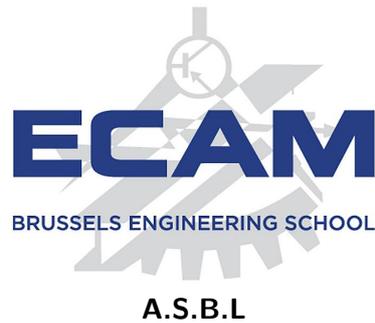


Haute École Léonard de Vinci



**Réalisation d'une architecture IoT distribuée
autonome basée sur les concepts de
micro-services.**

Travail de fin d'études présenté par

Selleslagh Tom

En vue de l'obtention du diplôme de

Master en Sciences de l'Ingénieur Industriel orientation Électronique

Promoteur : M. Laurent DERU

Tuteur : M. Sébastien COMBEFIS

Année académique 2018 - 2019

Abstract

Il y a près de quarante ans, la naissance d'Internet a permis l'émergence d'une nouvelle technologie qui consiste à exploiter la puissance de calcul et de stockage de machines distantes, par l'intermédiaire du réseau. Aujourd'hui, cette technologie est nommée le *cloud-computing*. Aujourd'hui, la quantité d'objets connectés ne cessant d'augmenter, conduisant à une explosion de la quantité de données à traiter, il devient indispensable de rapprocher le traitement et de mettre plus d'intelligence en périphérie du réseau afin de traiter plus efficacement ces données, d'éviter une saturation du réseau et garantir une plus grande réactivité. Cette nouvelle façon de travailler/ce nouveau paradigme est le *edge-computing*.

Concrètement, le travail réalisé dans ce TFE consiste à développer une architecture IoT présentant plusieurs passerelles de communication différentes : 6LoWPAN, LoRa et ZWave. Cette architecture se présente sous la forme d'un *cluster* de machines hétérogènes en terme de processeurs, de mémoires vives et d'interfaces. La gestion de son cycle de vie est réalisée par un orchestrateur. La « sortie » de notre *cluster* pointe vers une solution d'affichage, de traitement et de stockage basée, par exemple, sur **Grafana** ou **Prometheus**. Pour s'insérer dans une solution des plus génériques possible, tous les services utilisés dans le *cluster* se présentent sous la forme d'applications conteneurisées. On retrouve des services tels que : des passerelles vers des réseaux de capteurs sans fil, un broker Message Queuing Telemetry Transport (MQTT), une base de données locale, etc.

Une des spécificités de notre solution consiste à maintenir une continuité de service, et ce, même pendant les phases de mise à jour des services et des machines. Ceci est garanti grâce au déploiement automatique des services à jour ou en cours en fonction des *devices* connectés.

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon stage et qui m'ont aidée lors de la rédaction de ce mémoire.

Je voudrais dans un premier temps remercier mon promoteur de mémoire M. Laurent DERU, ingénieur de recherche expert au CETIC, pour sa patience, sa disponibilité et surtout ses judicieux conseils, qui ont contribué à alimenter ma réflexion.

Je remercie également mon superviseur, M. Sébastien COMBEFIS, d'avoir répondu à mes questions et poser les cadres de ce travail. Il a été d'un grand soutien dans l'élaboration de ce mémoire.

J'aimerais exprimer ma gratitude à tous les membres du CETIC trop nombreux pour les citer, qui ont pris le temps de discuter de mon sujet. Chacun de ces échanges m'a aidé à faire avancer mon projet.

Je remercie plus que tout ma femme qui, dans ce projet fou de reprendre une formation d'ingénieur industrielle, a su m'aider à garder le cap.

Cahier des charges relatif au travail de fin d'études de

Selleslagh Tom, inscrit en 2^{ème} Master, orientation électronique

- Année académique : 2018-2019
- Titre provisoire : Réalisation d'une architecture IoT distribuée autonome basée sur les concepts de micro-services.
- Objectifs à atteindre :

L'ambition du TFE est de réaliser une architecture distribuée pour la gestion de réseaux de capteurs IoT dont l'orchestration sera autonome. L'orchestrateur aura connaissance des spécificités des nœuds (architecture, mémoire vive ...) et saura aussi si un container peut-être nomade ou non c'est-à-dire s'il est lié à une interface physique ou pas. Le déploiement sera testé sur une architecture IoT présentant plusieurs passerelles de communication différentes (6LowPan, Lora, Zwave...). Il va sans dire qu'une des spécificités de ce cluster sera de maintenir une continuité de services, et ce, même pendant les phases de mise à jour.

- Spécificités :
 1. Services présents dans le cluster :
 - a. Base de donnée locale
 - b. Broker MQTT
 - c. Au moins 2 passerelles vers des WSN
 2. Déploiement automatiques des services
 3. Gestion autonome de la charge de travail au sein du cluster
 4. Auto-detection de la présence d'une interface physique sur un des noeuds
 5. Communication du cluster avec des application en dehors de celui-ci.
 6. Philosophie open-source:
 - a. Codes publics
 - b. Documentation complète

Fait en trois exemplaires à Charleroi , le 15 novembre 2018

L'étudiant
Nom – Prénom :
Selleslagh Tom

Le tuteur
Nom – Prénom :
Combéfis Sébastien

Le promoteur
Nom – Prénom :
Deru Laurent

Département/Unité :
Génie Électrique

Société :
CETIC asbl

Signature :

Signature :

Signature :

Table des matières

Abstract	i
Remerciements	ii
Tables des matières	vi
1 Introduction	1
2 Contexte et état de l'art	3
2.1 Architecture des processeurs	3
2.1.1 Architecture de type Reduced Instruction Set Computing (RISC)	4
2.1.2 Architecture de type Complex Instruction Set Computing (CISC)	5
2.1.3 Enjeux	6
2.2 Notion d'architecture distribuée	6
2.2.1 Micro-services conteneurisés	6
2.2.2 Orchestration	9
2.2.3 Choix des outils	10
2.3 Technologies utilisées pour la collecte de données	11
2.3.1 Technologie 1 : du 802.15.4 au COAP	11
2.3.2 Technologie 2 : Z-Wave (Z-Wave)	16
2.3.3 Technologie 3 : LoRa by LoRaWAN	19
2.4 Récapitulatif	21
3 Recherche de l'architecture optimale	22
3.1 Communication avec les réseaux de capteurs	22

3.1.1	Interface 6LoWPAN	22
3.1.2	Interface Z-Wave	25
3.1.3	Interface LoRa	26
3.2	Outils de conteneurisation	27
3.2.1	Stratégie de conteneurisation	27
3.2.2	Image de conteneur multi-architecture	29
3.3	Outils liés à l'orchestration des services	33
3.3.1	Comparaison de Docker Swarm et Kubernetes	33
3.3.2	Déploiement automatique de service	36
3.4	Intégration d'une application IPv6 dans un conteneur IPv4	39
3.4.1	Tayga	40
3.4.2	ToTD	41
3.4.3	Création de l'application 4LBR	41
3.5	Outils de traitements des données	45
3.5.1	Recherche d'un serveur LWM2M	45
3.5.2	Récupération des données du réseau Z-Wave	51
3.5.3	Récupération des données du réseau LoRaWAN	51
3.6	Choix du système de gestion de base de données locale	52
4	Solution concrète	54
4.1	Architecture matérielle distribuée	54
4.1.1	Le master	55
4.1.2	Les nœuds	55
4.2	Périphériques externes	56
4.2.1	Radio 802.15.4	56
4.2.2	Concentrateur Z-Wave	57
4.3	Architecture logicielle distribuée	58
4.3.1	de 802.15.4 à influxDB	59
4.3.2	de Z-Wave à influxDB	60
5	Perspectives futures	61
5.1	Phase de tests	61
5.1.1	Simulation d'un problème logiciel	61
5.1.2	Simulation d'un problème matériel	61
5.2	Intégration de la technologie basée sur LoRa	62
5.3	Intégration de services additionnels	63
5.3.1	Local Registry	63

5.3.2 Jenkins X	63
Conclusion	64
Bibliography	66
A Résultats d'expérimentations	69
A.1 Exclusion de Docker Swarm comme Orchestrateur	69
B Code sources	73
B.1 Fichier 6lbr.conf	73
B.2 Exemple de messages MQTT utilisés par EMQx	73
C Tableaux des figures	75
D Liste des tableaux	77
E Acronymes	78
F Glossaire	81

Chapitre 1

Introduction

La société actuelle tend à rendre plus intelligent un grand nombre d'objets, de bâtiments, de villes. Cette évolution passe par l'utilisation d'une profusion de capteurs ou d'actionneurs. Que l'on se trouve dans une implantation domestique ou industrielle, les informations récoltées par ces senseurs transitent généralement par Internet. Cette migration de l'utilisation d'Internet, par des objets, a vu l'émergence d'un paradigme : *l'Internet of Things (IoT)*.

" L'IoT peut être défini comme un réseau de réseaux permettant aux objets de s'identifier et de communiquer. Ces objets peuvent mesurer et échanger des données entre des mondes physiques et des mondes virtuels, au moyens de systèmes d'identification électronique normalisés et sans fils "(Achour et al., 2015).

Il est utopique de penser que toutes ces données peuvent être transmises de façon stable et rapide. L'utilisation massive du web, pour le développement d'architecture IoT, a des effets indésirables : Internet est saturé. De ce constat est né un second paradigme : Le *edge computing*.

Une simple analogie permet d'en comprendre les fondements : lorsque tous les navetteurs décident de se rendre à Bruxelles on observe une saturation du réseau routier. Si on donne la possibilité aux navetteurs de réaliser une partie de leurs tâches à domicile, l'engorgement diminue. Si Bruxelles désigne Internet (le Cloud) et les navetteurs représentent le flux d'informations transitant par Internet, alors le travail réalisé en bordure de la capitale peut symboliser la notion de *edge-computing*.

Ce projet a pour but de démontrer qu'une architecture IoT peut s'insérer dans une philosophie de *edge-computing*. Il s'agit de récolter des données en provenance de différents réseaux de capteurs, de les traiter localement et de les stocker dans une base de données. L'innovation de ce travail prend sa source dans la distribution de la charge de travail et dans l'automatisation du déploiement des éléments du réseau.

Une rapide analyse des besoins nous permet de définir certaines caractéristiques de l'architecture distribuée de ressources logicielles et matérielles envisagée :

- L'objectif est de simuler un parc informatique hétérogène. Il convient donc d'utiliser des machines différentes en terme de processeurs ou de mémoire RAM.
- La récolte de données doit passer par plusieurs protocoles de communication différents
- L'utilisation de services nécessitant une connexion Internet sont à éviter. De ce fait, un solution de stockage local doit être envisagée.
- Un utilisateur externe doit pouvoir interagir avec le flux de données, un système d'affichage des données doit donc être intégré.
- Comme l'architecture logicielle envisagée est distribuée, l'approche de développement s'appuie sur les concepts de microservices.
- Pour garantir le couplage le plus faible possible entre les services, les applications doivent prendre la forme de conteneurs.
- Plusieurs services doivent coexister et être déployés dans l'architecture logicielle, un solution d'orchestration doit donc être utilisée.

Ce travail est subdivisé en plusieurs phases. La première étape est une recherche documentaire au terme de laquelle des choix sont posés. Ces décisions ont permis de définir une trajectoire cohérente au projet. S'en suit une phase de recherche et développement dont les prérogatives sont de tester différents outils et d'implémenter les services indispensables précédemment identifiés. Au terme des jalons de ce projet, la solution dans sa version actuelle est présentée. Un dernier point est consacré aux perspectives d'évolution du projet.

Chapitre 2

Contexte et état de l'art

Ce chapitre a pour vocation de définir le contexte dans lequel s'insère ce travail. Pour ce faire, une première partie est consacrée aux architectures de processeurs que l'on peut retrouver dans les machines susceptibles de faire partie de ce projet.

Un deuxième volet revient sur les concepts clés des architectures distribuées (logicielles et matérielles) dont les applications prennent la forme des micro services conteneurisés. Une discussion est réalisée pour argumenter le choix des outils informatiques utilisés lors de la réalisation de ce travail.

Le dernier point de ce chapitre présente les trois technologies de récolte de données qui ont été sélectionnées. La compréhension de celles-ci permet de définir plus précisément les services qui doivent être intégrés dans notre architecture distribuée de ressources logicielles.

2.1 Architecture des processeurs

L'hétérogénéité des parcs informatiques en termes de processeurs trouve surtout son origine dans l'évolution des produits présentés par quatre grandes sociétés : Advanced Risk Machine (ARM), International Business Machines Corporation (IBM), Intel Corporation et Advanced Micro Devices (AMD). Il ne s'agit pas de faire une comparaison exhaustive ni de mesures comparatives de différentes technologies de Central Processing Unit (CPU). L'objectif est d'éclairer sur les différences les plus marquantes et de montrer les architectures matérielles et les contraintes qu'ils engendrent.

2.1.1 Architecture de type RISC

L'approche RISC se caractérise par la volonté d'une optimisation du jeu d'instruction. Ne sont donc implantés en machine que les mécanismes réellement utiles, ceux dont on a statistiquement montré l'utilité (Cazes and Delacroix, 2008). La structure interne du processeur permet donc d'exécuter directement chaque instruction sans pré-codage.

Cette architecture s'appuie sur un jeu d'instructions simple¹ de longueur fixe. Ce design de processeurs prévoit que seules les opérations d'écriture et de lecture accèdent à la mémoire. Les opérations de traitement quant à elles ne travaillent qu'avec les registres (Jorda, 2010).

La simplicité de son jeu d'instructions et de son mode d'adressage permet à la plupart des instructions de s'exécuter en un seul cycle mémoire. Les instructions n'ont généralement que trois adresses : deux registres servant d'opérandes et un registre de sortie (Patterson and Sequin, 1981).

ARM

En avril 1985, l'un des premiers processeurs RISC voit le jour : l' ARMv1. En raison de leur faible consommation électrique, les processeurs ARM sont rapidement devenus incontournables dans le domaine de l'informatique embarquée (mobile, tablette, micro-pc...)

Aujourd'hui, *ARM Ltd* propose la huitième itération de son cœur. Cette version est un réel tournant dans cette gamme de CPU, puisqu'elle propose désormais une architecture RISC dont la taille des instructions est de 64 bits.

IBM

La société IBM n'est pas en reste, dès 1990, elle met sur le marché sa vision du processeur RISC : IBM POWER². IBM se démarque, dès sa troisième mouture, en proposant un CPU travaillant avec des instructions d'une longueur de 64 bits³.

À l'heure actuelle, IBM vise un marché plus professionnel avec sa neuvième version du IBM POWER. Ce dernier est conçu, avant tout, pour des utilisations nécessitant de grande puissance de calculs (intelligence artificielle, informatique cognitive, datacenters, Cloud Computing...) (Sadasivam et al., 2017).

1. Arithmétique, logique et décalage

2. On regroupe souvent les machines incluant ce type de processeur sous le nom de PowerPC

3. Présent, entre autres dans les ordinateurs personnels proposés par la firme *Apple* jusqu'en 2006

2.1.2 Architecture de type CISC

L'approche CISC se caractérise par une volonté de rendre le matériel et le logiciel indépendants. On fait donc correspondre à chaque structure de données, exprimées dans un langage de haut niveau, un mode d'adressage adapté dans le langage machine (Cazes and Delacroix, 2008).

L'architecture s'appuie sur un jeu d'instructions riches et complexes dont la longueur n'est pas figée. Pour exemple, une opération sur un nombre à virgule flottante est réalisée en une seule instruction alors qu'il en faut des dizaines sur un processeur de type RISC de première génération (George, 1990).

Ce design de processeurs prévoit que toutes les instructions sont susceptibles d'accéder à la mémoire. Il utilise donc très peu de registres (Monnier, 2019).

Intel Corporation et AMD

En 1978, Intel présente le intel 8086 : un processeur CISC conçu pour des instructions de maximum 16 bits (Morse et al., 1978). C'est sur base de ce CPU que la famille de processeur x86 voit le jour. Il désigne tout processeur compatible avec le jeu d'instruction du 8086⁴. Ce processeur n'a cependant pas été imaginé par intel, mais par AMD , qui à l'époque conçoit des CPU sous licence pour intel.

Les deux sociétés prennent leur distance à la fin des années 90 avec le processeur Athlon pour AMD et le Pentium pour Intel. La série 64 bits d'AMD, baptisée AMD64, est proposée au public dès l'automne 2003. Intel répond en 2004 en faisant une mise à jour de son processeur : le Pentium 4. Dans ce cas de figure, l'évolution réside surtout dans la mise à jour du jeu d'instruction x86 qui permet entre autres la gestion des nombres sur 64 bits. C'est sur base de ce jeu d'instruction que sont regroupés les CPU de la famille x86-64

Malgré les différences présentes dans le jeu d'instruction de ses processeurs, les compilateurs sont capables de produire des exécutables qui fonctionnent sans modifications sur ces différentes architectures de type CISC.

4. qui est l'architecture de processeur la plus répandue encore aujourd'hui dans les ordinateurs de bureau.

2.1.3 Enjeux

Ce tour d'horizon des acteurs majeurs dans la conception de CPU permet de montrer que l'apparition des processeurs RISC, au début des années 80, a fortement remis en cause l'architecture CISC. L'importance de cette reconsidération a eu l'effet d'une révolution et non sans conséquences : il n'est pas rare de se retrouver avec un parc informatique présentant des technologies de CPU différentes.

Il est donc essentiel que l'ensemble des applications réalisées et utilisées durant ce travail soit conçu pour être *cross-architecture* c'est à dire fonctionnelles, peu importe le type de processeur présent sur les machines prévues lors du déploiement.

2.2 Notion d'architecture distribuée

L'architecture matérielle et logicielle distribuée envisagée prend la forme d'un *cluster*, que l'on peut définir comme un groupe de ressources agissant en un seul et même système. Il affiche ainsi une disponibilité élevée, voire dans certains cas, des fonctions de traitement en parallèle et d'équilibrage de la charge. Généralement dans un *cluster*, les applications sont sous la forme de services conteneurisés dont le cycle de vie est géré par un orchestrateur.

2.2.1 Micro-services conteneurisés

Une machine virtuelle est un ordinateur créé à l'intérieur d'un ordinateur. D'un point de vue de son implementation, elle se présente sous la forme d'un fichier informatique appelé une image. L'un des avantages de ce type de technologie est la possibilité d'utiliser la même image sur plusieurs types de matériel, sans avoir à installer de nouveaux pilotes.

Cependant, comme nous le montre la figure 2.1, la virtualisation implique de créer des instances virtuelles spécifiques d'une infrastructure matérielle. A contrario, la conteneurisation crée simplement des conteneurs isolés au niveau du système d'exploitation. Ces derniers sont gérés par une application tierce appelée *Conteneur Runtime*. L'Operating System (OS) est partagé par différents conteneurs plutôt que d'être cloné pour chaque machine virtuelle.

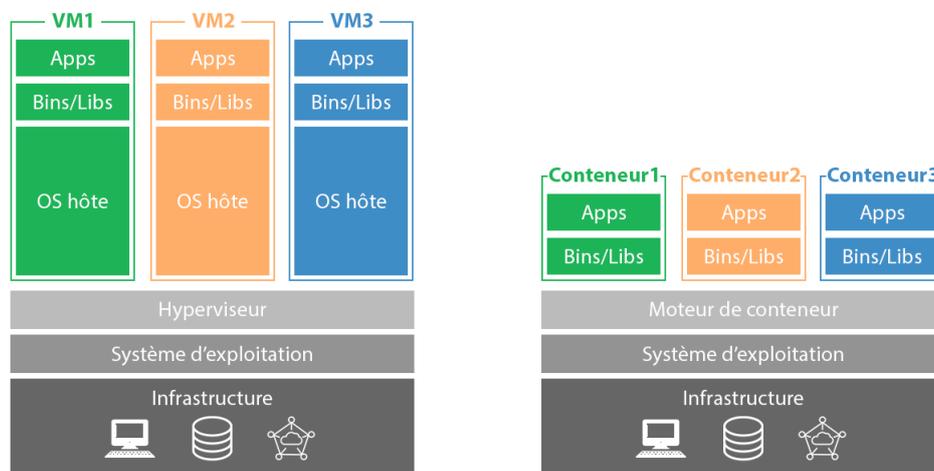


FIGURE 2.1 – Comparaison entre utilisation de machines virtuelles et de conteneur

Les conteneurs offrent donc une alternative légère, portable et de hautes performances à la virtualisation. La taille de l'image du conteneur, qui est plus petite que celle des images de machine virtuelle (VM)⁵, permet de lancer des applications plus rapidement qu'un périphérique basé sur une machine virtuelle⁶ et d'avoir plus de services sur une même machines (Ismail et al., 2015).

