POLYTECH LILLE

Université de Lille

PFE 13 Réseau de capteurs pour parking intelligent

Rapport Final

Etudiant : Baptiste CARTIER
Encadrants : Alexandre Boé & Xavier Redon & Thomas Vantroy

Table des matières

I. Présentation du contexte

La recherche de places de parking est une tâche fastidieuse, consommatrice de temps et polluante. Afin d'économiser du temps et de réduire la pollution, il convient de réduire le temps et les trajets empruntés pour trouver une place de parking. Il existe déjà plusieurs études proposant différentes solutions afin de palier à ce problème. En effet, il a été montré que l'utilisation de caméras de surveillance couplées avec de la reconnaissance d'image permet d'obtenir des résultats très satisfaisants. Il existe aussi des systèmes de capteurs à l'entrée des parkings et aussi des systèmes employant un capteur par parking, avec diverses méthodes de remonté de l'information. Ces mécanismes de remontée de l'information passe soit par des communication filaire, soit par des communications sans fils.

De plus, depuis certains projets antérieurs, mes tuteurs, M. BOE et M. VANTROYS, sont en possession d'une grande quantité de carte électroniques basées sur le microcontrôleur CC430.

Le but de ce projet est donc d'utiliser ces cartes afin de mettre en place un réseau de capteurs sans fils, si possible basé sur le protocole RPL, protocole de routage pour objets connectés. RPL est un protocole de routage dynamique destiné aux Low power and Lossy Networks (LLN), soit les réseaux à faible puissance et à perte.

L'utilisation de ce type de réseau est intéressante car permet d'obtenir une remontée robuste face aux changements de l'environnement, et il est intéressant de pouvoir analyser les réactions d'un tel réseau.

II. Présentation du cahier des charges

Après discussion avec les encadrants du projet, le cahier des charges suivant a été retenu :

Objectif : Déployer un réseau d'objets sans fils possédant des capacités de routage dynamique

En effet, l'objectif principal du projet est déployer un réseau dynamique en utilisant le routage RPL afin d'étudier son comportement face à un environnement changeant.

Plusieurs contraintes ont été imposées :

Contraintes :

- Utilisation de RPL
- Utilisation des CC430
- Taille du code produit inférieur à 32ko de ROM et 4 ko RAM

Comme l'observation des réactions d'un réseau avec des capacités de routage dynamique est l'un des objectifs de ce projet, l'utilisation de RPL est préférentielle. L'abandon du protocole RPL ne doit être fait qu'ne dernier recours, sil il est montré qu'une autre contrainte ne peut être respectée : la taille.

L'utilisation des CC430 est imposé par le cahier des charges car ces cartes sont déjà existantes, et disposent de capacités de communication radio grâce à une puce cc1101 intégrée. Ces cartes étant destinées à l'origine pour faire des objets connectés, les utiliser pour en faire des capteurs de détection de place de parking est tout à fait approprié.

Le CC430 est un objet contraint en mémoire ne disposant que de 32ko de ROM et 4 ko de RAM, le code produit ne doit pas excéder cette taille.

Certaines considérations ne font pas partie de ce projet :

- Durée de vie minimale des capteurs
- Type de détection voulu
- Budget
- Design du boitier du capteur
- Quel type d'affichage
- Alimentation des capteurs
- Utilisation d'un serveur

En effet, pour le moment le projet ne doit porter que sur la mise en place du réseau, la partie capteur en elle-même peut être mise de côté, mais si l'avancement du projet le permet, ces problématiques peuvent être aussi traitées.

Enfin, l'utilisation d'un Système d'Exploitation (OS) du monde de l'embarqué n'est pas imposé, mais est conseillé car de nombreuses sources existent déjà, et l'utilisation d'un OS permet plus de contrôle.

III. Présentation du travail effectué

1) Etude des OS RIOT et Contiki et faisabilité

Le protocole RPL est nativement présent dans deux des principaux OS open sources pour l'embarqué : Contiki et RIOT OS. Ces OS sont destinés à être déployer au sein d'objets connectés, donc de systèmes contraints en mémoire, que ce soit ROM ou RAM. Il existe d'autres OS du même type, mais correspondent moins au cadre du projet.

Ayant déjà travaillé sur les cartes du projet, en lien avec RIOT OS, je connaissais bien les limites du CC430 en terme de taille de code produit. En effet, la plupart des codes d'exemple en rapport avec RPL du côté de RIOT ne tiennent pas sur le CC430.

Les deux OS vantent une utilisation en ROM et RAM faible (environ 10ko de ROM et 2 ko de RAM), j'ai donc décidé d'orienter mes efforts vers ces deux systèmes.

La première étape du projet a consisté à étudier l'impact en mémoire des deux OS afin de déterminer la faisabilité du projet.

Dans l'étude des deux OS suivant, les compilations ont bien été effectuées avec toutes les options de compilation pour l'optimisation en taille de code.

a. RIOT

Ayant déjà travaillé avec RIOT OS et le CC430, j'ai déjà expérimenté des problèmes liés à la taille du code. Comme les codes d'exemples d'utilisation de RPL ne sont pas compatible avec le CC430 tel quel, j'ai choisi de commencer par utiliser une carte proche mais plus généreuse en taille disponible : le Telosb, ou Tmote Sky. Le Tmote Sky possède 48ko de ROM et 10ko de RAM, et est basé sur une microprocesseur MSP430 de TI, comme le CC430. Comme les deux plateformes sont proche, j'ai estimé que travailler sur le Tmote Sky était un bon début.

La première compilation, sans modification préalable, indique un overflow de 14ko, sur les 48ko disponible.

Cela est en partie dû au fait que le programme d'exemple *gnrc_networking* utilise un shell interactif pour mettre en place le réseau au travers de ligne de commande. Un tel fonctionnalité n'étant pas requise, et surtout très gourmande, j'ai effectué quelques modifications du programme d'exemple. J'ai retiré toutes références au shell dans le programme principal et dans les options de compilation via le makefile, et pour commencer, j'ai décidé de tester un programme vide, ne faisant rien. Même en effectuant ces modifications, la compilation indique un overflow de Rom pour le Tmote Sky de 7ko de ROM. Ces mêmes modifications indiquent un overflow de 1.6ko de RAM.

