RAPPORT DE CONCEPTION
Mycélium

Guillaume CHAUVEAU, Pieyre IACONE, Florian DABAT

Élodie JUVÉ, Yifan TIAN, Victoria Maria VELOSO RODRIGUES, Haoying ZHANG

Encadrants : Nikolaos PARLAVANTZAS, Christian RAYMOND

Intervenants : Laurent ROYER, Laurent LONGUEVERGNE, Nicolas LAVENANT, Guillaume PIERRE

2021-2022

Résumé

Une équipe pluridisciplinaire nous sollicite pour faire un suivi environnemental d'un espace naturel à l'aide d'un réseau de capteurs intelligents. Pour ce faire, nous avons à disposition divers matériels et technologies, notamment pour la collecte et l'envoi des données et la mise en place d'une infrastructure *fog*.

Table des matières

1 Introduction

1.1 Mycélium en quelques mots

Mycélium est un projet de suivi environnemental à l'aide d'un réseau de capteurs météorologiques intelligents. Il a pour but d'être déployé sur la zone de la Croix Verte du campus de l'université de Rennes 1, afin de suivre l'évolution de cet espace qui a été renaturé récemment.

Pour ce faire, nous disposons de capteurs connectés à un agrégateur, c'est-à-dire un boîtier disposant d'un micro-contrôleur et d'un module de communication LoRaWAN [1], appelé nœud SoLo. Conçu par le laboratoire de physique de Clermont, il a la possibilité d'utiliser d'autres capteurs tels qu'un pluviomètre. La collecte et le traitement des données mesurées sont effectués par une infrastructure logicielle polyvalente baptisée Mycélium Core, déployée sur un *cluster* de Raspberry Pi au département Informatique de l'INSA.

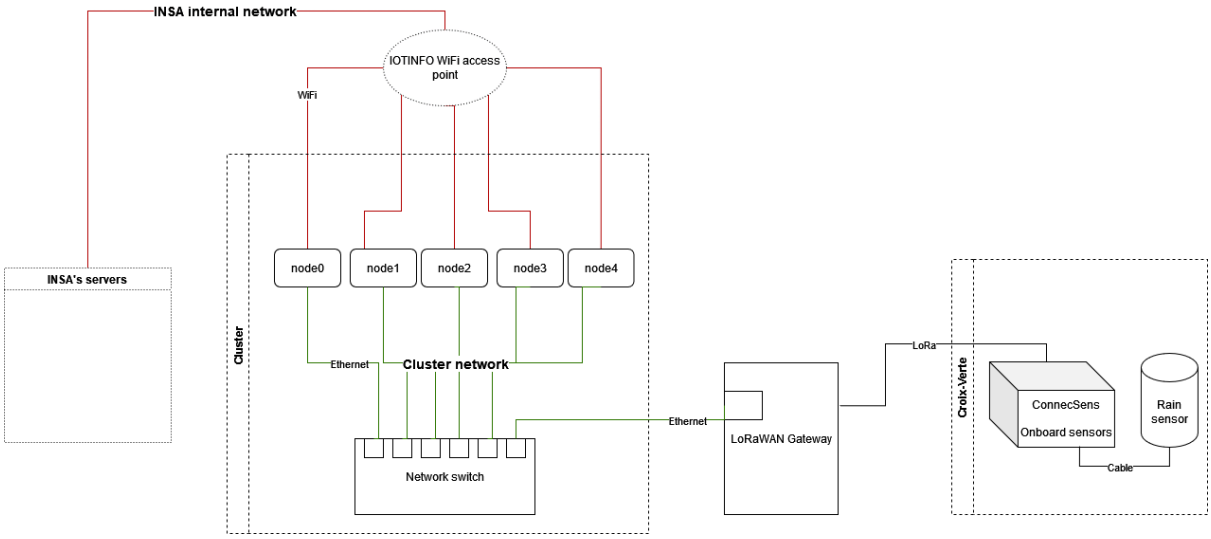

FIGURE 1 – Composants physiques de Mycélium

1.2 Le contenu de ce rapport

Ce rapport présente la conception du projet Mycélium. Il se doit donc de décrire l'architecture interne du logiciel, sa modélisation et l'interfaçage des différents modules entre eux. On peut diviser le projet Mycélium en 4 sous-parties pour décrire de manière optimale sa conception : dans un premier temps, on va présenter l'architecture du nœud SoLo, son modèle de fonctionnement et son système d'alarmes. Puis nous présenterons le fonctionnement du *cluster* de Raspberry Pi, afin d'expliquer l'architecture de Mycélium core, une infrastructure permettant au *cluster* de recvoir les données. Enfin nous verrons les fonctions de traitement appliquées sur les données dans le cadre du suivi environnemental de la Croix Verte.

2 Nœud SoLo

Nous avons des nœuds SoLo qui sont équipés de capteurs conçus pour réaliser des mesures de l'environnement sur la Croix Verte et transmettre des données. Quatre capteurs internes sont intégrés dans un nœud SoLo :

— un capteur de température et d'humidité,
— un capteur de luminosité,
— un accéléromètre,
— un capteur de pression.

Dans cette section, nous présenterons les interactions entre le système de nœuds et les acteurs, les paramétrages dans le fichier de configuration, le diagramme de classe du *firmware* de nœud, une amé-

lioration de la détection de dysfonctionnements au premier démarrage ainsi qu'un nouveau système d'alarmes.

2.1 Diagramme de cas d'utilisation

Dans un premier temps, nous allons montrer rapidement l'ensemble d'actions réalisées par le système en interaction avec les acteurs en présentant le diagramme de cas d'utilisation suivant 2.

Le système auquel nous nous intéressons dans cette partie est le nœud SoLo, et les acteurs sont les utilisateurs qui peuvent le configurer via le fichier de configuration que nous présenterons dans la section 2.2.

Dans ce diagramme, les fonctionnalités existantes que nous avons sur le nœud sont les suivantes :

— prendre des mesures à une fréquence définie pour chaque capteur,
— émettre des mesures à une fréquence définie pour tous les capteurs via le réseau LoRaWAN,
— configurer des alarmes,
— configurer des interruptions.

La fonctionnalité qui permet de configurer des alarmes imbriquées est en phase d'implémentation. Nous le présenterons dans la partie 2.5 pour le nouveau système d'alarmes.

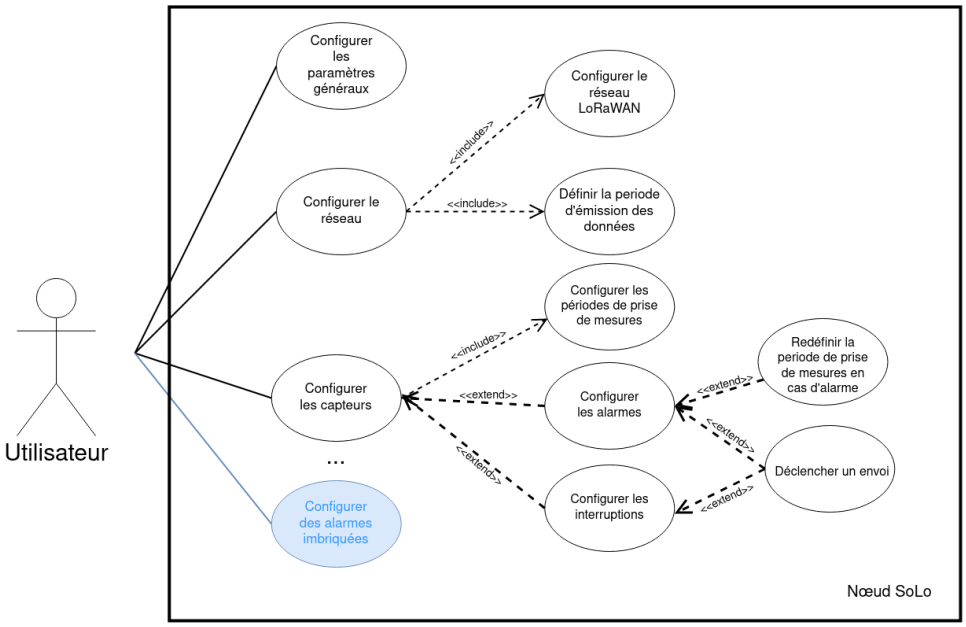

FIGURE 2 – Cas d'utilisation : interactions entre l'utilisateur et le nœud

2.2 Fichier de configuration

Après avoir présenté les cas d'utilisations du nœud, nous allons nous intéresser dans cette partie aux paramétrages dans le fichier de configuration, qui permet d'utiliser des fonctionnalités existantes pour répondre à nos besoins.

Pour déployer le nœud, il convient de brancher un PC sur le port USB afin de procéder à la configuration de nœud. Le nœud est reconnu comme un périphérique de stockage de masse (un disque ou une clef USB) par l'ordinateur.