De fait, comme énoncé précédemment, chaque conteneur est isolé. De plus, il a son propre sous-système indépendant de réseau, de mémoire et de système de fichiers. Comme les conteneurs partagent le même système d'exploitation, ils ont l'avantage d'offrir des performances presque natives en ce qui concerne le CPU, la mémoire, le disque et le réseau (Bernstein, 2014).

Un conteneur comprend également un environnement d'exécution complet : une application accompagnée de toutes ses dépendances (bibliothèques, binaires, et fichiers de configuration) rassemblées dans ce que l'on appelle une *image de conteneur*. L'un des inconvénients, dans l'utilisation d'applications conteneurisées, se situent justement à ce niveau. La création, la gestion et l'administration de conteneurs demande souvent plus de temps que de simplement exécuter une application dans une machine virtuelle, surtout pour un administrateur qui ne serait pas familier avec le système d'exploitation Linux.

5. Quelques dizaines de mégabytes, là où avec une machine virtuelle avec son système d'exploitation complet embarqué, on pourrait monter jusqu'à plusieurs gigabytes

6. Le fait que le déploiement d'un conteneur s'effectue sur un OS déjà opérationnel joue aussi un rôle déterminant sur ce temps de lancement.

2.2.1.1 Conteneur runtime de bas niveau

On peut définir le *container runtime* comme une interface logicielle du système d'exploitation, utilisé pour exécuter des commandes de cycle de vie sur une instance de conteneur tel que, créer, démarrer, mettre en pause, reprendre, arrêter ou encore supprimer.

Il y a un peu moins de 5 ans, le seul outil connu pour lancer et administrer des conteneurs était **Docker**. On ne parlait donc pas encore de *container runtime*. Cependant, l'arrivée de **Rocket (RKT)**, mis en évidence par **CoreOS**, a changé la donne. Leur idée était d'offrir une alternative à **Docker** en offrant une meilleure gestion de la sécurité et un respect de la philosophie KISS⁷ d'Unix.

Pour éviter une trop grande divergence des technologies, l' Open Container Initiative (OCI) a été créé en juin 2015. Son rôle a été d'implémenter un standard, spécifiant la manière de démarrer un conteneur⁸. Un *container runtime* de bas-niveau, basé sur les travaux de **Docker** a donc vu le jour : **runC**. A l'heure actuelle, **runC** est le *container runtime* le plus utilisé (Guichard, 2018).

2.2.1.2 Container runtime de haut-niveau

Les *containers runtimes* de bas-niveaux ne prennent pas en charge la gestion des images, les *pull*⁹ et ne fournissent pas non plus d'API, comme une interface en ligne de commande. Toutes ces tâches sont réalisées par un ou plusieurs autres *containers runtimes* de haut-niveaux. La figure 2.2 illustre l'interaction des différents *containers runtimes*.

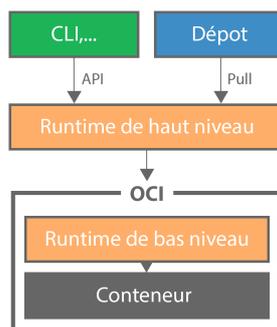


FIGURE 2.2 – Interactions entre les *containers runtimes*

7. Keep it simple, stupid

8. L'OCI a également standardisé le format des images de conteneurs

9. Récupération d'une image depuis un dépôt distant

2.2.2 Orchestration

Les architectures distribuées (logicielles et matérielles) reposent sur la possibilité d'utiliser des objets, s'exécutant sur des machines réparties sur un réseau et communiquant par messages au travers de celui-ci. D'un point de vue du vocabulaire, les objets utilisés sont appelés *Pods*. Ces *Pods* contiennent, dans la solution proposée, une (ou plusieurs) application conteneurisée. Les différentes machines sur lesquels sont déployés les *Pods* sont, quant à elles, appelées *nodes* ou nœuds.

L'orchestration désigne le processus d'organisation du travail de cette architecture distribuée, aussi bien du point de vue logiciel que matériel. Les outils fournis par l'orchestrateur permettent de gérer le déploiement de *Pods* sur les différents *nodes*, d'automatiser leurs mises à jour, de contrôler leur état et de relancer un *Pods* qui serait dans un état d'erreur. Certains orchestrateurs offrent également des services de gestion de la scalabilité c'est à dire, une répartition homogène de la charge de travail entre les *nodes*. Généralement, toutes les tâches de gestion du *cluster* sont réalisées par un *node* avec des privilèges élevés : le **master**. À noter que ces services d'orchestration sont, eux aussi, sous la forme de *Pods*. La figure 2.3 présente le schéma de principe d'un *cluster* orchestré.

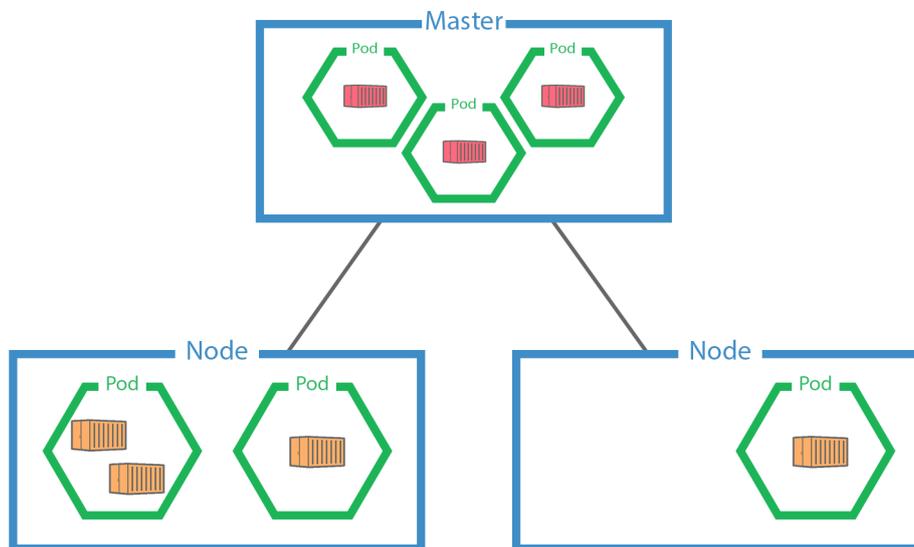


FIGURE 2.3 – Schéma de principe d'un *cluster* orchestré

2.2.3 Choix des outils

Les sections précédentes viennent de démontrer que la réalisation d'une architecture distribuée, aussi bien d'un point de vue logiciel que matériel, repose sur une série d'outils. Il est possible pour chacun d'eux d'en trouver plusieurs déclinaisons. Les sections suivantes présentent les choix et ce qui les a motivés pour chaque type d'outils nécessaires.

Le choix des outils se fait plutôt de l'orchestrateur en descendant jusqu'au *container runtime* de bas niveau. Cette manière de faire permet de poser la priorité sur le choix de l'orchestrateur.

2.2.3.1 Choix de l'orchestrateur

L'un des objectifs de ce travail est de proposer une solution la plus générique possible, cependant, certains orchestrateurs imposent un système d'exploitation sur la machine hôte. Deux orchestrateurs ont donc été écartés pour cette raison :

- **OpenShift** qui impose Red Hat Enterprise Linux 7
- **Marathon** qui impose DC/OS

Pour sa part, **CoreOS** semble se désintéresser de l'élaboration d'orchestrateur, en effet le créateur du *container runtime* RKT qui proposait **Fleet**, sa vision de l'orchestration, informe ses utilisateurs qu'il ne sera plus mis à jour et leur propose de se tourner vers **Kubernetes** : *"While fleet and Kubernetes are similar in functionality, some fleet features do not have direct equivalents in Kubernetes. Workarounds exist for many of these cases. Several of these features and workarounds are outlined below"* (CoreOS, 2017).

Les deux acteurs majeurs à départager dans le monde de l'orchestration sont donc **Kubernetes** et **Docker Swarm**.

2.2.3.2 Choix des *containers runtimes*

Les outils utilisant des standards offrent l'avantage de pouvoir normaliser l'intégration des services et garantissent une certaine interopérabilité. De ce fait, les outils s'écartant des standards ne sont pas choisis. **RKT**, ne supporte que la spécification du format d'image défini par l'OCI sans en implémenter la spécification d'exécution, n'est donc pas retenue comme *container runtime*.

Compte tenu de ce qui a été expliqué au point 2.2.1.1, l'utilisation de **runC** est privilégiée comme *container runtime* de bas niveau.

Il ne reste qu'à déterminer le *conteneur runtime* de haut niveau. On peut noter que **Kubernetes** a implémenté une interface qui définit la façon dont il s'adressera aux *containers runtimes* : Container Runtime Interface (CRI). Implémenté à la base par *Docker*, **Containerd** est désormais un projet de la Cloud Native Computing Foundation (CNCF). Compatible avec **runC** et implémentant la CRI, il fournit toutes les fonctionnalités de plus haut niveau, à l'exception de la construction d'image de conteneurs.

2.3 Technologies utilisées pour la collecte de données

Face à la pléthore de protocoles de communication proposés à l'heure actuelle, il devient de plus en plus difficile de choisir une technologie pour la récolte de données provenant de capteurs. Les trois protocoles qui ont été sélectionnés permettent de mettre en évidence une approche regroupant un maximum de cas d'utilisation. Ces derniers sont également utilisés par le CETIC¹⁰ dans le cadre de missions industrielles :

- **6LoWPAN** est utilisé par le CETIC de manière pro-active
- **Z-Wave** est l'un des protocoles les plus utilisés dans la domotique domestique.
- **LoRa** est une technologie offrant un réseau à longue portée.

2.3.1 Technologie 1 : du 802.15.4 au COAP

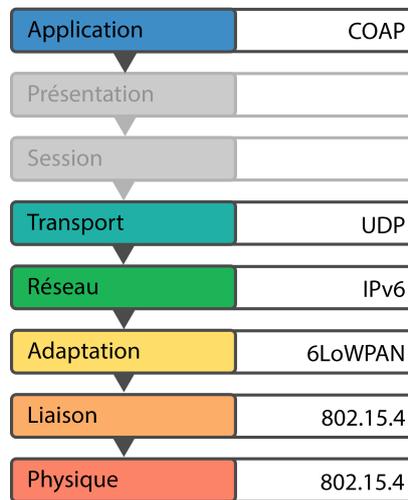
La première technologie implémentée dans la solution proposée par ce travail, peut être détaillée suivant le modèle OSI, dans lequel les couches de session et de présentation sont vides (cf. figure 2.4). La communauté qui entoure cette *protocol stack* est fortement soutenue par le CETIC.

2.3.1.1 Couche applicative (7) : Constrained Application Protocol (CoAP)

Les réseaux contraints tels que IPv6 Low power Wireless Personal Area Networks (6LoWPAN) prennent en charge la fragmentation des paquets IPv6 en petites trames de couche de liaison. Ceci a pour conséquence d'augmenter la probabilité de collision des messages et de ce fait, d'en diminuer la probabilité de livraison.

L'un des objectifs des concepteurs du CoAP était de limiter la surcharge des messages, diminuant ainsi le besoin de fragmentation. Le but de CoAP est de compresser les messages

10. Entreprise dans laquelle ce travail a été réalisé et supervisé

FIGURE 2.4 – *Protocol stack* de la technologie 1

de type HyperText Transfer Protocol (HTTP), en réalisant un sous-ensemble du style d'architecture logicielle Representational State Transfer (REST), optimisé pour les applications machine-to-machine (M2M) (Shelby et al., 2014).

Les principales caractéristiques de CoAP définies dans la littérature sont les suivantes :

- Protocole web répondant aux exigences M2M des environnements contraints
- Liaison de type User Datagram Protocol (UDP) avec fiabilité optionnelle prenant en charge les requêtes unicast et multicast.
- échanges de messages asynchrones.
- Faible surcharge d'en-tête
- Prise en charge des en-têtes de types *URI* et *Content-Type*.
- Proxy simple et capacités de mise en cache.
- Un mappage HTTP sans état permettant de créer des Proxys et donnant accès aux ressources CoAP via HTTP.
- Protocole de sécurité par Datagram Transport Layer Security (DTLS)¹¹

11. Cette caractéristique est optionnelle

2.3.1.2 Couche de transport (4) et réseau (3) :

Au niveau de la couche de transport, les messages CoAP sont placés dans des datagrammes UDP, ce qui permet d'économiser la bande passante et d'apporter le support du multi-cast.

Au niveau de la couche réseau, les datagrammes UDP sont placés dans des paquets IPv6. Le protocole IPv6 est conçu pour succéder à IPv4 et permet de pallier la raréfaction de l'espace d'adressage. Avec certitude, on peut affirmer que les appareillages et les instruments en réseau dépasseront considérablement les hôtes d'ordinateurs classiques. IPv6 étend l'espace d'adressage IP de 32 à 128 bits.

Dans le cas de réseau *mesh*, un routage est nécessaire. Si ce dernier est réalisé au niveau de la couche réseau, le paquet IPv6 est reconstitué sur chaque équipement intermédiaire, afin de prendre la décision de routage. Cette méthode, appelée *route-over*, est plus efficace dans des conditions dégradées du réseau.

2.3.1.3 Couche d'adaptation : 6LoWPAN

6LoWPAN est une couche d'adaptation pour IPv6 sur réseau limité, comme 802.15.4. Elle se situe entre la couche réseau et la couche de liaison du modèle OSI. On pourrait la qualifier de couche 2,5 (Hui et al., 2009)

En 802.15.4, la taille maximale du Physical layer Service Data Unit (PSDU) est de 127 octets. Dans ces conditions, on ne respecte pas les spécifications d'IPv6 qui imposent un Maximum Transmission Unit (MTU) minimal de 1280 octets. Pour s'adapter au support 802.15.4, l'équipement doit fragmenter un paquet IPv6 en plusieurs trames 802.15.4. L'équipement distant doit, quant à lui, réassembler toutes les trames 802.15.4 reçues pour régénérer le paquet IPv6 d'origine.

Dans ces conditions, sur les 127 octets seulement 21 représentent des données utiles lorsque le protocole de transport utilisé est TCP. On peut faire grimper ce chiffre à 33 en utilisant le protocole UDP (cf. point a. de la figure 2.5).

Pour avoir un meilleur taux d'encapsulation, la couche d'adaptation 6LoWPAN peut compresser l'en-tête IPv6. Cette méthode permet d'augmenter la taille des données utiles à 75 octets dans le meilleur des cas. Par contre, si l'on rajoute les informations de fragmentation, ce chiffre baisse à 70 octets. Ce qui représente, quand même, presque le double de données utiles par paquet, par rapport à la méthode sans compression des en-têtes comme le montre la figure 2.5.

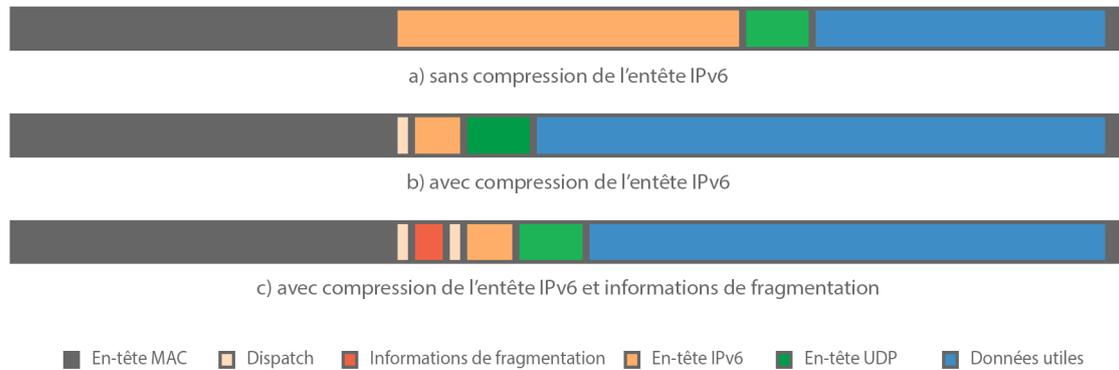


FIGURE 2.5 – Datagramme de paquet 802.15.4 après fragmentation par la couche d'adaptation

Si le routage est réalisé au niveau de la couche d'adaptation, la décision de routage se fait au niveau 6LoWPAN et donc seulement avec les fragments du paquet IPv6. Cette méthode est appelée *mesh-under* et permet d'avoir un délai de transmission plus court puisque le paquet IPv6 n'est reconstitué que sur l'équipement destinataire.

2.3.1.4 Couche de liaison

La norme IEEE 802.15.4¹² supporte deux types de topologie de réseau, à savoir la topologie en étoile et la topologie en réseau maillé (*mesh*). Dans ce projet, le protocole d'accès utilisé est le Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). Ce dernier utilise un mécanisme d'évitement de collision très simple : le nœud voulant émettre vérifie l'état du réseau, si celui-ci est occupé la transmission est différée.

La couche de liaison de ce standard définit deux types de nœuds :

- les Full Function Devices (FFD) sont équipés d'un ensemble complet de fonctions et fournissent des services de synchronisation, de communication et de liaison de réseau.
- les Reduced Function Devices (RFD) ne peuvent servir que de *end-device* et n'interagissent qu'avec un seul FFD

12. L'IEEE 802 est un comité de l'IEEE qui décrit une famille de normes relatives aux réseaux locaux et métropolitains basés sur la transmission de données numériques par des liaisons filaires ou sans fil.

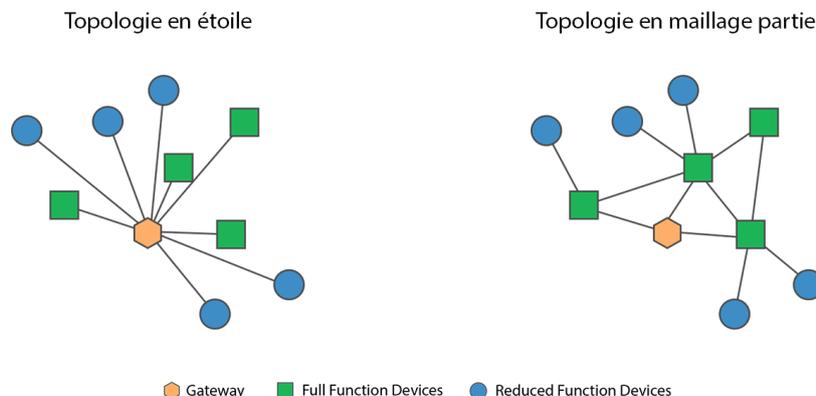


FIGURE 2.6 – Topologie de réseau supporté en 802.15.4

En fonction de la topologie choisie, les FFD pourront avoir différentes fonctions : dans la topologie en étoile, un des FFD assume la fonction de coordinateur, les autres nœuds ne communiquent qu'avec lui. Dans la topologie en *mesh*, un FFD peut communiquer avec d'autres FFD s'ils sont situés dans sa couverture radio. Il peut également relayer des messages vers d'autres nœuds (Baronti et al., 2007).

Cependant, si ce réseau est composé de FFD et de RFD, on est plutôt en présence d'un réseau maillé partiel (*partial-mesh*). Ces deux modes de fonctionnement sont présentés à la figure 2.6.

