

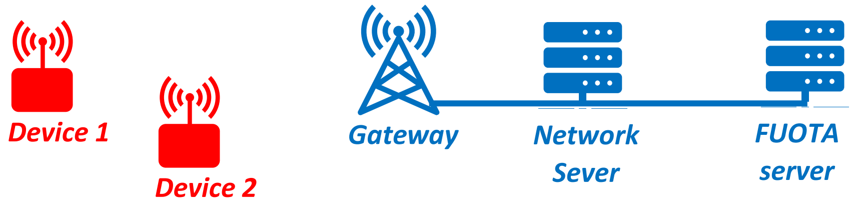


Figure 20: FUOTA Server

This one embeds several features such as:

- Clock Synchronization
- Multicast
- Fragmentation
- Firmware Management

4.2.5 Example Scenario

To sum up the FUOTA process. Let's deal with a simple scenario:

A device owner wants to update an end-device. It is already deployed. It is activated and works in class-C. During the fragmentation session, we assume that the end-device doesn't lose any fragment as it is in perfect network coverage conditions. The end-device implements the version 1 of the Data Fragmentation package.

Here are the steps of the scenario:

1. **The device owner prepares the session**
   a. As a first step, he creates the new firmware image to send. This step depends on the end-device he works with. To create an image to be sent via FUOTA, he shall follow its end-device maker recommendations.
   b. With a fragmentation tool (that uses FEC algorithm), the device owner fragments the firmware image.

2. **The device owner sends the image through a fragmentation session (see part 3.2.4)**
3. **The device owner launches the end-device update process**
   a. He sends a reboot command of the firmware management package (DevRebootTimeReq command or DevRebootCountdownReq command) to the end-device.
   b. When the end-device received the reboot command, it initiates the reboot at the mentionned time.
   c. When the end-device reboots, it swaps its current image with the new one.

4.2.6 How to test FUOTA feature?

To test the FUOTA feature, we need:

- An end-device with Fragmentation and Firmware Management packages enabled
- A LoRaWAN Gateway
- A LoRaWAN Network
- A Fragmentation tool (to turn files into fragments)
- A Fragmentation server and a Firmware Management server

For this demonstration, we will use:

- A STM32WL end-device with LoRaWAN® 1.0.3 and both Fragmentation and Firmware Management packages enabled
- A LoRaWAN Gateway
- A Python tool to get the fragments
- ThingPark Community [https://community.thingpark.org]
- Our open-source Fragmentation and Firmware Management server (Node-RED):
  [https://github.com/SylvainMontagny/fuota-server]

Our open-source FUOTA and Fragmentation servers work with any end-device, any gateway and any Network Server.

- **Timestamped actions such as scheduled reboots need the end-device to be on time. See part 1 to learn more about that.**

Depending on your Network Server, you might need to set up your own MQTT Broker to interface the Network Server (MQTT client) and the Clock Synchronization server (MQTT client). The figure below presents the overall infrastructure for the FUOTA demonstration.

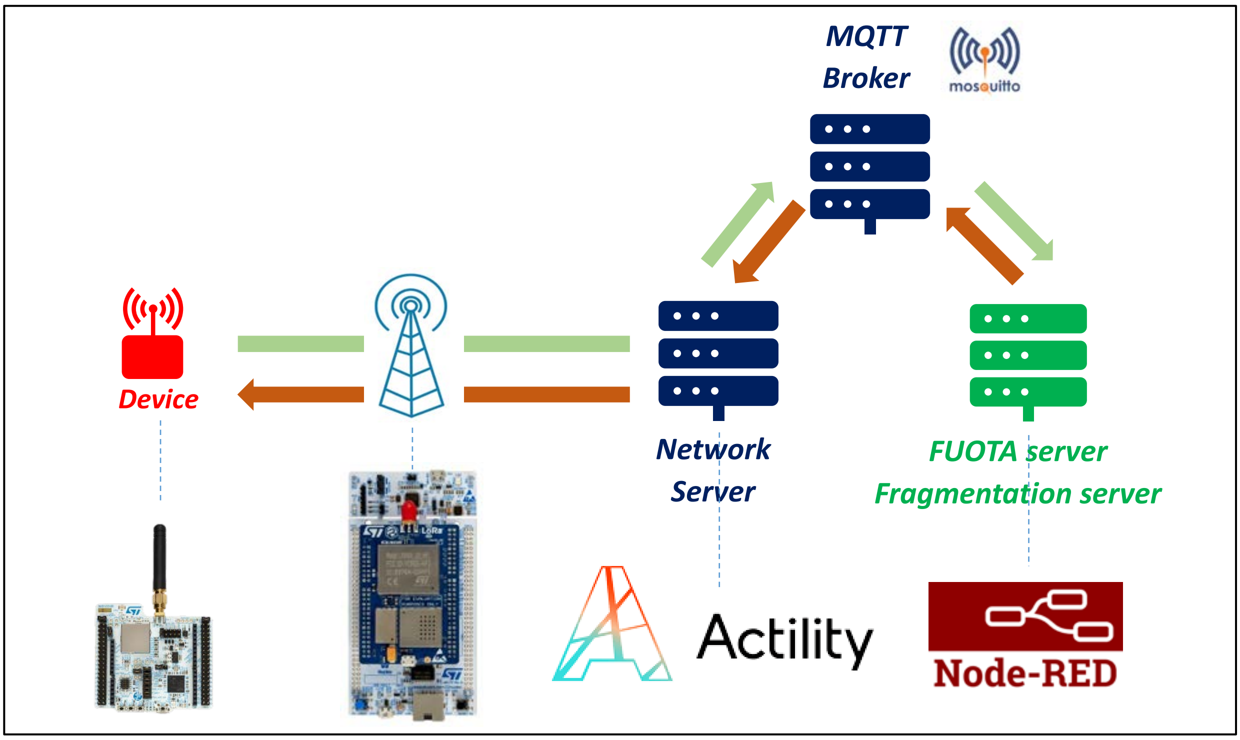


Figure 21: Demonstration infrastructure

Watch the video of the FUOTA demo: [FUOTA videos playlist]

5 The Join Server

5.1 The role of the Join Server

We know that the LoRaWAN 1.0.x protocol requires two keys to work:

- The NwSKey for authentication
- The AppSKey for encryption

The ABP activation mode provides these keys directly to the end-device. Though this simplifies the process it is not the most secure way to provision the end-device. The recommended activation mode is therefore OTAA so that a new set of NwSKey and AppSKey is generated at each Join-Request. But who exactly manages this Join-Request?

Until now, we assumed that this activation phase was handled by the Network Server. In reality, the activation is handled by a specific entity called the Join Server. The Join Server is directly connected to both the Network Server and the Application Server and has a unique 8-byte identifier (EUI) called the JoinEUI that specifies which Join Server will be used to process the activation procedure.

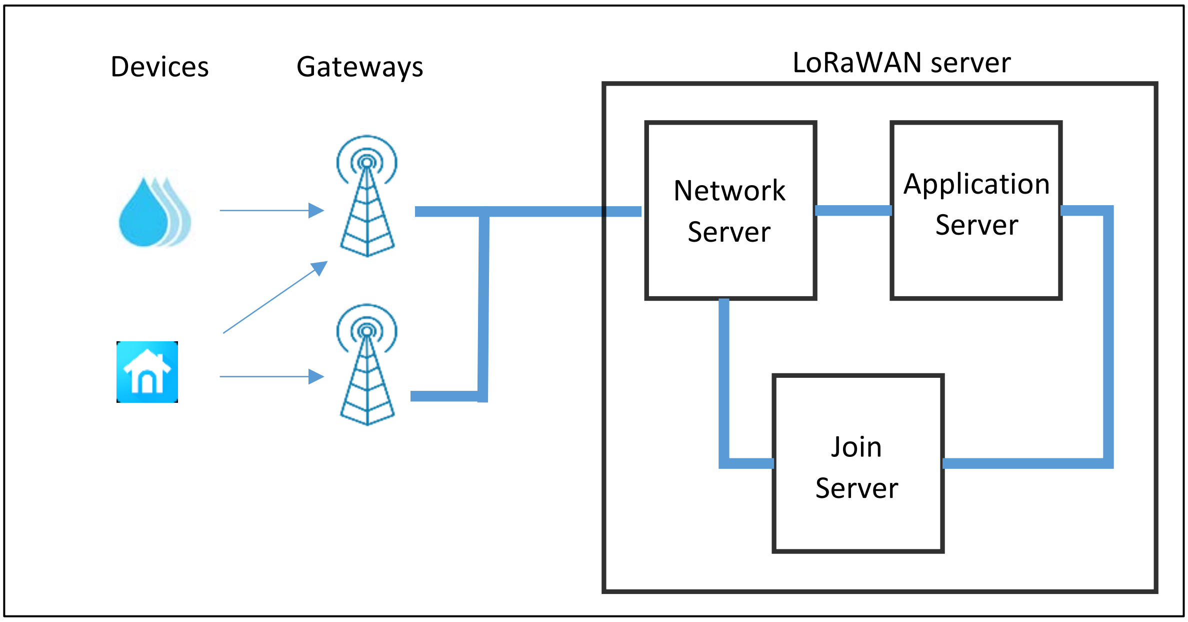
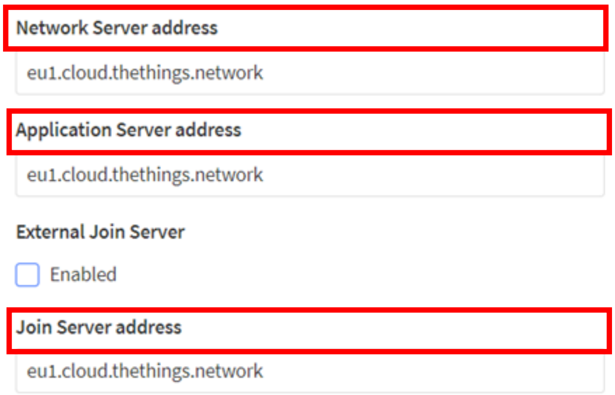


