Intégration des services à la plateforme pour démonstrateurs

Laurent Reynaud, François Jan, Laurent Ciarletta, Tom Leclerc, Gashti Shahab

Projet SARAH (Services Avancés pour Réseaux Ad Hoc)

Sous-Projet 5 (SP 5) : Expérimentations

L5.2 - Intégration des services à la plate-forme pour démonstrateurs

Auteurs :

Laurent Reynaud, François Jan - Orange Labs R&D
Laurent Ciarletta, Tom Leclerc – INRIA Madynes
Shahab Gashti - Ucopia

Date : 05/03/2010

Partenaires:
Orange Labs R&D, INRIA Madynes, Ucopia

Responsable de SP:

Laurent Reynaud
Orange Labs R&D
laurent.reynaud@orange-ftgroup.com

Résumé :

Ce livrable décrit d'une part dans le présent document les différents composants logiciels développés et mis en place par les partenaires dans le cadre du WP 5, et en particulier lors de la phase d'intégration des différents modules. D'autre part, ce livrable contient également les archives du code relatif aux différents composants logiciels décrits.

SOMMAIRE

1. Généralités

1.1. Objectif du document

Ce document a pour objectif de décrire le contenu des différentes archives de code fournies dans le cadre de ce livrable L5.2. En particulier, chaque composant utilisé dans le cadre du démonstrateur dans les locaux d'Ucopia ou à la Cité des Télécommunications à Pleumeur-Bodou (22) est ici décrit, à la fois dans son fonctionnement, de manière brève, mais aussi au niveau des implications en termes d'environnement de développement, de procédures d'installation et de mise en œuvre.

2. Démonstrateur Ucopia : détail des composants logiciels

Les protocoles de découverte de services standards ont en général une lacune, c'est qu'ils ne considèrent pas l'utilisation des profils utilisateurs afin de protéger les ressources de tout accès non autorisé. Notre architecture propose un mécanisme protégeant l'accès aux ressources spécifiques du réseau en fonction des profils utilisateurs ou groupes d'utilisateurs. Un profil décrit les appareils et services auxquels un utilisateur donné peut accéder.
Ce module a pour but d'offrir un mécanisme de découverte de services en fonction des profils utilisateurs ou groupes d'utilisateurs fournis par le module de gestion de contexte utilisateur situé dans le contrôleur UCOPIA. Nous illustrons notre extension de l'UPnP permettant la découverte de services en fonction du contexte utilisateur. Le rôle principal de ce module est de limiter l'accès au réseau et la découvrabilité des appareils pour tout utilisateur ou groupe d'utilisateurs non autorisés à utiliser les services du réseau. Notre solution est compatible avec le framework existant et actuel de l'UPnP.
Ce module est en charge d'obtenir une liste des services autorisés et disponibles pour un utilisateur donné. La liste de services est ensuite communiquée à l'utilisateur à sa demande. Ce module comporte deux composants interdépendants :

2.1. Catalogue des descriptions

Dans notre approche pour l'extension de l'UPnP, le contrôleur comportent deux catégories de descriptions. La description du contrôleur UPnP (UCOPIA) lui-même contenant des unités de données telles que le nom, le constructeur, le modèle, et qui permet à cet équipement UPnP d'annoncer sa présence et ses capacités. La seconde catégorie fournit un fichier de description attaché à chaque utilisateur, appelé « profil utilisateur UPnP ». Dans ce format XML, le contrôleur UCOPIA est considéré comme le périphérique racine (root device) et tous les autres appareils dans le réseau sont vus comme ses appareils embarqués. Ce document décrit en effet les services UPnP découvrables par utilisateur.
Il est en effet possible d'adopter deux approches, statique ou dynamique, afin de gérer les profils utilisateurs UPnP. Dans l'approche statique, le module de gestion de contexte utilisateur calcule tous les fichiers de description en fonction des différents contextes de l'utilisateur. Dans l'approche dynamique, le module de gestion de contexte utilisateur se charge d'adapter ce document en temps réel en fonction des changements de contexte. En fait, le profil utilisateur UPnP est utilisé par l'extension de l'UPnP, décrite ci-après, qui répond aux messages de recherche d'un utilisateur en fonction de son profil utilisateur UPnP.

2.2. Contrôleur UCOPIA augmenté de l'extension du protocole SSDP

Le protocole Simple Service Discovery Protocol (SSDP) est l'un des composants clés de l'architecture UPnP. C'est un protocole de découverte de service entièrement distribué qui fonctionne dans un environnement multicast. Le contrôle de réseau devient toutefois compliqué dans un tel environnement en termes d'authenticité de message, de permission d'accès, et de confidentialité. Dans un réseau UPnP, tous les appareils et services envoient périodiquement des annonces. Tous les appareils dans le réseau capables de faire correspondre les critères que le point de contrôle recherche émettent une réponse UDP en unicast indiquant l'URL de leur document de description. Par conséquent, n'importe quel nœud dans le domaine multicast local est capable de découvrir et aussi de contrôler les appareils UPnP. Notre extension de l'UPnP est donc motivée par le manque d'un point centralisé pour les environnements multicast. Cette extension de SSDP est développée dans le contrôleur UCOPIA, elle ne demande pas de modification dans les autres terminaux UPnP, et est compatible avec le protocole SSDP standard. L'idée générale de notre extension SSDP est tout à fait simple. Le contrôleur répond aux requêtes de découverte de service prenant en compte le profil utilisateur UPnP. Les messages de découverte (M-SEARCH) sont émis par des utilisateurs munis du point de contrôle UPnP en multicast, à l'adresse 239.255.255.250, afin d'interroger appareils et services. Ces messages de découverte multicast SSDP sont capturés par le