Quatre dossiers seraient apparus dans la racine du disque : «env», «log», «output» et «proc». Nous nous intéressons ici seulement aux dossiers «env», «log», et «output» illustrés sur la figure 3.

FIGURE 3 – La racine du disque

Le fichier de configuration se trouve dans le dossier «env». Il s'agit d'un fichier nommé *config.json* qui permet de personnaliser le fonctionnement du nœud SoLo mais dans la limite des options prévues. Il utilise le format JSON, un format texte d'échange et de stockage de données.

En particulier, il permet de :

— Configurer les paramètres réseaux utilisés par le nœud pour se connecter au réseau sans fil.
— D'informer le nœud des capteurs qui lui sont connectés.
— De configurer les capteurs en question.
— De configurer les périodes de mesure des capteurs et la période d'émission des données sur le réseau sans fil.

2.2.1 Configuration générale

Dans le cadre de la configuration générale, nous définissons les paramètres d'identité comme par exemple le nom du nœud, le nom de l'expérience et son numéro d'identité unique. Nous définissons aussi la période de prise de mesure, déverrouillage ou verrouillage du fichier de sortie CSV (qui se trouve dans le dossier «output» illustré sur la figure 3) et le mode de travail.

2.2.2 Configuration du réseau

La clé «network» de l'objet JSON correspond à la configuration du réseau. Nous pouvons préciser la période d'envoi pour tous les capteurs, le protocole de réseau utilisé et les paramètres nécessaires à la connexion.

Nous utilisons le protocole de réseau LoRaWAN, et passons tous les paramètres nécessaires pour le configurer. Avec les étiquettes «periodSec», «periodMn» , «periodHr» ou «periodDay», il est possible de définir la période d'émission des données, en secondes, en minutes, en heures ou en jours respectivement. L'envoi des mesures s'effectue toutes les 4 heures, comme indiqué dans la figure 4 : c'est un bon compromis, car cela permet de limiter le nombre d'envois via LoRaWAN dans une journée, et donc de moins solliciter la batterie du nœud. Cela évite les embouteillages du réseau, tout en faisant attention à ne pas dépasser la limite de la taille d'envoi à chaque fois (242 octets). Cela revient donc à effectuer 6 envois par jours.

Un autre paramètre que nous avons besoin de configurer pour le réseau est le débit de données (soit le terme anglais *data rate* utilisé dans le fichier de configuration) sur LoRaWAN. Plus le débit est grand, plus la vitesse de transmission est élevée et plus une trame peut contenir de données, mais moins de distance elle peut couvrir. Dans le fichier de configuration, nous pourrions personnaliser ce paramètre en mettant une valeur entre 0 et 5 pour la balise «dataRate» (qui n'est pas représentée dans la figure 4), et il faudra trouver la valeur optimale en combinant la distance qui peut être recouverte et la taille de données à chaque envoi. Nous pouvons également utiliser un débit de données adaptatif en mettant *true* à la balise «useAdapativeDataRate». Il n'est pas conseillé d'utiliser ce mode de fonctionnement pour les nœuds mobiles, ou si la passerelle est mobile, car la vitesse d'adaptation du réseau est très lente, mais comme nous ne déplaçons pas les nœuds et la passerelle dans la plupart du temps, nous avons choisi ce deuxième fonctionnement, qui est aussi indiqué dans la figure 4.

Le paramètre «sendTimeoutSec» est pour définir la durée maximale du temps d'attente pour la réception de l'accusé de réception (ACK[2]), en secondes. Nous le mettons par 60 secondes, qui est la valeur maximale, pour avoir assez de temps à recevoir un ACK après une transmission de données.

Le dernier paramètre «publicNetwork» est mis à *false* parce que nous utilisons le réseau LoRaWAN privé.

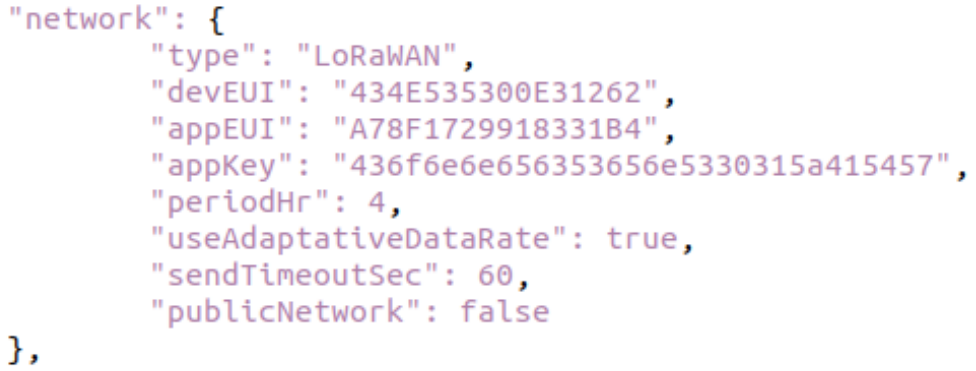

FIGURE 4 – Configuration du réseau

2.2.3 Configuration des logs

Nous disposons d'un fichier log qui contient toutes les lignes de log, qui se trouve dans le dossier «log» de la racine de disque 3. Les lignes de log permettent de suivre la vie de nœud et de détecter des problèmes éventuels.

Les messages de log ont un niveau de priorité qui permet de les filtrer selon le niveau de verbosité que vous souhaitez avoir.

Les deux niveaux de verbosité les plus utilisés sont «INFO» et «DEBUG» :

— **«INFO»** : Le niveau des messages ordinaires. Utilisé par le nœud pour indiquer ce qu'il fait, avec suffisamment de détails pour suivre le «fil de ses pensées» et ses choix les plus importants, mais sans s'attarder sur des détails trop précis.

— **«DEBUG»** : Le niveau des messages qui va beaucoup plus dans le détail. Le nombre de messages produits peut être important et leur utilité pas forcément justifiée, surtout si l'on ne s'attend pas à des problèmes particuliers. Ce niveau de détail est plutôt utilisé pour mieux cerner un problème identifié.

Dans l'environnement de développement, nous définissons le niveau de log par défaut par «DEBUG» et nous mettons «EXT_UART» pour le paramètre «serial», comme indiqué dans l'image 5. Ce deuxième paramètre permet de lire en temps réel les messages du log à partir d'une sortie série à l'aide d'un adaptateur USB à TTL 3,3V. Cela nous simplifie le processus de débogage, permettant au développement et au déploiement de se faire de manière plus agile.

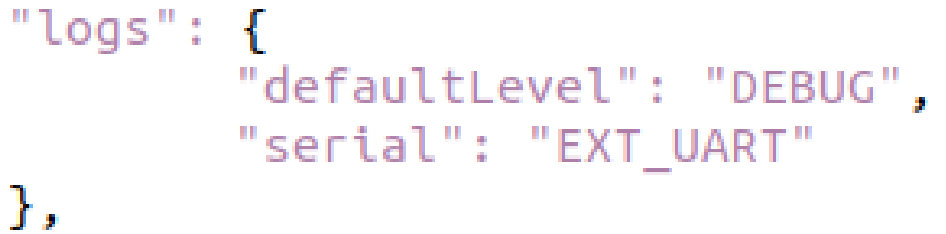

FIGURE 5 – Configuration du logs

2.2.4 Configuration des capteurs

Le nœud SoLo contient quatre capteurs internes qui sont disponibles. Ils peuvent mesurer les grandeurs suivantes :

— la luminosité, capteur concerné : OPT3001, nommé «Lumi»,

— la pression atmosphérique : capteur lié : LPS25, nommé «Press»,

— le degré d'humidité relative et la température de l'air, capteur utilisé : SHT35, nommé «Humi-Temp»,
— l'accélération et les mouvements du nœud, capteur concerné : LIS3DH, nommé «Accelero».

Nous définissons, pour tous les capteurs internes, une fréquence de prise de mesures une fois par heure par défaut. En s'inspirant par la fréquence de la mise à jour de la base de données de météo Rennes-St Jacques [21]. Pour rappel, les mesures sont envoyées sur le réseau LoRaWAN toutes les 4 heures. En effet, cela revient à envoyer 4 prises de mesures à la fois, ce qui est raisonnable par rapport à la capacité du réseau, expliqué dans la partie 2.2.2.

Autres paramètres importants que nous pouvons définir sont les interruptions et les alarmes. Ces paramètres sont indépendants des capteurs et sont liés avec les scénarios présentés dans la partie 5.2.

