Projet IoT

Jeremy Favre - Steven Liatti

Janvier 2020

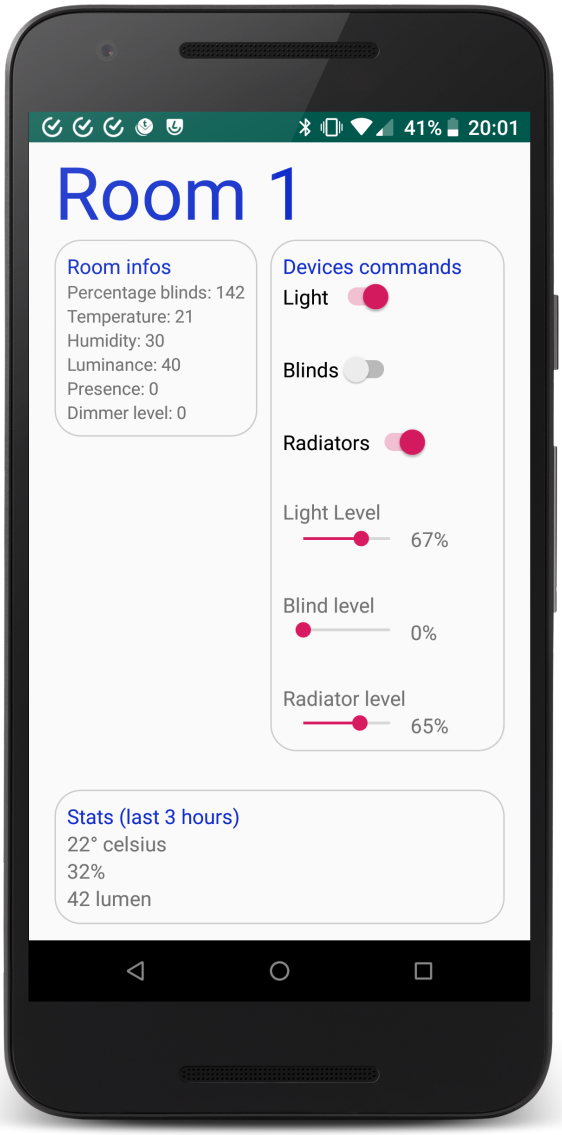

Table des matières

Table des figures

Table des listings de code source

Conventions typographiques

Lors de la rédaction de ce document, les conventions typographiques ci-dessous ont été adoptées.

- Tous les mots empruntés à la langue anglaise ou latine ont été écrits en *italique*.
- Toute référence à un nom de fichier (ou répertoire), un chemin d'accès, une utilisation de paramètre, variable, commande utilisable par l'utilisateur, ou extrait de code source est écrite avec une police d'écriture à `chasse fixe`.
- Tout extrait de fichier ou de code est écrit selon le format suivant :

```
fun main() {
    println("Hello World!")
}
```

Acronymes

API *Application Programming Interface*, Interface de programmation : services offerts par un programme producteur à d'autres programmes consommateurs. 6, 8

JSON *JavaScript Object Notation*, Format d'échange de données léger, facile à lire et écrire par les humains et les machines. 12

REST *representational state transfer*, style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web. 5–8

URI *Uniform Resource Identifier*, Identifiant uniforme de ressource, une courte chaîne de caractères identifiant une ressource sur un réseau. 8

1 Introduction

La forme "nous" est utilisée tout au long de ce rapport étant donné que ce projet est réalisé par binôme.

1.1 Buts

Le but de ce projet est de mettre en pratique les différentes librairies et technologies étudiées en cours d'internet des objets afin de réaliser un système de gestion de bâtiment intelligent qui offre des fonctionnalités de haut niveau (fonctionnalités pour l'utilisateur final) telles que :

— Abaisser la température d'une pièce à un seuil donné lorsqu'elle est vide

— Augmenter la température d'une pièce à un seuil donné lorsqu'elle est occupée

— Fermer les stores lorsque l'humidité est élevée

— Ouvrir les stores le jour, lorsque la luminosité est faible et que la pièce est occupée

— Afficher l'état d'un store et / ou d'un radiateur donné

— Surveiller manuellement les stores et les radiateurs de la pièce où se trouve l'utilisateur

— Fournir des statistiques

L'utilisateur intéragit avec le système à l'aide d'une application Android. Ces fonctionnalités peuvent être activées automatiquement ou manuellement.