2.3.1.5 Couche physique

La couche physique prend en charge trois bandes de fréquences dans la variante de 802.15.4 utilisée :

- 16 canaux dans la bande de fréquence de 2,4 à 2,4835 GHz avec une modulation par décalage de phase en quadrature (QPSK).
- 10 canaux dans la bande de fréquence de 902 à 928 MHz avec une modulation par décalage de phase binaire (BPSK) ,
- 1 canal dans la bande de fréquence de 868 à 868,6 MHz avec une modulation BPSK .

La couche physique prend en charge les fonctionnalités de sélection de canal et d'estimation de la qualité de la liaison.

2.3.2 Technologie 2 : Z-Wave

Z-Wave est un protocole de communication principalement utilisé pour des fonctions de surveillance et de petites installations commerciales ou privées. Z-Wave est la technologie la plus utilisée pour ce type d'installation, les chiffres sont là pour le prouver puisque 100 millions de produits Z-Wave seraient aujourd'hui sur le marché à travers le monde (zwaveme, 2017). Son utilisation massive rend cette technologie presque incontournable dans un travail comme celui-ci. En outre, Z-Wave est largement utilisé dans plusieurs projets du CETIC.

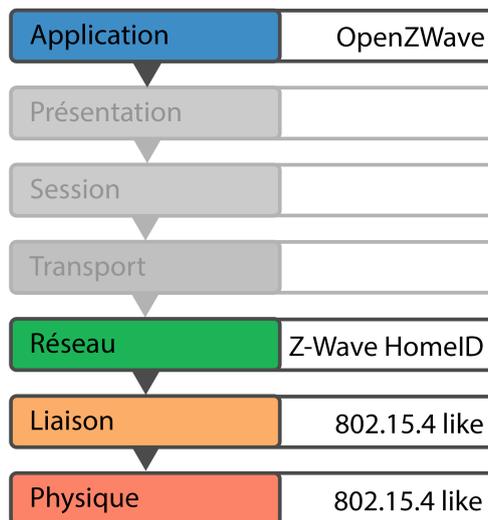
Cette *protocole stack* garantit une interopérabilité de ses *devices*. Pour ce faire, les fabricants doivent passer par la certification de leurs produits par l'Alliance Z-Wave avant leur mise sur le marché. Le produit certifié sera reconnaissable grâce à un logo (présenté à la figure 2.7). Pour faciliter l'interopérabilité, le protocole permet de préciser le type d'équipement avec une notion de classes (interrupteur binaire, capteur binaire, capteur multi niveaux, moteur multi niveaux, thermostat, alarme ...).

Dernièrement le protocole Z-Wave a subi une grosse mise à jour : Z-Wave+. Cette dernière apporte des améliorations au système en termes de bande passante et d'autonomie pour les modules non filaires. On notera que la rétrocompatibilité est totale avec les modules Z-Wave classiques. (Wiki, 2017)



FIGURE 2.7 – Badges officiels apposés sur les modules certifiés Z-Wave et Z-Wave+

Le protocole sera détaillé en essayant de tendre le plus possible vers un modèle se rapprochant de celui utilisé pour décrire la première technologie.

FIGURE 2.8 – *Protocol stack* de la technologie 2

2.3.2.1 Couche applicative : OpenZWave

Z-Wave est un protocole de communication sans fil propriétaire, on retrouve néanmoins une forte communauté derrière un projet open-source : OpenZWave.

"L'objectif du projet est d'apporter une contribution positive à la communauté Z-Wave en créant une bibliothèque qui prend en charge autant que possible la spécification Z-Wave, et qui peut être utilisée comme solution de type blackbox par quiconque souhaiterait ajouter Z-Wave à son application" (ozw, 2019).

La majorité du protocole Z-Wave a été publiée dans le domaine public en septembre 2016. Depuis lors, les mises à jour d' OpenZWave visent essentiellement la conformité aux spécifications Z-Wave publiées. Pourtant, il existe encore une partie du protocole qui n'a pas été publiée dans le domaine public à savoir, le protocole spécifiant comment parler à une clé USB via le port série COM. Cette implémentation dans OpenZWave s'appuie sur un travail de *rétro-engineering*.

2.3.2.2 Couche réseau

Le protocole z-wave utilise un champ spécifique pour séparer différents réseaux : Home ID. Ce dernier est un identifiant unique de 32 bits¹³ préprogrammé dans tous les dispositifs de contrôle. Le nombre d'appareillages sur un même réseau Z-Wave ne pourra pas excéder 232 appareils physiques (lotPoint, 2015).

2.3.2.3 Couche de liaison

Les spécificités de la couche de liaison sont fort proches de celles offertes par le standard 802.15.4. ZWave opère en réseau partiellement maillé (cf. figure 2.6) dans lequel les RFD sont appelés des esclaves. Les esclaves effectueront des actions en fonction des commandes qu'ils reçoivent. Ils ne seront pas toujours capables de transmettre des informations directement aux autres nœuds toutefois, s'ils sont en mesure de stocker des tables de routage, ils pourront agir comme répéteur.

Un autre type de nœud présent dans les réseaux Z-Wave est le contrôleur, équivalent des FFD en 802.15.4. Il existe toutefois une hiérarchie, le contrôleur qui crée initialement le nouveau réseau z-wave deviendra le contrôleur principal, les autres seront appelés contrôleurs secondaires. Chaque contrôleur dispose d'une table de routage complète du réseau maillé, il pourra donc communiquer avec tous les nœuds présents (lotPoint, 2015).

On notera que le contrôleur principal a des fonctionnalités plus spécifiques que les autres contrôleurs tel que :

- maître du réseau (unique)
- inclusion et exclusion des nœuds.
- gestion de l'allocation des ID des nœuds.
- connaissance de la topologie la plus récente du réseau.
- communication du home ID aux nœuds esclaves

13. à noter que les adresses IPv4 sont aussi codées sur 32 bits

2.3.2.4 Couche physique

À l'instar du standard IEEE 802.15.4, ZWave opère dans la bande de fréquence de 868 MHz. La littérature ne définit cependant pas la largeur de bande ni le nombre de canaux disponibles. Les débits de données peuvent aller de 40 à 100 kbps.

Récemment, l'Union internationale des télécommunications a intégré la couche physique et la couche liaison Z-Wave dans sa nouvelle norme G.9959. Cette dernière définit un ensemble de directives pour les dispositifs sans fil à bande étroite de moins de 1 GHz (Berdyeva, 2015).

2.3.3 Technologie 3 : LoRa by LoRaWAN

LoRa est une technologie de modulation radio sous licence de Semtech Corporation. Elle offre une connectivité de longue portée : 5km en milieu urbain et 15km en milieu rural. La modulation du signal est une modulation en fréquence avec étalement du spectre. Les bandes de fréquence pouvant être utilisées sont : 433, 868 et 915 MHz (Zhao et al., 2017). Le protocole de communication le plus largement utilisé pour la mise en place d'un réseau LoRa est LoRaWAN.

2.3.3.1 Spécifications LoRaWAN : spécificités du réseau

La topologie du réseau est une topologie de type étoile : Les *end-devices* communiquent directement avec une passerelle (appelées également concentrateurs). Les données sont ensuite transmises à un serveur.

Les transmissions sont bidirectionnelles et peuvent avoir une taille de 19 octets à 250 octets . Lorsque l'on utilise un réseau LoRa, la bande passante et les facteurs d'étalement sont configurables. Il est donc possible de choisir le débit de données, ce qui offre un plus grand contrôle de la consommation des périphériques.

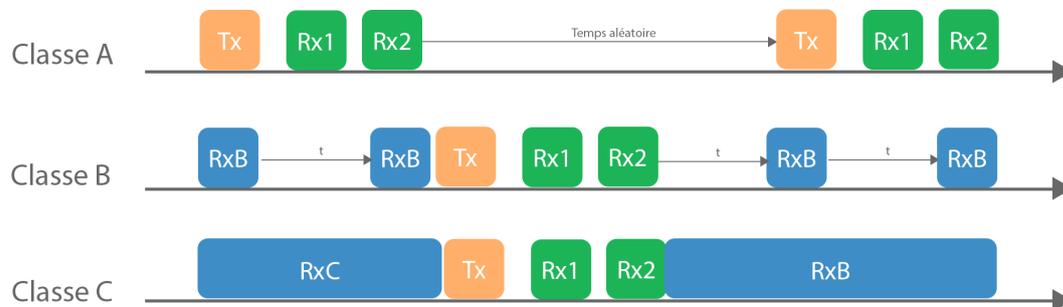


FIGURE 2.9 – Schéma de principe des communications LoRa en fonction des classes de périphériques

2.3.3.2 Spécifications LoRaWAN : spécificités des périphériques

La spécification LoRaWAN définit trois types de périphériques :

- Les périphériques de classe A prennent en charge la communication bidirectionnelle entre un périphérique et une passerelle. Des messages de liaison montante (de l'appareil au serveur) peuvent être envoyés à tout moment, et ce de manière aléatoire dans le temps. Le périphérique ouvre ensuite deux fenêtres de réception à des temps spécifiées (Rx1 et Rx2 sur la fig 2.9). Le serveur peut répondre soit dans la première fenêtre de réception, soit dans la seconde, mais ne doit pas utiliser les deux fenêtres. On notera que tous les périphériques LoRaWAN doivent implémenter cette classe. Les classes B et C ne seront quant à elles que des extensions de la classe A.
- Les périphériques de classe B étendent la classe A en ajoutant des fenêtres de réception planifiées pour les messages en liaison descendante (provenant du serveur) c'est-à-dire qu'il n'est pas obligatoire d'attendre que l'appareil commence à communiquer pour pouvoir lui envoyer des messages. Ces fenêtres sont ouvertes périodiquement à l'aide de balises synchronisées dans le temps et transmises par la passerelle (RxB sur la fig 2.9) .
- Les périphériques de classe C étendent la classe A en gardant les fenêtres de réception ouvertes sauf lorsqu'elles transmettent (RxC sur la fig 2.9). L'appareil ne passera jamais en mode veille, cela permet une communication à faible temps de latence. Notons que cela a pour conséquences une augmentation de la consommation énergétique.

2.3.3.3 Spécifications LoRaWAN : Réseaux

La spécification LoRaWAN définit deux types de réseaux :

Le premier type de réseau est un réseau de type *public*. Il existe deux moyens pour connecter ses périphériques à ce type de réseau :

- en passant par un opérateur téléphonique national. On notera que l'utilisation de ce type de réseau et les services fournis par ces opérateurs sont généralement payants.
- en utilisant le réseau LoRaWAN open source The Things Network qui peut être utilisé sans contrainte commerciale. Il s'agit d'un réseau dont la couverture s'étend sur 137 pays. Il se base sur le déploiement personnel de gateway partagée. On en compte à l'heure actuelle un peu plus de 7000 actives. (TTN, 2019)

Le deuxième type de réseau est un réseau de type privé. Il peut être exploité par des particuliers ou par des entreprises dans un contexte industriel (Nolan et al., 2016). Dans ce cas de figure, il sera nécessaire d'héberger son propre réseau **LoRaWAN**. La solution la plus souvent retenue est l'utilisation de LoRaServer, qui fournit des composants open source pour la construction de réseaux **LoRaWAN**.

2.4 Récapitulatif

Au terme de cet état de l'art, les choix posés pour la réalisation du travail sont les suivants :

- L'architecture matérielle doit être la plus hétérogène possible.
- Les *containers runtimes* sélectionnés sont : **Containerd** et **RunC**
- La construction des images est réalisée avec **Docker**
- L'orchestrateur utilisé est **Kubernetes** ou **Docker Swarm**
- Les technologies sont implémentées *verticalement*. C'est-à-dire qu'une technologie doit être complètement fonctionnelle avant d'entamer l'implémentation de la suivante.
- L'ordre d'implémentation des technologies suit celui défini par l'état de l'art en sélectionnant la technologie la plus contraignante comme point de départ.

Chapitre 3

Recherche de l'architecture optimale

Si le choix est posé d'intégrer les technologies une à une, dans un souci de clarté, la phase de recherches et développements est présentée horizontalement. C'est à dire, en commençant au plus près du capteur pour arriver au traitement et au stockage des données recueillies.

3.1 Communication avec les réseaux de capteurs

Ce point a pour but de présenter les applications qui servent de *portes d'entrées* aux différents réseaux de capteurs dans l'architecture logicielle distribuée.

3.1.1 Interface 6LoWPAN

Un élément clé, de tout réseau IoT, déployé est le *border router* qui le connecte à Internet. Le *Border Router* est une passerelle entre deux technologies de couche de liaison différentes. Le *Border Router* sépare également les plans de contrôle de deux domaines de routage différents (Deru et al., 2014).

Le CETIC a développé une solution de *Border Router* 6LoWPAN/Routing Protocol for Low power and Lossy Networks (RPL) *opensource* : 6LBR. Ce *Border Router* est utilisé pour l'implémentation de la technologie présentée au point 2.3.1 et appelée *du 802.15.4 au COAP*.

De fait, 6LBR peut fonctionner en tant que routeur autonome sur du matériel intégré ou sur un hôte Linux. Conçu pour la flexibilité, il peut être configuré pour prendre en charge diverses topologies de réseau tout en interconnectant intelligemment les Wireless Sensors Network (WSN) avec le monde IP (cf figure 3.1).

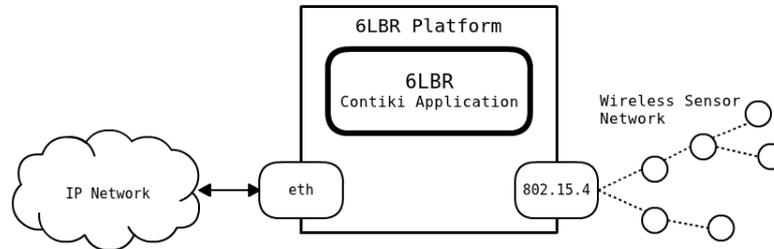


FIGURE 3.1 – Schéma de principe de l'application 6LBR.

3.1.1.1 Configuration des interfaces

La première étape, lorsque l'on souhaite travailler avec 6LBR, est de définir la manière dont les interfaces doivent être connectées entre elles. Pour ce faire, l'application 6LBR propose trois modes de configuration différents : **RAW-ETHERNET**, **BRIDGE** et **ROUTING**.

Le tableau 3.1 reprend ces modes de configuration. Le code source du fichier 6LBR.conf associé à chaque mode de configuration sera placé en annexe à ce travail (cf tableau B.1).

On aurait pu imaginer que le mode *RAW_ETHERNET* serait choisi pour sa facilité de mise en œuvre. Cependant, lors des premiers tests, un problème de communication est survenu : la page du webserver fourni par 6LBR n'était pas joignable. Les *logs* de 6LBR montrent que le *processing* du paquet ne se passe pas de la manière attendue. Comme l'illustre la figure 3.2, l'application reçoit bien une demande, mais ne répond pas¹.

```

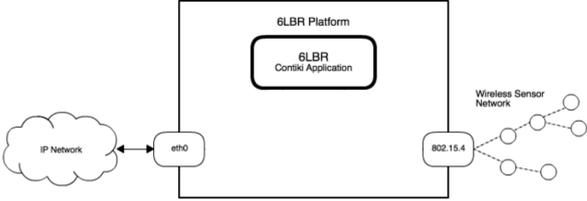
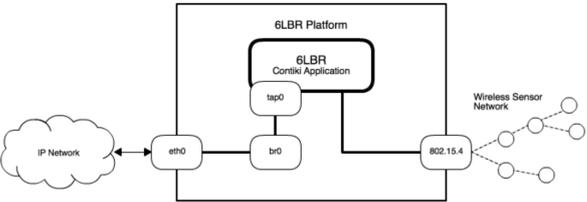
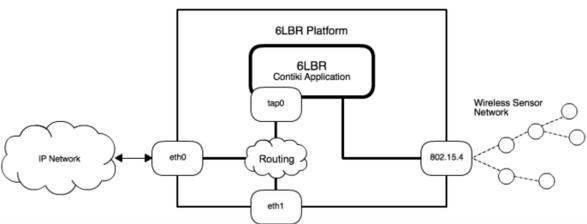
2018-09-19 12:31:52.872910: PACKET: PFE: eth_input: Processing frame
2018-09-19 12:31:52.873034: PACKET: PFE: bridge_output: Sending packet to 02:42:0c:ff:ff:aa:8d:cb
2018-09-19 12:31:52.873394: PACKET: PFE: dest prefix eth : bbbb:
2018-09-19 12:31:52.873513: PACKET: PFE: dest eth : bbbb:7147:b885:16f1:dcd8
2018-09-19 12:31:52.873754: PACKET: PFE: eth_output: 02:42:0c:ff:ff:aa:8d:cb
2018-09-19 12:31:52.874111: PACKET: PFE: eth_output: Sending packet to Ethernet
2018-09-19 12:31:52.874190: PACKET: ETH: write: 118
2018-09-19 12:31:52.874514: PACKET: TAP: write: 118
2018-09-19 12:32:03.514186: PACKET: TAP: read: 94
2018-09-19 12:32:03.514365: PACKET: ETH: read: 94
2018-09-19 12:32:03.514461: PACKET: PFE: eth_input: Processing frame
2018-09-19 12:32:04.578122: PACKET: TAP: read: 94
2018-09-19 12:32:04.578265: PACKET: ETH: read: 94
2018-09-19 12:32:04.578340: PACKET: PFE: eth_input: Processing frame
2018-09-19 12:32:06.658229: PACKET: TAP: read: 94
2018-09-19 12:32:06.658406: PACKET: ETH: read: 94
2018-09-19 12:32:06.658506: PACKET: PFE: eth_input: Processing frame
2018-09-19 12:32:10.738119: PACKET: TAP: read: 94
2018-09-19 12:32:10.738245: PACKET: ETH: read: 94
2018-09-19 12:32:10.738289: PACKET: PFE: eth_input: Processing frame

```

FIGURE 3.2 – Résultats de l'affichage des logs de l'application 6LBR

1. Ce problème a été corrigé depuis mais, aucun test n'a été réalisé pour vérifier si ce mode de configuration des interfaces est utilisable pour une version conteneurisée de 6LBR

TABLE 3.1 – Modes de configuration des interfaces dans 6LBR

schéma de principes	description
	<p>Le mode RAW-ETHERNET est le plus facile à mettre en œuvre, mais est aussi le plus limitant. 6LBR utilise directement l'interface Ethernet spécifiée dans le fichier de configuration pour envoyer et recevoir les paquets.</p>
	<p>Le mode BRIDGE crée une interface virtuelle, qui devra être reliée au moyen d'un bridge à l'interface Ethernet. Il est cependant nécessaire que le module bridge soit présent dans le <i>kernel</i> du système.</p>
	<p>Le mode ROUTING est le plus puissant, mais le plus complexe. Comme pour le mode BRIDGE, une interface virtuelle est créée, mais le routage doit néanmoins être réalisé par l'utilisateur en fonction des besoins.</p>

Un problème qui peut être rencontré avec Le mode *BRIDGE*, est que le *bridge* n'est pas créé lors du lancement du conteneur. Il est possible de le faire en créant un script qui utiliserait les outils fournis par le *package* Linux *bridge-utils*. De outre, des conflits ont été observés avec le *bridge* créé par l'outil d'orchestration.

Le mode *ROUTING* est donc le seul pleinement fonctionnel pour une utilisation de 6LBR au sein d'une solution orchestrée.

3.1.1.2 Mode de routage

Il existe plusieurs modes de routage disponibles, mais nous avons, lors de l'élaboration de ce projet, essentiellement travaillé avec le mode *ROUTER*. Les autres modes ne sont pas relevant pour l'architecture choisie et ne sont donc pas détaillés².

Dans le mode *ROUTER*, 6LBR agit comme un routeur IPv6 à part entière, interconnectant deux sous-réseaux IPv6. Le sous-réseau WSN est géré par le protocole RPL et le sous-réseau Ethernet par IPv6 Neighbor Discovery Protocol (NDP). Pour rappel, NDP est un protocole utilisé par IPv6. Il opère en couche 3 du modèle OSI (Réseau) et est responsable de la découverte des autres hôtes sur le même lien, de la détermination de leur adresse et de l'identification des routeurs présents.

Le mode *ROUTER* fonctionne donc comme une simple passerelle entre Ethernet et 6LoW-PAN/RPL.