Figure 22: The Join Server

These three entities can often be found in the same location as the Network Server often plays the Join Server role. But, the purpose is to have them on separates instances. For example, TTN gives you the option of choosing for each end-device where the associated Application Server and Join Server are located.



Network Server address

eu1.cloud.thethings.network

Application Server address

eu1.cloud.thethings.network

External Join Server

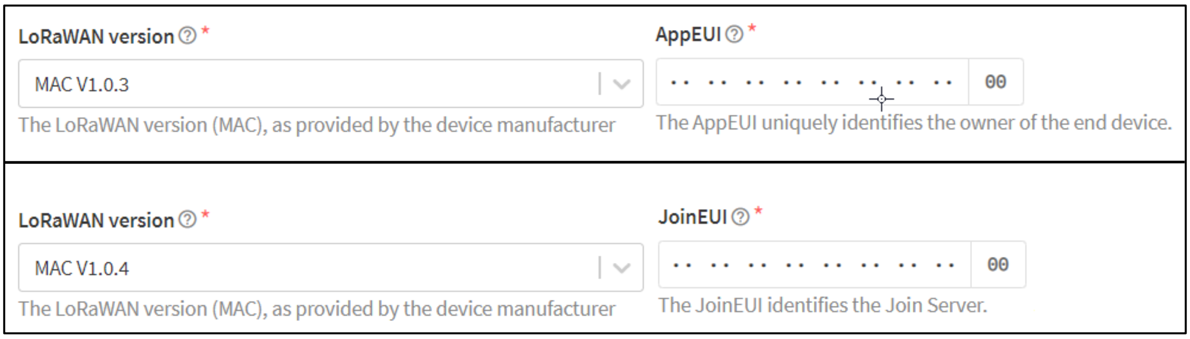
□ Enabled

Join Server address

eu1.cloud.thethings.network

Figure 23: Location of the Network Server, Application Server and Join Server on TTN

As of version 1.0.4 of the LoRaWAN standard, the JoinEUI replaces the AppEUI. We therefore have to fill out different fields, regardless of the LoRaWAN version we select. In the first line in Figure 24, we use LoRaWAN version 1.0.3, hence the AppEUI field. In the second line, we use LoRaWAN version 1.0.4, hence the JoinEUI field.



LoRaWAN version ⓘ *

MAC V1.0.3

The LoRaWAN version (MAC), as provided by the device manufacturer

AppEUI ⓘ *

·· ·· ·· ·· ·· ·· ·· ·· 00

The AppEUI uniquely identifies the owner of the end device.

LoRaWAN version ⓘ *

MAC V1.0.4

The LoRaWAN version (MAC), as provided by the device manufacturer

JoinEUI ⓘ *

·· ·· ·· ·· ·· ·· ·· ·· 00

The JoinEUI identifies the Join Server.

Figure 24: AppEUI or JoinEUI choice depending on the protocol version

You must know the LoRaWAN version of your end-device as you need to specify it during the end-device registration.

Your Network Server, Application Server and Join Server can be offered by different operators. We can imagine having an Actility Network Server, The Things Industries Join Server, and Common Sense IoT platform.

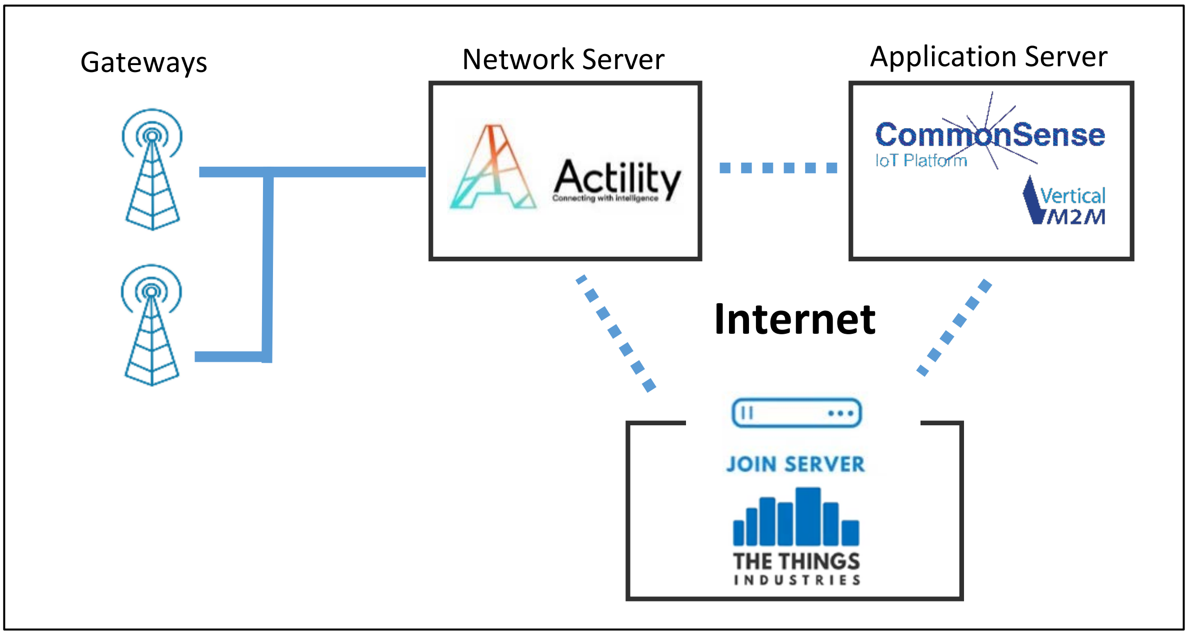


Figure 25: Network Server – Join Server – Application Server

5.2 The activation procedure in OTAA

In the ebook presented the join procedure. The Join-Request frame is composed of a JoinEUI, DevEUI and a DevNonce as shown in Figure 26.

Size (octets)	8	8	2
Join-Request payload	JoinEUI	DevEUI	DevNonce

Figure 26: Join-Request frame

The DevNonce is an important value for the Join-Request because it protects from replay attacks. In the first ebook (LoRa-LoRaWAN and IoT for beginners), we saw exactly the same situation with uplink frames, and we used frame counter to prevent replay attack. Here, we use DevNonce. DevNonce can't be used twice.

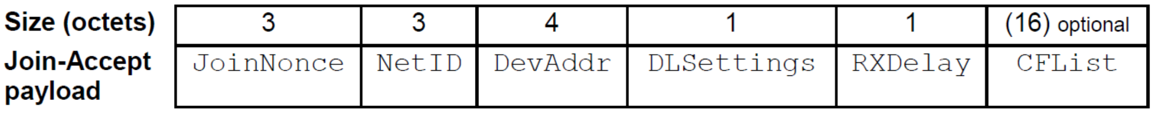
In cryptography, a nonce is an abbreviation for "number only used once.

The maximum number of Join-Requests is therefore 65,536, which should never happen because a LoRaWAN end-device rarely generates Join-Requests. The LoRaWAN specification proposes a different management of the DevNonce depending on the LoRaWAN standard we are working with:

- **Until LoRaWAN version 1.0.3**, the Join Server accepted the Join-Request only if the DevNonce value had never been previously used. Up to LoRaWAN version 1.0.3, the LoRaWAN end-device would randomly choose a DevNonce among the 65,536 available. If the end-device had not saved the values used previously for the DevNonce, at the next Join-Request it would generate a new random value with many chances to find a value that had never been used.
- **From LoRaWAN version 1.0.4 onwards**, the Join Server accepts the Join-Request only if the value used is higher than the one used previously. Since LoRaWAN version 1.0.4, the end-

device uses an incrementing counter for the DevNonce. If the end-device reboots, it must keep the last value of the DevNonce in a non-volatile memory. Otherwise, the Join-Request will be discarded.

