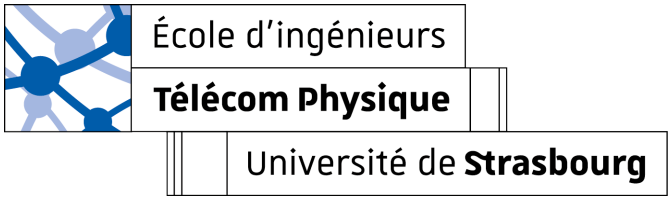


Internet des objets

Études des performances d'un réseau d'objets connectés à Internet IPv6 avec RPL

Loubna Hennach & Elisa Scheer

Janvier 2019

Table des matières

1 Contiki OS 2
2 Réseau RPL-CoAP basé sur nullrdc 2
3 Réseau RPL-CoAP basé sur contikimac 5
4 Réseau RPL-CoAP basé sur tsch 6
5 Études des résultats 7

Le chemin du dossier tp2 déposé sur le serveur `iotstras7` est le suivant : ∼`/tp2`. Ce dossier contient l'ensemble des codes qui nous ont servis à générer du trafic et à analyser les résultats des expériences. Un `README.md` à été fourni pour l'ensemble des scripts rendus. Le dossier contient également les 6 applications utilisées pour flasher les noeuds, nous les avons déposées dans ∼`/tp2/applications_noeud_m3` ainsi que les codes modifiés de `contikios` dans le dossier ∼`/tp2/source_contiki`. Enfin, le dossier contient trois dossiers nommés `nullrdc,contikimac,tsch`, ces dossiers quant à eux contiennent l'ensemble des expériences que nous avons réalisées : dans chaque dossier, nous avons des sous-dossiers correspondant à l'identifiant des expériences qui contiennent des fichiers `output` qui sont traités par des scripts d'analyse qui seront décrits dans ce rapport.

Les résultats présentés dans ce rapport ont tous été réalisés sur les noeuds m3-5, 6, 7, 8 et 9.

1 Contiki OS

Pour pouvoir installer l'environnement de compilation de Contiki OS [3] :

```
sudo ssh -X iotstras7@strasbourg.iot-lab.info
git clone https://github.com/iot-lab/iot-lab.git
cd iot-lab
make
make setup-openlab && make setup-contiki
```

Puis suivant l'application que nous allons déployer sur les noeuds, nous devrons utiliser les paramètres qui nous ont été assignés. C'est pourquoi, nous ajouterons dans chaque fichier `project-conf.h` d'une application les lignes suivantes :

```
#undef RF_CHANNEL
#define RF_CHANNEL                      17
#undef IEEE802154_CONF_PANID
#define IEEE802154_CONF_PANID           0x0007
```

qui permettent de paramétrer respectivement le canal radio utilisé et le PANID.

2 Réseau RPL-CoAP basé sur nullrdc

Pour déployer un réseau RPL comprenant un routeur de bordure et quatre noeuds CoAP [2], on compile les applications `rpl-border-router`

```
cd ~/iot-lab/parts/contiki/examples/ipv6/rpl-border-router
make TARGET=iotlab-m3
cp border-router.iotlab-m3 ~/
```

Puis on compile l'application `04-er-rest-example` avec les commandes :

```
cd ~/iot-lab/parts/contiki/examples/iotlab/04-er-rest-example
make TARGET=iotlab-m3
cp er-example-server.iotlab-m3 ~/
```

Ensuite un préfixe ipv6 nous a été assigné pour l'adressage des noeuds **2001 :660 :4701 :f0a6 : :/64**. On choisit le noeud **m3-5** comme border router ensuite on installe `tunslip6` qui va nous permettre d'obtenir la connectivité IPv6 au sein de notre réseau RPL [1] puis on lance l'instance `tunslip6` avec le préfixe qui nous a été attribué :

```
sudo tunslip6.py -v2 -L -a m3-5 -p 20000 2001:660:4701:f0a6::1/64
```

Enfin, on déploie les applications sur les noeuds :
Déploiement du *Border Router* avec la commande :

```
iotlab-node --update ~/iot-lab/parts/contiki/examples/ipv6/rpl-border-router/
border-router.iotlab-m3 -l strasbourg,m3,5
```

Déploiement du serveur COAP sur les quatres noeuds non-BR avec la commande :

```
iotlab-node --update ~/iot-lab/parts/contiki/examples/iotlab/04-er-rest-example/
er-example-server.iotlab-m3 -e strasbourg,m3,5
```