contrôleur. Suite à la réception d'un message de découverte SSDP, le contrôleur exécute les opérations suivantes :

```
SI (l'utilisateur est connu du contrôleur) {
    récupérer l'url du profil utilisateur UPnP;
    SI (le contrôleur fait correspondre ce que le point de contrôle cherche avec les services
    existants dans le fichier de description XML récupéré) {
        envoyer un message unicast UDP du type NOTIFY avec l'URL du document
        de profil utilisateur UPnP;
    }
}
SINON{
    ignorer la requête;
}
```

Par conséquent, si un utilisateur veut plus d'informations à propos des appareils et des services disponibles dans le réseau, il peut récupérer le document de description XML via l'URL fourni par le contrôleur. La Figure 1 montre également un diagramme de séquence des opérations expliquées ci-dessus.

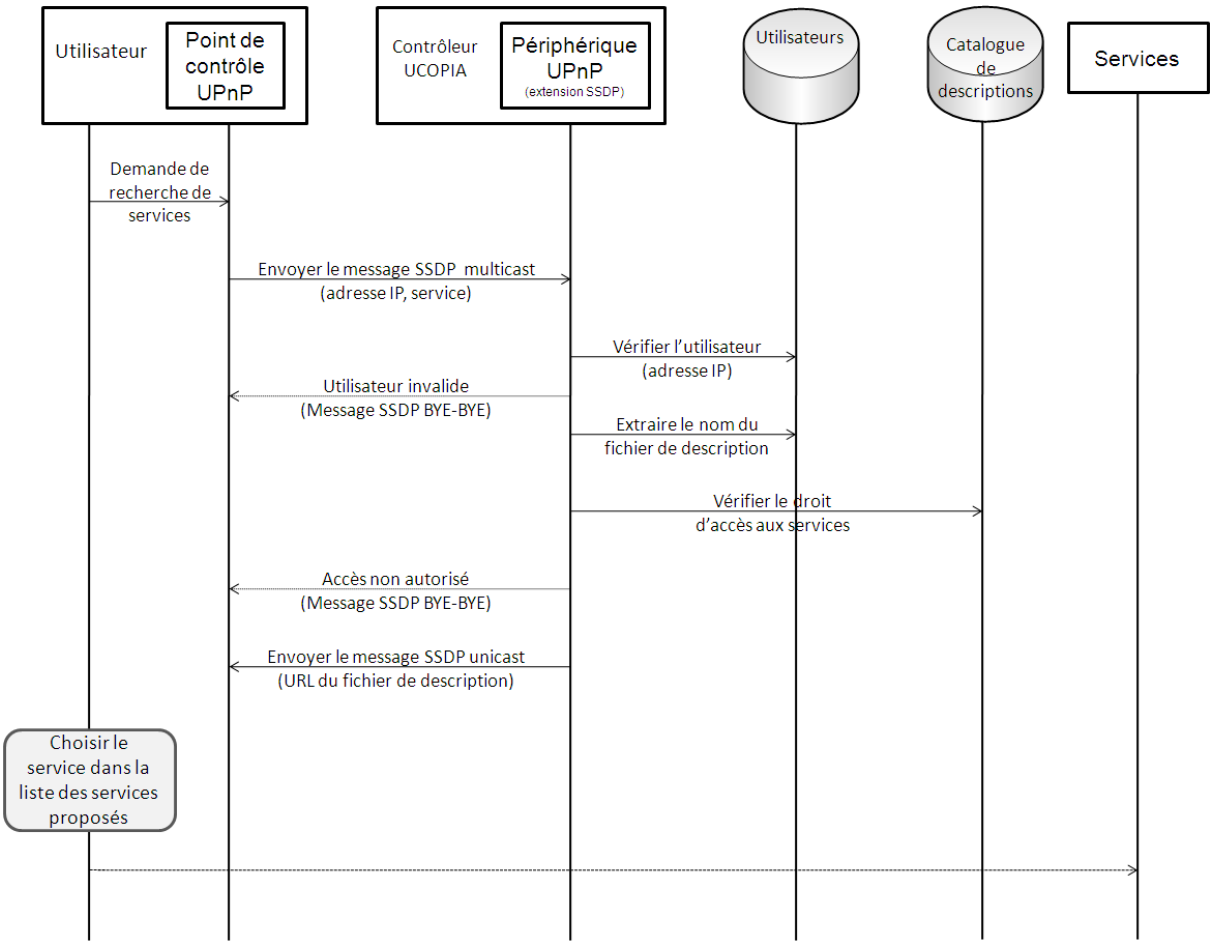


Figure 1 - déroulement de la découverte des services disponibles dans le réseau

2.3. Implémentation

Nous avons expliqué le fonctionnement du module de découverte de services selon les modifications que nous avons apportées à la technologie UPnP. Ces modifications concernent le protocole de la découverte de services dans l'UPnP afin d'offrir un mécanisme de découverte en fonction du profil utilisateur. Les modifications ont été faites au niveau du paquetage de développement pour UPnP qui rassemble l'ensemble des protocoles de la technologie UPnP tel que SSDP et fournit une API pour l'implémentation des appareils et points de contrôle UPnP. Nous avons choisit le paquetage de développement « Cyberlink » en Java mais la faisabilité de notre approche a également été vérifiée dans d'autre paquetage de développement UPnP à code source libre tel que « Portable SDK for UPnP Devices » (libupnp) développé initialement par Intel. En effet, un périphérique fournit son document de description quand celui-ci effectue une initialisation avec le paquetage de développement. Le protocole SSDP implémenté dans le paquetage de développement se sert alors de ce document de description pour annoncer le périphérique et répondre aux requêtes de recherche.
Par exemple, dans notre approche, le contrôleur UPnP (UCOPIA), implémente ses services et effectue l'initialisation dans le paquetage de développement de la même façon que l'approche standard. Quand le point de contrôle UPnP (client) envoie une requête de recherche pour un service, l'extension du protocole SSDP consulte le profil utilisateur et vérifie si l'utilisateur a le droit d'accéder au service qu'il cherche. Si c'est le cas, cette extension répond au point de contrôle avec l'URL du profil utilisateur UPnP.
Le module de découverte de services illustre plus en détail la façon dont le profil utilisateur est utilisée afin de contrôler la découvrabilité des appareils.

2.3.1 Module de découverte de service

Ce module est implémenté dans le contrôleur et utilise la table des profils utilisateurs pour récupérer le profil utilisateur UPnP. Afin d'atteindre cet objectif, nous avons ajouté des procédures suivantes au protocole SSDP dans le paquetage de développement UPnP :

Procédure identifier
Cette procédure permet d'identifier l'utilisateur par son adresse IP. Cette méthode vérifie si l'utilisateur a le droit d'accéder au réseau UPnP en consultant la table des utilisateurs grâce à une interface de communication avec le module de gestion de contexte utilisateur. Elle prend comme paramètre l'adresse IP de l'utilisateur et retourne une variable booléenne. L'adresse IP de l'utilisateur est alors récupérée dans le message SSDP SEARCH envoyé par l'utilisateur qui contient l'adresse IP de celui-ci.