When the Join-Request is accepted, the Join Server returns a Join-Accept with the information shown in Figure 27. Among these are the DevAddr, network configuration information, as well as everything the end-device needs to generate the NwkSKey and AppSKey on its side.



Size (octets)	3	3	4	1	1	(16) optional
Join-Accept payload	JoinNonce	NetID	DevAddr	DLSettings	RXDelay	CFList

Figure 27: Format of a Join-Accept frame

The Join Server purpose is to:

- Securely store the root keys (AppKey).
- Respond to the Join-Request issued by the authorized end-devices.
- Store and securely transmit the generated NwSKey to the Network Server.
- Store and securely transmit the generated AppSKey to the Application Server.

The Join Server contains the following information for each end-device under its control:

- DevEUI
- AppKey
- Network Server Identifier
- Application Server identifier
- LoRaWAN version of the end-device

One may wonder what the point is of such a structure since the management of keys is simply deported to another server. To answer this question, we need to understand that the Join Server is not supposed to be an internal service connected to a single Network Server and Application Server. Figure 28 presents the multiple possible connections between them.

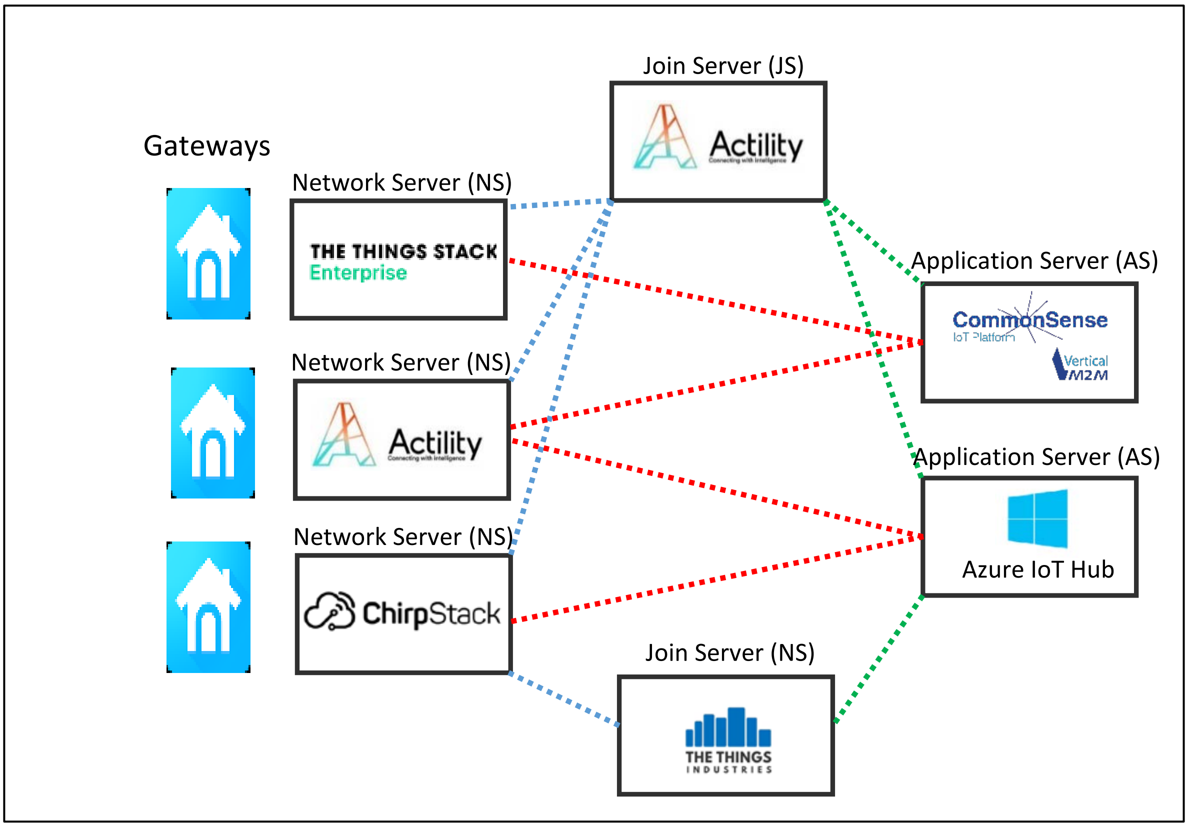


Figure 28: Network Server, Join Server, and Application Server connections

- A Network Server (NS) can be connected to many Join Server (JS). For example, ChirpStack can connect to Actility's Join Server and TTI's Join Server. Actility Network Server knows which JS to request thanks to the JoinEUI sent by the end-device during the Join procedure. An NS can also connect to many AS.
- A JS can connect many NS for the exchange of the NwSKey and many AS to exchange the AppSKey.

Such a structure has many benefits from a point of view of security and interoperability of devices within different networks.

5.3 Device key provisioning

When the OTAA activation process runs, we assume that the root keys are already stored in the end-device on one hand and in the Join Server on the other hand. For that purpose, the end-device maker, the end-user and the Join Server must exchange information before selling the end-device as shown in Figure 29.

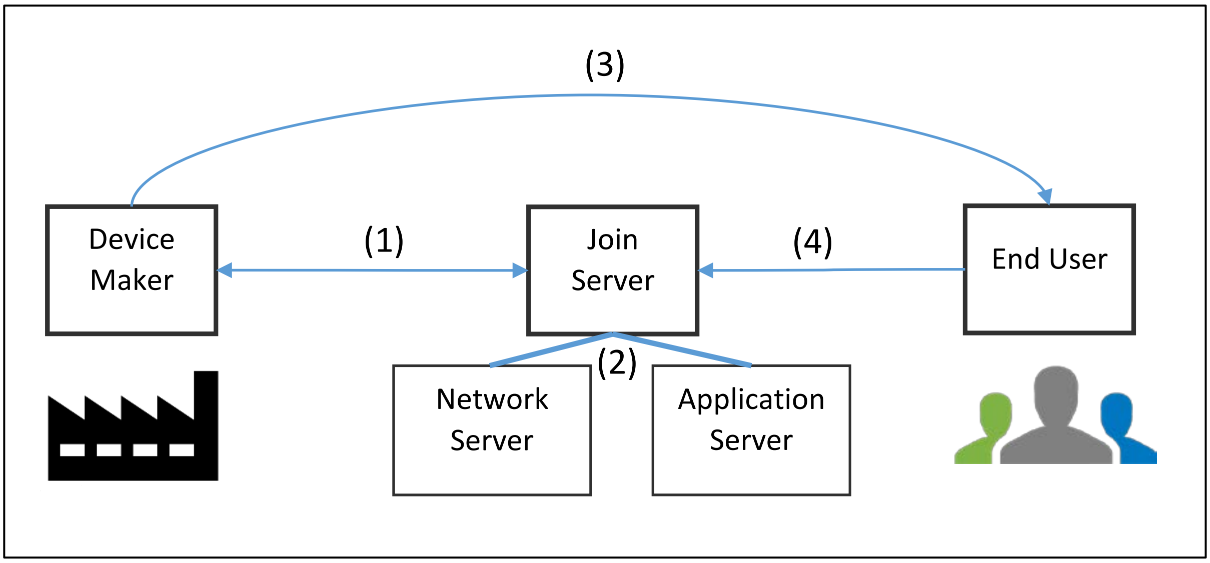


Figure 29: The end-device maker, the end-user and the Join Server

(1) **Provisioning**: During the industrialization phase, the device manufacturer will provision the end-devices with the keys (AppKey in LoRaWAN 1.0.x) as well as the JoinEUI and DevEUI. The DevEUI and AppKey must also be stored on the Join Server. When the device manufacturer and the Join Server provider share keys, they must agree on a secure way to ensure that the keys are not exposed. These keys must then be stored very securely on both entities.

(2). **Commissioning**: The NS and AS used are specified to the JS.

(3) The end-device is sold to the end-user.

(4) **On-boarding**: Finally, the end-user must prove ownership of the end-device by requesting it on the Join Server.

Activation: The JS is now ready to create session keys (NwSKey and AppSKey) whenever it receives a Join-Request.

5.4 Securing the keys

The security of the LoRaWAN protocol depends on the way the root keys are shared and stored (AppKey in LoRaWAN 1.0.x).

The challenges are as follows:

- **How do we provision the same root key (AppKey) in the end-device and in the JS without exposing them?** This operation is complicated since the LoRaWAN end-device manufacturers are not necessarily to be trusted. This means providing them the list of keys to be integrated in your components doesn’t go without precaution.
- **How to react in case of intrusion?** Physical access to the memories storing the keys must be as limited as possible. An intrusion detection must lead to the self-destruction of the keys.
- **How to prevent giving any hint to finding keys?** Calculations performed during encryption should not leave any clues to the values used, such as voltage variations or current draws.