Les paramètres sont définis à la suite de la clé «sensors» dans le fichier de configuration qui sont indiqués dans la figure 6.

```
"sensors": [{
        "name": "Lumi",
        "type": "OPT3001",
        "interruptChannel": "SHT35_INT",
        "sendOnInterrupt": true,
        "periodHr": 1
},{
        "name": "Press",
        "type": "LPS25",
        "periodHr": 1
},{
        "name": "HumiTemp",
        "type": "SHT35",
        "periodHr": 1,
        "alarm": {
                "sendOnAlarmSet": true,
                "temperatureHighSetDegC" : 40,
                "temperatureHighClearDegC" : 38,
                "temperatureLowSetDegC" : -5,
                "temperatureLowClearDegC" : 0,
                "periodMn": 5,
        }
},{
        "name": "Accelero",
        "type": "LIS3DH",
        "periodHr": 1,
        "alarm": {
                "sendOnAlarmSet": true,
                "SendOnAlarmCleared": true,
                "periodSec": 1,
                "motionDetection": true
        }
}],
```

FIGURE 6 – Configuration des capteurs

Pour le capteur «HumiTemp», nous définissons une alarme avec deux seuils, l'alarme sera activée si un ou plusieurs seuils sont dépassés. Deux seuils sont définis pour le capteur : 40 °C et -5 °C, pour détecter une température anormale. Le champs «sendOnAlarmSet» est mis à *true* pour déclencher un envoi lorsque l'alarme est activée. Quand le capteur est en mode d'alarme, nous pouvons aussi modifier sa fréquence de prise de mesures à travers les champs «periodSec», «periodMn», «periodHr» ou «periodDay».

En plus d'alarmes, nous pouvons définir des interruptions. Par exemple pour le capteur «Press», nous définissons une interruption de «SHT35_INT» qui sera levée si l'un des seuils d'alarme fixé au capteur interne de température et d'humidité (SHT35) est franchi. «sendOnInterrupt» permet d'envoyer

les données après que le capteur en question ai réalisé une mesure pour cause d'interruption. Notons qu'un envoi contient toutes les données en attentes d'envoi mais pas seulement celles de la mesure sur interruption.

Nous réalisons donc la fonctionnalité suivante : si la température est en-dessous de -5 °C ou au-dessus de 40 °C, des valeurs de température et d'humidité vont être envoyées pour cause d'alarme du capteur «HumiTemp». Ce capteur va prendre des mesures toutes les 5 minutes jusqu'à une valeur de température entre 0 °C et 38 °C. De plus, dès que l'alarme du capteur «HumiTemp» est déclenchée, une interruption de «SHT35_INT» va être déclenchée car le capteur «HumiTemp» est en mode d'alarme. Avec cette interruption, une mesure de la luminosité va aussi être envoyée pour cause d'interruption car nous avons mis l'interruption «SHT35_INT» dans le paramètre «interruptChannel» du capteur «Lumi». Avec ce paramétrage, nous pouvons détecter un événement où le nœud serait brûlé ou congelé.

Pour le capteur «Accelero», nous mettons le champs de la détection de mouvement («motionDetection») en *true*, pour détecter un vol du nœud ou une perturbation éventuelle près du nœud. «sendOnAlarmCleared» est possible pour ce capteur, qui permet de déclencher un envoi lorsque les valeurs suivies repassent sous le seuil d'alarme. Ce paramètre n'est pas disponible pour tous les capteurs, par exemple il n'est pas possible d'avoir cette option pour le capteur SHT35 («HumiTemp»).

2.3 Diagramme UML du *firmware* de nœud SoLo

Nous disposons des codes fonctionnels du *firmware* de nœud. Pour implémenter nos fonctionnalités, nous avons besoins de la classe «ConnecSenS» qui représente le nœud, et la classe abstraite «Sensor» qui représente les capteurs. Les 4 classes qui héritent la classe «Sensor» sont des classes de capteurs internes listés dans la section 2.2.4. Nous n'avons représenté que des fonctions qui nous ont servi à implémenter nos fonctionnalités et une autre fonction «sendAll» que nous avons créée que nous allons présenter dans la partie 2.4.

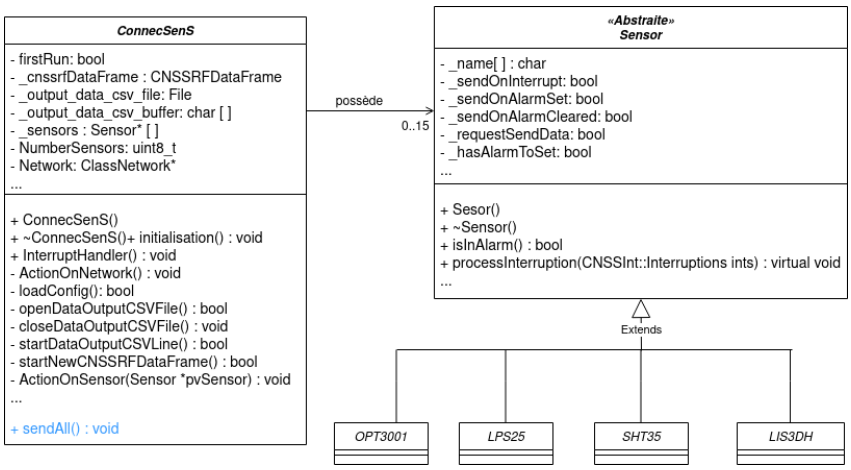

FIGURE 7 – UML de la classe «ConnecSenS» et la classe «sensor»

2.4 Détection de dysfonctionnements

Les capteurs intégrés dans le nœud SoLo peuvent éventuellement tomber en panne, et dans la pratique du projet, nous n'effectuons qu'un envoi toutes les 4 heures. De plus, le nœud redémarrerait éternellement en cas de panne. Afin de pouvoir détecter un dysfonctionnement d'un ou plusieurs capteurs lors du premier démarrage sans attendre une nouvelle période d'envoi et détecter des démarrages anormaux du nœud, nous envisageons d'ajouter une fonctionnalité pour forcer tous les capteurs concernés de prendre une mesure et de l'envoyer vers le *cluster*.

Nous avons créé une fonction «sendAll» dans la classe «ConnecSenS». Dans la fonction «sendAll()», indiquée sur le schéma 7, nous faisons une boucle qui parcourir le tableau de capteurs enregistrés. Pour chaque capteur, nous effectuons une prise de mesure et l'écrivons dans un *buffer*. Une fois que tous les capteurs ont effectuées leurs actions, nous écrivons dans le fichier CSV du dossier «output» les données enregistrées dans le *buffer* pour ensuite les envoyer.

Nous avons ensuite ajouté une condition de vérification du premier démarrage mais pas à chaque réveil du nœud dans la fonction d'initialisation de la classe «ConnecSenS». Si la condition est vérifiée, nous appellerons la fonction «sendAll».

2.5 Nouveau système d'alarmes

Dans le fichier de configuration, nous avons constaté les limites. On ne peut pas définir plusieurs seuils différents en fonction d'événements pour un capteur. Alors que pour des scénarios différents, nous aurions besoin de différentes valeurs de seuil pour le même capteur. De plus, nous ne pouvons pas vérifier une combinaison de contraintes de différents capteurs pour un événement auquel il lui est associé une réaction, car dans le fichier de configuration, nous ne pouvons que définir des seuils par capteur, donc les seuils sont indépendants de l'un à l'autre. Nous envisageons donc d'implémenter un nouveau système d'alarmes qui est plus flexible et qui peut mieux s'adapter à des scénarios de traitement.

Cette fonctionnalité est en phase d'implémentation. Nous allons ajouter dans le fichier de configuration, un nouveau champs «alarms», dédié à ce nouveau système d'alarme. Ensuite, nous créerons une nouvelle classe «new_alarm», représentée sur la figure 8, avec une liste d'objets de la classe «Scenario». Dans la classe «new_alarm», nous allons enregistrer dans la liste «liste_scenarios» des scénarios définis dans le fichier de configuration, et mémoriser son nombre. Cette configuration sera faite par la fonction «SetConfiguration».

La fonction principale est «OnAlarm» : elle vérifie pour chaque capteur dans la liste «conditions» si des mesures ont été prises et si ces mesures dépassent les seuils du capteur correspondant. Si oui, nous allons forcer aux autres capteurs qui sont liés à cet événement de vérifier ses seuils. Pour demander à un capteur de prendre une mesure et comparer la mesure avec le seuil pour un certain événement, nous avons la fonction «VerifySeuil». Si nous détectons un événement, les réactions seront lancées par la fonction «AlarmHandler».

