Table des matières

1. Introduction

1.1. Vue d'ensemble

L'Internet des choses est un système de dispositifs informatiques interdépendants, de machines mécaniques et numériques dotés d'identifiants uniques et qui permet le transfert de données sur un réseau sans nécessiter d'interaction humaine.

Ce système grandit en popularité chaque année, le nombre de dispositif continue de grandir et génère une énorme quantité de données à enregistrer et à traiter. Un environnement dédié, basé sur le cloud, est donc nécessaire.

La HES-SO Valais a développé en 2017 une API, cloud.iO pour satisfaire la demande d'un environnement dédié, ce projet fut amélioré avec une deuxième version en 2020. Il propose la solution suivante :

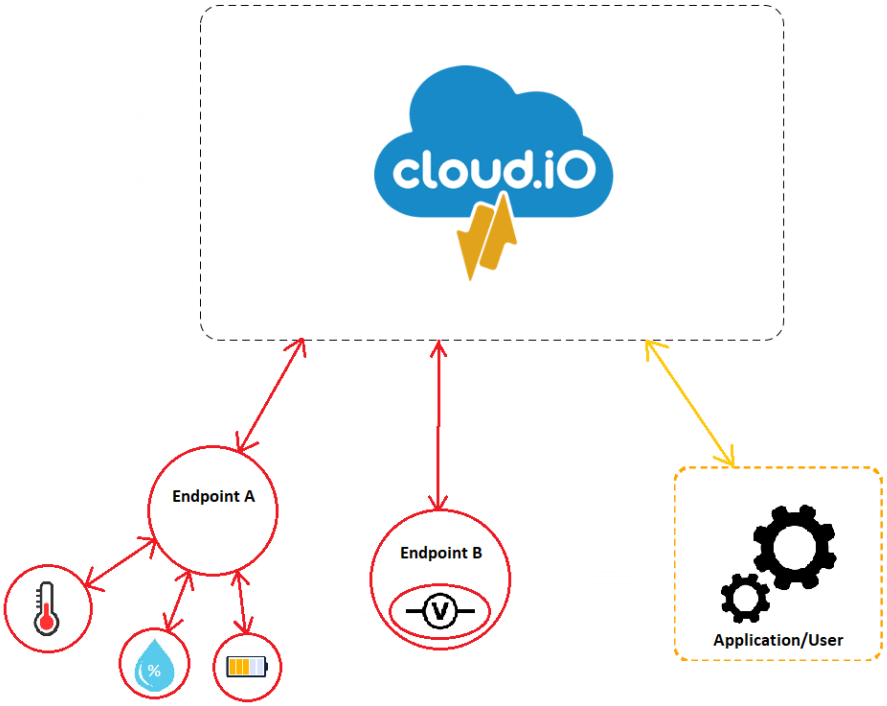


Figure 1: Architecture simplifiée de cloud.iO

Nous constatons sur cette image que des mesures provenant de différents capteurs d'un certain système sont transférées sur le serveur fournis par cloud.iO. Les mesures doivent être communiquées, selon l'envie des utilisateurs, selon un laps de temps ou selon des changements radicaux de mesures avec un certain protocole de communication (par exemple : http) au serveur. Pour cela, il nous faut donc une passerelle, un système embarqué qui sera appelé « Endpoint ».

Ensuite, l'utilisateur ou une application externe au système pourra consulter les données envoyées sur le serveur de cloud.iO, les traiter et si nécessaire modifier certains paramètres présents dans le Endpoint pour une meilleure configuration du système.

Si on prend par exemple, un système de chauffage simple possédant un capteur de température et un paramètre indiquant à quelle température la salle chauffée doit être, alors un utilisateur à l'extérieur de l'emplacement du système peut retrouver la température actuelle et la température paramétrée. Il peut changer à tout moment la température paramétrée et voir si la température actuelle de la salle correspond à la température souhaitée. Dans le cas contraire, il pourrait découvrir qu'un problème se situe dans le système de chauffage.

1.2. But du projet

Pour le moment, une version du Endpoint de cloud.iO est disponible en JAVA, cependant, ce langage de programmation est plutôt lourd et « high level », ce qui peut être un inconvénient car les Endpoint vont normalement être définit dans des systèmes embarqués, pour alléger le logiciel, pouvoir faire du temps réel en étant plus proche de ce qui se passe dans un système embarqué, créer un Endpoint en C/C++ serait plus cohérent.

Ainsi, le projet LTE cloud.iO consiste à créer une API en C/C++ qui devra être transportable et qui comportera les fonctionnalités présentes dans la version JAVA.

Pour réaliser ce projet, il faudra tout d'abord explorer le projet cloud.iO actuel en profondeur pour pouvoir au mieux exploiter les services qu'il propose. Ensuite, il faudra effectuer une analyse complète.

L'API sera d'abord effectuée sur l'environnement QT afin d'avoir une première version rapidement et sera ensuite transporté sur un nRF9160-DK. Une démonstration d'un Endpoint communiquant avec le serveur de cloud.iO sera finalement développé.

L'API sur l'environnement QT devra être « portable » pour simplifier l'implémentation au maximum de l'API sur le nRF9160-DK.

Le nRF9160-DK peut se connecter au réseau mobile via les technologies LTE-M ou NB-IoT, ainsi un test pratique sera effectué pour comparer ces deux antennes.

2. cloud.iO

Pour pouvoir réaliser ce projet, il fut indispensable de bien comprendre l'architecture de cloud.iO, les différentes fonctionnalités qu'il propose et le rôle de l'Endpoint dans cette architecture.

2.1. Architecture de cloud.iO

On peut définir dans ce projet trois acteurs principaux :

- Le serveur qui doit rassembler toutes les données utiles de plusieurs systèmes composés notamment de capteurs et de paramètres.
- L'Endpoint qui doit être capable de communiquer au serveur les données et les statuts du système.
- Un utilisateur ou une application qui peut accéder aux données enregistrées sur le serveur pour les traiter et ensuite pouvoir modifier certains paramètres du système à travers le serveur.

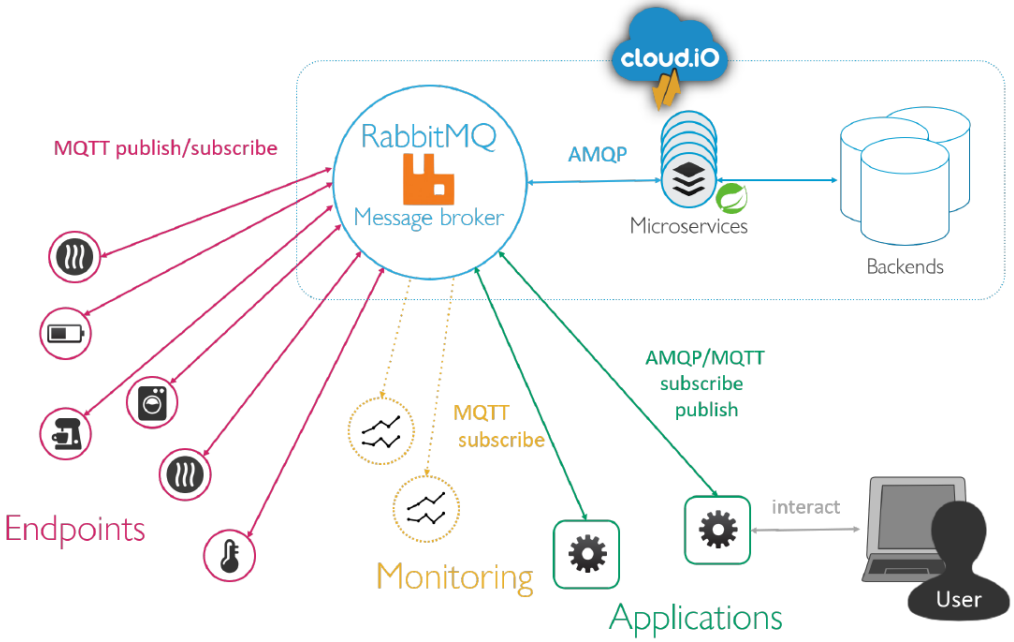


Figure 2: Architecture of cloud.iO

Cloud.iO met donc en place un logiciel d'agent de messages (RabbitMQ), sur lequel des Endpoint et des utilisateurs/applications peuvent communiquer avec un protocole MQTT ou AMQP. Le protocole AMQP est aussi utilisé pour pouvoir stocker/récupérer des données dans des bases de données au travers de services SPRING.
Les bases de données utilisées sont InfluxDB et MongoDB. InfluxDB est utilisé pour stocker des données avec une chronologie donc des mesures, des paramètres et des statuts envoyés par l'Endpoint. MongoDB s'occupe d'enregistrer les utilisateurs ainsi que leur mot de passe afin d'interdire l'accès à certains éléments des Endpoints pour les personnes non-concernées. Il enregistre aussi les Endpoint et leur certificat d'authentification, empêchant ainsi que des

données malveillantes viennent perturber/encombrer le serveur ou fausser certains résultats d'un Endpoint grâce au protocole de sécurisation SSL.

Le Endpoint est la partie principale de ce projet : il peut soit être une chose (Thing) supportant cloud.iO, soit être la passerelle entre une chose et cloud.iO. Il doit donc être capable de transférer les données des choses au serveur.

2.2. Fonctionnalités de cloud.iO

Cloud.iO permet l'utilisation des trois fonctionnalités les plus importantes pour une plateforme IoT :

- Accès aux paramètres des Things
- Stockage de l'historique des mesures et des paramètres des Things
- Visualisation de l'historique et donc de l'évolution des Things

Comme cloud.iO utilise un broker, l'utilisation de topic spécifique pour la communication avec l'Endpoint est utilisée.

Topic	Data
@update/*EndpointUUID*/.../AttributeName	The status or measure Attribute formatted in JSON
@set/*EndpointUUID*/.../AttributeName	The setpoint or parameter Attribute formatted in JSON
@online/*EndpointUUID*	The Endpoint Data Model formatted in JSON
@nodeAdded/*EndpointUUID*/*NodeNameToAdd*	The Node to add formatted in JSON
@nodeRemoved/*EndpointUUID*/*NodeNameToRemove*	no Data
@offline/*EndpointUUID*	no Data

Figure 3: Topic

A part pour le topic @set, les messages vont de l'Endpoint au serveur. Le topic @set permet de modifier certains paramètres dans un Endpoint.

@online doit être utilisé pour informer le serveur de la structure de l'Endpoint.

@update est utilisé pour indiquer les nouvelles mesures prises.

@nodeAdded et @nodeRemoved est utilisé pour changer la structure de l'Endpoint, vous retrouverez la définition d'un nœud au chapitre 4.1.

@offline est émit quand l'Endpoint se déconnecte.

La formation des topics sur un Endpoint est expliquée dans le chapitre 4.1.

Cloud.iO propose d'autres fonctionnalités disponibles pour l'utilisateur, plus d'informations sont disponibles sur le travail de Master Mr. Bonvin Lucas (voir chapitre 10).

3. nRF9160, Hardware, Tools

L'un des buts de ce travail est de pouvoir implémenter une application sur un système embarqué qui pourra communiquer avec le serveur cloud.io. Le système embarqué sera un nRF9160-DK de la société Nordic Semiconductor.

Le nRF9160-DK est un kit de développement monocarte pour l'évaluation et le développement sur le nRF9160 SiP (système en package) pour LTE-M, NB-IoT et GPS. Il intègre un processeur Arm Cortex-M33, un modem est intégré pour prendre en charge les technologies LTE-M et NB-IoT. Des librairies pour la cryptographie sont aussi utilisable.

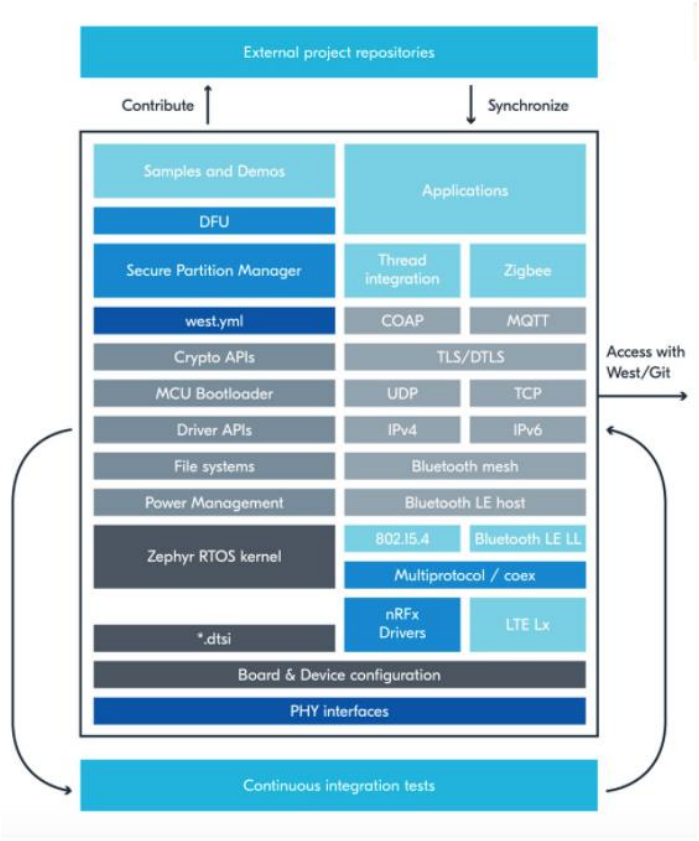


Figure 4: Architecture du nRF9160

LTE-M et NB-IoT sont des technologies de communication réseau sans fil conçues pour l'internet des choses. Les principales différences entre ces deux technologies sont le temps de réponse, le débit et le coût. LTE-M possède de meilleurs résultats que NB-IoT mais est plus coûteux, ainsi, pour de faibles volumes de données et des applications ne nécessitant pas un temps de réaction élevé, NB-IoT peut totalement convenir.