To secure the storage of these keys and to perform all encryption operations, specific hardware modules are required on both sides (end-device and JS).

- On the end-device we use a **secure element**. These are very small components close to the microcontroller.

For example, the ATECC608B-TNGACT component has all these characteristics, and is already provisioned, i.e. it already contains root keys stored in Actility's JS.

- On the JS side we use **HSM** (**H**ardware **S**ecurity **M**odule). These have the same characteristics as the secure element but with more power and functionalities.

5.5 Device claim

When the end-device is manufactured, it is provisioned with its keys. These same keys are stored on a JS, but it is currently not authorized to respond to the end-device's Join-Request since no end user has proven its ownership through the "end-device Claiming procedure". The proof of ownership can be done through a QR Code provided by the vendor, or through the end-device information (DevEUI) associated with a code (token) found on the end-device (see Figure 30).

The process of associating an owner and verifying permission to use a network is called onboarding. These devices may be off-boarded when a user no longer desires to be responsible for the device.

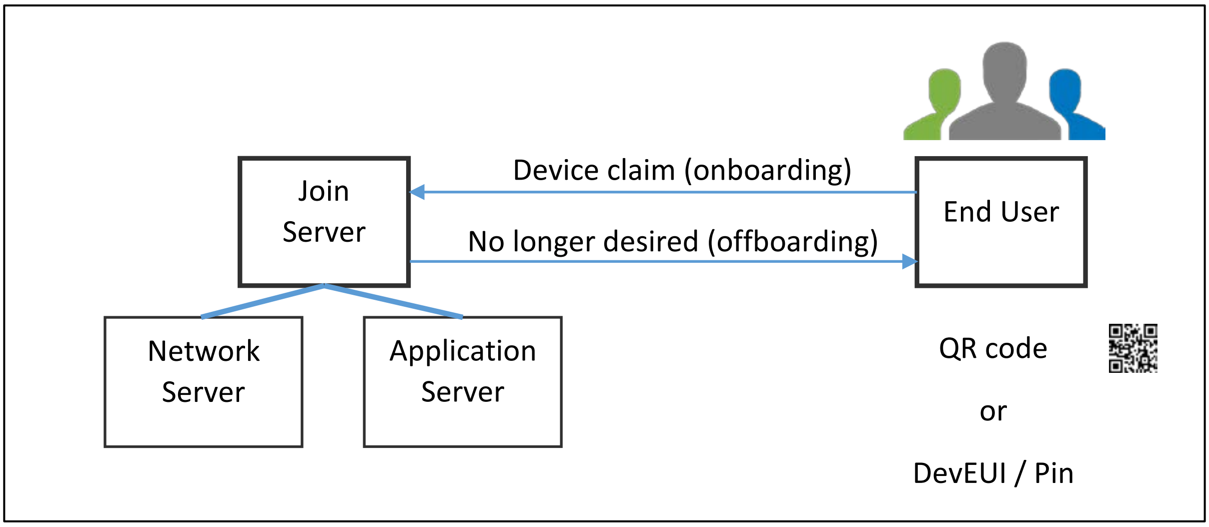


Figure 30: Device claim

If the claim is successful, the JS authorizes a Join-Request for this corresponding end-device. Session keys will then be distributed to the NS (NwkSKey) and AS (AppSKey).

If a new claim procedure is performed, for example by a new owner, then the end-device will have to perform a new Join-Request. The reason is that claiming an end-device only transfers the ownership of the end-device but does not transfer the session keys.

6 Roaming

Roaming is one of the key features of the LoRaWAN ecosystem. This chapter aims to explain why roaming is so important, and how it works.

The LoRa Alliance defines roaming specifications with many different flavors:

- Stateless passive roaming
- Stateful passive roaming
- Handover roaming

We will discuss later on the differences and purposes of these different types of roaming. In this book, however, we will only speak about **stateless passive roaming** since it is the only service available so far in the overall LoRaWAN ecosystem.

6.1 Why Roaming?

6.1.1 Definition

Roaming in telephony is formally defined as the practice of using a mobile phone on another network operator. Roaming in LoRaWAN is quite similar except that the end-device doesn't know whether it is in a roaming situation or not. That's why it is called “passive roaming”.

The LoRaWAN network where the end-device was originally registered is called the "home network". The network transferring the data to the “home network” is called the "forwarding network".

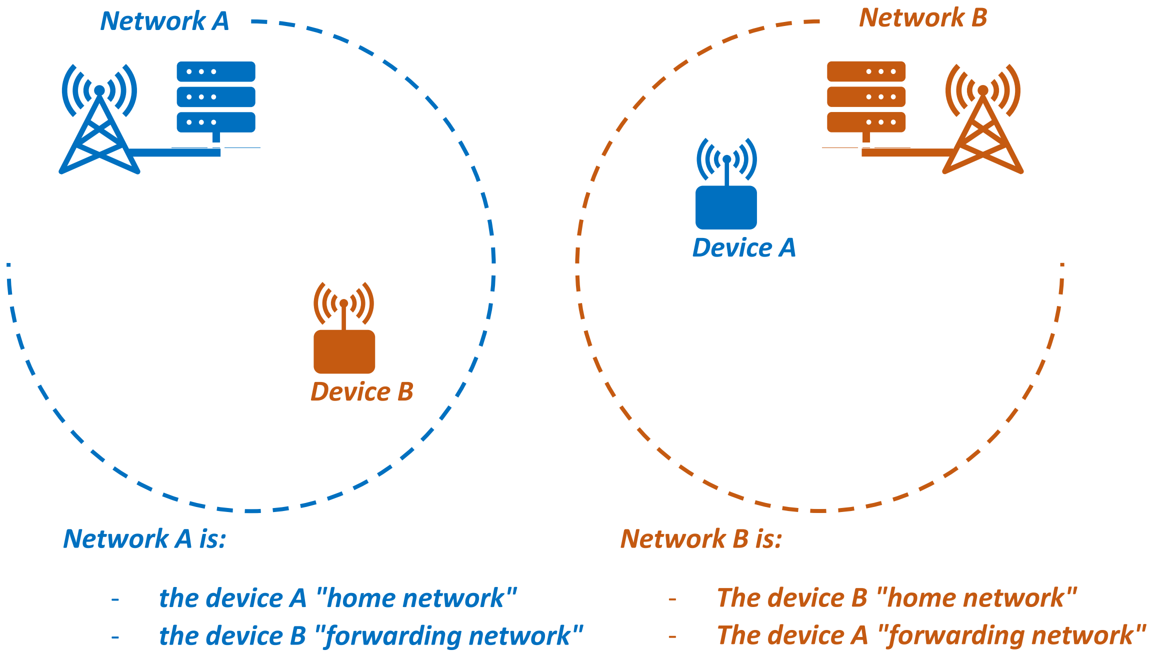

Network A
Network B
Device A
Device B
Network A is:
- the device A "home network"
- the device B "forwarding network"
Network B is:
- The device B "home network"
- The device A "forwarding network"

Figure 31: Roaming between two networks

Roaming has two main interests. The first one, which probably is the most well-known, is coverage extension. The second one, less known but just as useful, is the gateway densification.

6.1.2 Coverage extension

The idea is to extend the connectivity of an end-device by using the coverage area managed by another operator. Figure 32 shows the coverage extension of the two networks A and B when they agree to roam each other.

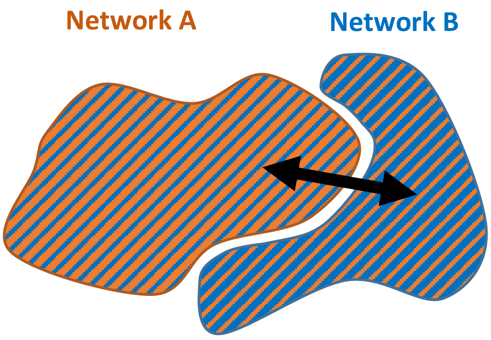


Figure 32: Coverage extension

6.1.3 Gateway densification

Even if we have the same coverage area, it’s always better to have several gateways rather than only one. Indeed, overlapping coverage has many advantages. In Figure 33, the two networks A and B have approximately the same coverage in terms of surface. However, the left one works with multiple overlapping gateways, whereas the right one works with a single gateway.

