Projet de Semestre 5

Rapport

Connectez votre machine à café à l'Internet (des Objets)

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Valentin Chassot et Michaël Berchier

Télécommunication, Classe T3-F

Fribourg, Automne 2015-2016

valentin.chassot@edu.hefr.ch
michael.berchier@edu.hefr.ch

Superviseurs :
Serge Ayer (serge.ayer@hefr.ch)
Patrick Gaillet (patrick.gaillet@hefr.ch)

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Table des matières

1 Historique des versions

Version	Date	Personnes	Modification
0.1	2015-10-14	V. Chassot, M. Berchier	Création du document
0.2	2015-10-20	V. Chassot M. Berchier	Introduction (chap. 2) L'Internet des objets (chap. 3.3) L'Internet classique (chap. 3.2)
0.3	2015-11-05	V. Chassot	Clarification suite à la séance (chap. 3.3.5, 3.3.6, 3.3.7, 3.3.8) Ajout des chapitres (3.3.9, 3.3.10, 3.3.11)
0.4	2015-11-11	V. Chassot M. Berchier	Analyse 6loWPAN sur BLE (3.3.6) Comparaison proxy/web server (3.3.8, 3.3.9) Mise à jour des schémas (3.3.12)
0.5	2015-11-18	V. Chassot M. Berchier	Précision (3.3.6) Analyse de MQTT (4.1) Précision (3.3.9) Architecture proxy avec db (3.3.10) Ajout des chapitres CoAP (4.2.1, 4.2.2, 4.2.3, 4.2.4)
0.6	2015-11-23	V. Chassot M. Berchier	Ajout du chapitre (4.2.8) Ajout des chapitres (4.2.5, 4.2.6, 4.2.7, 4.2.9, 4.2.10)
0.7	2015-12-04	V. Chassot M. Berchier	Ajout des chapitres (4.3 et 5.1) Précision du chapitre (4.2.10)
1.0	2016-01-26	V. Chassot M. Berchier	Ajout des chapitres (5.2, 6.1, 6.2, 6.5, 6.6, 7.1, 7.3, 7.5, 8) Ajout des chapitres (4.2.11, 4.2.12, 6.3, 6.4, 7.2, 7.4, 8)
1.01	2016-02-04	M. Berchier	Errata sur la page de titre

2 Introduction

2.1 But de ce document

Dans le cadre du projet de semestre 5 de la « Haute école d'ingénierie et d'architecture de Fribourg » (filière Télécommunication), le projet « Connectez votre machine a café à l'Internet des objet » a été attribué à Messieurs Michaël Berchier et Valentin Chassot. Le projet débute le 21 Septembre 2015 et se termine le 28 Janvier 2016. Les professeurs attitrés au projet sont Messieurs Serge Ayer et Patrick Gaillet. Le but de ce document est de présenter le travail accompli durant ce semestre.

2.2 Contexte (Rappel)

Depuis plusieurs années, l'Internet des Objets est un domaine en pleine expansion. Effectivement nous trouvons de plus en plus d'objets connectés, tel qu'un bracelet mesurant l'effort physique ou encore une corde à sauter, affichant à l'aide de LEDs le nombre de saut effectués. Ce sont là de simples exemples d'un domaine aux possibilités presque infinies.

Les prédictions actuelles annoncent plus de 50 milliards d'objets connectés dans le monde d'ici 2020. En partant du principe que tout ces objets doivent pouvoir être identifiables dans l'Internet mondial, un adressage IPv4 n'est plus du tout envisageable. C'est pourquoi nous devons nous tourner vers un adressage IPv6, afin que tous ces systèmes puissent obtenir un identifiant unique. Une architecture Internet des Objets n'est pas nécessairement IP de bout en bout, il est effectivement possible de communiquer au travers d'un gateway, qui lui est connecté à Internet, mais dans le cadre de ce projet, nous avons choisi une architecture IP de bout en bout. Il existe différentes technologies permettant l'utilisation de IPv6 pour l'Internet des Objets, de manière analogue à l'Internet classique. D'autres contraintes et possibilités sont à prendre en compte dans l'Internet des Objets tels que la consommation, la complexité, la sécurité et la connectivité.

Dans le cadre de ce projet, le but sera de mettre en œuvre un de ces objets, à savoir une machine à café. Il s'agit là d'un objet du quotidien, dont le fonctionnement reste très souvent analogique, et aux fonctions simples. La force de l'Internet des Objets est d'ajouter des fonctions pratiques, afin de simplifier le quotidien, et surtout d'en faire un membre à part entière de l'Internet, le rendant disponible en tout temps et depuis partout.

Une machine à café nécessitera toujours une interaction physique, afin de pouvoir consommer le café produit, mais le but de « connecter » cet objet se situe plutôt dans sa gestion. Bien souvent, lorsque qu'une machine commune est placée dans un bureau, le comptage des consommations s'effectue à l'aide d'une feuille, et chaque utilisateur marque d'une coche sa consommation. Il serait possible d'automatiser le comptage de ces consommations, et d'y incorporer une gestion de comptabilité.

Afin de reconnaitre à quelle personne le café doit être comptabilisé, nous pourrions reconnaitre la tasse posée sur ladite machine via un tag NFC incorporé ou simplement collé dessous.

3 Analyse de l'architecture

3.1 Introduction

La première phase d'analyse aura pour but d'étudier la possibilité d'utiliser de l'Internet classique (Ethernet-TCP/IP-HTTP) dans le cadre de l'Internet des objets. Dans un second temps, les solutions propres à l'Internet des objets seront énoncées et discutées.

3.2 L'Internet classique

L'internet classique tel que nous le connaissons, à savoir l'accès à une multitude de site web au travers d'un navigateur, repose sur plusieurs protocoles. Un premier protocole utilisé est HTTP (Hyper Text Transfer Protocol). Il s'agit du protocole de couche applicative utilisé pour la communication web. Comme son nom l'indique, son rôle de transférer des données hypertextes, du serveur jusqu'au client final qui souhaite cette ressource.

HTTP utilise TCP comme couche de transport. Effectivement, les mécanismes de TCP sont nécessaires au bon fonctionnement de HTTP car il est impératif de garantir l'intégrité des données reçues. Il ne serait pas envisageable de livrer une page web auxquelles il manquerait une partie du texte, ou lors de l'envoi de données via un formulaire, que ces dernières se retrouvent altérées.

3.2.1 Couche applicative

Le protocole de couche applicative–à savoir HTTP–fonctionne selon un modèle client-serveur, et sur la base de requêtes-réponses. Lorsqu'un client, par exemple un navigateur web souhaite obtenir une page web, il envoie une méthode GET au serveur. Le serveur reçoit cette demande, et envoie au client la page qu'il souhaite. Bien évidemment, les site web étant aujourd'hui dynamique, il arrive fréquemment que le serveur HTTP–dont le rôle est uniquement la distribution de ressources–soit couplé avec un applicatif dont le but est la génération de contenu dynamique (par exemple PHP).

Il existe différentes méthodes HTTP utilisables par le client pour communiquer avec le serveur :

- **GET** : Il s'agit de la méthode permettant de demander une ressource. Chaque fois que l'on clic sur un lien par exemple, une telle requête est produite par le navigateur web.
- **HEAD** : Cette méthode permet de demander des informations sur la ressource, mais sans recevoir la ressource en elle-même.
- **POST** : Cette méthode permet de fournir des informations au serveur, sur la base desquelles il pourra déclencher une action. Cette méthode est par exemple utilisée lors de l'envoi de formulaires.
- **OPTIONS** : Cette méthode permet de demander au serveur quelles méthodes sont disponibles.
- **TRACE** : Cette méthode demande au serveur de répondre en envoyant une copie de la requête qu'il a reçu. Cela permet par exemple de voir précisément la demande que le serveur à reçu, et d'en déduire si la requête aurait été modifiée lors du transfert.
- **PUT** : Cette méthode nécessitant généralement un accès privilégié permet d'effectuer un ajout ou une altération de ressource.
- **DELETE** : Cette méthode, également couramment protégée permet de supprimer une ressource.

Cette liste n'est pas exhaustive, il existe d'autres méthodes. Elles permettent de répondre aux demandes courantes des applications web. A noter que tous les serveurs web n'acceptent pas forcément de répondre à tous ces types de requêtes. Par exemple, il est fréquent que les serveur HTTP ne répondent pas à un TRACE, ceci afin de ne pas dévoiler des informations sensibles, comme d'éventuels proxy ou firewall traversés.

3.2.2 Couche transport

Le protocole de la couche transport utilisé est TCP (Transmission Control Protocol). Ce protocole fonctionnant en mode connecté permet d'assurer la livraison ordrée, fiable, et dépourvue d'erreur d'un flux d'octets entre différentes applications fonctionnant sur un réseau IP.

TCP fonctionne en mode connecté, c'est à dire qu'avant un quelconque transfert de données, il est nécessaire d'établir une connexion. C'est un processus durant lequel les deux partenaires vont s'entendre sur différentes options, et surtout s'assurer que l'interlocuteur soit prêt. Au terme de la discussion, la connexion sera fermée, ceci afin de s'assurer que toutes les données aient bel et bien été reçues.

Si TCP permet d'assurer une forte fiabilité, c'est au cout d'une plus forte complexité induite par le mécanisme de validation implémenté dans le protocole. Effectivement, chaque envoi nécessite une quittance en retour, ceci afin de tenir l'émetteur informé du déroulement du transfert. Dans le cas ou un segment serait perdu, le récepteur n'enverrait alors jamais de quittance. C'est grâce à cette absence de quittance que l'émetteur peut savoir quel élément à été perdu, et donc le transmettre à nouveau. A noter que la quittance n'est envoyée que si les données sont valides, ceci étant vérifié grâce à un calcul de checksum.

3.2.3 Exemple de requête

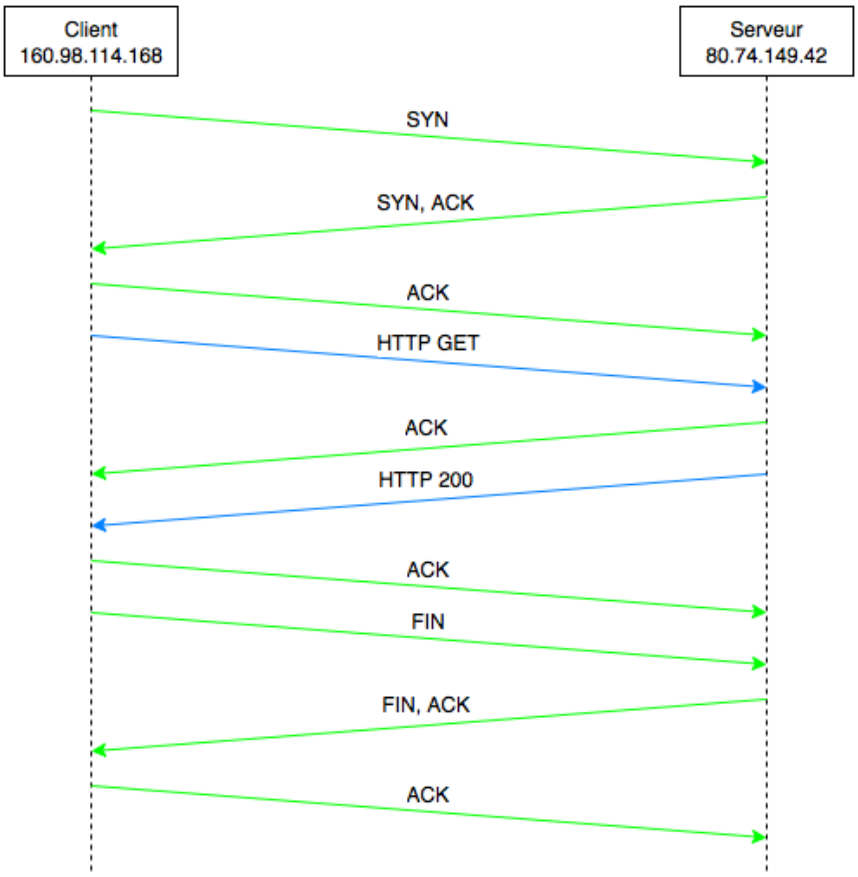

Figure 1, Echanges client-serveur pour un GET HTTP

Dans cet exemple, les segments TCP sont représentés en verts, et ceux transportant HTTP en bleus. A noter que cet exemple ne prend pas en compte les différentes requêtes DNS nécessaire à la résolution du nom de domaine en adresse IP.

3.2.4 Utilisation dans le cadre du projet

Si nous mettons en place sur l'objet un équipement implémentant la stack IP (tel qu'un Raspberry par exemple), il serait possible de réaliser une telle architecture. Cette architecture pourrait être comme représentée ci-dessous, connectée via un câble Ethernet, ou alors via une connexion sans fil 802.11 (Wifi).

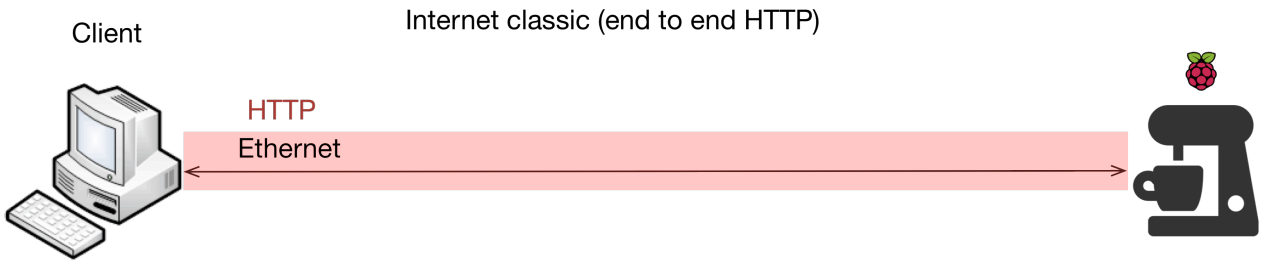

Figure 2, Internet classic (end to end HTTP)

Dans le cas concret et ciblé d'une machine à café, l'accès au réseau électrique est assuré et ne poserait donc pas de problème quand à l'alimentation du Raspberry. Nous verrons en revanche par après que si l'on souhaite généraliser cette architecture à n'importe quel objet, ce point deviendra problématique.

3.2.5 Efficacité de HTTP pour le transfert de données « courtes »

Dans le cadre de l'Internet des Objets, les communications sont souvent « petites », que ce soit en taille des données transmises ou en durée. Par exemple, la machine a café communiquerait son état, mais quelques octets de données seraient suffisant. Partant de ce constat, il est intéressant d'étudier l'efficacité de HTTP dans un tel contexte.

L'exemple de requête GET HTTP précédent peut être utilisé pour cette analyse. On constate que la partie utile de la requête HTTP GET */test* est de 20 octets. On ne tient pas compte des options type user-agent fournies par le browser, qui n'ont pas vraiment d'intérêt pour l'utilisation souhaitée.

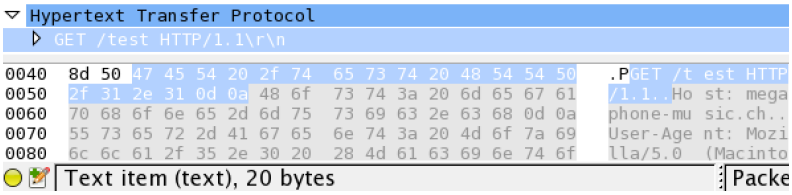

Figure 3, Taille de la requête GET

La taille du texte reçu (payload) est de 21 octets (« this is a simple test ») :

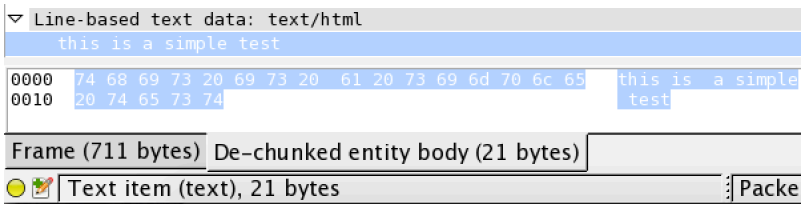

Figure 4, Taille de la réponse au GET

La taille des données utiles, à savoir la requête (URL) et la réponse (texte) est donc de 41 octets.

Il faut ensuite additionner la totalité des octets de tous les paquets ayant été nécessaires à l'échange afin de connaitre la taille totale du transfert :

- SYN : 78 octets
- SYN, ACK : 74 octets
- ACK : 66 octets
- HTTP GET : 372 octets
- ACK : 66 octets
- HTTP 200 OK : 711 octets
- ACK : 66 octets
- FIN : 66 octets
- ACK : 66 octets
- FIN, ACK : 66 octets
- ACK : 66 octets

Soit un total de 1'697 octets.

L'échange comprend donc 1'697 – 41 = 1'656 octets « overhead » sur un échange de 1'697 octets, soit 97% d'overhead.

Précisons que cet échange s'est déroulé sans pertes, à savoir dans une situation idéale. Si une perte était survenue, le segment TCP faisant défaut aurait alors été retransmis, ce qui aurait augmenté la quantité de données transférées lors de l'échange.

3.2.6 Conséquences sur la consommation électrique

Un overhead tel que celui calculé précédemment signifie implicitement qu'une majeure partie du courant électrique consommé sera « gaspillé ». Effectivement, les mécanismes de TCP sont idéaux pour la navigation web, mais dans le cadre de ce projet, à savoir l'Internet des objets, de tels mécanismes sont extrêmement couteux en termes d'efficience.

Dans l'Internet des objets, chaque échange doit être le plus succinct possible, c'est-à-dire éviter l'effet « ping-pong », et garantir des échanges de courte durée, car une longue communication signifie une transmission couteuse en énergie.

3.3 L'Internet des objets

3.3.1 Caractéristiques

Contrairement à une architecture d'Internet classique, l'Internet des objets, ou plus précisément les objets connectés doivent répondre à plusieurs critères.

I. Une connectivité sans fil

   Un smartphone, une corde à sauter connectée, un capteur d'humidité, une fourchette connectée ; autant d'objets qui ne seraient pas utilisables avec des câble d'alimentation et de données. Les objets connectés sont par définition « wireless ». Cette contrainte entraine d'autres contraintes, mentionnées ci-dessous.

II. La faible consommation d'énergie

   Les objets connectés sont souvent physiquement petits, voir très petit et ne peuvent embarquer d'énormes batteries, nécessaires à leur fonctionnement de manière autonome. C'est pourquoi une attention toute particulière doit être apportée à la consommation électrique des dispositifs. Les échanges de données « sans fil » doivent être limités au maximum, tout comme la complexité du programme embarqué. En effet, un programme mal conçu (ou mal adapté) utilisera plus d'énergie que nécessaire pour ses calculs.

III. Une découverte simplifiée

   La multitude des objets connectés nous oblige à penser à la connectivité et à la découverte de ceux-ci. Par « découverte » nous entendons par là l'identification dans le réseau. Comment un smartphone va-t-il pouvoir être à-même de « trouver » et de communiquer avec l'objet connecté. L'identification de l'objet de manière unique sur le réseau fait également partie de cette problématique.

IV. La sécurité

   La sécurité des données est un enjeu très important dans le monde actuel. L'objet doit communiquer de manière « privée » avec son correspondant. De la même manière, l'objet ne doit recevoir d'ordre que des personnes autorisées. Cette problématique ne sera pas traitée dans le cadre de ce projet.

3.3.2 Problématiques avec les une architecture conventionnelle

Par « architecture conventionnelle », il est entendu par là « Ethernet-TCP/IP-HTTP ». Reprenons les caractéristiques énoncées précédemment :

- Une connectivité sans fil

  Ce point peut être respecté avec une architecture conventionnel. Un réseau Wifi 802.11 permettrait de connecter notre objet au réseau, sans se soucier des caractéristiques suivantes.

- La faible consommation d'énergie

  Nous voici dans la problématique principale de l'architecture classique. L'utilisation de HTTP au niveau applicatif nous oblige à utiliser TCP, très gourmant en énergie car il fonctionne en mode « connexion », ce qui entraine beaucoup d'échanges radio.

- Une découverte simplifiée

  Là aussi, l'architecture classique n'est pas optimisée pour la découverte automatique. Nous pourrions imaginer une découverte à l'aide du nom d'hôte de l'objet, mais nous serions limités à un seul objet de ce type, ce qui n'est pas adéquat.

En conclusion, nous voyons que cette couche applicative n'est pas conçue pour l'Internet des objets. C'est pourquoi d'autres solutions ont été inventées.

3.3.3 Les alternatives

En réponse à ces problématiques, les ingénieurs ont développé différentes alternatives adaptées à l'Internet des objets. Nous recherchons une technologie sans fil, ayant une portée d'environ 10m et consommant peu d'énergie (PAN).

- Le Wifi (802.11) est rapidement écarté car trop gourmant en énergie pour fonctionner sur batterie.
- ZigBee (basé sur 802.15.4) est un bon candidat. Il consomme très peu d'énergie et utilise une topologie maillée, ce qui lui permet de couvrir de plus longues distance si nous avons beaucoup de nœuds. Il est cependant relativement « fermé » (ZigBee Alliance).
- Le Bluetooth classique est également un bon candidat, mais consomme encore trop d'énergie pour l'Internet des objets.

3.3.4 Bluetooth Low Energy

La meilleure alternative est l'utilisation de Bluetooth Low Energy (ou Bluetooth Smart), qui permet de satisfaire tous nos critères.

La première solution serait de connecter l'objet directement en Bluetooth avec le client. Mais cette solution ne permet pas à l'objet d'être indépendant dans le réseau Internet (avec une adresse IP propre). De plus, cette implémentation nécessiterait que les données soient stockées directement dans l'objet, ce qui n'est pas souhaitable. Il est préférable de déporter l'intelligence à l'extérieur de l'objet. Dans le cas de la machine à café, nous aimerions centraliser les données (nombre de cafés consommés par personne, …) dans une base de données externe.

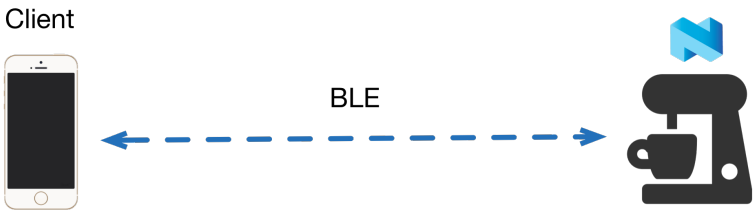

Figure 5, Bluetooth Low Energy

Arrive maintenant la problématique de l'identifications dans le réseau. Nous souhaitons que l'objet soit identifiable de manière unique et accessible dans le réseau IP mondial. La meilleure solution est de lui attribuer une adresse IPv6 publique propre, ce qui implique que l'objet devra implémenter l'entier de la stack IP, qui implique plus de puissance et de mémoire sur l'objet.

Pour ce faire, il existe une technologie appelée 6loWPAN, qui permet de faire passer des paquets l'IPv6 sur une couche physique Bluetooth.

3.3.5 Les versions de Bluetooth Low Energy

La taille maximale des trames Bluetooth Low Energy dépend des versions. Jusqu'à la version 4.1 de BLE, le MTU était de 27 bytes.

La version 4.2, dévoilée en décembre 2014 annonce un MTU de 251 bytes, ce qui est un pas de en avant pour Bluetooth dans le monde de l'Internet IP.

3.3.6 6loWPAN

6loWPAN permet de faire passer des paquets l'IPv6 sur une couche physique Bluetooth Low Energy. Ce protocole est spécifié dans la RFC4944 datant de 2007.

Afin de permettre la communication en IPv6 sur du Bluetooth Low Energy, 6loWPAN doit se plier aux contraintes imposées par IPv6 et par Bluetooth Low Energy. La principale tâche sera de minimiser au maximum la taille des données transmises, notamment en compressant l'entête IPv6 et en fragmentant les paquets IPv6 de manière à ce qu'ils rentrent dans les PDU Bluetooth.

Bluetooth Low Energy supporte des payloads de 27 bytes jusqu'à sa version 4.1. La version 4.2, qui est la dernière en date permet de transporter des payloads de 251 bytes. Le PDU total au niveau Link Layer est de 33 bytes.

Compression de l'entête IPv6

La première action est la compression de l'entête IPv6 qui est de 40 bytes dans sa version standard. Ce mécanisme est expliqué dans la RFC 6282. 6loWPAN a connu plusieurs méthodes de compressions, mais la plus utilisée actuellement s'appelle « LOWPAN_IPHC ». Avec cette méthode, le taux de compression dépend du type de communication.