RIOT propose bien un exemple de programme, *gnrcnetworking_minimal*, plus léger, mais n'intègre pas RPL pour gagner en taille de code, mais comme RPL est nécessaire au projet, ce genre d'exemple ne nous intéresse pas.

b. Contiki

En parallèle à mon travail sur RIOT, j'ai décidé d'explorer l'OS Contiki. Ce dernier n'a pas de support officiel pour le CC430, mais supporte aussi le Tmote Sky, j'ai donc conservé la même base de travail. J'ai par la suite décidé d'orienter mon travail sur cet OS car j'ai estimé que les sources étaient plus lisibles et plus réutilisables que celles de RIOT.

Sans modification, le programme d'exemple pour mettre en place un réseau RPL entraine aussi une surcharge pour le CC430 mais pas pour le Tmote Sky, ce qui était plus prometteur. La principale raison est que Contiki ne passe pas par un shell, mais déclarer directement les

threads dans le code. Cette méthode est plus rigide, on ne peut pas rajouter des threads en fonctionnement, mais est beaucoup plus légère.

text	data	bss
43380	310	6958

Figure 1 : Taille de base du programme d'exemple RPL de Contiki

J'ai ensuite commencé par modifier les fichiers de configuration de Contiki, en jouant sur la taille des buffers, la taille des tables de routage, retirer la gestion TCP, retirer la gestion de la fragmentation des paquets, retirer la gestion des logs.

Simplement en modifiant ces paramètres, je suis arrivé à :

text	data	bss
41174	296	3492

Figure 2 : Taille du programme d'exemple RPL après modification des configurations

On atteint donc une utilisation en RAM voulu, mais toujours un overflow de ROM.

Comme j'utilisais le Tmote Sky, il y avait plusieurs modules intégrés au projet dont je n'avais pas l'utilité, comme la gestion des GPIO ou les capteurs natifs du Tmote Sky par exemple. J'ai désactivé ces modules via le makefile, et je suis arrivé au résultat suivant :

text	data	bss
35762	210	3296

Figure 3 : Taille du programme d'exemple après retrait de sources inutiles

On peut remarquer une réduction assez forte de l'utilisation en Rom, mais toujours trop importante, mais une réduction plus faible en RAM.

En considérant que je devais faire fonctionner Contiki avec un cc430 doté d'une radio interne, le driver de la radio cc2420 du Tmote Sky ne m'était pas utile, je l'ai transformé en driver « vide » :

text	data	bss
31684	184	3118

Figure 4 : Taille du programme d'exemple sans radio

Mon driver pour la radio du cc430 occupe un peu moins de 2ko de Rom

text	data	bss
1616	0	82

Figure 5 : Taille de mon driver radio

ce qui conduit toujours à un overflow en ROM de 1300 octets.

J'ai de plus voulu évaluer la taille de l'OS en lui-même, en compilant un exemple simple : hello-world.

text	data	bss
41600	222	6946

Figure 6 : Taille du programme Hello World de base

On peut remarquer que cette exemple prend énormément de place pour un simple « Hello World ». En effet, Contiki de base met en place toute une couche IPv6 avec RPL, ce qui explique la taille prise par cet exemple simple. Il n'est pas possible de retirer complètement la couche IP, le protocole de routage, la gestion MAC et les drivers radios, mais Contiki permet néanmoins de s'en rapprocher avec des options de configurations. En retirant juste toute la partie réseau, le programme Hello-world fait la taille suivante

text	data	bss
11984	86	2588

Figure 7 : Taille du programme Hello World sans capacité de communication

On remarque que toute la partie réseau de Contiki occupe près de 30ko de ROM à elle seule, soit près de la totalité de l'espace disponible sur le cc430.

Si je rajoute à ces configurations mes retraits, j'arrive à la taille annoncée par Contiki :

text	data	bss
3258	22	422

Figure 8 : Taille du programme hello World avec seulement Contiki

Au vu des résultats obtenus, il a été décidé que si l'on voulait utiliser RPL, on devait se passer d'OS.

2) Extraction

a) Préparation

J'ai décidé d'exploiter les sources de Contiki, car je les trouve plus lisibles, plus facilement récupérable et elles ont donné le meilleur résultat pour le moment.

J'ai d'abord mit en place des scripts pour estimer la taille en RAM et en ROM des fichiers nécessaires. Contiki met en place des outils afin d'analyser la taille en mémoire des différentes fonctions. Par exemple,

make udp-client.flashprof

permet d'observer la tailler en ROM, celle qui nous préoccupé le plus. Cette commande exécute en réalité la commande

msp430-nm -S -td --size-sort udp-client.sky | grep -i " [t] " | cut -d' ' -f2,4

ce qui donne un résultat de la forme

```
{...}
00000384 nbr_table_add_lladdr
00000384 uip_icmp6_error_output
00000396 frame802154_parse
00000432 rpl_process_dio
00000476 transmit_from_queue
00000558 rpl_icmp6_dio_output
00000612 ns_input
00000680 dio_input
00000808 rpl_ext_header_update
00001196 uip_process
00001540 input
00001810 output
```

Figure 9 : Exemple d'output de la commande "make udp-client.flashprof"

avec en première colonne la taille en mémoire de chaque fonction.

En déterminant la localisation de chaque fonction, on peut estimer la taille nécessaire aux source qui nous sont vraiment utile.

J'ai déterminé selon la localisation de chaque fonction si cette dernière était importante, puis j'ai effectué la somme des tailles de chaque fonction. Le résultat est de 28564 octets, ce qui laisserait un peu plus de 3ko pour notre programme principal, ce qui devrait suffire à nos besoins, nos capteurs étant très simple.

b) *Analyse du fonctionnement de Contiki*

J'ai ensuite procédé à comprendre le fonctionnement de l'OS afin de pouvoir extraire les sources dont j'ai besoin. Cette étape m'a pris plus de temps que prévu, Contiki étant plus complexe que ce que j'avais anticipé.

Contiki utilise un système assez complexe de macro C afin de définir et mettre en place les threads. Par exemple, pour déclarer un thread, il faut passer par la macro

```
#define PROCESS(name, strname)          \
  PROCESS_THREAD(name, ev, data);       \
  struct process name = { NULL,         \
                          process_thread_##name }
```

Figure 10 : Macro de déclaration d'un process

définie dans le fichier *process.h* . Elle correspond en fait dans notre cas au code suivant

On remarque que au sein de cette macro, il y a une autre macro, et cela est vrai pour beaucoup des macros définies par Contiki. Cette particularité ma ralenti car cela rend le code plus difficile à extraire et à comprendre.