Pour le développement sur la cible, le logiciel nRF Connect de Nordic permet l'installation d'un IDE appelé Segger, l'installation de J-Link nous permettant de programmer le système embarqué via un câble USB, l'outil nRF Link Monitor qui est une interface de commande AT qui permet de voir l'état du modem et d'extraire des informations sur le réseau, mais aussi de

nombreux exemples pour pouvoir voir comment utiliser les différentes fonctionnalités du kit de développement.

Le système embarqué peut aussi utiliser les fonctionnalités de Zephyr, un système d'exploitation open-source, la librairie de Zephyr propose elle aussi plusieurs exemples pour l'utilisation des différentes ressources. Zephyr inclut aussi une « command line tool » nommé west qui permet la compilation, la programmation et le débogage d'une application.

Figure 5: OS Zephyr

Créer son application à partir d'un exemple semble être la meilleure solution. En effet, configurer soi-même son projet en utilisant les librairies du nRF et de Zephyr requièrent de l'expérience et du temps car leurs librairies ont des dépendances à d'autres librairies mais peut aussi amener des problèmes de compatibilité (par exemple, utilisation du même nom de structure entre deux librairies différentes).

Pour pouvoir utiliser un exemple, il est bon de jeter un coup d'œil aux fichiers « Kconfig » et prj.conf, le premier fichier définit des noms de variables qui peuvent être utilisés dans le projet. L'intérêt de ce fichier est que dans l'IDE Segger, nous pourrons retrouver ces variables dans le menu de configuration et nous pourrons les modifier à tout moment via Segger, à savoir qu'un projet qui va inclure les package de Zephyr dans le fichier CMakeLists, va ajouter beaucoup de variables avec une valeur par défaut que l'on peut modifier.

Si nous voulons que ces variables ne prennent pas leur valeur par défaut à chaque chargement de projet dans l'IDE, le fichier prj.conf permet de leur attribuer une nouvelle valeur.

Ce système remplace donc la certaine utilisation d'un fichier « define.h », fichier qui aurait pu être long à parcourir s'il devait contenir toutes les variables que Zephyr a défini pour le développement de ses librairies.

4. Analyse de l'Endpoint

Un Endpoint peut être une passerelle entre plusieurs choses et le cloud ou être lui-même une chose supportant le cloud, c'est-à-dire que la communication devra suivre un certain protocole pour que ces deux éléments puissent utiliser les données transmises.

La structure suivra un modèle de données spécifiques : il possèdera des nœuds, des objets et des attributs, ceci sera expliqué plus en détail au chapitre suivant. Ce modèle sera formaté en JSON.

Pour pouvoir se connecter au serveur et transmettre les données, l'Endpoint doit posséder avant sa mise en route de son UUID, son propre certificat, sa clé privée et le certificat d'autorité du serveur. Ces éléments servent à identifier l'Endpoint avec une authentification SSL. Ces éléments peuvent être obtenu grâce aux services REST de cloud.iO.

4.1. Spécifications d'un Endpoint

Voici les éléments qu'un Endpoint doit posséder pour un bon fonctionnement de l'application.

- Possède un schéma basé sur des nœuds, objets, attributs
- Possède un identifiant (UUID), un certificat et une clé donné par cloud.iO
- Unicité, un Endpoint doit bénéficier d'un identifiant unique
- Connectivité, un Endpoint doit toujours chercher à être connecté à son cloud
- Communication, un Endpoint transfert ses données au cloud avec un protocole donné (ici : MQTT)
    - Après une connexion, l'Endpoint transmet ses dernières données au cloud
    - Un Endpoint peut voir ses paramètres être modifiés par le cloud
    - La communication doit être sécurisé, l'Endpoint doit s'authentifier au serveur
- Mémorisation, si le Endpoint n'est pas connecté au serveur, alors il faut garder en mémoire les dernières données récoltées

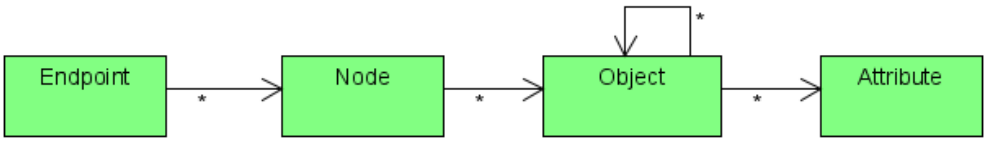
Les Nœuds représentent les choses. Ils peuvent avoir plusieurs capteurs ou d'autres composants (appelés ici Objets) dont on voudrait les données. D'ailleurs les Objets peuvent posséder d'autres objets selon la configuration du système. Finalement, les objets possèdent des données de différents types (valeur numérique, booléen, etc…) qui doivent être identifiés entre eux-mêmes, nous les appelleront pour cela des Attributs.

Supposons un système de chauffage d'une maison, la température de chaque pièce de la maison est mesurée par un ou plusieurs capteurs et contient des paramètres indiquant la température de la pièce souhaitée ainsi que la puissance à fournir au radiateur.

On peut ainsi construire le Nœud « dining_room » qui aura un Objet « temperatureController » qui possèdera des attributs pour les différents paramètres ainsi qu'un sous-Objet pour chaque capteur de température, ces sous-Objets possèderont eux-mêmes des Attributs pour la température actuelle de la pièce.

Nous pourrons ensuite ajouter d'autres nœuds, un pour chaque pièce, avec la même logique.

Ceci nous donne le schéma suivant pour la structure d'un Endpoint :



Ce modèle permet de créer des topics uniques pour chaque Attribut de la forme :

endpointUUID/nodeName/objectName/…/attributeName

L'Attribut représente les Data, il possède la valeur que les utilisateurs vont chercher à connaître et/ou à paramétrer. Sa valeur peut être de différents types : valeur numérique (entier ou à virgule), valeur booléenne ou une chaîne de caractères.

L'Attribut possède aussi un timestamp qui permet de retracer l'historique des valeurs dans une base de données orientée séries chronologiques.

Les Attributs peuvent êtres de différents types :

- Statique : Read-only, valeur constante
- Statut : Read-only, valeur calculée
- Mesure : Read-only, valeur mesurée par un procédé physique
- Paramètre : Read/Write, paramètre changeant le comportement du Endpoint, persistant donc mémoriser par le cloud
- SetPoint : Read/Write, non-persistent

4.2. Analyse comportementale

En premier lieu, lors de la mise en route, le système embarqué initialise ses différents composants tels que son RTOS, son LTE, ses interruptions, ses ports, etc… Ensuite, la fonction main peut instancier un EndpointController, cette instance aura pour tâche de contrôler le déroulement de l'API et la connexion avec le serveur en automatisant le plus de tâches possibles pour faciliter la vie des futurs utilisateurs de LTE cloud.iO.

Le EndpointController devra en premier tenter périodiquement de se connecter au serveur RabbitMQ. Tant que le serveur est offline, le EndpointController doit continuer à stocker les données dans une queue ou un fichier.

Nous utiliserons pour cela un Thread : tandis que la fonction main s'occupera de récupérer comme il l'entend les différents Data et de les transmettre à l'EndpointController, un Thread sera automatiquement lancé par l'EndpointController pour tenter des connexions périodiquement au serveur.

Quand la connexion est réussie, le EndpointController contrôle s'il y a des données mémorisées à envoyer. Pour éviter une surabondance de données transmises, une solution doit être développée pour trier les messages. En premier, les messages les plus récents doivent être

envoyés, ensuite il n'y a pas forcément d'utilité à envoyer toutes les mesures, un tri doit être effectué.

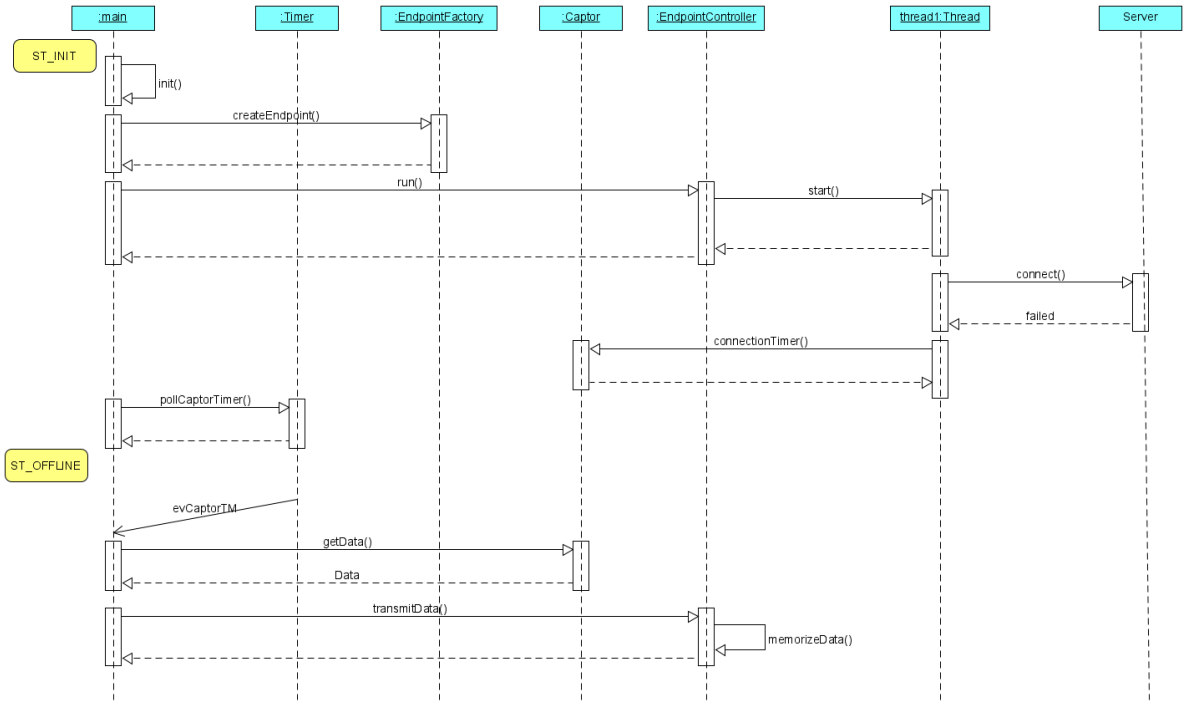


Figure 6: Diagramme de séquence, initialisation

Ce diagramme montre le comportement cité plus haut, ceci est une première ébauche pour ce qui sera la routine d'exécution pour notre projet. A noté que pour tenter une connexion périodiquement, permettre le poll des capteurs, il nous faut des Timers, que l'on pourra associer à une fonction callback lors du timeout, comme l'utilisation des Threads est utilisé, nous devons disposer d'un Operating System qui nous proposera certainement un moyen d'indiquer le callback que le timeout du Timer doit appeler. L'utilisation d'un OS obligatoirement sera discuté dans la partie analyse structurelle (chapitre 4.3).

L'utilisateur de LTE cloud.iO devra implémenter la récupération des Data (poll des capteurs) puis utiliser une méthode de EndpointController qui s'occupera lui-même de stocker ou d'envoyer les données au Cloud.

Quand la connexion est réussie, il n'y a plus besoin de tenter de se connecter, ainsi le thread va changer son rôle : il va devoir regarder toutes les données mémorisées par le EndpointController et envoyer ces informations au serveur, il sera aussi prévenu si la déconnexion arrive. Un 2ème thread ensuite sera utilisé pour lire les requêtes en provenance du serveur et faire le traitement des messages reçus.

Ensuite, le fonctionnement normal se produira : quand la fonction main voudra pousser de nouvelles valeurs, il utilisera une méthode d'EndpointController qui ira indiquer au thread les données à publier au serveur.

Pour exécuter cela, il faudra utiliser des queues protégées par des sémaphores pour empêcher l'utilisation d'une ressource par deux threads en même temps.

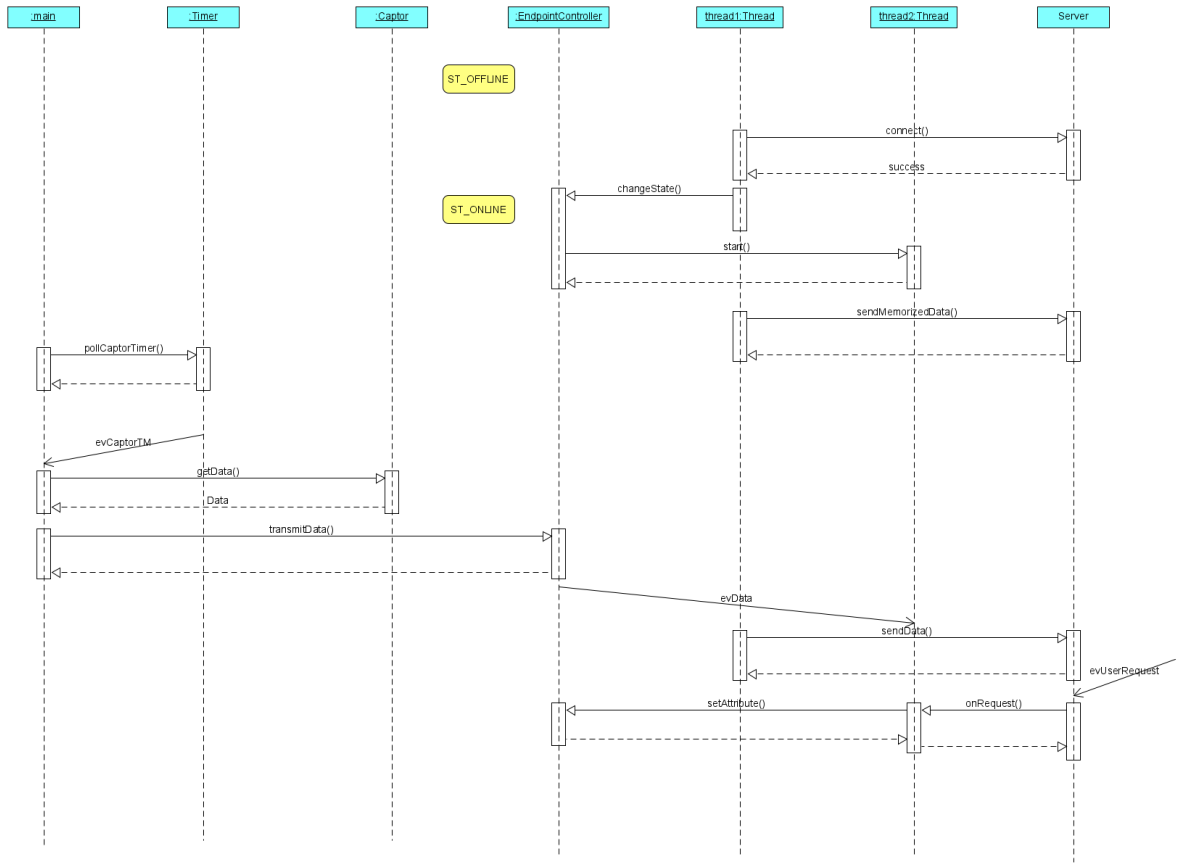


Figure 7: Diagramme de séquence, connexion réussie

La communication avec le protocole MQTT entre cloud.iO et l'Endpoint se passe de la manière suivante :

1. L'Endpoint démarre la connexion par un handshaking (SSL)
2. L'Endpoint publie un premier message sur le topic @online/EndpointUUID
3. L'Endpoint s'abonne au topic @set/EndpointUUID/#
4. L'Endpoint met à jour les valeurs des attributs sur le topic @update/EndpointUUID/…/AttributeName

4.3. Analyse structurelle

4.3.1. Paquets

Le projet sera séparé en plusieurs paquets (package) pour une meilleure lecture du code et surtout pour séparer l'implémentation des paquets qui seront universelle, possible sur tous les environnements supportant le C++ et disposant d'un OS, et l'implémentation des paquets qui dépendront donc de l'environnement, ces derniers seront nommés avec le préfix <<Core>>.