Procédure charger
Cette procédure permet de charger le profil utilisateur UPnP. Si l'utilisateur est identifié, cette procédure extrait le fichier de description associé à l'adresse IP dans la table des utilisateurs. Elle charge ensuite le fichier correspondant dans un serveur Web incorporé dans le paquetage de développement UPnP.

Procédure correspondre
Cette procédure permet de vérifier si l'utilisateur peut accéder au service demandé. Cette procédure fait une comparaison entre ce que l'utilisateur cherche et ce que le fichier de description comporte.

Procédure répondre
Cette procédure permet de réponde à l'utilisateur si le résultat de la comparaison fournit au moins un service qui correspond à ce que l'utilisateur cherche. Elle envoie en unicast (message SSDP) à l'utilisateur l'URL de son fichier de description (profil UPnP).

2.4. Vision générale de l'implémentation

Nos travaux dans le cadre du démonstrateur Ucopia ont principalement porté sur le développement du module de la découverte de service. Le choix de notre implémentation s'est basé sur les critères industriels. La solution UCOPIA, une implémentation propriétaire, offre des fonctionnalités telles que l'authentification, la gestion de contexte utilisateur, et le filtrage de paquets. De ce fait, nous avons exploité les modules écrits par la société UCOPIA. Par conséquent, nous avons apporté une innovation à l'UPnP, étant une technologie incontournable dans les réseaux résidentiels, afin d'offrir un mécanisme de découverte de services en fonction des profils utilisateurs. Par ailleurs, cette extension n'est pas limitée au contrôleur et elle peut être adoptée par n'importe quel périphérique UPnP voulant offrir des services contextuels sans aucune modification sur les autres périphériques existants. **Le code de l'implémentation de cette extension reste confidentiel**.

3. Démonstrateur à la Cité des Télécommunications

3.1. Composant de localisation par carte des signaux améliorée

3.1.1 Généralités

Ce paragraphe pour objectif de présenter le composant logiciel assurant la localisation des utilisateurs dans l'application du musée des télécommunications. Il s'appuie sur le mécanisme LAMBDA (*Location Aid Mechanism By Distance Adjustment*), dont le but est de fournir un algorithme de localisation en 802.11 complémentaire a celui de la localisation par carte des signaux et permettant de résoudre les faiblesses de ce dernier. Tandis que l'algorithme de localisation par carte des signaux exploite les informations issues des nœuds fixes, l'algorithme LAMBDA exploite les informations issues des nœuds mobiles. Il peut arriver que l'algorithme LAMBDA exploite également les informations issues des nœuds fixes en cas de défaillances du fonctionnement de l'algorithme de localisation par carte des signaux.

L'algorithme LAMBDA seul ne suffit pas à restituer une localisation précise pour les nœuds mobiles du fait des variations trop importantes des niveaux de signal en fonction de la mobilité et de l'environnement. Cependant L'algorithme LAMBDA trouve sa force dans la combinaison avec d'autres méthodes de localisation telle que la carte des signaux. En effet LAMBDA permet d'améliorer significativement la localisation obtenue par la carte des signaux.
Dans le fonctionnement, la carte des signaux s'applique uniquement aux informations reçues de la part des nœuds fixes et LAMBDA s'applique aux informations reçues de la part des nœuds mobiles.