En cherchant parmi les fichiers sources extraits, j'ai trouvé neuf fichiers incluant le mot clé ou la macro *PROCESS*. J'ai ainsi déterminé quels fichiers étaient ceux qui devraient le plus modifié :

Fichier	Nombre d'occurence de "PROCESS"	Importance de "PROCESS"
netstack.c	3	Aucune
netstack.h	1	aucune
resolv.c	18	forte
resolv.h	1	aucune
simple-udp.c	10	forte
tcpip.c	18	forte
tcpip.h	1	aucune
udp-client.c	6	moyenne
udp-socket.c	11	forte

Figure 11 : Liste des fonctions incluant le mot clé "PROCESS"

J'ai essayé de comprendre le fonctionnement des macros, et en utilisant l'option *–E* du compilateur gcc qui arrête la compilation à la phase de pré compilation, j'ai décomposé la définition d'un thread :

```
PROCESS_THREAD(tcpip_process, ev, data)
{
  PROCESS_BEGIN();

  tcpip_event = process_alloc_event();

  etimer_set(&periodic, CLOCK_SECOND / 2);

  uip_init();

  NETSTACK_ROUTING.init();

  while(1) {
    PROCESS_YIELD();
    eventhandler(ev, data);
  }

  PROCESS_END();
}
```

Figure 12 : Définition du thread tcpip

```
static char process_thread_tcpip_process(struct pt *process_pt, process_event_t ev, process_data_t data)
{
  { char PT_YIELD_FLAG = 1; if (PT_YIELD_FLAG) {;} switch((process_pt)->lc) { case 0:;

  tcpip_event = process_alloc_event();

  etimer_set(&periodic, 128UL / 2);

  uip_init();

  rpl_lite_driver.init();

  while(1) {
    do { PT_YIELD_FLAG = 0; (process_pt)->lc = 837; case 837:; if(PT_YIELD_FLAG == 0) { return 1; } } while(0);
    eventhandler(ev, data);
  }

  }; PT_YIELD_FLAG = 0; (process_pt)->lc = 0;; return 3; };
}
```

Figure 13 : Définition après expansion des macros

J’ai pris beaucoup de temps à comprendre le fonctionnement et l’utilité d’une telle architecture.

Contiki est un OS qui utilise des protothreads, et non des vrais threads. Les protothreads ont été mis au point par Adam Dunkels, Oliver Schmidt, Thiemo Voigt et Muneeb Ali, et publié dans le document : "Protothreads : Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems".

Les protothreads n'utilisent pas de mécanisme de sauvegarde de contexte, mais stockent simplement l'endroit où le thread s'est arrêté. En effet, tous les protothreads tournent dans une et unique pile, contrairement aux threads classiques qui utilisent une pile par thread. L'avantage des protothreads est que du coup la consommation en RAM est très faible, ne stockant que 2 octets en RAM en plus, peu importe la taille du thread. Si on compare à RIOT, qui a besoin d’environ 25 octets par thread, cela entraine un gain considérable.

Les protothreads sont basés sur un fonctionnement similaire à ce qui est appelé "La machine de Duff". Ils ont été développés afin de faciliter le développement d'application sur des systèmes très contraint en mémoire, comme notre cc430. L'objectif est de réduire, voire supprimer, les machines à états très courantes dans la programmation orientée évènements très courante dans les systèmes contraints, mais aussi très complexe.

Grâce à l'utilisation peu évidente du block `switch` il est possible de continuer l'exécution d'un protothread à partir d'un point précis de celui-ci : c'est le mécanisme appelé de "local continuation".

L'inconvénient majeur des protothreads est l'augmentation de l'utilisation en ROM.

Selon le document de Adam Dunkels & co, l'augmentation en ROM est variable en fonction de la complexité, pouvant atteindre plus de 70% d'augmentation dans les pires cas, et environ 15% dans les cas moyens. En revanche, il y a un gain énorme en RAM, passant de 18 octets utilisés en RAM pour un thread classique à 2 octets en protothread, soit une réduction de 89%.

L'utilisation des protothread est donc un compromis entre occupation en RAM, occupation en ROM et complexité de code.

Il est a noté que les protothread empêche l'utilisation du `switch` dans un thread et que l'utilisation de variable locale est très limitée, ces variables étant perdu au changement de "contexte".

c) uIP

En parallèle des recherches effectuées sur le fonctionnement des protothreads, j'ai continué à chercher des informations sur la couche IP de Contiki, qui a apparemment été extraite plusieurs fois. Je n'ai pas réussi à trouver des informations à ce sujet, mais je trouvé la source de la couche IP de Contiki : une implémentation réduite de la couche IP par Adam Dunkels.

Cette implémentation a été par la suite implémentée dans Contiki, ce qui est pratique car cela est la base de mon travail.

Adam Dunkels propose deux version de son implémentation : uIP (microIP) et lwIP (lightweightIP).

Feature	uIP	lwIP
IP and TCP checksums	x	x
IP fragment reassembly	x	x
IP options		
Multiple interfaces		x
UDP		x
Multiple TCP connections	x	x
TCP options	x	x
Variable TCP MSS	x	x
RTT estimation	x	x
TCP flow control	x	x
Sliding TCP window		x
TCP congestion control		x
Out-of-sequence TCP data		x
TCP urgent data	x	x
Data buffered for rexmit		x

Figure 14: Liste des fonctionnalités de uIP et lwIP

uIP et lwIP respectent les RFC concernant la pile TCI/IP, mais restreignent leur taille en n'implémentant pas certains mécanismes très utiles mais non vitaux.

Fonction	Taille (octet)
Gestion de la mémoire	3142
Interfaces Réseaux	458
checksum	1116
IP, ICMP, TCP	14840
Total	21756

Figure 15 : taille en ROM de lwIP

Fonction	Taille (octet)
checksum	712
IP, ICMP, TCP	4452
Total	5164

Figure 16 : Taille en ROM de uIP

uIP est à la fois très compact, et correspond parfaitement à nos attente, j'ai choisi d'étudier cette implémentation pour comprendre comment elle fonctionnait et comment je pourrais la mettre en lien avec les sources de Contiki. Il s'avère que les sources de uIP fonctionnent aussi avec des protothreads, mais ne nécessite aucun OS sous-jacent pour être opérationnelles.

d) RPL

Avant de continuer à explorer la couche IP de Contiki et l'implémentation de uIP, j'ai voulu voir comment fonctionnait la couche RPL de Contiki.