- Pour une communication entre deux liens locaux (fe80), l'entête IPv6 peut être réduit à 2 bytes. Un byte de « Dispatch » qui spécifie le type de paquet au-dessus, et un byte LOWPAN_IPHC.
- Pour une communication nécessitant plusieurs sauts IP, l'entête peut être compressé sur 7 bytes (1 byte de « Dispatch », 1 byte LOWPAN_IPHC, 1 byte Hop Limit, 2 bytes Source Address et 2 bytes Destination Address).

Le byte LOWPAN_IPHC spécifie les paramètres spécifiques de l'entête IPv6, sous une forme extrêmement compressée. Il se base sur des « suppositions » afin de faire cette compression.

Il est également possible de compresser l'entête UDP de 8 bytes à 4 bytes. Mais cela entraine une diminution drastique des ports à disposition car il n'en reste que 16.

Fragmentation du paquet IPv6

Le deuxième mécanisme que 6loWPAN met en place est la fragmentation des paquets IPv6 afin que ces derniers transitent sur des PDU limités. Cette fragmentation implique un entête de fragmentation 6loWPAN de 4 à 5 bytes. Il est défini comme suit :

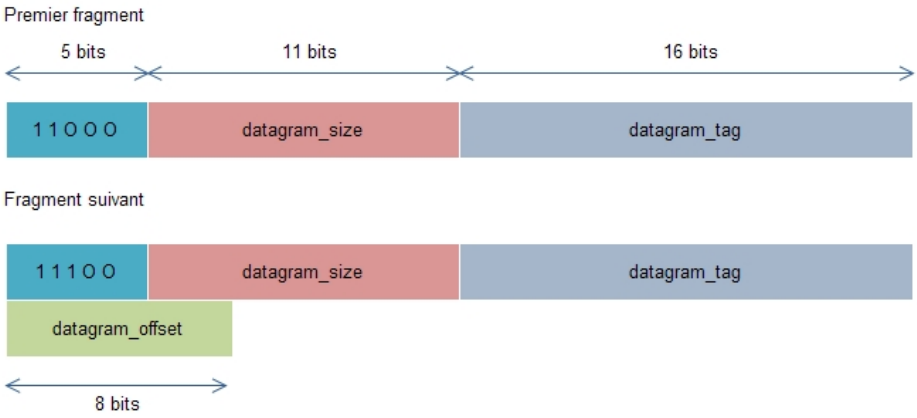

Figure 6, Représentation de la fragmentation 6loWPAN tiré de Wikipédia

- dispatch (5 bits) : permettent d'identifier qu'il s'agit d'un fragment
- datagram_size (11 bits) : taille du paquet IP avant fragmentation
- datagram_tag (16 bits) : identifiant à tout les fragments de ce paquet IP
- datagram_offset(8 bits) : position du fragment dans le paquet IP, uniquement sur les suivants

Nous pouvons noter qu'à l'heure actuelle, il n'existe pas de mécanisme d'acquittement et de récupération des fragments perdus.

Exemple avec une couche BLE

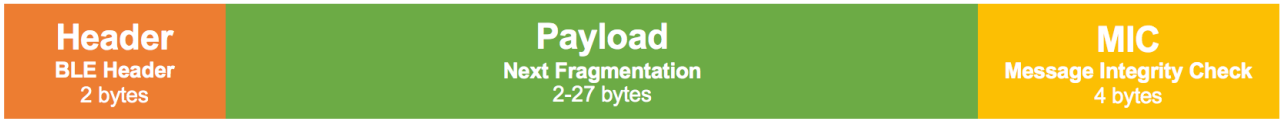

Figure 7, Représentation d'un PDU au niveau Link Layer (33 bytes)

Prenons une implémentation de 6loWPAN sur Bluetooth Low Energy 4.1, possédant un payload de 27 bytes. Comme spécifié dans la RFC 7668, l'entête L2CAP nécessaire à la communication Bluetooth est de 4 bytes.

Nous voulons transmettre le message « this is a simple test », qui une taille de 21 bytes. Deux transmissions BLE seraient nécessaires à l'envoi de ce message (plus si nous incluons une couche applicative). Ce message est transmis entre deux intervenants d'un réseau public (2001::…).

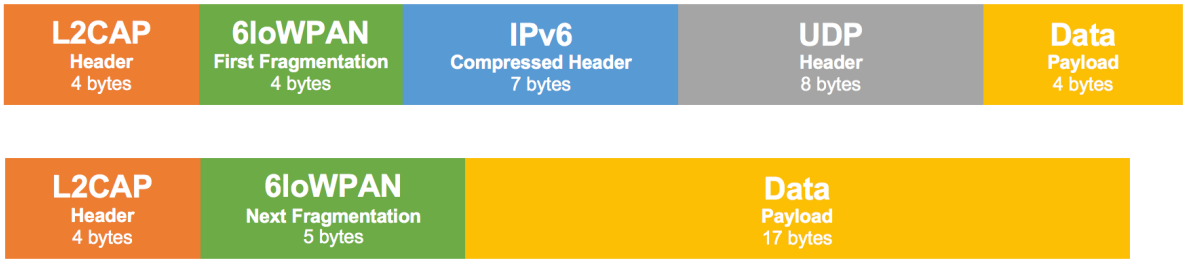

Figure 8, Représentation d'un transfert de données d'une taille de 21 bytes (hors header de la couche applicative)

3.3.7 Passage de 6loWPAN à l'Ethernet classique

Afin de passer du réseau Bluetooth au réseau Ethernet classique, il faut mettre en place un gateway, une passerelle. Ce dernier doit implémenter le 6loWPAN sur Bluetooth Low Energy d'un côté, et l'Ethernet de l'autre. C'est donc une passerelle qui transforme uniquement les couches physique et liaison d'un côté à l'autre. Les couches supérieures transitent de manière indépendantes.

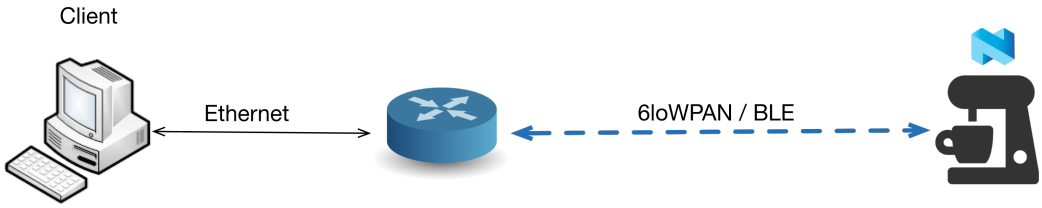

Figure 9, Passage de 6loWPAN à l'Ethernet classique

3.3.8 La couche applicative

Nous avons vu que l'utilisation du protocole HTTP pour la couche applicative n'est pas adapté, entre autre à cause de l'utilisation impérative de TCP. Il existe deux protocoles applicatifs se revendiquant plus « léger ». Il s'agit de CoAP et MQTT. Tout deux seront étudiés et comparés plus tard dans ce projet.

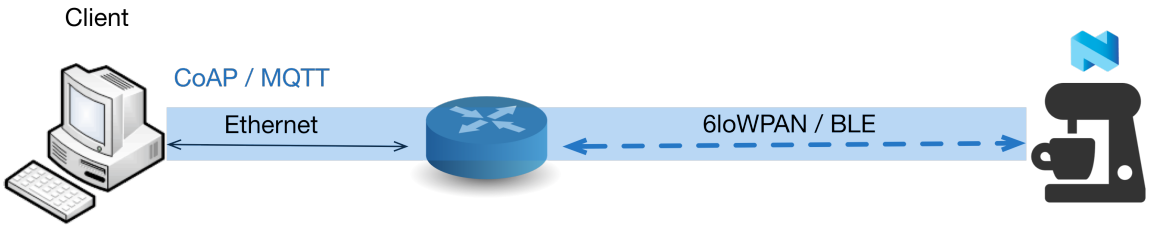

Figure 10, Couche applicative

La première idée est de faire du CoAP ou MQTT de bout en bout, de l'objet au client. L'inconvénient de cette architecture est que nos browsers actuels ne gèrent pas nativement CoAP ou MQTT. Il faudrait donc développer une application native à chaque client, permettant de communiquer en CoAP ou MQTT avec l'objet. Cette solution n'est donc pas retenue.

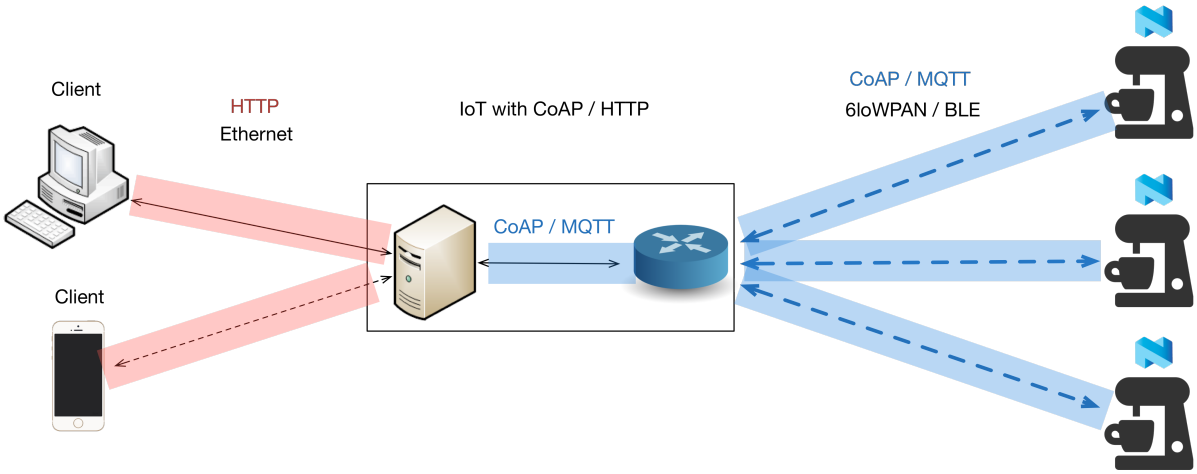

Figure 11, Proxy HTTP / CoAP / MQTT

Pour résoudre ce problème, la solution est d'y intégrer un serveur, implémentant un mécanisme permettant aux clients de communiquer en HTTP, et aux objets de communiquer en CoAP.

Du côté client, ce serveur devra agir comme un serveur web classique. Le client interagira avec lui en demandant une page web. Au terme du chargement de cette page, le client devra être à même de visualiser les informations sur la consommation de café des utilisateurs de manière simple. Une attention doit donc être apportée à la mise en page des informations.

Ce serveur devra également être capable de conserver les données à court terme en son sein, afin de ménager les objets. Si par exemple plusieurs clients questionnent les objets dans un court délai, il est nécessaire que le serveur mémorise l'information et la réutilise, ceci afin de réduire les communications aux objets à leur maximum, et ainsi économiser leur énergie.

Deux architectures répondant à ces besoins sont possibles : l'utilisation d'un proxy ou l'utilisation d'un serveur web. Ces deux solutions vont être présentées, comparées puis un choix basé sur cette analyse sera effectué.

3.3.9 Proxy VS Web Server

La première possibilité est la mise en place d'un proxy, dont le rôle sera de convertir les requêtes HTTP en CoAP. Les clients pourraient ainsi communiquer avec les objets via des requêtes HTTP. Cependant, les différents objets fourniront des données brutes, sans mise en forme. Cela implique donc que l'utilisation d'un proxy nécessitera malgré tout l'utilisation d'un serveur web pouvant délivrer la partie « graphique ». Effectivement, l'utilisateur final ne sera pas satisfait avec simplement du JSON retourné des différents objets.

Il serait donc possible d'utiliser un proxy ainsi qu'un serveur web. Le serveur web livrerait à l'utilisateur l'interface graphique, comprenant du JavaScript, et c'est ce dernier qui effectuerait des requêtes HTTP REST auprès du proxy, qui seraient finalement converties en requêtes CoAP puis transmises aux objets…

Voici un exemple d'interaction avec un proxy :

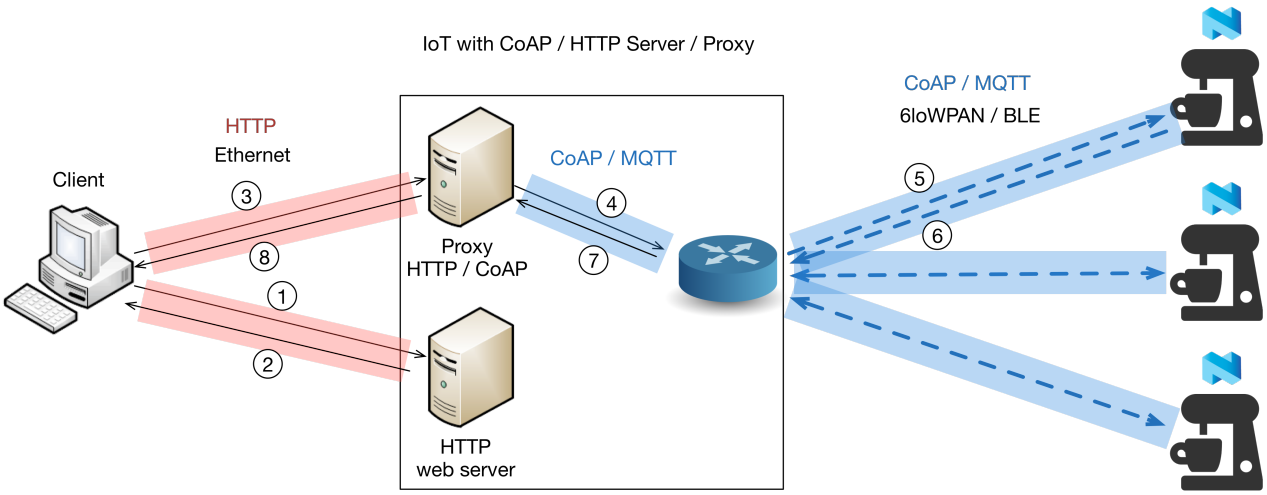

Figure 12, Le proxy et le serveur web ont volontairement été représentés comme deux entités logiques.

1) Le client demande l'accès à la page de *statistiques* (par exemple).
2) Le serveur web HTTP lui retourne une page statique, comportant du JavaScript dont le but sera d'effectuer des requêtes AJAX afin de collecter des données (format REST).
3) Le client effectue les requêtes AJAX spécifiées auprès du proxy.
4) Le proxy converti les requêtes HTTP en CoAP.
5) Le routeur compresse l'entête IPv6 et fragmente le paquet à l'aide de 6loWPAN afin de le transmettre en Bluetooth Low Energy auprès de l'objet.
6) L'objet répond à la demande.
7) Le routeur effectue la manipulation inverse, transformant à nouveau le paquet en IPv6 standard.
8) Le proxy converti la réponse CoAP en réponse HTTP, puis la retourne au client. Ce dernier traitera l'information reçu avec JavaScript et l'affichera.

La seconde solution pour résoudre ce problème est d'intégrer un serveur web dans l'architecture, capable de jouer le rôle de passerelle entre le CoAP|MQTT et le HTTP, ce dernier étant compris par tous les browsers.

Ce serveur web fera également office centralisateur des données, grâce à une mini base de données qui sera mise à jour par les évènements CoAP ou MQTT annoncés de par l'objet. Pour ce faire, un client CoAP|MQTT sera développé en Python.

Du côté HTTP, ce serveur agira comme un serveur web classique. Il permettra de donner au client une page HTML complète qui contiendra les informations sur la consommation de café des utilisateurs. Ce serveur sera très probablement également mis en place avec Python de manière à garder une infrastructure légère.

Il sera également possible de communiquer en CoAP avec l'objet directement depuis le réseau Ethernet. (Communication de bout en bout avec un client CoAP)

L'interaction avec l'architecture proposée serait la suivante :

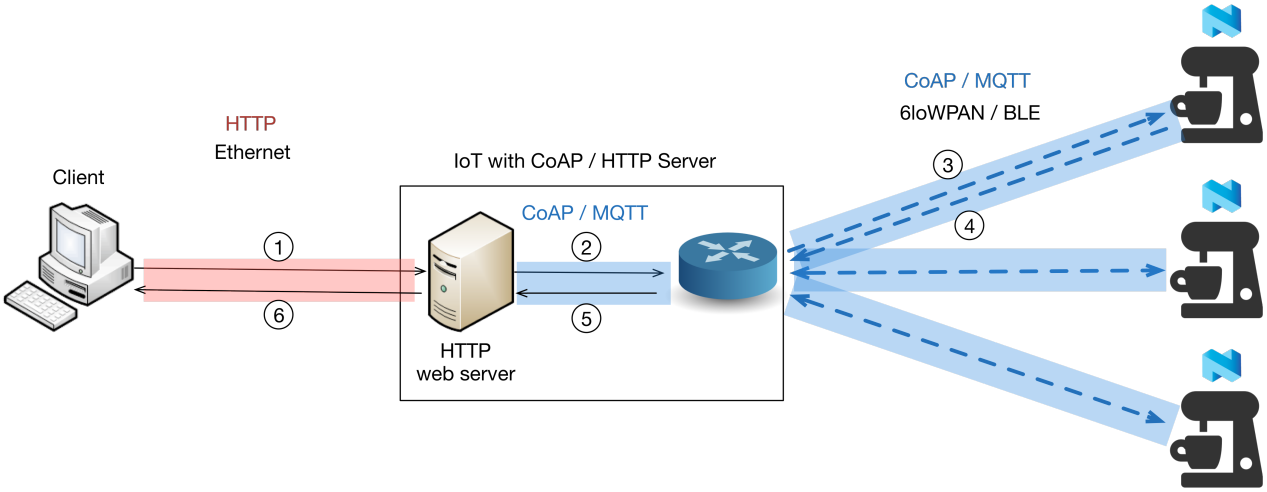

Figure 13, Interactions

1) Le client demande l'accès à la page de *statistiques* (par exemple).
2) Le serveur analyse la demande et effectue les requêtes nécessaires afin de satisfaire la demande du client. Si nécessaire, il génère la requête CoAP et l'envoie au routeur.
3) Le routeur compresse l'entête IPv6 et fragmente le paquet à l'aide de 6loWPAN afin de le transmettre en Bluetooth Low Energy auprès de l'objet.
4) L'objet répond à la demande.
5) Le routeur effectue la manipulation inverse, transformant à nouveau le paquet en IPv6 standard.
6) Le serveur génère une dynamiquement la page avec les informations obtenues des objets, puis livre cette page au client final.

Afin de limiter au maximum les requêtes envoyées aux devices, les deux architectures peuvent implémenter un mécanisme de mémorisation des informations, à savoir un cache.

De plus, le web serveur utiliserait également une base de données. Cette dernière permettrait de rendre les données persistantes. Il serait alors possible de faire des statiques étalées dans le temps. Ceci pourrait par exemple servir dans le cas de la machine à café à estimer la date de la prochaine maintenance. Cette architecture serait par ailleurs prête à accueillir en son sein le système de gestion et d'accounting par utilisateur.

Voici un petit récapitulatif des avantages/inconvénients sous la forme d'un tableau :

Web Server	Proxy
+ extensible pour implémenter l'accounting + mémorisation à long terme des données	+ page dynamique qui s'adapte aux valeurs retournées des devices + REST de bout en bout + accès direct aux objets
- plus lourd car base de données - doit générer les pages	- besoin malgré tout d'un web server - nécessite l'utilisation de requêtes AJAX pour charger dynamiquement le contenu

Les deux architectures ayant respectivement leurs points fort et faibles, un « merge » des deux va être présenté afin de tirer profit des forces de chacune d'entre elles.

3.3.10 Architecture Proxy avec base de données et web server

Cette architecture reprend le principe de l'architecture « Proxy » présentée précédemment, et y ajoute une base de donnée.

Concrètement, le client effectuant une requête recevra une page statique, comportant du JavaScript. C'est ce dernier qui effectuera différentes requêtes HTTP vers le proxy pour obtenir des ressources (nombre de cafés produits par exemple). Le proxy possédant un cache y cherchera d'abord les informations récentes, et dans le cas ou l'information n'existe pas ou est trop ancienne, il convertira la requête en CoAP qu'il enverra aux différent objets.

L'avantage de fournir directement une page au client et qu'il pourra tout de suite afficher du contenu sur son écran. Effectivement, il se peut que l'un ou l'autre des objets soit indisponible ou mette du temps à répondre. Dans ce cas cela n'est pas critique car le client aura déjà reçu la page demandée, et verra probablement une information de « loading ». Dès lors que le JavaScript obtiendra une réponse, la page sera mise à jour directement par le browser du client.

La base de données permettra elle de sauvegarder de manière persistante les données reçues par le proxy. En plus de répondre à l'utilisateur et de conserver les données dans le cache, le proxy pourra stocker les valeurs dans une base de donnée, permettant ainsi de générer une page de statistiques d'après l'historique de données.

L'architecture serait la suivante :

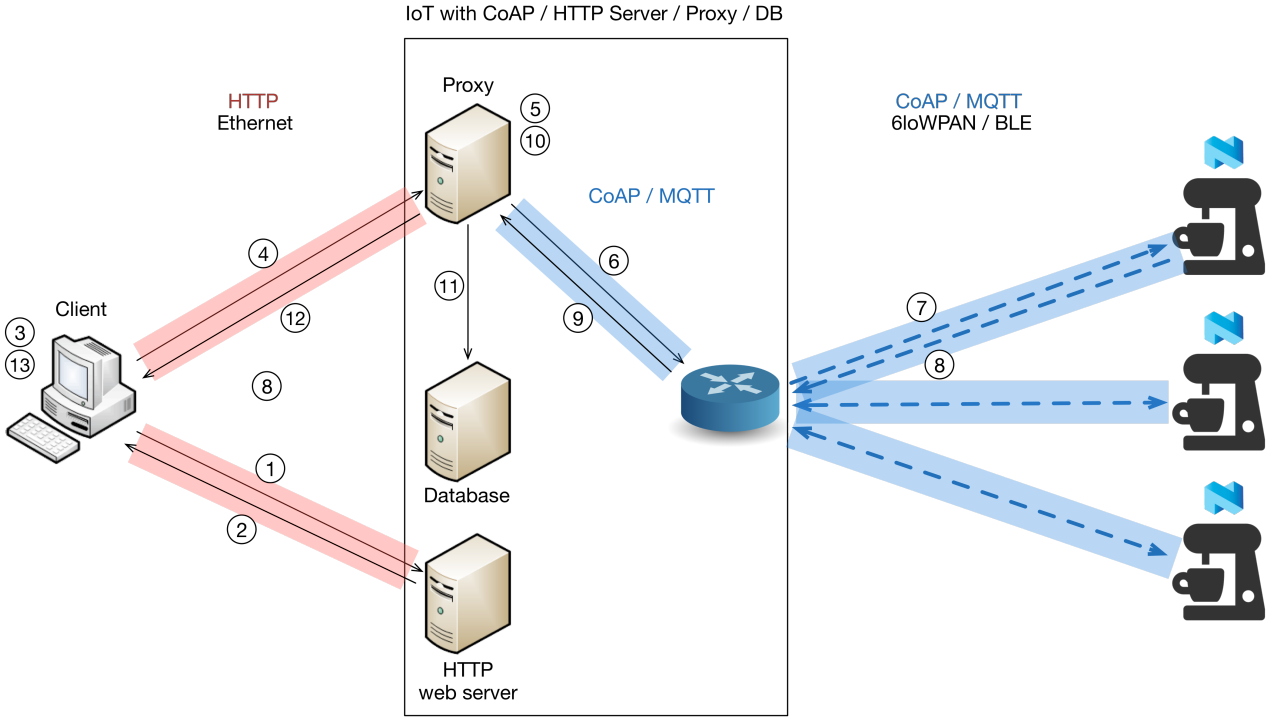

Figure 14, Le proxy, la base de données et le serveur web ont volontairement été représentés comme trois entités logiques.

1) Le client demande l'accès à la page de consommation de cafés (par exemple)
2) Le serveur web lui retourne une page statique HTML comportant du JavaScript
3) Le client traite localement le JavaScript, et effectuera donc les requêtes AJAX nécessaires afin d'obtenir les informations des objets
4) Le client effectue une requête HTTP nécessaire auprès du proxy, afin d'obtenir les informations d'un device
5) Le proxy vérifie tout d'abord si son cache possède ou non une information pertinente
6) Dans la négative, il converti la requête HTTP en CoAP et l'envoie au device en question
7) Le routeur compresse l'entête IPv6 et fragmente le paquet à l'aide de 6loWPAN afin de le transmettre en Bluetooth Low Energy auprès de l'objet
8) L'objet recompose le paquet IPv6, le traite et répond à la demande par un nouveau paquet IPv6 compressé et fragmenté avec 6loWPAN
9) Le routeur effectue la manipulation inverse, transformant à nouveau le paquet en IPv6 standard
10) Le proxy met en cache la réponse afin de pouvoir répondre rapidement à une requête similaire si une telle se produit
11) Le proxy sauve également l'information dans la base de donnée, afin de garantir le stockage à long terme de l'information et de permettre au système de générer des statistiques
12) Le proxy retourne à l'utilisateur l'information qu'il avait demandé au point 4)
13) Le client met à jour dynamiquement la page à l'aide de JavaScript, selon les informations reçues

3.3.11 Architecture du système Bluetooth Low Energy

Bluetooth Low Energy utilise une architecture spécifique divisée en deux parties principales, *Host* et *Controller*. Les deux sont connectées au moyen d'une interface *Host Controller Interface*.