3.1.2 Interface Z-Wave

Comme expliqué au point 2.3.2.1, OpenZWave est une bibliothèque permettant de créer une application de gestion d'un WSN Z-Wave. Les solutions d'utilisation d'OpenZWave comme *plugin* d'une solution domotique, à l'instar de HomeAssistant, Jeedom ou Domoticz sont écartées. En effet, la taille de ces applications logicielles est démesurée dans le cadre de la solution envisagée³.

Les porteurs du projet OpenZWave proposent déjà une série d'outils. L'un d'entre eux a été retenu : OpenZWave Control Panel (OZWCP). La taille de la version conteneurisée de ce service n'excède pas les 150Mb, il s'insère donc parfaitement dans le dessein de ce travail.

2. La description détaillée des autres modes est disponible sur la page suivante : <https://github.com/cetic/6lbr/wiki/6LBR-Modes>

3. Les versions conteneurisées de ces applications avoisinent une taille de l'ordre du gigaoctet

OpenZWave Control Panel

Controller Interface

Device name:

USB:

Buttons: Reset, Soft Reset, Close, Initialize

Controller Status

Home ID:

Controller Mode:

Node Count:

SVC Node:

Backup Controller

Buttons: Save

Text: Changes need saving...

Network

Select an operation:

Controller

Select an operation:

Functions

Select an operation:

Devices

Node Id	Basic Type	Generic Type	Product	Name	Location	Value	Last Heard	Status
1 LBR	Static Controller	Static PC Controller	Aeotec ZW090 Z-Stick Gen5				11:43:34 PM	Ready
2 LBR	Slave	Multilevel Power Switch	Airline Mechanical ZDS-100			0	11:43:45 PM	Ready
3 LBR	Routing Slave	Binary Power Switch	GE 12727 In-Wall Smart Switch (Toggle)			off	11:43:45 PM	Ready
4 LBR	Routing Slave	Binary Power Switch	GE 12727 In-Wall Smart Switch (Toggle)			off	11:43:45 PM	Ready
5 LBR	Slave	Multilevel Power Switch	Airline Mechanical ZDS-100			99	11:43:45 PM	Ready
6 LBR	Slave	Multilevel Power Switch	Airline Mechanical ZDS-100			99	11:43:45 PM	Ready
7 LBR	Routing Slave	Secure Barrier AdoOn	Linear Unknown, type=4744, id=3030				11:43:45 PM	Ready

Current Values: Basic:

Buttons: Submit, Refresh

Links: Configuration, Information

FIGURE 3.3 – Interface WEB de gestion fournie par OZWCP

La phase de recherche a démontré que cette application est avant tout une interface web d'administration d'un réseau Z-Wave. Un module d'intégration doit donc être développé pour s'interfacer avec la base de données locale.

3.1.3 Interface LoRa

Le relais entre le concentrateur LoRa et un serveur est réalisé par une application appelée le **Packet Forwarder**. Il s'exécute sur l'hôte d'une passerelle LoRa.

De ce fait, il transmet les paquets radio reçus par le concentrateur à un serveur et inversement, il émet des paquets radio basés sur des requêtes faites par le serveur. Dans notre cas de figure, le protocole de communication défini par Semtech se base sur UDP.

Le **Packet Forwarder** assure également la gestion du concentrateur (bande de fréquences autorisées, règles d'accès au médium radio...). Une version conteneurisée du **Packet Forwarder** a déjà été développée par le CETIC.

3.2 Outils de conteneurisation

La solution envisagée s'appuie sur le concept de micro services conteneurisés. Comme un grand nombre d'applications doivent être déployées, cette partie du travail va revenir sur la phase de recherche qui a permis d'uniformiser les étapes de conteneurisation.

3.2.1 Stratégie de conteneurisation

Lorsque l'on travaille avec **Docker**, les étapes de construction d'une image de conteneur se présentent sous la forme d'un fichier appelé : le **Dockerfile**. Il s'apparente à un document texte contenant une série de commandes exécutées ligne par ligne (Docker, 2019).

En effet, Il est possible de construire une image de conteneur en utilisant qu'un petit nombre de commandes. Voici la description des plus usuelles d'entre elles :

1. FROM : initialise une nouvelle étape de construction et définit l'image de base pour les instructions suivantes.
2. ADD : copie de fichiers depuis l'hôte.
3. RUN : exécute une commande.
4. CMD : définit la commande qui est lancée lorsque le conteneur est déployé.

3.2.1.1 Réduction de la taille des applications

Le mécanisme de Docker pour créer une image peut alourdir considérablement la taille finale de celle-ci. À la différence d'un simple script d'installation, chaque commande ajoute un calque à l'image final. Les opérations étant purement additives, il n'est pas possible de supprimer l'espace utilisé par un calque précédent en effaçant des fichiers dans le calque suivant. Il en résulte comme observation que, plus il y a de calques plus l'image est volumineuse.

En voici l'illustration, deux containers ont été créés, le résultat d'un point de vue applicatif est strictement identique : il s'agit d'une application sur base du système d'exploitation **Alpine**, dans lequel un dossier en prévenance de *github* doit être copié. Le paquet **git** doit être installé préalablement. Un fois la copie terminée, comme le *versioning* de ce dossier n'est pas indispensable, on souhaite supprimer le paquet **git**.

Ci-dessous, aux figures 3.1 et 3.2, voici le code source de deux **Dockerfile** dont la stratégie de construction est différente. Le premier multiplie les commandes **RUN** et le second réalise toutes les actions en une seule commande :

```
1 FROM alpine
2 RUN apk add git
3 RUN git clone https://github.com/Sellto/TFE
4 RUN apk del git
```

Listing 3.1 – Dockerfile avec de multiples commandes RUN

```
1 FROM alpine
2 RUN apk add git && \
3 git clone https://github.com/Sellto/TFE && \
4 apk del git
```

Listing 3.2 – Dockerfile avec une seule commande RUN

Si on regarde la taille finale des deux images construites sur base des *Dockerfiles* présentés aux figures 3.1 et 3.2, on peut observer que la taille finale a triplé (cf. figure 3.4). Cependant, dans le cadre d'un travail visant à délivrer une architecture logicielle distribuée, il est important que les services soient le plus léger possible.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
scripted	latest	b22f96aa5e55	About an hour ago	7.96MB
not_scripted	latest	f47e8faef14d	About an hour ago	22.9MB
alpine	latest	cdf98d1859c1	5 days ago	5.53MB

FIGURE 3.4 – Taille des images réalisées

3.2.1.2 Standardisation des *Dockerfiles*

L'objectif est donc de créer un DockerFile avec le minimum de commandes. Pour standardiser tous les *Dockerfile*, un template a été créé. Ce modèle se construit comme suit :

1. L'image de départ
2. L'ajout d'un dossier avec deux scripts Shell
3. Le lancement d'un script d'installation avec la commande RUN
4. Le lancement d'un script d'initialisation avec la commande CMD

Le script d'installation a pour missions :

- l'installation des dépendances
- l'installation ou compilation de l'application
- la suppression de tout ce qui n'est pas essentiel

On place dans le script d'initialisation toutes les tâches à réaliser lors du lancement du conteneur. On peut, par exemple, attribuer à ce script les tâches suivantes :

- le lancement de l'application
- la configuration de l'application
- les règles de routages
- ...

Voici le *template* utilisé pour la création des images de conteneurs :

```
1 FROM debian
2 ADD needed_files ./needed_files
3 RUN chmod -R +x /needed_files && needed_files/Install.sh
4 CMD needed_files/OnBoot.sh
```

Ce qui rend cette manière de procéder intéressante, réside dans le fait que les scripts peuvent être réutilisés pour l'installation et l'initialisation d'une application, en dehors d'un projet basé sur la conteneurisation.

3.2.2 Image de conteneur multi-architecture

Dans le cadre de ce travail, tous les services prennent la forme de conteneurs. Il convient donc de construire les images de conteneurs pour qu'elles soient compatibles avec toutes les architectures de processeurs présentes dans le cluster. Pour rappel, ce travail ambitionne l'implémentation d'une architecture matérielle distribuée hétérogène.

La première approche envisagée est itérative :

- Copier les fichiers nécessaires à la construction de l'image (cf point 3.2.1.2) sur une machine.
- Builder l'image avec **Docker**.
- Définir un *tag*⁴ spécifique en fonction de l'architecture sur laquelle l'image vient d'être construite.
- Publier l'image sur un dépôt.
- Recommencer sur une autre machine jusqu'à ce que toutes les architectures aient été couvertes.

4. Un *tag* peut être défini comme une étiquette pointant vers une version spécifique d'une image de conteneur

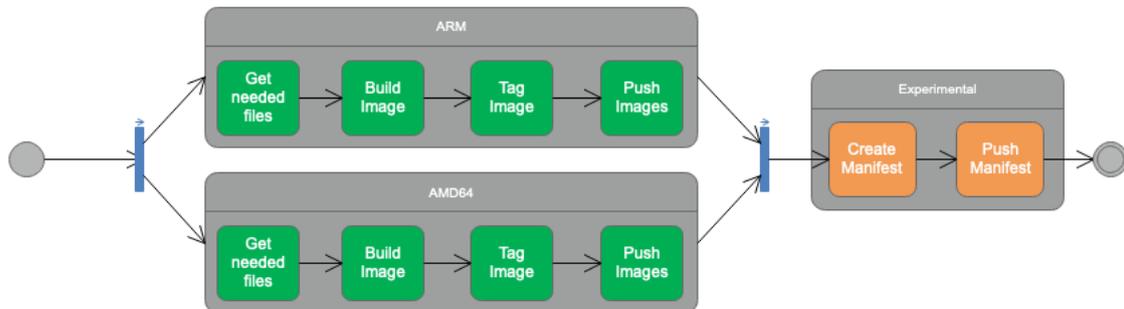


FIGURE 3.5 – Schéma Bloc du playbook Ansible MultiArchBuilder

Cette manière de fonctionner peut cependant vite devenir fastidieuse et chronophage.

3.2.2.1 Ansible

La seconde piste envisagée, est d'utiliser *Ansible*, une plateforme essentiellement utilisée pour la configuration et la gestion d'ordinateurs distants. Cette plateforme permet, entre autres, d'exécuter des tâches sur différentes machines en parallèle via une communication SSH. Il est important de définir deux composants utilisés lorsque l'on travaille avec *Ansible* :

1. l'**inventaire**, est un fichier qui définit les machines sur lesquelles les tâches sont envoyées. C'est dans ce même fichier que sont définis les groupes auxquels appartiennent les hôtes. Des tâches spécifiques peuvent être définies pour ne s'exécuter que sur un groupe d'hôtes bien précis.
2. le **playbook**, orchestre quant à lui l'exécution des tâches à réaliser sur les différents hôtes. Il se présente sous la forme d'une série de fichiers Yet Another Markup Language (YAML).

3.2.2.2 Le *playbook* MultiArchBuilder

Ce volet n'a pas pour but de décrire la manière d'utiliser le *playbook*, ni la manière dont sont rédigés les fichiers YAML⁵ qui le compose, mais plutôt de décrire les tâches qu'il réalise.

Le *playbook* exécute donc, dans un premier temps, les mêmes tâches que celles présentées pour la méthode itérative. Cependant, elles sont réalisées parallèlement sur tous les hôtes. Il s'agit des tâches en vert dans la figure 3.5.

La seconde partie du *playbook* n'est exécutée que sur un hôte particulier. Cette machine doit avoir préalablement activé le mode *experimental* de **Docker**. Ce mode permet de créer, sur l'hôte, une *manifest list*⁶ d'images (en orange sur la figure 3.5). Pour expliquer ce concept, il est plus aisé de s'appuyer sur un exemple :

- La première partie du *playbook* nous a permis de construire une série d'images et de les envoyer dans un dépôt. Ces images sont répertoriées selon un schéma prédéfini : `<user>/<imagename> :<tag>`. Dans notre cas, le nom du *user* et le nom de l'image restent les mêmes. Le *tag* prend, quant à lui, le nom d'une l'architecture de processeur (arm, amd64...). La *manifest list* offre la possibilité de regrouper toutes ces *versions* d'une même image derrière un nom unique : `<user>/<imagename> :latest`.
- La *manifest list* se présente sous la forme d'un fichier au format JavaScript Object Notation (JSON) dans lequel les caractéristiques de plusieurs images sont enregistrées, comme par exemple leur taille ou l'architecture sur laquelle ont été construites ces dernières. Elle agit donc comme un groupement de pointeurs vers des images appropriées à un contexte donné. En d'autres termes, si je me trouve sur une machine avec un processeur de type AMD64 et que je tente de déployer l'image `<user>/<imagename> :latest`, c'est l'image `<user>/<imagename> :amd64` qui sera rapatriée.

5. L'ensemble des fichiers qui compose ce *playbook*, ainsi que sa documentation, est rendu disponible à l'adresse <https://github.com/Sellto/MultiArchBuilder>

6. Ce concept informatique n'est volontairement pas traduit. Il est utilisé comme présenté dans la littérature

3.2.2.3 L'inventaire

L'inventaire créé pour la réalisation de ce playbook définit plusieurs groupes différents :

- **all** : Ce groupe reprend tous les hôtes utilisés par le *playbook*. C'est à cet endroit que les informations de connexion sont placées. Seul ce groupe est susceptible d'accueillir plusieurs hôtes.
- **experimental** : Ce groupe est utilisé pour spécifier l'hôte dont l'option *experimental* de **Docker** est active.
- **architecture** : Les groupes suivants sont prévus pour séparer les hôtes en fonction de leur architecture de processeurs.

```
1 [all]
2 dev1_amd64          ansible_connection=local
3 dev2_arm64          ansible_host=sudouser@machine_ip
4 dev3_arm             ansible_host=sudouser@machine_ip
5
6 [experimental]
7 dev1_amd64
8
9 [amd64]
10 dev1_amd64
11
12 [arm64]
13 dev2_arm64
14
15 [arm]
16 dev3_arm
17
18 [ppc64le]
```

Listing 3.3 – Template de l'inventaire utilisé dans le playbook MultiArchBuilder

3.3 Outils liés à l'orchestration des services

Lors de l'élaboration de l'état de l'art, deux outils d'orchestrations ont été sélectionnés : **Kubernetes** et **Docker Swarm**. En outre, ce travail tend à concevoir une architecture la plus automatisée possible. Une période au cours de la phase de recherche et développement a donc été consacrée à la réalisation d'une application capable de monitorer les périphériques servant d'interfaces de communications avec les WSN 6LoWPAN, LoRa et Z-Wave. Dans les deux points suivants, le premier revient sur les critères qui ont permis de départager les deux orchestrateurs et le second détaille l'élaboration de l'application appelée l' **Auto-Labeler** qui a pour objectif d'automatiser le déploiement de services sur certaines machines en fonction de leurs spécificités.

3.3.1 Comparaison de Docker Swarm et Kubernetes

Kubernetes et **Docker Swarm** ont été départagés sur base de deux critères : La gestion d'un réseau dualstack IPv4/IPv6 et la communauté entourant chacun de ces outils.

3.3.1.1 La gestion d'un réseau dualstack IPv4/IPv6

Un orchestrateur proposant cette fonctionnalité serait certainement choisis puisque, l'une des technologies que l'on désire mettre en place dans le *cluster* repose sur l'adressage IPv6 des capteurs. Il s'agit de 6LBR (cf. point 2.3.1).

La documentation de **Docker Swarm** annonce une compatibilité avec le monde IPv6. Cependant, une série de tests a permis de mettre en avant certains dysfonctionnements.

Docker Swarm déploie les conteneurs sur une forme de réseau particulier appelé, *ingress*⁷. Ce type de réseau ne supporte, par défaut, qu'un adressage *IPv4*. Pour créer un réseau *dual stack* IPv4/IPv6 sous **Docker Swarm**, il est nécessaire de supprimer le réseau existant pour en recréer un autre. Pour exemple, la commande de la figure 3.6 peut être utilisée pour créer un réseau dual stack de type *ingress*.

Selon la documentation fournie par **Docker**, l'ordre dans lequel les arguments de création des sous-réseaux IPv4 et IPv6 sont placés, dans la commande, n'a pas de réelle importance. Le résultat de la commande présentée à la figure 3.7 est normalement équivalent à celui illustré à la figure 3.6.

7. Ce réseau est créé avec le driver *overlay*

```
1 Docker network create --ingress --driver=overlay --ipv6 --subnet  
=2001::/64 --subnet=172.30.0.0/16 myDualStackNet
```

FIGURE 3.6 – Commande utilisée pour la création d'un réseau *dual stack* de type *ingress*

```
1 Docker network create --ingress --driver=overlay --subnet  
=172.30.0.0/16 --ipv6 --subnet=2001::/64 myDualStackNet
```

FIGURE 3.7 – Commande alternative utilisée pour la création d'un réseau *dual stack* de type *ingress*

Dans les deux cas, l'architecture n'est pas fonctionnelle. Les causes de ce problème ont été mises à jour grâce à l'outil *inspect* de **Docker**, qui permet de récupérer des informations sur différents composants (conteneurs, réseaux ...). Les résultats des *inspections* réalisées sur les différents réseaux obtenus, en fonction des différentes commandes possibles, sont présentés en annexe de ce travail (Annexe A.1).

Les observations que l'on peut en faire à ce stade sont les suivantes :

1. Lorsque le sous réseau IPv4 est placé en premier dans la commande (cf figure 3.7), **Docker** désactive l'IPv6 et ne donne pas d'adresse IPv6 au container responsable de la gestion de la charge de travail, dans le *cluster*. Ce conteneur s'appelle le **load-balancer**
2. Lorsque le sous-réseau IPv6 est placé en premier lieu, le **load-balancer** n'est pas déployé. De ce fait, aucun nouveau conteneur ne peut être lancé. Cette observation est faite également pour un réseau de type *ingress*, dont l'adressage n'est réalisé qu'avec des adresses IPv6.

Kubernetes n'annonce pas le support de l'adressage IPv6 dans ses dernières versions. Ce point ne permet donc pas de se fixer sur l'un ou l'autre outil.

3.3.1.2 La communauté

Il est important d' éviter de choisir un orchestrateur qui déciderait, à l'instar de CoreOS, de ne plus mettre à jour son produit. Généralement, un outil soutenu par une large communauté ne connaît pas ce type d'issue. On peut observer sur les graphiques de la figure 3.8, que **Kubernetes** est plus largement soutenu que ne l'est **Docker Swarm**, aussi bien par une communauté de développeurs, que par une communauté de scientifiques.

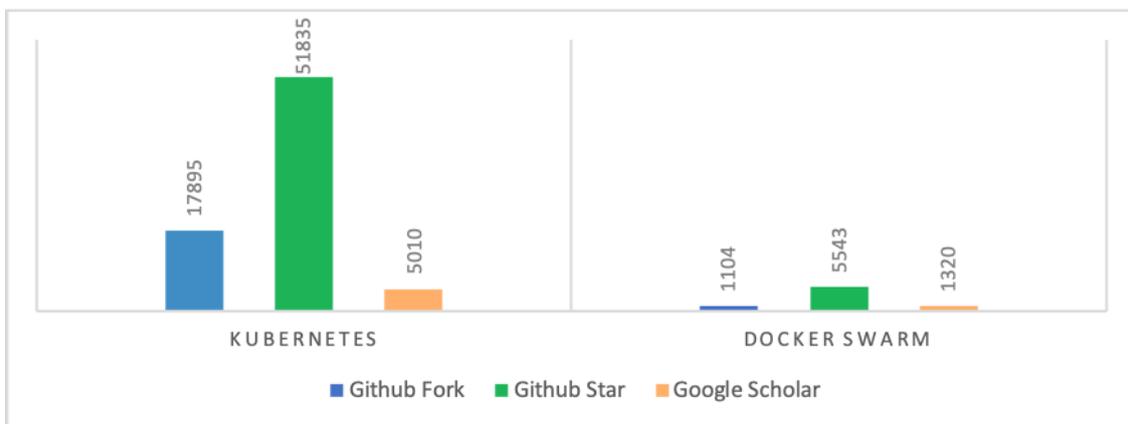


FIGURE 3.8 – Comparaison des communautés de Docker Swarm et Kubernetes au 28/04/2019

3.3.1.3 Discussions

Docker Swarm présente des *bugs* dans une version considérée comme stable. En outre, au vue de la communauté qui l'entoure, **Kubernetes** est préféré à **Docker Swarm**.

