PFE 13 Réseau de capteurs pour parking intelligent

Rapport intermédiare

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 ROM.

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.

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 4 : 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 5 : 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 6 : 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 7 : 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 8 : 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 orienté 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 9: 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 10 : taille en ROM de lwIP

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

Figure 11 : 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.

J'ai extrait la liste de toutes les fonctions extérieures à ces fichiers appelés par les fonctions de la bibliothèque RPL-lite et j'ai commencé à les explorer pour voir si elles étaient reliées aux protothreads. Il s'avère que plusieurs fonctions appellent des fonctions définies dans `sys/ctimer.h/c` qui lancent des protothreads.

Au vue de cette information, je pense que retirer les protothreads seraient trop compliqué et trop long, et les protothreads permettent un développement simplifié, sachant que chaque block qui m'intéresse les utilisent.

IV. Présentation du travail restant et planning

1) Travail restant

Des recherches sur les différentes implémentations sont à prévoir. Je n'ai pas encore terminé d'explorer les sources RPL de Contiki et leur intrication avec l'OS.

Du coté des protothreads, je dois encore bien comprendre leur fonctionnement pour pourvoir bien effectuer la transition vers un système sans OS.

Je dois encore établir l'interface entre uIP et RPL.

Je dois aussi extraire la couche 6LowPAN de Contiki, pour compresser les entêtes des trames IP, nécessaire à RPL.

Une fois ces étapes réussies, je dois assembler tous les blocks ensemble afin de pouvoir mettre en place un vrai réseau RPL, puis je devrais tester ce réseau.

2) Planning

Je prévois encore un mois de travail pour que tout soit bien compris et assimiler, puis encore un moins pour implémenter, résoudre les problèmes qui surviendrons et tester les fonctionnalités du réseau.

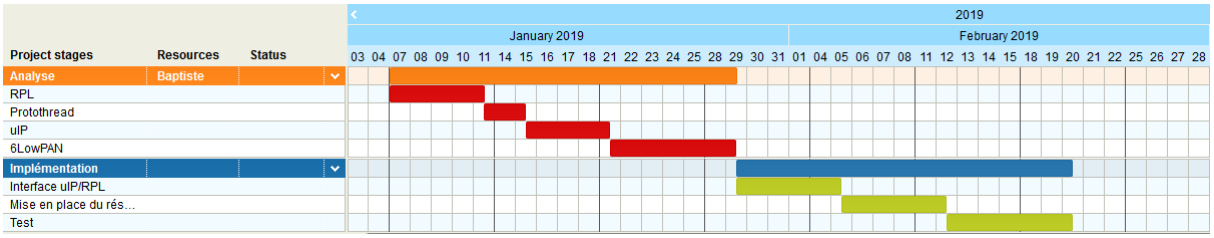

Figure 12 : Planning prévisionnel