Contiki propose 2 versions de RPL : `rpl-classic` et `rpl-lite`.

`rpl-classic` propose une implémentation plus complète et robuste mais `rpl-lite` propose une implémentation plus légère. Comme nous sommes contraint par la taille, je suis parti sur l'utilisation de RPL-lite.

Les fichiers sources de l'implémentation de RPL sont au nombre de 20 et inclus 18 fichiers headers externes à RPL.

J'ai extrait la liste de toutes les fonctions extérieures aux fichiers sources de RPL appelées par RPL, et j'ai estimé leur importance et si ces fonctions étaient déjà existantes dans uIP.

Il se trouve qu'une très grande partie de ces fonctions ne sont pas présente dans uIP de base, et sont nécessaires. Elles servent pour la plupart à la gestion des nœuds voisins et à la gestion des adresses IPv6.

3) Routage semi statique semi dynamique

Au vu des problèmes de place, j'ai proposé une solution alternative pour mettre en place un réseau de capteur routé avec RPL utilisant les cartes cc430 : des nuages de cc430 chacun routés statiquement vers un Tmote Sky, et ce sont les Tmote Sky qui s'occuperons de la remontée d'information via RPL. Cette solution a été accepté par mes encadrants, car elle permettait d'utiliser les cc430 toujours dans le cadre du projet en contournant le problème de la place en mémoire.

a) Principe

Le principe de cette solution est simple : chaque cc430 transmet l'état de la place de parking à son Tmote Sky « maitre » et le Tmote Sky ensuite transmet les informations relevées à la racine via un routage RPL. Les cc430 émettent dans la bande des 800MHz et les Tmote Sky utilisent ma bande des 2.4GHz, il faut donc utiliser une passerelle pour relier les cc430 des places aux Tmote Sky. Les deux cartes disposent d'une liaison série disponible et accessible, j'ai donc décidé d'utiliser un cc430 comme passerelle. Ce cc430 recevra les informations radio des autres cc430, qui les transmettra à son tour via la liaison série au Tmote Sky. Comme cela allège énormément la charge en ROM pour les cc430, il a été décidé que chaque élément fonctionnerait avec l'OS Contiki.

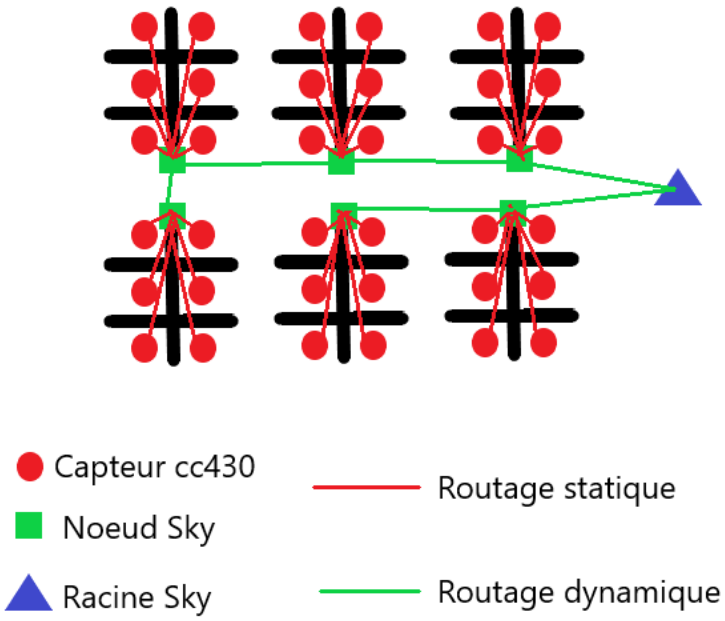

Figure 17 : Routage semi statique semi dynamique

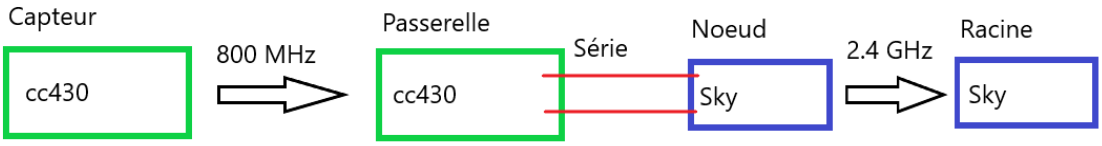

Figure 18 : Communication entre les éléments

b) RPL et Tmote Sky

J'ai vérifié le bon fonctionnement de RPL avec les Tmote Sky en deux temps : en simulation puis en condition réelle.

Contiki propose un outil de simulation très facile à utiliser : Cooja. Grâce à cet outil, j'ai pu vérifier que RPL était bien déployé avec les Tmote Sky, et j'ai pu facilement modifier leur programme d'exemple pour qu'il corresponde à mes besoins.