Comme les classes de bases voudront utiliser des objets dépendants de l'environnement comme les sémaphores par exemple, il faut ajouter un package avec des interfaces, des classes comprenant uniquement des méthodes virtuelles donc sans implémentation. Les classes des package Core devront hériter de ces interfaces et utiliser des instances propres à leur environnement pour implémenter les méthodes.

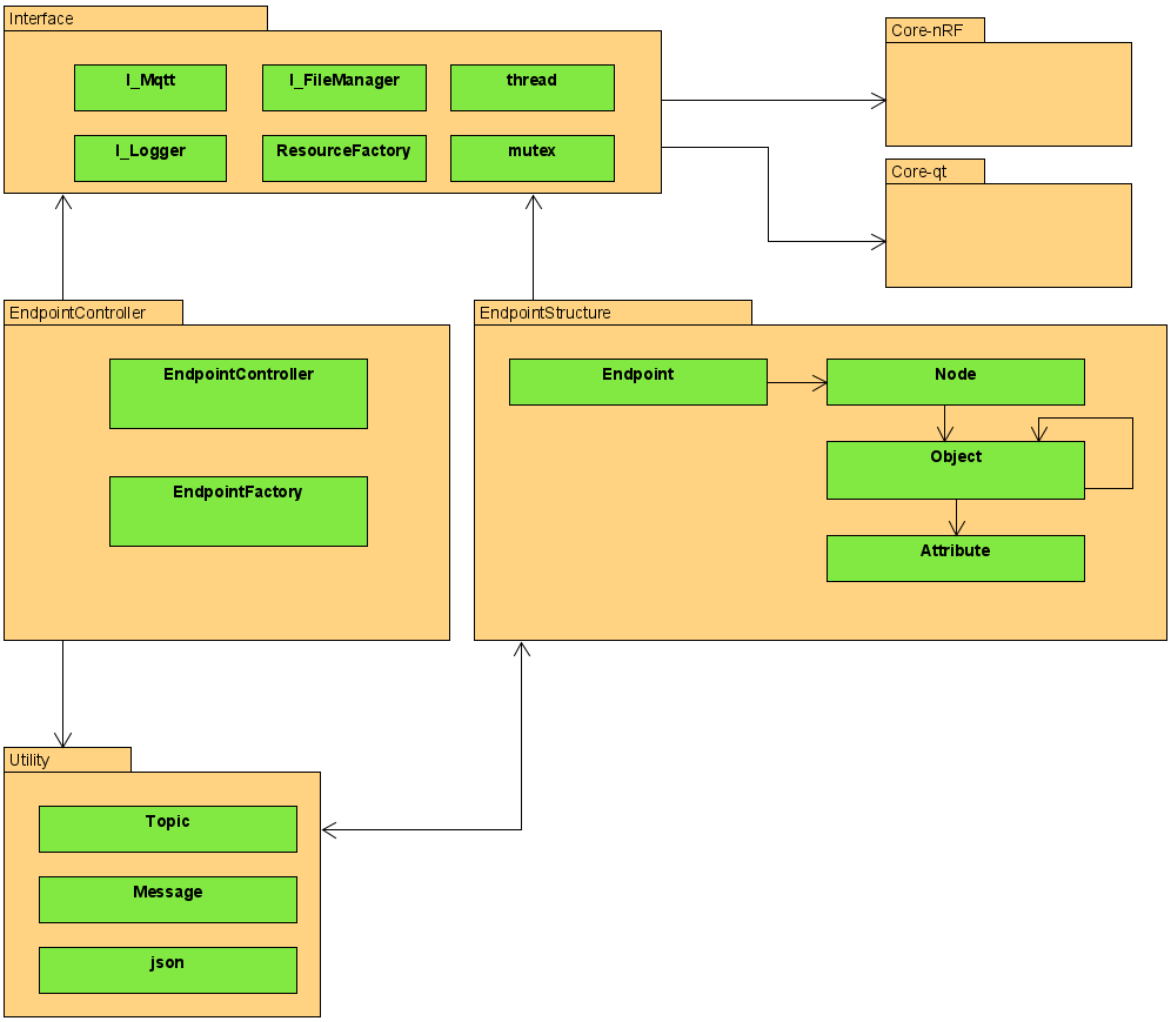


Figure 8: Schéma des paquets

Cette figure est une représentation simplifiée : elle illustre les package ainsi que l'utilisation des interfaces : une classe de base peut appeler une méthode présente dans une classe du

package core uniquement en passant par la classe virtuelle présente dans les interfaces. Ceci permet de ne pas devoir modifier le code des classes de bases en changeant d'environnement.

De plus amples informations sur les différents paquets et leurs classes sont présentés au chapitre 5.

4.3.2. Utilisation d'un operating system

L'évolution au travers des années nous a permis de construire des systèmes embarqués avec beaucoup de mémoire et un hardware utilisant des clocks avec une fréquence élevée et donc beaucoup d'instructions exécutées par seconde, ainsi nous avons pu commencer à construire des systèmes possédant des RTOS, utile pour faire de la programmation temps réel et pour pouvoir exécuter des logiciels multi-tâches avec du pseudo-parallélisme avec un micro-processeur.

Ainsi, en regardant la complexité du projet, sachant qu'il doit gérer les capteurs, envoyer les Data au serveur et potentiellement répondre à une requête de ce dernier, le projet peut être découper en plusieurs tâches (Thread) ce qui rendra la structure du logiciel bien moins complexes et une prise en main du projet plus facile.

Ceci demandera donc l'utilisation d'un operating system, on pourra en profiter pour utiliser les autres fonctionnalités : les sémaphores, les évènements, etc…

Ce projet donc ne pourra pas forcément être transportable dans tous les systèmes embarqués existant, surtout dans de vieux micro-processeurs. Ceci n'est pas un problème si l'on regarde les nouveaux systèmes industriels implémentés qui veulent utiliser l'Internet Of Things.

5. Classes communes

5.1. Structure du code

Les classes sont réparties en quatre différents paquets.

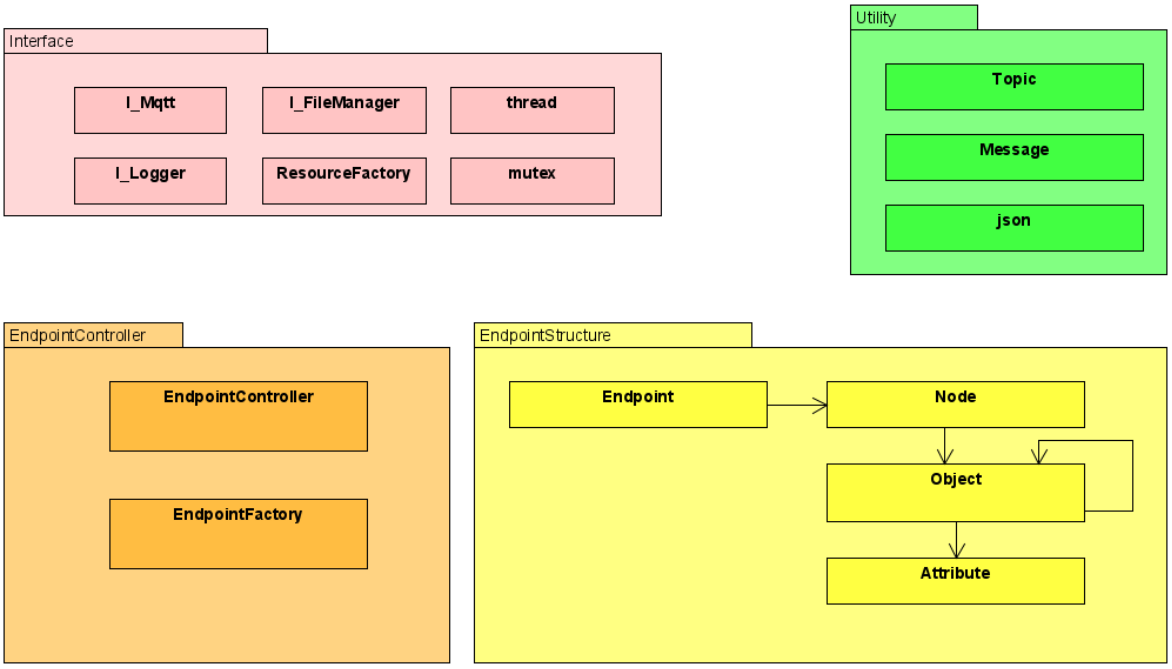


Figure 9: Classes communes

- Le paquet EndpointStructure représente le schéma selon le modèle de donnée spécifique d'un Endpoint expliqué au chapitre 4.1.
- Le paquet Utility contient des classes utiles pour fonctionnement de l'Endpoint qui ne dépendent pas de l'environnement.
- Le paquet Interface contient les méthodes virtuelles utiles pour le fonctionnement de l'Endpoint qui dépendent de l'environnement, ces méthodes devront être définies dans des classes implémentant les différentes interfaces dans un autre paquet. Ces interfaces sont donc appelées des « Abstraction Layer ».
- Le paquet EndpointController contient les classes que l'utilisateur pourra utiliser pour créer la structure de son Endpoint, modifier les valeurs des attributs et notifier les changements au serveur.

5.2. Paquet EndpointStructure

5.2.1. Classe Cloudio_Endpoint

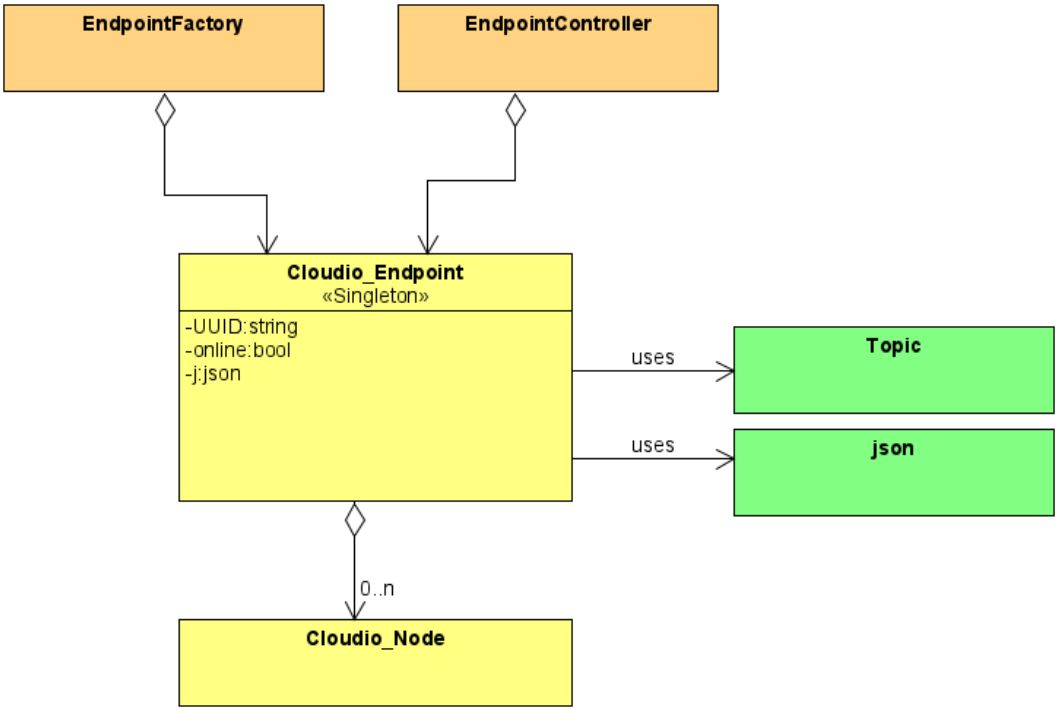


Figure 10: Cloudio_Endpoint

La classe Cloudio_Endpoint est la base de la structure, comme il est unique pour chaque Endpoint, cette classe est un singleton.