2.1 Question 1

Par défaut, le RDC (Radio Duty Cycling) de la couche MAC est configuré à `contikimac_driver`. Pour le changer, on édite le fichier `/iot-lab/parts/contiki/platform/openlab/conti ki-openlab-conf.h` ou bien on redéfini la configuration MAC dans le fichier `project-conf.h` de notre application en ajoutant :

```
#undef NETSTACK_CONF_RDC
#define NETSTACK_CONF_RDC     nullrdc_driver
```

2.2 Question 2

Paramètres RPL utilisés :

- mode opératoire : nous avons changé le mode par défaut `STORING` en `NON-STORING` afin que ce soit uniquement le routeur de bordure qui stocke la table de routage. Ainsi, comme les noeuds CoAP évoluent dans un environnement contraint, nous avons jugé que ce mode convenait. Nous modifions ce paramètre dans le fichier `project-conf.h` :

  ```
  #ifndef WITH_NON_STORING
  #define WITH_NON_STORING 1 /* Set this to run with non-storing mode */
  #endi
  ```

- fonction objectif : on garde la fonction par défaut MRHOF, on la trouve dans le fichier `/iot-lab/parts/ contiki/core/net/rpl/rpl-conf.h`

  ```
  #ifdef RPL_CONF_OF_OCP
  #define RPL_OF_OCP RPL_CONF_OF_OCP
  #else /* RPL_CONF_OF_OCP */
  #define RPL_OF_OCP RPL_OCP_MRHOF
  #endif /* RPL_CONF_OF_OCP */
  ```

- métrique utilisée : `ETX` que nous modifions dans `/iot-lab/parts/ contiki/core/net/rpl/rpl-conf.h`

  ```
  /* The MC advertised in DIOs and propagating from the root */
  #ifdef RPL_CONF_DAG_MC
  #define RPL_DAG_MC RPL_CONF_DAG_MC
  #else
  #define RPL_DAG_MC RPL_DAG_MC_ETX
  #endif /* RPL_CONF_DAG_MC */
  ```

2.3 Question 3

Vérification de la connectivité ipv6 et table de routage RPL du noeud Border Router :

```
iotstras7@strasbourg:~$ lynx -dump http://[2001:660:4701:f0a6::9889]
   Neighbors
fe80::a887
fe80::9287
fe80::b184
fe80::b885

   Routes

   Links
2001:660:4701:f0a6::b184 (parent: 2001:660:4701:f0a6::9889) 1795s
2001:660:4701:f0a6::b885 (parent: 2001:660:4701:f0a6::9889) 1796s
2001:660:4701:f0a6::9287 (parent: 2001:660:4701:f0a6::9889) 1797s
2001:660:4701:f0a6::a887 (parent: 2001:660:4701:f0a6::9889) 1798s
```

(a) Table de routage RPL du BR

```
iotstras7@strasbourg:~$ coap get coap://[2001:660:4701:f0a6::a887]:5683/sensors/pressu
(2.05)  1019
```

(b) Requête CoAP

FIGURE 1 – Consultation des routes sur le BR et vérification de la connectivité

2.4 Question 4

Puis pour pouvoir visualiser les évènements RPL, on active le DEBUG dans le fichier `/iot-lab/parts/contiki/core/net/rpl/rpl.c` en modifiant :

```
#define DEBUG DEBUG_FULL
```

2.5 Question 5

En lançant `serial_aggregator`, nous obtenons l'historique des événements RPL de chaque noeud du réseau, ainsi nous pouvons calculer le temps de convergence de RPL pour que tous les noeuds soient joignables. En effet le temps de convergence du `DAG` RPL est défini comme la durée nécessaire à tous les nœuds du réseau pour rejoindre le `DAG`. Ce temps correspond donc au temps auquel le dernier noeud du réseau rejoint le `DODAG` construit par le *Border Router*. Avec un script python, on parse les logs du *serial_ aggregator* puis on relève le temps auquel le dernier noeud rejoint le `DODAG` et on lui soustrait le temps du début de construction du `DODAG`, ainsi on obtient le temps de convergence du réseau. Pour avoir une première idée sur le temps de convergence des noeuds nous lançons une première expérience d'une durée de `5 minutes` et nous lançons le script `convergence.py` comme suit :

```
python convergence.py ~/nullrdc/experiment_1/1.serial_aggregator.out
```