1.2 Motivations

Le périmètre de ce projet semblait parfaitement adapté pour exploiter la majorité des techniques et méthodes vues durant le cours d'IoT. Ce projet est une formidable occasion pour relier la mise en pratique des connaissances acquises en cours et notre passion pour le développement d'un système complet multi couches (dit *full stack*).

1.3 Méthodologie de travail

Sur la base d'une analyse préliminaire, nous avons séparé le travail en plusieurs tâches que nous avons assigné à chaque membre du binôme de manière équitable afin d'effectuer le travail en parallèle. Nous avons adopté une pseudo méthode "agile", en factorisant le projet en petites tâches distinctes et en nous fixant des délais pour les réaliser. Le partage du code s'est fait avec git et gitlab. Nous nous sommes servis des *issues* gitlab pour représenter nos tâches et du "*board*" du projet pour avoir une vision globale du travail accompli (par qui et quand) et du travail restant.

2 Conception et analyse

Notre système, visible à la figure 1, s'architecture autour du *message broker* Kafka [1]. Les contrôleurs KNX et Openzwave gèrent respectivement les stores et les radiateurs ainsi que les lumières et capteurs multi-fonctions (présence, luminance, température, humidité). Ils peuvent d'une part recueillir les informations des *devices* associés, mais également attribuer des nouvelles valeurs pour chacuns. Un utilisateur muni de l'application mobile et se trouvant à portée d'un Beacon peut intéragir avec les *devices* de la pièce (*room*) associée avec le Beacon. Ces interactions comportent la visualisation de l'état actuel des *devices* (présence, niveau de température, d'humidité et de lumière, ouverture des stores et des radiateurs) mais aussi leur contrôle (lumière, stores et radiateurs). L'application échange avec un serveur HTTP REST qui fait office de *backend*, lisant dans la base de données les relations entre Beacons, pièces, *devices* et utilisateurs. Un module d'authentification (non implémenté) vérifie qu'un utilisateur dans une pièce donnée a le droit d'utiliser les *devices* associés. La base de données garde également en mémoire les données des *devices*, pour des éventuelles statistiques. Un module indépendant lit la base de données pour produire dans Kafka la liste des *devices* pour les mettre à disposition des modules KNX et Openzwave. Le module "Automatic Controller" exécute des actions de manière automatique et intelligente, selon certaines conditions, par exemple si quelqu'un se trouve dans une pièce alors qu'il fait sombre, la lumière s'allume.