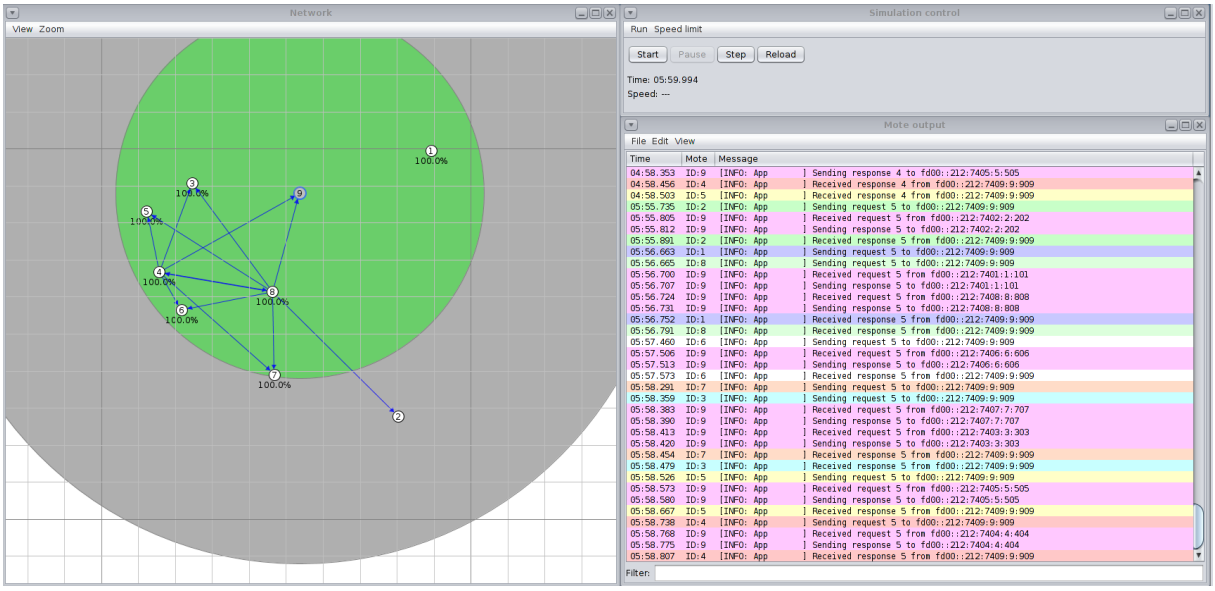

Figure 19 : Exemple de routage RPL avec Cooja

Les tests en conditions réelles ont été moins concluant en revanche. Avec les paramètres de base de Contiki, aucun message, que ce soit les messages de mon application ni même de création de l'arbre DODAG nécessaire pour RPL. Grâce au mode débug de Contiki, j'ai identifié la source : Contiki utilise par défaut le mode CCA (Clear Channel Assessment) qui vérifie que le canal utilisé par la radio est disponible avant d'émettre. Or pour une raison que je n'ai pas encore déterminée, le test CCA échoue à chaque fois, bloquant toute transmission. En retirant ce mode, le réseau se déploie sans soucis. J'ai testé avec seulement 3 nœuds, ce qui limite beaucoup les risques de collision, donc l'utilité du CCA n'est pas flagrante, mais l'est pour un réseau plus grand. Pour le moment le mode CCA est resté désactivé, mais il serait intéressant à l'avenir de corriger ce bug pour permettre un déploiement à plus grande échelle dans de meilleures conditions.

c) Liaison série et Tmote Sky

L'un des module clés du fonctionnement de la solution est la communication série entre le cc430 et le Tmote Sky.

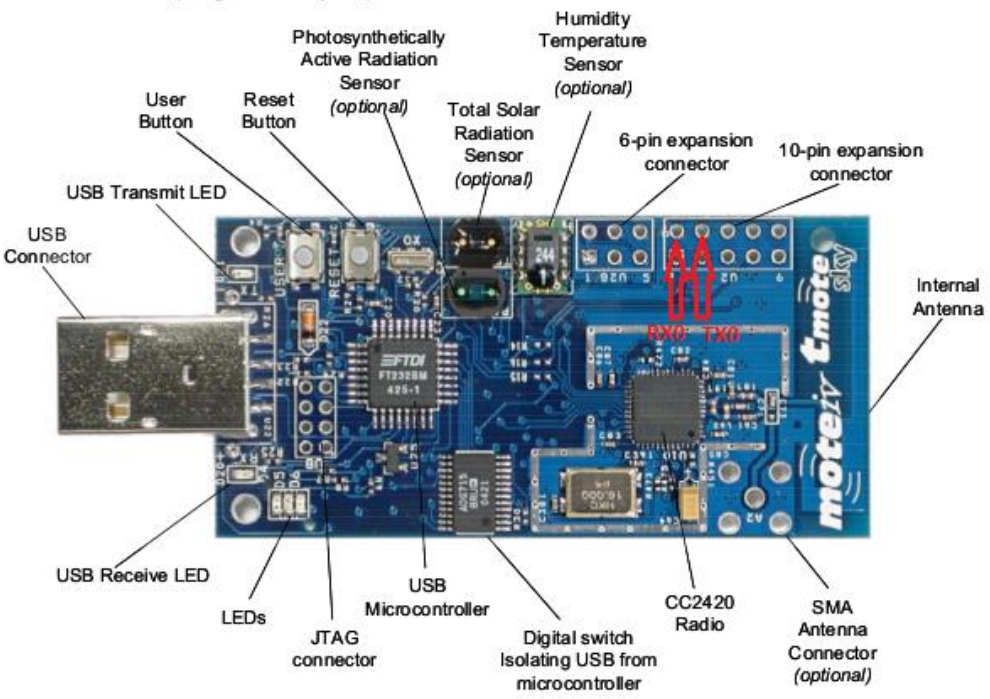

Figure 20 : Port RX et TX du Tmote Sky

La première étape a consisté en la communication série entre 2 Tmote. Le module UART accessible pour le Tmote Sky est le module UART0. Contiki ne supporte pas nativement UART0 pour les msp430f1xxx, modèle du microcontrôleur du Tmote Sky, mais supporte UART1. Selon la datasheet de TI, les deux modules sont identiques, donc les sources pour faire fonctionner UART1 sont très proche de ce qui est nécessaire pour faire fonctionner UART0 : il suffit juste de changer les noms des registres.

La liaison série a très bien fonctionné rapidement sans utiliser Contiki, mais une fois intégrée à l'OS, elle ne répondait plus correctement, et présentait un comportement aléatoire au boot.

