Grenoble INP
Ensimag
$2^{\text{ème}}$ année

Juin 2012

Projet de spécialité :
Sensors networks : déploiement d'un réseau de capteurs

Documentation

Membres : SOUMARE Mouhamed
TOLLARDO Thomas
VIPRET Julien

Ce document est une documentation des travaux réalisés dans le cadre du projet de spécialité **Sensors networks : déploiement d'un réseau de capteurs**. Il a pour objet l'implémentation d'un environnement d'étude des réseaux de capteurs afin de faciliter la mise en place de tests sur de tels réseaux. Ce document comportera aussi des détails d'implémentation qui permettront une compréhension plus exhaustive du code source. Enfin la dernière partie énumère des pistes d'optimisation et d'amélioration possibles de notre programme, des points qui auraient été abordés avec plus de temps.

1 Description de l'environnement

L'objectif de l'environnement d'étude des réseaux de capteurs est d'avoir un accés simple sur un réseau de capteurs. L'environnement est composé de :

1. 48 PCs connectés en réseaux possédant un serveur ssh.
2. 48 cartes *stm32w*, branchées à un ordinateur *via* un câble USB (qui émule un port série). Les données des cartes seront récupérées par les ordinateurs par cette liaison série.
3. Un ordinateur, utilisé comme contrôleur qui sera chargé d'envoyer des ordres aux autres PCs. Sur ce PC contrôleur sont acheminées les données provenant des autres ordinateurs.

Au lancement du programme, chaque carte doit être bien branché aux ordinateurs. Elles doivent être reconnues comme une liaison série par l'ordinateur (\dev\ttyACM*).

2 Documentation utilisateur

Le programme est en trois parties : une partie client (lancée sur les ordinateurs appelés noeuds où sont branchés les cartes), une partie serveur lancée sur le controleur et un scenario lancé par le serveur. La partie client fonctionne comme un daemon, elle est lancée par le serveur par la connection ssh.

2.1 Le contrôleur

L'environnement d'étude se lance à l'aide de la commande : `python launch.py` avec les arguments :

1. **-e exp** : charge le module python 'exp' et lance son scenario.
2. **-log** : crée des fichiers de logs.
3. **-h, –help** : affiche ce message et quitte le programme.
4. **-l** : liste les scénario d'expériences disponibles.
5. **-s hh :mm** : débute l'expérience dans hh :mm
6. **-w** : lance un wireshark temps-réel
7. **-g** : affiche sous forme de graphe, la topologie du réseau

Le contrôleur se connecte au noeuds en utilisant une paire clé privée-clé public au lieu d'un mot de passe. Un agent ssh est également lancé afin d'automatiser la connexion.

Le contrôleur permet différents affichages des résultats récupérés, des réglages sont nécessaires pour une utilisation optimale de l'environnement d'étude :

1. Les traces : récupérées sur le contrôleur, elles sont affichées sur la sortie standard mais peuvent également être redirigées vers une socket pour un affichage dans une ihm. Les variables `host` et `port` doivent être définies alors définies dans le fichier `controler.py`

2. Affichage des packets sniffés : le programme lance wireshark avec l'option `-w` pour afficher les packets reçus par le sniffer. Il est nécessaire d'avoir une version de wireshark ≥ 1.6

3. Affichage de la topologie : le programme lance une interface web où l'on peut visualiser un graphe avec l'option `-g`. Cependant des réglages plus spécifiques doivent être réalisés dans les scénarios.

2.2 Les scénarios

Les modules correspondant aux scénarios doivent être placés dans le dossier **experiments**. Chaque module doit être composé d'une classe Experiment qui définit l'environnement et le scénario. La classe Experiment possède deux méthodes :

1. La méthode **init** : définit l'environnement et retourne un dictionnaire dont les clés sont :
   - La clé `nodes` est liée à un dictionnaire qui répertorie les numéros des noeuds et indique par un booléen si les données récupérées sont des données brutes (`'1'`) (utilisées pour wireshark) ou des traces (`'0'`).
   - La clé `graphe` est un dictionnaire qui décrit les traces à parser reçues par le controleur (utilisée pour l'analyse de la topologie)
   - La clé `so` définit la Superframe Order
   - La clé `pan` indique quel est le noeud qui joue le rôle de PAN dans l'environnement

2. La méthode **execute** : lance le scenario. Voici les fonctionnalités possibles dans le scénario :
   - **flash** : prend en argument le noeud à flasher (ou all) ainsi que le firmware.
   - **configure** : prend en argument un noeud (ou all) ainsi que la puissance en dbm de la radio.
   - **launch** : prend en argument un noeud (ou all), lance l'écoute des capteurs et le traitement.
   - **quit** : prend en argument un noeud (ou all), ferme les connections ssh (les daemons) et toutes les connections TCP.
   - **set_beacon** : prend en argument un noeud (ou all), indique à la carte de commencer à beaconner.
   - **stop_listen** : prend en argument un noeud (ou all), arrête l'écoute de la carte.