Nous obtenons un temps de convergence de `0.00996s`. Le temps de convergence moyen obtenu sur 10 expériences réalisées dans les mêmes conditions est présenté dans la section 5 à l'aide du script `convergence_mean.py`

2.6 Question 6

Pour monitorer la consommation énergétique des noeuds de réseau, on définit un profil **conso** via l'interface web de FIT IOT-LAB, en suivant les étapes du tutoriel [4]. On choisit de monitorer la consommation en termes de courant, de puissance et de tension électrique, puis on se connecte sur la plateforme en activant le *X11Forwarding* : `sudo ssh -X iotstras7@strasbourg.iot-lab.info` des fichiers m3-<id>.oml se remplissent au fur et à mesure de l'expérience, ils contiennent toutes les grandeurs mesurées toutes les 8244 μ secondes, ensuite on peut tracer la courbe de la puissance(-p) ou bien du courant(-c) avec la commande :
`plot_oml_consum -p -i  /.iot-lab/<experiment id>/consumption/m3-<id>.oml`.
Nous avons ensuite automatisé le traçage des courbes avec un script python **plot.py** qui nous servira dans la partie 5 afin de tracer une courbe moyennant la consommation énergétique sur 10 expériences réalisées dans les mêmes conditions. Notons que le profil conso est activé dans le script d'automatisation `experiment.sh` avec la ligne :
`iotlab-experiment submit -d 10 -l strasbourg,m3,5-9,,conso`

2.7 Question 7

Le script `coap_client.py` permet de lancer un client CoAP qui va effectuer des requêtes vers les noeuds CoAP avec un intervalle de 5 secondes entre chaque requête pendant une durée de 5 minutes. Nous avons utilisé la librairie `CoAPthon` en ayant modifié le fichier `/CoAPthon/coapthon/client/coap.py` pour récupérer le nombre de paquets envoyés et reçus. En lançant une première expérience de 10min et en lançant le client :

```
python coap_client.py -o GET > coap.out
```

On peut consulter le fichier `coap.out` qui décrit les évènements du protocole. A la fin de ce fichier, on trouve le nombre total de paquets émis, reçus puis le pourcentage de paquets perdus (figure 2).

```
2019-01-04 20:32:59,919 - MainThread - __main__ - DEBUG - Statistics
2019-01-04 20:32:59,920 - MainThread - __main__ - DEBUG - Packets Sent: 60
2019-01-04 20:32:59,920 - MainThread - __main__ - DEBUG - Packets Received : 60
2019-01-04 20:32:59,920 - MainThread - __main__ - DEBUG - Percentage Packet Loss : 0.0
```

FIGURE 2 – Statistiques à la fin du fichier log coap.out

Puis en analysant le fichier log avec le script `statistics.py` en lançant la commande `python statistics.py -path=<chemin vers le fichier coap.out>`, on peut récupérer le temps de latence moyen sur l'expérience. La

méthode consiste à regarder le numéro de séquence `CON` envoyé par le client puis le numéro de séquence `ACK` renvoyé par un noeud et faire la différence entre les deux durées d'émission et réception. Ainsi, on obtient un temps de latence moyen de `14.98ms`.

2.8 Question 8

Nous avons rencontré de nombreux problèmes en testant cette partie. Le problème principal est que lorsque notre client ne souhaite plus observer la ressource, il envoie un message `GET` en plaçant `Observe` à 1. Mais ce dernier attend un acquittement que le noeud CoAP n'envoie pas. Pour essayer de comprendre, nous avons modifié dans les fichiers
`/iot-lab/parts/contiki/apps/er-coap/er-coap.c`,
`/iot-lab/parts/contiki/apps/er-coap/er-coap-observe.c`
`/iot-lab/parts/contiki/apps/er-coap/er-coap-engine.c`
la ligne

```
#define DEBUG 1
```

afin de pouvoir retracer les évènements de l'application `CoAP`. Ce que nous avons remarqué est que le noeud reçoit bien le message de dés-inscription, met à jour sa table d'`observers` mais le client ne reçoit pas d'`ACK` et donc retransmet sa requête plusieurs fois. Nous n'avons malheureusement pas trouvé de solutions que ce soit en se penchant sur le code de la librairie `CoAPthon` ou le code source de `contikiOS`.