Nous retrouvons dans le Controller :

- La couche physique, qui représente la transmission radio des données
- La couche link layer, qui s'occupe principalement de l'advertising, du scanning et de l'attribution d'une adresse au dispositif. Cette dernière fait office de MAC adresse.

Dans la partie Host (les principales) :

- Le L2CAP (Logical Link Control And Adaptation Protocol), qui est une couche d'adaptation (encapsulation) des couches supérieures en paquets BLE standardisés.
- Le GATT (Gensèric Attribute Profile), qui définit comment les données sont organisés échangées entre les dispositifs Bluetooth à l'aide de « profile ».
- Le GAP (Generic Access Profile), qui définit comment les périphériques interagissent entre eux. Il gère la découverte, la connexion et la sécurité.

Dans le cas de l'utilisation de 6loWPAN, c'est le profile IPSP (Internet Protocol Support Profil) qui est utilisé. IPSS, GATT et ATT sont utilisés uniquement pour la découverte. GAP est utilisé pour la découverte et la connexion. Une fois la connexion établie, la communication s'effectue directement sur la couche d'encapsulation L2CAP.

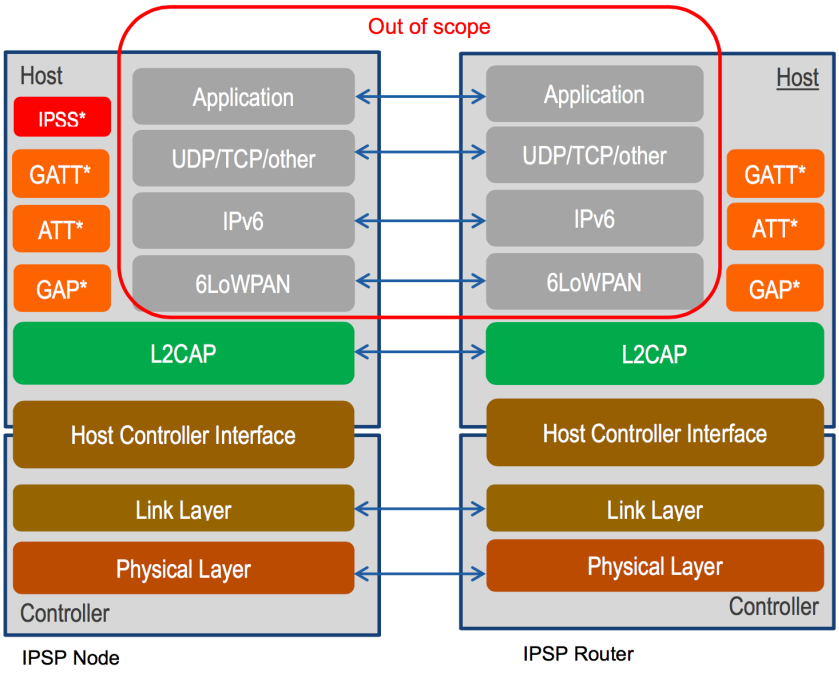

Figure 15, Schéma tiré de la définition du profile IPSP [3]

3.3.12 IPv6

Afin que le dispositif puisse communiquer sur Internet, il doit recevoir une adresse publique IPv6. Pour ce faire, il existe un mécanisme d'attribution automatique d'adresse, propre à IPv6 appelé « Stateless auto-configuration ». La procédure complète de l'Internet classique a été simplifiée afin limiter au maximum les échanges entre le router et l'objet. (Pas de vérification si l'adresse locale est unique)

Premièrement, l'objet crée sa propre adresse IPv6 locale (FE80…) en utilisant l'algorithme EUI-64, qui permet de générer une adresse IPv6 à l'aide des 48 bits de la MAC adresse. Le routeur possède déjà une adresse locale, et une liste de préfixes publiques.

Une fois la connexion Bluetooth établie à l'aide du profile IPSP, l'objet envoie un « Router Solicitation Message » (RS) afin de trouver un routeur IPv6 sur le réseau.

Le routeur lui répond par un « Router Advertisment » (RA) en lui proposant un préfix publique qu'il peut utiliser. L'objet peut alors construire sa propre adresse IPv6 publique en utilisant le même suffixe que pour son adresse locale.

Une fois son adresse publique complète établie, l'objet effectue un « Neighbors Solicitation » afin de vérifier que son adresse IPv6 est unique sur le réseau.

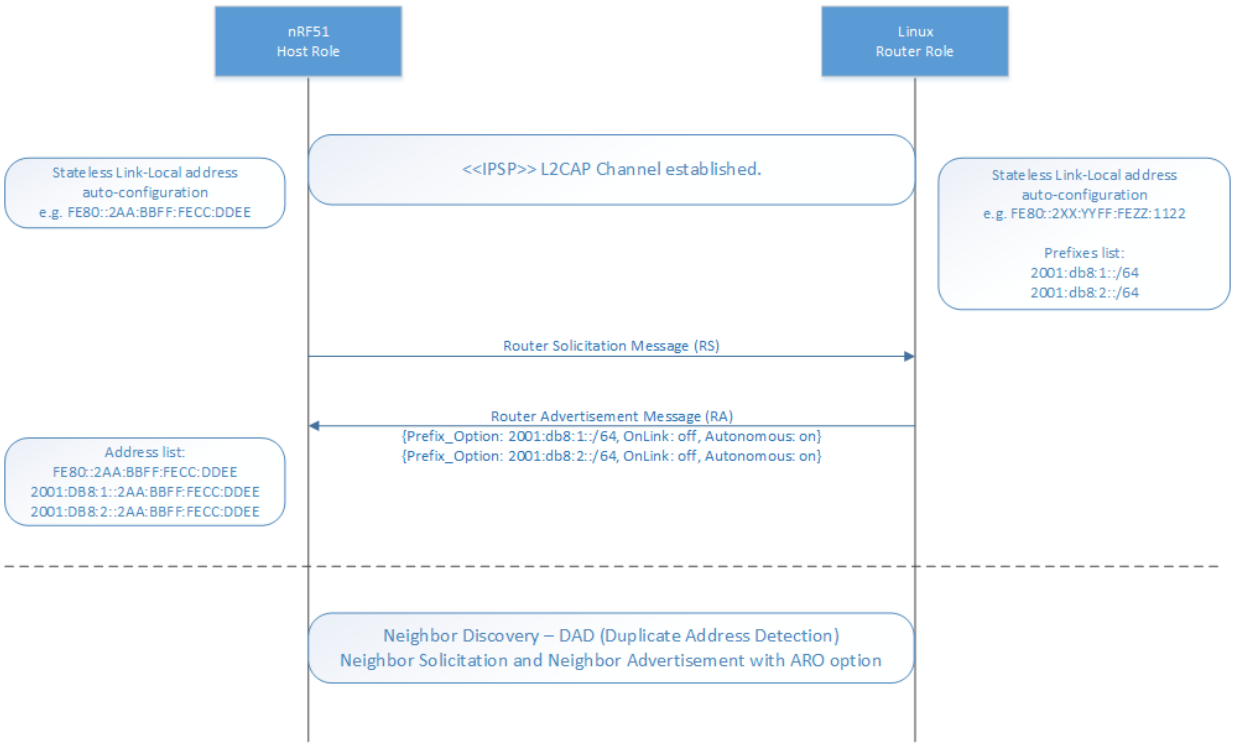

Figure 16, Procédure pour l'attribution d'adresse IPv6 tiré du SDK nRF51

Dans notre infrastructure, la passerelle n'aura qu'une fonction de conversion du réseau Ethernet vers le Bluetooth avec 6loWPAN. Les couches supérieures ne seront pas affectées (mesh-under).

L'objet et le serveur recevrons une adresse IPv6 publique depuis un routeur « réellement » intégré à l'Internet mondial. (Par exemple le routeur IPv6 de l'école)

3.3.13 Résumé illustré du dispositif

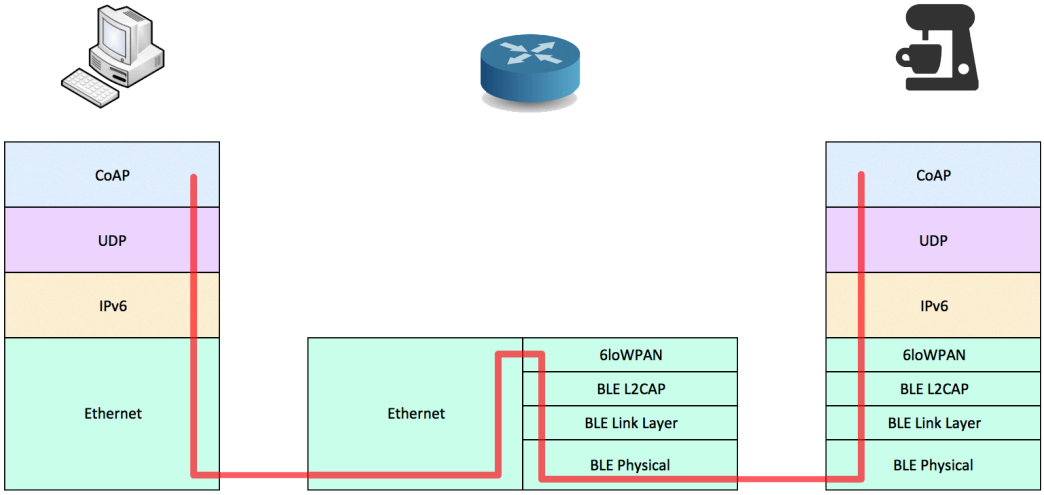

Figure 17, Dans le cas d'une communication entre un client CoAP Ethernet et l'objet CoAP 6loWPAN. Toute personne connectée à l'internet mondiale peut donc communiquer directement avec l'objet en CoAP

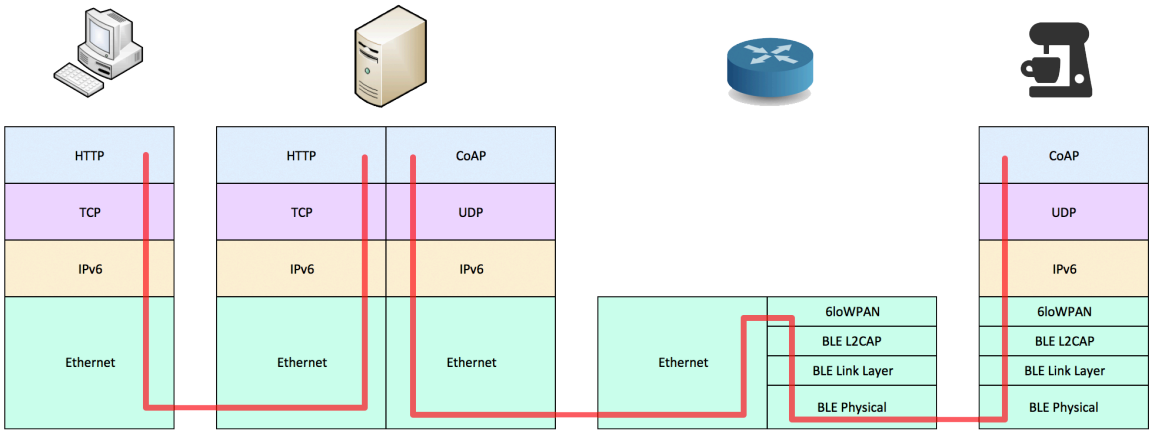

Figure 18, Dans le cas d'une communication à l'aide d'un client HTTP sur Ethernet.
Le « router » fonctionne en mode « mesh-under »

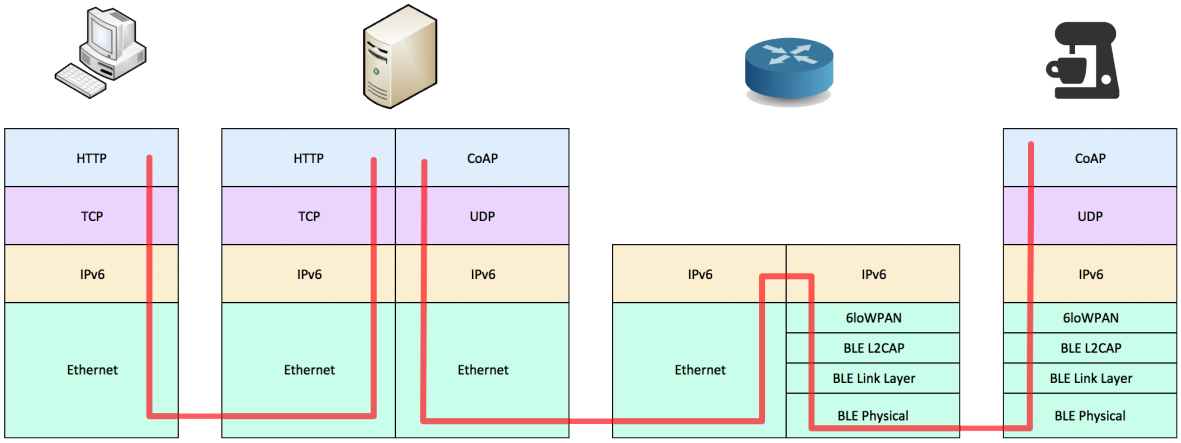

Figure 19, Dans le cas d'une communication à l'aide d'un client HTTP sur Ethernet.
Le « router » fonctionne en mode « router-over » (Pas utilisé dans ce projet)

4 Couche applicative

4.1 MQTT

4.1.1 Introduction

MQTT (*Message Queuing Telemetry Transport*) est un protocole de communication applicatif de type « publish/subscribe » développé par IBM au début des années 2000. Il est maintenant open source et standardisé par OASIS (Advancing open standards for the information society). MQTT est originellement basé sur une connexion TCP/IP. Les ports 1883 et 8883 (SSL) sont réservé par l'IANA pour MQTT. Le protocole est actuellement en version 3.1.1.

Il existe une version alternative appelée MQTT-SN (Sensing Network) destinée aux systèmes embarqués qui ne peuvent pas gérer de lourdes connexions TCP. Ce protocole utilise alors UDP pour le transport. MQTT-SN est en version 2.1.

4.1.2 Architecture

MQTT fonctionne selon un principe de client/server, où chaque capteur est un client, et se connecte sur TCP à un serveur appelé *broker*. Les principaux *brokers* disponibles sont Mosquitto, Rabbit MQ et ActiveMQ. Il existe des plateformes en ligne qui permettent d'obtenir un *broker* dans le cloud (cloudmqtt.com) au travers d'Internet.

Chaque message est *publish* à une adresse, définie par un *topic*. Les clients qui souhaitent s'abonner à un *topic* peuvent faire un *subscribe*. Ils recevront alors tous les messages envoyés à ce *topic*.

Les schémas ci-dessous présente le fonctionnement global de MQTT. Les clients A, B et C se connectent en TCP au *broker*. Les clients B et C *subscribe* au *topic* « temperature ». Le client A *publish* une nouvelle valeur pour « temperature ». Le *broker* retransmet le *publish* aux deux *subscriber*.

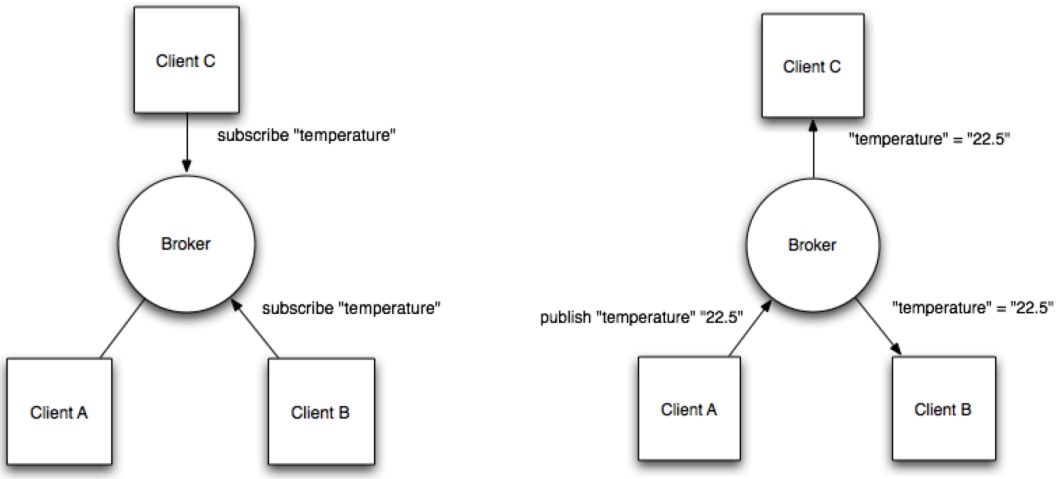

Figure 20, Schémas : http://www.eclipse.org [6]

4.1.3 Les topics

Les topics sont organisés de manière hiérarchique.

Par exemple :
```
bureau/1/machine/1/temperature
bureau/1/machine/1/stock
bureau/2/machine/1/temperature
```

Il existe des « wildcard » qui permettent de souscrire à plusieurs *topics* à la fois (uniquement pour le *subscribe* et non pour le *publish*). Le « + » remplace n'importe quel sous-répertoire, et le « # » remplace toute la descendance.

Par exemple :	`bureau/+/machine/+/stock`	les stocks de toutes les machines
	`bureau/1/#`	tous les *topics* du bureau 1

4.1.4 QoS

MQTT possède trois QoS, qui définissent avec quelle importance le client/broker veut que ses messages soient reçus. Utiliser des priorités élevées entraine des latences et de plus grandes exigence de bande passante.

- 0 : Le client/broker envoie le message qu'une seule fois, sans confirmation. Ce mode est également appelé « *fire and forget* ». (Advienne que pourra)
- 1 : Le client/broker envoie le message au moins une fois, avec confirmation. Il peut être amené à le retransmettre s'il ne reçoit pas de confirmation.
- 2 : Le client/broker envoie le message une fois, mais en utilisant une procédure sécurisée en 4 étapes. Informations complémentaires [8]

Il est possible de définir une QoS pour le *publish*, et une autre pour le *subscribe*. Le broker relayera les messages selon la QoS la plus petite des deux.

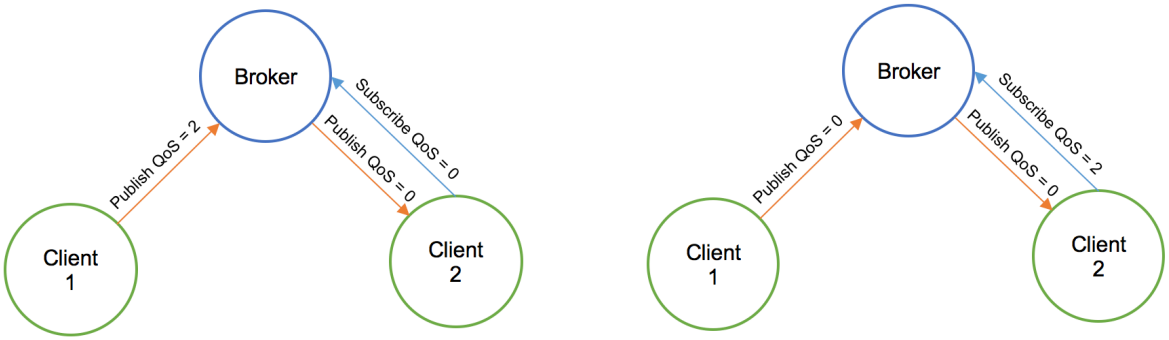

Figure 21, Deux exemples de relation QoS broker/clients

4.1.5 Persistance et « last will and testament »

MQTT permet au *broker* de sauver une version de la dernière valeur reçue pour un *topic* dans sa mémoire. Les clients peuvent demander au *broker* de « persister » la valeur. Quand un autre client souscrit à ce *topic*, il recevra du *broker* la dernière valeur actualisée.

Les clients peuvent souscrire à un message « *last will and testament* » qui sera envoyé par le *broker* afin de les informer qu'un autre client s'est déconnecté. Ceci peut être utile dans le cas d'une messagerie instantanée en ligne, pour notifier qu'un partenaire a quitté la conversation.

4.1.6 Format des messages

Cette partie est basée sur la spécification de MQTT 3.1 [9].

Le header

Les messages MQTT possèdent un header fixe de 2 bytes. Celui-ci contient :

- Message Type (4 bits) : Par exemple *SUBSCRIBE*, *CONNECT*, *PUBLISH*, …
- DUP flag (1 bit) : 1 dans le cas d'une tentative de *re-deliver* d'un message (QoS > 0)
- QoS level (2 bits) : Niveau de QoS comme expliqué au point 4.1.4
- Retain (1 bit) : Indique au serveur de persister la valeur donnée dans un *publish*
- Remaining Length (8 bits) : Nombre de bytes restant dans ce message

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP flag	QoS level		RETAIN
byte 2	Remaining Length							

Figure 22, Représentation du header fixe MQTT [9]

Il existe différents headers optionnels qui peuvent être utilisés tels que *Protocol Version, Topic Name, Keep Alive Timer*…

Connexion

Une fois la connexion TCP établit entre le client et le *broker*, le client envoie un message CONNECT spécifiant un username et un password. Le *broker* lui répond par un CONNACK.

Le client peut ensuite envoyer des messages de type PUBLISH, SUBSCRIBE, … et le *broker* le notifier par les PUBLISH des *topics* auquel il est inscrit.

Lors de la connexion, le client définit un délai de « Keep Alive Timer » avec le *broker*. Ce timer représente le temps maximal en seconde sans que le broker ne reçoive de message du client. Passé ce délai, le broker peut fermer la connexion TCP. La valeur maximale est d'environ 18 heures. Le client est responsable d'envoyer dans la période de « Keep Alive Timer » des messages de type PINGREQ que le *broker* confirme avec des PINGRESP. Ces messages ont pour but d'attester que le client est toujours connecté.

Message PUBLISH

Lors d'un PUBLISH (nouvelle valeur poussée vers le *broker*), le message est composé du header fixe, puis des headers optionnels *Topic Name* et *Message Identifier*. La valeur vient ensuite et son « formatage » est définit par l'application qui utilise MQTT (format libre).

Le *Topic Name* est encodé en UTF-8 et sa taille peut être de 32'767 caractères.

Le *Message Identifier* permet d'identifier le message. Il a une taille de 16 bits (65'636 ID uniques).

	Description	7	6	5	4	3	2	1	0
Topic Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (3)	0	0	0	0	0	0	1	1
byte 3	'a' (0x61)	0	1	1	0	0	0	0	1
byte 4	'/' (0x2F)	0	0	1	0	1	1	1	1
byte 5	'b' (0x62)	0	1	1	0	0	0	1	0
Message Identifier									
byte 6	Message ID MSB (0)	0	0	0	0	0	0	0	0
byte 7	Message ID LSB (10)	0	0	0	0	1	0	1	0

Figure 23, Exemple des headers Topic Name « a/b » et Message Identifier « 10 » [9]

4.1.7 Performances

Afin de tester ce protocole, un *broker* « en ligne » a été choisit. Il s'agit de cloudmqtt.com, qui propose une version d'essaie gratuite limitée à 10 connexions et bridée à 10kbit/s, ce qui est suffisant pour nos tests. Deux utilisateurs ont été créés via l'interface graphique de cloudmqtt. Le client a accès en lecture au *topic /home/office/device/*+ et le device (représentant l'objet ou la machine à café) en mode écriture.

User	Topic	Read	Write
client	/home/office/device/+	true	false
device	/home/office/device/+	false	true

Figure 24, Configuration du broker sur mqttcloud.com

Au niveau applicatif, une implémentation de MQTT 3.1 (TCP) en Python a été choisit. C'est un projet Opensource propulsée par Eclipse qui est disponible à cette adresse : www.eclipse.org/paho/.

Le client va *SUBSCRIBE* au *topic* en utilisant un wildcard, et le device va PUBLISH des valeurs sur différents « id », représentant divers objets.

Du côté du client, la connexion sera maintenue de manière permanente. Des messages *PINGREQ* et *PINGRESP* seront échangés toutes les 60 secondes (valeur définie dans le programme).