Le type Seuil est défini par une *map* avec «SBornes» en tant que clé et «réel» comme valeur, où «SBornes» est un type d'énumération avec *low* pour représenter la borne inférieure et *high* la borne supérieure. Le diagramme de classe pour *«new_ alarm»* est le suivant :

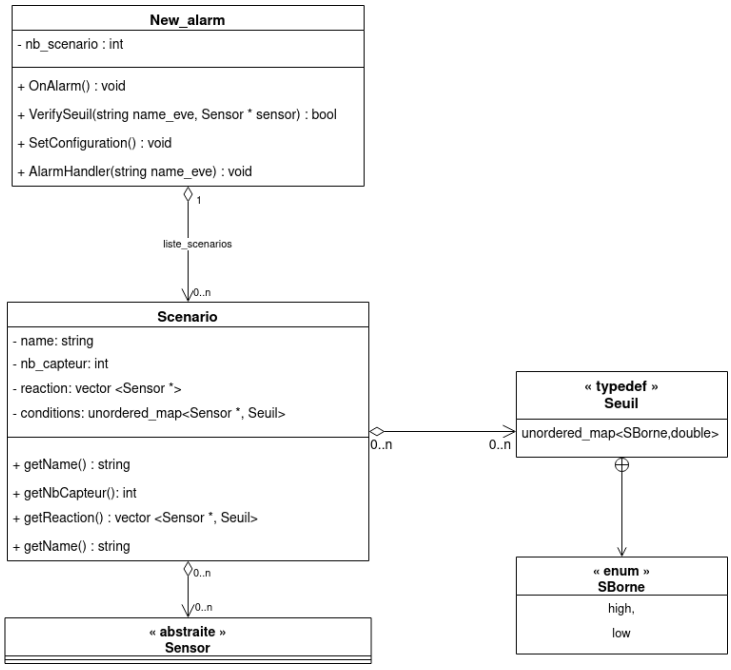

FIGURE 8 – UML de la classe «new_alarm»

3 *cluster* de Raspberry

Dans notre projet, nous disposons d'un *cluster* de cinq Raspberry Pi connectés qui prend le role du serveur. Il est près d'une passerelle LoRaWAN et des nœuds SoLo, pour réduire la consommation énergétique. Les applications sur le *cluster* seront mises en œuvre dans des conteneurs pour permettre une architecture distribuée de microservices. Cependant, avec cinq Raspberry Pi identiques, il faut répéter le processus de configuration pendant la phase de préparation et nous devons automatiser ce processus autant que possible afin d'être efficaces. Dans cette partie, nous nous intéresserons donc la disposition physique des matériaux, la gestion des conteneurs afin que nous puissions gérer la mise à l'échelle des ressources, l'équilibrage de charge et la distribution sur le serveur et comment mettre en œuvre l'automatisation des configurations.

Pour le déploiement d'un réseau de capteurs intelligents sur la zone de la Croix Verte dans le campus de l'université de Rennes 1, nous allons y placer trois noœds SoLo identiques à des endroits différents. La sécurité, la précision des mesures et les limites de distance serons prises en compte lors du choix de la position. Une fois placés, les boitiers resteront actifs et enverront des données à une certaine fréquence ou en cas d'alarme selon la configuration via LoRaWAN à une passerelle LoRaWAN. En utilisant la passerelle comme intermédiaire, le *cluster* recevra les données et les traitera. Les Raspberry Pi communiquent entre eux et avec la passerelle via un réseau Ethernet local accessible et avec l'extérieur par un réseau WiFi. Ils seront donc disposés au bâtiment informatique d'INSA Rennes afin d'utiliser le WiFi «IOTINFO» fourni par la DSI. Ce réseau est isolé et accessible seulement par des appareils autorisés, ce qui empêche tout accès à nos appareils depuis internet.

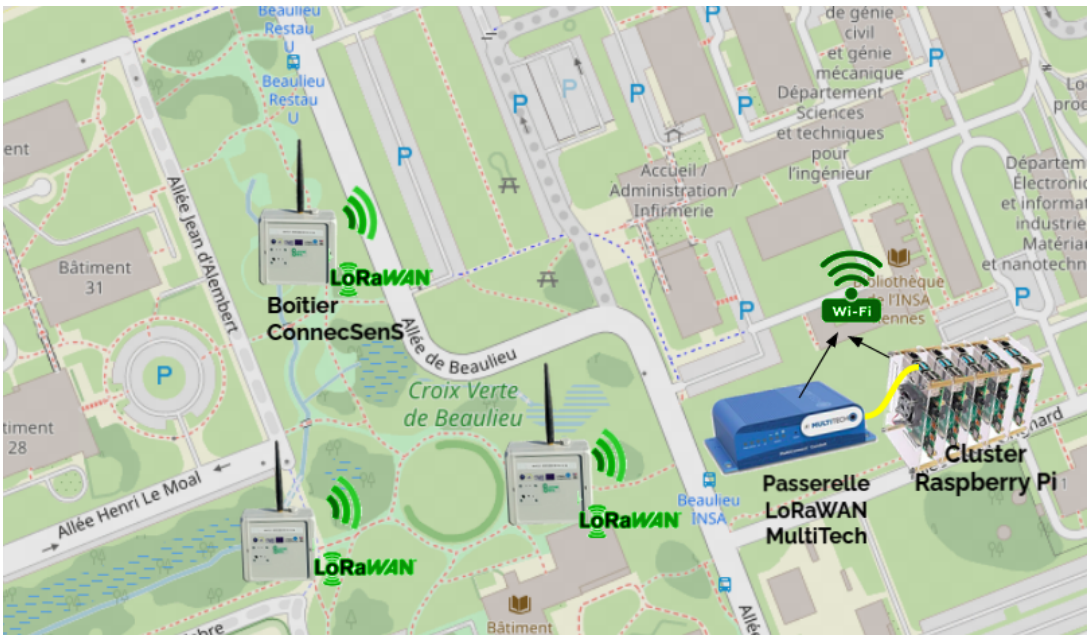

FIGURE 9 – Disposition physique des éléments physique

Cette méthode d'installation présente certaines contraintes pour l'accès au *cluster* à distance : la connexion doit se faire sous le même réseau, c'est-à-dire que nous ne pouvons y intervenir que si nous nous trouvons à la portée d'IOTINFO.

3.1 Kubernetes et GlusterFS

Kubernetes est un système d'orchestration de conteneurs. Pour comprendre le principe de base de Kubernetes, nous commençons par son architecture qui comporte une hiérarchie claire :

— Pod[3] : Un pod est l'unité la plus petite et la plus base dans le modèle d'objets de Kubernetes que les utilisateurs peuvent créer ou déployer. Donc, pour être plus précis, Kubernetes gère les pods qui possèdent des conteneurs plutôt que directement les conteneurs. Un pod encapsule un

conteneur applicatif (ou des conteneurs qui sont étroitement liés et qui partagent des ressources), une adresse IP locale, un ou plusieurs volumes associés à ces containers, c'est-à-dire des ressources de stockage. Ainsi que des options qui contrôlent comment le ou les conteneurs doivent s'exécuter.
- Nœud : Un ou plusieurs pods s'exécutent sur un nœud, dans le projet un Raspberry Pi est un nœud.
- Déploiement[4] : Un déploiement est un objet de type ressource dans Kubernetes, qui fournit des mises à jour déclaratives pour des applications. Un déploiement nous permet de décrire le cycle de vie des pods, en spécifiant par exemple le nombre de pods à exécuter et la façon dont ils doivent être mis à jour. Par conséquent, l'application peut maintenir constamment en vie, même si un pod disparaît si son nœud tombe en panne, redémarre ou est supprimé, car le déploiement met à jour automatiquement le pod, cela assure que l'application soit toujours en état souhaité sans intervention.
- Volume[5] : Dans l'éventualité qu'un pod soit supprimé (suite à une panne par exemple), toutes les données dans ses conteneurs sont supprimées. Les volumes permettent de persister des données en dehors d'un pod, en montant un dossier dans le système de fichier.
- Service[6] : Kubernetes dispose d'un objet appelé Service qui décrit la façon dont un groupe de pods peut être accédé via le réseau, pour que l'utilisateur n'ait pas à se soucier des changements à l'intérieur du nœud.

Mycélium est constitué d'un ensemble de déploiements, mais aussi de volumes, qui pointent vers un *cluster* de stockage distribué GlusterFS[7]. Chaque Raspberry a une copie de chaque volume, ce qui permet aux pods de se déployer sur n'importe quel nœud du *cluster* et de retrouver les données qui leur correspondent. La figure 10 montre les différents logiciels installés sur chacun des nœuds (trois parmis les cinq). Nous utilisons une distribution de Kubernetes appelée K3S. On peut observer que le nœud 0 est le maitre du *cluster*, c'est lui qui décide où déployer les différents pods. GlusterFS est installé sur chaque nœud de travail, et chaque instance communique avec les autres en formant un réseau maillé pour se synchroniser.

