Installation and monitoring procedures

Tanguy Kerdoncuff, Fabrice Guillemin

▶ **To cite this version:**

Tanguy Kerdoncuff, Fabrice Guillemin. Installation and monitoring procedures. [Technical Report] Acklio. 2022. hal-03826254

HAL Id: hal-03826254
https://hal.science/hal-03826254

Submitted on 24 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Installation and monitoring procedures

Project Deliverable D.4.1.1

INTELLIGENTSIA project

Executive summary

This document describes the practical setup and external usage of the IoT platform based on Long Range Wide Area Network (LoRaWAN) and deployed for the experiments of the Intelligentsia project.

Authors: Tanguy Kerdoncuff

Revised by: Fabrice Guillemin

Approved by: Fabrice Guillemin

Date: October 24, 2022

Contents

List of Figures

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

IoT Internet of Things.
IP Internet Protocol.

LNS LoRaWaN Network Server.
LoRaWAN Long Range Wide Area Network.

RAM Random Access Memory.
REST Representational state transfer.

SCHC Static Context Header Compression.

VPN Virtual Private Network.

Chapter 1

Introduction

This document is a deliverable of the Intelligentsia project and is a contribution to workpackage 4 *Proof of Concept and Experimentation* by presenting the details of the experimental platform where the findings of the projects will be demonstrated and evaluated. It consists of the items of an Internet of Things (IoT) network deployed *in situ* (namely, the gateways and devices) and the cloud elements for the core network and the final IoT data exploitation servers. Its general architecture is presented in Section 3. All the elements considered are implemented by using state of the art industrial products with which members of the consortium already have experience, and are able to adapt to the need of the project.

The cloud elements themselves are hosted on a kubernetes cluster dedicated to the project, which will not only host the containers implementing the core networks, but also metrics and log gathering tools used to feed the decision engine. These tools, as well as the entry points that the decision engine will use to control the platform are described in Sections 5 and 6, respectively.

Chapter 2

Delivery and Integration planning

Table 2.1 presents the different tasks planned for the delivery of the platform. The first part relates to the setup of the IoT communication chain, with most of the interaction happening between Acklio and Aguila, in order to configure the devices, gateways, LoRaWaN Network Servers (LNSs), IPCore and application server that will implement the use cases considered by the project.

The second part contains the tasks responsible for interfacing the platform to the project's controller. One the one hand, the logs and metrics gathered in the cluster need to be forwarded to this controller; these aspects are described in Section 5. Then, the controller will need entry points to the platform in order to perform the actions determined by the decision engine ; these aspects are addressed in Section 6.

Task	Partners
IoT Communication chain setup (part 1)	
Lorawan gateway purchase and agent configuration	Acklio
Additional gateway logging and metric tools	Acklio
Gateway shipping to deployment site	Acklio
Gateway deployment and connectivity validation	Acklio,Aguila
LNS Access opened	Acklio
Simple Lorawan connectivity test	Acklio,Aguila
Acklio SDK Integration	Acklio,Aguila
Platform integration to controller (part 2)	
Kubernetes access opened	Acklio
Log export setup	Acklio,?
Prometheus Mirroring	Acklio,?

Table 2.1: Platform delivery tasks

Chapter 3

Platform Architecture

Figure 3.1 presents the general architecture of the IoT testbed, in terms of communication chain. It is divided into three blocks, which are, from the left to the right