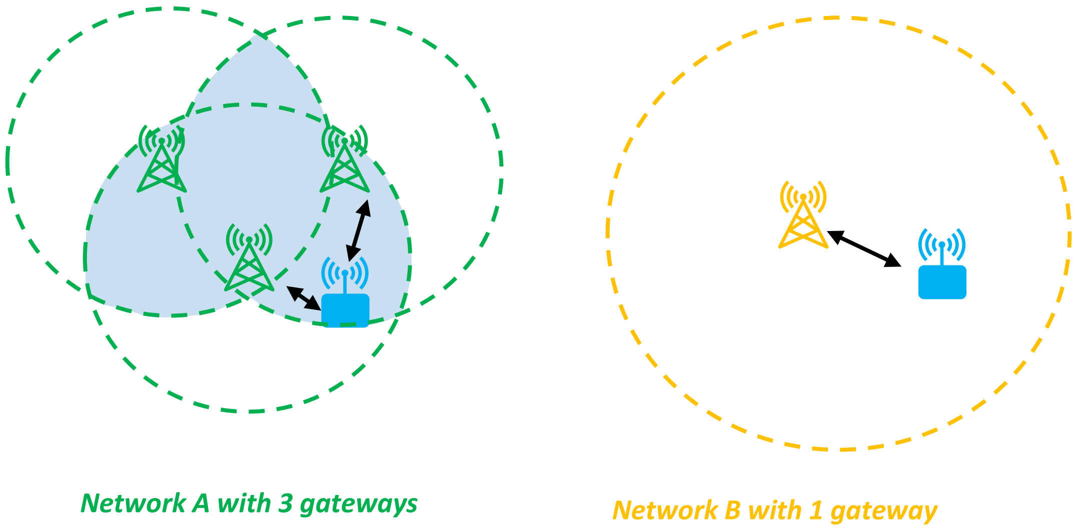


Figure 33: Gateways coverage

Connectivity loss prevention: In network A, when several gateways receive the LoRaWAN packet, they all transmit it to their Network Server. It's the Network Server’s responsibility to ensure that packet duplications are properly managed. Of course, we don't need the packet to be duplicated, but if for any reason the transmission between the end-device and one gateway fails, then:

- In the case of network A, there are still two other gateways that receive a frame.
- In the case of network B; we experience packet loss.

<u>Network capacity improvement:</u> More gateways also means increased downlink capacity for the network. Indeed, in the case of network A, there are 3 gateways able to transmit downlink frames, so the downlink capacity is multiplied by three.

<u>Low-power improvement:</u> This is probably one of the major reasons why having more gateways is important in LoRaWAN. When we deploy an end-device in the field using network A, we have more chances to have a gateway close to the end-device. The ADR algorithm will converge towards a lower power transmission and a lower Data Rate, leading to a longer battery life.

We've seen that gateway densification improves resilience, capacity, and battery life. So, how can we add gateways to our network?

1. We can buy them and set them up.
2. We can set up a roaming agreement with another network operator.

6.1.4 Coverage extension and densification

When we establish a roaming agreement with another network operator, we benefit from both coverage extension and gateway densification.

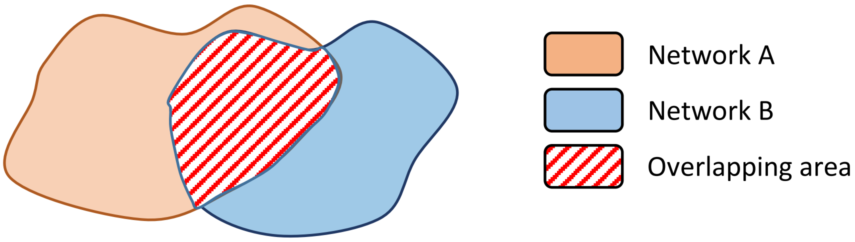


Figure 34: Two overlapping network

- 🟦 <u>Coverage extension:</u> network A extends network B’s coverage. Network B extends network A’s coverage.
- ▨ <u>Gateway densification:</u> the overlapping area is where the densification will ensure more resilience, a higher capacity, and a better low-power management of end-devices.

6.2 How does roaming work?

All the documentation on roaming’s technical definition is available in the LoRaWAN backend interfaces specification.

6.2.1 Network identification and device identification

In a LoRaWAN network, we use a 4 bytes Device Address (DevAddr) to identify end-devices. The DevAddr holds two major pieces of information:

1. The network identifier **(1)** that indicates the network it belongs to.
2. The network address **(2)** which is the address **of the end-device** within this network.

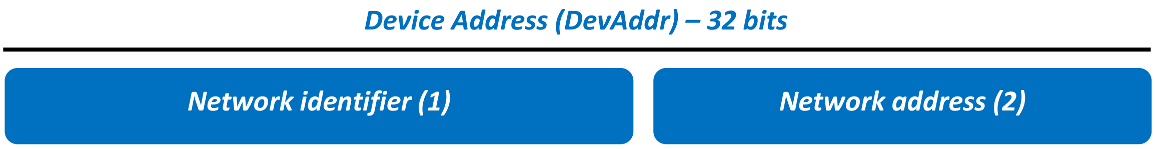


Figure 35: Device Address (DevAddr)

- The network identifier has a variable length.
- Therefore, the network address also has a variable length.

Example:

If an operator buys a 22 bits large network identifier, then there are $2^{(32-22)}$ = 1024 end-device addresses (network address) available on that network. On this network, all DevAddr will start with the same network identifier. Only the network address will change.

DevAddr are not unique. An operator can re-use them as many times it wants.

6.2.2 Roaming process

Let's explain the roaming process through the example in Figure 36. Network A and B have an active roaming agreement. In this example, device B (belonging to network B) sends an uplink frame from another network (network A).

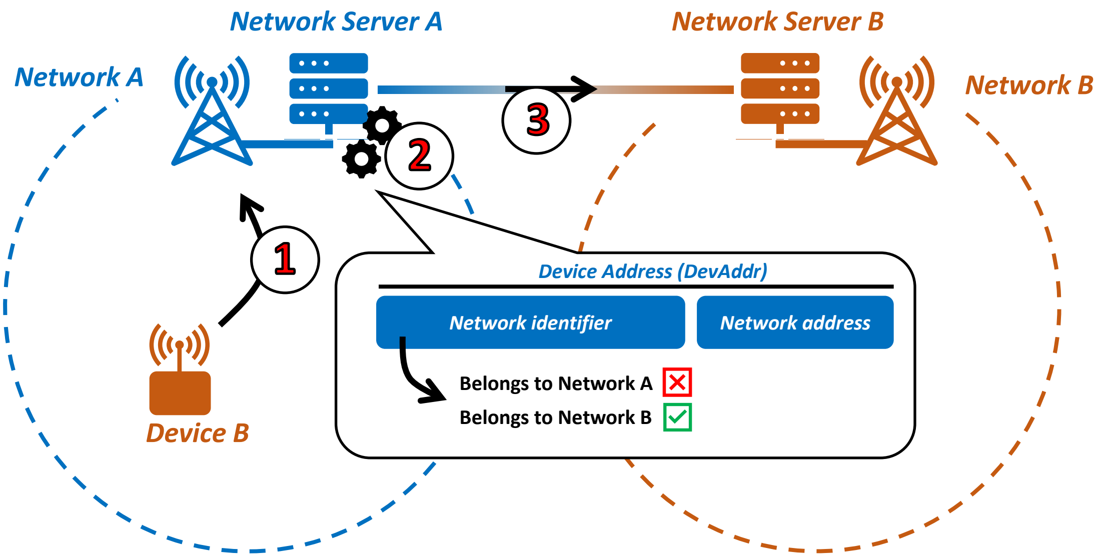


Figure 36: Roaming process

Network Server A is called the forwarding Network Server (fNS), and Network Server B is called the home Network Server (hNS).

1 **Step 1:**
Device B transmits an uplink packet from network A's coverage. Network Server A (fNS-A) receives this packet.

2 **Step 2:**
Network Server A (fNS-A) checks the DevAddr and specifically the "network identifier". From this network identifier, fNS-A finds out that the end-device comes from network B.

3 **Step 3:**
fNS-A and hNS-B exchange with each other to verify their roaming agreement, then fNS-A transmits the frame to hNS-B.

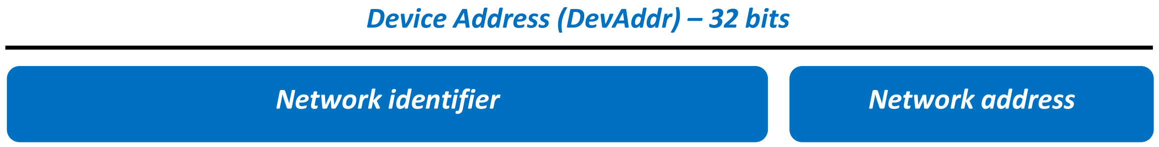
In the case of overlapping coverage areas, both fNS and hNS can receive the same frame. As usual, the duplication matter is carried out by the hNS. However, commercially speaking, when you use passive roaming, the fNS can charge you even if you already received a frame by yourself.

6.3 LoRaWAN NetID

6.3.1 DevAddr construction

In the previous chapter, we saw that the DevAddr was built in two parts:

- The network identifier
- The network address (address of the end-device in the network)



As an operator, we need to buy a network identifier so we can be sure to be the only one using this network. The network identifier allocation is managed by the LoRa Alliance. The LoRa Alliance doesn't provide the network identifier, but instead provides a value called the NetID from which we can compute the network identifier.