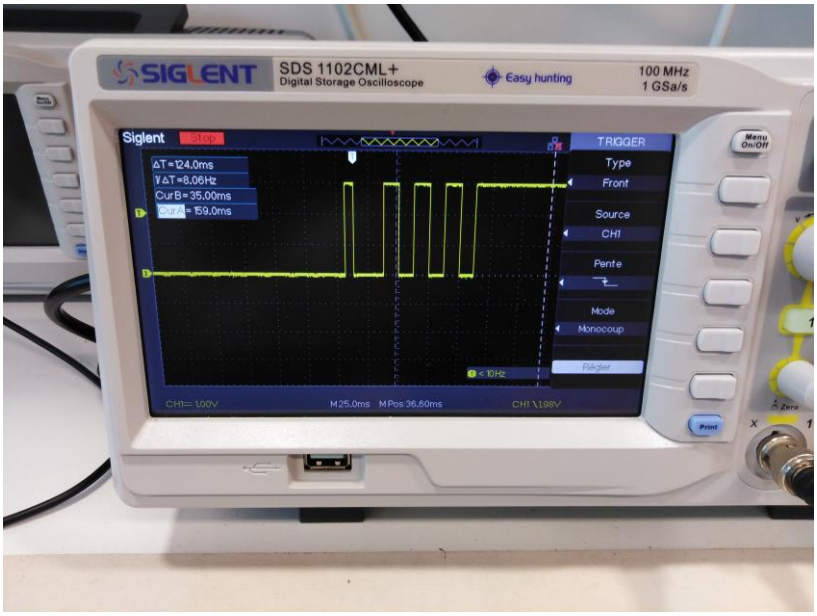

Figure 21 : Caractère 0xAA sans erreur

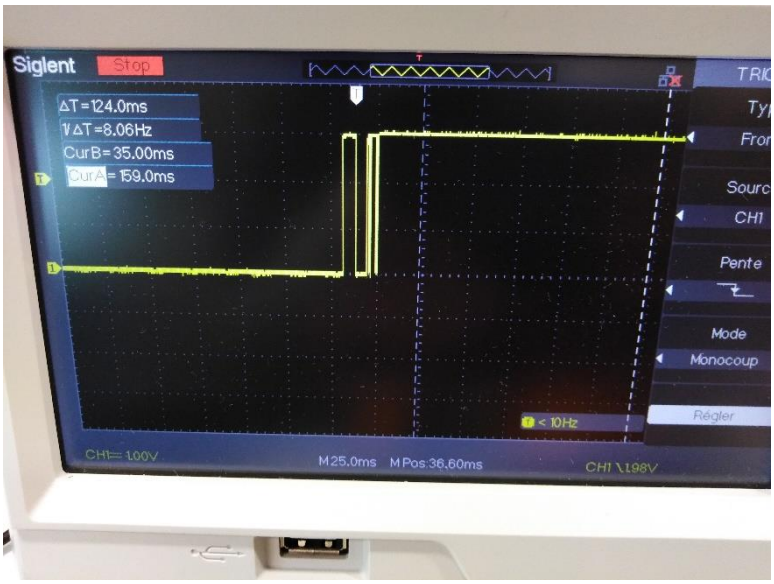

Figure 22 : Caractère 0xAA erroné

Les deux images précédentes illustrent ce comportement : il s'agit du même programme qui transmet le caractère 0xAA juste après l'initialisation de la liaison série.

Pour trouver la source de ce problème, j'ai bloqué l'exécution de l'OS ligne par ligne avec une boucle attendant la fin de transmission du caractère. Ce procédé a été assez long à faire, mais a permis de trouver la source du problème :

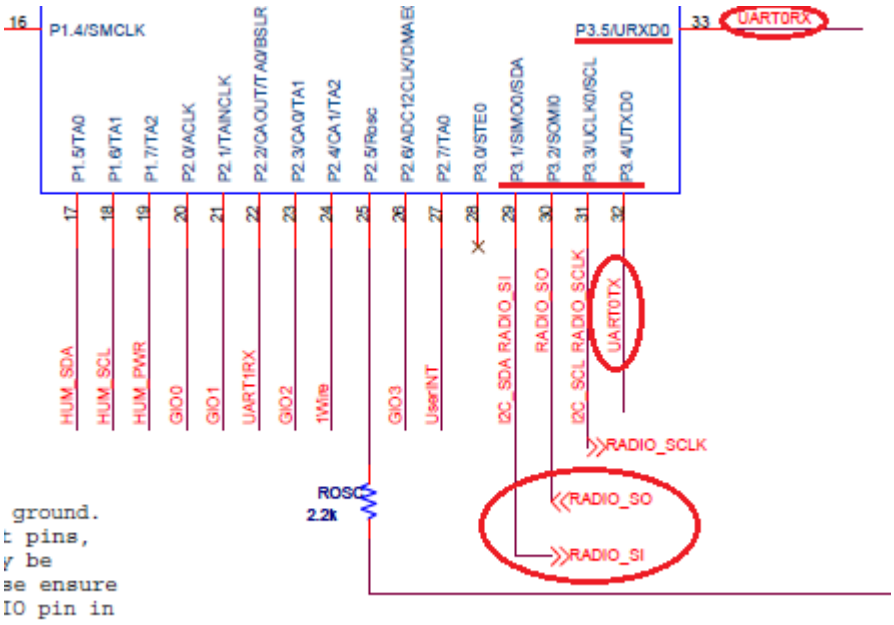

Figure 23 : Localisation des Pin RX0 et TX0 et liaison SPI vers la radio

Un élément que je n'avais pas remarqué est que la radio du Tmote Sky, un cc2420, était commandé par une liaison SPI, géré par le module UART0, et le module UART0 ne peut pas fonctionner à la fois en mode série et en mode SPI

SYNC	Bit 2	Synchronous mode enable
	0	UART mode
	1	SPI Mode

Figure 24 : Extrait de la datasheet du msp430f16xx

Ce problème mettait en péril le projet, ma solution dépendait de la communication série entre les Tmote Sky et les cc430.

La solution apportée à ce problème est la suivante :

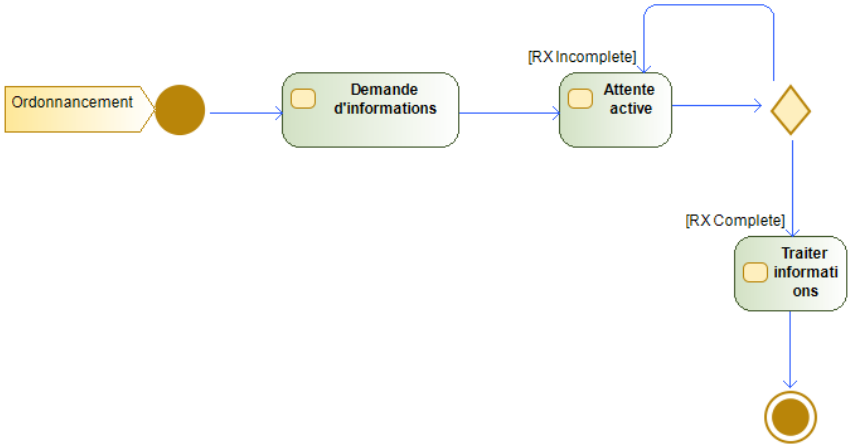

Figure 25 : Algorithme de gestion de la liaison série

Contiki est un OS participatif, c'est-à-dire que c'est à chaque thread de faire en sorte de ne pas occuper tout le temps processeur et de rendre la main à l'OS. Ainsi une boucle d'attente dans un thread « gèle » l'OS.