Du côté du device, nous partons du principe qu'il ne peut pas maintenir une connexion TCP ouverte en permanence, ceci pour des questions d'énergie. De plus, si le device envoie une nouvelle valeur par heure, il n'est pas nécessaire de maintenir une telle connexion. Par conséquent, la connexion sera ouverte puis close pour chaque *PUBLISH*. Nous allons mesurer l'overhead généré pour qu'une nouvelle valeur soit *PUBLISH* par le *device* vers le *broker* (Connexion-Publish-Déconnexion).

Source	Destination	Protocol	Length	Info
192.168.1.106	54.75.180.40	TCP	78	57337→18440 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1009438535 TSecr=0 SACK_PERM=1
54.75.180.40	192.168.1.106	TCP	74	18440→57337 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=209063097 TSecr=1009438535 WS=128
192.168.1.106	54.75.180.40	TCP	66	57337→18440 [ACK] Seq=1 Ack=1 Win=131744 Len=0 TSval=1009438576 TSecr=209063097

Figure 25, Le device (client) initie une connexion TCP avec le broker (server)

Source	Destination	Protocol	Length	Info
192.168.1.106	54.75.180.40	MQTT	122	Connect Command
54.75.180.40	192.168.1.106	TCP	66	18440→57337 [ACK] Seq=1 Ack=57 Win=29056 Len=0 TSval=209063108 TSecr=1009438576
54.75.180.40	192.168.1.106	MQTT	70	Connect Ack
192.168.1.106	54.75.180.40	TCP	66	57337→18440 [ACK] Seq=57 Ack=5 Win=131744 Len=0 TSval=1009438619 TSecr=209063108

Figure 26, Une fois la connexion TCP établit, un message MQTT de connexion « Connect Command » est envoyé par le device au broker. Il contient notamment les informations d'identification (username & password). Le broker lui répond par un « Connect Ack ». Chaque message envoyé est acquitté au niveau TCP.

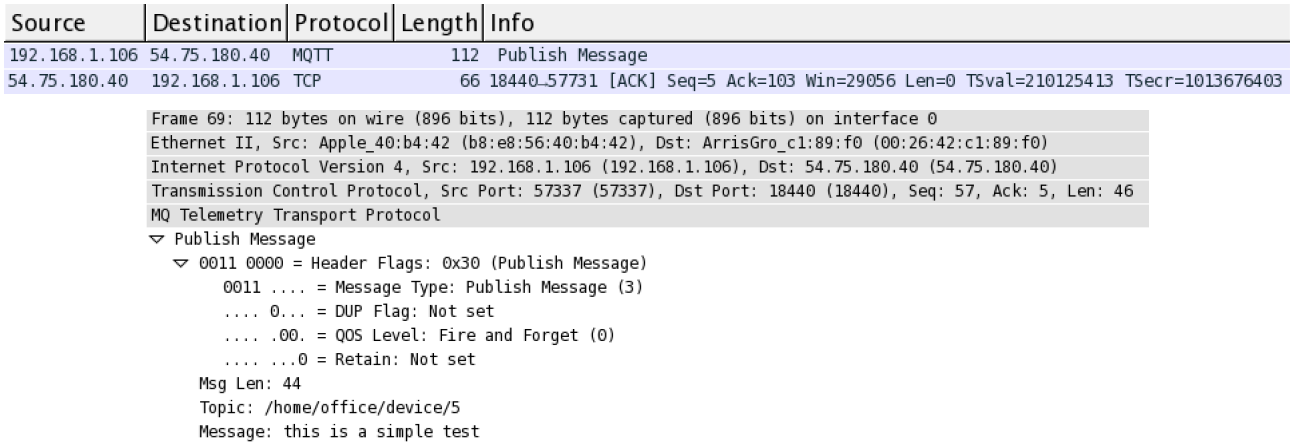

Figure 27, Intervient ensuite le message PUBLISH envoyé du device vers le broker. Nous voyons qu'il est envoyé en QoS 0 (Fire and Forget), le broker n'a pas besoin de stocker la valeur de manière persistante, le topic est « /home/office/device/5 » et la valeur « this is a simple test ».

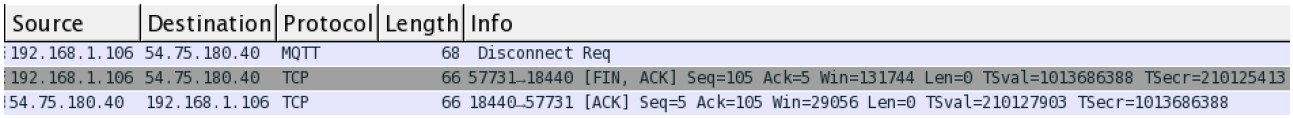

Figure 28, Le device envoie ensuite un message de déconnexion (Disconnect Req) au broker et met fin à la connexion TCP.

Sachant que nous avons un message « this is a simple test » de 21 bytes, et un nom de topic de 23 bytes, la charge utile est de 44 bytes. Nous pouvons trouver l'overhead en additionnant la taille de tout les paquets échangés. Précision, ce test est effectué sur une couche physique Ethernet, en IPv4 :

- TCP SYN 78 bytes
- TCP SYN, ACK 74 bytes
- TCP ACK 66 bytes
- MQTT CONNECT 122 bytes
- TCP ACK 66 bytes
- MQTT CONNACK 77 bytes
- TCP ACK 66 bytes
- MQTT PUBLISH 112 bytes
- TCP ACK 66 bytes
- MQTT DISCONNECT 68 bytes
- TCP FIN, ACK 66 bytes
- TCP ACK 66 bytes

Soit un total de 927 bytes échangés, ce qui représente 95.25% d'overhead en mode TCP.

4.1.8 Conclusion

Bien que l'overhead soit important, nous pouvons souligner le fait que la taille totale de la requête est presque divisée par 2 par rapport à un simple GET HTTP (1'697 en HTTP contre 927 en MQTT).

Ces deux protocoles applicatifs sont difficilement comparables. En effet, HTTP est uniquement basé un un principe de Client-Serveur, alors que MQTT devient intéressant dès lors que plusieurs clients sont connectés au même *broker*. Le principe de « *subscribe* » et de « *publish* » avec le système de *topic* est très puissant, mais il n'est en rien comparable avec HTTP.

L'inconvénient principal de MQTT est l'utilisation d'une connexion TCP. L'overhead généré par cette connexion est très important. La version SN de MQTT utilise UDP, ce qui doit être beaucoup plus optimisé pour l'internet des objets. Ce dernier ne sera pas analysé durant le projet.

En conclusion, MQTT 3.1.1 est une couche applicative bien plus légère que HTTP 1.1, mais elle n'est pas optimale pour l'internet des objets. MQTT-SN est une meilleure alternative, mais peine encore à se démocratiser. MQTT serait utilisable dans ce projet, mais n'est pas pleinement adapté à nos besoins.

4.2 CoAP

4.2.1 Introduction

Afin de mettre en place une architecture de type requête-réponse nous avons précédemment étudié la possibilité d'utiliser le protocole HTTP. Il en est ressorti que son utilisation, nécessitant l'utilisation de TCP en couche transport n'était pas optimale pour l'Internet des objets de par l'importante quantité d'overhead généré, ainsi que la complexité des mécanismes de connexion TCP.

L'utilisation de HTTP est néanmoins possible, et ce même sur une connexion Bluetooth, car l'utilisation de la couche d'adaptation 6loWPAN permet la fragmentation des paquets, et donc leur transport en de multiples petits paquets.

Nous allons maintenant nous intéresser à un protocole plus léger, et donc mieux adapté à un réseau « restreint », tel que Bluetooth Low Energy. Il s'agit du protocole CoAP, défini dans la RFC 7252 [4].

Il est important de préciser que CoAP n'est pas une compression de HTTP. Il s'agit d'un protocole indépendant reprenant les concepts REST mais optimisé pour des systèmes limités, en offrant par exemple des mécanismes de découverte, de multicast et de communication asynchrone.

4.2.2 Fonctionnement du protocole

Comme décrit précédemment, CoAP fonctionne selon un modèle client-serveur, de manière analogue à HTTP. Cependant, dans le cas d'une interaction entre deux objets, les deux interlocuteurs peuvent avoir les rôles de client et de serveur.

Les requêtes CoAP fonctionnent de manière similaires à celles de HTTP, le client appelle une certaine méthode sur une ressource choisie. Le serveur lui renvoie alors une réponse.

La différence principale avec HTTP est le fonctionnement asynchrone des échanges en CoAP. Effectivement, dans le cas d'une perte menant à une retransmission, il est possible que la réponse B arrive avant la réponse A. Il apparait donc clairement que CoAP est étudié pour des échanges ne nécessitant pas de temps réel ni de respect dans l'ordre de traitement.

CoAP est construit sur UDP, il n'est ainsi pas dépendant des mécanismes de session de TCP. Néanmoins, afin de fournir de la fiabilité dans ses échanges, un système d'acquittement utilisant l'exponential back-off est mis en place ([4] section 2.1, [14]).

CoAP défini quatre types de messages :

- **Confirmable « CON »** : ce type de message est fiable, car il demande au partenaire un acquittement. Tant que l'acquittement n'est pas reçu, le message initial est ré-envoyé, tout d'abord après une durée définie par défaut, puis selon l'exponential back-off.
- **Acknowledgement « ACK »** : ce type de message est envoyé en réponse d'un message « confirmable ». L'ID du message acquitté est inclus dans ce message, de sorte que l'interlocuteur puisse savoir à quel message initialement envoyé se rapporte l'acquittement.
- **Non-confirmable « NON »** : ce type de message ne demande pas d'acquittement. La donnée transmise n'est donc pas garantie d'arriver au destinataire. Ce type de message peut par exemple être utilisé dans le cas d'un capteur effectuant une mesure toutes les minutes. Si sur une longue période d'utilisation quelques mesures sont perdues, cela n'affecte pas la qualité de la mesure. Cela permet d'économiser l'écoute d'un acquittement, et donc de l'énergie.
- **Reset « RST »** : ce type de message est utilisé lorsque le récepteur n'est pas en mesure de traiter le message reçu. Il signale ainsi à l'émetteur qu'il a reçu le message, mais qu'il ne peut pas l'utiliser.

Les messages *Confirmable*, *Acknowledgement* et *Non-confirmable* peuvent contenir des données.

4.2.3 Echanges requêtes-réponses

En utilisant les différents types de messages proposés par CoAP, il est possible d'effectuer les échanges simples suivants :

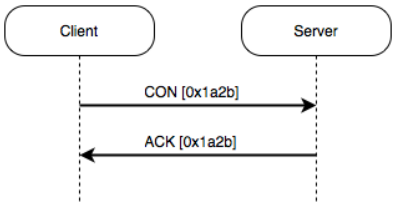

Figure 29, Requête « confirmable » obtenant une réponse, sans pertes

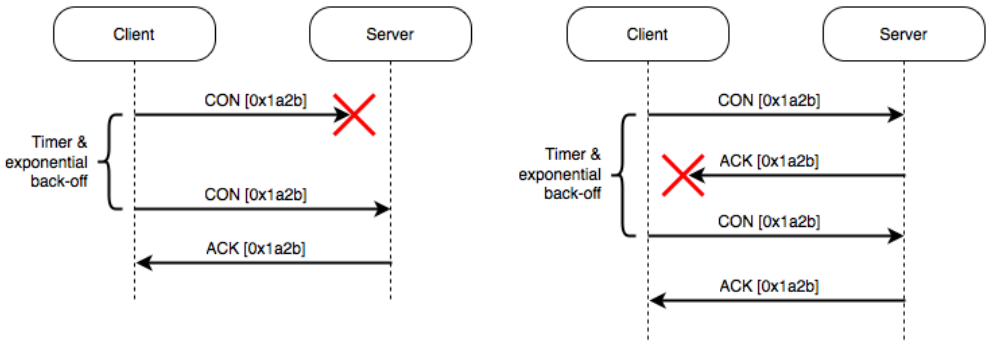

Figure 30, Requête « confirmable » subissant une perte lors de l'envoi (à gauche) et lors de l'acquittement (à droite), puis étant retransmise suite au dépassement du time

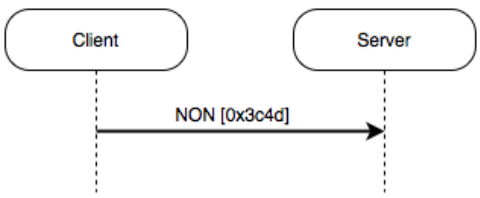

Figure 31, Requête « non-confirmable » envoyée. Aucun acquittement n'est demandé.

4.2.4 Méthodes CoAP

CoAP propose les méthodes suivantes. Les codes de retour sont cités et seront résumés au chapitre suivant :

- **GET** : cette méthode retourne la ressource identifiée par le paramètre URI de la requête. Il est possible de spécifier le format souhaité de la réponse à l'aide du paramètre « Accept », comme avec HTTP. Lors d'un succès d'envoi, le serveur doit joindre le code de retour 2.05.
- **POST** : cette méthode demande le traitement des données qu'elle fourni au serveur. Le traitement sera dépendant de la ressource ciblée. Cela peut mener à la création de nouvelle ressource, dans ce cas le code de retour est 2.01. Si cela mère à une mise à jour de données, dans ce cas le code de retour est 2.04. Finalement, si suite à cette requête des données sont supprimées, le code de retour sera 2.02.
- **PUT** : cette méthode demande la création, ou la mise à jour en cas d'existence préalable, de la ressource ciblée avec les données transmises. Les codes d'erreur sont similaires à ceux utilisés pour le POST.
- **DELETE** : cette méthode demande la suppression de la ressource ciblée. Le code de réponse utilisé est identique que pour une suppression avec la méthode POST.

4.2.5 Codes de retour

Les des différents échanges, des codes de retour sont joints. Ci-dessous leur signification :

Les messages de type « Success 2.xx » signifient que la requête a bien été reçue, comprise et acceptée par l'interlocuteur :

Success 2.xx	2.01	Created	Indique que la ressource a bien été créée suite à un PUT ou POST
	2.02	Deleted	Indique que la ressource a bien été supprimée suite à un DELETE
	2.03	Valid	Dans le cas d'utilisation d'un cache, indique à ce dernier que l'information stockée est à jour
	2.04	Changed	Indique que la ressource a bien été modifiée suite à un PUT ou POST
	2.05	Content	Utilisé en réponse à un GET avec la ressource demandée en tant que payload

Les messages de type « Client error 4.xx » signifient que le client a commis une erreur lors de sa requête :

Client error 4.xx	4.00	Bad Request	Indique au client qu'il a mal formulé sa requête
	4.01	Unauthorized	Indique au client qu'il n'est pas autorisé à effectuer l'action demandée car il ne s'est pas authentifié
	4.02	Bad Option	Indique au client qu'une ou plusieurs options fournies sont inconnues ou erronées
	4.03	Forbidden	Indique au client que le serveur a connaissance de son identité mais l'accès lui est refusé
	4.04	Not Found	Indique au client que la ressource demandée n'a pas été trouvée

	4.05	Method Not Allowed	Indique au client que la méthode qu'il a employé n'est pas permise (p.ex. utiliser un GET sur un formulaire requérant un POST)
	4.06	Not Acceptable	Indique au client que la ressource demandée ne peut fournir du contenu compatible avec le paramètre « Accept » qu'il a fourni
	4.12	Precondition Failed	Indique au client que le serveur ne rempli pas les conditions qu'il a spécifié
	4.13	Request Entity Too Large	Indique au client que sa requête est plus grand que le maximum traitable pas le serveur
	4.15	Unsupported Content-Format	Indique au client que sa requête demande un type de format non supporté par le serveur

Les messages de type « Server error 5.xx » signifient que le serveur a effectué une erreur, ou n'est pas capable de traiter la demande.

Server error 5.xx	5.00	Internal Server Error	Une erreur inattendue s'est produite sur le serveur, et aucun autre message plus précis n'est disponible
	5.01	Not Implemented	Le serveur n'a pas reconnu la demande ou n'est pas capable de la traiter, cela implique généralement qu'il sera capable de la traiter dans le futur
	5.02	Bad Gateway	Le serveur agissait en tant que gateway ou proxy et a reçu une réponse invalide du serveur cible
	5.03	Service Unavailable	Le serveur est actuellement indisponible, due à une surcharge ou une maintenance, il s'agit d'un état temporaire
	5.04	Gateway Timeout	Le serveur agissait en tant que gateway ou proxy et n'a pas reçu de réponse dans les délais du serveur cible
	5.05	Proxying Not Supported	Le serveur ne peut ou ne veut agir en tant que proxy

L'utilisation de ces codes de retour fait sens dans le monde de l'internet des objets, encore plus que dans l'internet classique, car ils permettent d'économiser des transferts de données. Effectivement, dans le cas ou un client tente d'accéder à une ressource interdite, il est plus efficace de transmettre un message contenant le code de retour adapté, plutôt que d'envoyer un message contenant textuellement l'information d'interdiction.

4.2.6 Piggybacking

Lorsque le client envoie une requête via un message « confirmable » le serveur devra acquitter la demande. Dans ce cas, un système de piggyback peut être utilisé afin d'économiser un échange supplémentaire sur le réseau. Ce mécanisme permet de mettre directement les données dans le paquet « Ack » qui est renvoyé au client. Ainsi, au lieu d'envoyer le « ack », puis ensuite les données en elles-mêmes, le serveur peut regrouper ces deux actions en une seule.

Ce mécanisme peut être utilisé lorsque la demande peut être adressée immédiatement. Lorsque la demande nécessite plus de temps à traiter, le serveur peut choisir d'envoyer séparément l'acquittement puis les données. Ceci permet d'éviter que suite à un délai trop long le client renvoie sa demande (en croyant à une perte).

4.2.7 Format des trames

Le format des trames CoAP, tel qu'indiqué dans le RFC 7252 [4] est le suivant :

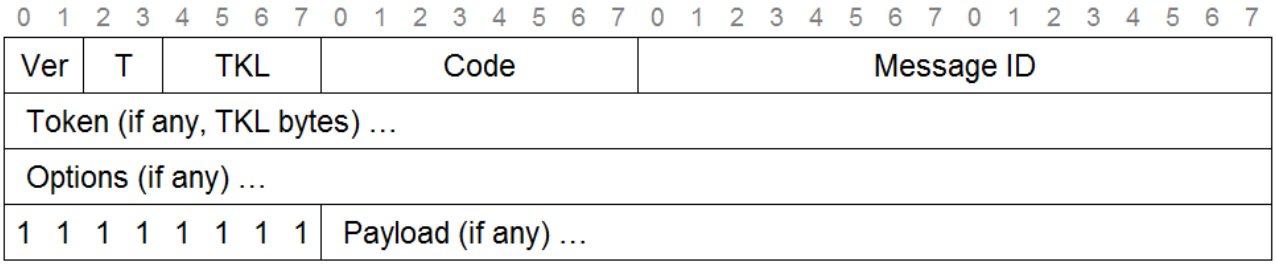

Figure 32, Format des trames CoAP

Version (Ver) : Indique la version de CoAP utilisée. La spécification actuelle et utilisée doit indiquer la version « 01 ».

Type (T) : Indique le type du message, à savoir :

- « 00 » : Confirmable
- « 01 » : Non-confirmable
- « 10 » : Acknowledgement
- « 11 » : Reset

Token Length (TKL) : Indique la longueur du champ « Token »

Code : Indique le code du message, tel qu'indiqué au point 4.2.5. Les 3 bits de poids fort indiquent la classe (2.xx / 4.xx / 5.xx) et les 5 bits de poids faible le message.

Message ID : Indique l'ID du message, ceci permettant de faire correspondre un acquittement à un message confirmable par exemple.

Token : Le token, également appelé « request ID » est utilisé pour regrouper les requêtes-réponses

Options : Indique les options choisies.

Payload Marker (0xFF) : Si un payload est présent, indique la fin des options et le début du payload.

Payload : Représente les données. La longueur du payload représente la taille restante depuis le marker jusqu'à la fin du datagramme UDP.

4.2.8 Conversion des requêtes HTTP <-> CoAP

Le proxy devra être à-même de convertir les requêtes HTTP en CoAP et inversement. Plus précisément, il devra permettre aux clients HTTP de communiquer avec des dispositifs CoAP de manière standardisé. Il n'existe actuellement qu'un draft d'une RFC expliquant ce fonctionnement [11].

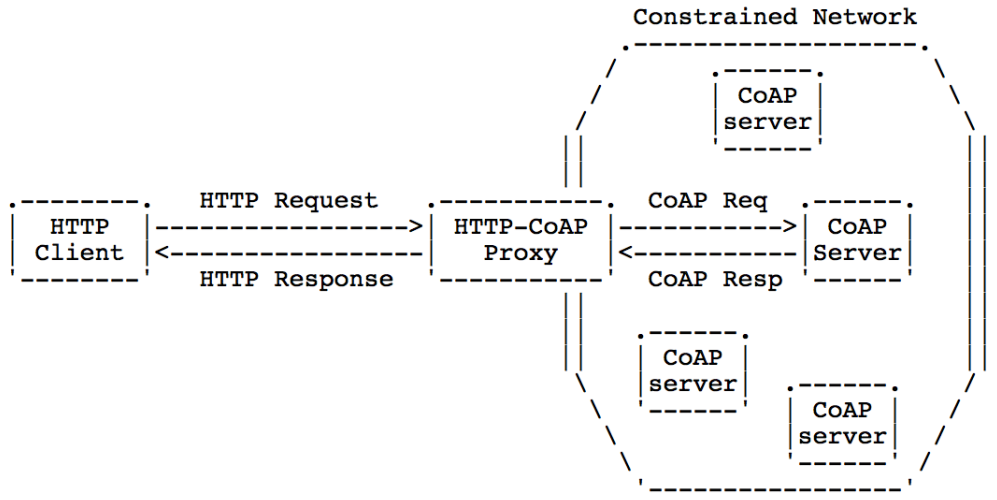

Figure 33, Schéma de fonctionnement général tiré de [11]

La première solution est de faire du mapping d'URI. Le client ne sait pas que c'est un dispositif CoAP qui sera interrogé. Le proxy réceptionne la requête HTTP du client, il effectue la requête CoAP correspondante et renvoie la réponse en HTTP au client. On parle alors de « *déréférencement de ressources CoAP* ». L'URI est construite de manière similaire du côté HTTP et CoAP. Cette solution était proposée dans la première version du draft [12]. Le proxy est donc complètement dédié à cette translation. Il ne peut pas répondre à d'autres URI particulière, par exemple pour renvoyer une page HTML, ou du contenu formaté. Toutes les requêtes sont transformées.

Exemple :

la requête HTTP :	http://something.net/node/foo
est mappée à	coap://something.net/node/foo

Nous pouvons imaginer toutes les variantes de mapping, par exemple en utilisant une URI spécifique aux requêtes CoAP. Cela peut limiter la complexité d'implémentation au niveau du proxy.

Exemple :

la requête HTTP :	http://something.net/**coap**/node/foo
est mappée à	coap://something.net/node/foo

La deuxième solution (expliquée dans la dernière version du draft [11]) consiste à concaténer les deux requêtes. Le client a donc une vision de l'arborescence mise en place dans le réseau CoAP. Il connait la requête CoAP à exécuter. Une fois que le proxy a extrait la requête CoAP à exécuter, il procède de manière similaire à la première solution.

Exemple :

la requête CoAP :	coap://something.net/foo
est transmise par HTTP	http://something.net/coap://something.net/node/foo

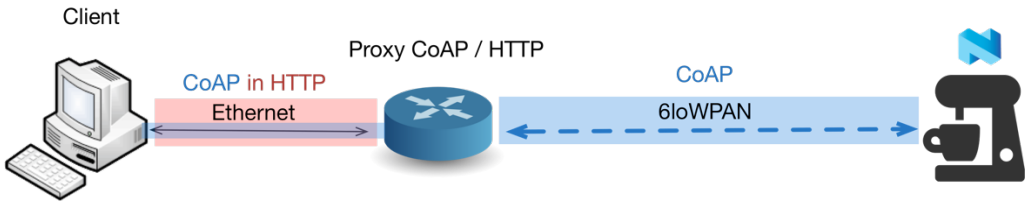

Figure 34, Représentation de la conversion CoAP – HTTP avec concaténation d'URI

4.2.9 Performances

Afin de tester ce protocole, un petit serveur Python a été mis en place. Il utilise la librairie Aiocoap [13] implémentant le protocole selon la RFC 7252 [4].

Le serveur a été configuré pour écouter sur l'url « coap://localhost/home/office/device/4 » afin d'obtenir une mesure similaire à celle réalisée au point 4.1.7 avec MQTT.