Dans le dictionnaire de la méthode `init`, seule la clé `nodes` est obligatoire, les autres sont facultatives et sont utilisées en fonction des options mises en paramètres de `launch.py`.

Pour éteindre le programme principal, il est necessaire de rentrer `exit` dans l'entrée standard. Une commande `quit` est envoyée à tous les noeuds pour fermer le daemon. Si l'option `-log` est activé, la commande ferme aussi tous les descripteurs de fichiers.

2.3 Le daemon

Le daemon est le script python tournant sur les ordinateurs où sont connectés les cartes. Le fichier se nomme `daemon.py`, il est nécesessaire de paramétrer les variables globales `host` et

`port` qui definissent l'adresse IP et le port où doivent être envoyées les données, c'est l'adresse IP du contrôleur. Un dictionnaire commun répertoriant les firmwares disponibles doit être défini dans les fichiers `controler.py` et `daemon.py`, chaque firmware possède un unique numéro. Ainsi afin de flasher avec tels firmwares, il est nécessaire de le rajouter dans les deux dictionnaires et d'utiliser le même nom dans les scénarios.

2.4 Les firmwares Contiki

L'OS utilisé sur les cartes est Contiki, un système léger et open source adapté pour les systèmes embarqués. Différents firmwares ont été implémentés pour tester notre environnement d'étude, ils sont placés dans un dossier firmwares :

- Le firmware `sniffer` : permet de récuperer tous les paquets réseau captés par l'antenne du capteur dans le format slip. L'utilisation d'un programme de désencapsulation de slip est nécessaire (par exemple le binaire `slip2pcap`)

- Les firmwares `udp-server` et `udp-client` : le client envoie des messages au serveur et le serveur répond au client par le même message. Ces deux firmwares peuvent être utilisés conjointement avec le firmware `sniffer`. A ce moment la, le sniffer récupère les messages échangés entre le client et le serveur.

- Les firmwares `firmware_pan` et `firmware_nonpan` utilisent le protocole de routage LoadNG. Le PAN commence à beaconner au démarrage de l'experience et les noeuds non PAN recoivent ces beacons. Lorsque les noeuds se coordonnent avec un père, ils commencent à beaconner à leur tour. Différents slots sont alloués aux coordinateurs pour éviter un maximum de collisions.

Les firmwares utilisés doivent être compatibles avec les scripts python. Il envoie des requêtes asynchrones aux capteurs, un code est spécifié dans le fichier `daemon.py`. Le capteur doit pouvoir répondre aux requêtes :

- `0` : le capteur renvoie un message à parser avec son adresse mac.
- `2 $val` : le capteur paramètre la puissance de sa radio en dbm avec la valeur donnée en paramètre.
- `3 $delay $BO $SO` : le capteur commence à beaconner et paramètre les valeurs du start-time, du beacon order et de la superframe order avec les valeurs données en paramètres.

Dans le cas du firmware `firmware_pan` et `firmware_nonpan`, des traces sont placées dans le code au niveau de la couche mac pour voir comment se découvre les noeuds (réceptions de beacons, association response).

3 Améliorations possibles

Cette partie regroupe toutes les améliorations envisageables de notre environnement d'étude ainsi que quelques points à reprendre en utilisant des structures plus adaptées.

3.1 La synchronisation

La synchronisation dans notre environnement n'est pas optimisée. Elle se fait par l'utilisation de divers booléens dont voici la liste exhaustive :

- `daemon.py` : les variables `listen` et `config`.
- `controler.py` : un dictionnaire `deb_flash` (verrous qui attendent la fin du flash).

De plus une méthode qui teste le verrou a été implémentée : `checkAndWait` :

```
def checkAndWait(k):
    """Waits until the sensor finishes to flash"""
    while (deb_flash[k] != 1):
        None
```

Pour optimiser la synchronisation, il faudrait utiliser des structures adaptées comme des sémaphores ou des mutexs. Ces structures doivent être accessibles *via* des librairies python.

3.2 La gestion des erreurs

Arêter tous les programmes python est quelque chose de difficile dans une application qui fonctionne avec plusieurs ordinateurs. Le problème étant que pour arrêter les daemons, il faut envoyer la commande `quit` *via* la connexion ssh et que parfois cette connexion se coupe car une erreur survient au niveau du programme du contrôleur. Une solution serait de faire une récuperation d'erreur et de s'assurer que tous les daemons soit bien fermés à la fin de l'application. Actuellement un script `clean.sh` se connecte sur toutes les machines ultèrieurement et kill les processus python. Si un daemon n'est pas arrêté, à la prochaine exécution de celui-ci, il ne pourra pas accéder au capteur car la ressource est déjà prise par le premier daemon. Une récuperation des erreurs doit aussi être faite lors du flashage des noeuds car des problèmes de détections des cartes surviennent.

3.3 L'interface web

Actuellement l'interface web n'affiche que la topologie du réseau. Une amélioration possible serait de permettre également l'affichage des traces affichées sur le terminal ou de permettre un arrêt du rafraichissement de l'image.