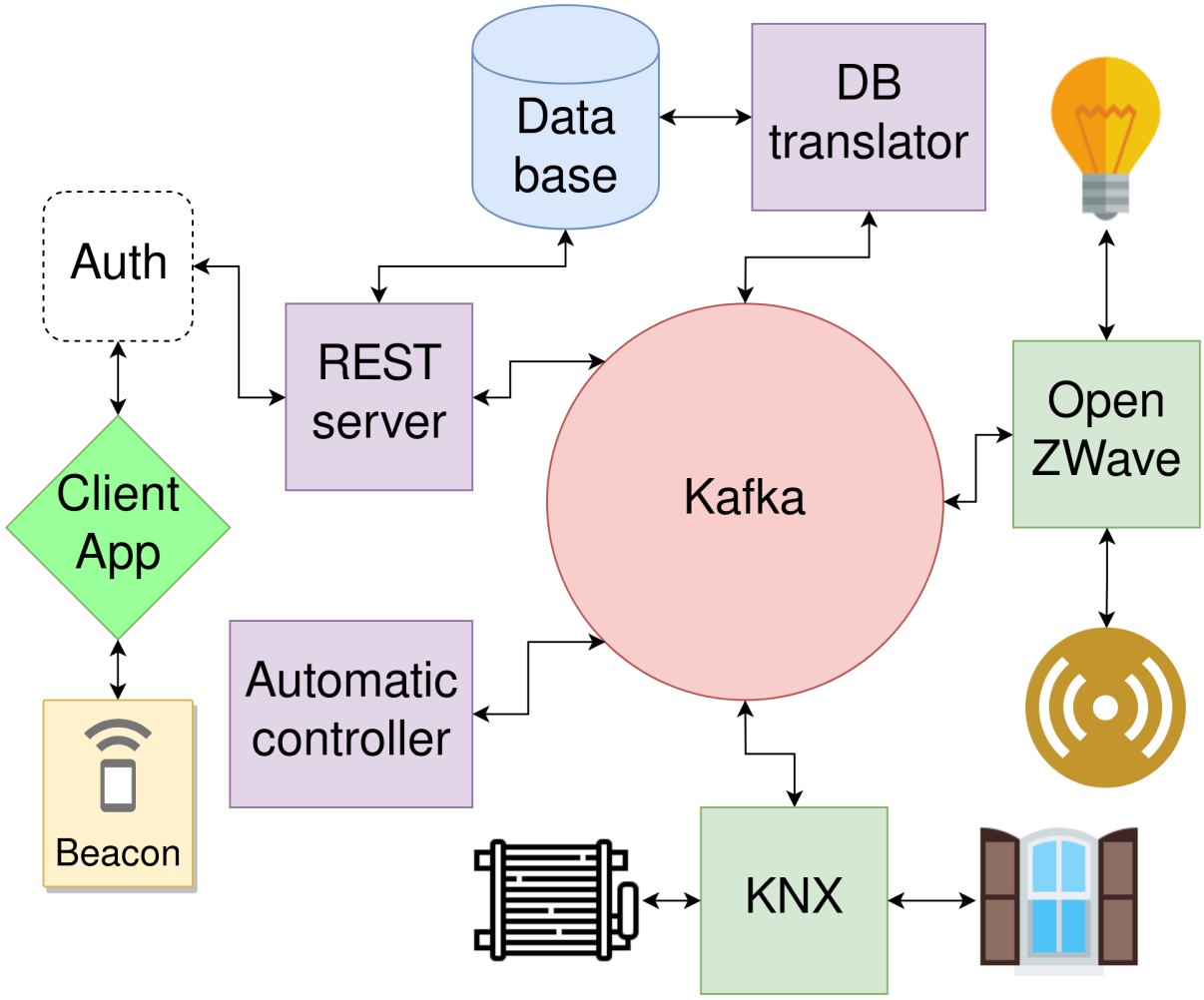

FIGURE 1 – Architecture globale du système

3 Kafka

La technologie d'échange des messages était au choix, nous avons choisi d'expérimenter Kafka [1], étant donné qu'elle est de plus en plus répandue dans différents projets IoT, il était donc intéressant de la prendre en main au travers de ce projet.

3.1 Généralités

Apache Kafka est une plate-forme de diffusion d'événements distribuée open source, tolérante aux pannes, développée initialement par LinkedIn puis reprise par l'*Apache Software Foundation*, écrite en Scala. Service de journalisation distribué, Kafka est utilisé en tant que *hub* de messagerie haute performance, en raison de son débit, de son évolutivité, de sa fiabilité et de son mécanisme de réplication. Les messages Kafka peuvent être étiquetés et publiés/consommés dans un "sujet" (ou "*topic*"). Chaque message est constitué d'une clé, d'une valeur et d'un *timestamp*. Étant donné que Kafka est un système distribué, les rubriques peuvent être partitionnées et répliquées sur plusieurs noeuds. Les fonctionnalités phare de Kafka sont :

— Publier et s'abonner à des flux de messages, similaires à une file d'attente de messages ou à un système de messagerie d'entreprise.

— Stocker les flux de messages d'une manière durable et tolérante aux pannes.

— Traiter les flux de messages au fur et à mesure qu'ils se produisent.

Kafka offre quatre APIs principales :

— L'API *Producer* permet à une application de publier un flux de messages sur un ou plusieurs *topics* Kafka.