Le client, également en Python, envoie une requête GET au serveur. Pour la mesure, une configuration IP statique a été utilisée.

Lorsqu'il reçoit la requête, le serveur regarde si la ressource existe bel et bien et est disponible. Si oui, il renvoie un message texte « this is a simple test » à destination du client, avec le code de retour 2.05. Dans le cas ou la ressource n'existe pas, il renvoie un message « not found » avec le code d'erreur 4.04.

Ci-dessous, la capture Wireshark de l'échange :

Source	Destination	Protocol	Length	Info
160.98.113.198	160.98.115.160	CoAP	87	CON, MID:52363, GET, TKN:00 00 30 c1, coap://160.98.115.160/home/office/device/4
160.98.115.160	160.98.113.198	CoAP	72	ACK, MID:52363, 2.05 Content, TKN:00 00 30 c1 (text/plain)

Figure 35, Le client effectue une requête GET et le serveur renvoie un ACK piggybacked

Seulement deux paquets sont envoyés, car ce protocole fonctionne sur UDP. Il n'y a donc pas de mécanisme de connexion. Par ailleurs, il est possible de voir le mécanisme de piggybacking à l'œuvre, de par le fait que le « ACK » contienne directement le payload. Ceci évite une paire d'échange / ack supplémentaire.

Sachant que nous avons une requête de 21 bytes « home, office, device, 4 » et une réponse de 21 bytes « this is a simple test », la charge utile est de 42 bytes. Nous pouvons trouver l'overhead en additionnant la taille totale des messages échangés :

- CON : 87 bytes
- ACK : 72 bytes

Soit un total de 159 bytes échangés, ce qui représente 73.58% d'overhead.

4.2.10 Découverte dans CoAP

Afin de découvrir les différents nœuds CoAP présents dans le réseau, il faut utiliser les adresses multicast IPv6. Il est recommandé par la RFC, Chapitre 7 [4], d'utiliser l'adresse multicast « All CoAP Nodes » afin de découvrir les serveurs CoAP. Cette adresse est « ff02::fd ».

Les serveurs CoAP offrant la possibilité d'être découverts doivent impérativement écouter sur le port par défaut, à savoir le port 5683.

Afin de découvrir les ressources offertes par les serveurs, ces derniers doivent proposer à l'adresse connue suivante une liste de leurs ressources : « /.well-known/core ». Ceci est défini dans la RFC 6690, chapitre 1.2.1 [18].

Les ressources doivent être fournies selon un format proche du XML, par exemple :

```
</test>,
</home/office/device/4>,
</.well-known/core>;
ct=40
```

Chaque ressource est indiquée entre chevrons (<>), et son chemin complet doit être spécifié. Chaque ressource est séparée par une virgule, et terminées par un point virgule. Finalement, le « ct », à savoir le « content-format » est indiqué à 40, représentant une « liste de ressources ».

En théorie, il est donc possible d'envoyer un GET « /.well-known/core » à l'adresse « ff02::fd » afin d'obtenir une liste de toutes les ressources de tous les nœuds CoAP présents dans le sous-réseau actuel.

En pratique, cela n'est pas tout à fait le cas. L'adresse multicast IPv6 « ff02::fd » est validée dans la RFC 7252 (CoAP) [4], et également précisée dans la liste des adresses multicast IPv6 produite par l'IANA [19], mais après avoir réalisé des tests sur diverses machines, à savoir ordinateurs personnels, Raspberry PI et même le chip Nordic nRF51, il en est ressorti qu'aucune de ces machines ne répond à l'adresse multicast « All CoAP nodes » (ff02::fd).

Cependant, ces machines répondent si la requête est envoyée à l'adresse multicast « All nodes address » (ff02::1). La RFC n'est donc pas appliquée par les différentes implémentations disponibles, mais il est en revanche possible de contourner ce problème en utilisant l'adresse multicast « All nodes address » présentée ci-dessus. Etant donné que la requête est effectuée en CoAP, seules les machines disposant d'un serveur CoAP seront à même d'interpréter et de répondre à une telle requête. Lorsqu'à l'avenir les implémentations de CoAP seront à même de prendre en compte les requêtes à destination de cette adresse, il sera intéressant d'utiliser l'adresse « officielle » CoAP afin de suivre le standard.

4.2.11 Mécanisme d'observation

CoAP possède un mécanisme dit « d'observation ». Il s'agit d'une fonction permettant à un serveur ou un client, d'envoyer un message sans avoir expressément reçu de requête de la part de son interlocuteur. Cela permet par exemple à un serveur d'envoyer un message à un client lors de la mise à jour d'une ressource.

Le client devra préalablement avoir effectué une demande « d'observe ». Cette demande pourrait se traduire en « je souhaite être notifié des prochains changements de cette ressource ». Le serveur tient donc à jour une liste des clients à notifier par ressource. Lorsqu'un changement se produit, c'est sur la base de cette liste qu'il décidera à qui envoyer les notifications.

Voici une représentation d'un échange classique :

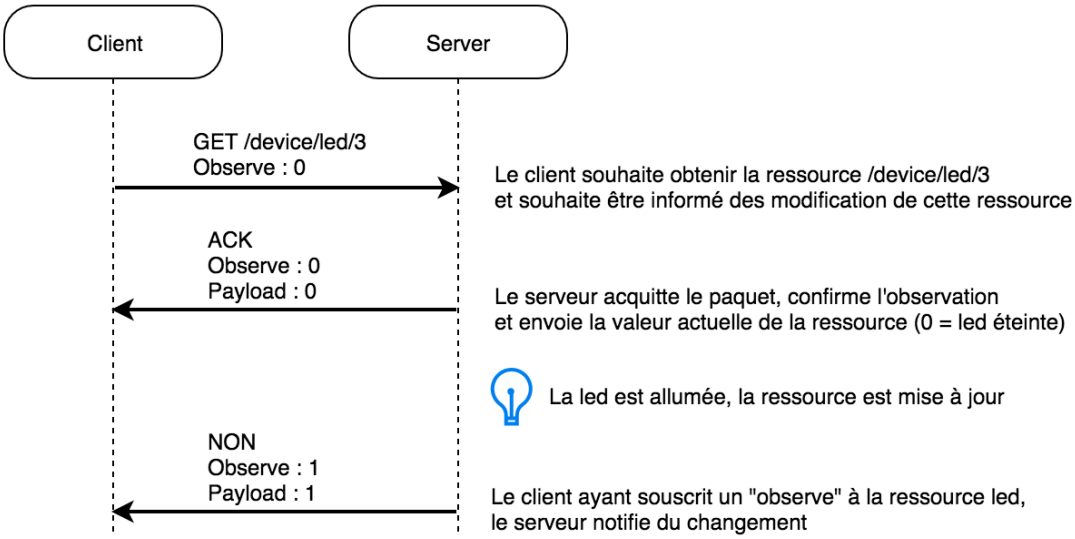

Figure 36, mécanisme d'observation

Concrètement, pour souscrire à la ressource, le client indique « Observe : 1 » dans le champ d'option du message. Le paramètre observe a pour numéro d'option « 6 », tel qu'indiqué dans la RFC 7641 [25]. Un client souhaitant s'inscrire envoie une valeur de « 0 » pour la paramètre observe. Pour la désinscription, il s'agit de la valeur « 1 ».

Le serveur répond en précisant le token correspondant à la requête, ainsi qu'un paramètre observe dont le rôle est de numéroter l'ordre des messages. Cette numérotation est nécessaire car CoAP ne proposant pas de mécanisme de synchronisation, ne peut pas garantir l'ordre d'arrivée des paquets.

Les messages sont envoyés en « best effort ». S'agissant de messages NON, le serveur ne sait pas si le client a bel et bien reçu la notification, il se contente de l'envoyer.

Afin d'éviter d'envoyer « à vide » des notifications pendant des heures dans le cas ou un client oublierait de se désinscrire, le serveur peut à tout moment envoyer la notification dans un message CON au lieu de NON. Un message CON nécessitant un acquittement de la part du client, si la notification n'est pas acquittée, le serveur est libre de considérer le client comme n'étant plus intéressé par l'évolution de cette ressource. Il peut alors l'effacer de sa liste de clients à informer.

4.2.12 Conclusion

L'utilisation de CoAP comparativement à MQTT ou HTTP représente une réduction de l'overhead généré de plus de 20%. Cela peut sembler minime, mais en valeur absolue cela représente presque 6x moins de trafic que MQTT et 10x moins que HTTP.

Ce résultat confirme l'idée que le protocole CoAP est adapté à l'utilisation de l'internet des objets. Ce sera donc ce protocole qui sera utilisé pour la réalisation du projet.

4.3 Solution sans IPv6

4.3.1 Présentation

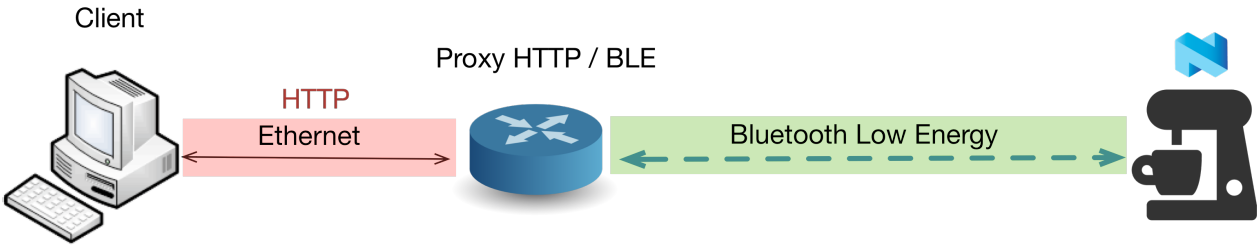

Figure 37, Schéma de l'architecture « Bluetooth Low Energy » pure

Maintenant que nous avons analysé les possibilités qu'offrent 6loWPAN et de l'utilisation d'IPv6, il est possible de se demander quel serait l'overhead généré si nous utilisions en lien « Bluetooth Low Energy » pur entre un proxy et les objets. Ce proxy pourrait répondre aux requêtes HTTP des clients web au format REST en interrogeant les dispositifs à l'aide de leurs « caractéristiques » BLE.

Cette architecture présente l'avantage d'être réellement « Low Energy ». En effet, BLE a été conçu de manière à échanger des caractéristiques à l'aide de la stack GATT/GAP, et non pour faire de l'IPv6 au-dessus. De plus, les objets consommeraient moins du fait de la complexité de la couche IPv6 et TCP/UDP à gérer sur ces petits systèmes embarqués.

4.3.2 Fonctionnement de GAP et GATT

Bluetooth Low Energy fonctionne selon un modèle composé de plusieurs couches (voir 3.3.11). Nous avons une architecture similaire jusqu'à la couche L2CAP, que nous utilisions du 6loWPAN ou du Bluetooth pur.

Dans le cas où nous utilisons la stack Bluetooth, nous avons les couches supérieures GATT et GAP, qui ont les rôles suivants :

- Le GATT (Generic Attribute Profile) définit comment les données sont organisés échangées entre les dispositifs Bluetooth à l'aide de « profile ».
- Le GAP (Generic Access Profile) définit comment les périphériques interagissent entre eux. Il gère la découverte, la connexion et la sécurité.

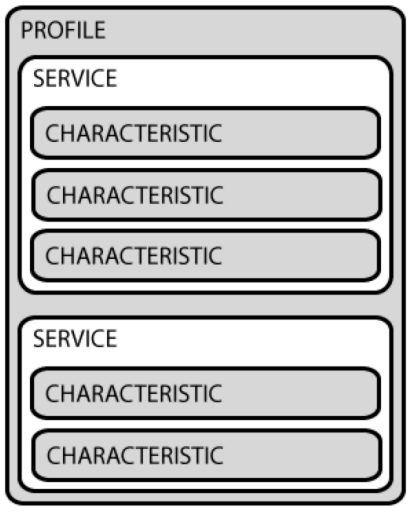

Figure 38, Hiérarchie GATT [17]

Le profil est organisé comme ceci. Il contient des services, qui eux-mêmes contiennent des caractéristiques. Par exemple, un profile pourrait être *Coffee Machine*, un service *State* et une caractéristique *Temperature*. C'est ces valeurs que nous allons accéder en HTTP, via le proxy.

4.3.3 Proxy HTTP/BLE

Bluetooth SIG propose des « White Paper » qui présentent différentes manières d'améliorer Bluetooth. Deux de ces « White Paper » traitent de cette problématique : GATT REST API [16] et GAP REST API [15].

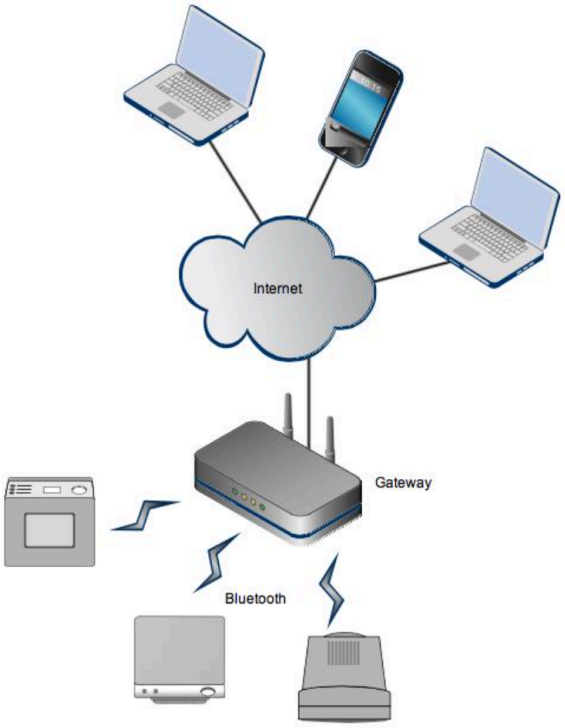

Figure 39, Illustration de l'architecture tirée de [16]

Dans cette architecture, nous avons un gateway qui fait l'interface entre le monde de l'Internet et le monde Bluetooth Low Energy. Le gateway doit connaitre les profils BLE supportés par les objets, ainsi que les informations GAP nécessaires à la connexion.

Du côté Internet, une interface REST est proposée de manière à définir de quelle manière les ressources sont accessible (GET pour obtenir une valeur et PUT pour en inscrire une nouvelle). Cette interface est stateless, ce qui signifie que le gateway n'a pas besoin de mémoriser l'état des clients, et que plusieurs clients peuvent accéder la même ressource en parallèle. Si la fonction « notification » est activée pour une caractéristique, (nouvelles valeurs envoyée automatiquement aux paires connectés), le gateway stockera la dernière valeur reçue.

Le « White Paper » [16] propose de structurer les URI HTTP comme ceci (exemple) :

- GET http://<gateway>/gatt/nodes
  - Découvrir les nœuds (objets) disponibles
- GET http://<gateway>/gatt/nodes/<node>
  - Découvrir les données d'un nœud spécifique
- GET http://<gateway>/gatt/nodes/<node>/services
  - Découvrir tous les services d'un nœud spécifique
- GET http://<gateway>/gatt/nodes/<node>/services/<service>/characteristics
  - Découvrir toutes les caractéristiques d'un nœud et d'un service spécifique
- GET http://<gateway>/gatt/nodes/<node>/serivces/<service>/characteristics/<value>
  - Lire la valeur d'une caractéristique
- PUT http://<gateway>/gatt/nodes/<node>/serivces/<service>/characteristics/<value>
  - Ecrire la valeur d'une caractéristique

Le format des réponses est également proposé. Le proxy doit retourner des réponses au format JSON. Par exemple en réponse à la lecture d'une caractéristique :

HTTP 200 – OK - application/json

```
{
      "self" : { "href" = "http://<gateway>/gatt/nodes/<node>/
                                          characteristics/<characteristic>"},
      "handle : "<characteristic>",
      "value" : "<value>" //Value as HEX string}
}
```

4.3.4 Overhead

Afin d'avoir une certaine cohérence avec les mesures sur Ethernet, nous allons comparer l'overhead généré en-dessus de la couche L2CAP, d'une part pour une architecture *6loWPAN-IPv6-UDP-CoAP*, et d'autre part avec du Bluetooth Low Energy utilisant *GATT*. La partie GAP est utilisés dans les deux cas et sera par conséquent omise de l'analyse.

Nous allons utiliser un nom de topic similaire aux tests précédents, à savoir « home/office/device/4 », d'une taille de 21 bytes. La valeur retournée est également de 21 bytes, à savoir « this is a simple test ». La charge utile est donc de 42 bytes dans le cas de la stack IPv6.

Dans le cas de BLE sans IP, la requête interroge un ID (appelé *handle*), codé sur 16 bits. Les 21 bytes représentant le nom de la caractéristique ne sont donc pas transmis lors de la requête. La réponse sera également « this is a simple test » (21 bytes). Nous avons donc une charge utile de 23 bytes.

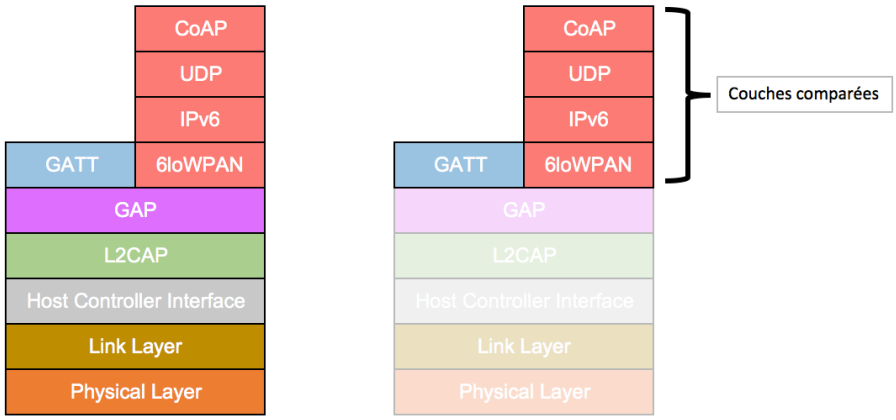

Figure 40, Stack BLE IPv6 vs No-IP

Dans le cas de l'architecture IPv6 :

- La requête CoAP (couche applicative uniquement) fait 44 bytes, et la réponse 30 bytes
- Nous prenons un entête UDP de 8 bytes dans les deux sens
- Un entête IPv6 compressée à 7 bytes (pire des cas) dans les deux sens
- Un entête 6loWPAN de 4 bytes pour le premier fragment, puis 5 bytes pour les suivants
    - 4 paquets BLE 4.1 pour la requête
    - 3 paquets BLE 4.1 pour la réponse

Ce qui représente un total de 137 bytes pour une charge utile de 42 bytes, soit 70% d'overhead.

Dans le cas de l'architecture BLE No-IPv6 :

- La requête est de 3 bytes (1 byte pour l'opération et 2 bytes de *handle*)
- La réponse est de 22 bytes (1 byte pour l'opération et 21 bytes pour la valeur)

```
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
Bluetooth Attribute Protocol
  Opcode: Read Request (0x0a)
  Handle: 0x000f
```

Figure 41, Exemple de requête BLE

```
Bluetooth Low Energy Link Layer
Bluetooth L2CAP Protocol
Bluetooth Attribute Protocol
  Opcode: Read Response (0x0b)
  Value: 5f
```

Figure 42, Exemple de réponse BLE

Ce qui représente un total de 25 bytes pour une charge utile de 23 bytes, soit 8% d'overhead.

4.3.5 Conclusion

L'utilisation d'une architecture sans IP allègerait grandement le nombre d'échanges entre les objets et la passerelle, ce qui permettrai d'importants gains d'énergie.

Cependant l'apport d'une architecture IPv6 nous permet de rendre plus intelligent l'objet lui-même, qui peut répondre aux requêtes directes des clients CoAP. Le proxy n'a pas besoin d'interpréter les requêtes des clients pour les exécuter. Il lui suffit juste de la transmettre sur le réseau Bluetooth pour que l'objet puisse répondre au client.

Avantages BLE only	Avantages IPv6 sur BLE
Nettement moins d'overhead nécéssaire pour les données transmisent.	Objet identifiable de manière unique dans le monde.
Gain en énergie du fait du nombre de paquet à échanger entre le pont et les objets.	L'objet peut communiquer via des protocoles applicatifs indépendants du type de réseau physique (Bluetooth, ZigBee,...). Le proxy peut alors travailler de manière agnostique.
Découverte des objets et des caractéristiques incluse dans BLE.	Découverte au niveau IPv6 et CoAP.
L'interaction entre le proxy et les objet est complètement normalisée.	
Désavantages BLE only	**Désavantages IPv6 sur BLE**
L'implémentation du proxy dépend du type de protocole de communication inférieur utilisé (BLE, ZigBee,...).	Grande quantité d'overhead généré par la stack TCP/IPv6 ou UDP/IPv6. Cela demande plus d'énergie aux objets, que ce soit pour la transmission des bytes sur plusieurs paquets BLE, ou pour la gestion de la stack.
Gros travail d'interprétation des requêtes HTTP des clients par le proxy.	Pas encore complètement normalisé (6loWPAN spécialement)
Le proxy doit connaître les caractéristiques des objets.	

En conclusion, l'utilisation d'IPv6 jusqu'aux objets nécessite moins « d'intelligence intermédiaire », au détriment de l'énergie nécessaire à maintenir cette stack. Elle apporte également plus d'intelligence au niveau applicatif, tel que la découverte dans des objets par CoAP. Dans le cadre de contraintes d'énergie très limitées (très longue durée de vie sur batterie), cette solution n'est pas adéquate. Il faudrait alors privilégier une architecture BLE pur.

Ce projet utilisera une architecture IPv6, telle que présentée dans le cahier des charges.

5 Interconnexion basique

Les choix concernant le protocole à utiliser ainsi que l'architecture du projet ayant été effectuée, il convient maintenant de concevoir concrètement ce que nous allons mettre en place afin de réaliser tout d'abord une interconnexion basique entre un client et les objets.

Tel qu'abordé au point 3.3.13, afin de pouvoir atteindre les objets depuis l'internet, l'idée était d'utiliser la passerelle en mode « mesh-under », afin de permettre une transparence de la couche 3. Cependant, ceci aurait pour effet de placer les objets directement dans le domaine de broadcast du réseau IP classique. Autrement dit, chaque paquet serait converti à l'aide de 6loWPAN et envoyé aux différents devices. Les devices devant être low energy, et communicant à l'aide de Bluetooth, cela pose un problème. Effectivement, une telle architecture ne permettrait alors pas de garantir une économie d'énergie.

Afin de pouvoir isoler les objets de tout trafic non-souhaité, il est nécessaire de les placer dans un sous-réseau différent. Le gateway fonctionnerait alors comme un router, reliant deux sous-réseau distincts, un pour les objets, et un autre pour le réseau classique.

5.1 Problème d'adressage IPv6

5.1.1 Présentation

Nous souhaitons que les objets obtiennent une adresse IPv6 globale, de manière à pouvoir être adressés de manière unique dans l'Internet mondiale. Pour ce faire, les objets doivent construire leur adresse à partir du préfix /64 annoncé par le default gateway IPv6, qui a lui-même reçu un préfix de l'Internet. Le préfix qu'il annonce est routé au niveau mondial. Les objets du LAN construisent leur adresse globale IPv6 à l'aide de ce préfix (64 premiers bits) et de leur propre mac adresse (48 bits à 64 bits à l'aide de la procédure EUI-64).

Le routeur par défaut « annonce » le préfix via des messages appelés « Router Advertisment » (RA). Ces messages doivent parvenir jusqu'aux objets. Pour ce faire, le proxy Ethernet/6loWPAN doit agir en mode « mesh-under ». Il doit forwarder tout le trafic IP d'un côté et de l'autre.

L'inconvénient est que tout le trafic du LAN sera envoyé sur le réseau Bluetooth, ce qui n'est évidemment pas souhaitable pour des questions d'énergie et d'optimisation du médium.

Cette solution nécessiterai de « bridger » l'interface Ethernet et l'interface Bluetooth sur le proxy (niveau 2). Or il n'est pas possible de le faire sur un système linux, car le bridging s'effectue au niveau de la couche Ethernet. Dans notre cas, nous n'avons pas d'Ethernet du côté Bluetooth. Le système refuse donc de « lier » les interfaces.