La localisation se fait à partir d'échanges d'informations de localisation de la part de tous les nœuds du réseau. Le réseau étant composé de nœuds fixes et mobiles, les nœuds fixes connaissent leurs localisations, les autres nœuds (les nœuds mobiles) la calculent.
Tous les nœuds du réseau transmettent leurs localisations à tous leurs voisins. D'autres parts, les nœuds possèdent des informations sur les puissances du signal reçu pour chacun des nœuds du voisinage. A partir de ces informations les nœuds mobiles mettent à jour leurs localisations.

3.1.2 Informations de localisation nécessaires

Informations envoyées dans le voisinage	Informations obtenues de la part des voisins
Coordonnée de localisation X Coordonnée de localisation Y Statut du nœud : Fixe, faible mobilité, mobilité moyenne, mobilité élevée, non fiable.	Information de localisation et statut du nœud Niveau de puissance du signal reçu. (cette information est récupérée à chaque réception d'un message provenant d'un autre nœud)

Chaque nœud envoie à son voisinage ses informations de localisation et de statut. A la réception de ces messages, les nœuds récupèrent également le niveau de puissance de signal reçu.
Le niveau de puissance servira à établir une distance entre les nœuds. La relation entre la puissance et la distance peut être établie de diverses manières connues. On pourra par exemple réaliser un tableau de conversion de puissance en distance de manière empirique à partir de mesures réelles.
Le statut "fixe" désigne des nœuds ayant une localisation fixe. Ces nœuds connaissent leur localisation et celle-ci ne change pas du fait de leur immobilité. Le statut "faible mobilité" désigne des nœuds dont la vitesse est inférieure à 1 m/s. Le statut "mobilité moyenne" désigne des nœuds dont la vitesse est comprise entre 1 m/s et 2 m/s. Le statut "forte mobilité" désigne des nœuds dont la vitesse est supérieure à 2 m/s. A la connexion, un nœud ne pourra pas calculer sa localisation avant d'avoir reçu des informations de localisation de 3 nœuds différents. De même les informations de

localisation qu'il enverra ne seront pas fiables. Le nœud sera dit "non fiable" jusqu'à ce que sa localisation soit établie.

3.1.3 Implémentation du composant

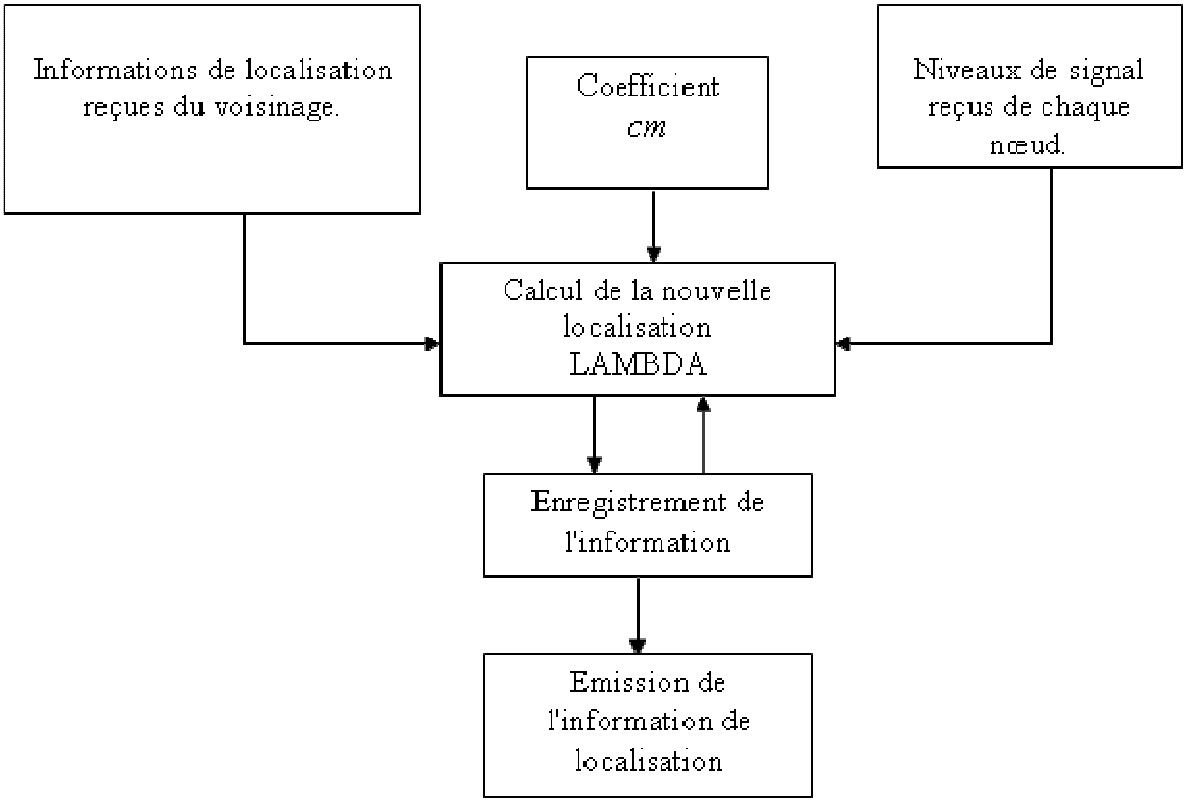


Figure 2 - Décomposition fonctionnelle du composant de localisation LAMBDA

Ce composant comporte deux processus qui s'exécutent en parallèle.