— L'API *Consumer* permet à une application de s'abonner à un ou plusieurs *topics* et de traiter le flux de messages produit.

— L'API *Streams* consommant un flux de messages à partir d'un ou plusieurs *topics*, en leur appliquant un traitement, puis les reproduisant dans un ou plusieurs *topics*.

— L'API *Connector* permet de construire des producteurs ou consommateurs Kafka réutilisables et connectés à des applications existantes. Par exemple un connecteur à une base de données qui capture tout changement dans celle-ci.

3.2 Usage dans le projet

Dans le cadre de ce projet, nous avons utilisé Kafka afin de communiquer entre les différentes entités de l'application. Chacunes d'entre elles (KNX, OpenZWave, API REST Flask, DB translator et Automatic Controller) écoute et produit différents messages dans Kafka afin de réagir à certaines actions provenant du client et également de produire des informations à intervalles réguliers, sans dépendre de la demande des client. Au sein de ce projet, nous faisons usage de trois topics :

1. `knx` : concerne toutes les commandes KNX.
2. `zwave` : concerne toutes les commandes OpenZWave.
3. `db` : pour la production de la liste des *devices* présents en base de données.

4 Implémentation

4.1 Application client

Le client développé est une application Android. Nous avons choisi d'utiliser le langage Kotlin qui est plus performant et optimise les opérations et la structure par rapport à Java. C'est également le langage que nous utilisons au sein du cours d'Android, c'est donc un bon moyen de mettre en relation les deux cours. Le rôle principal de ce client est de détecter (grâce au bluetooth) le Beacon le plus proche. En fonction de ce Beacon, le serveur Flask REST est interrogé pour déterminer (via la base de données) dans quelle salle se trouve la personne et les *devices* présents (stores, radiateurs, lumières, etc.). Une fois l'emplacement déterminé et les *devices* détectés, les contrôles et les informations des différents *devices* apparaissent sur l'interface graphique de l'utilisateur, lui permettant de contrôler ou obtenir les différentes informations actuelles et passées des *devices* (sur les trois dernières heures) à proximité. La figure 2 illustre le résultat obtenu.

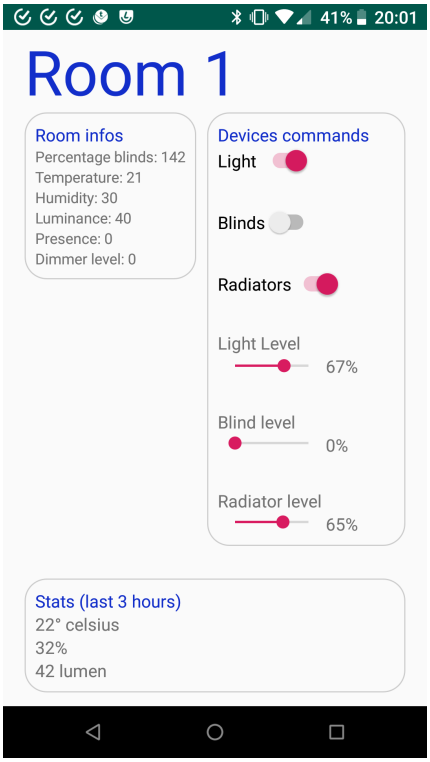

FIGURE 2 – Application Android

4.2 Broker Kafka

«««< HEAD Le broker Kafka joue le rôle de coordinateur entre les différents clients qui consomment et produisent des messages dans les différents topics. Nous avons choisi de déployer ce broker Kafka sur une instance AWS [2] avec Docker [3] et docker-compose [4], le rendant accessible par toutes les entités du système. ======= Le broker Kafka joue le rôle de coordinateur entre tous les clients qui consomment et produisent des messages dans les différents topics. Nous avons choisi de déployer ce broker Kafka sur une instance AWS [2] avec Docker [3] et docker-compose [4], le rendant accessible par toutes les entités du système. »»»> bd590d95568292c3e7eaa012b6beab3759a83597 Nous avons également associé un nom de domaine à cette instance, ce qui permet de la référencer de manière plus lisible et agréable par les clients. Nous avons fait usage des images docker de Confluent [5] pour Kafka [6] et son Zookeeper [7]. Elles intègrent un broker Kafka configurable. Le listing 1 montre un exemple d'utilisation de ces images dans un `docker-compose.yml`. Dans