Cette classe possède comme attribut :

- L'UUID de l'Endpoint (un identifiant unique pour se distinguer de tous les Endpoints utilisant le serveur cloud.io.
- Une liste de Nœuds (Cloudio_Node)
- Un booléen indiquant si l'Endpoint est connecté au serveur
- Une variable contenant sa structure en JSON

Les méthodes de cette classe permettent de :

- Ajouter/enlever un Nœud et ses composants (les objets)
- Trouver une instance d'un nœud présent dans la liste de nœuds avec son nom
- Fabrication d'un document JSON contenant toute sa structure
- Fabrication d'un document JSON contenant uniquement les informations des nœuds (pour la fabrication du premier message à envoyer sur le topic @online (voir chapitre 2.2.))

- Transformer un topic en une liste de string (utile pour retrouver l'instance d'un Attribut avec un topic)
- Retrouver un Objet ou un Attribut à l'aide d'un topic

Les méthodes et attributs de cette classe sont utilisées par les classes EndpointController, EndpointFactory et Topic.

Autres points à noter :

- L'ajout d'un Nœud est refusé s'il existe déjà un Nœud dont le nom est identique (ceci causerait des problèmes pour distinguer les deux nœuds).
- L'attribut UUID est fourni automatiquement par l'interface I_FileManager

5.2.2. Classe Cloudio_Node

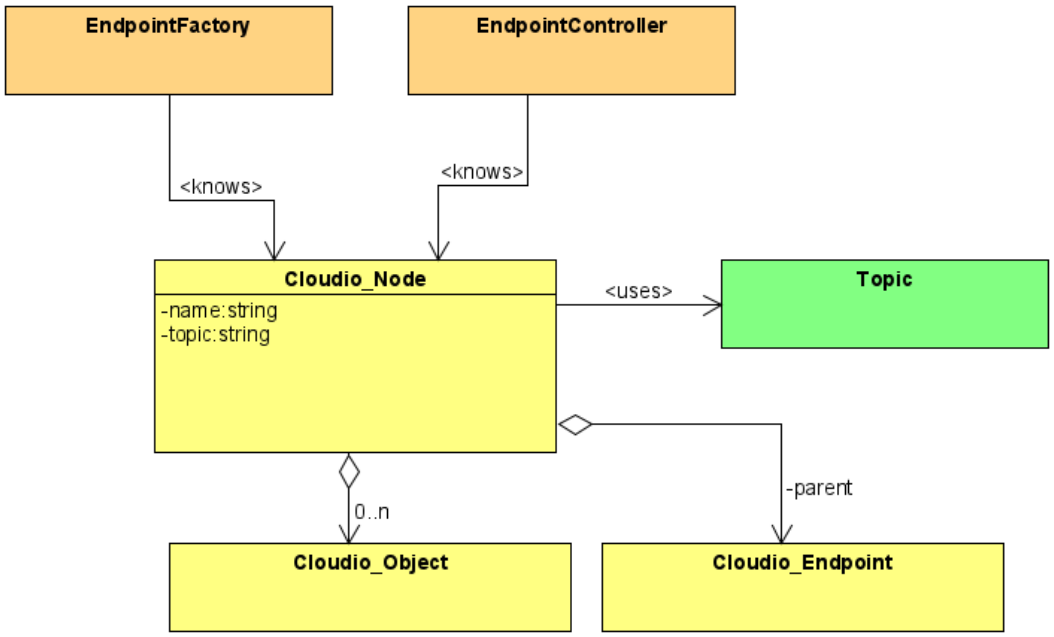


Figure 11: Cloudio_Node

La classe Cloudio_Node possède les informations qu'un Nœud doit posséder :

- Une liste d'Objets
- Un nom (unique)
- Un topic
- Un parent (un pointeur sur le singleton de Cloudio_Endpoint)

Les méthodes présentes dans la classe permettent de :

- Ajouter/supprimer des Objets
- Trouver un Objet dans la liste avec son nom

Autres points à noter :

- Son topic est automatisé selon l'UUID de l'Endpoint et son nom

- Le pointeur « parent » servait à vérifier qu'un utilisateur lie le nœud au Cloudio_Endpoint avant d'ajouter des objets (ceci permettait de pouvoir assurer l'automatisation des topics), maintenant, ce pointeur est seulement utilisé par la classe Topic pour faciliter cette automatisation.
- L'ajout d'un Objet dans la liste est refusé s'il existe déjà un Objet dont le nom est identique (ceci causerait des problèmes pour distinguer les deux Objets).

5.2.3. Classe Cloudio_Object

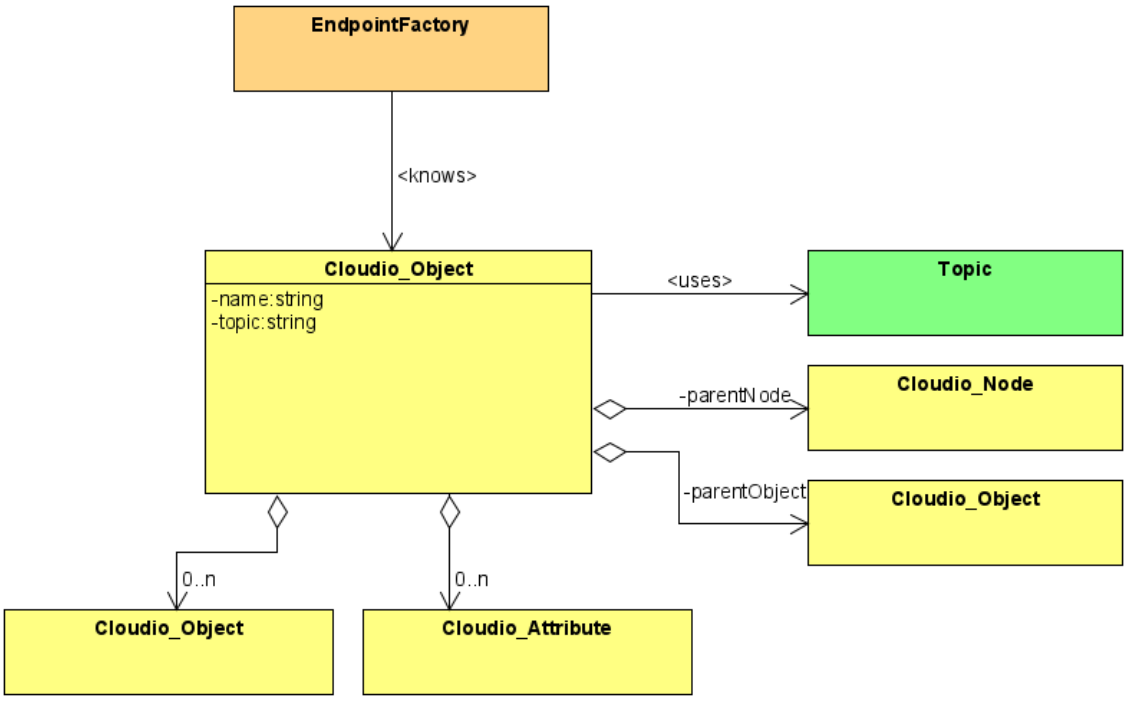


Figure 12: Cloudio_Object

Cette classe représente un Objet dans notre structure de l'Endpoint, elle est plus complexe que le nœud car elle possède non seulement des attributs mais aussi des sous-Objets, ce qui veut dire deux listes.

Les attributs de la classe sont :

- Une liste d'Objets
- Une liste d'Attributs
- Un nom (unique dans l'arborescence)
- Un topic
- Un parent (qui sera soit un Objet, soit un Nœud)

Les méthodes présentes dans la classe permettent de :

- Ajouter/supprimer des Objets et des Attributs
- Trouver un Objet ou un Attribut dans la liste avec son nom

Autres points à noter :

- Son topic est automatisé selon la structure de son parent et de son nom
- L'utilité du pointeur parent est mentionnée plus haut pour la classe Cloudio_Node, à savoir ici que comme le parent peut soit être un Nœud, soit être un Objet, deux pointeurs sont présents dans la classe, lorsqu'un pointeur prend la référence d'une instance, l'autre pointeur est forcé à rester à sa valeur par défaut.
- L'ajout d'un Objet ou d'un Attribut dans une des listes est refusé s'il existe déjà un Objet ou un Attribut dont le nom est identique, cela permet d'éviter de ne pas pouvoir distinguer deux instances semblables dans les listes de l'Objet.

5.2.4. Classe Cloudio_Attribute

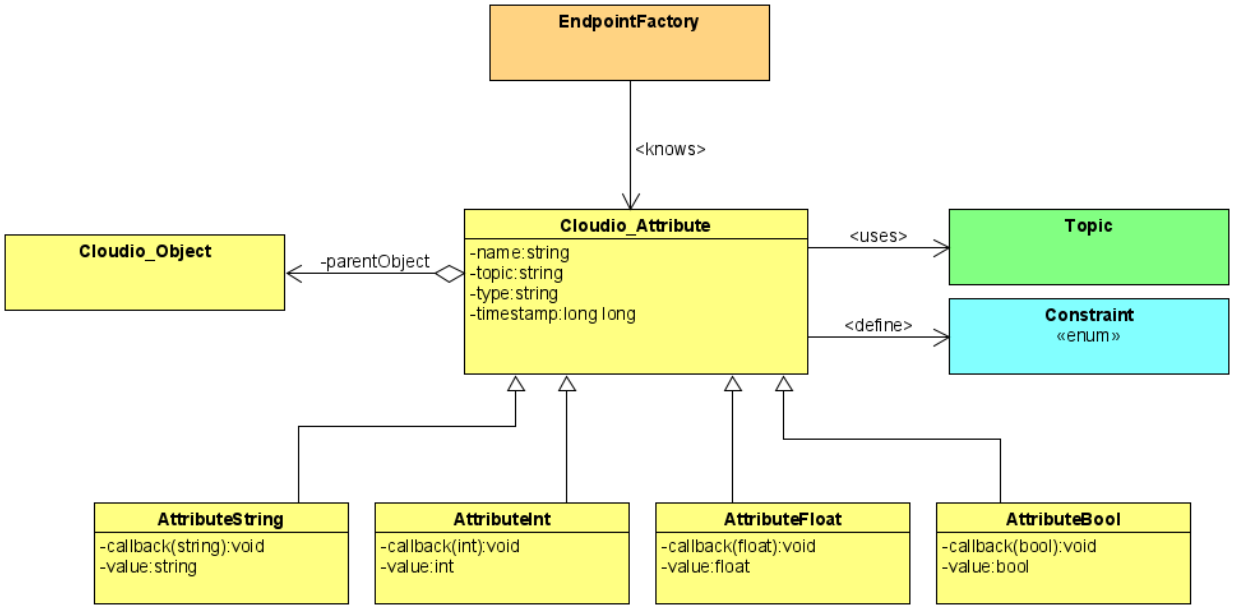


Figure 13: Cloudio_Attribute

La classe Cloudio_Attribute possède la valeur à envoyer au serveur. Cependant un Attribut peut prendre différentes formes, ainsi une instance d'un Attribut peut être fondamentalement différent d'un autre.

Tout d'abord, comme vu au chapitre 4.1., un Attribut peut être de plusieurs types, ceci sera appelé « Constraint » (contraintes), ensuite la valeur d'un attribut peut être un float, un int, un bool ou un string, donc la valeur d'un attribut n'a pas de type spécifique.

Ainsi les attributs d'un Cloudio_Attribute sont :

- Un nom (unique dans l'arborescence)
- Une contrainte (qui sera une énumération)
- Un type (int, float, bool ou string)

- Un timestamp (pour pouvoir représenter l'évolution des valeurs dans le temps)
- Un topic
- Un parent
- Un pointeur de fonction

Le topic, le nom et le parent fonctionnent de la même manière que la classe Cloudio_Node si vous voulez en savoir plus.

Comme dit précédemment, l'attribut valeur peut prendre différent type, pour se parer à cela, quatre classes furent créées : AttributeInt, AttributeBool, AttributeFloat, AttributeString avec un attribut valeur possédant un type définit. Ces classes héritent de la classe Cloudio_Attribute.

Il y a deux possibilités pour que la valeur d'un Attribut change.

Soit il est modifié par l'application (une nouvelle mesure de capteur par exemple), à ce moment-là, une fonction dédiée de la classe EndpointController (qui est disponible pour l'utilisateur) change la valeur de l'attribut et signale le changement au serveur.

Quand l'attribut est modifié, une méthode fournie par l'interface ResourceFactory permet de récupérer le timestamp actuel.

Un autre moyen est de modifier la valeur de l'Attribut via le serveur, à ce moment-là, l'application vérifie la contrainte de l'Attribut (la valeur d'une mesure ne peut pas être modifié par exemple), si l'Attribut est un « Parameter » ou un « SetPoint », alors la valeur est modifiée.

Si on essaie de modifier via le serveur un Attribut dont la valeur est un nombre par un string, la valeur ne changera pas non plus.

Le pointeur de fonction permet de lancer une fonction (callback) que l'utilisateur aura implémenté pour modifier les registres, le paramétrage du microcontrôleur pour influencer physiquement le système. Ce callback est appelé quand la valeur de l'Attribut fut modifiée via le serveur.