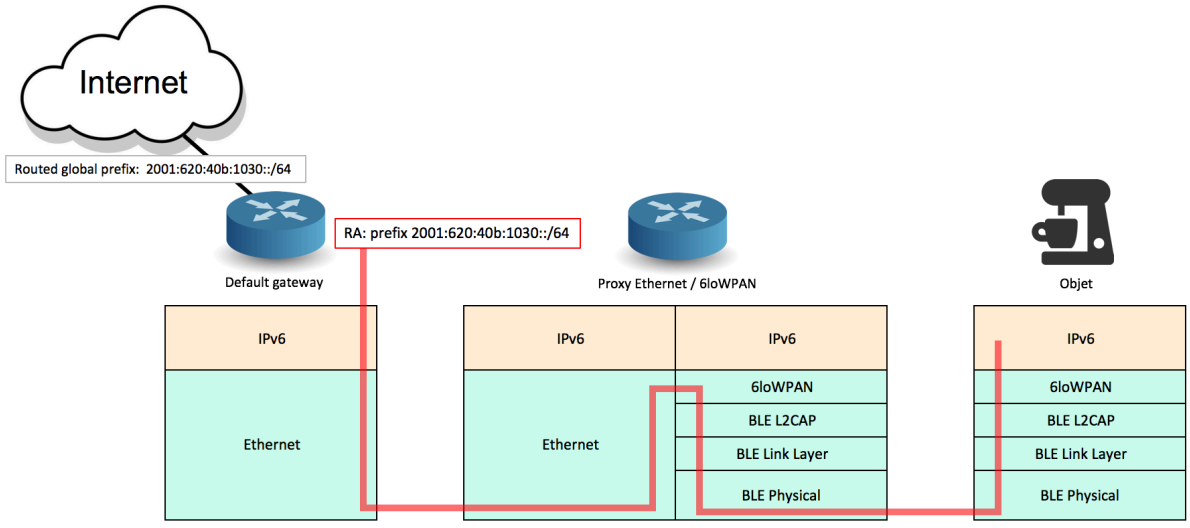

Figure 43, Architecture initiale « mesh-under » avec attribution de préfix externe

5.1.2 Solutions

Plusieurs solutions sont envisageables :

- Première solution (idéale) : Développer une application dont le rôle sera de transmettre uniquement le trafic à destination des objets, dont les *Router Advertisment* nécessaire à l'attribution du préfix. Cette solution ferait certainement l'objet d'un travail de semestre à part entière.
- Deuxième solution : Configurer le proxy comme un routeur à part entière sur le réseau, qui route un préfix unique dans l'Internet mondiale. Dans le cadre de l'école, cela nécessiterai que les routeurs de l'écoles ajoutent une route vers notre proxy/routeur, qui gèrerait un préfix /64 unique. Rendre le processus dynamique avec un protocole de routage tel qu'OSPF compliquerait la tâche. De plus, cette solution n'est pas envisageable dans un cadre autre que l'école.
- Troisième solution : Sur le proxy, ajouter une interface Ethernet « virtuelle » pour chaque objet connecté en Bluetooth. L'interface recevra une adresse IPv6 de manière standard, qu'il sera possible de lier (NAT) à l'adresse *Link Local* (fe80::) des objets. Cette solution présente l'avantage de conserver des échanges à l'aide des adresses *Link Local* sur le réseau Bluetooth, ce qui permet une meilleure compression de l'entête IPv6 par 6loWPAN.
- Quatrième solution : Attribuer un préfix /64 imaginaire (par exemple 5000::/64) du côté Bluetooth et le diffuser vers les objets. Laisser le routeur de de l'école nous donner le préfix /64 pour l'interface Ethernet. Activer le routage sur le proxy afin de permettre aux paquets de transiter d'une interface à l'autre si l'adresse IP de destination l'exige. Ajouter une route statique sur nos ordinateurs de développement qui indique de rediriger le trafic à destination de ce réseau imaginaire vers l'adresse globale du proxy (2001:620:40b…). Ceci est une solution de laboratoire, qui ne serait pas applicable dans le cas d'une commercialisation du produit.

Compte tenu du temps à disposition et de l'orientation du projet, il est décidé d'opter la quatrième solution, qui présente l'avantage d'être facilement et rapidement mise en place. Cette option nous permettra de faire abstractions des problèmes liés à IPv6 et aller de l'avant dans le déroulement du projet.

5.2 Installation du proxy BLE – Ethernet

Nous avons choisi d'utiliser un « Raspberry Pi 2 » afin de mettre en place le proxy Bluetooth Low Energy – Ethernet. Le Raspberry possède un port Ethernet RJ45, et il est nécessaire de lui rajouter un dongle USB Bluetooth 4.0 (au minimum). Le système d'exploitation Raspbian utilisé est « Debian Wheezy », dont la dernière version stable est sortie en Mai 2015.

Une fois le système d'exploitation installé, il faut lancer les commandes suivantes afin d'effectuer la connexion avec le nRF51 via Bluetooth. *Note : Il faut que le nRF51 soit chargé avec un programme gérant au minimum la stack IPv6. Dans notre cas, n'importe quel programme de test mis à disposition dans le SDK IoT de Nordic fera l'affaire.*

5.2.1 Préparation

Installation de bluez pour les outils Bluetooth, et radvd pour distribuer un préfix IPv6 du côté Bluetooth.

```
$ sudo apt-get install bluez radvd
```

Editer le fichier de configuration /etc/radvd.conf comme ceci.

```
interface bt0
{
        AdvSendAdvert on;
        prefix 5000::/64
        {
                AdvOnLink off;
                AdvAutonomous on;
                AdvRouterAddr on;
        };
};
```

Trouver la mac@ du nrf51 avec la commande :

```
$ hcitool lescan
```

Cette mac@ ne va pas changer. Nous pouvons ensuite déterminer l'adresse IPv6 correspondante à l'aide du processus EUI-64 (rfc 7043).

Par exemple, 00:7D:07:5A:11:F8 devient 5000::27D:7FF:FE5A:11F8 *(Ou local avec FE80::)*

L'adresse IPv6 trouvée sera toujours la même pour cette mac@ !

5.2.2 Configuration au démarrage

La configuration suivant est à faire après chaque démarrage du système. Il est possible d'automatiser cette procédure. Cependant, il faut faire attention à bien laisser quelques secondes entre chaque commande. Certaines peuvent prendre du temps à s'exécuter.

```
# mount debugfs file system
$ mount -t debugfs none /sys/kernel/debug

# set PSM channel as 0x23 (35)
$ echo 35 > /sys/kernel/debug/bluetooth/6lowpan_psm

# connect the device by using his mac address (can be founded with « hcitool lescan » command)
echo "connect 00:7D:07:5A:11:F8 1" > /sys/kernel/debug/bluetooth/6lowpan_control

# add ipv6 public route to the bt0 interface
ifconfig bt0 add 5000::1/64

# enable IPv6 packet forwarding
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

5.2.3 Ping6

Maintenant que le nrf51 est connecté et a reçu une adresse IPv6 publique, nous pouvons tester la connectivité.

Depuis le Raspberry, avec l'adresse IPv6 link-local (spécifier l'utilisation du port bt0) :

```
pi@capsule ~ $ ping6 fe80::27d:7ff:fe5a:11f8%bt0
PING fe80::27d:7ff:fe5a:11f8%bt0(fe80::27d:7ff:fe5a:11f8) 56 data bytes
64 bytes from fe80::27d:7ff:fe5a:11f8: icmp_seq=1 ttl=64 time=111 ms
64 bytes from fe80::27d:7ff:fe5a:11f8: icmp_seq=2 ttl=64 time=123 ms
64 bytes from fe80::27d:7ff:fe5a:11f8: icmp_seq=3 ttl=64 time=134 ms
```

Depuis le Raspberry, avec l'adresse IPv6 publique :

```
pi@capsule ~ $ ping6 5000::27d:7ff:fe5a:11f8
PING 5000::27d:7ff:fe5a:11f8(5000::27d:7ff:fe5a:11f8) 56 data bytes
64 bytes from 5000::27d:7ff:fe5a:11f8: icmp_seq=1 ttl=64 time=179 ms
64 bytes from 5000::27d:7ff:fe5a:11f8: icmp_seq=2 ttl=64 time=189 ms
64 bytes from 5000::27d:7ff:fe5a:11f8: icmp_seq=3 ttl=64 time=201 ms
```

Nous constatons que les pings effectués avec l'adresse link-local (fe80::) sont plus performant que ceux effectués avec l'adresse publique (5000::). Cela est probablement dû à la compression de l'entête IPv6 qui est plus importante pour les adresses link-local.

Pour effectuer un ping depuis un PC quelconque du LAN, il est nécessaire de rajouter la route statique vers le réseau 5000::/64 en passant par l'adresse IPv6 publique du Raspberry. Sur mac os, la commande ressemble à ceci :

```
$ sudo route -n add -inet6 5000::/64 2001:620:40b:1030:36d9:89af:2ed7:c9db
```

Ping depuis un ordinateur du LAN :

```
MacBook-Pro-de-Mickey:~ valentinchassot$ ping6 5000::27d:7ff:fe5a:11f8
PING6(56=40+8+8 bytes) 2001:620:40b:1030:bc0c:21af:f3c8:f1b7 --> 5000::27d:7ff:fe5a:11f8
16 bytes from 5000::27d:7ff:fe5a:11f8, icmp_seq=0 hlim=63 time=75.570 ms
16 bytes from 5000::27d:7ff:fe5a:11f8, icmp_seq=1 hlim=63 time=81.727 ms
16 bytes from 5000::27d:7ff:fe5a:11f8, icmp_seq=2 hlim=63 time=98.887 ms
```

6 Réalisation

6.1 Démonstrateur

Un démonstrateur sera mis en place afin de tester diverses fonctionnalités. Il sera notamment possible de :

- Afficher une page « formaté » fournie par un serveur HTTP installé sur le Raspberry
- Obtenir l'état d'une led du nrf51
- Allumer ou éteindre cette même led depuis la page web (en cliquant sur la led). Une requête AJAX « CoAP in HTTP PUT » sera alors envoyée.
- Obtenir l'état d'un compteur situé dans le nrf51
- Activer la mise à jour « live » de ces informations
    - Par des requêtes AJAX périodiques qui font des requêtes « CoAP in HTTP GET » à l'attention du nrf51
    - Par un web socket ouvert entre le client et serveur, qui *PUSH* les changements dès qu'un évènement CoAP parvient au Raspberry. Le Raspberry est un client CoAP du nrf51 en mode « Observe »

6.2 Choix du langage serveur

Un des critères prépondérants au choix du langage était l'existences de librairies. Effectivement, développer nous-même une librairie CoAP n'était pas envisageable, ne serait-ce qu'à cause du temps à disposition. Nous nous sommes donc concentrés sur les librairies disponibles pour différents langages.

Après l'analyse de plusieurs librairies Java, Python et Node.js, il s'est avéré que la librairie « node-coap » [26] disponible pour le langage Node était la plus complète et la mieux documentée. Notre choix du langage s'est donc porté sur Node.js. Il s'agit en fait de code JavaScript exécuté côté serveur.

6.3 Proxy CoAP – HTTP

Le code ci-dessous permet de convertir une requête HTTP en requête CoAP. Concrètement, l'url de la ressource CoAP est extraite de l'url HTTP, conformément à la concaténation expliquée au point 4.2.8. Selon les paramètres extraits, une requête CoAP est ensuite créée puis exécutée.

Les données retournées par le nœud CoAP sont tout d'abord stockées dans une variable, puis lorsque la réponse est complète, le contenu est défini comme body de la page web retournée au client.

Si par hasard le format de l'url concaténée n'est pas valide, une erreur « 400, Bad request » est envoyée au browser.

```
// handle CoAP in HTTP GET and act as a proxy
app.get('/:coap*', function(req, res) {

    // extract trailling part from http request
    var request_params = req.params.coap+req.params[0];

    // check if it is a coap request
    if(request_params.substr(0,7) == 'coap://') {

        // print debug in console
        console.log("COAP REQ:\t"+request_params);

        // get params
        var coap_params = request_params.substr(7).split('/');
        var coap_host   = coap_params[0];
        var coap_path   = '/'+coap_params.slice(1).join('/');

        // create request
        var coap_request = coap.request({
            hostname: coap_host,
            pathname: coap_path
        });

        // get output
        var coap_return = '';
        coap_request.on('response', function(coap_res) {
            coap_res.on('data', function(coap_data) {
                // append data
                coap_return += coap_data;
            });
            coap_res.on('end', function(){
                // query is finished, all data is now in coap_return variable
                // return JSON format
                res.writeHead(200, {"Content-Type": "application/json"});
                res.end('{"payload" : '+coap_return+'}');
            });
        });

        // execute request
        coap_request.end();
    } else {
        //not a coap request
        res.writeHead(400);
        res.end("400 Bad Request");
    }
});
```

6.4 Codage du nRF51

6.4.1 Installation du softdevice

Tout d'abord, le nRF51 nécessite un softdevice particulier afin d'être à même d'utiliser 6lowPAN. Afin d'installer ce softdevice, il est nécessaire d'installer l'application « nRFgo Studio ». Cette application est disponible sous [20].

Une fois installée, il est nécessaire de télécharger le softdevice disponible à l'adresse de la ressource [21], plus précisément le fichier présent sous

« nrf51/components/softdevice/s1xx_iot/ s1xx-iot-prototype2_softdevice.hex ».

Une fois cette ressource prête, il est possible de lancer l'application nRFgo Studio. Dès lors que le nRF51 est connecté, il apparait dans la liste « Device Manager » située sur la gauche :

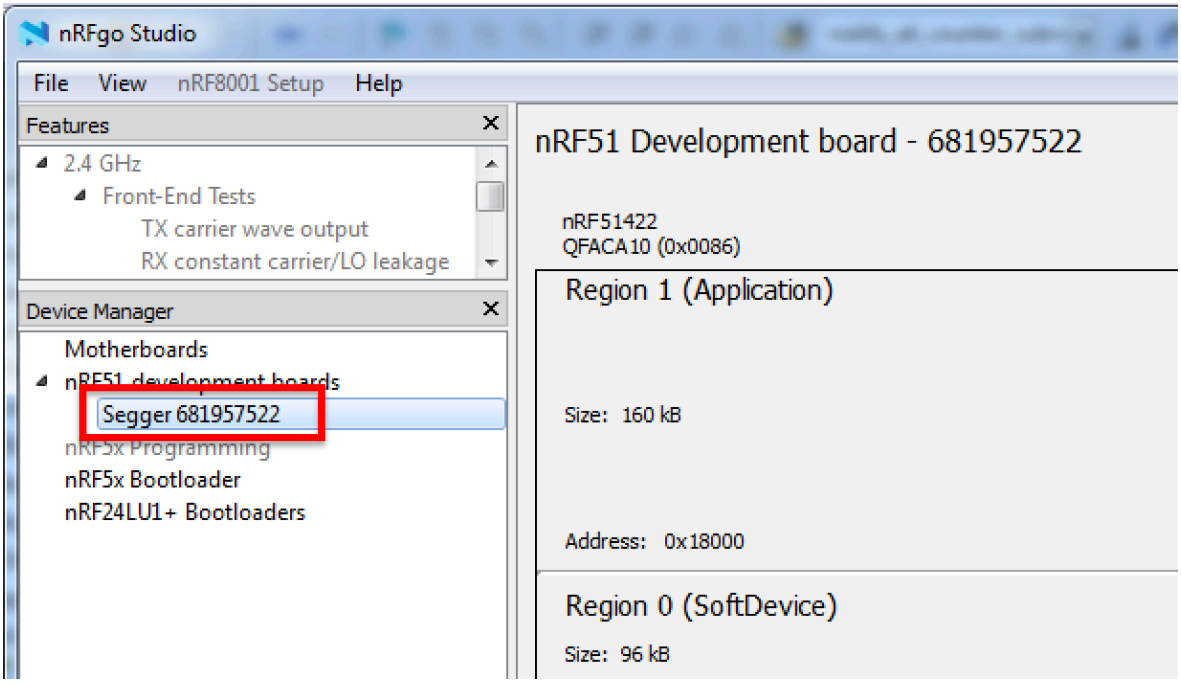

Figure 44, Liste des devices

Il faut sélectionner ce périphérique puis, sur la droite de l'application choisir « Browse » afin de sélectionner le fichier « .hex » (le softdevice) préalablement téléchargé :

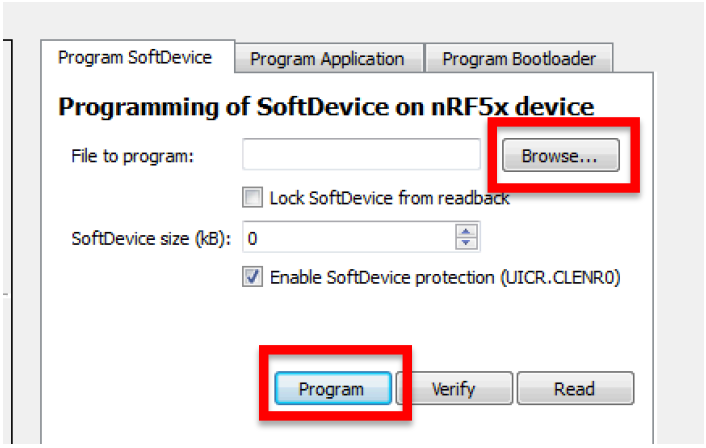

Figure 45, Installation du softdevice

Dès lors que le fichier « **s1xx-iot-prototype2_softdevice.hex** » est sélectionné, cliquez Program.

Le softdevice est maintenant installé.

6.4.2 Mise en place et utilisation de Keil

Afin de travailler sur le code et de le flasher sur le nRF51, nous allons utiliser Keil. La version du logiciel utilisé est la 5.17.0.0. Une fois le logiciel installé, il faut encore ajouter le « pack » Nordic. Pour ce faire, il faut cliquer sur l'icône correspondante :

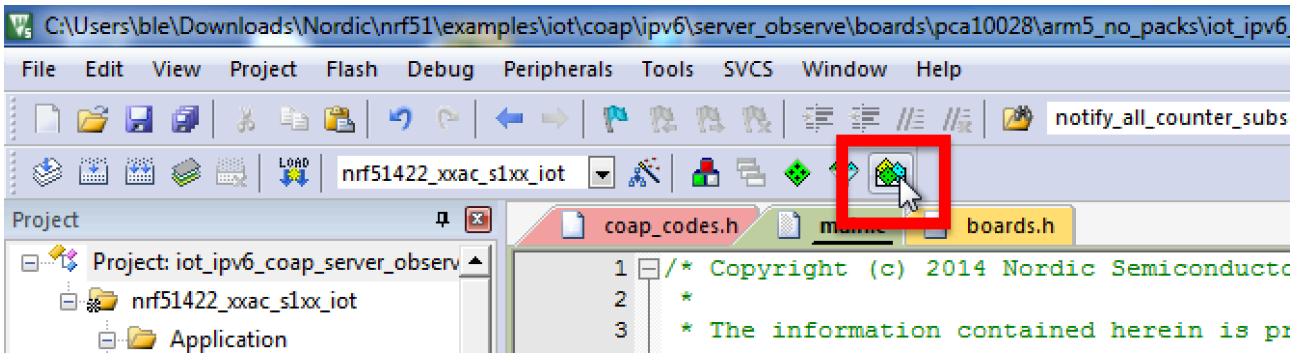

Figure 46, Accès au menu des « packs additionnels »

Il faut ensuite installer les packs suivant en cliquant sur le bouton « install ». Les packs à installer sont indiqués ci-dessous :

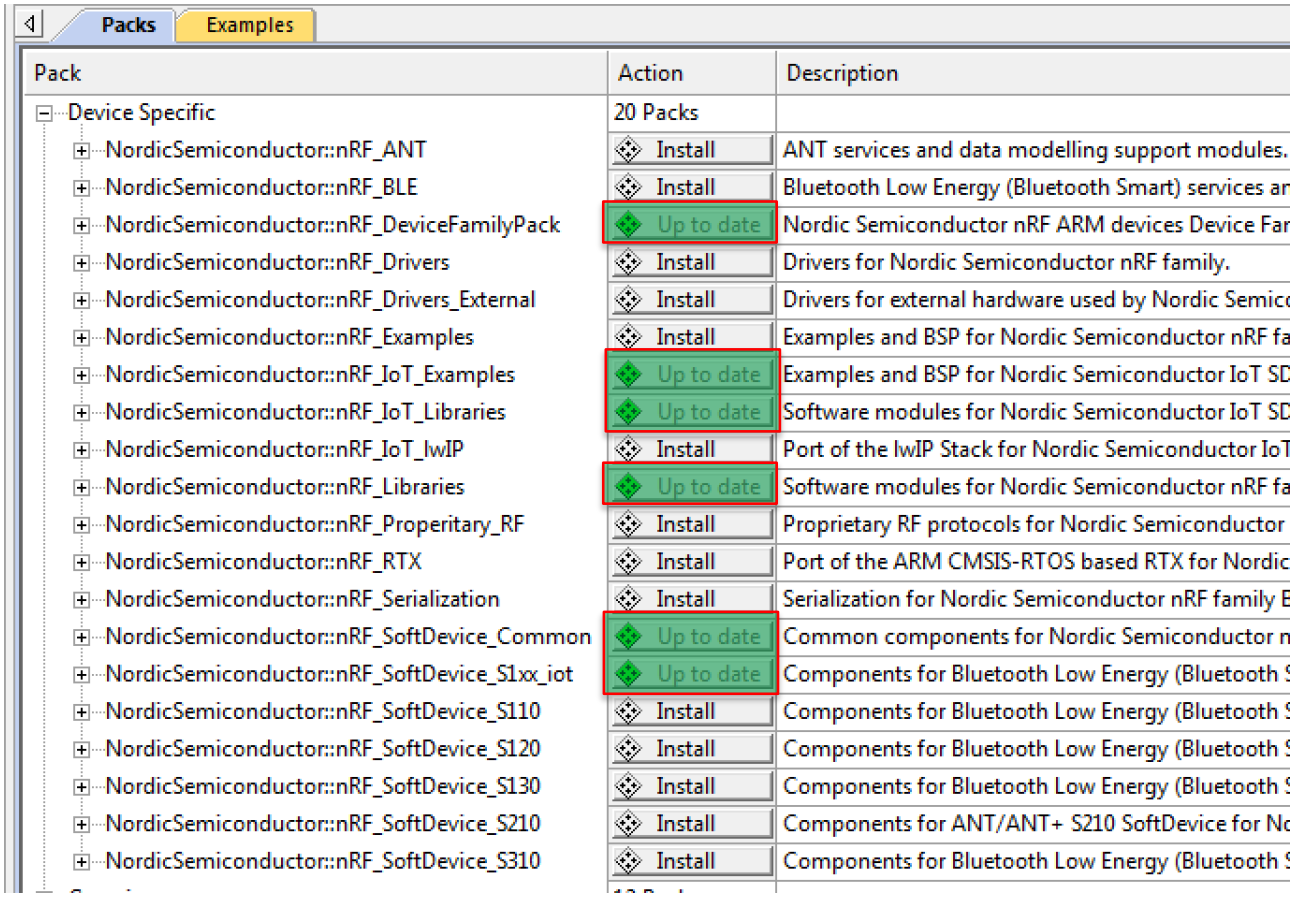

Figure 47, Packs à installer

Il est ensuite possible d'ouvrir le projet Keil correspondant au code que nous souhaitons utiliser. Nous allons utiliser comme base le code « server_observe » présent dans la ressource [21] téléchargée précédemment.

Le fichier projet se trouve plus précisément sous le répertoire :

« nrf51/examples/iot/coap/ipv6/server_observe/boards/pca10028/arm5_no_packs/ »

Il s'agit du fichier nommé « **iot_ipv6_coap_server_observe_pca10028.uvprojx** ».

Il est alors possible de l'ouvrir :

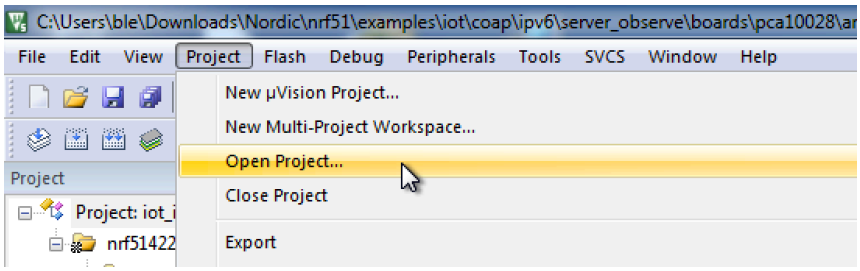

Figure 48, Ouverture du projet

Puis choisir le fichier « iot_ipv6_coap_server_observe_pca10028.uvprojx ».

Il est alors possible de parcourir l'arborescence située à gauche pour trouver le fichier « main.c » :

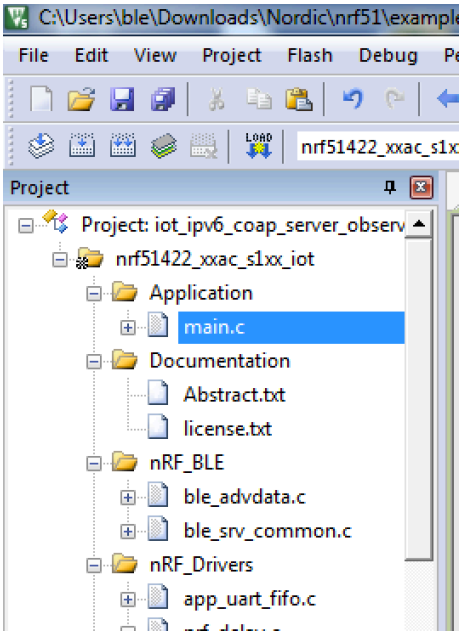

Figure 49, Fichier « main.c »

Il est alors possible de l'ouvrir en effectuant un double clic dessus.

La compilation du code s'effectue en cliquant sur l'icône suivante (il est préférable de sauver le fichier avant de compiler) :

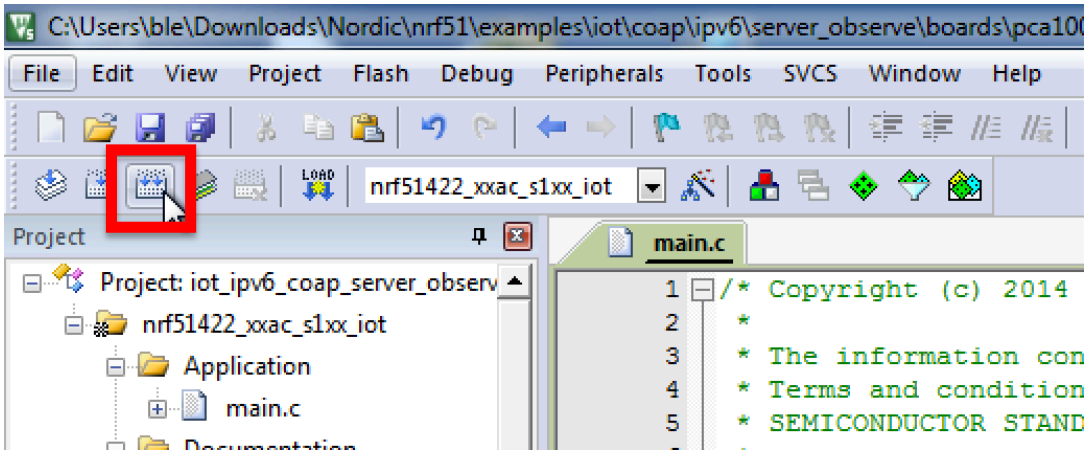

Figure 50, Compilation du projet

Un log indique le résultat de la compilation en bas de l'application :

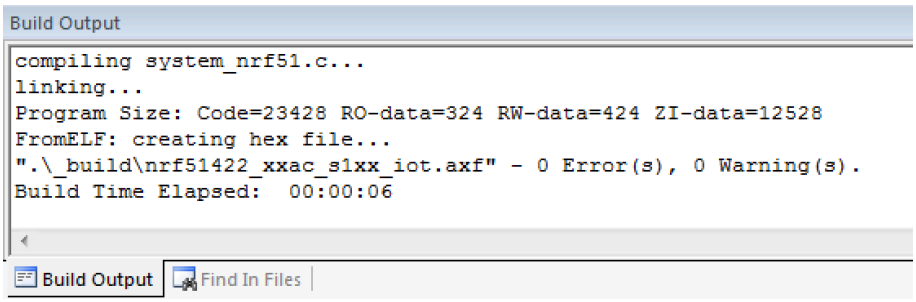

Figure 51, Log de la compilation

Dans ce cas précis, le projet a été compilé sans erreur, sans warning, en 6 secondes.

Une fois le projet correctement compilé, il est possible de le charger dans la mémoire flash du nRF51. Pour ce faire il faut cliquer sur l'icône suivante :

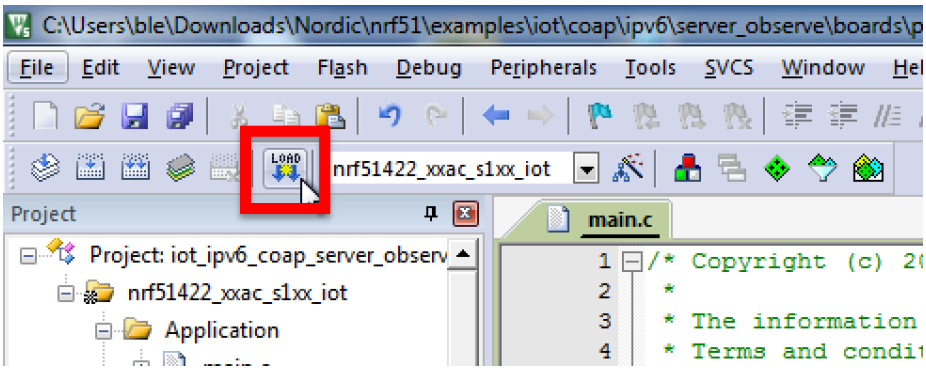

Figure 52, chargement du projet dans la mémoire flash

Un log indique le résultat de chargement en bas de l'application :

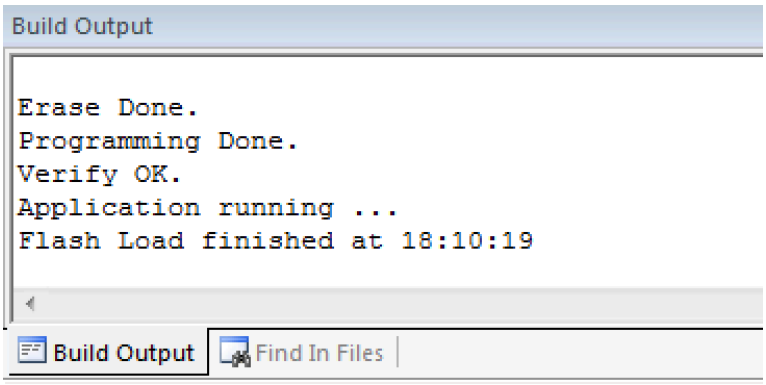

Figure 53, Log du chargement

Dans ce cas précis, le projet a correctement été chargé dans la mémoire flash du nRF51.

Au cours du développement, les étapes de compilation et de chargement dans la mémoire flash du nRF51 seront à répéter.

6.4.3 Structure du code

Le code du fichier « main.c » du projet « server_observe » est structuré comme suit :

- Includes et defines
- Variables globales
- Fonctions système
- Initialisation des LEDs
- Initialisation des timers
- Initialisation de l'advertisment Bluetooth
- Fonction Bluetooth
- Handler IP
- Fonctions de l'application pour la récupération des variables
- Fonction de callback des observer CoAP
- Fonctions de notification des ressources CoAP
- Fonctions de callback des différentes ressources CoAP
- Initialisation des endpoints CoAP
- Handler des boutons
- Initialisation des boutons
- Initialisation de la stack IP
- Fonction système Power Manage
- Fonction de gestion du timer CoAP
- Fonction de gestion des erreurs
- Main

Cette structure est celle choisie par Nordic dans le code d'exemple, nous n'allons donc pas la remettre en question et nous y conformer.

Les éléments nous intéressant principalement sont :

Variables globales :

C'est ici que nous définirons une variable qui retiendra le nombre de cafés consommés. C'est également ici que nous créerons les variables globales représentant nos diverses ressources.

Fonctions de l'application pour la récupération des variables :

Ici nous ajouterons une ou plusieurs fonctions dont le but sera de récupérer la valeur des variables souhaitées et de préparer leur format pour les envoyer sur le réseau.

Fonctions de notification des ressources CoAP :

Il s'agit de fonctions qui seront appelées lorsqu'une notification aux différents observers CoAP sera requise, lors d'une mise à jour de la variable par exemple. Ces fonctions devront donc lors de leur appel générer et envoyer un message CoAP aux observateurs ayant souscrit à la ressource observable en question.

Fonctions de callback des différentes ressources CoAP :

Il s'agit de fonction qui seront appelées lorsqu'une requête est adressée à une ressource particulière. Ces fonctions devront donc lors de leur appel générer et envoyer un message CoAP à destination de l'utilisateur ayant demandé la ressource.

Initialisation des endpoints CoAP :

C'est ici que sont défini les différentes ressources CoAP disponibles sur le nRF51. Les ressources sont d'abord créées, puis attachées au point racine « / ». Ensuite, il est possible de définir la fonction de callback correspondance à cette ressource. C'est également ici que sont définies quelles fonctions offre la ressource (GET, PUT, OBSERVE).

Handler des boutons :

Ici, les actions effectuées lors de l'appui sur les boutons sont définies. C'est par exemple ici qu'il convient d'effectuer l'incrémentation du conteur. Si le bouton met à jour une ressource, c'est également ici qu'il faut préciser la méthode de notification des observers à appeler.

Initialisation des boutons :

Par défaut les boutons n'ont pas d'interaction configurée. C'est ici qu'il faut définir quelles actions seront effectuées lors de l'appui sur ces derniers. Nordic a défini que chaque bouton déclenche l'appel de la méthode handler (expliquée ci-dessus) qui elle dispatch ensuite l'action. Nous allons donc suivre cette organisation. Nous définirons donc que les boutons que nous souhaitons employer appellent le handler des boutons.

Fonction de gestion du timer CoAP :

C'est ici que sont gérés les compteurs de timer de chaque ressource observable. Ces timer définissent à quelle moment les messages observe doivent être de type CON au lieu de NON. Comme expliqué au point 4.2.11, afin de s'assurer qu'il y ait encore des observers intéressés par la réceptions de mises à jour, le serveur peut envoyer une notification de type CON, qui doit être acquittée par les utilisateurs toujours actifs.

6.4.4 Exemple de code

Variables globales :