- The Field elements, regrouping the IoT devices as well as the gateways. End devices and gateways will be deployed at the same location in order to ensure proper radio coverage. A first device deployment site has been identified at Aguila premises, but other may be decided later in the project.
  - The IoT devices are manufactured by Aguila and respond to the use cases addressed in this work. An integration work between Acklio and Aguila will also be performed for these devices in order to experiment with the Static Context Header Compression (SCHC) standard.
  - The Gateways will be packaged by Acklio, with pre-configured gateway agent letting them connect securely to the cloud elements of the platform. Additional logging and monitoring tools will be added to the gateways before dispatch (see Section 5).
- The Edge cloud elements represent the network core interfacing the (potentially non Internet Protocol (IP)) IoT world with the classical Internet. These elements are hosted on a kubernetes cluster in the Acklio cloud.
  - The platform contains LNS instances (Acklio or Chirpstack), each consisting of a series of containers. Their configuration can be tweaked, and the number of different instances can be adapted in order to experiment with the different slicing strategies identified in the project.
  - The platform also feature IPCore instance(s) that interface upstream of the LNSs. They reception the Lorawan payload and rebuild the original IP packets they contain, before forwarding the device's traffic over the internet, to its final destination.
- Finally, the central cloud represents the place where packets are actually exploited in the respective use cases. Depending on the experiment, classic LoRaWAN application server, receiving payloads over HTTP, Websocket or other means, or IP Application server receiving the device traffic directly through a Virtual Private Network (VPN). Deployment of these servers can be done in the Acklio cloud, although some experiment may be ran more effectively if whichever partner piloting a given experiment has direct access to both ends of the communication chain.

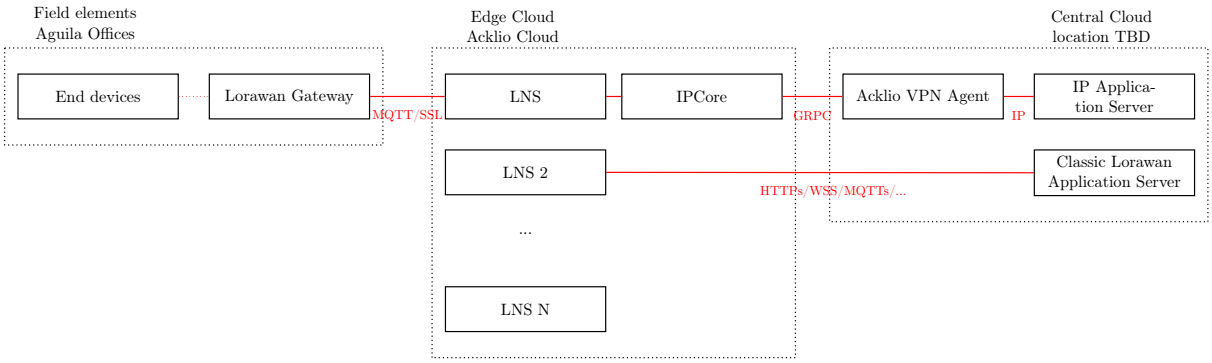

Figure 3.1: General Platform Architecture

Chapter 4

Onboarding

4.1 Device onboarding

Devices will need register to one of the vLNS available on the cluster. This should, at least in the first phases, be done manually through the chirpstack web interface, refer to their online documentation [1] for details.

4.2 gateway onboarding

Gateways should have already been configured by Acklio for backbone connectivity to the platform's cloud elements. Depending on the experiment, further configuration may be required, in particular for band selection and channel allocation. Again, refer to the chirpstack documentation for more details [1].

4.3 IP Application onboarding

By using the Acklio IPCore, one can receive IP traffic from IoT devices. In order to enable connectivity of a given IP prefix between the cloud elements of the platform, and the final destination of these IP packets, we rely on a VPN client deployed in the central cloud. Final customers will therefore be delivered an archive containing the client binary, access certificates and a readme. More details on this connection can be found in Section 3

Chapter 5

Logs and metrics

5.1 Metrics

Figure 5.1 presents the platform's logging and monitoring architecture. This platform's monitoring relies on prometheus. This open source software has been gaining popularity in the devops community since its original creation at SoundCloud. Prometheus relies on the scraping of HTTP endpoint that expose metrics in order to populate its time series database. With its current level of adoption, many other applications have made the choice of exposing prometheus metrics, it is in particular the case of chirpstack.