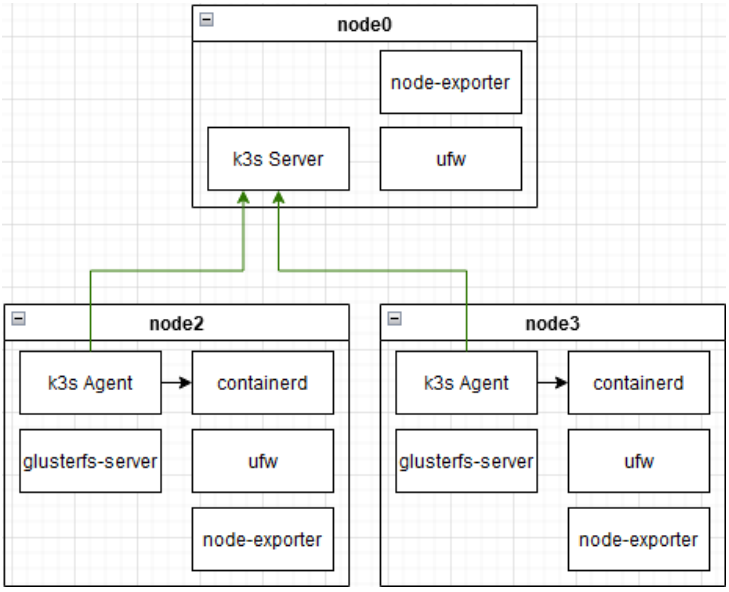

FIGURE 10 – Logiciels installés sur les nœuds du *cluster*

3.2 Déploiement du *cluster* automatisé

L'installation et la configuration des logiciels présentés précédemment sont des tâches fastidieuses et répétitives. Nous les avons automatisées à l'aide d'Ansible[8], un logiciel conçu pour ce genre d'usages qui permet d'exécuter des taches sur un ensemble de machines. Les responsables de l'infrastructure renseignent des paramètres tels que les identifiants du réseau WiFi, et l'outil s'occupe du reste en une commande. Le résultat de l'exécution est un *cluster* prêt pour le déploiement de l'application.

4 Mycélium core

Dans cette partie, nous étudions les fonctionnalités au niveau du *cluster*, avec Kubernetes, comment déployer l'infrastructure de Mycélium sur les Raspberry Pi pour qu'ils puissent recevoir les données passées par la passerelle, les traiter par des fonctions *serverless*, et en plus automatiser le déploiement autant que possible.

4.1 Gestion du réseau LoRaWAN

ChirpStack[9]est un ensemble de logiciel pour la gestion de réseaux et appareils LoRaWAN. Il contient les composants essentiels qui permettent de recevoir les informations passées par la passerelle LoRaWAN et les envoyer à d'autre parties afin de les traiter comme le schéma11 ci-dessous.

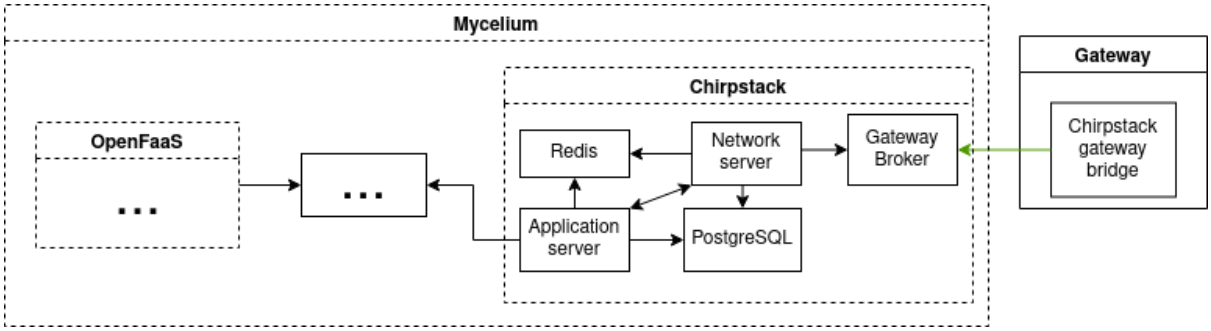

FIGURE 11 – Structure de Mycélium core : ChirpStack

Tout d'abord, nous avons installé *ChirpStack Gateway Bridge[10]* sur la passerelle. C'est un service qui transforme les paquets envoyés par le nœud SoLo en un format de données reconnu par le serveur réseau ChirpStack, et qui les lui envoie via le protocole MQTT[11]. La communication entre ce pont et ChirpStack passe par le réseau Ethernet local.

MQTT est un protocole de messagerie publier-s'abonner (*Publish-Subscribe*) basé sur le protocole TCP/IP, qui est très souvent utilisé dans l'Internet des objets pour sa légèreté. Dans l'architecture MQTT, il existe deux types d'entité : les clients et les courtiers (*Brokers*). Le courtier est le serveur avec qui les clients communiquent. Il reçoit les communications qui émanent des clients et les transmet à d'autres clients, en les séparant par sujet (*Topic*). Ainsi, les clients ne communiquent pas directement entre eux, mais par l'intermédiaire du courtier. Chaque client peut être soit éditeur, soit abonné, ou les deux. MQTT est un protocole orienté événements, cela permet de réduire considérablement la communication non nécessaire. Un client publie uniquement quand il a des informations à transmettre, et un courtier n'envoie des informations aux abonnés que quand il reçoit de nouvelles données.

Après avoir reçu le message publié par la passerelle, le courtier de la passerelle (*Gateway Broker* dans la figure11) va le transmettre à l'abonné restant : *ChirpStack Network Server*. Pour rappel, c'est un serveur réseau qui traite les messages à la couche LoRaWAN MAC. Au niveau de l'utilisateur, c'est l'*Application Server* qui gère l'inventaire des appareils du réseau et offre une interface Web où les utilisateurs peuvent gérer les applications, les organisations etc. Il permet aussi d'intégrer avec des services externes via une *API* gRPC et *RESTful*. Enfin, pour faire fonctionner les deux serveurs, nous avons aussi besoin de Redis et PostgreSQL comme bases de données.

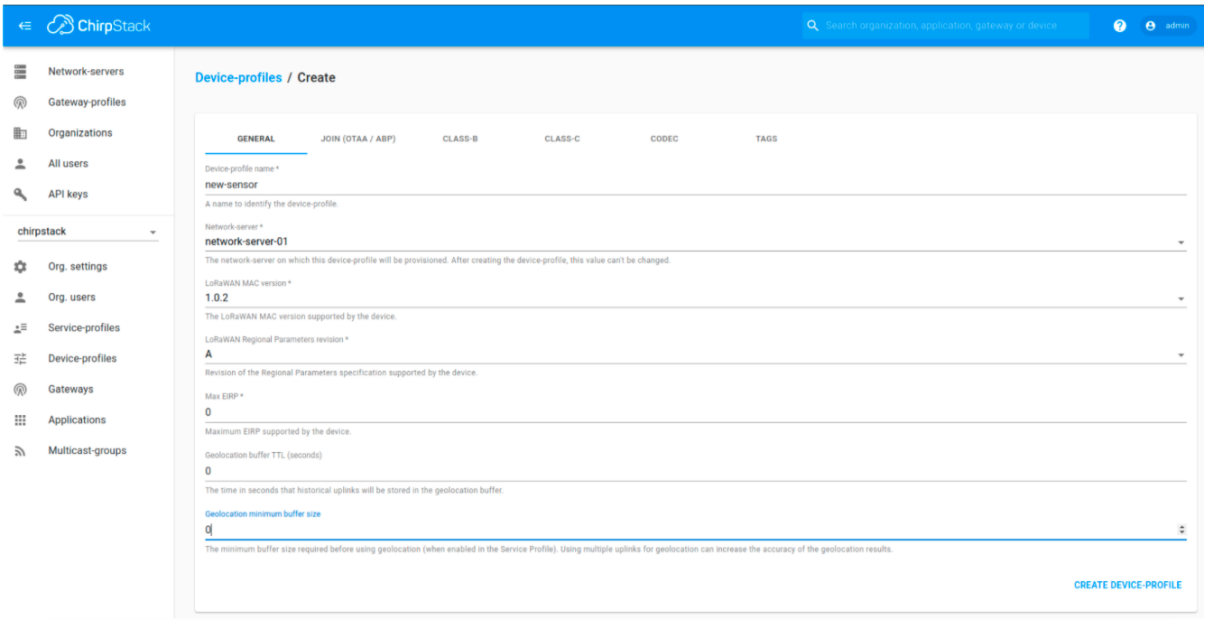

FIGURE 12 – Configuration d'un appareil LoRaWAN

Mais la configuration manuelle de l'inventaire présente plusieurs désavantages : le risque d'erreur humaine, l'absence de traçabilité, et le manque de la gestion des versions. Pour les résoudre, nous utilisons Terraform[12], un outil qui permet de réaliser d'infrastructure en tant que code. Il utilise un fichier d'état pour garder une trace de la relation entre la configuration demandée et l'infrastructure actuelle. Nous avons créé un *Plugin* pour que Terraform puisse interagir avec ChirpStack via son *API* gRPC.