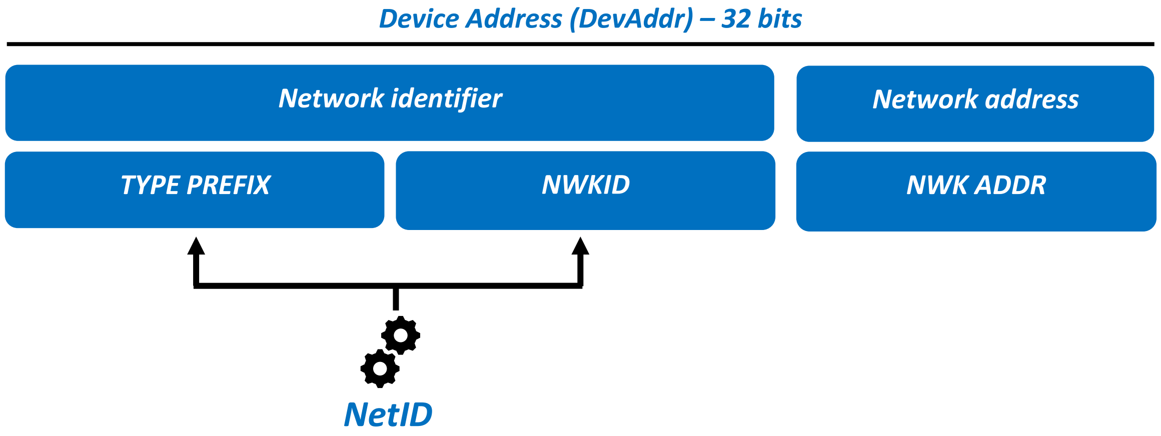


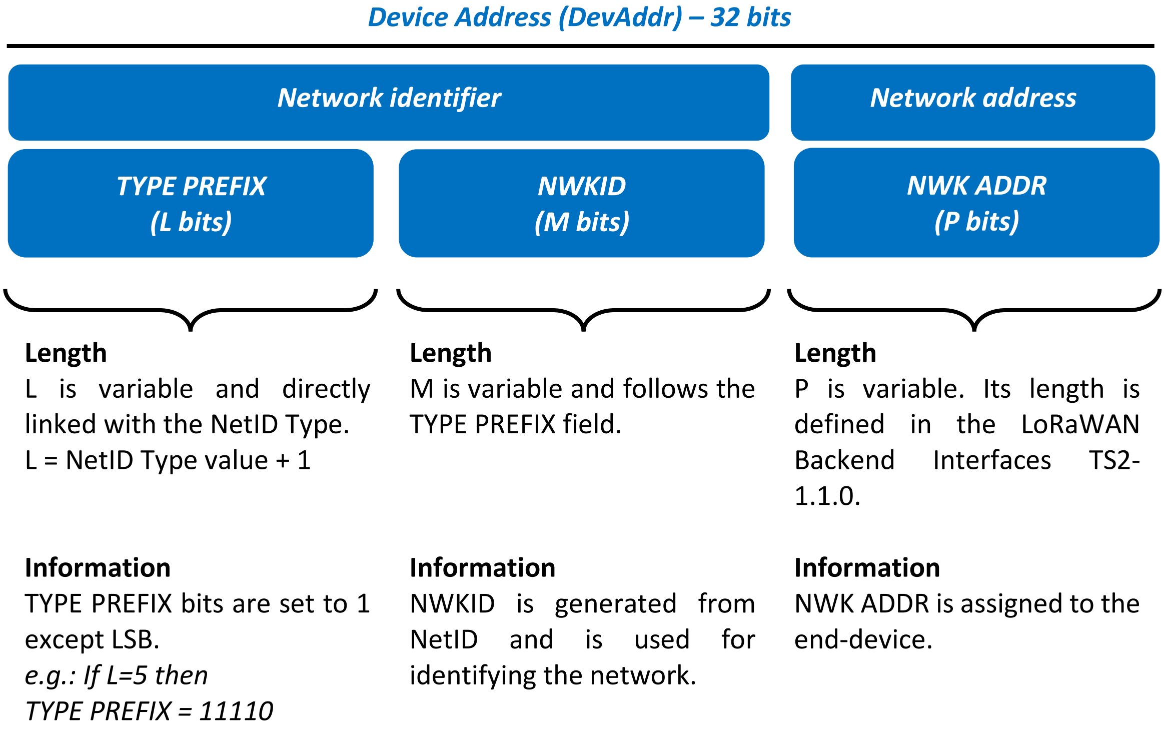
Figure 37 : Network identifier construction

6.3.2 DevAddr range

In this chapter, we are going to explain how the network identifier is built from the NetID provided by the LoRa Alliance. From the NetID, two fields are generated:

- TYPE PREFIX
- NWKID

How the TYPE PREFIX and the NWID are built, as well as their meaning, is not really an important matter. The only thing to remember is that the resulting network identifier will be a fixed value, uniquely identifying our network.

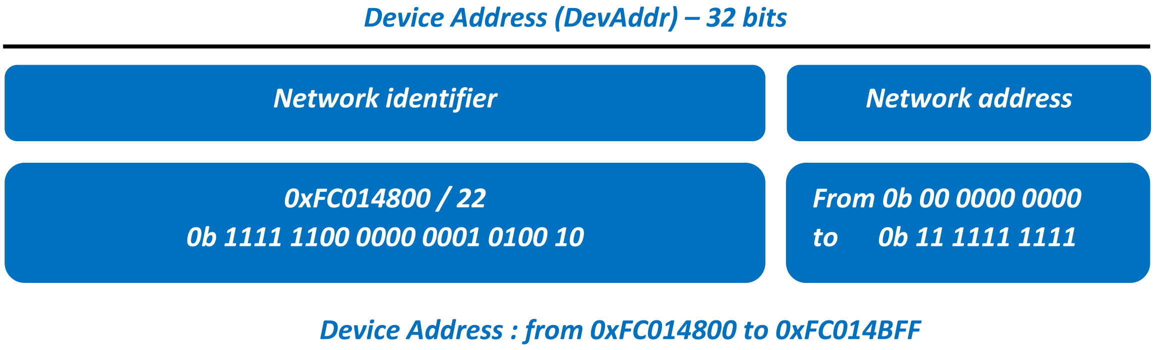


Device Address (DevAddr) – 32 bits

Network identifier		*Network address*
TYPE PREFIX *(L bits)*	*NWKID* *(M bits)*	*NWK ADDR* *(P bits)*
Length L is variable and directly linked with the NetID Type. L = NetID Type value + 1	**Length** M is variable and follows the TYPE PREFIX field.	**Length** P is variable. Its length is defined in the LoRaWAN Backend Interfaces TS2-1.1.0.
Information TYPE PREFIX bits are set to 1 except LSB. *e.g.: If L=5 then* *TYPE PREFIX = 11110*	**Information** NWKID is generated from NetID and is used for identifying the network.	**Information** NWK ADDR is assigned to the end-device.

Figure 38: DevAddr construction and field definition

At Savoie Mont Blanc university, our network identifier is 0xFC014800/22. This means that only the 22 most significant bits represent the network identifier.



Device Address (DevAddr) – 32 bits

Network identifier	*Network address*
0xFC014800 / 22 *0b 1111 1100 0000 0001 0100 10*	*From 0b 00 0000 0000* *to 0b 11 1111 1111*

Device Address : from 0xFC014800 to 0xFC014BFF

The specification contains many terms and notations. The only thing that matters is knowing the DevAddr range that belongs to a network.