- Un processus client qui va extraire les niveaux de signal issus du driver WiFi, réaliser le calcul de localisation, et va ensuite envoyer à tous ses nœuds voisins sa nouvelle position.
- Un processus serveur, qui lui va recevoir toutes les informations de localisation des nœuds voisins dans le réseau, et va ensuite les stocker pour que le processus client puisse y accéder.

Remarques d'implémentation :
Ce composant est développé en c. Une partie des variables utilisées est déclarée dans le fichier `net.h`, *i.e.* une série 'define' dont le rôle est le suivant :

- PORT : définit sur quel port seront ouverts les sockets de communications entre les nœuds.

Ensuite la structure `node` est déclarée, celle-ci, permet de stocker la position et le statut de chaque nœud du réseau (`statut : non fiable = 0, fixe = 1, faiblement mobile = 2, moyennement mobile = 3,` fortement mobile = 4). Il y aussi une structure `rssi`, qui va permettre le stockage d'informations sur les niveaux de signal reçus depuis les autre nœuds du réseau (min, max, moyenne et variance).

La seconde partie commune est déclarée dans le fichier `main.c`. Il s'agit des *vectors* déclarés avant le `main()` (tableaux de données) contenant des informations pour chaque nœud connu, ou chaque nœud est repéré par un indice unique dans chacun des *vectors*. Ceux-ci contiennent MAC, IP, structure `node`, structure `rssi` et délai de mise à jour de la position pour chaque nœud mobile voisin.

3.1.3.1 Processus serveur :

Ce processus est appelé dès le lancement du programme par un *thread* (si l'algorithme choisit n'est pas uniquement la carte des signaux). Il va ouvrir un socket en UDP, et va ensuite attendre de recevoir des structures "`node`" envoyées par les nœuds voisins. Une fois une structure reçue, il vérifie si l'IP du nœud émetteur est connue dans le *vector* IP. Si oui, il remet à 0 la valeur du délai `delay`. (variable qui définit le temps depuis la dernière réception des informations d'un nœud mobile) et met à jour la structure du nœud avec celle reçue, sinon il se remet en attente.

3.1.3.2 Processus client :

Ce processus tourne en boucle et réalise les mêmes opérations à intervalles de temps réguliers (définit par la variable durée d'analyse spécifiée lors du lancement du programme), il découvre les nœuds du réseau, il fait remonter les puissances de signaux reçues depuis le driver, il réalise le calcul localisation, et enfin il envoie en *broadcast* la nouvelle position calculée.
Ce processus ayant plusieurs fonctionnalités, sont détaillées chacune de ses parties afin de mieux comprendre le fonctionnement global du programme.

- **Découverte des nœuds voisins**

Pour découvrir les nœuds voisins, et stocker dans les vecteurs les paramètres utiles à sa reconnaissance (MAC et IP). On utilise plusieurs fonctions, la première se nomme "`read_fixes`" et elle va lire dans le fichier "`fixes.txt`" ou se trouvent les adresses de tous les nœuds fixes ainsi que leur position.
La seconde fonction, se nomme "`read_arp`" et va aller chercher dans la table ARP du nœud toutes les adresses MAC et IP de chaque nœud connu, et si le nœud n'est pas encore stocké dans les *vectors* (MAC, IP) il sera ajouté (un module de `olsrd`, "`arprefresh`", a été ajouté pour que la table ARP soit remplie avec tous les nœuds connus).
La dernière fonction est "`read_MAC`", elle récupère les adresses MAC et la puissance du signal reçue (issus du driver WiFi) des nœuds ayant émis des messages. Si cette adresse MAC n'existe pas dans les *vectors* elle sera ajoutée, et les autres vecteurs seront initialisés.
Lors de la découverte des paramètres permettant d'identifier un nœud, on initialise en même temps les autres vecteurs (signaux reçus, structure `node`, `delay`), afin que la condition de l'indice identique de chaque nœud dans les vecteurs soit respectée.
Une fois toutes ces informations récoltées, en fonction des choix d'algorithme réalisés au démarrage du programme, la fonction va appeler une ou plusieurs fonctions de calcul de localisation après avoir réalisé des statistiques sur les niveaux de signal reçus.

- **Calcul de la position du nœud**

Le calcul de la position du nœud est réalisé à la fin de la fonction "`read_MAC`", il est précédé par une fonction qui assure le calcul de statistiques sur les niveaux de signaux relevés par bornes (moyenne et variance). Ensuite, il peut faire appel à deux fonctions :

- "`localise`" : pour calculer la position à partir de la carte des signaux.
- "`compute`" : localisation à partir de l'algorithme décrit dans le document.

La fonction "`localise`" est appelée une fois pour tous les nœuds fixes, la fonction "`compute`" est elle appelée pour chaque nœud mobile (ou pour tous les nœuds) connu dans le réseau.