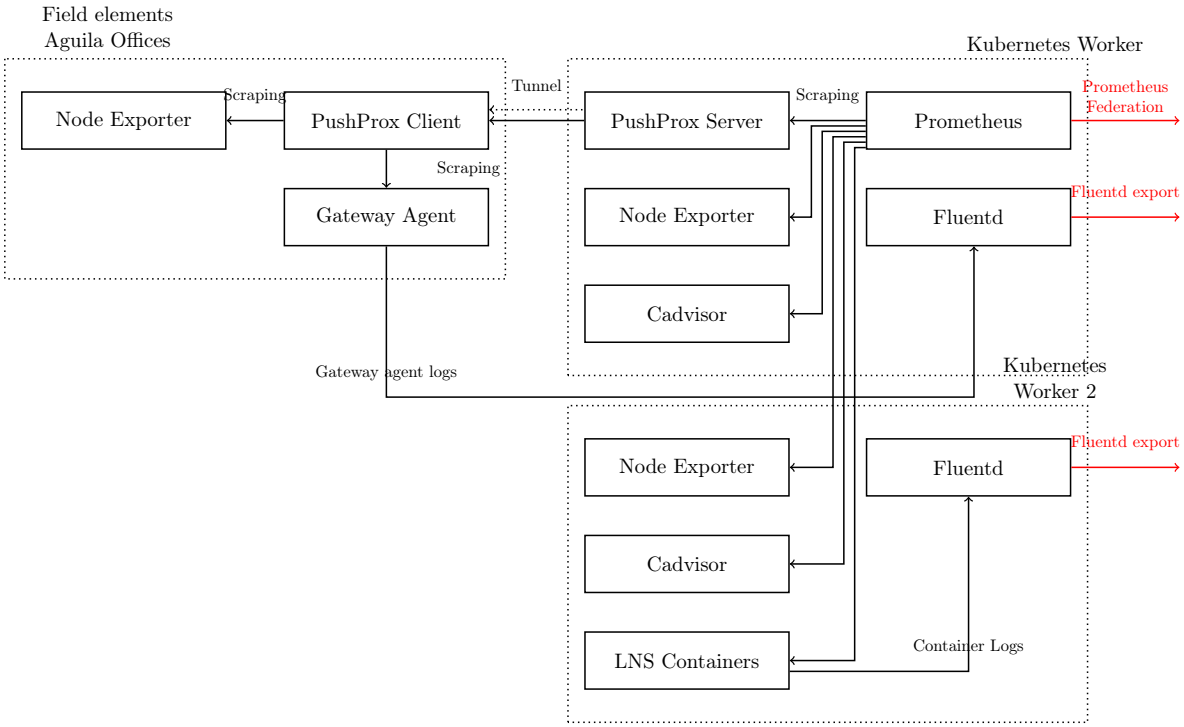

Figure 5.1: Logs and metrics export elements

In this platform, the target for prometheus scraping will be the following

- The containers of the Chirpstack LNSs providing LoRaWAN related time series.
- The nodeexporter service, that publishes metrics relating to a linux machine's state (Central Processing Unit (CPU) usage, Random Access Memory (RAM), Traffic, IOps, etc). The use of a kubernetes daemonset ensures that exactly one copy of this software runs on each worker node of the cluster. It is also able to interface with prometheus in order to

discover scrapeable endpoint, meaning that time series are automatically built as soon as a new machine joins the cluster.

- The cadvisor service, that publishes information on docker containers that run on a docker host (CPU usage, RAM, Traffic, IOps, etc). Usually this was done by an external software but now this feature is inbuilt in kubernetes, enabling to get detailed statistics on each container running on the cluster.

- Gateways also have prometheus metric sources, with both the chirpstack agent endpoint and an instance of nodeexporter. In order to query these sources, usually inside an operator's network, we rely on the pushprox prometheus component [6], with components gateway and cluster side.

Prometheus time series will then be analyzed outside of the cluster. While it is possible to directly interrogate the prometheus instance in k8s, we should avoid putting stress on this instance and apply our algorithms on an external time series database. This is done by running a second instance of prometheus wherever the controller is hosted, and relying on prometheus federation to have the controller side prometheus instace scrape the time series of the edge cloud side prometheus instance.