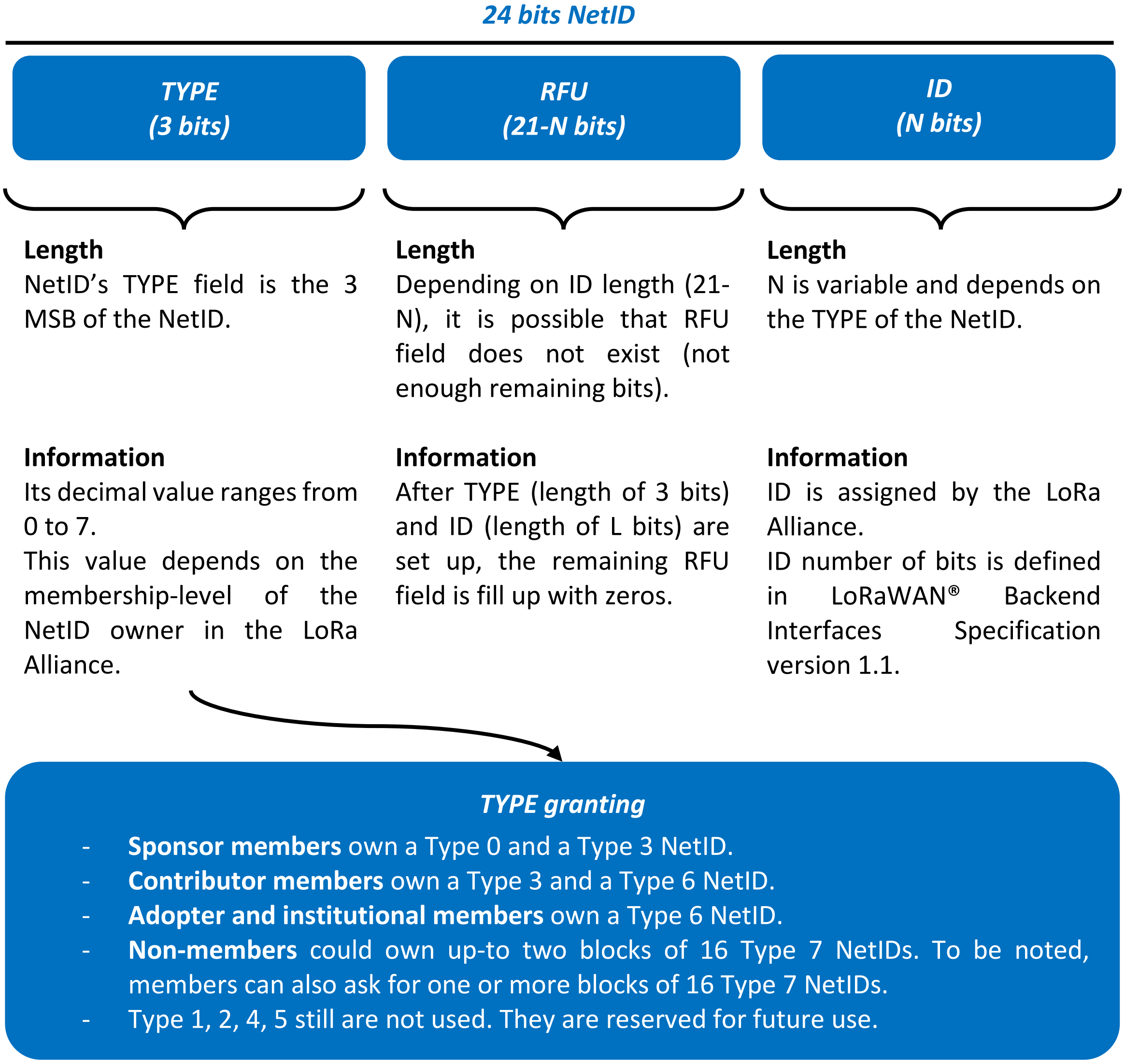
6.3.3 NetID

There are eight types of NetIDs, indicated from 0 to 7. The lower the NetID type, the bigger the network identifier size, the more addresses for our end-devices. The relation between the type and the number of addresses available is presented in Table 1:

NetID Type	*Number of end-device addresses*
0	25 bits ~ 33 millions
1	24 bits ~ 16 millions
2	20 bits ~ 1 millions
3	17 bits ~ 131 000
4	15 bits ~ 32 000
5	13 bits ~ 8 000
6	10 bits ~ 1000
7	7 bits ~ 128

Table 1: Number of end-device addresses available in a network.

The NetID is represented by the 24 bits that we receive from the LoRa Alliance when we want to buy a network identifier. Through this NetID, we can find useful values to finally determine our network identifier.



24 bits NetID

TYPE (3 bits)

RFU (21-N bits)

ID (N bits)

Length
NetID's TYPE field is the 3 MSB of the NetID.

Information
Its decimal value ranges from 0 to 7.
This value depends on the membership-level of the NetID owner in the LoRa Alliance.

Length
Depending on ID length (21-N), it is possible that RFU field does not exist (not enough remaining bits).

Information
After TYPE (length of 3 bits) and ID (length of L bits) are set up, the remaining RFU field is fill up with zeros.

Length
N is variable and depends on the TYPE of the NetID.

Information
ID is assigned by the LoRa Alliance.
ID number of bits is defined in LoRaWAN® Backend Interfaces Specification version 1.1.

TYPE granting

- **Sponsor members** own a Type 0 and a Type 3 NetID.
- **Contributor members** own a Type 3 and a Type 6 NetID.
- **Adopter and institutional members** own a Type 6 NetID.
- **Non-members** could own up-to two blocks of 16 Type 7 NetIDs. To be noted, members can also ask for one or more blocks of 16 Type 7 NetIDs.
- Type 1, 2, 4, 5 still are not used. They are reserved for future use.

Figure 39: NetID construction and filed definition

24bit NetID

Type	TYPE (3 bits)	RFU	ID
0	000	15 bits	6 bits
1	001	15 bits	6 bits
2	010	12 bits	9 bits
3	011	none	21 bits
4	100	none	21 bits
5	101	none	21 bits
6	110	none	21 bits
7	111	none	21 bits

Table 2: ID field length

6.3.4 NetID and DevAddr relation: How does it work?

According to the LoRaWAN Backend Interfaces Specification version 1.1, we have extracted Table 2 and Table 3 from which we can build the DevAddr.

32bit DevAddr

Type prefix decimal value	TYPE PREFIX binary value	Number of type prefix bits	Number of NWKID bits	Number of NWK ADDR bits
0	0	1	6	25
2	10	2	6	24
6	110	3	9	20
14	1110	4	11	17
30	11110	5	12	15
62	111110	6	13	13
126	1111110	7	15	10
254	11111110	8	17	7

Table 3 : DevAddr construction

Let’s try an example with the NetID our university received from the LoRa Alliance: **0xC00052**.

First, the 3 MSB reflect the membership-level. And, according to Table 2, the ID could have a 6-, 9- or 21-bits length.

USMB example

The NetID is 0xC00052, so:

0xC00052 = **0b1100.0000.0000.0000.0101.0010**
- **TYPE:** 3 MSB of the NetID are 0b**110**, i.e., 6 in decimal. So, it is a Membership-level **type 6**
- **ID:** Knowing type is type 6, ID length is **21 bits** (see Table 2).
So our ID is 0b0.0000.0000.0000.0101.0010
- **RFU:** Knowing ID length is 21 bits, RFU length is null in our case

Then, the NWKID can be deduced from the NetID. Its length is specified in Table 3.

USMB example

With the USMB NetID, the NWKID is:

- 15 bits length (according to Table 3)
- So, NWKID is 0b000.0000.0101.0010

Finally, we can find the other fields of the DevAddr

USMB example

With the USMB NetID, DevAddr fields are:

- TYPE PREFIX is 0b111.1110 (according to DevAddr array)
- NWK ADDR length is 10 bits (according to DevAddr array)
- By inference, DevAddr shape is:

0b1111.1100.0000.0001.0100.10xx.xxxx.xxxx

(With TYPE PREFIX in green, NWKID in orange, NKW ADDR in red)

It can also be noted as a network identifier: FC014800/22 also called DevAddr subnet.

DevAddr range goes from 0xFC014800 to 0xFC014BFF.

6.3.5 Private and experimental NetID

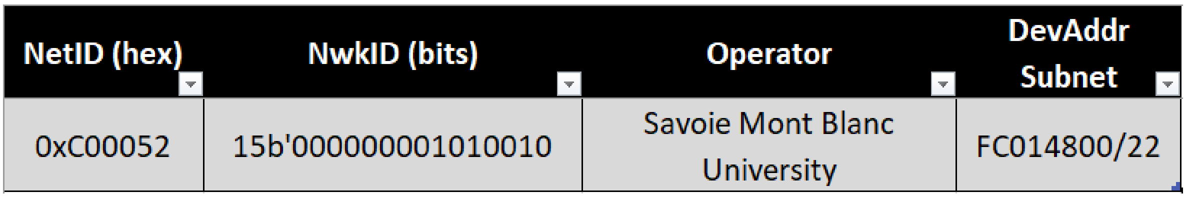
When using the LoRaWAN protocol, we may not be interested in using the roaming capability. This is the case for many private infrastructures that do not own a specific NetID. They can use 2 NetIDs available for private and experimental purposes. These NetIDs are:

- 0x000000 (offering a DevAddr range from 0x00000000 to 0x01FFFFFF)
- 0x000001 (offering a DevAddr range from 0x02000000 to 0x03FFFFFF)

6.3.6 Receiving a NetID from the LoRa Alliance

Anyone can buy a NetID from the LoRa Alliance as long as they fulfill the conditions reported in the document called *NetID Policy and Terms*.

As an example, here is what Savoie Mont Blanc university has received after our NetID request. We obviously found the NetID, from which the NwkID and our network identifier (0xFC014800/22) were computed. In this document, the network identifier is called the DevAddr Subnet.



NetID (hex)	NwkID (bits)	Operator	DevAddr Subnet
0xC00052	15b'000000001010010	Savoie Mont Blanc University	FC014800/22

Figure 40: Information provided by the LoRa Alliance.