3 Réseau RPL-CoAP basé sur contikimac

3.1 Question 1

On change le paramètre `NETSTACK_CONF_RDC` en `contikimac_driver` dans `project-conf.h` ou bien `/iot-lab/parts/contiki/platform/openlab/contiki-conf.h`

3.2 Question 2

Les paramètres RPL utilisés sont les suivants :

- Mode opératoire : `NON STORING`
- Fonction objectif : nous avons laissé la fonction objectif par défaut `mrhof`
- Métrique : `ETX`

3.3 Question 3

Lorsque le réseau s'est bien mit en place, nous pouvons observer la table de routage du BR (voir figure 5a) et nous pouvons vérifier la connectivité du serveur vers chacun des noeuds (voir figure 5b).

```
iotstras7@strasbourg:~$ lynx -dump http://[2001:660:4701:f0a6::9889]
   Neighbors
fe80::9287
fe80::a887
fe80::b184
fe80::b885

   Routes

   Links
2001:660:4701:f0a6::b184 (parent: 2001:660:4701:f0a6::9889) 1796s
2001:660:4701:f0a6::b885 (parent: 2001:660:4701:f0a6::9889) 1797s
2001:660:4701:f0a6::9287 (parent: 2001:660:4701:f0a6::9889) 1798s
2001:660:4701:f0a6::a887 (parent: 2001:660:4701:f0a6::9889) 1798s
```

(a) Table de routage RPL du BR

```
iotstras7@strasbourg:~$ coap get -o coap://[2001:660:4701:f0a6::b885]/test/push
(2.05)  VERY LONG EVENT 12
(2.05)  VERY LONG EVENT 13
```

(b) Requête CoAP

FIGURE 3 – Consultation des routes sur le BR et vérification de la connectivité

3.4 Question 4

Le debug a été activé dans la partie ***Réseau RPL-CoAP basé sur nullrdc***

3.5 Question 5

Nous utilisons le même script présenté dans la partie ***Réseau RPL-CoAP basé sur nullrdc*** pour calculer le temps de convergence du réseau. Pour avoir une première idée sur le temps de convergence des noeuds nous lançons une première expérience d'une durée de **5 minutes** et nous lançons le script **convergence.py** comme suit :

```
python convergence.py ~/contikimac/experiment_1/1.serial_aggregator.out
```

Nous obtenons un temps de convergence de **0.0853s**. Le temps moyen sera calculé dans la section 5.

3.6 Question 6

Nous utilisons le même profil **conso** pour monitorer la consommation énergétique des noeuds. Les courbes obtenues sur 10 expériences seront présentées dans la section 5.

3.7 Question 7

On obtient les informations sur le nombre paquets sur la figure 6 en observant la fin du fichier `coap.out`, produit par `python coap_client.py > coap.out`

```
2019-01-04 22:26:57,467 - MainThread - __main__ - DEBUG - Statistics
2019-01-04 22:26:57,467 - MainThread - __main__ - DEBUG - Packets Sent: 57
2019-01-04 22:26:57,467 - MainThread - __main__ - DEBUG - Packets Received : 56
2019-01-04 22:26:57,468 - MainThread - __main__ - DEBUG - Percentage Packet Loss : 1.75438596491
```

FIGURE 4 – Statistiques à la fin du fichier log coap.out

Puis en calculant le temps de latence on trouve une latence moyenne de `326.04ms`.

4 Réseau RPL-CoAP basé sur tsch

4.1 Question 1

Pour pouvoir activer `tsch`, on utilise le fichier présent dans `/iot-lab/parts/contiki/examples/iotlab/00-configuration/tsch-project-conf.h`

Les paramètres TCSH utilisés se trouvent dans le fichier `/iot-lab/parts/contiki/core/net/mac/tsch/tsch-conf.h` :

- Nombre de canaux : on utilise 4 canaux
- Séquençage des canaux : les noeuds scannent les canaux 15, 25, 26 et 20

```
#define TSCH_HOPPING_SEQUENCE_4_4 (uint8_t[]){ 15, 25, 26, 20 }
#ifdef TSCH_CONF_DEFAULT_HOPPING_SEQUENCE
#define TSCH_DEFAULT_HOPPING_SEQUENCE TSCH_CONF_DEFAULT_HOPPING_SEQUENCE
#else
#define TSCH_DEFAULT_HOPPING_SEQUENCE TSCH_HOPPING_SEQUENCE_4_4
#endif
```

4.2 Question 2

Les paramètres RPL utilisés sont les suivants :