4.2 Traitement *serverless*

Nous utilisons OpenFaaS[13] comme moteur de fonctions *serverless*. Il est composé d'un point d'entrée appelé passerelle OpenFaaS, d'un fournisseur Kubernetes et d'un système de suivi basé sur Prometheus[14]. La passerelle permet de lancer les fonctions via une requête HTTP. Le fournisseur est l'implémentation du moteur, qui déploie un ou plusieurs pod par fonction, selon la demande. Pour déterminer la demande et la charge de chaque pod, le système de suivi collecte des métriques et envoie des alertes à la passerelle, qui commande au fournisseur d'adapter ses déploiements (figure 13).

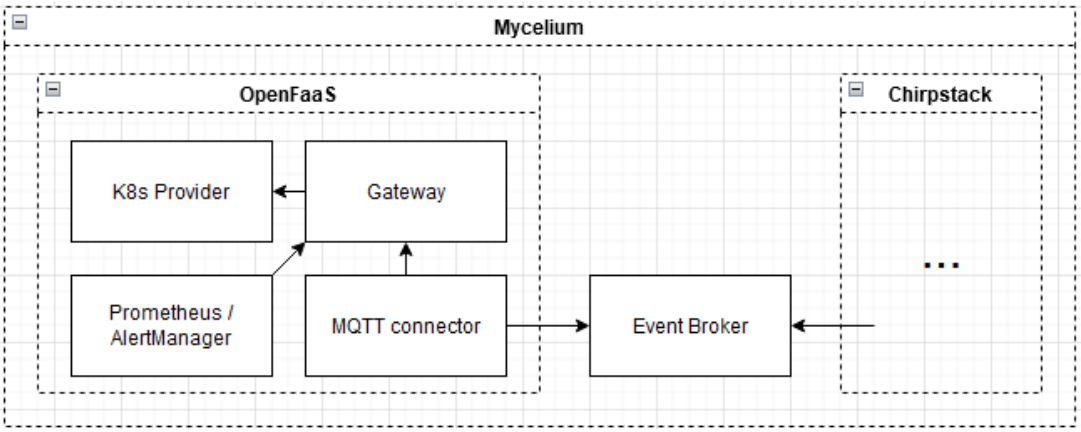

FIGURE 13 – Composants d'OpenFaaS

Le lancement de fonction par HTTP est simple, mais limite les interactions entre fonctions à des appels directs. Dans notre cas, il est intéressant d'ajouter une notion d'évènement pour traiter les données sous

forme de flux. Un autre courtier MQTT, interne à Mycélium Core et appelé *Event broker*, est utilisé pour permettre le lancement de fonctions par évènements. Un service supplémentaire, *MQTT connector*, est ajouté à OpenFaaS pour s'abonner aux sujets du courtiers d'évènements. Les fonctions OpenFaaS peuvent être associées à des évènements, ce qui permet de les lancer via le courtier. Ce système permet de séparer l'arrivée ou la création de nouvelles données de leur traitement, proposant une plus grande flexibilité. Nous verrons dans la partie 5 des cas d'utilisation concrets.

Les fonctions sont en réalité des pods : le code de l'utilisateur est placé dans une image Docker qui sera utilisée lors du déploiement par le fournisseur Kubernetes. Cette image peut être construite automatiquement par un logiciel en ligne de commande, pour ensuite être publiée sur un registre d'images Docker.

4.3 Déploiement de Mycélium Core

Nous avons vu précédemment que Mycélium Core est constitué d'une multitude d'éléments, chacun avec un déploiement Kubernetes associé. Déployer un grand nombre de ressources individuelles sur Kubernetes est une tâche complexe qui laisse la place aux erreurs de manipulation. Pour y remédier le projet Helm[15] propose un système pour regrouper les définitions de ressources Kubernetes, les rendre paramétrables, ainsi que les distribuer, sous la forme d'un paquet : une Chart. Mycélium Core est en réalité une Chart Helm, qui permet de coordonner le déploiement de ChirpStack et d'OpenFaaS, ainsi que les services qui les font communiquer. L'utilisateur de la Chart peut indiquer des paramètres haut niveau, comme le port du courtier de passerelle LoRaWAN. La Chart va aussi configurer certain paramètres de ses composants automatiquement, comme l'association entre le serveur d'application et celui de réseau de ChirpStack. Cette configuration s'effectue grâce à un travail Kubernetes (*Job*), une sorte de déploiement pour un pod qui s'arrête une fois son travail accompli.

Grâce à la Chart, il est possible de déployer Mycélium Core d'une seule commande, pour obtenir une plateforme prête à fonctionner. Toutefois, le déploiement de fonctions *serverless* et la configuration de ChirpStack restent manuels.

4.4 Environnement de développement avec Mycelium Garden

Pour faciliter le développement d'applications avec Mycélium Core, nous avons créé un utilitaire capable de déployer et de configurer la plateforme, les fonctions et les inventaires automatiquement. Il s'appuie sur Garden[18], un moteur de production spécialisé dans les applications en micro services. L'application est découpée en modules, chacun avec leur phase de construction et de déploiement. L'utilitaire, baptisé Mycélium-Garden, permet au développeur de créer un module par fonction *serverless* et par inventaire. Tout le reste du déploiement est abstrait, ce qui simplifie l'utilisation de Mycélium Core.

Mycélium-Garden utilise le système de partage de configuration de Garden et les modèles de modules. Un modèle de module pour les fonctions permet, à partir de quelques paramètres tels que le nom de la fonction, d'abstraire les procédures pour construire et déployer la fonction. Un modèle de module pour les inventaires permet un fonctionnement similaire. La figure 14 présente un projet Mycélium Core avec une fonction *serverless* «Ma fonction». Le groupe «Mon projet» contient un module pour la fonction, ainsi qu'une définition de projet Garden. Cette dernière fait référence à un dépot Git qui contient un autre projet Garden. Tous les modules de ce second projet seront utilisés pour le projet principal. Parmis eux, un module Mycélium Core est responsable du deploiement de Mycélium Core sur un *cluster* Kubernetes, en général local pour du développement. Deux autres modules sont des modèles de module, c'est-à-dire des définitions réutilisables qui ne font rien d'elles-mêmes : elles sont utilisées par les modules du projet principal, comme le module «Ma fonction», qui fait référence à la définition du modèle «fonction».

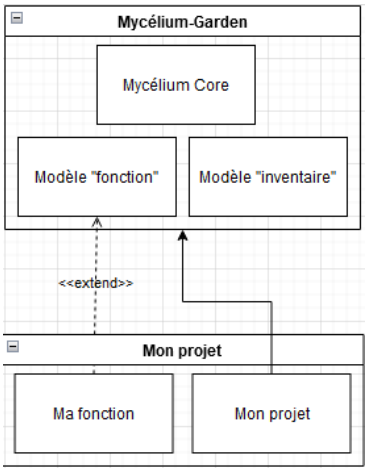

FIGURE 14 – Exemple d'utilisation de Mycélium-Garden

La définition du modèle «inventaire» contient le code nécessaire à l'application d'un inventaire Terraform, ainsi que la sauvegarde du fichier d'état dans un volume Kubernetes. Mycélium-Garden permet aussi de debugger facilement les fonctions et services déployés en configurant des redirections de ports. Par exemple, OpenFaaS, qui est normalement inaccessible depuis l'extérieur du *cluster*, est disponible sur le port local 8081 pour pouvoir lancer manuellement des fonctions. Les journaux d'exécution sont aussi facilement accessible via une interface web. Enfin, le code des fonctions déployées est synchronisé avec le code du développeur, pour éviter d'avoir à redéployer au moindre de changement.

4.5 Déploiement continu

Mycélium-Garden peut aussi être utilisé pour déployer en production. Toutefois, cela nécessite l'accès au réseau IOTINFO, ce qui peut se révéler problématique en cas de confinement. La DSI a ouvert l'accès au maître du *cluster* au GitLab de l'INSA, ce qui nous permet d'exécuter des *pipeline CI/CD* via un exécuteur GitLab installé sur le *cluster*.

Mycélium Core, Mycélium-Garden et Mycélium sont trois projets GitLab séparés. En plus des fonctions *serverless* et des inventaires Terraform, accompagnés de leur module Garden, le projet Mycélium sur GitLab est doté d'un registre d'images Docker, qui stocke les versions construites des modules. Ces images sont téléchargées par les agents Kubernetes lors de la création des pods. Le déploiement est coupé en deux, pour d'abord construire les images en dehors du *cluster* (pour limiter la charge de ce dernier), sur les exécuteurs GitLab de l'INSA (INSA BOO). Ensuite, l'exécuteur présent sur le *cluster* va mettre à jour les déploiement Kubernetes pour utiliser la nouvelle version de chaque image. Ce processus est illustré par la figure 15.