```
// ressource name
static const char         m_counter_name[] = "counter";
// counter ressource
static coap_resource_t    m_counter;
// var holding number of time button 4 is pressed
uint32_t                  counter = 0;
```

Ici nous ajoutons une variable de ressource pour le compteur, ainsi que son label. Nous ajoutons aussi la variable qui retiendra l'état du compteur.

Récupération des variables :

```
static void counter_value_get(coap_content_type_t content_type, char ** str)
{
	// create an array of char for the response
	static char response_str[6];
	// fill it with '\0'
	memset(response_str, '\0', sizeof(response_str));
	// set it to the content of the counter value
	sprintf(response_str, "%d", counter);
	// save it to the given pointer
	*str = response_str;
}
```

Il s'agit d'une simple méthode qui récupère l'état actuel du compteur et le stock dans un tableau de char. Un tableau de char est requis car il se peut que le code retourne du texte en lien et place de nombre. Cette fonction pourrait également générer plusieurs types de données selon le support du client, par exemple il serait possible ici de mettre en forme la ressource selon le type JSON.

Notification des ressources CoAP :

```
static void notify_all_counter_subscribers(coap_msg_type_t type)
{
  // Fetch all observers which are subscribed to this resource.
  // Then send an update value too each observer.
  coap_observer_t * p_observer = NULL;
  while (coap_observe_server_next_get(&p_observer, p_observer, &m_counter) == NRF_SUCCESS)
  {
    // Generate a message.
    coap_message_conf_t response_config;
    memset(&response_config, 0, sizeof(coap_message_conf_t));

    response_config.type              = type;
    response_config.code              = COAP_CODE_205_CONTENT;
    response_config.response_callback = observer_con_message_callback;

    // Copy token.
    memcpy(&response_config.token[0], &p_observer->token[0], p_observer->token_len);
    // Copy token length.
    response_config.token_len = p_observer->token_len;
    response_config.port.port_number = COAP_SERVER_PORT;

    coap_message_t * p_response;
    uint32_t err_code = coap_message_new(&p_response, &response_config);
    APP_ERROR_CHECK(err_code);


    // Set custom misc. argument.
    p_response->p_arg = p_observer;

    // Set remote address
    err_code = coap_message_remote_addr_set(p_response, &p_observer->remote);
    APP_ERROR_CHECK(err_code);

    // Set observe sequence number
    err_code = coap_message_opt_uint_add(p_response, COAP_OPT_OBSERVE, m_observer_se-
quence_num++);
    APP_ERROR_CHECK(err_code);

    // Set observe max age
    err_code = coap_message_opt_uint_add(p_response, COAP_OPT_MAX_AGE, m_counter.expire_time);
    APP_ERROR_CHECK(err_code);

    // Set payload
    char * response_str;
    counter_value_get(p_observer->ct, &response_str);
    err_code = coap_message_payload_set(p_response, response_str, strlen(response_str));
    APP_ERROR_CHECK(err_code);

    // Send message
    uint32_t msg_handle;
    err_code = coap_message_send(&msg_handle, p_response);
    APP_ERROR_CHECK(err_code);

    // Delete message
    err_code = coap_message_delete(p_response);
    APP_ERROR_CHECK(err_code);
  }
}
```

Cette méthode génère et envoie un message CoAP lors de son appel. Le message est une notification envoyée à tous les observeurs de la ressource compteur.

Callback des ressources :

```
// Set response payload to the actual counter value.
char * response_str;
counter_value_get(ct_to_use, &response_str);
err_code = coap_message_payload_set(p_response, response_str, strlen(response_str));
APP_ERROR_CHECK(err_code);
```

Cette méthode est appelée lors d'une requête auprès de la ressource « counter ». Seul un exemple du code est présenté ici. C'est cette méthode qui a pour rôle en plus de répondre aux requête de mettre à jours la liste des observeurs. Effectivement, si la requête contient l'option « observe », cela signifie que le client veut être notifié en cas de changement. Le code présenté ici représente l'appel à la fonction « counter_value_get » qui récupère la valeur du compteur pour la mettre dans le payload.

Initialisation des endpoints CoAP :

```
// create resource "counter"
err_code = coap_resource_create(&m_counter, m_counter_name);
APP_ERROR_CHECK(err_code);

// append "counter" to / ==> /counter
err_code = coap_resource_child_add(&root, &m_counter);
APP_ERROR_CHECK(err_code);

// set params for counter
m_counter.permission      = (COAP_PERM_GET | COAP_PERM_PUT | COAP_PERM_OBSERVE);
m_counter.callback        = counter_callback;
m_counter.ct_support_mask = COAP_CT_MASK_APP_JSON | COAP_CT_MASK_PLAIN_TEXT;
m_counter.max_age         = 15;
```

Ce code issu de la méthode “coap_endpoints_init” démontre l'ajout d'une ressource. Ici il s'agit de la ressource “counter”, elle est tout d'abord créée, puis ajoutée à la ressource “root”. Finalement, différents paramètres tels que le “callback” sont définis.

Handler des boutons :

```
if (pin_no == BUTTON_FOUR)
{
  counter++;
  notify_all_counter_subscribers(COAP_TYPE_NON);
}
```

Ce code est ajouté au handler existant. Il permet de définir ce qui doit être effectué lors de l'appui sur le bouton 4. S'agissant du bouton de compteur, la variable est tout d'abord incrémentée, puis la méthode de notification des abonnés est appelée.

Initialisation des boutons :

```
static app_button_cfg_t buttons[] =
{
  {BUTTON_ONE, false, BUTTON_PULL, button_event_handler},
  {BUTTON_FOUR, false, BUTTON_PULL, button_event_handler}
};
```

Le code d'exemple définissait une action au bouton 1, ici nous avons ajouté l'action pour le bouton 4. Les deux boutons appellent la méthode « button_event_handler », et c'est dans cette méthode que l'action à proprement parler est exécutée.

Gestion du timer des observers :

```
// Create a message counter for the "counter" resource
static uint32_t msg_count_counter = 0;
// If the "max age" of a resource is reached
if (m_counter.expire_time <= (0 + OBSERVE_NOTIFY_DELTA_MAX_AGE))
{
  // Reset the expire timer
  m_counter.expire_time = m_counter.max_age;

  // Every 1 in 4 messages, send a CON, otherwise, send a NON
  if (msg_count_counter % 4 == 0)
  {
    notify_all_counter_subscribers(COAP_TYPE_CON);
  }
  else
  {
    notify_all_counter_subscribers(COAP_TYPE_NON);
  }

  // Increment the message counter
  msg_count_counter++;
}
else
{
  // Decrement the "remaining time"
  m_counter.expire_time--;
}
```

Ce code est ajouté à la fonction “app_coap_time_tick”. Cela permet tous les 4 messages de notification d’observation déclenchés à l’aide du timer “max age” d’envoyer un message CON.

6.5 Code du démonstrateur, côté serveur

Le code du fichier server.js est disponible en intégralité sur repository git du projet.

6.5.1 socket.io

Node.js possède un module très puissant appelé socket.io. Il permet de maintenir une connexion ouverte entre le serveur et le client. Ceci permet aux deux intervenants de s'échanger des messages à tout moment, et dans les deux sens. Le code ci-dessous tiré de *server.js* présente la manière dont réagit le serveur quand un nouveau client se connecte, ainsi que le traitement accordé aux messages qu'il reçoit.

```
// on connection
io.sockets.on('connection', function (socket) {

    // ack connection
    socket.emit('connected', true);

    // print client IP address on connection
    console.log('Connect:\t'+socket.handshake.address);

    // on disconnect
    socket.on('disconnect', function () {
        console.log('Disconnect:\t'+socket.handshake.address);
        // remove subscriber from subscriber's array if present
        remove_subscriber(socket.id);
    });

    // on "push" message from client
    socket.on('push', function (flag) {
        // if flag = true, we add this client's socket to subscriber's list
        if(flag) {
            console.log('Enable Push:\t'+socket.handshake.address);
            push_subscribers.push(socket);
            // directly refresh values
            socket.emit('update_cnt', counter_value);
            socket.emit('update_led', led_value);
        } else {
            // otherwise remove it, this client doesn't want push anymore
            console.log('Disable Push:\t'+socket.handshake.address);
            remove_subscriber(socket.id);
        }
    });
});
```

6.5.2 Route avec express

L'extension Node « express » permet de traiter les requêtes HTTP parvenant au serveur. Il est possible de traiter différemment les requêtes GET, POST, PUT ou DELETE. L'extrait ci-dessous présente le traitement d'un GET sur la racine « / ». Le serveur lui retourne un rendu de la page index.ejs. Cette page est générée à l'aide de l'extension Node « ejs ».

```
// serve index for HTTP GET /
app.get('/', function(req, res) {
    // send back index.ejs with initials values injected into the code
    res.render('index.ejs', {
      initial_counter: counter_value,
      initial_led: led_value
    });
});
```

6.6 Code du démonstrateur, côté client

Le code du fichier index.ejs est disponible en intégralité sur repository git du projet.

6.6.1 socket.io

Du côté client, la connexion doit être ouverte avec le serveur. Une fois mise en place, le client recevra des notifications sous forme de messages « update_led » ou « update_cnt ».

Ces notifications parviendront au client uniquement s'il s'est abonné aux notifications Push avec un « *socket.emit('push', 1);* », déclenché par l'activation du bouton adéquat.

```
// connect socket.io
socket = io.connect();

// log connected event
socket.on('connected', function () {
    console.log("Connected");
});

// update counter on push from server
socket.on('update_cnt', function (value) {
     cnt.update(value);
});

// update led state on push from server
socket.on('update_led', function (value) {
     set_led_state(value);
});
```

6.6.2 Requêtes Ajax

Dans le cas d'une mise à jour à l'aide de requêtes Ajax périodiques, le client effectue des requêtes GET afin d'obtenir l'information en « live », directement de l'objet. Pour ce faire, il fait une requête CoAP en HTTP. La fonction de proxy CoAP-HTTP du serveur sera activée. La librairie JavaScript jQuery 2.1.4 est utilisée dans ce projet. Ci-dessous, une requête GET « CoAP in HTTP » afin de connaitre l'état de la led.

```
$.ajax({
    method: "GET",
    url: "/coap://5000::27d:7ff:fe5a:11f8/lights/led3",
}).done(function (msg) {
    set_led_state(msg.payload);
});
```

Dans le cas d'une requête de mise à jour de l'état de la led (écriture), il faut effectuer un PUT. Notons que le fait d'écrire « 2 » a pour effet d'inverser l'état actuel de la led.