3.3.2 Déploiement automatique de service

L'ambition est de réaliser une application capable de détecter l'ajout à un nœud d'un périphérique correspondant à une interface. Si tel est le cas, les services nécessaires à son intégration dans l'architecture vont être déployés. Dans le même esprit, si le périphérique est retiré, les services associés vont être automatiquement supprimés.

Plusieurs pistes ont été envisagées :

1. la réalisation d'un environnement client-server dans lequel, les clients sont les nœuds du cluster et le serveur, le *master*. Lorsqu'une action sur un périphérique est repérée par un des clients, il envoie une requête au serveur pour lui notifier cet événement. Le serveur du cluster peut ensuite déployer les services adéquats. Cette technique implique de donner la possibilité de placer un container sur le master, autre que ceux destinés à la gestion du *cluster*.
2. l'exécution sur chaque machine d'une application indépendante. Si un événement est détecté, l'application vérifie dans une base de données externe si le périphérique correspond à une interface prise en charge par l'architecture. Sur cette base, un label est donné au nœud. **Kubernetes** met en place un mécanisme qui monitorise les labels de chaque nœud. En outre, il est possible de configurer le déploiement d'une image sur des nœuds ayant un label spécifique. Cependant, cette technique sous-entend que l'application ait des privilèges élevés.

Cette seconde solution est sélectionnée, car elle permet de ne pas surcharger le *master* et évite de le corrompre en cas de *bugs*.

3.3.2.1 Sécurité

La méthode choisie implique d'élever les privilèges de l'application déployée sur chaque nœud. **Kubernetes** fournit deux ressources qui définissent les opérations autorisées sur d'autres ressources, par un ensemble d'utilisateurs : **Role** et **ClusterRole**. La ressource de type **ClusterRole** permet de gérer les autorisations d'action sur des ressources au niveau du *cluster*, indépendamment du **Namespaces**⁸, dans lequel cette ressource se trouve. À l'inverse, la ressource de type **Role** permet de gérer les ressources en fonction du **Namespaces** dans lequel elle se trouve.

8. Kubernetes peut prendre en charge plusieurs *clusters* virtuels sauvegardés par le même *cluster* physique. Ces *clusters* virtuels sont appelés **Namespaces**.

Il est important de garder à l'esprit, lors de la structuration des autorisations, qu'il n'existe pas de règles de refus, mais uniquement des règles d'ajouts d'accès. (Kubernetes, 2019)

Un des problèmes est qu'un container déployé à l'intérieur d'un cluster ne connaît pas le nom du nœud sur lequel il se trouve. Il est cependant possible de récupérer le nom du container. Une première élévation des privilèges réalisée, donne la possibilité au service de lister les *Pods* présents dans son *namespace* et de connaître le nœud sur lequel ils sont déployés. Il n'est cependant pas capable d'agir sur les *Pods*.

La deuxième élévation de privilèges, pour sa part, donne la possibilité de modifier les nœuds, mais pas de les lister.

La combinaison de ces deux privilèges permet de réaliser les deux opérations souhaitées : la récupération du nom du nœud et sa labellisation.

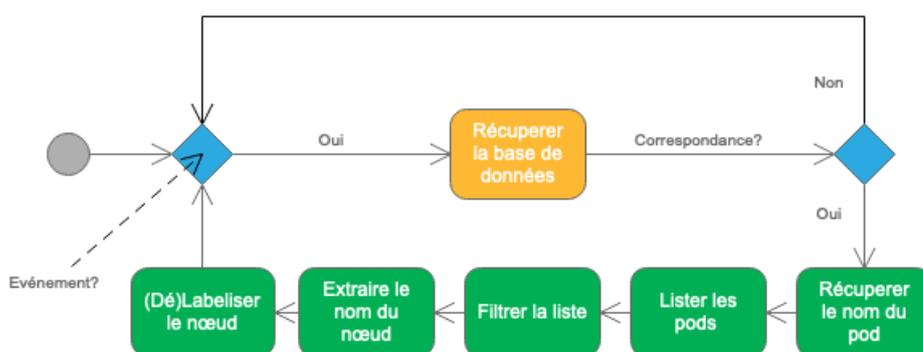


FIGURE 3.9 – Schéma Bloc du fonctionnement de l'application AutoLabeler

Ce volet a pour dessein l'explication des mécanismes de l'application et non pas un développement autour du code source de l'application⁹. Le fonctionnement de l'application est illustré à la figure 3.9

Correspondance avec la base de données

La gestion événementielle de l'application repose sur une librairie : *libudev*. Cette librairie permet de monitorer les périphériques d'une machine. Lorsque l'on est en présence d'un périphérique de type USB, elle donne la possibilité de récupérer des informations concernant le *device* connecté.

9. Le code source est rendu disponible à l'adresse https://github.com/Sellto/K8S_AutoLabeler

Une base de données au format YAML a été créée dont les entrées se présentent de la manière suivante :

```
1 10c4:ea60:
2   - Manufacturer : "Silicon_Labs"
3     Product : "Zolertia_Firefly_platform"
4     Label:
5       - "IEEE_802_15_4"
```

Les éléments de la base de données sont référencés au moyen d'un identifiant formé par la composition du *vender ID* et du *Product ID* d'un périphérique. Le champ label permet de caractériser le produit, dans ce cas-ci, il s'agit d'une carte radio compatible avec un réseau 6LoWPAN.

Récupération du nom du nœud

L'application s'appuie sur le résultat de la commande suivante :

```
1 kubectl get pods -o wide | grep $HOSTNAME
```

La première partie de la commande¹⁰ permet de lister les *Pods*. La structure d'autorisations mise en place au point 3.3.2.1 n'admet l'affichage que des pods présents dans le même *namespace*. La seconde partie réalise un filtrage des résultats sur base de la variable `HOSTNAME`. En effet, comme cette variable d'environnement renvoie le nom du container, la sortie standard de cette commande est une simple ligne de la forme suivante :

```
autolabeler-ds-f8q57 1/1 Running 1 6d7h 10.42.0.162 mbpro <none> <none>
```

L'extraction du nom du nœud sur base de ce résultat n'est qu'une manipulation de chaîne de caractères.

Labelisation du nœud

Après avoir vérifié si un périphérique appartient à la base de données, d'extraire le label associé et de récupérer le nom du nœuds sur lequel il se trouve. L'application peut labeliser le nœud en utilisant la commande :

```
1 kubectl label node <nodename> <labelvalue>=yes
```

10. avant le symbole |

3.4 Intégration d'une application IPv6 dans un conteneur IPv4

L'utilisation de 6LBR, dans une architecture logicielle distribuée orchestrée avec **Kubernetes**, implique de tenir compte de deux faits importants :

1. Comme expliqué précédemment, 6LBR n'existe que dans un monde IPv6. En outre, à l'heure actuelle un grand nombre de solutions d'orchestration et d'applications reste cantonné dans le monde IPv4. L'utilisation d'un Network Address Translation (NAT), pour convertir les adresses IPv6 en IPv4, est donc indispensable. Un Network Address Translation 6to4 (NAT64) est déjà implémenté dans l'application 6LBR. Cependant, le mécanisme utilisé pour la traduction n'est pas compatible avec les principes de conteneurisation. L'application attribue un port d'une adresse IPv4 fixée préalablement à un *endpoint* du WSN, ce qui implique d'exposer manuellement les 65 535 ports du conteneur. Dans l'attente d'une avancée dans ce domaine, une solution temporaire peut être envisagée : l'utilisation de l'application **Tayga**, un NAT sans état.
2. Un autre enjeu, de l'utilisation d'un orchestrateur dans une architecture logicielle distribuée, est de définir la manière dont les services communiquent entre eux. Généralement, l'orchestrateur attribue une adresse IP à chaque *pod*. Dans l'éventualité où deux *pods* désireraient *communiquer*, ils peuvent le faire via cette adresse. Mais, imaginons que le nœud sur lequel se trouve un des deux services vient à s'éteindre brutalement, les processus, mis en place par l'orchestrateur, permettraient de redéployer l'application sur un autre nœud, mais en lui attribuant une adresse IP différente.

L'utilisation d'un Domain Name System (DNS) permet de s'affranchir de ces problèmes puisqu'il permet traduire des noms de domaine en adresse IP. Ce service est facilement déployable avec Kubernetes mais, le résultat de la traduction prend la forme d'une adresse IPv4. L'utilisation de *TOTD*, un DNS64, permet de rediriger les requêtes DNS faites depuis une adresse IPv6 vers un serveur DNS IPv4.

Au vue de ces constatations, il apparaît que 6LBR peut servir de support à l'élaboration d'un conteneur, possédant un réseau interne IPv6 transparent pour le réseau IPv4 de l'orchestrateur et exploitant le serveur DNS utilisé par les nœuds de l'architecture matérielle distribuée.

3.4.1 Tayga

Le NAT sans état effectue simplement une substitution des adresses IP à l'aide d'une table de mappage fournie par l'administrateur du réseau. Par exemple, une organisation dont l'allocation d'adresse globale est 198.162.1.0/24 mais, dont les hôtes utilisent les adresses dans le réseau 10.1.1.0/24 pourraient utiliser un NAT sans état pour réécrire 192.168.1.2 en 10.1.1.2, 192.168.1.35 en 10.1.1.35, etc., dans la direction sortante, et inversement dans la direction entrante. **Tayga** fonctionne exactement de cette manière.

Quand on traduit des paquets d'IPv4 à IPv6 (ou IPv6 à IPv4), l'adresse de la source et de la destination sont substituées dans les en-têtes des paquets en utilisant un mappage un pour un. Ce qui signifie que, pour échanger des paquets sur NAT64, chaque hôte IPv4 doit être représenté par une adresse IPv6 unique et chaque hôte IPv6 doit être représenté par une adresse IPv4 unique.

3.4.1.1 Mappage d'IPv6 en IPv4

Comme expliqué dans le point précédent, en tant que NAT sans état, Tayga nécessite l'attribution d'une adresse IPv4 unique pour chaque hôte IPv6. Cette affectation peut être faite statiquement par l'administrateur réseau ou dynamiquement par Tayga, à partir d'un pool d'adresses IPv4.

Le mappage dynamique permet à Tayga d'attribuer des adresses IPv4 aux hôtes IPv6. Par défaut, il est garanti que ces assignations restent utilisables endéans les deux heures après le dernier paquet vu. Elles sont néanmoins conservées jusqu'à deux semaines, tant que le pool d'adresses n'est pas vide¹¹ (Lutchansky, 2011).

3.4.1.2 Mappage d'IPv4 en IPv6

Tayga mappe les adresses IPv4 vers le réseau IPv6 conformément à la norme *RFC 6052*. Cette dernière stipule qu'une adresse IPv4 32 bits doit être ajoutée à un préfixe IPv6. Celui-ci est nommé le préfixe NAT64. Cette adresse IPv6 résultante peut être utilisée pour contacter l'hôte IPv4 via le NAT64.

11. Les assignations sont écrites sur le disque, elles sont donc persistantes lors du redémarrage du démon. Ce qui permet aux sessions TCP et aux sessions UDP existantes de ne pas être interrompues.

Le préfixe NAT64 doit être attribué dans le range d'allocation d'adresse IPv6 globale d'un site. Par exemple, si le réseau 2017:5:8::/48 est alloué à un site, le préfixe 2017:5:8:ffff::/96 peut être réservé pour le NAT64¹². L'hôte IPv4 198.51.100.10 est alors accessible via le NAT64 à l'aide de l'adresse 2017:5:8:ffff::c633:640a. Dans la pratique, il est tout à fait possible d'utiliser la syntaxe 2017:5:8:ffff::198.51.100.10 à la place.

3.4.2 ToTD

Il existe différents types d'enregistrement DNS signalant au serveur la manière de les aborder. Lorsque l'on travaille avec un DNS64, seulement deux sortes des requêtes sont utilisées : A et AAAA. Le type A renvoie une adresse IPv4, pour un nom de domaine donné, le type AAAA quant à lui renvoie une adresse IPv6.

ToTD peut-être vue comme un relais. Il intercepte les requêtes AAAA pour générer une requête A. Il ajoute également à la réponse un préfixe IPv6. Pour que l'ensemble fonctionne de façon cohérente, le préfixe d'un DNS64 doit être le même que celui utilisé par le NAT64, pour le mappage IPv4 à IPv6.

3.4.3 Création de l'application 4LBR

La figure 3.10 a été réalisée dans le but de rendre plus visuel les mécanismes et les services nécessaires au fonctionnement de 6LBR, dans l'architecture logicielle distribuée ambitionnée par ce travail. Pour le monde extérieur au conteneur, 6LBR fonctionne désormais dans un univers purement IPv4, c'est pourquoi il a été décidé de renommer cette version *patchée* du *Border Router* sous le patronyme de **4LBR**.

Comme décrit au point 3.2.1.2, tous les *Dockerfiles* sont réalisés de la même manière. A ce stade, il convient donc de définir l'image de départ et de rédiger les scripts d'installation et d'initialisation de l'image de conteneur.

12. Il y a plusieurs options pour la longueur du préfixe NAT64, un /96 est cependant recommandé.

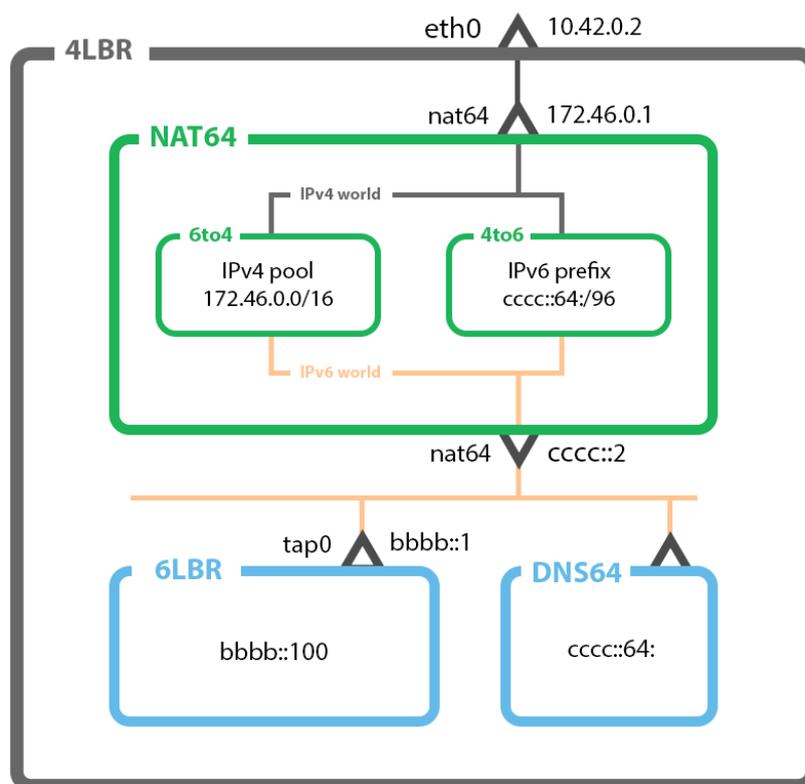


FIGURE 3.10 – Schéma de principe de l'application conteneurisée 4LBR

3.4.3.1 Image de départ

Le choix de l'image de départ, lors de l'étape de construction, peut être un facteur déterminant dans la taille finale de l'application. Souvent le choix se porte sur **Alpine**, une distribution Linux ultra légère (l'image Docker ne fait que 4Mo). Cependant, les premiers essais réalisés avec **Alpine** ne sont pas concluants.

L'application 6LBR utilise des fonctions de contextualisation obsolètes issues de la librairie *GNU c*, **Alpine** n'utilise pas cette dernière et lui préfère *musl*, dans laquelle toutes ces fonctions utilisant la notion de contexte ne sont pas présentes. Les essais sont donc réalisés, sur base de l'image **Debian Jessie**(58Mo) en attendant la mise à jour de 6LBR.

3.4.3.2 Scripts d'installation

Le script d'installation suit scrupuleusement les instructions d'installation disponibles dans la documentation de 6LBR¹³, Tayga¹⁴ et TOTD¹⁵

3.4.3.3 Scripts d'initialisation

Le script d'initialisation, présenté à la figure 3.4, s'exécute au démarrage du conteneur et réalise plusieurs tâches dans un ordre bien déterminé :

1. Configuration de l'application 6LBR.
2. Démarrage de service 6LBR.¹⁶
3. Création du fichier de configuration de Tayga. Ce fichier n'est pas construit durant la phase d'installation, des informations telles que l'adresse IP du conteneur doivent y apparaître. Cette donnée ne peut être récupérée qu'une fois le conteneur lancé.
4. Réalisation du routage et configuration des interfaces.
5. Démarrage du NAT64 (Tayga)
6. Démarrage du DNS64 (Totd)
7. Maintient du conteneur *en vie* en utilisant une commande d'avant plan qui ne se termine pas, ici un *tail -f* des logs de 6LBR.

13. <https://github.com/cetic/6lbr/wiki/Other-Linux-Software-Configuration>

14. <http://www.litech.org/tayga/>

15. <https://github.com/fwdillema/totd>

16. Une pause de 10 secondes est rajoutée pour être sûre que l'application est bien déployée avant de continuer.

```
1 #6LBR Conf
2 nvmtool --update --dft-router bbbb::1 /etc/6lbr/nvm.dat
3 nvmtool --update --mdns-enable 0 /etc/6lbr/nvm.dat
4 nvmtool --update --dns-server bbbb::1 /etc/6lbr/nvm.dat
5 echo "host=$LWM2MHOST" >> /etc/6lbr/nvm.conf
6 echo "port=$LWM2MPORT" >> /etc/6lbr/nvm.conf
7 sleep 2
8 #Start 6LBR
9 service 6lbr start
10 sleep 10
11 #Tayga config file
12 cat >/usr/local/etc/tayga.conf <<EOF
13 tun-device nat64
14 ipv4-addr $(netinfo r)
15 prefix cccc::64:0:0/96
16 dynamic-pool $(netinfo p)
17 data-dir /var/db/tayga
18 EOF
19 #Routing
20 echo 0 > /proc/sys/net/ipv6/conf/tap0/disable_ipv6
21 echo 1 > /proc/sys/net/ipv6/conf/tap0/forwarding
22 echo 1 > /proc/sys/net/ipv6/conf/eth0/forwarding
23 echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
24 ip addr add bbbb::1/64 dev tap0
25 tayga --mktun
26 ip link set nat64 up
27 echo 0 > /proc/sys/net/ipv6/conf/all/disable_ipv6
28 echo 0 > /proc/sys/net/ipv6/conf/nat64/disable_ipv6
29 echo 1 > /proc/sys/net/ipv6/conf/nat64/forwarding
30 ip addr add $(netinfo r) dev nat64
31 ip addr add cccc::2/64 dev nat64
32 ip route add cccc::64:0:0/96 dev nat64
33 ip route add $(netinfo p) dev nat64
34 #Start Tayga
35 tayga
36 #Start Totd
37 totd -c /etc/totd/totd.conf
38 sleep 5
39 #Let container open
40 tail -f /var/log/6lbr.log
```

Listing 3.4 – Script d'initialisation de l'application 4LBR

3.5 Outils de traitements des données

Les points décrits précédemment ont permis d'établir le lien entre des réseaux de capteurs et l'architecture logicielle distribuée. La phase étudiée dans cette section a vocation à définir les mécanismes de récolte de données, pour chaque technologie utilisée.

3.5.1 Recherche d'un serveur LWM2M

Comme présenté au point 2.3.1, la première technologie de récolte de données proposée utilise CoAP comme couche applicative. CoAP étant un protocole de communication, chaque utilisateur peut en soumettre une implémentation différente. Il existe un standard de communication, que l'on peut utiliser lorsque ce protocole de communication est utilisé : Lightweight Machine to Machine (LwM2M). 6LBR exploite ce standard au travers d'un plugin.