- **La fonction "`localise`" :**

Cette fonction, va lors de son premier appel lire la carte des signaux grâce à la fonction "`read_map`", qui elle va enregistrer les valeurs du fichier texte dans des variables globales.
Une fois la carte des signaux lue, on calcule la distance entre les valeurs de signaux reçues en provenance de chaque borne, et les valeurs de signaux pour chaque point de la carte. Ensuite, on prend la distance calculée la plus faible et on calcule notre nouvelle position, avec 30% de la position de la mesure ayant la distance la plus faible et 70% de la position précédente. Toutefois si la distance calculée est supérieure à un seuil fixé, le calcul localisation ne sera pas réalisé et nous utiliserons la fonction "`compute`" décrite ci-dessous avec les nœuds fixes du réseau.

- La fonction "`compute`" :

  Dans cette fonction, on utilise l'algorithme LAMBDA. Dans la première partie, on va calculer le coefficient multiplicateur *cm* en fonction de la structure `node` envoyée par le nœud voisin et des niveaux de signal relevés. Si $cm \leq 0$, on quitte le programme, sinon on calcule la distance entre les deux nœuds (à partir d'un tableau de correspondance signal en dBm et distance en pixels). Ensuite, on calcule la distance entre les deux nœuds à partir de leurs coordonnées, et on peut calculer les nouvelles coordonnées à partir des deux formules décrites dans LAMBDA.

- **Envoi des informations**

Après avoir calculé la nouvelle position du nœud, on va tout d'abord mettre à jour son statut (sa mobilité) pour cela on utilise la fonction "`get_statut`". Celle-ci va calculer la distance entre la position précédente et la nouvelle, on divise ensuite cette distance par la valeur en pixels d'un mètre réel et par notre durée (pour obtenir une vitesse de déplacement en mètres par seconde). Ensuite selon le résultat on ajuste le statut.
Une fois toutes notre structure `node` à jour, on peut l'envoyer sur le réseau pour cela on utilise la fonction "`send`". Dans cette fonction, on commence par configurer le socket en lui spécifiant que l'on utilisera le protocole UDP (SOCK_DGRAM), et que l'envoi se fera en *broadcast* (un script permet au démarrage de récupérer l'adresse de *broadcast*) en spécifiant une option sur le *socket*. Une fois ceci effectué, il suffit d'envoyer on peut envoyer la structure locale "`node`" de notre nœud sur le réseau et de fermer le socket.

- **Débogage**

Le programme comporte aussi un composant de *debug*, celui-ci est activable par un `define` dans le fichier « `geoloc.h` ». Celui-ci stocke ensuite dans un fichier nommé "`debug`" les informations suivantes :

- Position calculée par carte des signaux (si utilisée)
- La distance calculée entre chaque position de mesure de la carte des signaux (dans le cas ou on l'utilise).
- Position calculée à partir de chaque nœud mobile (si utilisés avec carte des signaux)
- Position calculée à partir de chaque nœud du réseau (sans utilisation de la carte des signaux)
- Niveaux de signal relevés depuis les nœuds n'utilisant pas la carte des signaux, avec la moyenne, variance, min et max.

- **Fichiers utilisés en entrée / sortie**

Pour les communications entre le programme et l'interface, ou bien pour la récupération des niveaux de signal, le programme va lire dans un certain nombre de fichiers.

`fixes` : contient les adresses MAC et IP ainsi que les coordonnées des bornes fixes
`messages` : sert de fichier de stockage au *buffer* `dmesg` dans lequel sont stockés les niveaux de signal.
`signaux` : contient les adresses MAC et les niveaux de signal correspondant pour chaque message reçu sur le driver (le script `sig.sh` réalise la création des fichiers `mess` et `res`).
`pos` : contient les coordonnées calculées du nœud et son statut (indice de mobilité).
`signal_map` : contient la carte des signaux.
`version` : contient la version de l'algorithme donnée au démarrage.
`nœuds` : contient tous les nœuds fixes du réseau et tous les nœuds fixes desquels on à reçu des informations, avec leur adresses MAC et IP leur coordonnées et le niveau de signal reçu depuis ces nœuds.
`broadcast` : contient l'adresse de *broadcast* du réseau (récupérée par le script `broadcast.sh`).
`IP` : contient l'adresse IP locale du nœud (récupérée par le script `ip.sh`).
`debug` : contient des informations sur le déroulement du programme

3.1.3.3 Utilisation du composant

Au lancement, l'algorithme nécessite trois paramètres, le premier sert à définir quel type d'algorithme de localisation on souhaite utiliser.

- 0 : On utilise uniquement la carte des signaux
- 1 : Carte des signaux + nœuds mobiles
- 2 : Algorithme identique à celui des nœuds mobiles pour les bornes fixes

Le deuxième paramètre est la durée d'analyse du programme, c'est la durée pendant laquelle on va stocker les niveaux de signal reçus, et c'est donc la durée entre deux calculs de localisation. Le troisième paramètre lui définit le temps de conservation des informations de localisation des nœuds mobiles. En fait, il correspond au nombre de durée d'analyse durant la quelle on souhaite conserver les informations reçues (uniquement utile pour les nœuds mobiles).

3.2. Composant de démonstration à la Cité des Télécommunications

Ce composant est une application développée en c++ au moyen des librairies graphiques Qt, pour une mise en place sur des terminaux de type Nokia N810. L'utilisation du code de ce composant nécessite donc quelques manipulations préalables, qui sont décrites dans les paragraphes suivants.