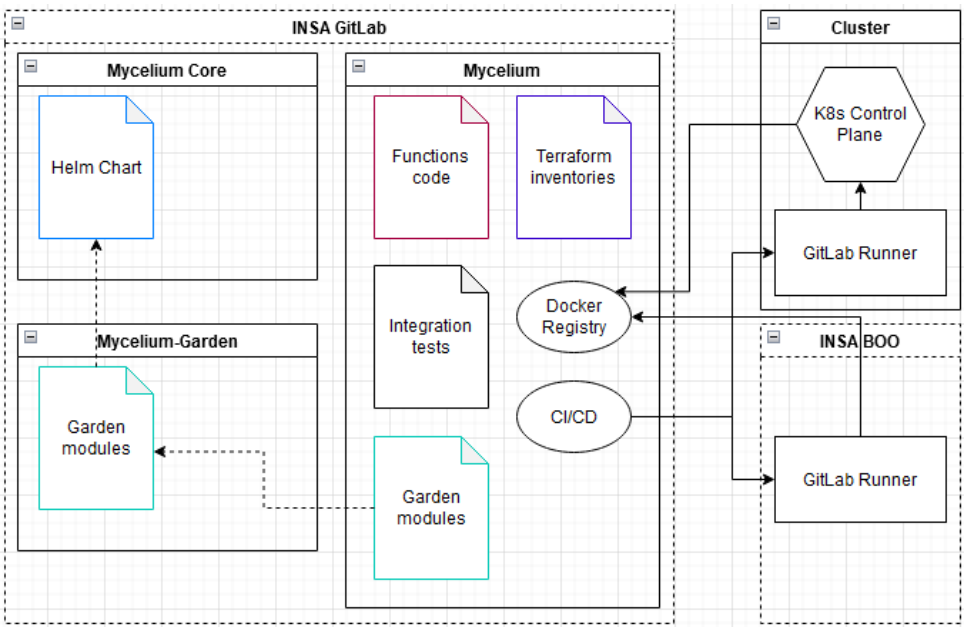

FIGURE 15 – Déploiement continu de Mycélium

Une fois le projet déployé, les données collectées sont accessibles via Grafana[17], un logiciel de visualisation des données. Comme pour le reste du *cluster*, ce service n'est accessible que sur le réseau IOTINFO. Nous utilisons donc l'intégration de Grafana dans GitLab pour afficher des graphiques directement dans un quatrième projet GitLab : Croix Verte Monitoring. Nous pouvons donner accès à ce projet aux scientifiques de Rennes Géosciences pour qu'iels puissent les utiliser.

5 Fonctions appliquées au suivi environnemental de la Croix Verte

Ce sont des fonctions OpenFaaS, c'est-à-dire *serverless*. Elles sont donc toutes composées d'un handler en python contenant le code associé. Elles ont pour but d'effectuer des traitements sur les données.

5.1 Fonctions de décodage

Ces fonctions communes à tous les scénarios. Elles permettent l'envoi des données de ChirpStack à InfluxDB[16]

- *chirpstack-handler* : Fonction qui récupère les données de ChirpStack. Cette fonction se déclenche automatiquement, dès que des mesures provenent du nœud SoLo arrivent dans l'infrastructure Mycélium Core, plus précisément dans ChirpStack.
- *cnssrf-decoder* : Fonction qui permet de décoder, puis d'encoder les données au format JSON
- *frame-recorder* : Fonction qui parcours les données au format JSON et qui envoie les informations à InfluxDB pour les stocker

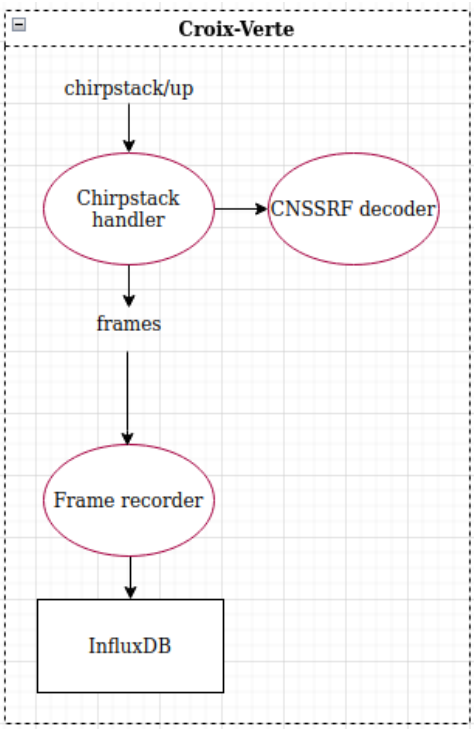

FIGURE 16 – Diagramme des fonctions permettant le décodage des données

5.2 Rappels sur les scénarios

Pour rappel, dans des conditions optimales où aucun scénario ne se déclenche, les données sont envoyées six fois par jour au *cluster*, c'est-à-dire toutes les 4h.

Cette fréquence d'envoi permet d'alléger la quantité de données transférées via LoraWAN. Cela permet aussi de faire en sorte que le *cluster* de Raspberry Pi n'est pas de charge de travail trop importante à réaliser avec les fonctions de traitement.

Mais la fréquence d'envoi est susceptible d'augmenter si le système d'alarmes au niveau du nœud SoLo se déclenche. Des données liées aux capteurs concernés par l'alerte peuvent donc arriver n'importe quand.

Ces envois, qu'ils soient prévus ou non, vont déclencher des fonctions de traitement sur les données. Ainsi, selon les résultats obtenus, des scénarios sont susceptibles d'être déclenchés, c'est-à-dire des fonctions spécifiques associées à un événement climatique spécifique.

Parmi les différents scénarios établis dans le rapport de spécification, on en sélectionne deux qui paraissent pertinents pour décrire aux mieux les différentes fonctions qui leur seront associées :

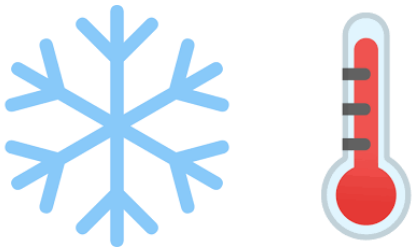

FIGURE 17 – Scénarios «Neige» et «Températures anormales»

— Le scénario «Neige» : Dès que la température est négative et que l'humidité dépasse un certain seuil, une alerte au niveau du capteur HumiTemp est déclenchée. Le nœud SoLo envoie donc immédiatement les données d'humidité et de température relevées pour pouvoir les analyser rapi-

dement. On va ensuite comparer les données météorologiques reçues avec celles récoltées en temps réel par la station météo la plus proche de la Croix Verte.
— Le scénario «Températures anormales» : Les données de températures sont considérées comme anormales si un trop grand écart survient entre deux températures, si elles dépassent un certain seuil (minimum ou maximum) fixé par le nœud SoLo, ou si elles ne correspondent pas aux données météo de Rennes.

Les autres scénarios fonctionneront de manière similaire, et pourront faire appel à des fonctions communes à tous les scénarios dans certains cas.

5.3 Fonctions caractéristiques au scénario «Neige»

Les fonction qui déclenchent le scénario «Neige» sur les données dans InfluxDB s'exécutent lorsque qu'une alerte du nœud SoLo est déclenchée au niveau du capteur HumiTemp. En effet, si la température est négative et que l'humidité est élevée, les données sont immédiatement envoyées au *cluster* pour un traitement. Les principales fonctions déclenchées suite à ces envois de données exceptionnels sont :

— *compare-temperature* : une fonction de comparaison de la température reçue du nœud SoLo, avec les températures recueillies sur internet en temps réel. Elle renvoie true si les valeurs sont les mêmes (à 2°C près).
— *compare-humidity* : une fonction de comparaison de l'humidité reçue du nœud SoLo, avec l'humidité sur Rennes recueillie sur internet en temps réel. Elle renvoie true si les valeurs sont les mêmes (à 2°C près). Cette fonction est déclenchée lorsque des données provenant du capteur HumiTemp du nœud SoLo sont collectées
— *snow-processing* : une fonction déclenchant le traitement propre au scénario. Elle s'exécute si *compare-temperature* ET *compare-humidity* renvoient true.

Les valeurs-seuil de comparaison fixées dans ce scénario au niveau du nœud SoLo sont les suivantes :

Seuil d'humidité déclenchant l'alerte du capteur HumiTemp	88%
Seuil de température déclenchant l'alerte du capteur HumiTemp	0°C

Une fois que le scénario «Neige» est reconnu, on déclenche la fonction de traitement *send-message*. Elle permet d'envoyer un message d'alerte sur Gitlab et Discord.