Une phase de recherche a donc été effectuée dans l'objectif, de trouver un serveur LwM2M pouvant s'insérer dans l'architecture logicielle distribuée élaborée dans ce travail. Les différents serveurs ont été évalués suivant trois critères :

- L'adressage des *end-devices* : les mises à jour régulières de Kubernetes laissent entrevoir la possibilité d'arriver à un orchestrateur dualstack IPv4/IPv6¹⁷. Sur le moyen terme, l'espoir est donc mis sur le fait de pouvoir ôter le patch placé sur 6LBR (cf. 3.4). Le critère sera donc, la prise en charge de *end-devices* avec une adresse IPv6 ou IPv4. Il ne doit donc pas être un antagoniste à l'amélioration espérée de Kubernetes.
- Les mécanismes de scalabilité du service : L'un des piliers de ce travail est la continuité des services. Si plusieurs instances peuvent être présentes lors de l'initialisation, le service est assuré même si l'une des instances *crash*. Un autre volet est l'auto scalabilité, si une forte demande est faite au serveur, il doit être capable de se répliquer pour mieux partager cette charge de travail accrue.
- La facilité d'intégration : L'accent est également mis sur le nombre de services additionnels à ajouter pour que le service s'insère dans l'architecture logicielle distribuée.

17. <https://github.com/kubernetes/enhancements/issues/508>

3.5.1.1 Leshan Server Demo

L'un des protagonistes principaux est Leshan Server qui n'est autre qu'une librairie Java destinée aux utilisateurs désireux de développer leur propre serveur ou client LwM2M. Pour pouvoir démontrer le potentiel de cette bibliothèque, Leshan met à disposition un serveur démo composé d'une interface Web et d'une API REST.

Une forte communauté déploie ce serveur démo comme solution de production.

En effet, Leshan Server Demo gère très bien les *end-devices*, qu'ils aient une adresse IPv4 et/ou une adresse IPv6. Il ne propose cependant aucun mécanisme de scalabilité. Son intégration devra s'accompagner d'un service supplémentaire faisant le pont entre l' API, qu'il propose, et la base de données.

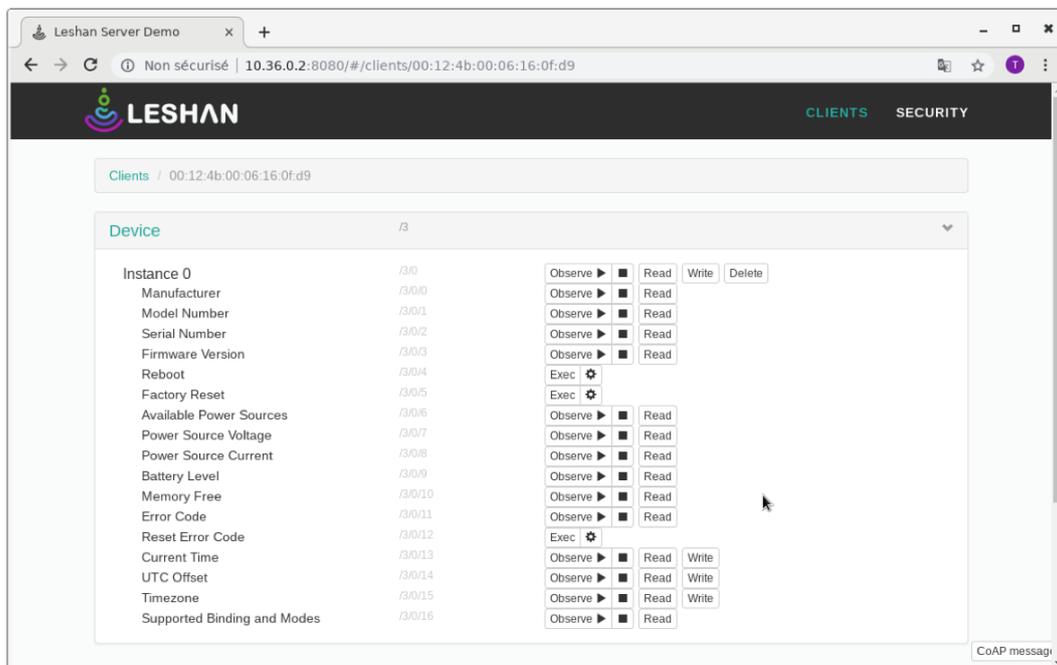


FIGURE 3.11 – Interface WEB de gestion de Leshan Server Demo

3.5.1.2 EMQx

EMQX est un broker MQTT. Son principal atout réside dans le plugin LwM2M qu'il propose. EMQx fournit deux topics MQTT par *device* :

1. lwm2m/<epn>/dn
2. lwm2m/<epn>/up/resp

Où <epn> est le nom du endpoint.

Un premier point négatif, est que EMQx ne permet pas de gérer des devices avec des adresses IPv6. Si cette philosophie n'évolue pas, le patch placé sur 6LBR devient incontournable.

De plus, le mécanisme de communication avec les end-devices rend l'intégration complexe. Si on désire envoyer un message à un device du WSN, un publish est soumis au premier topic. La réponse quant à elle arrivera sur le second topic. En outre, il n'existe pas de topic permettant de monitorer l'arrivée d'un nouveau capteur. L'intégration de ce senseur, dans le flux de données, ne peut donc pas être automatisée.

Cependant, EMQx avance une bonne gestion de la scalabilité, ce qui en fait un service hautement disponible et dynamique.

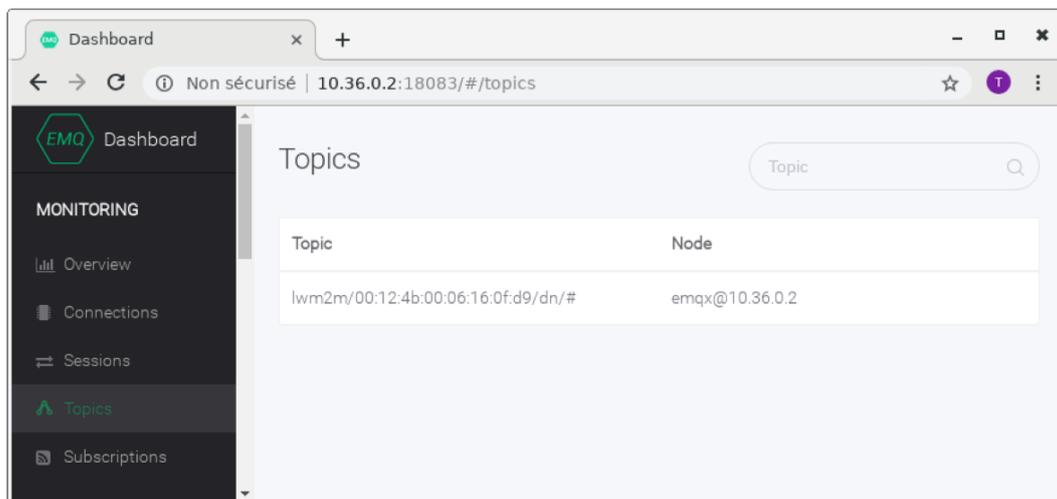


FIGURE 3.12 – Interface WEB de gestion de EMQx

3.5.1.3 Creation d'un serveur

Une étude de faisabilité de l'implémentation d'un serveur LwM2M a été réalisée. Elle se présente sous la forme d'un serveur minimaliste prenant en charge les fonctions *Register*, *Discover* et *Update* du standard LwM2M¹⁸.

Comme le démontre la figure 3.13, le serveur se présente sous la forme d'une file de tâches. La communication est de type half-duplex. Le serveur ne peut pas recevoir de messages lorsqu'il transmet des informations. Les collisions seront évitées par un mécanisme d'accusé de réception. Lorsque la file de tâches est vide, le serveur se place en mode "écoute" en ajoutant une tâche de type *Listen* dans la file. Seule l'implémentation des tâches **Listen** et **Register** est développée. Les autres tâches suivent un mécanisme plus ou moins similaire.

Listen Task

Un système de *timeout* a été mis en place pour permettre aux tâches qui seraient arrivées dans la file de s'exécuter (Read, Write ...) et ainsi éviter de bloquer la file sur une tâche d'écoute. Si un message arrive endéans le temps limite fixé, la tâche s'arrête et le message est traité. Dans le cadre de ce *proof of concept*, le serveur s'attend à recevoir deux types de messages : une demande d'enregistrement d'un nouveau device ou une mise à jour récurrente d'un device déjà enregistré¹⁹.

Pour chaque type de messages reçus, la tâche adéquate est placée dans la file d'attente.

Register Task

Cette tâche est purement linéaire et comprend trois étapes :

- Envoi d'un accusé de réception au device.
- Stockage des informations du device dans une base de données.
- Ajout de la tâche **Discover** dans la file d'attente.

Discussion

Les premiers tests de ce serveur sont très concluants. Son intégration dans l'architecture logicielle distribuée pourrait être relativement aisée, puisqu'il a été conçu dans ce sens. La gestion des adressages IPv4 ou IPv6 est pleinement fonctionnelle. Cependant, aucune gestion de la scalabilité n'a été, pour le moment, imaginée.

18. Le code complet de ce serveur est mis à disposition à l'adresse <https://github.com/Sellto/go-lwm2m>

19. La réception d'une réponse à une requête faite par le serveur est gérée par la tâche dans laquelle cette requête a été lancée.

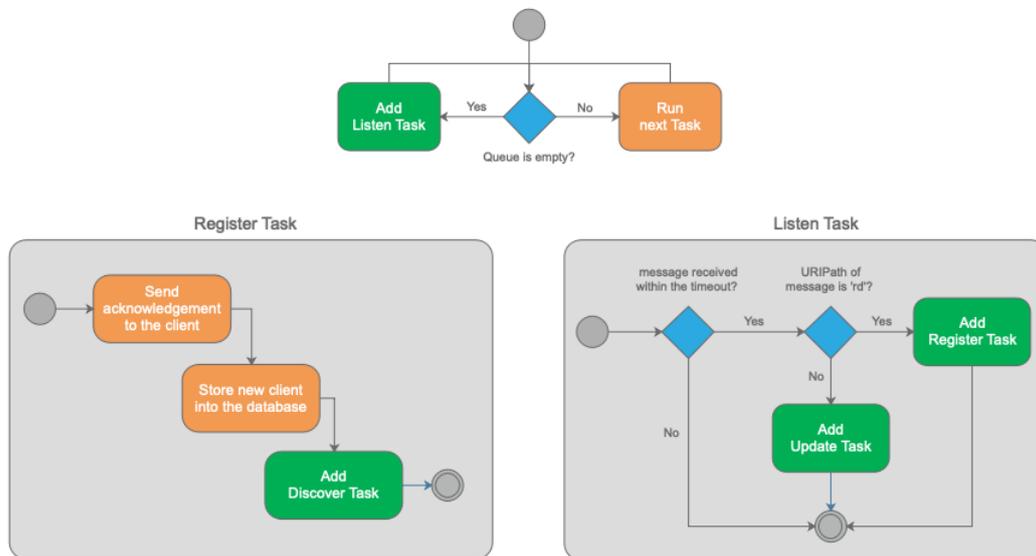


FIGURE 3.13 – Schéma bloc de l'implémentation de go-lwm2m

3.5.1.4 Leshan Cluster Server

Leshan Server ne peut pas être facilement déployé dans un cluster. Un travail expérimental a été réalisé par Eclipse²⁰, pour essayer de permettre le déploiement d'un serveur LwM2M sur base de la librairie **Leshan**. Selon eux, ce projet, baptisé Leshan Cluster Server, ne devrait pas encore être utilisé en production. Ce travail s'inscrit dans une dynamique de recherche et développement, il est donc normal d'y prêter attention.

Basé sur la même librairie que Leshan Server Demo, sa gestion de l'adressage IPv4 et IPv6 est donc complètement fonctionnelle. Son originalité se trouve dans sa gestion de la scalabilité, qui repose sur une base de données REmote DIctionary Server (Redis)²¹.

La réception d'informations en provenance des **end-points** prend la forme de souscriptions aux canaux d'écoute fournis par la base de données :

- **LESHAN_REG_NEW** : pour les demandes d'enregistrements de nouveaux devices.
- **LESHAN_REG_UP** : pour les mises à jour récurrentes de devices déjà enregistrés.
- **LESHAN_REG_DEL** : pour les demandes de dés-enregistrements de devices.

20. la fondation Eclipse vise à développer un environnement de production de logiciels libre. Le langage de programmation principalement utilisé pour la conception de leur solution est Java

21. Redis est un système de gestion de base de données clef-valeur scalable

La transmission de messages aux *devices* a été implémentée autour de requêtes faites directement à la base de données Redis, sous la forme de publication sur un canal spécifique : **LESHAN_REQ**. La réponse est réceptionnée sur le canal **LESHAN_RESP**.

L'intégration, dans une architecture logicielle distribuée de récolte de données, n'est donc pas immédiate. Un service servant de passerelle entre les canaux de la base de données Redis et une base de données globale doit être développé.

3.5.1.5 Récapitulatifs

TABLE 3.2 – Tableau comparatif récapitulatif des différents serveurs LwM2M envisagés

•	LSD	EMQx	go-lwm2m	LCS
Adressage	+	-	+	+
Scalabilité	--	++	n/a	+
Intégration	-	--	++	-

légende :

- ++ fonctionnel et très efficace
- + fonctionnel
- à implémenter
- difficile à implémenter
- n/a non applicable

Sur base de ces constatations, le serveur LwM2M sélectionné est Leshan Cluster Server.

3.5.2 Récupération des données du réseau Z-Wave

La documentation de OpenZWave Control Panel (OZWCP) ne précise pas si une Application Programming Interface (API) est disponible. Cependant, une rapide investigation du code source de l'application a démontré que les mécanismes de communications s'appuient sur des requêtes de type HTTP.

Le processus de récolte de données de la page web de OZWCP réside dans une requête de type *GET* sur la page `http://<ozwcp.ip.or.domain.name>:8090/poll.xml` à intervalle régulier. Il est donc tout à fait possible d'utiliser le même mécanisme non pas pour alimenter la page d'administration du réseau Z-Wave, mais bien en vue de stocker les données. L'inconvénient est une dépendance sur le format, non documenté, du fichier *poll.xml*.

3.5.3 Récupération des données du réseau LoRaWAN

L'objectif de ce travail étant d'opérer en complète indépendance, le type de réseau LoRa envisagé est un réseau privé. Son hébergement sera pris en charge par **LoRa Server** qui fournit tous les composants nécessaires à la construction de ce type de réseau.

Il se compose de trois éléments distincts :

- *LoRa-gateway-bridge* : qui reçoit les paquets UDP transmis par le **Packet Forwarder**. Ces paquets sont alors stockés dans une file de messages MQTT ²².
- *LoRa Server* : qui répond aux demandes d'activation et déchiffre les données reçues en prévenance des nœuds pour lesquelles une clé de session a été configurée ²³
- *LoRa app Server* : interface web permettant la création et la configuration de nœuds.

Des images de conteneurs officielles de ces services sont présentes sur le dépôt officiel de **Docker**.

22. Un *broker* MQTT doit donc être implanté dans l'architecture logicielle distribuée

23. Un système de gestion de base de données relationnelle est indispensable pour garantir la persistance des données notamment pour les clés de session utilisées. Dans ce cas de figure, La base de données imposée est PostgreSQL

3.6 Choix du système de gestion de base de données locale

Une information importante, lorsqu'on réalise une récolte de données en provenance de capteurs, est l'heure de la prise des mesures. Une forme de base de données spécifique est idéale dans ce cas de figure : les bases de données chronologiques appelées plus souvent *Time series database* (TSDB). Ces bases de données sont optimisées pour les données horodatées ou en série chronologique. Citons comme exemple, la sauvegarde au long court d'une mesure de température réalisée par un capteur à intervalle régulier.

Un rapide tour d'horizon des TSBD conduit à se tourner vers InfluxDB, et ce pour trois raisons :

1. InfluxDB est certes une TSBD, mais s'inscrit dans l'écosystème TICK (Telegraf, InfluxDB, Chronograf et Kapacitor) proposé par InfluxData dont la philosophie s'insère parfaitement dans une architecture IoT distribuée (cf. figure 3.14). Voici les autres applications qui composent cet écosystème :
 - **Telegraf** permet de collecter et d'envoyer des métriques et des événements à partir de bases de données, de systèmes et de capteurs IoT. Il est utilisé dans l'architecture logicielle distribuée pour collecter des données sur les nœuds du cluster (charge du CPU, nombre de conteneurs déployés, mémoire RAM utilisée...)
 - **Chronograf** est une interface utilisateur et le composant administratif de la plateforme InfluxDB. Ce service offre la possibilité de créer rapidement des *dashboards* avec des visualisations en temps réel des données. Il se présente sous la forme d'une application WEB. Un exemple de tableau de bord est présenté à la figure 3.15.
 - **Kapacitor** est un moteur de traitement de données. Il peut être utilisé pour créer des alertes. **Kapacitor** n'est pas encore exploité dans la solution proposée.
2. Les images de conteneurs pour **Telegraf**, **InfluxDB**, **Chronograf** et **Kapacitor** sont poussées sur le dépôt officiel de Docker par l'équipe derrière ses services.
3. Il existe dans **LoRa Server** une intégration native permettant de sauvegarder les données directement vers InfluxDB.

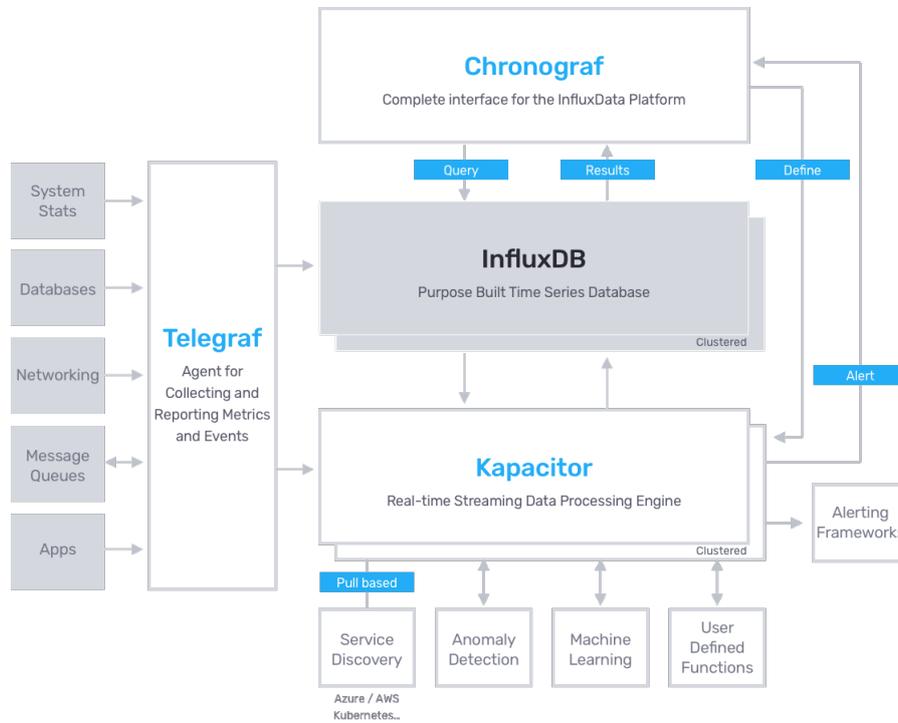


FIGURE 3.14 – Composants de l'écosystème TICK proposé par InfluxData

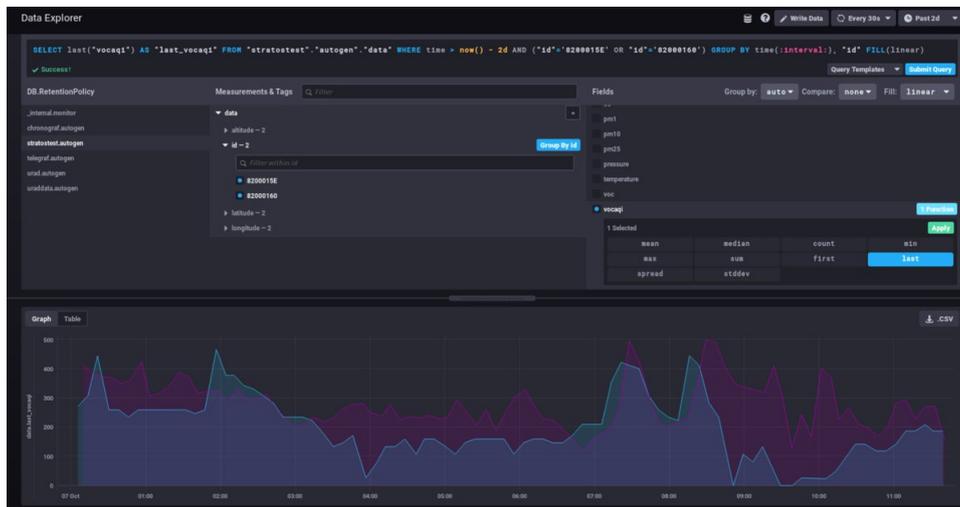


FIGURE 3.15 – Aperçu d'un tableau de bord créé avec Chronograf

Solution concrète

Après cette phase de recherche et d'évaluation, l'architecture IoT peut être déployée. Elle est présentée autour de trois axes : Les ressources matérielles sélectionnées, les périphériques externes utilisés et les ressources logicielles choisies durant la phase de recherche.