5.3. Paquet Utility

5.3.1. Classe Topic

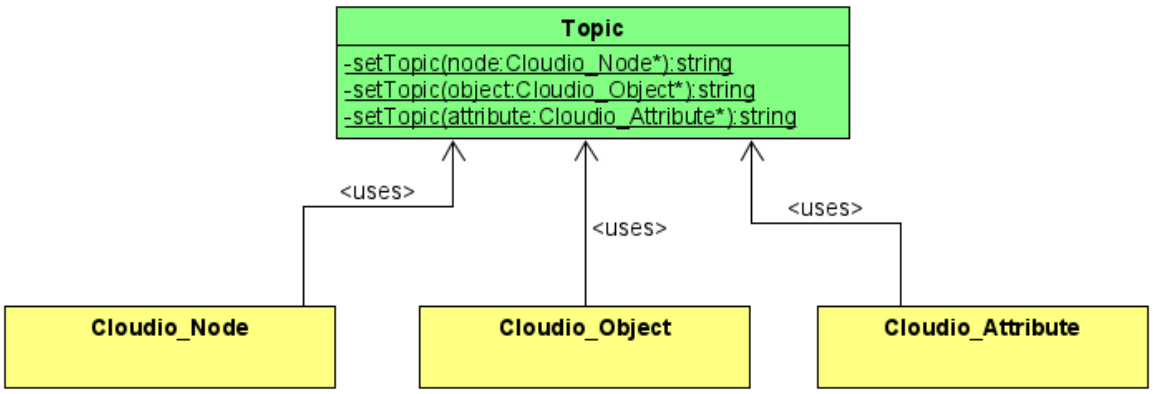


Figure 14: Topic

Comme signalé au-dessus, les topics de toutes les instances présentes dans la structure de l'Endpoint sont automatisés et formés à la création des instances, la classe topic possède l'implémentation des méthodes permettant cela pour les Objets, les Nœuds et les Attributs.

5.3.2. Classe Message

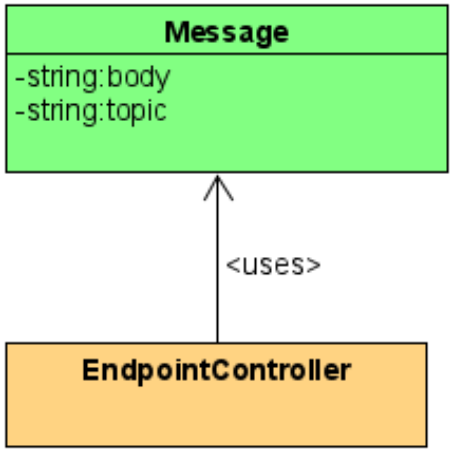


Figure 15: Message

Cette classe est utilisée pour abstraire le format des messages envoyés et reçus par la classe EndpointController. Par exemple, la classe QMqttClient de QT permet la réception de message avec une instance d'un QByteArray pour les données et un QMqttTopicName pour le topic, comme il ne doit pas avoir de référence à l'environnement de QT dans la classe EndpointController, le message reçu doit être convertit en une instance de la classe Message que l'application sait utiliser.

Elle permet aussi de stocker les deux informations (un topic et une donnée) qu'un message possède dans une liste simple.

La classe message comporte deux attributs de type string pour mémoriser le topic et les données.

5.3.3. Classe json

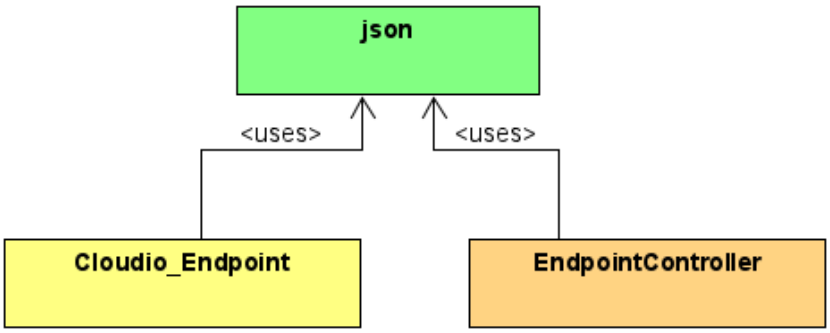


Figure 16: json

Cette classe est définie dans un projet open source, il permet la sérialisation de la structure de notre Endpoint en JSON et la désérialisation de document JSON. Ceci facilite l'implémentation et le traitement des messages pour la communication de données avec le serveur. Ce projet nous permet aussi d'utiliser la sérialisation/désérialisation binaire avec CBOR pour diminuer la quantité de données envoyées.

Vous retrouverez la référence au projet open source de Nlohmann qui implémente la classe json utilisée dans le projet dans le chapitre 10.4.

5.4. Paquet Interface

5.4.1. Interface I_Logger

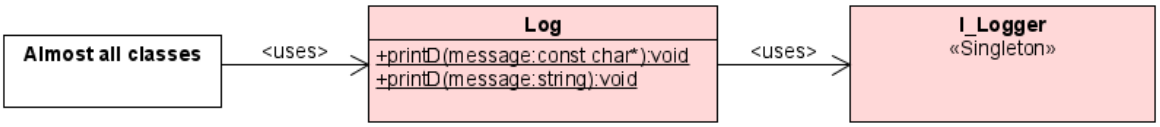


Figure 17:I_Logger

Cette interface a permis de créer une Abstraction Layer pour les fonctions « print » de l'environnement donc forcer la classe enfant à implémenter des méthodes servant à écrire des données sur une console, il fut utilisé pour déboguer l'application sur l'environnement QT, il pourra être utilisé pour transmettre des messages d'erreurs sur le serveur sur le topic @log dans une implémentation ultérieure dans un système embarqué.

Il indique aussi que la classe enfant doit être un singleton et qu'elle doit implémenter sa fonction getInstance().

Pour éviter la redondance de devoir écrire « I_Logger::getInstance()->Print() », la classe Log fut ajoutée, dorénavant il suffit d'écrire Log::printD().

5.4.2. Interface I_Mqtt

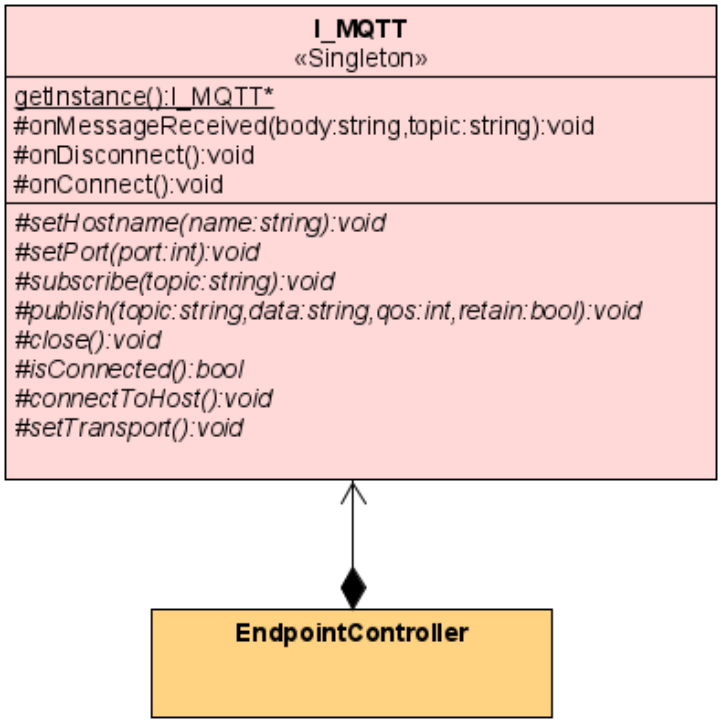


Figure 18: I_MQTT

Cette interface fournit au EndpointController les méthodes pour se connecter au serveur et utiliser les fonctionnalités de bases qu'une classe MQTT client possède généralement.

Ainsi, des méthodes pour ajouter le nom de l'hôte (nom DNS ou adresse IP du serveur), le numéro de port, pour s'abonner à des topics ou pour publier sur des topics par exemple ont été mis à disposition.

Ainsi, l'EndpointController peut grâce à cette interface :

- Informer le nom de l'hôte (nom DNS ou adresse IP du serveur)
- Informer le numéro de port à utiliser
- Se souscrire à un topic
- Publier sur un topic
- Tenter une connexion au serveur
- Fermer la connexion
- Ajouter un moyen de transport (ici : SSL)

Dans la situation du projet actuel, l'EndpointController n'utilise pas toutes les fonctionnalités citées ci-dessus, en effet il est inutile d'informer le nom de l'hôte et le numéro de port, les

fonctions sont cependant gardées dans le cas où des modifications subviendraient pour une meilleure portabilité.

Les méthodes citées ci-dessus sont virtuelles, elles doivent être définis dans une autre classe où l'on utilisera les spécificités de l'environnement, elle va aussi signaler la classe qui l'implémente à être un singleton avec sa méthode getInstance().

Cette interface implémente aussi des méthodes pour signaler à l'EndpointController que la connexion est établie, qu'une déconnexion est établie et pour transmettre un message du serveur.

La communication avec le serveur se fait via le protocole SSL, or il n'y a pas de « Abstraction Layer » dédié à SSL, ceci est dû à la forte différence qu'il y a de cette utilisation entre les deux environnements explorés dans ce projet (QT et nRF).

Pour le moment, il faudra donc implémenter la méthode pour le moyen de transport de manière à ce qu'elle initialise un socket utilisant le SSL, le projet fait sur QT montre une manière de faire. Pour le nRF, il faudra juste écrire les certificats au modem et sélectionner le bon tag, pour plus d'explications sur ces sujets, un regard dans le code devrait suffire.

5.4.3. Interface I_FileManager

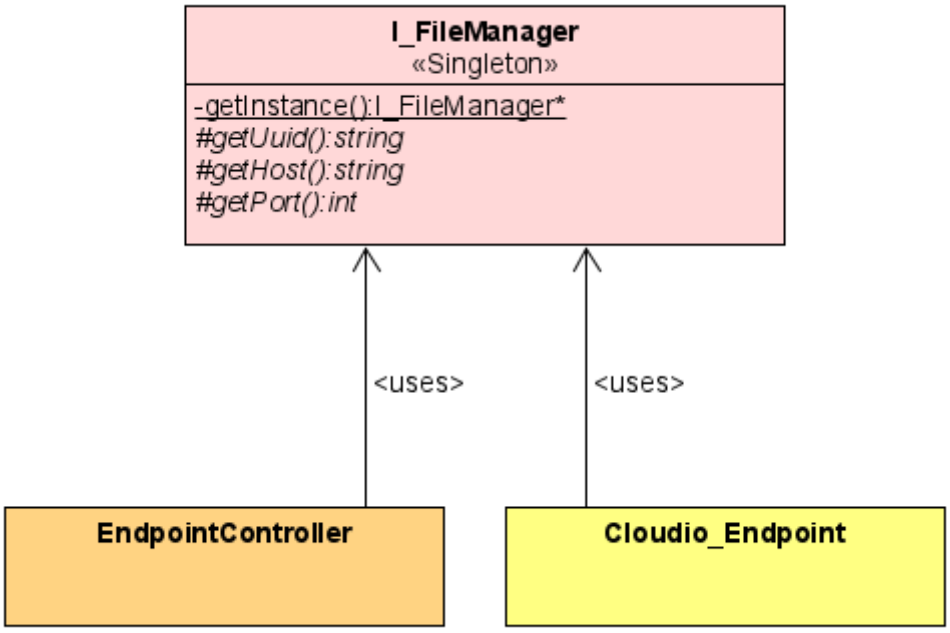


Figure 19: I_FileManager

Dans le projet JAVA, les différentes informations nécessaires à l'application étaient fournies via un keystore (magasin de clé, fichier au format .jks), une clé privée et un fichier avec les différentes propriétés de bases (l'UUID du Endpoint, l'URI du serveur, le numéro de port du serveur, mot de passe pour le fichier .jks, etc…).

Le keystore ne peut pas être utilisé dans notre projet en C++, ainsi les trois fichiers nécessaires pour le protocole SSL (le certificat d'autorité, la clé privée et son propre certificat) doivent être fournis dans l'application ainsi que le fichier avec les propriétés de bases.

L'interface I_FileManager a pour but de pouvoir lire ces fichiers et d'extraire les informations.

Il impose à la classe enfant d'implémenter des méthodes pour extraire le numéro de port, l'adresse du serveur et l'UUID de l'Endpoint du fichier propriétés. Il signale aussi la classe enfant à être un singleton.

Comme expliqué au point précédent pour I_Mqtt, il n'y a pas d'abstraction layer pour formaliser le transport SSL, ainsi, la classe I_FileManager ne s'occupe pour le moment pas des certificats, c'est une classe dans le package Core de l'environnement qui le fait.

5.4.4. Interface ResourceFactory, Thread, Mutex

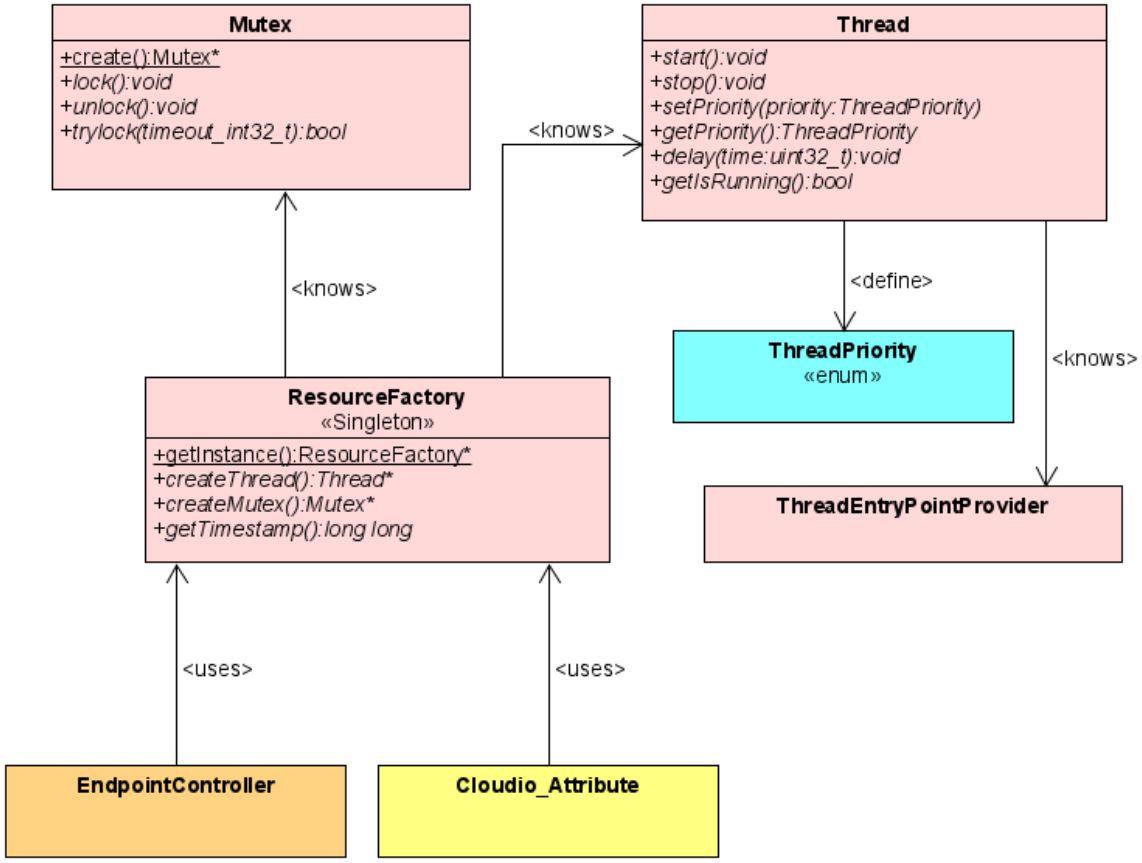


Figure 20:ResourceFactory, Mutex, Thread

Ces interfaces ont été reprises d'un ancien projet, ainsi il n'y a pas eu d'effort à les implémenter et à les tester. La classe ResourceFactory propose la création de Thread et de Mutex qui peuvent être ensuite utilisées dans l'application.