cet exemple, le broker Kafka utilisé est disponible avec l'URI `iot.liatti.ch` sur le port 29092.

```yaml
version: '3'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
      PLAINTEXT://kafka:9092,PLAINTEXT_HOST://iot.liatti.ch:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
      PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Listing 1 – Utilisation des images Docker de Confluent pour Kafka

4.3 REST server Flask

Etant donné qu'il n'existe pas encore de librairie permettant d'utiliser directement le client Kafka sur Android, nous avons du mettre en place un adapteur entre Kafka et Android. Pour ce faire, nous avons utilisé la librairie Flask [8] de Python 3 qui permet fournit une API REST. Ce serveur est connecté à la base de données et au broker Kafka. Le producteur Kafka se charge de transformer les actions reçues sous forme de requêtes HTTP en messages Kafka envoyés directement dans le bon topic, ce qui permet d'effectuer les actions demandées en conséquence (ouverture des stores par exemple). Pour récupérer l'état actuel des *devices* ou de la pièce, les derniers logs sont récupérés avec des requêtes à la base de données. Le serveur est déployé dans un container Docker avec docker-compose.

4.3.1 Routes exposées

Voici la liste exhaustive des différentes routes mise à disposition par notre API REST :

KNX

- `Endpoint: "/open_blinds" - Method: GET`
  - Params :
    - uuid : string
    - major : int

        - minor : int
    - Response :
        - OK : { "success": true }
        - Error : { "success": false }
- Endpoint: "/close_blinds" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true }
        - Error : { "success": false }
- Endpoint: "/percentage_blinds" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
        - percentage : int
    - Response :
        - OK : { "success": true }
        - Error : { "success": false }
- Endpoint: "/percentage_radiator" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
        - percentage : int
    - Response :
        - OK : { "success": true }
        - Error : { "success": false }

Infos

- Endpoint: "/read_percentage_blinds" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "percentage": 42 }
        - Error : { "success": false }

OpenZWave

- Endpoint: `"/percentage_dimmers"` - Method: GET
  - Params :
    - uuid : string
    - major : int
    - minor : int
    - percentage : int
  - Response :
    - OK : `{ "success": true }`
    - Error : `{ "success": false }`

Infos

- Endpoint: `"/sensor_get_temperature"` - Method: GET
  - Params :
    - uuid : string
    - major : int
    - minor : int
  - Response :
    - OK : `{ "success": true, "value": 23 }`
    - Error : `{ "success": false }`
- Endpoint: `"/sensor_get_humidity"` - Method: GET
  - Params :
    - uuid : string
    - major : int
    - minor : int
  - Response :
    - OK : `{ "success": true, "value": 23 }`
    - Error : `{ "success": false }`
- Endpoint: `"/sensor_get_luminance"` - Method: GET
  - Params :
    - uuid : string
    - major : int
    - minor : int
  - Response :
    - OK : `{ "success": true, "value": 23 }`
    - Error : `{ "success": false }`
- Endpoint: `"/sensor_get_motion"` - Method: GET
  - Params :
    - uuid : string

        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "value": 23 }
        - Error : { "success": false }
- Endpoint: "/dimmer_get_level" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "value": 23 }
        - Error : { "success": false }

Autres routes

- Endpoint: "/get_beacons" - Method: GET
    - Response :