- Mode opératoire : `NON-STORING`

- Fonction objectif : `MRHOF`
- Métrique : `ETX`

4.3 Question 3

```
iotstras7@strasbourg:~$ lynx -dump http://[2001:660:4701:f0a6::9889]
   Neighbors
fe80::b184
fe80::9287
fe80::a887
fe80::b885

   Routes

   Links
2001:660:4701:f0a6::b184 (parent: 2001:660:4701:f0a6::9889) 1790s
2001:660:4701:f0a6::a887 (parent: 2001:660:4701:f0a6::9889) 1792s
2001:660:4701:f0a6::b885 (parent: 2001:660:4701:f0a6::9889) 1793s
2001:660:4701:f0a6::9287 (parent: 2001:660:4701:f0a6::9889) 1792s
```

(a) Table de routage RPL du BR

```
iotstras7@strasbourg:~$ coap get coap://[2001:660:4701:f0a6::a887]:5683/sensors/pressure
(2.05)  1019
```

(b) Requête CoAP

FIGURE 5 – Consultation des routes sur le BR et vérification de la connectivité

4.4 Question 4

Le debug a été activé dans la partie `Réseau RPL-CoAP basé sur nullrdc`

4.5 Question 5

Pour mesurer le temps de convergence, on procède de la même manière que précédemment.

```
python convergence.py ~/tsch/experiment_1/1.serial_aggregator.out
```

Nous obtenons un temps de convergence de **0.01016s**. Le temps moyen sur 10 expériences sera présenté dans la section 5.

4.6 Question 6

Nous utilisons le même profil **conso** pour monitorer la consommation énergétique des noeuds. Les courbes obtenues sur 10 expériences des noeuds seront présentées dans la section 5.

4.7 Question 7

On obtient les informations sur le nombre paquets sur la figure 6 en observant la fin du fichier `coap.out`, produit par `python coap_client.py > coap.out`

```
2019-01-05 07:17:05,459 - MainThread - __main__ - DEBUG - Statistics
2019-01-05 07:17:05,460 - MainThread - __main__ - DEBUG - Packets Sent: 47
2019-01-05 07:17:05,460 - MainThread - __main__ - DEBUG - Packets Received : 44
2019-01-05 07:17:05,460 - MainThread - __main__ - DEBUG - Percentage Packet Loss : 6.3829787234
```

FIGURE 6 – Statistiques à la fin du fichier log coap.out

5 Études des résultats

5.1 Question 1

Nous rejouons la même expérience dans les mêmes conditions à l'aide du script **ssh_script.sh** qui se lance sur un poste et qui va passer des commandes `ssh` au serveur. Il prend en argument le nombre d'expérience à réaliser

ainsi que la couche mac sur laquelle nous réalisons les expériences. Par exemple, si l'on veut rejouer 10 fois une expérience en ayant configuré la couche MAC pour le duty cycle `contikimac` et pour lancer le client CoAP en mode `GET` on lancera :
`./ssh_script.sh -number=10 -mac=contikimac -operation=GET`
Ainsi, l'ensemble des fichiers de logs générés par le script se trouveront dans le dossier
`contikimac/experiment_id` Pour les résultats que nous analyserons dans les sections suivantes, nous avons rejoué les expériences **10 fois**. Chaque expérience dure **10 minutes**.

Temps de convergence

Pour calculer le temps de convergence, nous avons programmé un script python `convergence.py` qui permet de parser le fichier des logs de `serial_aggregator` et de relever les temps correspondant au début de la construction du `DODAG` et au temps auquel le dernier noeud rejoint le `DODAG`, et de calculer la différence entre ces deux temps. Pour obtenir le temps de convergence moyen : `python convergence_mean.py ./nullrdc|./contikimac|./tsch` selon la configuration souhaitée (nullrdc, contikimac ou bien tsch)