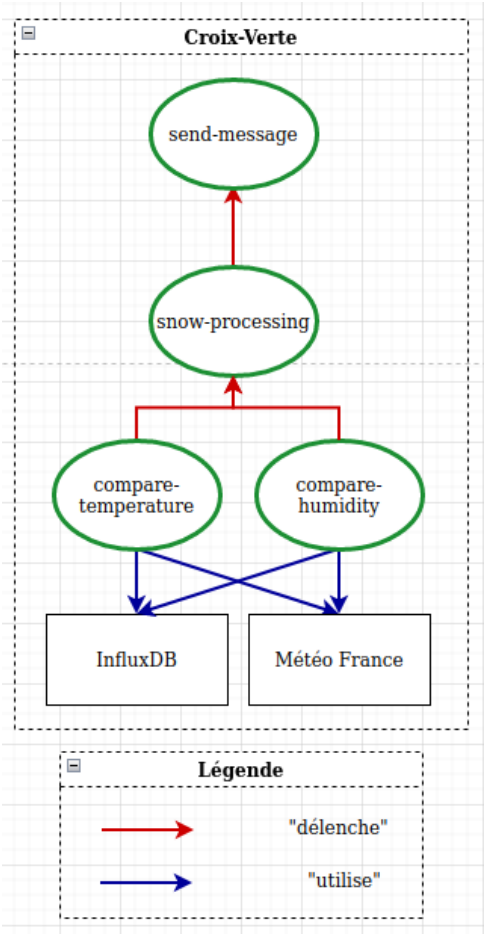

FIGURE 18 – Diagramme des fonctions utilisées pour le scénario «Neige»

5.4 Fonctions caractéristiques au scénario «Températures anormales»

Les fonction qui déclenchent le scénario «Températures anormales» sur les données dans InfluxDB s'exécutent toutes les 4h, au moment où les données sont reçues. Elles peuvent aussi s'exécuter dans le cas où l'on reçoit une alerte du nœud SoLo au niveau du capteur HumiTemp.

Voici ci-dessous les fonctions déclenchées à l'arrivée de n'importe quelle donnée (qu'elle soit prévue ou qu'elle provienne d'une alerte :

— *is-alerte* renvoie true si elle provient d'une alerte. Elle peut être utilisée par d'autres scénarios.

La fonction *is-alerte* déclenche ces fonctions

— *compare-temperature* : une fonction de comparaison de la température reçue du nœud SoLo, avec la température sur Rennes en temps réel recueillie sur internet. Elle renvoie true si les valeurs sont les mêmes (à 2°C près). C'est la même fonction qui sert à déclencher le scénario «Neige».

— *temperature-difference* : une fonction qui permet de comparer deux données consécutives dans InfluxDB. Sachant que les données sont relevées toutes les heures, si un écart de 5°C ou plus est trouvé entre deux températures, cela n'est pas cohérent. La fonction renvoie donc true dans ces cas-là.

— *temperature-processing* : une fonction déclenchant le traitement propre au scénario. Elle s'exécute si *compare-temperature* OU *temperature-difference* renvoient true.

Parfois, on n peut pas se permettre d'attendre l'envoi de données pour effectuer un traitement. En effet, quand le matériel est soumis à un danger immédiat parce qu'il est soumis à des températures extrêmes, il faut réagir vite. Voici ci-dessous les fonctions déclenchées lors d'une alerte envoyée par le botier :

— *temperature-high* : une fonction se déclenchant si le nœud SoLo envoie une alerte comme quoi la

température est trop élevée.

— *temperature-low* : une fonction se déclenchant si le nœud SoLo envoie une alerte comme quoi la température est trop basse.

Les valeurs-seuil de comparaison fixées dans ce scénario au niveau du nœud SoLo sont les suivantes :

Seuil de température haute déclenchant l'alerte du capteur HumiTemp	45°C
Seuil de température basse déclenchant l'alerte du capteur HumiTemp	-5°C

Une fois que le scénario «Températures anormales» est reconnu, on déclenche la fonction de traitement *send-message*, comme pour le scénario précédent. La nature du message est différente selon la fonction qui a déclenché *temperature-processing*.

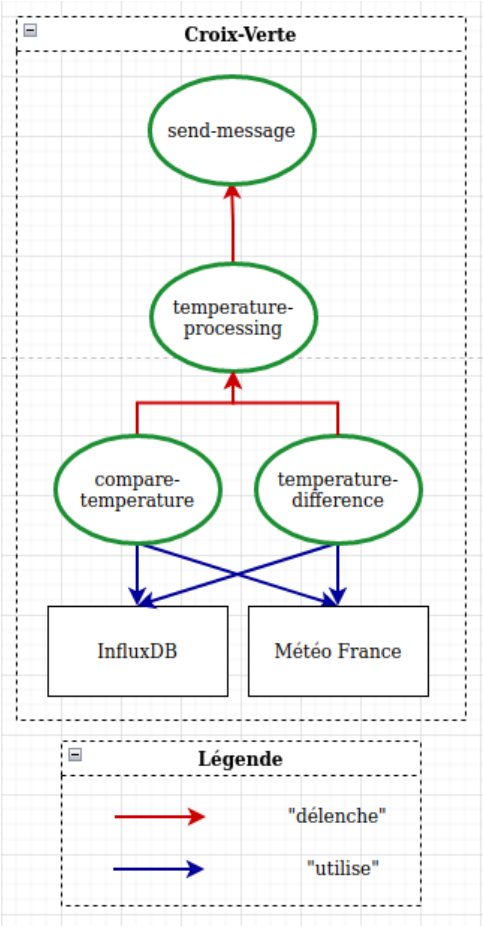

FIGURE 19 – Diagramme des fonctions utilisées pour le scénario «Températures anormales»

5.5 Récupération des données météorologiques depuis sur internet

On a besoin de données météorologiques afin que nos données puissent être comparées à d'autres données météorologiques récoltées sur Rennes. Pour cela, on va utiliser l'API de Météo France[**?**] : on pourra ainsi comparer une donnée reçue avec les données de Météo France[19] en temps réel.

Les données météo qui nous intéressent proviennent de la station Rennes-Saint-Jacques[21]. Située à une vingtaine de kilomètres au Sud-Ouest de la Croix Verte, c'est la station météo la plus proche.

De plus, Météo France met à disposition MeteoNet[20], un site sur lequel on peut télécharger un jeu de données météorologiques. En effet, des mesures provenant de deux zones géographiques (Nord-ouest et Sud-Est de la France) de 2016 à 2018, ont été récoltées et stockées dans des fichiers CSV. On pourra donc stocker ces données dans InfluxDB. Elles peuvent s'avérer utile pour avoir des données-types selon une époque de l'année donnée.

6 Conclusion

Pour conclure, ce rapport donne un aperçu de la conception du projet Mycélium. Nous avons expliqué ses fonctionnalités en divisant ce document en quatre parties.

Tout d'abord nous avons présenté les fonctionnalités du nœud SoLo et ses interactions avec les autres acteurs, les paramétrages dans le fichier de configuration, le diagramme de classe du *firmware* de nœud, les améliorations apportés à propos de la détection de dysfonctionnements au premier démarrage ainsi que le nouveau système d'alarme. Tous ces points ont déjà été réalisés à la date du rendu de ce rapport, excepté le nouveau système d'alarmes qui est en cours d'implémentation.

Ensuite, nous avons étudié le fonctionnement du *cluster*, ainsi que la structure et les fonctionnalités avancées de Mycélium Core. Cette partie du projet est à l'heure actuelle opérationnelle.

Enfin, nous présentons les fonctions de traitements associées aux scénarios, c'est-à-dire leurs rôles et comment elles communiquent entre elles. Celles-ci restent à implémenter pour la majorité d'entre elles au niveau des scénarios, mais les fonctions de décodage existent et fonctionne déjà.

Ce rapport permet donc de visualiser le trajet des données, depuis leur création dans les capteurs du nœud SoLo, jusqu'à leur stockage et leur traitement. Nous avons donc une architecture en micro-service pour notre système afin de contrôler notre réseau de capteurs et développer des fonctions de traitement.

Références

[1] LoRaWAN : lien de la documentation du LoRaWAN

[2] Acquittement : lien de la documentation de l'ACK

[3] Qu'est-ce qu'un pod ? : lien de la documentation de Pod

[4] Qu'est-ce qu'un déploiement dans Kubernetes ? : lien de la documentation de Déploiement

[5] Stockage dans Kubernetes : lien de la documentation de Volume

[6] Qu'est-ce que la service dans Kubernetes ? : lien de la documentation de Service

[7] GlusterFS : lien de la documentation de GlusterFS

[8] Ansible : site web d'Ansible

[9] ChirpStack : lien de la documentation de ChirpStack

[10] ChirpStack Gateway Bridge : introduction de ChirpStack Gateway Bridge

[11] MQTT : introduction de MQTT

[12] Terraform : lien de la documentation de Terraform

[13] OpenFaaS : lien du site

[14] Prometheus : introduction de Prometheus

[15] Helm : site web de Helm

[16] InfluxDB : Nous utilisons la version 1.8 d'InfluxDB qui n'a pas d'interface graphique. lien du site

[17] Grafana : lien du site

[18] Garden : lien de la documentation de Garden

[19] Météo France : lien du site

[20] MeteoNet : lien pour télécharger la base de données

[21] Station Météo Rennes-Saint-Jacques : lien du site de la station