        - OK : { "success": true, "beacons": 23 }
        - Error : { "success": false }
- Endpoint: "/get_devices" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "devices": 23 }
        - Error : { "success": false }

Stats

- Endpoint: "/avg_temperature" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "avg_temperature": 23 }
        - Error : { "success": false }
- Endpoint: "/avg_humidity" - Method: GET
    - Params :

        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "avg_humidity": 23 }
        - Error : { "success": false }
- Endpoint: "/avg_luminance" - Method: GET
    - Params :
        - uuid : string
        - major : int
        - minor : int
    - Response :
        - OK : { "success": true, "avg_luminance": 23 }
        - Error : { "success": false }

4.4 Modules KNX et Openzwave

En se basant sur le code des exercices KNX et Openzwave réalisés en cours, nous avons créé deux modules reliant d'une part Kafka et KNX et d'autre part Kafka et Openzwave. Ces deux modules ont un fonctionnement similaire, ils sont constitués d'une librairie reprenant les méthodes pour communiquer avec KNX ou Openzwave et exposant les différentes méthodes relatives à l'utilisation des *devices*. Ils sont également constitués d'un fichier principal (`knx.py` et `zwave.py`), chaque fichier démarre un thread producteur Kafka, envoyant à intervalles réguliers les valeurs des stores, des senseurs et *dimmers* Openzwave, et un thread consommateur Kafka, écoutant sur le topic "knx" ou "zwave" les commandes à effectuer sur les *devices*.

4.5 Protocole des messages

En ce qui concerne les messages consommés par KNX et Openzwave, nous avons choisi de définir notre propre protocole. Celui-ci fait correspondre la clé du message reçu à l'action à effectuer et le contenu du message aux éventuels paramètres à transmettre. Ces différents paramètres sont au format JSON puis encodés en bytes afin d'être transmis au broker Kafka et consommés par une autre entité.

4.5.1 KNX

Pour KNX nous retrouvons les messages suivants :

Production

- `read_percentage_blinds` : Ce messages est produit à intervalles de 5 secondes, il permet d'envoyer dans le topic `knx` le pourcentage d'ouverture de tous les stores de toutes les salles.

Consommation L'étage et la chambre permettent d'identifier l'appareil sur lequel il faut agir, pour cela ces deux paramètres sont donnés dans le corps du message ce qui permet d'intéragir avec le bon *device* en fonction de la position de l'utilisateur.

- `open_blinds` : permet d'ouvrir les stores (100%)
- `close_blinds` : permet de fermer les stores (0%)

— `percentage_blinds` : met les stores à un certain pourcentage qui est passé dans la valeur du message

— `percentage_radiator` : met les radiateurs à un certain pourcentage qui est passé dans la valeur du message

4.5.2 Openzwave

Openzwave est capable de traiter les messages suivants :

Production Les messages suivants sont produits à intervalles de 5 secondes.

— `sensors_get_temperature` : produit la valeur de température de chaque senseur

— `sensors_get_humidity` : produit la valeur de l'humidité de chaque senseur

— `sensors_get_luminance` : produit la valeur de la luminance de chaque senseur

— `sensors_get_motion` : produit une indication de présence de chaque senseur

— `dimmers_get_level` : produit la valeur de chaque variateur (*dimmer*)

Consommation

— `dimmers_set_level` : met les *dimmers* à un certain pourcentage

4.6 Base de données

En ce qui concerne la base de données, nous avons opté pour une base relationnelle avec MySQL [9] qui nous a permis de représenter les différentes entités de manière efficace. Comme représenté sur la figure 3, la base comporte six tables :