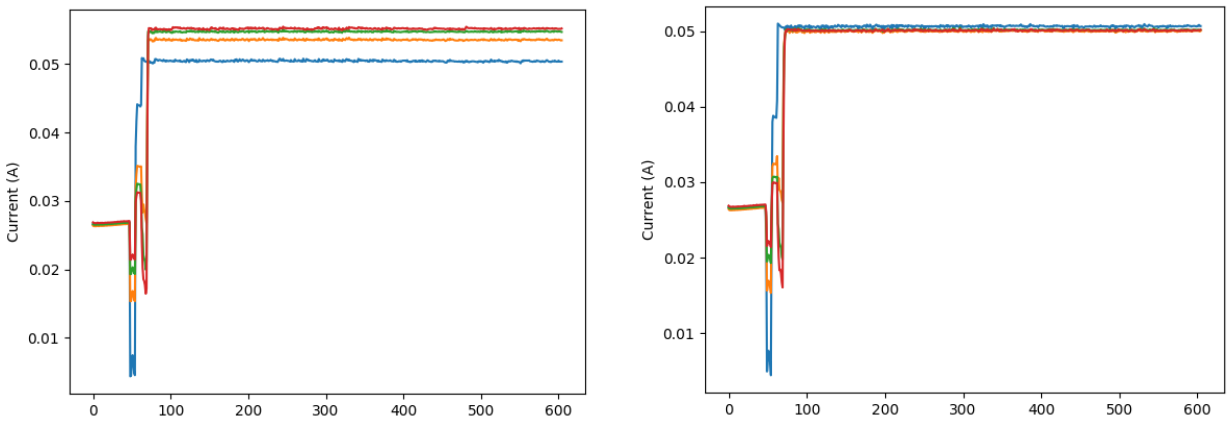
```
iotstras7@strasbourg:~/tp2$ python convergence_mean.py ./nullrdc
Command: python convergence_mean.py ./nullrdc|./contikimac|./tsch
Le temps de convergence moyen est 0.010388 secondes
iotstras7@strasbourg:~/tp2$ python convergence_mean.py ./contikimac
Command: python convergence_mean.py ./nullrdc|./contikimac|./tsch
Le temps de convergence moyen est 2.023679 secondes
iotstras7@strasbourg:~/tp2$ python convergence_mean.py ./tsch
Command: python convergence_mean.py ./nullrdc|./contikimac|./tsch
Le temps de convergence moyen est 44.667022 secondes
```

FIGURE 7 – Temps de convergence moyen

Nous en déduisons donc que `nullrdc (0.0103 s)` est le plus rapide en terme de temps de convergence, suivi de `contikimac (2.023 s)` et `tsch` qui a un temps de convergence très grand `(44,667 s)` vu que les expériences ont été faites pendant que tous les autres groupes faisaient leurs expériences. Pour `tsch`, nous avons rejoué 10 autres expériences lorsque la plate-forme était moins occupée et nous avons obtenu un nouveau résultat de `(35,019 s)`.

Consommation énergétique

Pour estimer la consommation énergétique moyenne de chacun des noeuds pour les 10 expériences, nous avons programmé un script python **plot.py** qui permet de tracer la courbe de la consommation énergétique moyenne des résultats obtenus dans les 10 expériences effectuées. Notons que ce script parse les fichiers `.oml` générés, ces fichiers ayant des nombres de lignes légèrement différents d'une expérience à une autre, nous avons donc décidé de calculer une moyenne toute les secondes sur ces fichiers et ensuite de faire une moyenne sur les 10 expériences. Pour obtenir les courbes : `python plot.py ./nullrdc|./contikimac|./tsch` selon la configuration souhaitée (nullrdc, contikimac ou bien tsch)