5.2 Logs

We rely on fluentd in order to export the logs of all containers in the platform. Each worker node runs a copy of fluentd which accesses the logs files of the other containers running on the same machine. These logs are enriched with kubernetes metadata such as namespaces, labels or deployment modes, then forwarded to their destination outside the cluster. Many output formats are supported, and documented in [2].

Chapter 6

Instrumentalizing the platform

Figure 6.1 presents different interactions in the platform. In this section we detail in particular how the Application Programming Interfaces (APIs) of different elements can be exploited, how Kubernetes itself offers automation tools, and also precise some interesting prometheus features.

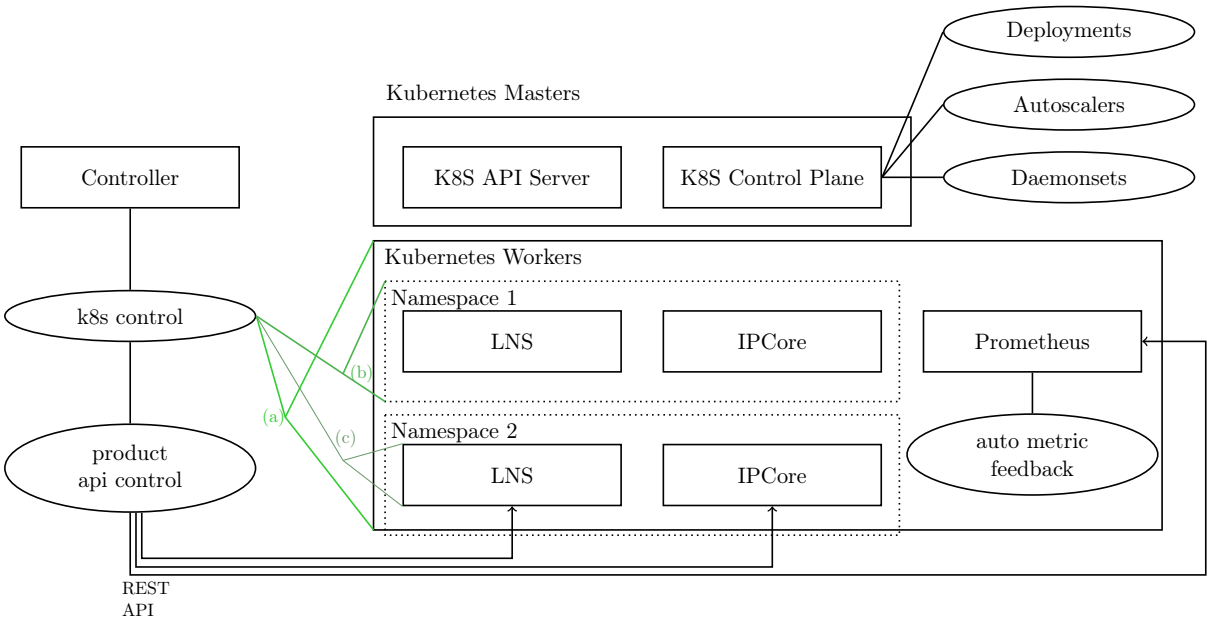

Figure 6.1: Platform control

6.1 Direct platform interaction

6.1.1 Kubernetes control

Many LNSs have made the choice to make their services available via series of docker containers. While this already helps deployment and administration by running these LNSs on docker hosts, they remain tied to a single Linux host and its limitations in terms of resources. In order to cope with use cases with high availability requirements or considerable traffic peaks, scaling up the docker deployment to multiple nodes is necessary. Technologies such as docker swarm or kubernetes provide this functionality; this project will use the latter.

The most straighforward way of interacting with this platform would therefore be through the kubernetes API, in the same manner one would administer any cluster. This is presented by the *k8s control* node of Figure 6.1. Kubernetes's very rich API is already detailed in [3], and we will not enter into details in this document. It is however relevant to note that one can directly call this API through an HTTP client as a standard Representational state transfer (REST) API, or by using its official client: *kubectl*.