4.1 Architecture matérielle distribuée

D'un point de vue matériel, la solution proposée est composée de quatre nœuds. L'un d'entre eux prend le rôle de *master* du cluster déployé avec **Kubernetes**. Les autres, sont de simples nœuds dont la fonction est d'exécuter les différents services de l'architecture logicielle distribuée.

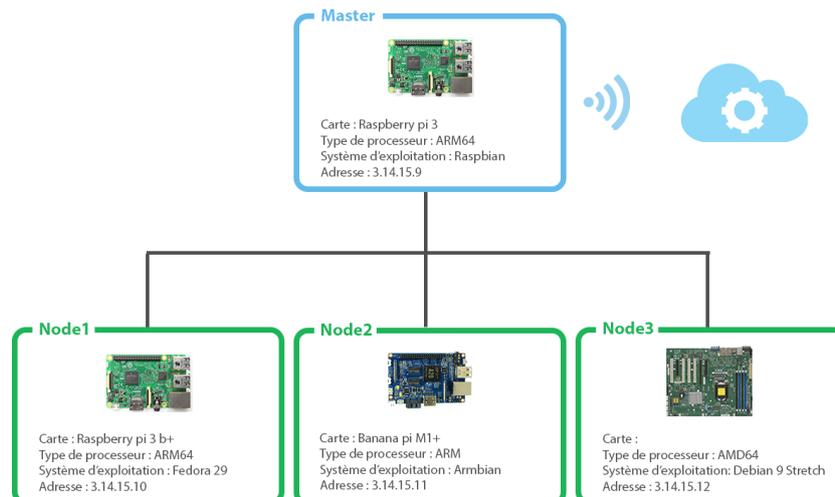


FIGURE 4.1 – Présentation de l'architecture matérielle distribuée déployée

4.1.1 Le master

Deux tâches distinctes incombent aux *master* :

1. Se plaçant dans un contexte de *Edge Computing*, l'architecture matérielle ne doit pas dépendre d'un routeur externe pour la communication entre les différentes machines. De ce fait, le *master* héberge un serveur Dynamic Host Configuration Protocol (DHCP) qui fournit une adresse IP du réseau 3.14.15.8/29 (réseau PI) aux différents nœuds. Le réseau Pi offre une plage de six adresses : quatre sont utilisées par les nœuds du cluster et deux sont rendues disponibles pour la communication avec les hôtes par une machine extérieure. Une règle de routage permet à chaque hôte du réseau PI de se connecter à Internet via l'antenne wifi du *master*.
2. Le *master* prend la fonction de nœud maître du *cluster*, par conséquent, il se doit d'exécuter un ensemble de trois services :
 - (a) **kube-apiserver** qui valide et configure les différents objets fournis par l'API de **Kubernetes** (pods, role...)
 - (b) **kube-controller-manager** qui surveille l'état partagé du cluster via l'apiserver et effectue des modifications en tentant de déplacer l'état actuel vers l'état souhaité. Ce processus prend la forme d'une boucle de contrôle.
 - (c) **kube-scheduler** qui prend en charge la planification des *pods* sur les nœuds.

4.1.2 Les nœuds

Comme définit au point 2.4, l'un des choix posés est que l'architecture matérielle doit être la plus hétérogène possible. C'est dans ce sens que trois périphériques ont été sélectionnés pour intégrer le *cluster*, dont en voici les caractéristiques :

- Le premier nœud est une carte Raspberry Pi de troisième version. Son processeur est un Broadcom BCM2837 64 bits à quatre cœurs ARM Cortex-A53 cadencés à 1,2 GHz. Il existe encore peu de systèmes d'exploitation compatibles avec des processeurs de type ARM64. Ce choix restreint nous a conduit à utiliser **Fedora**. Ce micro-ordinateur a été sélectionné, car il est souvent utilisé dans le développement de solutions IoT.
- Le second nœud est une carte Banana Pi . Équipé d'un processeur AllWinner A20 bicœurs ARM cadencés à 1GHz, le système d'exploitation installée sur ce périphérique est Armbian : un OS open source générique utilisé sur un grand nombre de micro-ordinateurs. Ce device fait office de périphérique *exotique* dans notre architecture matérielle.

- Le troisième nœud est un ancien ordinateur de bureau considéré comme obsolète. Le système d'exploitation installé sur cet ordinateur est Debian Stretch avec un environnement de bureau. Ce device s'insère dans une démarche de développement durable en utilisant un périphérique tombé en désuétude.

4.2 Périphériques externes

La communication entre les réseaux de capteurs et l'architecture passe par des périphériques externes prenant la forme de *dongle* USB.

4.2.1 Radio 802.15.4

La communication avec le WSN, utilisant le protocole de communication 802.15.4, est une carte Zolertia Firefly qui intègre les spécificités suivantes :

- un processeur ARM Cortex-M3 cadencé a 32MHz qui intègre une mémoire flash de 512Ko et une mémoire RAM de 32Ko
- une antenne on-board optimisée pour des signaux infra gigahertz
- deux connecteurs UFL pour des antennes externes de 2,4 GHz et infra gigahertz.
- un moteur de chiffrement AES 128 bits et 256 bits
- un moteur d'accélération matérielle pour un échange sécurisé de clés
- un micro contrôleur CP2104 / PIC intégré pour flasher depuis un port USB.
- une led permettant sept combinaisons de couleurs.



FIGURE 4.2 – Zolertia Firefly ZOL-BO001 revA1

4.2.2 Concentrateur Z-Wave

Le contrôleur Z-Stick Gen5 d'Aeon Labs est une antenne de communication Z-Wave+ avec batterie intégrée. Il se connecte sur un port USB pour communiquer avec le réseau Z-Wave. La batterie intégrée permet de procéder à l'inclusion ou l'exclusion des périphériques Z-Wave sans que le contrôleur ne soit connecté à un ordinateur. Étant donné qu'il s'agit d'un périphérique propriétaire, les spécificités ne sont pas documentées.



FIGURE 4.3 – Z-Stick Gen5 de chez AEOTEC

4.3 Architecture logicielle distribuée

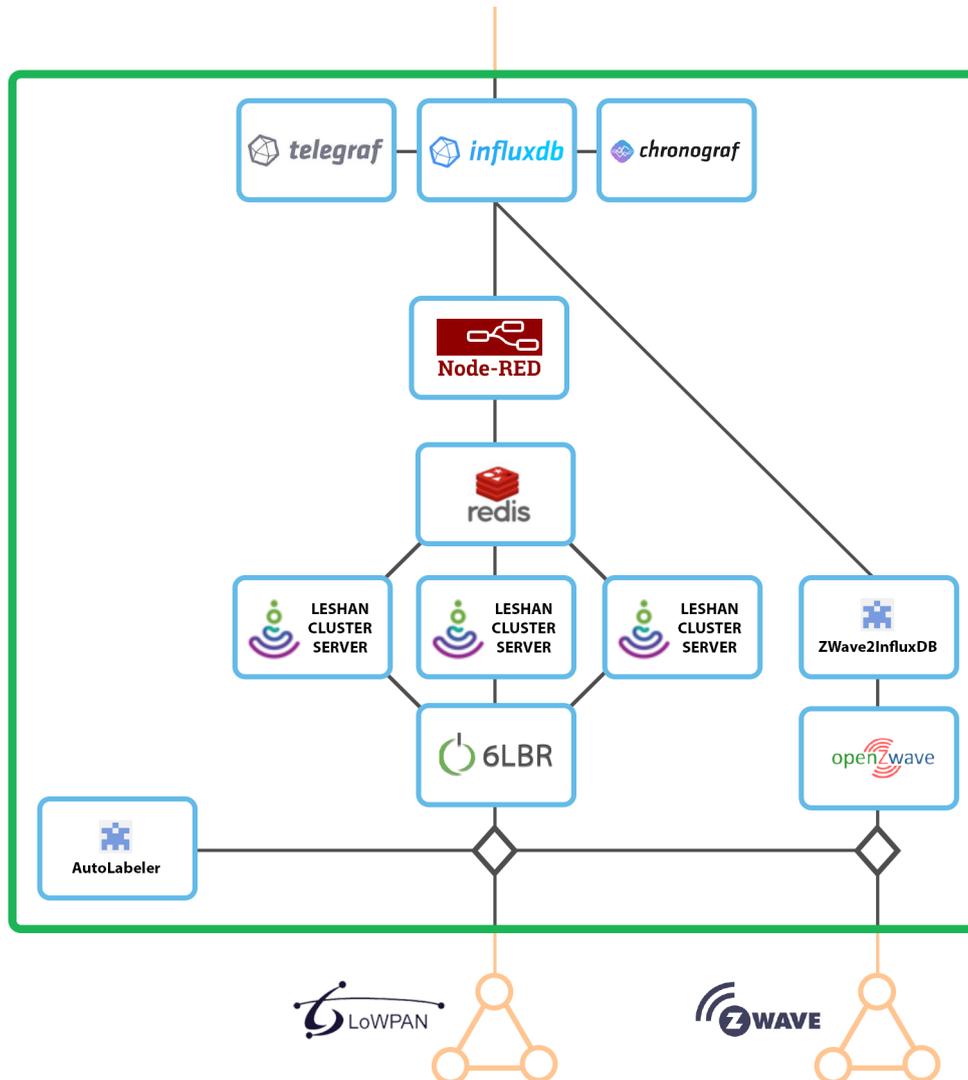


FIGURE 4.4 – Présentation de l'architecture logicielle distribuée déployée

4.3.1 de 802.15.4 à influxDB

L'intégration de la technologie baptisée du *802.15.4 au CoAP* dépend de 2 services : de 6LBR et d'un serveur LwM2M.

Comme annoncé au point 3.2, le serveur LwM2M déployé dans la solution est Leshan Cluster Server. Pour rappel, la gestion de la scalabilité de ce service repose sur une base de données **Redis**. La transmission de messages aux devices est implémentée autour de requêtes sur un canal spécifique : **LESHAN_REQ**. La réponse est réceptionnée sur le canal **LESHAN_RESP**.

La documentation autour des formats de message n'étant pas encore pleinement complète, la solution proposée pour connecter la base de données Redis à InfluxDB est d'utiliser **NodeRed** (comme présenté à la figure 4.4) .

Node-RED fournit une interface graphique dans laquelle les utilisateurs peuvent interagir avec des périphériques, des plates-formes logicielles et des services Web. Les composants prennent la forme de bloc que l'on peut interconnecter. Cette plateforme est complètement ouverte et autorise l'utilisation de nouveaux blocs développés par des tiers. Nous utilisons deux de ces contributions : *node-red-contrib-influxdb* et *node-red-contrib-redis*

De plus, une version conteneurisée de NodeRed, soutenue par ses propres concepteurs, est régulièrement mise à jour sur le dépôt officiel de Docker. Son intégration dans l'architecture logicielle distribuée est donc facilitée.

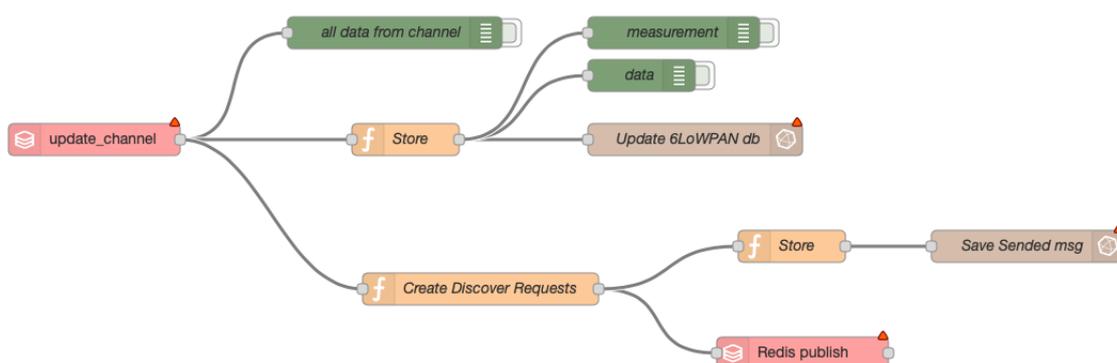


FIGURE 4.5 – Flow nodered faisant le lien entre les bases de données redis et influxDB

4.3.2 de Z-Wave à influxDB

OZWCP ne propose aucune intégration de sauvegarde vers une base de données. Gardant à l'esprit que l'architecture logicielle de ce projet est distribuée, le choix est fait de développer un service tiers, plutôt que de proposer une mise à jour de OZWCP¹ pour sauvegarder les données².

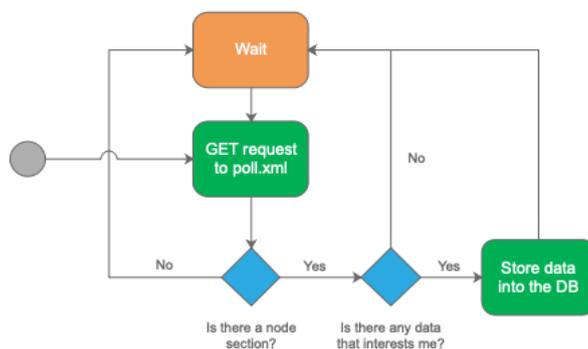


FIGURE 4.6 – Schéma bloc de l'implémentation de zwave2influxdb

Le processus de récolte de données de la page web de OZWCP réside dans une requête de type *GET* sur la page `http://<ozwcp.ip.or.domain.name>:8090/poll.xml` à intervalle régulier. Comme le montre le schéma 4.6, l'application développée utilise le même mécanisme pour récupérer les informations. Le service fera deux vérifications sur le fichier eXtensible Markup Language (XML) réceptionné avant de se connecter à la base de données :

1. La présence de la section *node* : lorsqu'un nœud met à jour ses données et les envoie à la passerelle, une section *node* est ajoutée au fichier xml, dans laquelle toutes les informations dudit nœud sont regroupées. Si aucun nœud n'envoie d'informations, cette section est inexistante.
2. La pertinence des données reçues : Les types donnés qui doivent être stockés sont regroupés dans une liste (Température, niveau de la batterie ...). Si un ou plusieurs de ces types est présents dans la section *node*, la valeur associée est stockée dans la base de données.

Cette implémentation offre l'avantage de réaliser un premier tri pour éviter de surcharger la base de données avec des informations qui ne sont pas congruentes.

1. Par exemple, sous la forme d'un *pull request* sur le dépôt **github** de la l'application
 2. Le code complet de ce service est mis à disposition à l'adresse <https://github.com/Sellto/zwave2influxdb>

5.1 Phase de tests

L'une des prérogatives de ce travail était, de déployer une architecture distribuée de ressources matérielles et logicielles offrant une continuité des services. Des tests de validation de cette haute disponibilité n'ont pas pu être réalisés, cependant les outils et mécanismes à utiliser, pour ces observations, ont déjà été identifiés.

5.1.1 Simulation d'un problème logiciel

En 2011, la société Netflix a inventé un outil qui désactive de manière aléatoire leurs instances de production : Chaos Monkey. L'objectif était de s'assurer que certaines défaillances n'avaient pas d'impact sur le client (Izrailevsky and Tseitlin, 2011).

Dans le même esprit, des concepteurs de logiciels ont implémentés **Kube-Monkey**, une application qui supprime de manière aléatoire des pods d'un cluster **Kubernetes**. Cet outil permet de tester la robustesse et la résilience de l'architecture logicielle distribuée.

5.1.2 Simulation d'un problème matériel

La défaillance d'un nœud ne doit pas avoir d'impact sur les services qui ne sont pas dépendants d'une interface physique. Il est difficile de simuler un problème matériel. Les essais prennent donc la forme de coupure de l'alimentation d'un nœud de manière contrôlée. Cette technique permet de monitorer et de mesurer le temps de réaction de l'architecture logicielle face aux pertes de connexion avec un nœud.

5.2 Intégration de la technologie basée sur LoRa

Par manque de temps, la technologie basée sur un WSN de capteurs LoRa n'a pas encore été intégrée dans l'architecture logicielle. Néanmoins, les ingénieurs de recherche du CETIC s'intéressent depuis un moment à la récolte de données depuis un réseau LoRaWAN privé. Par conséquent, les images de conteneurs nécessaires à son intégration dans l'architecture logicielle existante sont déjà construites et mises à disposition sur le dépôt d'image Docker. C'est lors de ce travail que le CETIC a émis l'idée de le généraliser et proposer ce travail de fin d'études.

En outre, LoRa Server propose nativement une intégration vers une base de données InfluxDB. La figure 5.1 présente la branche à ajouter à la solution déjà mise en place.

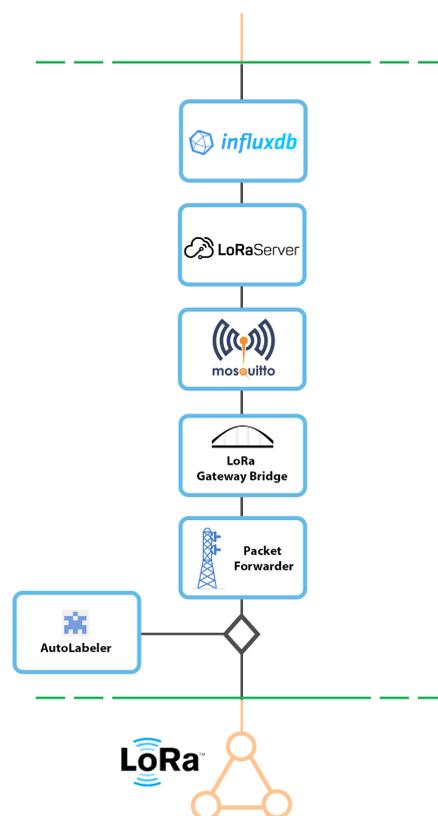


FIGURE 5.1 – Branche de l'architecture logicielle distribuée dédiée au réseau de capteurs LoRa

5.3 Intégration de services additionnels

Il va s'en dire que la solution proposée peut-être améliorée en y insérant des services additionnels. Le point suivant présente deux applications envisagées pour une évolution de l'architecture logicielle.

5.3.1 Local Registry

Dans une démarche de *edge computing*, l'utilisation de services situés dans le Cloud doit être minimisée. Cependant, l'architecture logicielle actuelle repose sur des images de conteneurs mises à disposition sur le dépôt officiel de Docker. Chaque rapatriement d'une image sur un nœud nécessite d'avoir une connexion Internet opérationnelle.