```
$.ajax({
    method: "PUT",
    url: "/coap://5000::27d:7ff:fe5a:11f8/lights/led3",
    data: {payload: "2"},
    dataType: "json"
}).complete(function (msg) {
    // do nothing on return, led will be updated by updaters (PUSH or AJAX)
    console.log("PUT Request return: "+msg.status);
});
```

7 Tests de fonctionnement

7.1 Démonstrateur

Le démonstrateur mis en place remplit toutes les fonctionnalités annoncées. Il est ainsi possible de :

- Afficher une page « formaté » fournie par un serveur HTTP installé sur le Raspberry
- Obtenir l'état d'une led du nrf51
- Allumer ou éteindre cette même led depuis la page web (en cliquant sur l'image représentant la led). Une requête AJAX « CoAP in HTTP PUT » est alors envoyée.
- Obtenir l'état d'un compteur situé dans le nrf51
- Activer la mise à jour « live » de ces informations à l'aide de boutons toggle (on/off)
    - Par des requêtes AJAX périodiques qui font des requêtes « CoAP in HTTP GET » à l'attention du nrf51
    - Par un web socket ouvert entre le client et serveur, qui *PUSH* les changements dès qu'un évènement CoAP parvient au Raspberry. Le Raspberry est un client CoAP du nrf51 en mode « Observe »

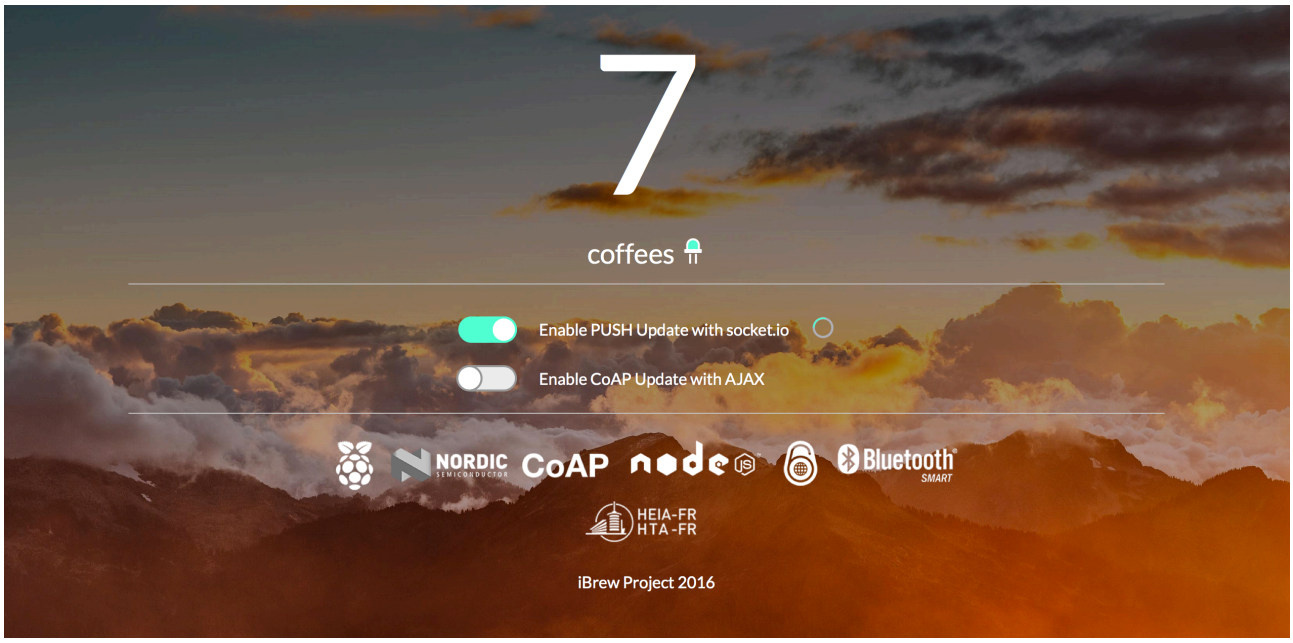

Figure 54, Vue de la page finale délivrée par le serveur situé sur le Raspberry Pi

7.2 Test énergétique

Afin de tester la durée de vie du périphérique sur la pile, nous avons effectué une mesure de consommation de courant. Nous réalisons la mesure dans une condition type « standard » d'utilisation définie selon nos use case. Ensuite, nous extrapolons les valeurs afin d'estimer la durée de vie de la pile du dispositif.

7.2.1 Préparation

Tout d'abord, il est nécessaire de faire quelques adaptations physiques sur le nRF51. Ces modifications sont documentées dans le document [22] au chapitre 5.7 « Measuring Current ». Il faut tout d'abord sectionner le contact « SB9 » :

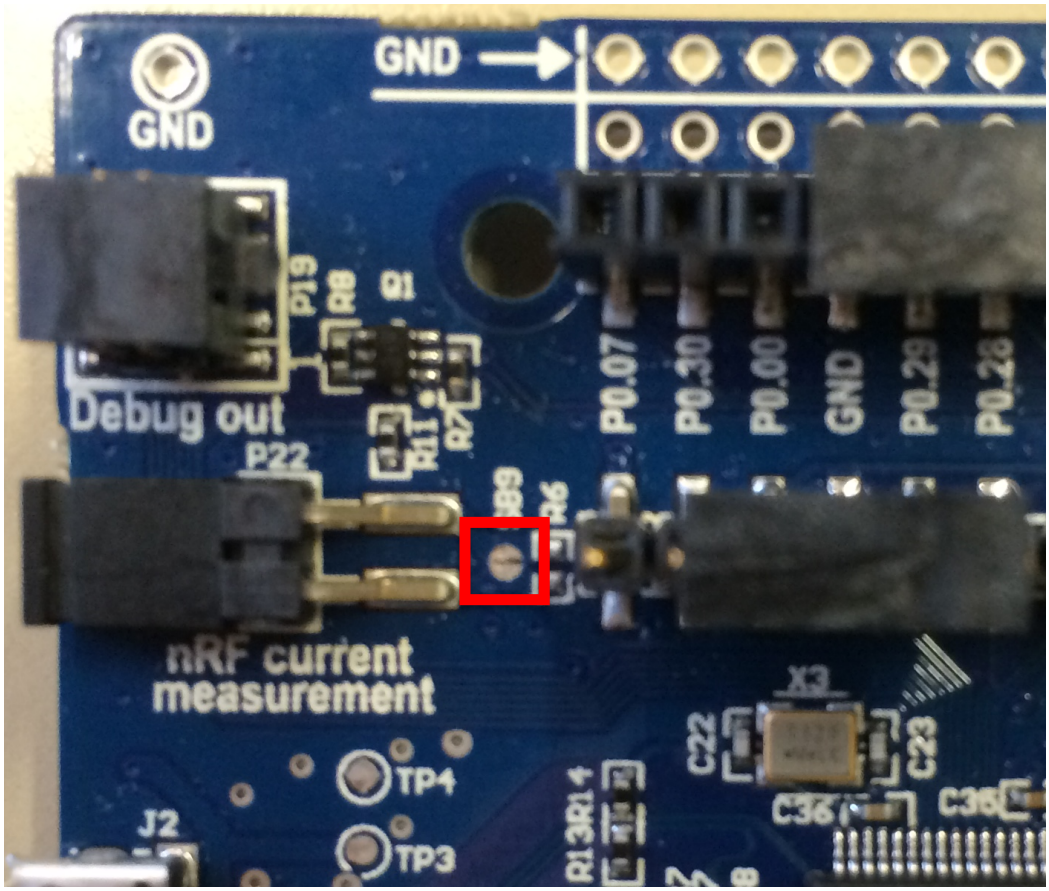

Figure 55, Localisation du SB9 sur le nRF51

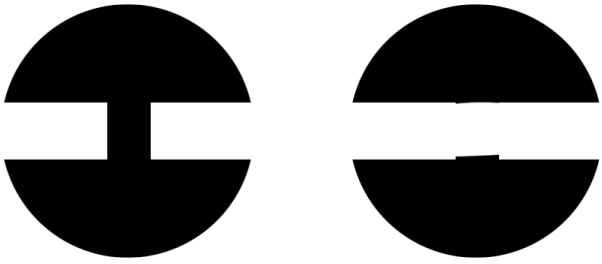

Figure 56, A gauche, SB9 non altéré, à droite SB9 sectionné.

Le SB9 est en fait deux demi-cercles reliés par un pont métallique. Il faut rompre se pont en le sectionnant. Cela peut être réalisé à l'aide d'un couteau ou d'un cutteur.

Lorsque le nRF51 a été modifié, il convient de le connecter au « Digilent Analog Discovery ». Ce petit appareil est en fait (entre-autres) un oscilloscope fonctionnant sur un ordinateur par le biez d'un port USB. Il fonctionne à l'aide du programme « WaveForms » [23].

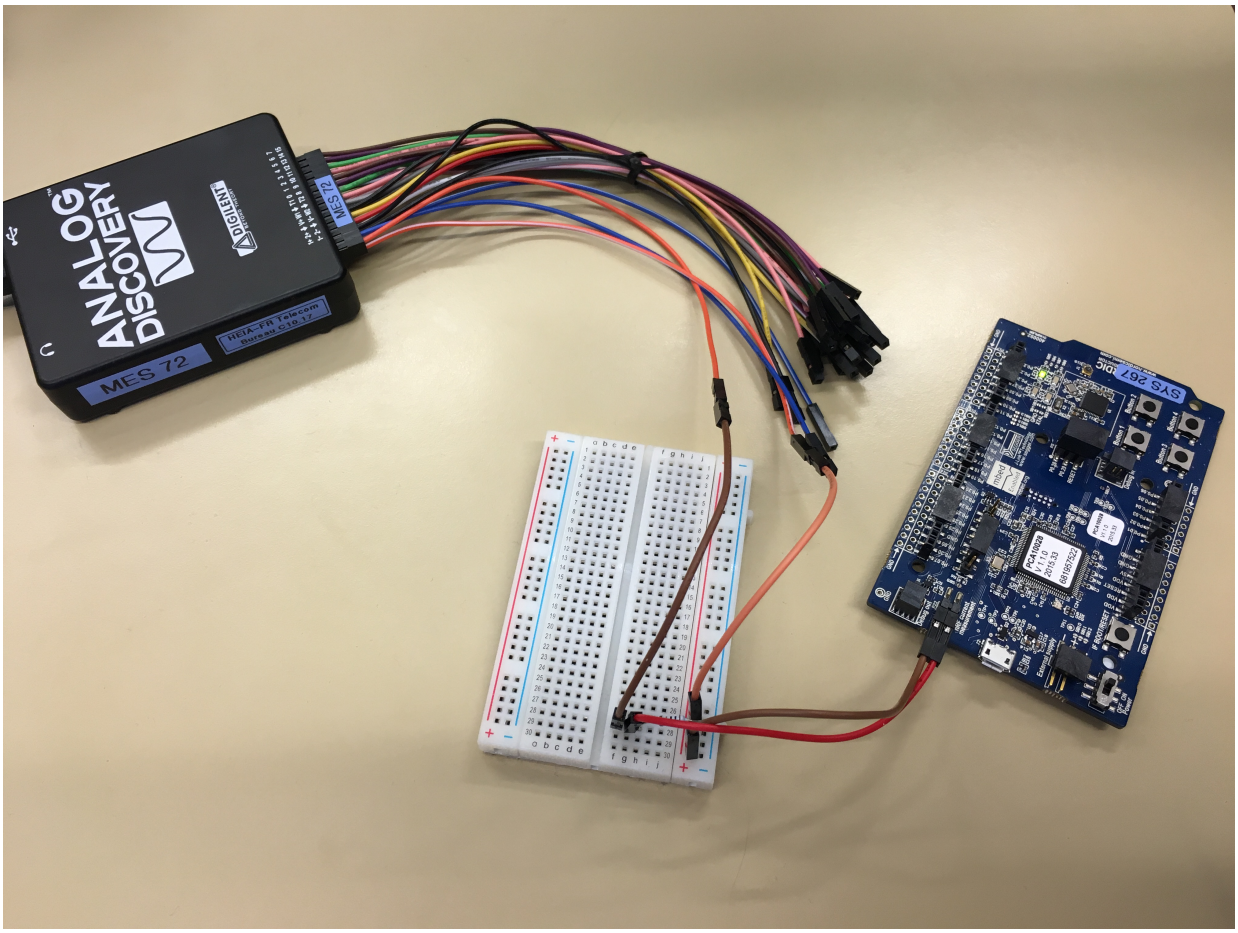

Figure 57, Mise en place du dispositif

Nous avons relié les pins « nRF current measurment » au port « 1+ » et « 1- » de l'analyseur. Il est donc relié en parallèle. Il est aussi nécessaire d'ajouter une résistance de 10 Ohm entre les deux pins du nRF.

7.2.2 Analyse

Nous avons effectué plusieurs analyses sur une période d'environ 5 minutes. Durant cette période, nous avons appuyé une fois sur le bouton d'incrémentation du compteur, ce qui représente 288 cafés par jour. L'appui sur ce bouton déclenche l'envoi d'un message de type NON en CoAP afin de notifier les observeurs du changement de la ressource.

Chaque prise de mesure était enregistrée dans un fichier CSV. Voici ce qu'il était possible de voir sur l'oscilloscope :

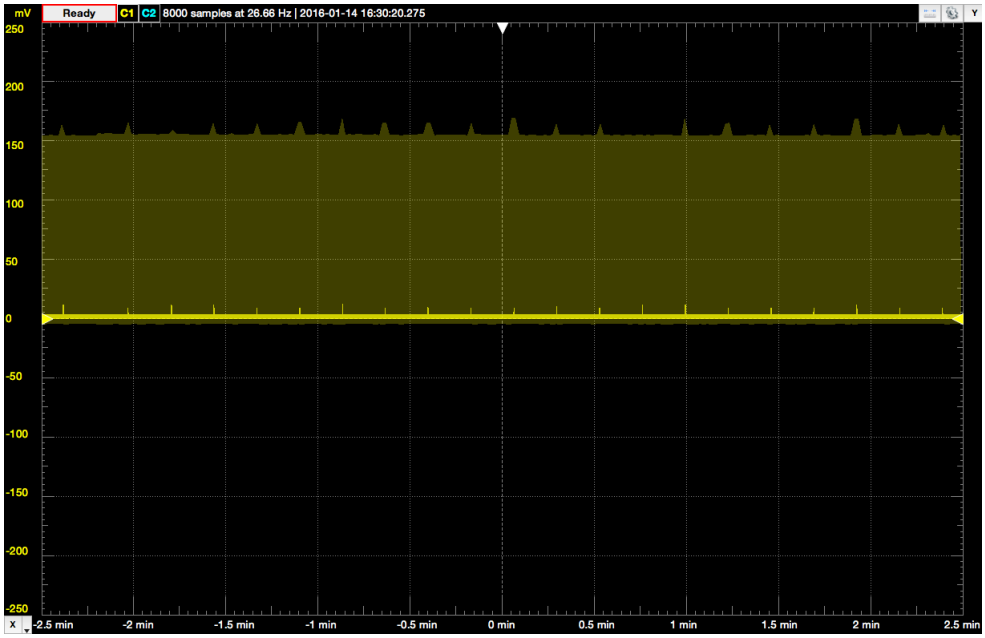

Figure 58, Affichage de l'oscilloscope

7.2.3 Résultats

La moyenne de la valeur des mV mesurée est de 2.5038 mV. D'après la documentation de Nordic, l'utilisation d'une résistance de 10 Ohm était recommandée. Nous avons respecté cette recommandation et nous devons donc diviser la valeur en Volt par la valeur de la résistance pour obtenir la consommation en Ampères (loi d'Ohm).

Le nRF51 consomme donc :

$$\frac{2.5038\ mV}{10\ Ohm} = 0.2503\ mA$$

La pile utilisée est une Panasonic CR2032 [24]. Elle possède une capacité de 220 mAh. Le nRF51 pourrait donc fonctionner non-stop en condition définies ci-dessus pendant :

$$\frac{220}{0.2503} = 878.94\ heures$$

Soit 36.62 jours.

Cela peut paraitre peu, néanmoins cet objet connecté implémente une full stack IP. De plus, cette estimation prend en compte un fonctionnement non-stop. Nous avons également effectué une mesure durant laquelle aucun message CoAP n'était envoyé. Malgré tout, la consommation mesurée était similaire.

Nous pouvons donc en déduire que la cause principale de la consommation n'est pas l'envoi de messages CoAP, mais tout ce qui tourne en dessous. (Gestion de la stack IP, compteurs, etc.) Le code fourni par Nordic est par ailleurs un code d'exemple, dans un cas de production il conviendrait de l'optimiser afin de réduire l'utilisation du CPU et donc augmenter la durée de vie.

7.3 Problèmes connus

7.3.1 Perte de la connexion Bluetooth après un certain temps

Un test a été effectué afin de tester la connectique Bluetooth entre le Raspberry et le nrf51 dans le temps. En effet, au bout de quelques heures, la connexion Bluetooth est perdue. La connexion fait passer des paquets IP au-dessus. (Tel qu'utilisé dans ce projet)

Nos tests ont démontré qu'une connexion Bluetooth entre le Raspberry et le nrf51 dure environ 10 heures au maximum, avec une interaction régulière (un ping par minute).

Nous suspectons que le problème vient du côté Raspberry. En effet, une fois la connexion perdue, il devient impossible d'utiliser les fonctions Bluetooth basiques. Après un redémarrage du Raspberry, il est à nouveau possible de connecter le nrf51 (sans le redémarrer). De plus, le dongle Bluetooth qui nous a été fournit n'est pas des plus récent. Il serait intéressant de tester d'autre dongle afin de savoir si le problème est lié.

7.3.2 Traitement de la connexion au niveau du serveur

Le serveur Node.js ne se soucie pas de savoir si le nrf51 avec lequel il communique est connecté. Par conséquent, si la connexion est perdue le serveur Node.js implémentant la fonction de proxy et de serveur web devient inutilisable (plantage). Il conviendrait de traiter les différentes erreurs efficacement de manière à notifier l'utilisateur et, le cas échéant reconnecter l'objet automatiquement.

7.4 Problèmes rencontrés

7.4.1 Utilisation de l'adresse « link-local » pour l'observe avec la librairie node-coap

Nous avons remarqué que dans le cas d'une observation, lorsque le nRF51 envoie une notification de type CON suite à un changement d'une ressource, le Raspberry n'acquittait pas la requête. Nous avons tout d'abord pensé à un problème de la librairie, car lorsque le Raspberry envoyait une requête, la librairie fonctionnait parfaitement.

Cependant, lorsque nous avons adressé l'interface bt0 de manière globale, avec un préfixe « 5000:: », nous avons découvert par sérendipité que la librairie node-coap était en fait fonctionnelle. Le problème venait en fait de la documentation lacunaire mise à disposition par le développeur. En effet, lorsque cette librairie est utilisée avec des adresses globales, elle fonctionne parfaitement. Les adresses link-local sont donc à éviter dans le cas d'un serveur supportant l'observe.

7.4.2 Support IPv6 de la librairie socket.io

Nous avons rencontré un problème de support IPv6 avec la librairie socket.io. Effectivement, durant ce projet nous avons travaillé en IPv6, mais cette librairie présente visiblement un problème quant à son support.

Le problème est que les adresses IPv6 contiennent 8 blocs séparés par des « : ». Or la librairie interprétait les « : » comme étant le séparateur entre adresse et port en IPv4. Autrement dit, l'adresse http://[2001:620:40b:1030:36d9:89af:2ed7:c9db]/ était considérée comme une adresse dont le port était « c9db ».

Afin de remédier à ce problème, nous avons édité le code comme suit ; tout d'abord nous cherchons des éventuels crochets, utilisés par les browsers pour différencier les adresses IPv6 des adresses IPv4 :

```
//search for brackets
var src = str,
b = str.indexOf('['),
e = str.indexOf(']');
```

Si « b » et « e » sont différents de « -1 », cela signifie que des crochets ont été trouvés dans le string et que donc l'adresse est IPv6. Afin que les « : » de cette adresse ne soient pas confondus avec un séparateur de port, nous allons temporairement les substituer par des « ; ».

```
if (b != -1 && e != -1) {
  str = str.substring(0, b) + str.substring(b, e).replace(/:/g, ';') + str.substring(e,
str.length);
}
```

Ensuite le code « normal » de la librairie est exécuté. Finalement, on effectue la manœuvre inverse afin d'avoir à nouveau une adresse IPv6 valide :

```
uri.host = "["+uri.host.substring(1, uri.host.length - 1).replace(/;/g, ':')+"]";
```

7.5 Perspectives

7.5.1 Plusieurs objets connectés

Durant ce projet, la connexion d'un seul nRF51 a été effectué. Dans la théorie, il serait possible de connecter plusieurs dispositifs sur le même gateway Raspberry, et en utilisant le même dongle Bluetooth 4.1. Il serait intéressant de tester cette possibilité par la pratique.

7.5.2 Connexion avec une réelle machine à café

Comme l'indique le nom de ce projet, il serait intéressant que l'objet connecté en question soit une réelle machine à café, et non un board de développement. Cela nécessiterait quelques notions de soudure et de bricolage afin de modifier une machine à café existante. Il faudrait récupérer le signal du bouton et le connecter à une des entrées du nRF51. Le signal en question devra potentiellement être traité afin de correspondre à ce que peut supporter un terme de voltage un pin d'entrée de ce board. Cette installation permettrait ainsi d'incrémenter un compteur de cafés.

7.5.3 Paramètres codés en dur

Actuellement, la MAC adresse du nRF51 et son adresse IPv6 sont inscrit en dur dans les différents scripts. Il serait intéressant d'automatiser la connexion et de la rendre dynamique. De plus, la procédure de connexion Bluetooth et de configuration de l'interface *bt0* est faite « manuellement » en lançant des scripts bash.

Une des pistes pour la réalisation de ces automatismes est l'utilisation d'une mini base de données qui pourrait contenir la (ou les) MAC adresses des objets connectés, ainsi que leur adresse IPv6. Ces informations seraient inscrites au moment de la connexion avec un dispositif Bluetooth découvert par un scanning. Le serveur Node.js pourrait utiliser ces propriétés pour générer des pages « dynamiquement », en fonction des objets connectés.

7.5.4 Persistance des données

La persistance des données permettrait d'obtenir des statistiques sur la consommation dans le temps, même en cas de coupure d'électricité (pile vide, coupure électrique). Cette persistance pourrait se faire au niveau du Raspberry, à l'aide d'une mini base de données. Ou alors directement sur le nRF51, en stockant un certain nombre de données dans la flash.

7.5.5 Utilisation d'un tag NFC et système d'accounting

L'utilisation d'un tag NFC permettrait d'identifier les utilisateurs. Par exemple en collant un tag NFC sous la tasse personnelle de chaque utilisateur. Il serait alors possible de mettre en place un système d'accounting, capable de déterminer la consommation de café par personne, et donc d'établir une éventuelle facturation. Cette logique serait bien entendu implémentée sur le Raspberry. L'ID du tag NFC serait envoyée en CoAP par le nRF51, et traité côté Raspberry.

Le successeur du nRF51, le nRF52 possède un lecteur NFC directement intégré au board. Il serait donc intéressant d'utiliser la nouvelle version pour ce genre d'application.

8 Conclusion

Arrivés au terme de ce projet de semestre 5, il est temps de faire le point sur les résultats obtenus. Les objectifs principaux ont été atteints :

- Plusieurs designs de l'architecture ont été élaborés, puis comparés afin de définir le plus adapté à notre cas d'utilisation.
- Les couches applicatives CoAP et MQTT ont été comparées. Par ailleurs, HTTP a également été inclus dans cette comparaison. Finalement, CoAP a été choisi comme étant le protocole le plus adapté.
- La communication entre l'objet et l'appareil a été démontré par la réalisation d'un ping IPv6 ayant donné réponse.
- Une implémentation concrète de la solution a été mise en œuvre. Il est possible d'effectuer une requête CoAP au travers d'un proxy agnostique, à destination de la machine a café, représentée par le nRF51. L'appui sur un bouton de la machine incrémente un compteur. Le client final peut consulter cette valeur par le truchement d'une interface web.

Faire passer de l'IPv6 au travers d'une connexion Bluetooth Low Energy est un pas vers la connectivité globale des Objets. Effectivement, si chaque objet disposait de sa propre adresse IP, les objets seraient alors directement atteignables au travers de l'Internet et pourraient par exemple communiquer entre eux sans avoir recours à une passerelle, tel qu'un Smartphone par exemple.

Cependant, malgré l'utilisation du protocole applicatif CoAP, dont le rendement est nettement plus avantageux que HTTP ou MQTT, le cout énergétique d'une telle technologie reste dispendieux. La IoT SDK pour le nRF51 étant une version alpha proposée par Nordic, cela peut expliquer en partie le cout énergétique de la solution. Néanmoins, pour que l'utilisation d'IP sur BLE se démocratise, il faudra encore attendre des améliorations quant à la consommation.

Finalement, nous pouvons dire que l'utilisation de technologies récentes, couplée aux puissantes fonctionnalités asynchrones d'un serveur Node.js permet de dynamiser le monde de l'Internet des Objets.

9 Déclaration d'honneur

Nous, soussignés Berchier Michaël et Chassot Valentin, déclarons sur l'honneur que le travail rendu est le fruit d'un travail personnel. Nous certifions ne pas avoir eu recours au plagiat, ou à toute autre forme de fraude. Toutes les sources d'informations utilisées et les citations d'auteur ont été clairement mentionnées.

Valentin Chassot Michaël Berchier

10 Annexes

10.1 Références

[1] « Page 6LoWPAN Wikipédia »
https://fr.wikipedia.org/wiki/6LoWPAN

[2] « Bluetooth® Core Specification 4.2, Décembre 2014»
https://www.bluetooth.org/en-us/specification/adopted-specifications

[3] « Internet Protocol Support Profile 1.0, Décembre 2014»
https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=296307

[4] « The Constrained Application Protocol (CoAP) », Juin 2014
http://tools.ietf.org/html/rfc7252

[5] « MQTT official website »
http://mqtt.org

[6] « MQTT and CoAP, IoT Protocols, Eclipse Newsletter », Février 2014
http://www.eclipse.org/community/eclipse_newsletter/2014/february/article2.php

[7] « Mosquitto Man Page »
http://mosquitto.org/man/mqtt-7.html

[8] « Internet of Things MQTT Quality of Service Levels », Avril 2015
http://www.ossmentor.com/2015/04/internet-of-things-mqtt-quality-of.html

[9] « MQTT V3.1 Protocol Specification »
http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

[10] « Overview and Evaluation of Bluetooth Low Energy », Août 2012
http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3478807/

[11] « Guidelines for HTTP-CoAP Mapping Implementations »
https://tools.ietf.org/html/draft-ietf-core-http-mapping-07

[12] « Best practices for HTTP-CoAP mapping implementation »
https://www.ietf.org/proceedings/81/slides/lwig-2.pdf

[13] « aiocoap – The Python CoAP library »
https://aiocoap.readthedocs.org

[14] « Exponential backoff »
https://en.wikipedia.org/wiki/Exponential_backoff

[15] « GAP REST API »
https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=285911

[16] « GATT REST API »
https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=285910

[17] « Bluetooth Low Energy, Getting Started »
https://www.safaribooksonline.com/library/view/getting-started-with/9781491900550/ch04.html

[18] « Constrained RESTful Environment (CoRE) Link Format »
https://tools.ietf.org/html/rfc6690

[19] « IPv6 Multicast Address Space Registry »
https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xml

[20] « nRFgo Studio »
https://www.nordicsemi.com/chi/node_176/2.4GHz-RF/nRFgo-Studio

[21] « nrf51_iot_sdk_0.8.0 »
https://developer.nordicsemi.com/nRF5_IoT_SDK/nRF51_IoT_SDK_v0.8.x/nrf51_iot_sdk_0.8.0_f1f6187.zip

[22] « nRF51 Development Kit User Guide v1.0 »
https://www.nordicsemi.com/eng/nordic/download_resource/38677/1/62632445

[23] « WaveForms »
http://store.digilentinc.com/waveforms-2015-download-only/

[24] « Panasonic CR2032 »
http://industrial.panasonic.com/ww/products/batteries/primary-batteries/lithium-batteries/coin-type-lithium-batteries-cr-series/CR2032

[25] « Observing Resources in the Constrained Application Protocol (CoAP) », septembre 2015
https://tools.ietf.org/html/rfc7641

[26] « node-coap »
https://github.com/mcollina/node-coap

10.2 Versions des applications

- Raspbian Wheezy (2015-05-05) Linux kernel 3.18
  - RADVD : 1.8.5
- Node : v5.1.1
  - body-parser : 1.14.2
  - coap : 0.12.1
  - compression : 1.6.0
  - ejs : 2.3.4
  - express : 4.13.3
  - socket.io (edited) : 1.3.7
- jQuery : 2.1.4
- nRFgo Studio : 1.21.0.2
- Keil µVision : 5.17.0.0
- WaveForms : 3.1.5

10.3 Documents

- Cahier des charges
- Planification
- Repository Git : https://gitlab.forge.hefr.ch/michael.berchier/ibrew/tree/master