— `Beacon` : contient tous les Beacons et dans quelle pièce

— `Room` : contient les différentes pièces et leur étage

— `Device` : contient tous les *devices* (KNX et Openzwave), identifiés de manière unique et leur pièce respective

— `KnxNode` : contient les informations concernant les *devices* KNX uniquement (type, bloc et étage)

— `ZwaveNode` : contient les informations concernant les *devices* Openzwave uniquement (identifiant dans le réseau Openzwave et nom)

— `Log` : contient les logs produits par tous les *devices* avec la valeur, le type de valeur et le *timestamp* (à la seconde) de la mesure

Ces relations permettent des usages tels que :

— Pour un Beacon donné, retrouver tous les logs des *devices* de la pièce associée

— Avec un type, bloc et étage KNX donnés, retrouver dans quelle pièce ils se trouvent

— Réaliser des statistiques sur les logs, comme la température moyenne dans une pièce

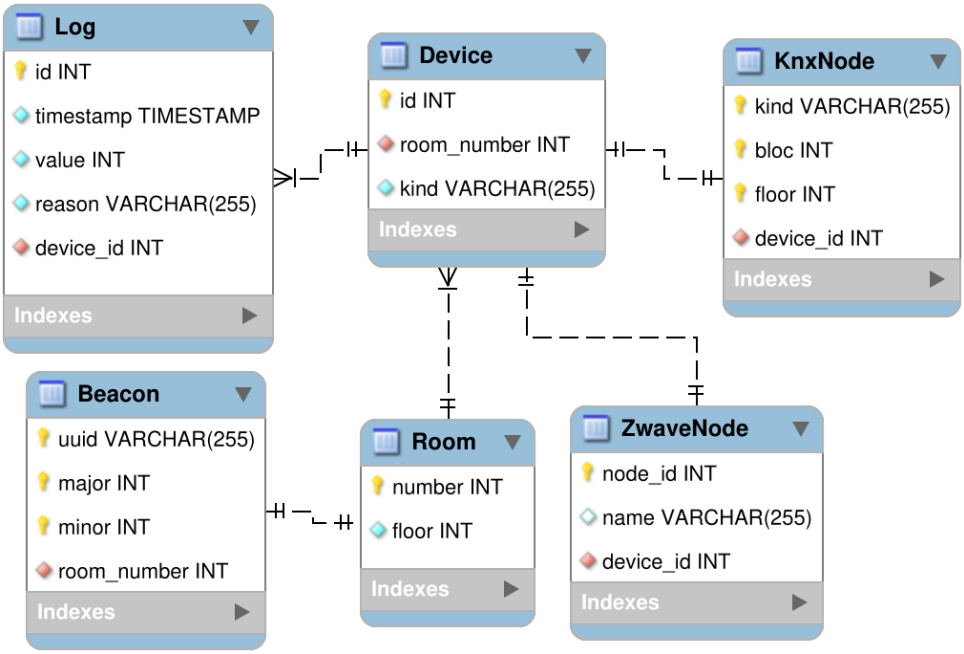

FIGURE 3 – Schéma relationnel de la base de données

4.7 Automatic controller

Le contrôleur automatique est déployé dans un container Docker avec docker-compose. Son rôle est de gérer les états des *devices* dans les différentes salles. Pour ce faire, il lit les messages dans les topics Kafka propres à KNX et Openzwave. En fonction des informations récupérées sur les *devices*, il applique certaines règles logiques afin de les contrôler automatiquement. Les différentes règles sont les suivantes (à savoir que les différentes valeurs définies sont arbitraires et modifiables très facilement) :

- Si aucune présence est détectée dans la pièce, le radiateur se met automatiquement à 10% afin d'économiser de l'énergie
- Si quelqu'un est dans la pièce, le radiateur se règle à 90% afin que la personne n'ait pas froid
- Si l'humidité est supérieure à 50% les stores de la pièce se ferment
- S'il est plus tôt que 19 heures, que quelqu'un est présent dans la pièce et que la luminosité est faible, les stores s'ouvrent automatiquement
- Si l'heure actuelle est comprise entre 19 heures et 7 heures du matin et que quelqu'un est présent dans la pièce, ou, si la luminosité est trop faible, la lumière s'allume alors automatiquement. Dans un des cas contraires, la lumière s'éteint.

4.8 DB to Kafka

Pour minimiser les dépendances à la base de données pour les modules KNX, OpenZWave et Automatic controller, nous avons réalisé un connecteur, `db_to_kafka.py`, qui produit dans le topic Kafka `db` la liste de tous les *devices* avec leur type effectif (KNX ou OpenZWave) et leur emplacement physique (dans quelle pièce ils se trouvent). Ainsi, les modules reliés à Kafka nécessitant les informations sur les *devices* peuvent relire le topic `db` à partir de l'*offset* zéro pour obtenir tous les *devices*. Ce module est déployé dans un container Docker avec docker-compose.

4.9 Logs et statistiques

Pour garder une trace des valeurs des différents *devices*, un module `logs.py` écoute et consomme les messages Kafka des topics `knx` et `zwave` concernant la lecture des valeurs et réalise des insertions en base de données dans la table `Log`. Il est donc aisé de retrouver les anciennes valeurs des capteurs et réaliser des statistiques avec. Ce module est déployé dans un container Docker avec docker-compose.

5 Conclusion

5.1 Bilan personnel

