Mid-Term Report
Core Architecture Development V1

Matthias Benkort - Software Engineer
The Smiths | The Things Network

01 february 2016

Contents

Forewords

The following report stands for a general feedback on The Things Network v1 project. As I am writting those words, the project is still ongoing, hopefully a bit more than half completed. Another report will follow this one in two months to chart the evolution and the global progress of the project.

1 Recap

As a reminder, we first divided the project in six unbalanced milestones:

- #1: Create a gateway + nodes simulator for testing purpose.
- #2: Support for uplink message
- #3: Support for registration by personnalization
- #4: Support for downlink message
- #5: Support for Over-The-Air Activation (OTAA)
- #6: Support for LoRaWAN MAC commands

So far, the first three milestones are done yet partially integrated. The fourth milestone should be relatively straightforward regarding to the way the uplink has been handled (more details on the third section).

1.1 Milestone #1

The first milestone is somehow independant from the five others. That part aims at creating a software able to simulate a traffic coming from nodes. The traffic could either be randomly generated or be dicted by a specific schedule (only the scheduled mode is available at the moment, even if everything needed to generate random nodes exists).

Because it was needed to build a packet forwarder, we should also keep in mind that it might be a good starting point if we need to replace the current forwarder implementation of semtech that is being used by loads of (all?) gateways. That implementation never intended to be put in production devices. `Go` makes shipping and building fairly easy, let's take advantage of this in a near future.

1.2 Milestone #2

Ideally, we would replace the current prototype implementation of the network by a partial realisation of this first version – let's call it `0.5.x`. We have set up all milestones the following way in order to perform an incremental work that would firstly replace the existing architecture and from then on, bring additionnal features. Naturally, the real first milestone related to the architecture is about sending an uplink packet from a node to an associated application.

Regarding to what have been discussed in the different meetings about the Architecture, the logic has been divided in three components: router, broker and handler. Incidentally, applications are not considered to be part of the network, and network controllers are not yet implemented. The uplink process that has been implemented is described below:

1.2.1 Router

Router components listen to a given `udp` port for incoming requests. It expects an exchange protocol described in the following semtech document. Thus, only valid `rxpk` packets reach the router core logic (other are either acknowledged immediately, `pull_data` for instance, or simply ignored, so are `push_data` with `txpk` or `stats`).

Because a single packet may hold several messages from possibly different nodes, the router interprets each message independently, one after the other. For each message, it determines the associated device addresses and lookup in its internal storage for known brokers.

Therefore, the router maintains an in-memory mapping of end-devices and brokers. The mapping is a one-to-one relationship, meaning that a single node might be associated to only one broker. Each entry automatically expires after a given delay (8 hours for the moment) to prevent malicious brokers from registering themselves once and for all.

The storage is populated on the fly. For each unknown node (no valid entry associated to the node address), the router will broadcast the request to a list of pre-configured brokers. A maximum of one broker is expected to give a positive answer which indicates its responsibility towards the node. Multiple positive answers are considered as an error (indicates the presence of either an errored broker or a malicious one). For each positive reply, a new entry is created in the internal storage.

The current system has *strong* limitations. Firstly, the broadcasting and sending are done over `http`. Secondly, Because of the address range constraints forced by LoRaWAN, device addresses may collide (and will) such that multiple nodes could send packets with a same end-device address. An address is registered for a given delay which implies that a router could possibly forward a packet to a wrong broker. In such a case, the action will invalidate the registry entry and causes the packet to be broadcast again to determine the right broker. This isn't really efficient though.

The process is likely to stay like this until a proper solution is found. The issue is well-known and its seemingly a hot-topic in the backend team. The idea isn't about wasting time on that point because it's still in discussion; we are focusing on the other parts in the meantime.

1.2.2 Broker

In the current stage of the project, brokers are merely `http` balancers. They listen on `http` requests from routers, receive packets, compute MIC, possibly forward to handlers and reply back to routers. They also offer a simple API to allow handler registration given an `http` address, a node address and a network session key.

Like routers, they hold an internal storage of every association node `<->` handler. Thus, similar issues to what we face between router and broker arise. Currently, there's no way to ensure that a handler which registered itself isn't a malicious one. Meaning that the storage could be easily polluted with crappy entries. The broker is nevertheless able to compute a MIC check which should prevent packets from being wrongly forwarded.

1.2.3 Handler

Handler are not completely done yet. The very last communication part between the handler and the application needs to be setup (we're still exploring some tools in order to establish application authentication on the one hand, and packet publication on the other hand). Beside, Handlers are currently able to receive packets, through `http` as well, and "bufferize" the request in order to possibly get more than one packet. Every gateway receiving a packet would transmit the given packet such that, all packets are supposed to end up in the same handler in a quite short time frame.

Currently, a handler will wait for a configured delay (300ms at the moment) and gather every packet coming from the same node as long as they refer to a same frame counter. After a delay, all common packets are deduplicated, the payload is decrypted, and everything should be safely published to the related application (we're going to use an MQTT broker with topics under authentication). The handler keeps track of the last frame counter received for each node and will refuse any packet with an identical counter (a late packet). This works under the assumption that a node won't send more than one message at a time. If we consider such a case, then the waiting mechanism would have to evolve.