Une méthode a été ajouté dans ResourceFactory pour pouvoir récupérer le timestamp, ceci sera utilisé lors de la modification de la valeur d'un attribut.

L'application fut pensée pour effectuer en pseudo-parallèle trois tâches distinctes :

- L'envoi des messages au serveur ou leur mémorisation si l'Endpoint n'est pas connecté
- La réception des messages en provenance du serveur

- Le poll des capteurs ou la gestion des interruptions programmés par l'utilisateur

A savoir que l'envoi des messages s'occupait aussi de tenter de se connecter au serveur en cas de déconnexion.

Les Threads ont besoin de communiquer entre eux les différents messages et aussi le statut de l'application (offline, online), les messages sont donc placés dans des queues dont l'accès est protégé (tout comme le statut) par des Mutex.

Une dernière chose à signaler est la classe ThreadEntryPointProvider. Un thread doit avoir une fonction à exécuter périodiquement, pour que le thread exécute une méthode d'une instance précise, il lui faut un pointeur de cette instance. Ainsi, ThreadEntryPointProvider permet à la classe qui l'hérite de pouvoir donner la méthode à exécuter au Thread.

5.5. Paquet EndpointController

5.5.1. Classe EndpointFactory

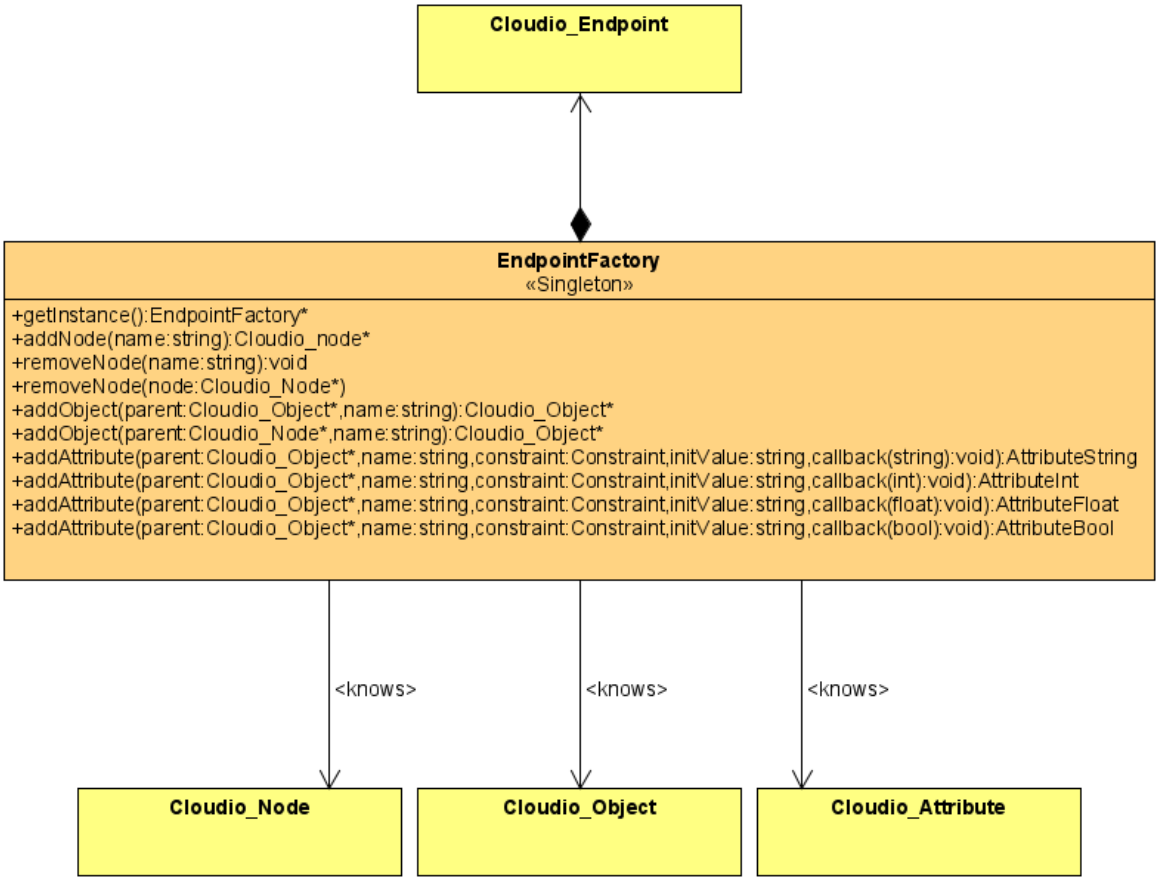


Figure 21: EndpointFactory

Cette classe est un singleton et permet la création d'instances des différentes classes du paquet EndpointStructure.

Ainsi, cette classe propose les méthodes pour créer des Nœuds, des Objets et des Attributs, tout en faisant en sorte qu'aucun problème particulier ne puisse arriver comme la création de deux nœuds ayant le même nom, ou alors la création d'un Nœud n'ayant pas de référence à son parent Cloudio_Endpoint. Utiliser cette classe permet l'automatisation de la structure du Endpoint avec le plus de simplicité possible pour permettre à l'utilisateur une prise en main plus rapide et efficace.

5.5.2. Classe EndpointController

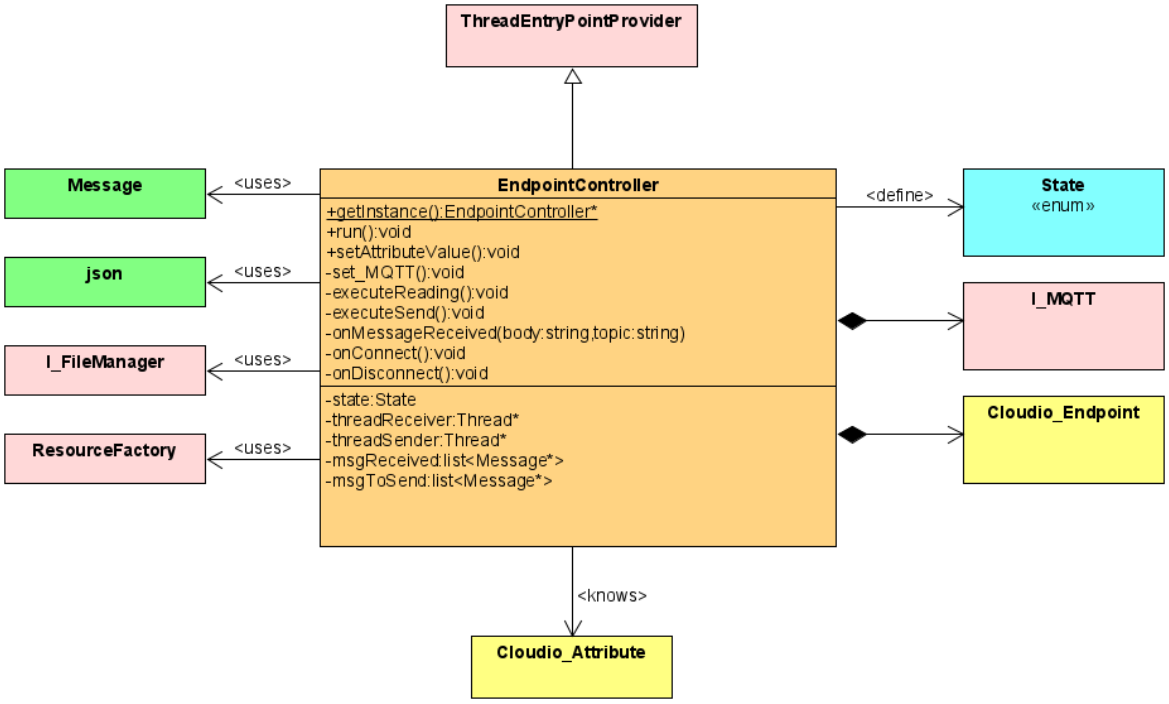


Figure 22: EndpointController

Cette classe, la dernière classe présentée, est la classe la plus importante. Cette classe est un singleton.

Elle permet la mise en route de l'application au moyen d'une seule méthode : la méthode run(). Cette méthode doit être appelée une fois que la structure de l'Endpoint soit entièrement construite.

Les rôles de cette classe sont :

- Se connecter au serveur et gérer le statut de la connexion
- Permettre l'envoi de données au serveur ou leur mémorisation
- Récupérer et traiter les messages envoyés par le serveur
- Permettre à l'utilisateur des méthodes pour modifier la valeur d'un Attribut

Ces rôles sont répartis dans les trois Threads de notre application, ceci est expliqué plus bas.

Ce singleton possède les attributs suivants :

- Une variable possédant l'état de l'application (dépend surtout du statut de la connexion)
- Une référence vers le singleton de Cloudio_Endpoint (pour récupérer des informations sur la structure de l'Endpoint et pour pouvoir atteindre les Attributs à modifier)
- Deux variables Thread dont le rôle est expliqué plus bas
- Une référence vers le client MQTT pour pouvoir envoyer/recevoir des messages
- Une liste pour les messages reçus, une liste pour les messages à envoyer au serveur
- Des mutex pour protéger les sections critiques (la variable state et les listes qui sont utilisés par les différents Threads)

Le premier Thread s'occupe des deux premiers rôles : la connexion et l'envoi des messages, pour cela il exécute continuellement la méthode executeSend().

Si l'Endpoint est déconnecté, il cherchera à se connecter. Les messages à envoyer restent stockés dans la liste.

Lorsque l'Endpoint est connecté, il enverra un premier message sur le topic @online. Ce message indique au serveur la structure de l'Endpoint, il servira donc de schéma avec lequel le serveur pourra créer les topics pour les Attributs (@update) et indiquer à l'utilisateur de cloud.iO les Attributs qu'il peut modifier sur le Endpoint.

Après, tant que la connexion est établie, le Thread enverra les messages présents dans la liste de message à envoyer.

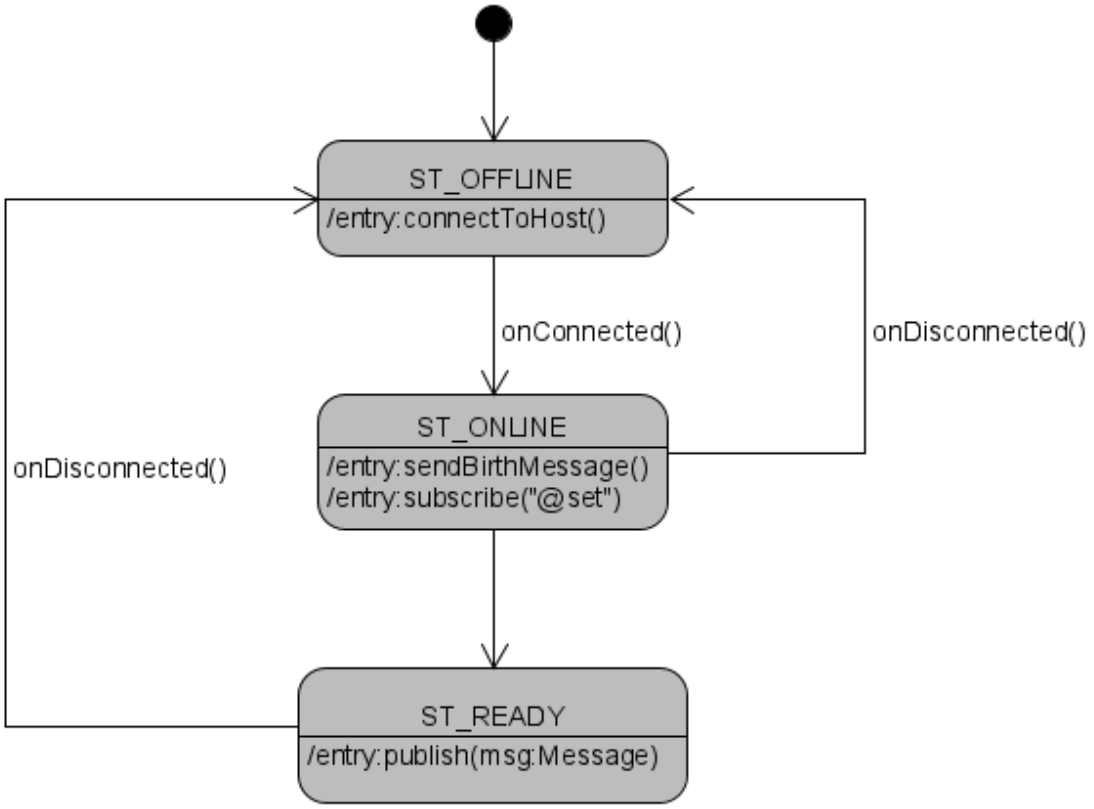


Figure 23: Activité du Thread

Le deuxième Thread s'occupe des messages reçus, les messages sont placés dans une liste grâce à une méthode de I_MQTT, le Thread ainsi traite le message reçu.