3.2.1 Driver 802.11

Le driver 802.11 utilisé est celui d'origine du N810 (`cx3110x` *chipset* connexant) avec une modification, qui stocke dans le buffer `dmesg` pour chaque paquet reçu l'adresse MAC et le niveau du signal. Mais pour cela, il a fallut tout d'abord récupérer les sources du noyau Linux afin de pouvoir compiler le driver. Les sources sont disponibles à cette adresse en archive.

```
http://repository.maemo.org/pool/maemo4.0/free/source/k/kernel-source-rx-34/
```

Une fois les sources récupérées et décompressées (dans un répertoire de `scratchbox`, voir le chapitre suivant), il faut générer le fichier de configuration pour celles-ci. On peut ensuite compiler notre driver en précisant le chemin vers ces sources. Voici un bref enchainement des commandes à réaliser pour compiler les sources du noyau et du *driver*.

```
apt-get source kernel-source-rx-34
cd kernel-source-rx-34-2.6.21/
make n800_defconfig
make oldconfig
```

Une fois les sources du noyau prêtes, on peut compiler le *driver*, pour cela il faut récupérer les sources de celui-ci qui sont disponibles à cette adresse :

https://garage.maemo.org/projects/cx3110x/

Une fois ceci effectué, pour compiler le *driver*, il faut se placer dans le premier dossier de l'archive du driver et entrer la commande suivante (avec le chemin adéquat):

```
make KERNEL_SRC_DIR=/chemin vers les sources kernel/kernel-source-rx-34-2.6.21/ modules
```

Le module du driver est ensuite disponible dans le deuxième répertoire de l'archive de celui-ci, sous le nom de `cx3110x.ko`. Il ne reste plus qu'à le mettre en place, pour cela on peut le placer sur la carte de stockage du N810. Ensuite, il faut réaliser les commandes suivantes pour retirer le driver existant insérer le notre et le calibrer.

```
rmmod cx3110x
insmod /chemin vers le driver/cx3110x.ko
```

```
chroot /mnt/initfs /usr/bin/wlan-cal
```

Le driver est maintenant près à l'emploi et pour le lancer à chaque démarrage du PDA il faut placer celui-ci dans le répertoire suivant :

```
/mnt/initfs/lib/modules/
```

Dans le driver développé, dans la fonction recevant tous les paquets "`mask_framerx`" (qui est présente dans le fichier "`sm_drv.c`") on va récupérer les niveaux de signal reçus. Pour cela, on récupère l'adresse MAC du nœud émetteur (pour être sur que ce soit un nœud voisin) et on récupère ensuite le niveau de signal reçu à partir de ce nœud en utilisant une interruption du chipset 802.11. On extrait ensuite dans le *buffer* `dmesg` l'adresse MAC du nœud émetteur du message et le niveau de signal obtenu.

3.2.2 Environnement de développement / compilation

L'outil de cross-compilation utilisé pour le composant de démonstration à la Cité des Télécoms est *scratchbox*. Il peut supporter différentes distributions de Linux, et plusieurs architectures de processeurs (ARM, x86, ...). Dans le cas du composant, il faut l'utiliser avec la version 4 de Maemo, et une architecture ARM qui correspondent au Nokia N810.

Il faut installer l'outil de cross-compilation avec Maemo par l'intermédiaire d'un script. Ce script se trouve à cette adresse :

`http://repository.maemo.org/stable/diablo/maemo-sdk-install_4.1.2.sh`

3.2.3 Disposition des sources du code du composant

Le composant de démonstration à la Cité des Télécoms est principalement constitué d'une arborescence en 3 répertoires :

```
- conf
- images
- videos
```

Le premier répertoire, conf, contient l'ensemble des fichiers de configuration nécessaire au bon fonctionnement de ce composant. Il contient également l'exécutable ainsi que les fichiers d'entrée/sortie du composant précédent de localisation.
Le second répertoire, images, contient l'ensemble des icones et images utilisées par l'application, tandis que le répertoire videos contient les vidéos lues pendant le déroulement de l'application.
L'ensemble des fichiers est fournie dans l'archive fournie avec ce livrable, à l'exception des vidéos, pour des raisons de droit d'usage. (Néanmoins, n'importe quelle vidéo XviD/divX peut-être utilisée avec cette démonstration.)

3.3. Composant de découverte de service dans la partie Ad hoc

3.3.1 Généralités

Ce paragraphe pour objectif de présenter le composant logiciel assurant la découverte de service dans la partie ad hoc du démonstrateur. La découverte de service assuré par le protocole Zeroconf (http://www.zeroconf.org/). Zeroconf, raccourcis pour zéro configuration est un ensemble de techniques qui permettent l'usage normal d'un réseau IP bien que aucun serveur ne soit présent dans le réseau. Zeroconf repose sur le standard DNS et est composé de 3 éléments : Auto-IP, permet d'assigner automatiquement (sans serveur DHCP) une adresse IP à un nœud. Multicast DNS (mDNS), permet d'avoir les fonctionnalités d'un serveur DNS (ex : résolution de nom à partir de

l'adresse IP) sans aucun serveur central. DNS-SD, permet de structurer les entrées DNS (et mDNS, puisque mDNS repose sur le standard DNS) pour y insérer des descriptions des services et ainsi permettre la découverte de service en utilisant des requêtes DNS standard.