La base de l'algorithme est la suivante : tous les x temps (dans notre cas, une mise à jour des places toutes les minutes semble approprié), Contiki réveille le thread de gestion de la liaison série. Ce thread envoi un token par la liaison série afin de demander la mise à jour des informations au cc430 passerelle puis gèle l'OS en attendant la réception complète. Une fois les informations reçues, le thread continue son exécution en transmettant les informations via le routage RPL vers la racine.

Une première version de cet algorithme a été testée mais la réception de 6 caractères prenait très longtemps (plus de 15 secondes), alors qu'en 115200 bauds, un caractère transmit en 8N1 prend 86 µsecondes. La raison venait d'une erreur dans le port de l'uart0, le module n'était pas relié à la bonne Clock. Ce problème a été résolu en comparant la sortie de la liaison série d'un Tmote Sky sensé être paramétré en 115200 bauds à celle d'un cc430 configuré de la même manière.

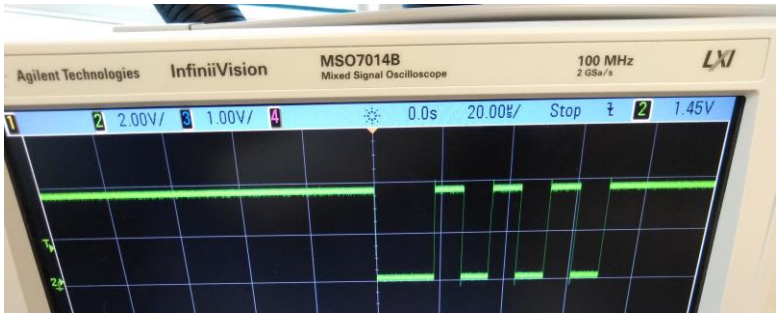

Figure 26 : Caractère 0xAA en 115200 8N1 correcte

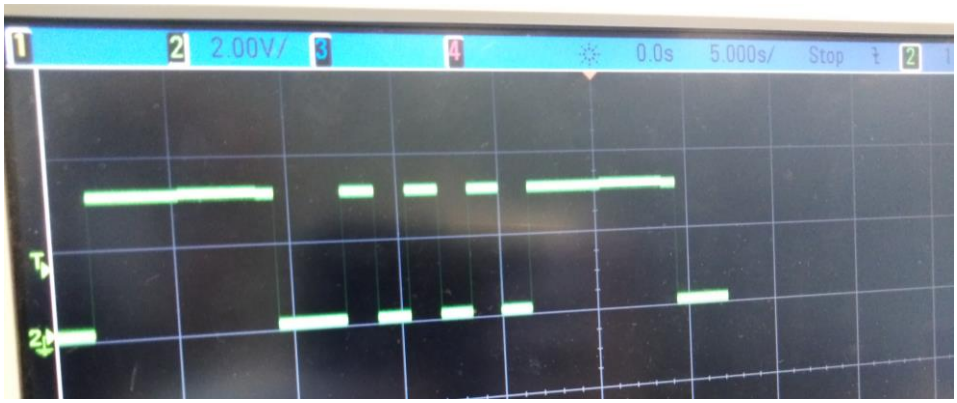

Figure 27: Caractère 0xAA en 115200 8N1 incorrecte

On remarque bien que les caractères sont identiques, mais la base de temps très différentes, 20 µsecondes par division pour la figure 28, contre 5 secondes pour la figure 29.

Cette méthode a été testé et marche, mais présente quelques inconvénients dont le principale est que cela rend le Tmote Sky inaccessible pour le reste du réseau le temps de la communication série. En effet, on coupe la radio afin d'être sûr qu'elle n'agit pas sur le module UART, et on l'empêche de recevoir des paquets, qui sont donc perdus. Cela rend le réseau plus instable, mais cela est en partie compensé en grande partie grâce à la nature même du protocole RPL : il a été conçu pour des réseaux instable. Si un paquet n'arrive pas à la racine par une branche de l'arbre, il peut emprunter une autre branche.

d) Portage de Contiki pour le cc430

Comme il a été décidé que tous les éléments devront fonctionner sous Contiki, il faut aussi faire le portage de Contiki sur le cc430. Pour commencer, les cc430 utiliseront leurs capacité radio de manière ad hoc, sans utiliser les fonctionnalités de Contiki, mais les émissions devront être ordonnancées par Contiki. De plus, une première version du port se contentera du protocole Aloha pour les transmission radio, c'est-à-dire que la transmission sera faite sans tenir compte des perturbations, des collisions et de l'utilisation de la fréquence de la porteuse. Une gestion CSMA (Carrier Sense Medium Access, utilisation de la fréquence porteuse seulement si aucun maquet n'est en transmission) serait préférable, mais peut être mis en place ultérieurement.

Ce portage doit faire l'objet d'un Pull Request auprès de Contiki.

Contiki supporte déjà en grande partie les msp430, dont la famille des msp430f5xxx. J'ai basé mon portage sur une version modifiée du portage des Tmote Sky, une architecture proche basée sur un msp430f1xxx.

Lors de la modification des fichiers nécessaire, je me suis rendu compte que le support du msp430 était approximatif, et suite à une discussion avec l'un des développeurs de Contiki, j'ai appris que ce port était l'un des premiers à avoir été effectué. Parmi ces approximations, on peut retrouver des références à des boards dans la partie réservée au microcontrôleur même, alors que ces références devraient être situé dans le module réservé à la board en elle-même. De plus, parmi ces références, certaines sont en lien avec une board qui n'est plus supporté par Contiki (comme la board Wismote).

De plus, les headers de TI ne sont pas tous cohérents entre eux. En effet, pour certain microcontrôleur, le registre Interrupt Enable se nomme IEx pour les msp430f1xxx mais SFRIEx (SFR pour Special Function Register) pour les msp430f5xxx, alors qu'il s'agit du même registre avec les mêmes fonctionnalités. Il a été décidé en accord avec le développeur de Contiki que lorsque j'effectuerais mon Pull Request, je peux y ajouter une mise à jour du support général du MSP430, en corrigeant notamment une partie des approximations.