Si la liste est vide, le Thread est arrêté. Il sera réveillé lors du prochain message. Ceci fut pensé comme cela car l'Endpoint ne va généralement pas recevoir constamment des messages en provenance du serveur.

Le troisième et dernier Thread n'est pas implémenté dans EndpointController, il s'agit du Thread créer par l'utilisateur dans la fonction main() où il récupèrera les différentes informations du système. Il utilisera quand il voudra la méthode proposée par l'EndpointController pour changer les valeurs des Attributs qu'il aura créé, ce qui génère automatiquement un message que le premier Thread enverra au serveur.

5.6. Fonction main

Avec notre API, la création d'un Endpoint est facilitée pour l'utilisateur. Voici un exemple :

```
void callback_test(int i);
void mainThread();

int main(int argc, char **argv){

    EndpointFactory* ef = EndpointFactory::getInstance();

    Cloudio_Node* node1 = ef->addNode("Node1");
    Cloudio_Object* object3 = ef->addObject(node1,"object3");
    Cloudio_Object* object1 = ef->addObject(node1,"object1");
    Cloudio_Object* object2 = ef->addObject(node1,"object2");
    Cloudio_Object* object1_1 = ef->addObject(object1,"object1_1");
    AttributeInt* attribute1 = ef->addAttribute(object1, "Attribute1",Constraint::SetPoint,0, callback_test);
    AttributeInt* attribute1_1 = ef->addAttribute(object1_1, "Attribute 1.1",Constraint::Parameter,0, callback_test);
    AttributeInt* attribute20 = ef->addAttribute(object2, "Attribute 20",Constraint::Measure,0);
    AttributeInt* attribute3 = ef->addAttribute(object3, "Attribute 3",Constraint::StaticAttribute,0);
    AttributeInt* attribute21 = ef->addAttribute(object2, "Attribute 21",Constraint::SetPoint,0, callback_test);

    Cloudio_Node* node2 = ef->addNode("Node2");
    Cloudio_Object* object4 = ef->addObject(node2,"object4");
    Cloudio_Object* object5 = ef->addObject(node2,"object5");
    AttributeInt* attribute4 = ef->addAttribute(object4, "Attribute 4",Constraint::SetPoint,0, callback_test);

    EndpointController* ec = EndpointController::getInstance();
    ec->run();
    mainThread();
}

void mainThread(){
    while(1){
        //For example launch a timer and poll the measurment when timeout, or put a delay function
        //If the measurement is different enough from the last,
        //Call function ec->setAttributeValue(...), it will send the change to the server
    }

}

void callback_test(int i){
    //What to do when the value of the attribute changes
    Log::printD("Callback sent");
}
```

Figure 24: Exemple d'une fonction main()

Ce qui donnerait la structure suivante :

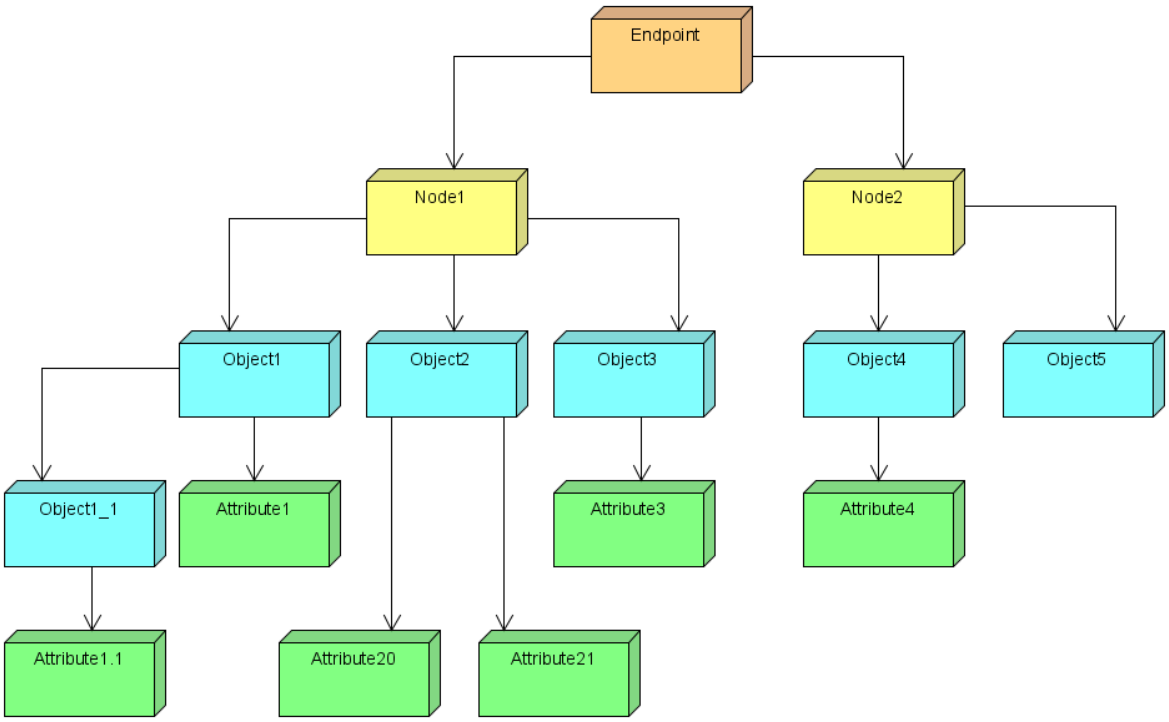


Figure 25: Structure de l'Endpoint crée

Les seuls points à respecter sont les suivants : après configuration des différents composants de la cible, la structure du Endpoint doit être fait en premier, après, l'utilisateur de LTE cloud.iO doit appeler la méthode run() pour lancer la routine d'exécution du EndpointController, ensuite il peut s'occuper librement des mesures à faire.

5.7. Remarques sur les classes de bases

Pour la classe Cloudio_Attribute, le code est redondant entre les classes enfants. L'utilisation d'un attribut Template serait plus adapté pour la valeur.

L'interface I_FileManager était prévue à la base pour suivre en partie le projet JAVA et donc utiliser le fichier propriétés, cependant, cette interface demande qu'il y ait dans l'environnement un gestionnaire de fichier et l'implémentation de méthodes pour extraire les différentes informations.

Il semble donc que cette classe n'a pas lieu d'être dans de futurs versions, un fichier .h qui possèdent toutes les informations dans des #define serait beaucoup plus simple et appréciable. Cela faciliterait la transportabilité.

Plusieurs fonctionnalités présentes dans le Endpoint JAVA n'ont pas été implémenté. Un objet peut posséder une option « conforms », un nœud doit posséder un tableau d'option « implements », la signification de ces deux éléments n'a pas été trouvée dans la documentation, ainsi le Nœud possède ce tableau pour des raisons de compatibilité avec cloud.iO mais n'est pas configurable pour l'utilisateur, la même chose équivaut pour les Objets.

La « persistency » présente dans le fichier properties n'a pas été implémenté.

La dernière implémentation qui manque est la configuration de la queue qui contient les messages à envoyer au serveur. Si l'Endpoint est déconnecté, alors la liste retient les cinq derniers messages. Il serait intéressant d'implémenter des options pour cette liste pour par exemple définir le nombre de messages à retenir pour chaque Attributs différent et de former des messages contenant la valeur moyenne des Attributs des messages qui ont été effacés.

6. Implémentation des Interfaces

Un premier projet sur QT en C++ fut créé pour permettre une implémentation plus rapide des classes de bases. Ceci permettra aussi d'avoir une première vision de la transportabilité du projet quand le portage sera fait sur le nRF9160 et d'effectuer les modifications requises des interfaces.

6.1. Core-QT

Pour pouvoir garantir aux classes de base les fonctionnalités promises, les classes de QT suivantes ont été utilisées :

- QFile (FileManager_qt)
- La méthode qDebug() (Logger_qt)
- QMutex (mutex-default)
- QThread (thread-default)
- La méthode QDateTime::currentMSecsSinceEpoch() (ResourceFactory pour le timestamp)

Pour MQTT_qt, la classe QMqttClient a été utilisé, cette classe utilise une particularité de QT : les signaux. Une classe SSL_qt fut aussi créé, elle instancie et paramètre un QSslSocket que notre QMqttClient utilisera comme transport. Ce QSslSocket est d'ailleurs aussi un QObject.

Le signal/slot de QT propose une solution alternative pour les callbacks. Ainsi, il n'y a pas besoin lors de la création d'un QMqttClient de devoir lui fournir tous les callbacks donc des pointeurs de fonctions pour pouvoir utiliser correctement l'instance de QMqttClient, tout cela est abstrait dans l'utilisation de signal et de slot.

Cependant, pour le projet, l'utilisation de signal/slot cause des problèmes à la structure du projet.

Premièrement, la classe MQTT_qt devra hériter de QObject pour pouvoir utiliser les slots pour les lier aux signaux de QMqttClient, comme MQTT_qt doit hériter aussi de I_MQTT, alors soit I_MQTT doit hériter de QObject (ce qui serait une mauvaise implémentation), soit elle doit utiliser la macro Q_DECLARE_INTERACE de QT, mais I_MQTT ne doit posséder aucune relation avec l'environnement QT.

Ce problème fut réglé par l'utilisation d'un #if QT_PROJECT juste devant la macro, QT_PROJECT est tout simplement un #define que l'on peut modifier.

Deuxièmement, l'utilisation d'objets QObject n'est pas d'une simplicité dérisoire avec l'utilisation de Threads. Cette utilisation nécessite certainement l'utilisation de pattern assez strict pour que Thread et QObject coexistent sans perturber le fonctionnement des signaux et des slots.

Ce problème étant survenu à la fin du projet, aucune bonne solution utilisant les QObject en même temps que les Threads ne fut mis en place. A la place, le projet QT ne met plus de Thread en exécution. Une autre solution serait l'utilisation d'une autre librairie comme Paho qui propose un MqttClient qui demande un callback qui traite les différents évènements qui peuvent survenir.

Troisièmement, pour le bon fonctionnement d'un QObject, il faut l'utilisation dans la fonction main d'une « QApplication », ceci enlève la possibilité pour l'utilisateur de faire une boucle infinie dans la fonction main, d'ailleurs utiliser une boucle infinie dans un projet utilisant le signal/slot de QT est une mauvaise idée, il devra utiliser des QObject comme des QTimer pour pouvoir poll les mesures en un certain laps de temps.

6.2. Core-nRF

Les différentes classes n'ont pas pu être implémentées dans le nRF, cependant on peut tenter de voir si l'environnement possède déjà de bons outils pour une future implémentation.

- Pour l'interface I_logger, la fonction printk() fait parfaitement l'affaire, cette fonction envoie les données à travers le câble USB à l'ordinateur, le débogage avec l'utilisation de cette fonction est possible.
- Pour les threads et les mutex, Zéphyr propose la structure k_thread et k_mutex
- Pour le timestamp, il existe une documentation sur Date_Time sur le site de Nordic, à tester, autrement Zéphyr possède une bonne librairie sur les Clocks du Kernel et leur utilisation, on peut facilement connaître le temps écoulé après que l'application fut lancée, mais il faut connaître l'heure à laquelle l'application fut lancée.
- Pour I_FileManager, on peut trouver une documentation de Zephyr, on peut utiliser par exemple la fonction fs_read(fs_file_t, size_t), cependant, comme signalé au point 5.7., il serait plus intéressant d'abandonner l'implémentation de cette interface.
- Pour I_MQTT, la structure mqtt_client existe, l'exemple Mqtt_Simple fournis par Nordic présente déjà une bonne structure de base sur lequel on peut travailler et propose même un lien pour pouvoir utiliser le protocole SSL. Plus d'informations sur ce sujet seront présentés dans le point 7.1.

Les références aux structures présentées sont présentes dans le chapitre 10.

Cependant, la fusion des classes de bases du projet (C++) et des librairies Zéphyr (C) restent à être vérifiée, des problèmes de compatibilités peuvent survenir. Des échos aussi disent que de réussir à build un projet conséquent en C++ dans le nRF avec l'outil west n'est pas une chose aisée.

7. Travaux sur le nRF

Créer un Endpoint sur le nRF sans vérifier et étudier ses fonctionnalités n'auraient pas été une bonne idée, sans cela, il aurait été difficile de trouver quelles sont les problèmes lors du débogage : utiliser un client MQTT qui doit utiliser le protocole SSL pour communiquer avec un serveur de manière à ce que les messages soient compatibles, en ajoutant à cela d'éventuels petits problèmes d'implémentation de toutes les autres classes de base, le débogage aurait été un immense défi.

Pour éviter cela, des petites applications ont été testés pour assurer la bonne fonctionnalité de la communication au travers du réseau mobile. Les autres petits problèmes d'implémentations auront normalement disparu avec les tests effectués sur QT.

7.1. Fonctionnalités testées

- Connexion au réseau mobile et envoi de données au serveur nrfcloud.com grâce à l'exemple asset_tracker. Pour cela, un upgrade du firmware du modem et un ajout de certificats au modem ont dû être effectués.
- Envoi de message à un broker de test (Mosquito) grâce à l'exemple Mqtt_Simple qui utilise la structure mqtt_client.
- Envoi de message au même broker de test avec le protocole SSL, les certificats sont transférés au modem avant le lancement de l'application grâce au logiciel LTE Link Monitor proposé par Nordic
- Envoi de message au même broker de test avec le protocole SSL en fournissant les certificats au lancement de l'application, avant que le modem ne soit connecté au réseau
- Envoi de message sur le serveur cloudIO avec la structure mqtt_client en utilisant le protocole SSL

7.2. LTE-M et NB-IoT

Comme expliqué dans le chapitre 3, on s'attend à ce que LTE-M soit plus rapide que NB-IoT, cependant, la pratique ne correspond pas toujours à la théorie.