Kubernetes comes with a pretty accurate role based permission system, allowing the administrator to determine the scope of what element of the cluster a user or a program is allowed to interact with, and with which actions. As shown in Figure 6.1 it is for example possible to give access to the entire cluster as shown in *(a)*, or to a given namespace *(b)*, or even to a series of elements, as for example in *(c)* the containers of an LNS (this is usually done by labeling objects when initially creating them).

6.1.2 Controlling deployed products

Figure 6.1 also presents how the platform controller can directly access the different products of the cluster.

- Both the LNS and the IPCore have REST APIs that can be instrumentalized by the controller in order to control the data flows. One could for example dynamically allocate channels to a slice by configuring them on the LNS, or create an entire IP network for a given family of devices by configuring both the IPCore and the LNS. Both products come with onboard API documentation provided by swaggerui [7].

- Prometheus can also be addressed through a REST API. This can in particular be useful to automatically mute alerts after the appropriate measures have been taken, but could also be relevant in order to configure alerts on the fly, or even dynamically configure new scraping targets (e.g. addition of a new agent on a shared gateway).

6.2 Kubernetes automation

Deploying containers on kubernetes is done by writing configuration elements to its control plane (the master nodes), which are then responsible to translating this configuration into active containers executed on the master nodes. These configuration elements are many, and control container related aspects, but also storage, networking, security, etc. The detail of these elements can be found in the official kubernetes documentation [4]. We will however shortly mention a few of them here.

- Deployments are some of the simplest elements. They represent a series of containers with their associated metadata, and how many replica of each are requested. This can for example be relevant when delay sensitive tasks need to be parallelised. This project's controller could for example use the API to adjust the *replica* parameter of a deployment in order to scale up or down its processing.

- Horizontal Pod Autoscaler are elements that allow the internal automation of such scaling up. They target a given workload and are given a metric threshold that each replica of this workload should not cross. If the threshold appears to be crossed, the autoscaler will then scale up the workload create new pods, thus balancing the load between more containers (and potentially, underlying machines).

- Daemonsets are kubernetes workloads that will ensure that every worker node in a collection (or all of them) run a given pod. This is in particular useful for prometheus gathering containers (each monitoring the worker node executes) or elements where proximity to all other pods is interesting (network ingress, database cache). In the latter, more accurate deployment strategies can be used in order to define affinities or anti-affinities between services.

It is possible to control, how such workloads are scheduled on nodes, by setting policies that place containers that share a lot of traffic on the same node, or on the contrary spread containers with high resource usage over different machines. It is also possible to restrict pod scheduling to reserved nodes, dedicating for example a set of worker nodes to a given slice.

Security wise, kubernetes can be configured to enforce namespace isolation. This lets the administrator assign containers to separate namespaces and prevent inter namespace network communication, thus making lateral movements much more difficult for attackers. Combined with slice-dedicated worker nodes, quite satisfactory separation can be achieved inside a cluster.

6.3 Prometheus automation

Since this project relies on prometheus in order to gather metrics, the prometheus ([5]) executor is worth mentioning. This service plugs itself in prometheus's alert routing, e.g. when a prometheus alert is created, instead, or in parallel of sending a mail, it may be sent to the prometheus executor for automated recovery actions. The executor will react to incoming alerts by executing predefined commands, with possible filtering depending on the labels of the alert.

Bibliography

[1] Chirpstack lns. https://www.chirpstack.io/.

[2] Fluentd output formats. https://docs.fluentd.org/output.

[3] Kubernetes api. https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[4] Kubernetes concepts. https://kubernetes.io/docs/concepts/.

[5] Prometheus executor. https://github.com/imgix/prometheus-am-executor.

[6] Prometheus pushprox. https://github.com/prometheus-community/PushProx.

[7] Swagger ui. https://swagger.io/tools/swagger-ui/.