Pour conclure, ce projet nous a permis d'apprendre et pratiquer Kafka, une technologie initialement inconnue pour nous, ce qui fût très instructif. Il nous a également permis d'approfondir les notions vues en cours concernant KNX, Openzwave et la détection des Beacons depuis un appareil Android. En ajoutant à tout cela une partie développement Android, une base de données SQL et l'utilisation de python. Ce projet est un très bon cas pratique mettant en lien toutes ces technologies plus intéressantes les unes que les autres.

5.2 Problèmes rencontrés

Néanmoins, plusieurs difficultés ont été rencontrées durant le développement de ce projet :

— Toute l'architecture Kafka est déployée avec docker-compose. Afin de configurer le tout de manière automatique, nous avons rencontrés quelques problèmes, notamment liés à la sortie standard des containers basés sur l'image python. En effet, le container n'affiche pas la sortie standard des scripts python dans les logs si une option spécifique n'est pas renseignée dans le `docker-compose.yml`. Pendant un bon moment nous croyions que le consommateur ne fonctionnait pas, alors que c'était uniquement l'affichage qui était défaillant.

— En ce qui concerne KNX, la documentation fournie ne correspondait pas parfaitement avec le fonctionnement réel du protocole, notamment au niveau des trames renvoyées après la lecture/écriture des données dans le socket. Ce protocole est également peu pratique lorsqu'on souhaite l'utiliser pour transmettre plusieurs messages consécutifs sans fermer la connexion.

— La procédure d'utilisation fournie pour Openzwave manque d'informations concernant le reset manuel du *controller*. La documentation officielle manque également cruellement de précision pour la plupart méthodes.

— De manière générale, l'utilisation de python pour ce genre de système distribué est à notre avis peu pratique, python n'étant pas typé et compilé, toutes les cas ont dus être massivement testés à l'exécution pour être certains que le code marche. De plus, dans un contexte de sérialisation et désérialisation fréquente, il est plus judicieux et pratique de disposer d'une conversion fortement typée des données.

5.3 Améliorations possibles

Les améliorations suivantes pourraient être apportées au projet :

— Gestion des droits des utilisateurs : chaque utilisateur devrait pouvoir s'authentifier dans l'application à l'aide d'un *username/password*, stockés en base de données. Chaque utilisateur aurait également une liste des salles associées à son profil. Etant donné que l'application détecte la salle à l'aide du Beacon le plus proche, il serait alors très simple de savoir si un utilisateur dispose des droits nécessaires pour accéder aux fonctionnalités proposées par les *devices* de la salle ou il se trouve.

— Perfectionnement de l'expérience utilisateur de l'application Android.

6 Références

[1] Fondation Apache. Apache kafka. https://kafka.apache.org/. Consulté en 2019.

[2] Amazon. Amazon web services. https://aws.amazon.com/. Consulté en 2019.

[3] Docker. Docker documentation. https://docs.docker.com/. Consulté en 2019.

[4] Docker. Overview of docker compose. https://docs.docker.com/compose/. Consulté en 2019.

[5] Confluent. Confluent : Apache kafka and event streaming platform for the enterprise. https://www.confluent.io/. Consulté en 2019.

[6] Confluent. confluentinc/cp-kafka. https://hub.docker.com/r/confluentinc/cp-kafka. Consulté en 2019.

[7] Confluent. confluentinc/cp-zookeeper. https://hub.docker.com/r/confluentinc/cp-zookeeper. Consulté en 2019.

[8] Flask. Welcome to flask - flask documentation (1.1.x). https://flask.palletsprojects.com/en/1.1.x/. Consulté en 2019.

[9] MySQL. Mysql. https://www.mysql.com/. Consulté en 2019.