(a) Courbe de la consommation énérgétique des noeuds en nullrdc

(b) Courbe de la consommation énérgétique des noeuds en contikimac

FIGURE 8 – Consommation énergétique des noeuds

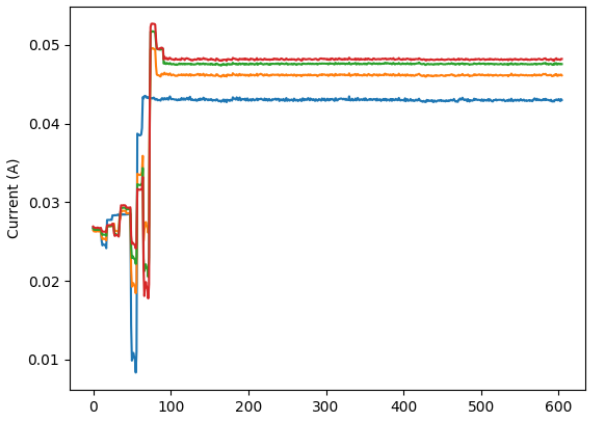


FIGURE 9 – Courbe de la consommation énérgétique des noeuds en TSCH

Les trois courbes ci-dessus représentent la consommation énergétique (courant électrique) des noeuds selon la couche MAC utilisée, dans la figure (a) qui concerne la couche MAC nullrdc la courbe en bleue représente la consommation du BR qui est en moyenne de `50 mA`, les autres courbes représentent les autres noeuds qui ont en moyenne une consommation de `55 mA`. Dans la figure (b) qui concerne la couche MAC `contikimac`, tous les noeuds ont une consommation moyenne de `50 mA`, par rapport à `nullrdc`, les noeuds non-BR consomment moins. Dans la figure 7, le noeud BR (courbe bleue) a une consommation de `42 mA`, les autres noeuds ont des consommations qui varient entre `45 mA` et `47 mA`. Nous en déduisons donc que tsch est meilleur en terme de consommation énergétique.

Temps de latence et Pertes

Le script `all_stats.py` permet d'obtenir les moyennes des paramètres suivants :

- Temps de latence
- Paquets émis
- Paquets reçus
- Pourcentage de pertes

Le script trace les différents paramètres en fonction de la couche MAC pour les 10 expériences que nous avons lancées sur chaque couche.

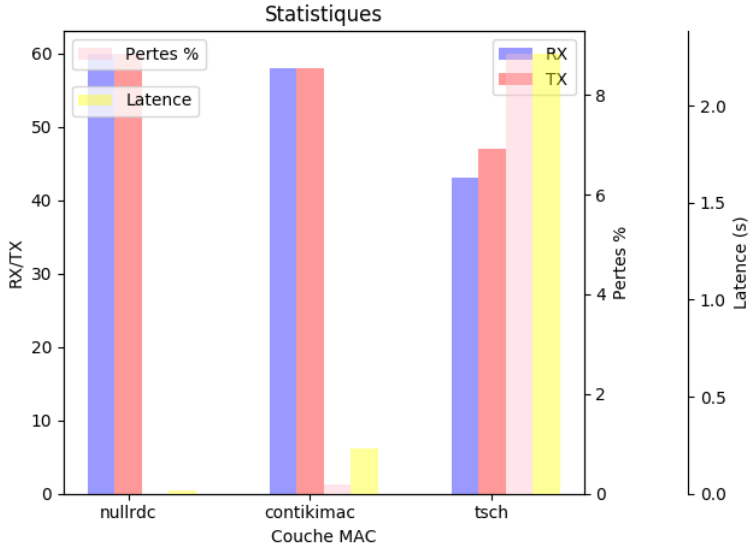


FIGURE 10 – Latence et pertes moyennes en fonction des couches MAC

Sur la figure 10, on constate que `tsch` perd environ `8.8%` des paquets et qu'il engendre une latence plus élevée que `nullrdc` et `contikimac` (Plus de `2sec` contre `0.2sec (contiki)` et `0.02sec (nullrdc)`). Nous pouvons donc en déduire que `nullrdc` a de bonne performance en terme de pertes et de latence, ce qui est logique car la radio reste allumée 100% du temps. La couche `contikimac` quant à elle présente également de bonnes performances pour

une consommation énergétique plus faible.

5.2 Conclusion

Ce TP nous a permis d'évaluer les performances des différentes couches MAC, ainsi nous pouvons conclure d'après nos analyses qu'en terme de consommation énergétique la configuration tsch est meilleure que les deux autres couches mais présente tout de même un temps de convergence très lent et un pourcentage de paquets perdus plus élevés. Bien que la couche `nullrdc` montre de bonnes performances que ce soit au niveau de la latence et des paquets perdus, cette configuration ne convient pas à un environnement contraint car elle consomme plus d'énergie et ce n'est pas ce qu'on l'on recherche dans ce type de réseau. Enfin, la configuration `contikimac` semble être la meilleure configuration car elle apporte de bonnes performances sur les quatre critères étudiés. Néanmoins, il faudrait rejouer les expériences à grande échelle pour observer l'impact du nombre de noeud sur les différentes couches mac.

Bibliographie

[1] Building contiki's tunslip6
. https://www.iot-lab.info/tutorials/build-tunslip6/.

[2] Coap server with public ipv6/rpl/6lowpan network
. https://www.iot-lab.info/tutorials/contiki-coap-m3/.

[3] Get and compile firmware for m3 and a8-m3 nodes
. https://www.iot-lab.info/tutorials/contiki-compilation/.

[4] Monitor the consumption of m3 node during an experiment
. https://www.iot-lab.info/tutorials/monitoring-consumption-m3/.