Une autre solution, est de sauvegarder localement les images de conteneurs susceptibles d'être utilisées par l'architecture logicielle distribuée.(Savic, 2019)

L'utilisation d'un dépôt privé en production permet de découpler la continuité des services et la disponibilité d'un service *Cloud-based*, tel que le dépôt officiel de Docker.

5.3.2 Jenkins X

L'approche CI/CD garantit une automatisation et une surveillance continues tout au long du cycle de vie des applications, des phases d'intégration et de test, jusqu'à la distribution et au déploiement. L'un des outils les plus connus dans ce domaine est Jenkins.

Jenkins automatise l'intégration continue, il réunit la livraison continue avec des scripts personnalisés, malheureusement son utilisation dans une architecture logicielle distribuée, basée sur des applications conteneurisées, est laborieuse. De ce fait, la fondation Jenkins a développé une solution adaptée à **Kubernetes** : **Jenkins X** (Jenkins, 2019).

Jenkins X se focalise sur l'expérience utilisateur pour rendre l'adoption d'un process de CI/CD sur Kubernetes aussi simple et rapide que possible. Selon la documentation, Jenkins X met automatiquement en place la CI/CD d'une application ce qui comprend : Le build du code, son packing dans un conteneur Docker, son déploiement dans le *cluster* **Kubernetes** (De Loof, 2019).

L'un des aspects de ce travail est l'automatisation d'un maximum de tâches, l'intégration de **Jenkins X** dans la solution existante peut certainement y contribuer.

Conclusions

Dans le but d'optimiser la récolte et le traitement de données à l'échelle d'un bâtiment ou d'une cité, il est primordial de décentraliser le travail à réaliser. Pour répondre à cette problématique, une architecture IoT moins dépendante des services *cloud-based* est proposée. Ainsi, l'architecture matérielle distribuée réalisée dans ce travail de fin d'études se matérialise sous la forme d'un *cluster* de machines hétérogènes (avec différentes spécifications) interconnectées localement. La preuve de concept est amenée par un *cluster* de quatre machines.

D'un point de vue du logiciel, le projet a été imaginé autour de trois types de technologie de récolte de données. On en compte deux qui sont d'ores et déjà opérationnelles : l'une basée sur le protocole propriétaire Z-Wave et la seconde reposant sur le réseau de capteurs 6LoWPAN. La technologie LoRa, quant à elle, n'est pas encore incluse dans la solution. Néanmoins, le travail de recherche effectué pour l'intégration des deux premières technologies a permis de développer une architecture logicielle conçue pour faciliter l'addition de nouveaux protocoles de communication.

Les services sont déployés automatiquement en fonction des interfaces physiques présentes sur chaque machine. Localement, les applications prennent la forme de micro services conteneurisés. De ce fait, un orchestrateur est utilisé pour garantir une continuité de services et assurer la gestion de la scalabilité. En outre, la phase de recherche a permis de mettre au point une solution permettant d'intégrer un service conteneurisé utilisant un adressage IPv6 dans une infrastructure IPv4. La méthode mise en place est pleinement fonctionnelle et réutilisable.

Enfin, les données en provenance des réseaux de capteurs sont rapatriées dans une base de données commune. Par conséquent, l'accès aux données est uniformisé et indépendant de la source. Seules certaines informations prétraitées sont partagées sur le *cloud*.

En conclusion, ce projet a permis de mettre en évidence qu'il est tout à fait envisageable de se distancer d'Internet, en particulier pour notre cas d'utilisation. Il apporte une vision plus automatisée aux solutions existantes. Cependant, des améliorations peuvent être apportées pour éviter toute intervention humaine en cas de pannes.

Certains aspects de la mondialisation sont remis en doute, l'humanité tend à redonner de l'importance à la production et le traitement local. Dans cet esprit, on voit émerger de plus en plus de projets technologiques souhaitant réduire leur recours aux services centralisés. De plus, la tendance actuelle aspire à transformer nos villes en *smart-cities*. Ou encore, à faire converger nos entreprises vers un mode de fonctionnement de plus en plus numérique, en les élevant au rang d'industries 4.0. Ces aspirations passent inévitablement par l'utilisation de capteurs et d'actionneurs, sans oublier le traitement efficace de la masse de données qu'ils génèrent et qui dépend souvent d'Internet. Ce travail a permis de démontrer que ces deux tendances sont conciliables.

Ce travail apporte une solution face aux perspectives sociétales actuelles. Il est sans aucun doute encore amené à évoluer, car, comme Bob Khan l'inventeur d'Internet l'a dit : "On ne peut pas regarder dans une boule de cristal pour voir l'avenir. Ce qu' Internet va devenir est ce qu'en fera la société".

Bibliographie

- A. Achour, L. Deru, and J. C. Deprez. Mobility Management for Wireless Sensor Networks A State-of-the-Art. *Procedia Computer Science*, 52 :1101–1107, 2015. doi : 10.1016/j.procs.2015.05.126.
- P. Baronti, P. Pillai, V. W. Chook, S. Chessa, A. Gotta, and Y. F. Hu. Wireless sensor networks : A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications*, 30(7) :1655–1695, May 2007. doi : 10.1016/j.comcom.2006.12.020.
- E. Berdyeva. ITU-T RECOMMENDATION. *T G.*, page 128, Dec. 2015.
- D. Bernstein. Containers and Cloud : From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1 (3) :81–84, Sept. 2014. doi : 10.1109/MCC.2014.51.
- A. Cazes and J. Delacroix. *Architecture des machines et des systèmes informatiques : cours et exercices corrigés*. Dunod, 2008. ISBN 978-2-10-053945-1. OCLC : 520980626.
- CoreOS. docs/deprecated-features.md at master · coreos/docs · GitHub, Feb. 2017. URL <https://github.com/coreos/docs/blob/master/fleet/deprecated-features.md>.
- N. De Loof. A la découverte de Jenkins. *Programmez*, pages 41–46, 2019.
- L. Deru, S. Dawans, M. Ocaña, B. Quoitin, and O. Bonaventure. Redundant Border Routers for Mission-Critical 6lowpan Networks. In *Real-World Wireless Sensor Networks*, volume 281, pages 195–203. Springer International Publishing, 2014. doi : 10.1007/978-3-319-03071-5_20.
- Docker. Dockerfile Reference, 2019. URL <https://docs.docker.com/engine/reference/builder/>.
- A. George. An overview of Risc vs. CISC. In *The Twenty-Second Southeastern Symposium on System Theory.*, pages 436–438, Mar. 1990. doi : 10.1109/ssst.1990.138185.
- R. Guichard. Le point sur les container runtimes, Dec. 2018.

- J. Hui, D. Culler, and S. Chakrabarti. 6lowpan Incorporating IEEE 802.15.4 into the IP architecture. page 17, 2009.
- lotPoint. Z-Wave Tutorial – IoT-Point, Mar. 2015. URL <https://iotpoint.wordpress.com/z-wave-tutorial/>.
- B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of Docker as Edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135. IEEE, Aug. 2015. doi : 10.1109/ICOS.2015.7377291.
- Y. Izrailevsky and A. Tseitlin. The Netflix Simian Army – Netflix TechBlog – Medium, July 2011. URL <https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116>.
- Jenkins. Jenkins X, 2019. URL <https://jenkins.io/projects/jenkins-x/>.
- J. Jorda. *Processeurs ARM, Architecture et langage d'assemblage*. Dunod, 2010. ISBN 2-10-056073-5.
- Kubernetes. Using RBAC Authorization - Kubernetes, 2019. URL <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>.
- N. Lutchansky. TAYGA - NAT64 for Linux, June 2011. URL <http://www.litech.org/tayga/>.
- S. Monnier. Techniques et caractéristiques modernes qui permettent de donner aux ordinateurs actuels toute leur puissance., 2019.
- S. P. Morse, W. B. Pohlman, and B. W. Ravenel. The Intel 8086 Microprocessor : a 16-bit Evolution of the 8080. *Computer*, page 10, 1978. doi : 10.1109/C-M.1978.218219.
- K. E. Nolan, W. Guibene, and M. Y. Kelly. An evaluation of low power wide area network technologies for the Internet of Things. In *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 439–444. IEEE, Sept. 2016. doi : 10.1109/IWCMC.2016.7577098.
- ozw. Introduction to OpenZWave, Apr. 2019. URL <http://www.openzwave.net/dev/>.
- D. A. Patterson and C. H. Sequin. RISC I : a reduced instruction set VLSI computer. In *25 years of the international symposia on Computer architecture (selected papers) - ISCA '98*, pages 216–230. ACM Press, 1981. doi : 10.1145/285930.285981.
- S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. IBM Power9 Processor Architecture. *IEEE Micro*, 37(2) :40–51, Mar. 2017. doi : 10.1109/MM.2017.40.
- D. Savic. How To Set Up a Private Docker Registry on Top of DigitalOcean Spaces and Use It with DigitalOcean Kubernetes | DigitalOcean, Apr. 2019. URL <https://www.digitalocean.com/community/tutorials>.
- Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). Technical Report RFC7252, RFC Editor, June 2014.

TTN. The Things Network, 2019. URL <https://www.thethingsnetwork.org/>.

Wiki. Z-Wave — Wikipédia, Jan. 2017. URL <https://fr.wikipedia.org/wiki/Z-Wave>.

W. Zhao, S. Lin, J. Han, R. Xu, and L. Hou. Design and Implementation of Smart Irrigation System Based on LoRa. In *2017 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, Dec. 2017. doi : 10.1109/GLOCOMW.2017.8269115.

zwaveme. Z-Wave.Me – Smart Home Engineering, 2017. URL <https://z-wave.me/>.

Annexe A

Résultats d'expérimentations

A.1 Exclusion de Docker Swarm comme Orchestrateur

```
[
  {
    "Name": "myIPv6net",
    "Id": "cgxs2wvs4jftluhx2ce2ptmq",
    "Created": "2018-11-13T17:49:27.151440467Z",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": true,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "2001::/64",
          "Gateway": "2001::1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": true,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": null,
    "Options": {
      "com.docker.network.driver.overlay.vxlanid_list": "4096"
    },
    "Labels": null
  }
]
```

FIGURE A.1 – Description du réseau lorsque seul le réseau IPv6 est initialisé

```

{
  "Name": "myDSnet",
  "Id": "32oqmdu3sh5mpn33kwk0epewz",
  "Created": "2018-11-13T18:50:38.926961706+01:00",
  "Scope": "swarm",
  "Driver": "overlay",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.30.0.0/16",
        "Gateway": "172.30.0.1"
      },
      {
        "Subnet": "2001::/64",
        "Gateway": "2001::1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": true,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "myDSnet-sbox": {
      "Name": "myDSnet-endpoint",
      "EndpointID": "67a961957d3ae551a0c39be9373e77d3ac1a0a64126f74f9edea12cafb774e6a",
      "MacAddress": "02:42:ac:1e:00:02",
      "IPv4Address": "172.30.0.2/16",
      "IPv6Address": ""
    }
  },
  "Options": {
    "com.docker.network.driver.overlay.vxlanid_list": "4097,4098"
  },
  "Labels": {},
  "Peers": [
    {
      "Name": "6b9212d478c8",
      "IP": "10.9.2.106"
    }
  ]
}

```

FIGURE A.2 – Description du réseau initialisé par défaut

```
[
  {
    "Name": "myDSnet",
    "Id": "xqlgmxwii0elw7ewhly75m03m",
    "Created": "2018-11-13T17:52:58.54367013Z",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": true,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "2001::/64",
          "Gateway": "2001::1"
        },
        {
          "Subnet": "172.30.0.0/16",
          "Gateway": "172.30.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": true,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": null,
    "Options": {
      "com.docker.network.driver.overlay.vxlanid_list": "4099,4100"
    },
    "Labels": null
  }
]
```

FIGURE A.3 – Description du réseau lorsque le réseau IPv6 est initialisé en premier

```

{
  "Name": "myDSnet",
  "Id": "32oqmdu3sh5mpn33kwk0epewz",
  "Created": "2018-11-13T18:50:38.926961706+01:00",
  "Scope": "swarm",
  "Driver": "overlay",
  "EnableIPv6": false,
  "IPAM": {
    "Driver": "default",
    "Options": null,
    "Config": [
      {
        "Subnet": "172.30.0.0/16",
        "Gateway": "172.30.0.1"
      },
      {
        "Subnet": "2001::/64",
        "Gateway": "2001::1"
      }
    ]
  },
  "Internal": false,
  "Attachable": false,
  "Ingress": true,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {
    "myDSnet-sbox": {
      "Name": "myDSnet-endpoint",
      "EndpointID": "67a961957d3ae551a0c39be9373e77d3ac1a0a64126f74f9edea12cafb774e6a",
      "MacAddress": "02:42:ac:1e:00:02",
      "IPv4Address": "172.30.0.2/16",
      "IPv6Address": ""
    }
  },
  "Options": {
    "com.docker.network.driver.overlay.vxlanid_list": "4097,4098"
  },
  "Labels": {},
  "Peers": [
    {
      "Name": "6b9212d478c8",
      "IP": "10.9.2.106"
    }
  ]
}

```

FIGURE A.4 – Description du réseau lorsque le réseau IPv4 est initialisé en premier

Annexe B

Code sources

B.1 Fichier 6lbr.conf

TABLE B.1 – Codes source du fichier 6LBR.conf en fonction des modes de configuration

RAW-ETHERNET	BRIDGE	ROUTING
<pre>1 RAW_ETH=1 2 BRIDGE=0 3 DEV_ETH=eth0</pre>	<pre>1 RAW_ETH=0 2 BRIDGE=1 3 CREATE_BRIDGE=1 4 DEV_ETH=eth0 5 DEV_BRIDGE=br0 6 DEV_TAP=tap0</pre>	<pre>1 RAW_ETH=0 2 BRIDGE=0 3 DEV_ETH=eth0 4 DEV_TAP=tap0</pre>

B.2 Exemple de messages MQTT utilisés par EMQx

FIGURE B.1 – Exemple de message MQTT envoyé via le plugin LwM2M de EMQx

```
1 {
2   "reqID": "2",
3   "msgType": "discover",
4   "data": {
5     "path": "/3/0"
6   }
7 }
```

FIGURE B.2 – Exemple de message MQTT reçu par le plugin LwM2M de EMQx

```
1 {
2   "reqID": "2",
3   "msgType": "discover",
4   "data": {
5     "code": "2.05",
6     "codeMsg": "content",
7     "content": [
8       "</3/0>;dim=8",
9       "</3/0/0>",
10      "</3/0/1>",
11      "</3/0/4>",
12      "</3/0/16>"
13    ]
14  }
15 }
16 \end{figure}
```

Tableaux des figures

2.1	Comparaison entre utilisation de machines virtuelles et de conteneur	7
2.2	Interactions entre les <i>containers runtimes</i>	8
2.3	Schéma de principe d'un <i>cluster</i> orchestré	9
2.4	<i>Protocol stack</i> de la technologie 1	12
2.5	Datagramme de paquet 802.15.4 après fragmentation par la couche d'adaptation	14
2.6	Topologie de réseau supporté en 802.15.4	15
2.7	Badges officiels apposés sur les modules certifiés Z-Wave et Z-Wave+	16
2.8	<i>Protocol stack</i> de la technologie 2	17
2.9	Schéma de principe des communications LoRa en fonction des classes de péri- phériques	20
3.1	Schéma de principe de l'application 6LBR.	23
3.2	Résultats de l'affichage des logs de l'application 6LBR.	23
3.3	Interface WEB de gestion fournie par OZWCP	26
3.4	Taille des images réalisées	28
3.5	Schéma Bloc du playbook Ansible MultiArchBuilder	30
3.6	Commande utilisée pour la création d'un réseau <i>dual stack</i> de type <i>ingress</i> . .	34
3.7	Commande alternative utilisée pour la création d'un réseau <i>dual stack</i> de type <i>ingress</i>	34
3.8	Comparaison des communautés de Docker Swarm et Kubernetes au 28/04/2019	35
3.9	Schéma Bloc du fonctionnement de l'application AutoLabeler	37
3.10	Schéma de principe de l'application conteneurisée 4LBR	42
3.11	Interface WEB de gestion de Leshan Server Demo	46
3.12	Interface WEB de gestion de EMQx	47
3.13	Schéma bloc de l'implémentation de go-lwm2m	49

3.14 Composants de l'écosystème TICK proposé par InfluxData	53
3.15 Aperçu d'un tableau de bord crée avec Chronograf	53
4.1 Présentation de l'architecture matérielle distribuée déployée	54
4.2 Zolertia Firefly ZOL-BO001 revA1	56
4.3 Z-Stick Gen5 de chez AEOTEC	57
4.4 Présentation de l'architecture logicielle distribuée déployée	58
4.5 Flow nodered faisant le lien entre les bases de données redis et influxDB	59
4.6 Schéma bloc de l'implémentation de zwave2influxdb	60
5.1 Branche de l'architecture logicielle distribuée dédiée au réseau de capteurs LoRa	62
A.1 Description du réseau lorsque seul le réseau IPv6 est initialisé	69
A.2 Description du réseau initialisé par défaut	70
A.3 Description du réseau lorsque le réseau IPv6 est initialisé en premier	71
A.4 Description du réseau lorsque le réseau IPv4 est initialisé en premier	72
B.1 Exemple de message MQTT envoyé via le plugin LwM2M de EMQx	74
B.2 Exemple de message MQTT reçu par le plugin LwM2M de EMQx	74

Liste des tableaux

3.1	Modes de configuration des interfaces dans 6LBR	24
3.2	Tableau comparatif récapitulatif des différents serveurs LwM2M envisagés . . .	50
B.1	Codes source du fichier 6LBR.conf en fonction des modes de configuration . . .	73

Acronymes

6LoWPAN IPv6 Low power Wireless Personal Area Networks

AMD Advanced Micro Devices

API Application Programming Interface

ARM Advanced Risk Machine

BPSK décalage de phase binaire

CISC Complex Instruction Set Computing

CoAP Constrained Application Protocol

CPU Central Processing Unit

CRI Container Runtime Interface

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

DTLS Datagram Transport Layer Security

FFD Full Function Devices

HTTP HyperText Transfer Protocol

IBM International Business Machines Corporation

IoT Internet of Things

JSON JavaScript Object Notation

LwM2M Lightweight Machine to Machine

M2M machine-to-machine

MQTT Message Queuing Telemetry Transport

MTU Maximum Transmission Unit

NAT Network Address Translation

NAT64 Network Address Translation 6to4

NDP Neighbor Discovery Protocol

OCI Open Container Initiative

OS Operating System

OZWCP OpenZWave Control Panel

PSDU Physical layer Service Data Unit

Redis REmote DIctionary Server

REST Representational State Transfer

RFD Reduced Function Devices

RISC Reduced Instruction Set Computing

RKT Rocket

RPL Routing Protocol for Low power and Lossy Networks

UDP User Datagram Protocol

WSN Wireless Sensors Network

XML eXtensible Markup Language

YAML Yet Another Markup Language

Z-Wave Z-Wave

Border Router Un *Border Router* est une passerelle entre deux technologies différentes d'un point de vue de la couche de liaison

BPSK Technique de modulation par décalage de phase qui utilise deux phases qui sont séparées de 180° . Cette modulation est la plus robuste de toutes les modulation par décalage de phase car il faut une grande déformation du signal pour que le démodulateur se trompe sur le symbole reçu. Cependant on ne peut moduler qu'un seul bit par symbole.

Datagram Transport Layer Security est un protocole conçu pour protéger les données privées en prévenant la falsification, les écoutes, et la contre-façon dans les communications. Il est basé sur le Transport Layer Security (TLS), qui est un protocole qui sécurise les réseaux de communications informatiques. La différence principale entre le DTLS et le TLS est que le DTLS utilise l'UDP et le TLS utilise le TCP.

Proxy est un composant logiciel informatique qui joue le rôle d'intermédiaire en se plaçant entre deux hôtes pour faciliter ou surveiller leurs échanges.

QPSK Technique de modulation par décalage de phase en quadrature qui utilise un diagramme de constellation à quatre points, à équidistance autour d'un cercle. Ce type de modulation peut coder deux bits par symbole.