1.3 Milestone #3

Milestones 2 and 3 were tighly coupled. Should we consider an uplink transmission of a packet, we need a recipient to which carry the given packet. The "easiest" way to define such recipients for the moment is by the use of personnalization because it does not require any additionnal internal command or management. In the current stage, the registration process should be started by the handler. That precise part is not implemented yet (it echoes to the above section about the handler). Basically, we'll consider an MQTT broker to which application can register. The handler is thus intended to have root accesses on the broker in order to query and retrieve the list of registered application. Then, each handler register each device of the application to a broker using an `http` request. This means that for each device, one request is done, and there is currently no way to cancel or edit a registration.

1.4 Milestones #4, #5 and #6

The last 3 milestones are currently pending. The downlink transmission is already partially implemented (handlers can reply to brokers which can reply to routers; responses are nevertheless ignored by routers).

The OTAA won't require a huge amount of work either. However, before dealing with those new milestones, we've to get rid of several issues and discussions on the first three. Working on milestone 4,5 and 6 isn't planned before March.

2 What's Next

This section will cover topics that could be seen as current technical debts. These issues have to be addressed before moving to the next milestones in order to keep building on something reliable and robust.

2.1 Error Handling

2.1.1 Current state

First of all, the error handling is so far relatively chaotic. After having digged into *Docker*, *Github* and the *Golang core libraries* we are now full of insights to deal with those issues in a clean and consitents manner. We mainly face three types of errors in the architecture:

- Recoverable: Those which could be expected
- Failure: Those which are unrecoverable failures
- Fatal: Those which lead to an interruption of the program

In the first case are placed every error we create on purpose, in order to communicate a wrong operation which could nevertheless be recovered. For instance, a router which looks up for a device may encounter an error when no entry exists for the related device. In such a case, the router can just broadcast a message to figure out the recipient and store it for a future message. The error is handled internally.

The second type of errors are unexpected errors for which there is no computable solution or possible recovery. For instance, unmarshalling a datagram from bytes to a `Go` structure might fail because of unknown reasons. Because the unmarshalling is completely deterministic, there is more or less nothing we could attempt to recover from such an error. However, it does not prevent the program from continuing its main process. The packet is merely ignored (nonetheless after having triggered an appropriate log entry). In some other cases, when the computation isn't totally deterministic, when we deal with side effects such as sending over a connection for example, our best chance is to give another try, fingers crossed. After a given amount of trials (let's say 3), the process should just be aborted and ignored.

Finally, the last type of errors concerns every failure that is a blocking for the program. For instance, to fail to create and access a component internal storage is one of those cases. Most of the time, we can handle those cases by properly interrupting the program and wait for a manual intervention (or at least, an external one. A container in failure can be automatically restarted by a monitoring system). We shouldn't technically have any crash or unhandled wild error popping out from nowhere. Should this happens however, we might be able to detect the failure and restore correctly the routine.

2.1.2 Foreseen solutions

We'll define three types of errors to handle the three cases listed above. In the second and third case, we'll merely encapsulate the given error in either a `unexpected failure` or a `fatal failure`. This will allow us to treat errors consistently depending of their nature. In fact, in both case, we'll also define an amount of expected trials such that, the caller could know whether the issue might be solved by retrying. In any case, we have to provide a meaningful log entry to help to quickly identify unexpected behavior.

2.2 Definition of a Packet

We call packet the underlying data structure transferred between component. Currently, a packet is composed of a LoRaWAN physical payload and a bunch of Metadata which could be seen as a dictionnary of values associated to a specific key (usually represented as a JSON object). This is slightly different, though inspired, from the definition of a packet from the semtech udp protocol (`rxpk` and `txpk`). Therefore, instead of carrying out a packet with a base64 encoded version of the payload, we simply directly transfer that payload with the associated Metadata.

However, that representation of a packet is stricly identical for the router, the broker and the handler which might be an issue. This feeling has been reinforced during past meetings where the idea of a partial transfer between the router and the broker was pronounced. The representation of a packet should probably evolve over transfer, getting more and more precise and well-defined.

As an illustration, if we consider a same definition of what is a packet from brokers and handlers point of view. Then, when a handler receive a packet (which is merely a physical payload), the only identifier available is the device address. However, the handler thereby faces the exact same issue as the broker when several devices share the same address and thus, has to perform a MIC check to determine the right packet. That action was already performed by a previous broker which means that at some point, the packet could be identified by a device address and a network session key (and we might consider the union of those informations as unique, might we?)

That's why I do believe the current definition of a packet isn't flexible enough to perform well in the network. This question is still ongoing and will be resolved as soon as possible.

2.3 API & internal communication

2.3.1 Security issues

One of the biggest challenge with The Things Network is to keep it open. One says open where it actually means that you cannot trust neither an emitter of a request, nor a recipient with which you are communicating. This implies to set up some security processes to ensure one's talking to the one it is expecting to talk. We face the issue at almost every level of the network. When a router sends a message to a broker, and symmetrically, when a broker receives a transmission from a router. Also, brokers and handlers offer API to register end-devices to a given address. How do we ensure that a registration is legit? So far we don't. This should nonetheless be addressed quickly.

In a near future, we'll review the communication protocols between each party of the network to come up with secure ways of transmitting information while ensuring the trustyness of the communication. This will probably cause a lot of overhead but we have no choice in an open and non-federate network.

2.3.2 Registration management

We primarly focused on the uplink transmission of datagram. For those reasons, the API defined to register devices are extremely simple. As mentionned above, there's no way to cancel or edit a registration. This is not convenient at all. However, because protocols are likely to change (should we get rid of HTTP?), defining such API is merely a waste of ressources for the moment. The above parts should be addressed in priority.