6.4 Stateless passive roaming between two private networks

This part aims to present a PoC of a stateless passive roaming between two private LoRaWAN networks.

6.4.1 Equipment

The objective of this PoC is to roam frames emitted from a fNS to a hNS.

To do so, there are:

- An ABP end-device that is activated and registered into its hNS: The end-device is a NUCLEO-L073RZ working with a I-NUCLE-LRWAN1 shield. It emits a frame when its blue button is pushed.
- A home Network Server (hNS): The hNS is a Chirpstack NS. Let's call it CS1 (Chirpstack n°1). It is installed into a Docker Container.
- A forward Network Server (fNS) and its gateway: The forward NS is Chirpstack NS. Let's call it CS2 (Chirpstack n°2). It is installed into a Docker Container. A gateway forwards LoRaWAN frames to CS2. This Gateway is a NUCLEO-F746ZG.

The schematic below shows the infrastructure.

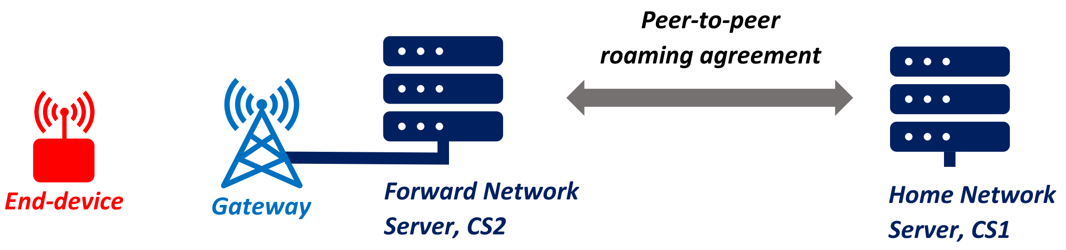


Figure 41: Equipement for Passive Roaming PoC

6.4.2 Configuration

To enable the roaming between our two NS, they must have a peer-to-peer roaming agreement. That means:

- CS2 has a roaming configuration setup with CS1 IDs (NetID and server address).
- CS1 has a roaming configuration setup with CS2 IDs (NetID and server address).

- **Our roaming configuration does not use the official DNS resolving. It simply looks up to the IP address of the matching NetID.**
- **This PoC does not use TLS services.**

To configure a roaming agreement with a CS Network Server, there is a dedicated section to add into the *Chirpstack-network-server.toml* file.

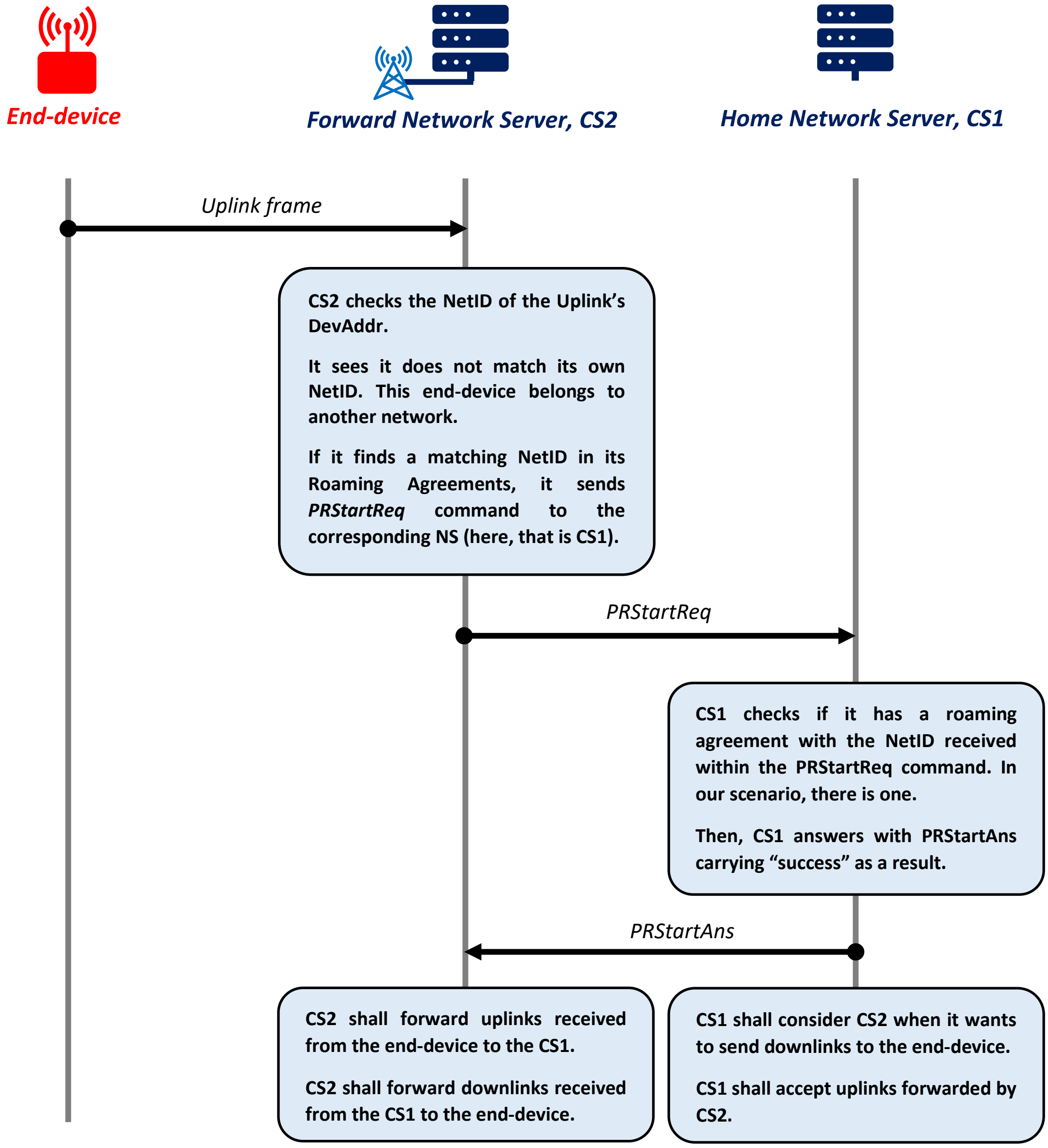
For the CS1, the section is:

```
[roaming]

  # Resolve NetID domain suffix.
  resolve_netid_domain_suffix=".netids.lora-alliance.org"

  [roaming.api]
  # Interface to bind the API to (ip:port).
  bind="0.0.0.0:9000"
  ca_cert=""
  tls_cert=""
  tls_key=""


  # Per roaming-agreement server configuration.
  [[roaming.servers]]
    # NetID of the roaming server.
    net_id="NETID of the CS2"

    # Use the async API scheme.
    async=true

    # Async request timeout.
    async_timeout="1s"

    # Allow passive-roaming.
    passive_roaming=true

    # Passive-roaming session lifetime.

    # When set to 0s, the passive-roaming will be stateless.
    passive_roaming_lifetime="0s"

    # Passive-roaming KEK label (optional).
    # When set, the session-keys will be encrypted using the given KEK when these
    # are exchanged.
    # passive_roaming_kek_label=""

    # Server (optional).
    # When set, this will bypass the DNS resolving of the Network Server.
    server="http://ADDRESS-OF-THE-CS2:9000"

    # CA certificate (optional).
    # When configured, this is used to validate the server certificate.
    # ca_cert=""

    # TLS client certificate (optional).
    # When configured, this will be used to authenticate the client.
    # This mist be configured together with the tls_key.
    # tls_cert=""

    # TLS key for client certificate (optional).
    # tls_key=""
```

For the CS2, the section is the same as the previous one (with CS1 NetID and server address).

6.4.3 Scenario

The end-device will emit a frame into the CS2 coverage. The CS2 will roam the frame to the CS1. Here are the steps.



CS2 checks the NetID of the Uplink's DevAddr.

It sees it does not match its own NetID. This end-device belongs to another network.

If it finds a matching NetID in its Roaming Agreements, it sends *PRStartReq* command to the corresponding NS (here, that is CS1).

CS1 checks if it has a roaming agreement with the NetID received within the PRStartReq command. In our scenario, there is one.

Then, CS1 answers with PRStartAns carrying "success" as a result.

CS2 shall forward uplinks received from the end-device to the CS1.

CS2 shall forward downlinks received from the CS1 to the end-device.

CS1 shall consider CS2 when it wants to send downlinks to the end-device.

CS1 shall accept uplinks forwarded by CS2.

Figure 42: Stateless passive roaming between two private LoRaWAN Networks

In this scenario, CS1 is hNS. When a roaming session is launched, the Backend Interface could also call CS1 as sNS (in addition to be hNS). A NS called sNS (Serving Network Server) handles the MAC layer.

The get more information about passive roaming commands, see the LoRaWAN backend interfaces specification.