Le port basique est fait, et l'ordonnanceur permet bien de gérer les différents threads. J'ai testé ce port avec l'ordonnancement de LED et de transmission radio ad hoc.

L'autre partie importante du côté du cc430 est la liaison série

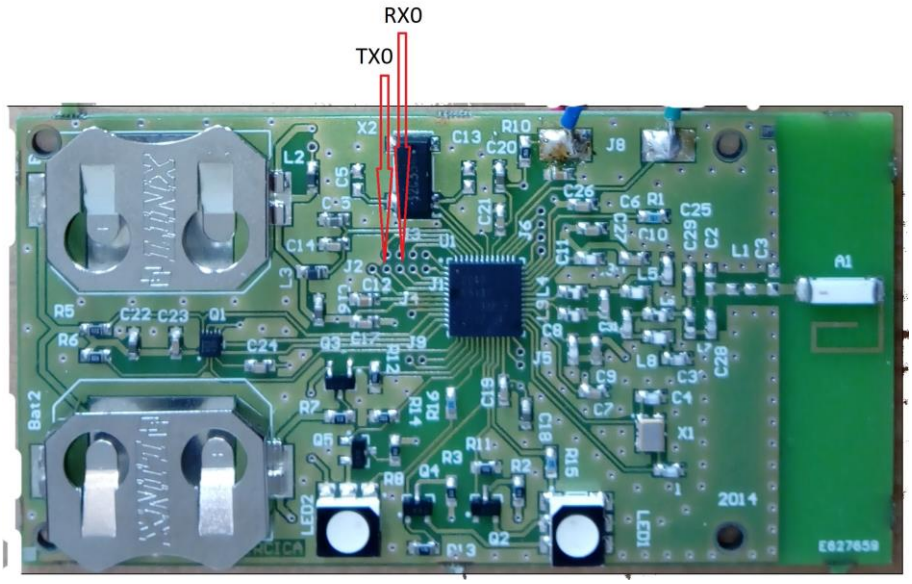

Figure 30 : Localisation des port RX0 et TX0 pour le cc430

La liaison série pour le msp430f5xxx semble à première vue fait de la part de Contiki, mais dans les faits il existe une différence entre les pins du cc430f5137 et des msp430f5xxx. Pour les msp430f5xxx, les Pins RX et TX sont situés sur le Port3 tandis que pour le cc430f5137 elles sont sur le Port1. J'ai donc ajouté cette différence dans l'architecture de Contiki.

Comme sur les cc430 il n'y a aucun conflit pour l'utilisation du module uart0, mettre en place cette liaison série a été très simple. J'ai néanmoins perdu du temps pour confirmer le bon fonctionnement de cette liaison série à cause du problème évoqué a propos de la liaison série des Tmote Sky : ces derniers communiquant en « 115200 » entre eux, les cc430 aussi, mais aucunes communications entre les cc430 et les Tmote Sky était possible.

e) Mise en place du réseau

Une fois toutes les parties terminées et testées de leur côté, j'ai assemblé tous les blocs afin de tester la mise en place du réseau de capteur.

Le protocole de test est composé des éléments suivants : un Tmote Sky racine, deux Tmote Sky nœud connecté par liaison série a chacun un cc430, et 3 cc430 capteurs répartis en un groupe de deux capteurs dirigé vers un nœud et l'autre capteur dirigé vers le second nœud.

Comme les cc430 capteur ne possèdent pas réellement de capteur, ils se contentent de transmettre des informations (une valeur sur 1 octet) toutes les minutes, valeur incrémentée par la suite.

Les cc430 passerelles récupèrent les paquets des capteurs qui leurs sont adressés, et attendent la commande du nœud pour transmettre à leur tour les informations.

Les nœuds créés de manière autonome le DODAG pour RPL, et une fois que la racine est joignable, demande par liaison série toutes les minutes à la passerelle les informations collectées, puis les transmettes à la racine.

La racine transmet les données reçues à son tour par liaison USB au « serveur », dans mon cas mon ordinateur.

Suite à ce test, j'ai pu remarquer que le réseau se déployait bien et que les informations des capteurs étaient bien reçues sur mon ordinateur.

IV. Travail restant et améliorations

En l'état actuel du projet, un prototype fonctionnel est disponible. Il reste cependant beaucoup d'améliorations à apporter.

Tout d'abord, la gestion du CSMA sur les cc430 n'est pas faite, et bien que cela n'a pas posé de problème lors de mes tests grâce au nombre restreint de capteur, pour un déploiement à plus grande échelle cela doit être réalisé. De plus, il reste à corriger le bug du Tmote Sky concernant le CCA.

Un véritable driver radio correspondant au attente de Contiki serait intéressant à faire, car pour le moment la radio du cc430 est utilisé de manière ad hoc est n'est pas véritablement intégrée à l'OS.

Différents débits pour les modules UART devraient être supporté, car actuellement, seul le mode 115200 8N1 est véritablement supporté.

Le port du cc430 à l'heure actuelle n'est pas tout à fait terminé, il me reste une partie de la mise à jour du support à moderniser, mais je compte terminer cette partie même après la fin du projet.

Il faudrait aussi tester ma solution dans des conditions réelles, avec de vrai perturbations dans le réseau afin de voir comment RPL réagit.

Il est aussi a noté que je ne me suis pas intéressé au problématique de portée des cc430 et des Tmote Sky, ni à la consommation d'énergie, problématique pourtant important dans l'IoT.

V. Conclusion

Le cahier des charge d'origine imposait l'utilisation des cartes basées sur le cc430 et l'utilisation du protocole RPL. J'ai montré que ce cahier des charges n'était pas faisable avec seulement les cc430, RPL et IPv6 étant trop gourmand en mémoire malgré une implémentation déjà minimale. J'ai pu apporté une solution alternative permettant de répondre partiellement au cahier des charges.

Le projet possède maintenant un prototype utilisable, basé sur une combinaison de cc430 et de Tmote Sky et ce prototype permet la remonter d'information de capteur via le protocole RPL.

Je suis content du résultat du projet, j'ai pu fournir un prototype et une solution innovante aux problèmes qui m'étaient donnés. J'ai acquis beaucoup de compétences, principalement en langage C avec la compréhension de la machine de Duff et des protothread. J'ai pu aussi laissé une trace dans le monde de l'IoT avec un l'intégration de mon travail dans l'OS Contiki