En effet, plusieurs évènements peuvent perturber le transfert de données.

Ces tests ont été effectué sur le nRF9160-DK, la mise en place des tests étaient le suivant :

- Utilisation d'un broker de test (https://test.mosquitto.org), port 1883, connexion sans SSL
- Se souscrire et publier sur le même topic
- Envoi d'une chaîne de caractères (8bits par caractère)
- Chronométrer le temps entre l'envoi et la réception des données
- Exécuter les deux derniers points plusieurs fois (200 messages publiés)

Les tests ont été effectué trois fois dans le bâtiment de la HES-SO Valais. Pour 6 caractères envoyés, nous obtenons :

Heure	Temps moyen :NB-IoT [ms]	Temps moyen :LTE-M [ms]
8h30	234ms	310ms
11h	324ms	650ms
15h	428ms	785ms

Pour 128 caractères :

Heure	Temps moyen :NB-IoT [ms]	Temps moyen :LTE-M [ms]
8h45	1344ms	1357ms
11h30	1718ms	2256ms
15h	1915ms	2430ms

Nous remarquons donc que NB-IoT semble plus performant pour le lieu indiqué, cependant ceci n'est pas dû à la technologie mais au réseau mobile.

En effet, lors des tests, le système embarqué se connectait au réseau mobile de Sunrise quand il utilisait NB-IoT et il utilisait le réseau de Swisscom avec LTE-M. Ainsi la comparaison des technologies avec ce test n'a pas lieu d'être.
Le réseau mobile aussi semble plus saturé selon l'heure de la journée, de petits tests ont été effectué pour montrer que ce n'est pas le broker Mosquito qui est saturé avec un réseau Wifi.

Une autre différence notable est le temps que met notre système embarqué pour se connecter au réseau : en utilisant LTE-M, seul quelques secondes suffisent et le système est prêt pour l'envoi des données, cependant pour NB-IoT, il faut en moyenne plus de onze minutes pour qu'il puisse trouver une bonne configuration !

En conclusion, même si LTE-M est une technologie supérieure en théorie en termes de quantité de données à envoyer, en pratique, cela n'a pas beaucoup d'importance, il faut tout d'abord s'assurer de la qualité du fournisseur de réseau mobile avant de penser à la technologie à utiliser.

8. Conclusion

Avant d'être totalement exploitable, le LTE cloud.iO a besoin d'un peu plus de temps de développement, les points manquants sont cités dans le chapitre 5.7. et les éléments pour l'implémentation de l'application sur le nRF sont cités dans le chapitre 6.2.

Cependant, le travail qu'il reste à faire pour le code de base est faible et les travaux sur le nRF possèdent tous les éléments liés à la connectivité que les classes de bases vont utiliser, ainsi une grande partie du travail est effectuée.

Pour les éléments implémentés, l'API est fonctionnelle et facilite le travail de manière conséquente pour un développeur d'Endpoint sur cloud.iO.

Ainsi, le projet sur QT communique bel et bien avec le serveur, il peut transférer les données et modifier ses paramètres selon la volonté de l'utilisateur de cloud.iO.

Le nRF9160 peut lui aussi envoyer les valeurs de ses attributs au serveur et recevoir les requêtes de ce dernier.

Prochainement, ce projet recevra quelques petites modifications pour combler les lacunes signalées dans le chapitre 5.7. Ensuite une démonstration sera faite sur les deux environnements.

Comme signalé au dernier paragraphe du chapitre 6.2., si la compatibilité du projet C++ avec l'outil west de Zephyr pour programmer le nRF est compliquée à résoudre, alors une démonstration basique (sans automatisation des messages) sera tout de même effectuée.

9. Remerciements

Pour terminer, je tiens à remercier M. Gabioud Dominique pour m'avoir suivi sur ce projet mais également pour m'avoir soutenu et conseillé à diverses reprises.

Je remercie également M. Clausen Michaël pour ses aides et conseils sur divers sujets ainsi que M. Pannatier Valentin pour m'avoir initié à l'utilisation des requêtes REST de cloud.iO avec le logiciel Insomnia.

10. Liens et références

10.1. Cloud.iO et Endpoint en JAVA

Travail de master de Bonvin Lucas, 2020, “Low-complexity scalable IoT framework”, https://github.com/cloudio-project/cloudio-documents/tree/master/MasterThesis%20Scalable%20IoT%20Framework

Installation du serveur cloud.iO : https://github.com/cloudio-project

Endpoint réalisé en JAVA : https://github.com/lucblender/cloudio-endpointHeater-demo

Site internet de cloud.iO : http://cloudio.hevs.ch/

10.2. Liens utiles pour le développement sur le nRF

Présentation du nRF9160 ainsi que lien vers le téléchargement du dernier firmware de modem : https://www.nordicsemi.com/Software-and-tools/Development-Kits/nRF9160-DK/Download#infotabs

Présentation des principales parties pour mieux comprendre l’architecture du nRF : https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/ug_nrf9160.html

Présentation des tags de sécurité : https://developer.nordicsemi.com/nRF_Connect_SDK/doc/1.3.0/nrfxlib/bsdlib/doc/security_tags.html

BSD Socket et TLS credentials subsystem de Zephyr : https://docs.zephyrproject.org/latest/reference/networking/sockets.html

Exemple Asset-tracker : https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/applications/asset_tracker/README.html

Exemple Simple MQTT :

https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/samples/nrf9160/mqtt_simple/README.html

10.3. Lien à suivre pour l’implémentation de l’API sur le nRF

Thread sur Zephyr : https://docs.zephyrproject.org/latest/reference/kernel/threads/index.html

Basic Thread Example :
https://docs.zephyrproject.org/latest/samples/basic/threads/README.html

Mutex sur Zephyr :
https://docs.zephyrproject.org/latest/reference/kernel/synchronization/mutexes.html

File Systems sur Zephyr :
https://docs.zephyrproject.org/latest/reference/file_system/index.html

Date-Time sur le nRF :
https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/include/date_time.html

10.4. Autres liens

Requêtes REST pour cloud.iO :
https://app.swaggerhub.com/apis/luc_blender/cloud.iO_2.0/1.0.0

Installation de Ubuntu server : https://ubuntu.com/tutorials/install-ubuntu-server#1-overview

Installation de Docker sur Ubuntu : https://docs.docker.com/engine/install/ubuntu/

Projet Nlohmann json : https://github.com/nlohmann/json

11. Annexe

A. Installation de l'environnement pour le nRF

1. *Téléchargement*

Product Name	Description	lien
nRF Connect	Cross-platfrom development software	link
Modem Firmware	Micrologiciel pour le modem	link
Librairies	Librairies sur github	link

L'installation de nRF Connect permettra aussi l'installation du logiciel J-Link pour pouvoir réussir à connecter votre système embarqué à votre ordinateur avec un câble USB.

2. *Avant de commencer*

Notez les informations suivantes sur un fichier .txt : ICCID et PUK de votre carte SIM, Device ID (IMEI) et HWID de votre système embarqué.

3. *nRF Connect*

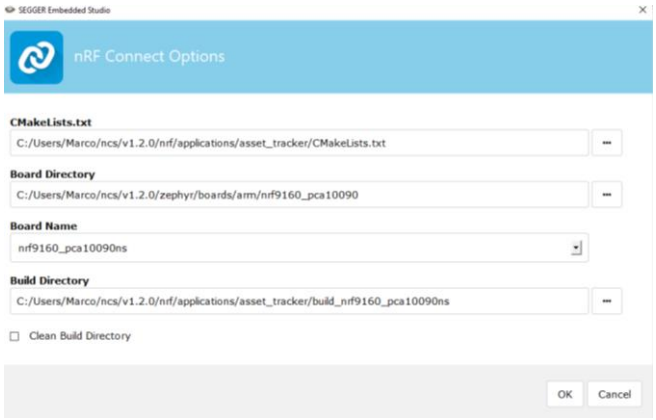
Dans les applications de nRF Connect, installez LTE Link Monitor, Programmer, Toolchain Manager.

Lancez Toolchain Manager et installez l'un des produits indiqués, ceci vous permettra ensuite de pouvoir accéder à l'IDE Segger.

Après avoir branché votre système embarqué à votre ordinateur, ouvrez Programmer, sélectionnez votre appareil en haut à gauche puis cliquer sur Update Modem en bas à droite, sélectionnez le fichier compressé Modem Firmware que vous avez téléchargé.

4. *Lancement de asset_tracker*

En lançant l'IDE que vous avez choisis dans Toolchain Manager, ouvrez l'application asset-tracker en cliquant sur File->Open nRF Connect SDK Project, configurez le projet comme le montre la figure suivante.



Programmez ensuite votre système embarqué en sélectionnant Build->Build and Run.

Lors de vos premiers projets lancés sur le SDK, il peut y avoir quelques soucis avec les fichiers inclus, ils peuvent être inexistants dans le repertoire installé par le toolchain manager, ces fichiers se retrouvent dans les librairies github.

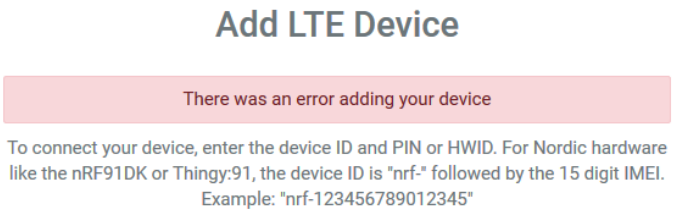
5. *LTE Link Monitor*

Lancez l'application LTE Link Monitor, sélectionnez votre appareil en haut à gauche. Appuyez sur le bouton reset présent sur votre système embarqué, contrôlez le statut de votre appareil en sélectionnant la commande AT+CFUN ? de LTE Link Monitor.

6. *nRF Cloud*

Allez sur https://nrfcloud.com/, créez un compte. Vous vous retrouverez alors sur votre dashboard. Vous pourrez ajouter votre appareil et finalisez la connexion.

Il se peut que vous ne puissiez pas enregistrer votre appareil, si tel est le cas, il s'agit peut-être d'une erreur de certificat. Essayez ceci.



B. Installation du serveur cloud.iO

Le serveur fut installé sur une machine virtuelle Ubuntu server, le logiciel VMware fut utilisé pour faire tourner la machine virtuelle sur windows.

Ubuntu server fut installé avec les paramètres de base avec l'option ssh (cela peut toujours être utile), l'installation de Mosquito peut aussi servir.

Ensuite, avant d'essayer de mettre en place le serveur, nous avons besoin de :

- Make

  Sudo apt-get install build-essential

- Keytool

  Sudo apt install openjdk-11-jre-headless

Ensuite, il est temps d'installer Docker Engine sur Ubuntu :

https://docs.docker.com/engine/install/ubuntu/

N'oubliez pas de faire la partie pour créer le groupe pour Docker (lancement directement de depuis l'utilisateur root)

Vous pouvez git clone le projet suivant : https://github.com/cloudio-project/cloudio-deployment-examples.git , n'hésitez pas à consulter la documentation pour plus d'informations.

A savoir que pour le bon fonctionnement, n'hésitez pas à faire la commande ./stop.sh suivit de ./start.sh après avoir exécuter un ./up.sh, cela peut régler certains inconvénients.

Pour obtenir le mot de passe d'administrateur pour les requêtes REST sur le serveur cloud.iO :

```
docker-compose logs services | grep WARN
```

La commande docker-compose logs possède les logs de tous les containers. Il pourra être utile de voir les connexions entrantes au serveur par exemple en utilisant par exemple :

```
docker-compose logs rabbit
```

Pour éviter certains problèmes, vous pouvez vous log en root avec la commande -sudo -i.

Autres problèmes rencontrés à la base :

https://forums.docker.com/t/can-not-stop-docker-container-permission-denied-error/41142/3

Solution :

```
sudo apt-get purge --auto-remove apparmor
sudo service docker restart
docker system prune --all –volumes
```

C. Création et lancement d'un Endpoint

Vous trouverez le mot de passe d'administrateur lors du lancement des différents containers et donc de RabbitMQ, veuillez-vous référer à l'annexe A. Vous pourrez alors faire les requêtes REST suivant au serveur :

{{ base_url }}/api/v1/endpoints //Création d'un endpoint, récupération de l'UUID

{{ base_url }}/api/v1/getCaCertificate //Récupération du certificat d'autorité de cloud.io

{{ base_url }}/api/v1/createCertificateAndKeyAsPEM?endpointUuid=<<UUID>>
//Récupération du certificat client et de la clé privée

Ensuite, modifier le fichier properties de l'API (avec le nouvel UUID), remplacer le certificat d'autorité, la clé privée et le certificat client.

Par défaut, le nom du Peer du certificat d'autorité est « rabbit », si cela venait à changer, utiliser la commande :

```
openssl s_client -host <<hostname>> -port 8883 -showcerts
```

Vous devriez pouvoir retrouver le CN.

Ensuite vous pourrez lancer le Endpoint. Vérifier si la connexion SSL est établie, soit dans une console, soit avec les logs rabbit du serveur.

Si tel est le cas, vérifiez que la structure du endpoint est présent dans les bases de données avec la requête REST :

{{ base_url }}/api/v1/endpoints/<<UUID>>

Si vous ne trouvez rien, trouver des traces dans les logs du serveur, cela pourrait venir d'une incompatibilité avec la construction de votre Endpoint, les logs possèderont une erreur que vous pouvez « grep » avec comme terme « fasterxml.jackson.databind ».