Zeroconf utilise la structure multicast pour diffuser ses requêtes dans le réseau. Zeroconf à été conçu pour les réseaux filaires où la présence de multicast est de nos jours couramment le cas. Dans les réseaux de type Ad hoc, cependant, obtenir une structure multicast est très difficile. De plus nous n'avions aucun protocole multicast adapté aux réseaux ad hoc à notre disposition. Pour remplacer la structure multicast nous avons développé une structure reposant sur un protocole de *clustering* (NLWCA) avec au dessus un protocole de diffusion (SLSF : *Stable Linked Structure Flooding For Mobile Ad Hoc Networks* - ISWPC 2010) proche du *bordercast* pour assurer la communication entre les clusters formés par NLWCA.

La combinaison d'un structure de dissémination (SLSF[1]) avec en son cœur une structure de clusters (NLWCA[2]) et Zeroconf permet une sélection aisé et rapide des nœuds servant de cache (nœud intermédiaire stockant des informations de manière altruiste) à savoir les *clusterheads*.

3.3.2 Implémentation

Pour l'implémentation de l'ensembles des composantes la structure de découverte de service nous avons utilisés JANE (*Java Ad-hoc Network Emulator*) qui est un outil permettant de faire de la simulation de réseau ad hoc et par la suite reprendre le même code pour le faire tourner sur des appareils réels tel que les Nokia N800/N810. NLWCA et SLSF ont tous deux été codés dans JANE tandis que pour la partie Zeroconf, une version Java, JmDNS (http://jmdns.sourceforge.net/), a été adapté pour tourner sur JANE.

3.3.3 Installation sur les Nokia N800/N810

JANE étant codé en JAVA et de plus prévu pour tourner sur des appareils à petites capacités l'installation sur le Nokia ou autre appareil ne requiert qu'une seule chose : Java. Concernant les Nokias N800/N810, Sun n'a pas prévu de version officielle, il faut donc installer une version fournie par la communauté Maemo (communauté dédiée au Nokia Nxxx). L'installation sur le nokia est très simple et revient à l'installation d'un exemple montrant l'interface SWT sous maemo à savoir le « Jalimo_swt_example ». Pour plus d'informations : `http://wiki.forum.nokia.com/index.php/Getting_started_with_Java_on_maemo`.

Pour la suite les protocoles et JANE sont empaquetés dans des JAR. Pour le lancement, il faut copier tous les fichiers dans un répertoire créé pour l'occasion « `/javan800/` », celui-ci étant hard codé afin de retrouver les fichiers de configuration nécessaires.

Le lancement du programme doit se faire depuis le répertoire `/javan800/`. Une première étape de configuration consiste à choisir l'interface réseau en lançant le script suivant :

```
#!/bin/sh
# This file is autogenerated by the maven-pkg-plugin. The source
responsible
# for this script can be found in class:
de.tarent.maven.plugins.pkg.WrapperScriptGenerator

# You can provide additional VM arguments by setting the VMARGS
environment variable.
CLASSPATH_ARG="-cp /usr/share/java/swt-gtk-
3.4.M3.jar:/javan800/ALLJANE.jar:/javan800/nlwcawcpd.jar:/javan800/slsfjm
dns.jar:/javan800/"

MAIN_CLASS=platform.gui.NetworkAdapterChooser
```

[1] Tom Leclerc and Laurent Ciarletta and Andre Schaff, "Stable Linked Structure Flooding For Mobile Ad Hoc Networks", ISWPC 10
[2] A. Andronache and S. Rothkugel, "Nlwca node and link weighted clustering algorithm for backbone-assisted mobile ad hoc networks," Networking, 2008. ICN 2008. Seventh International Conference on, pp. 460-467, April 2008.

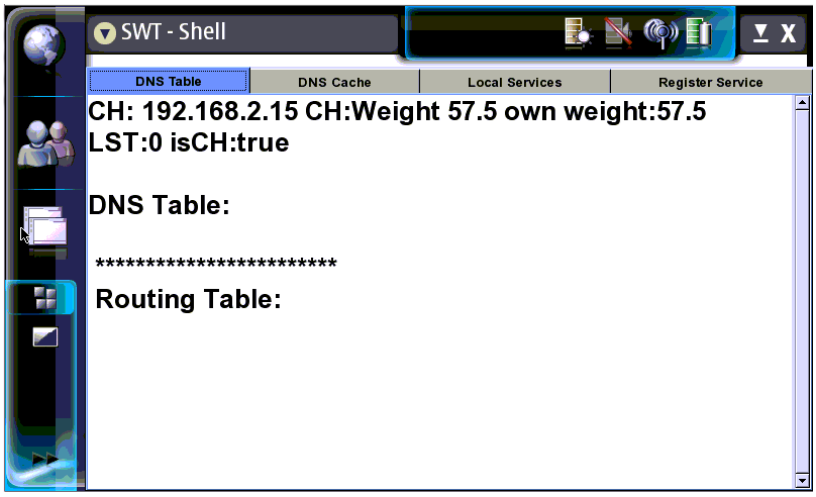
```
# Path of the shared libraries (e.g. for SWT
LIBRARY_PATH_ARG="-Djava.library.path=/usr/lib/jni"

# Path to BC-ABI compiled classes. Has no effect on runtimes other than
GCJ.
DB_PATH_ARG="-Dgnu.gcj.precompiled.db.path=/var/lib/gcj-4.1/classmap.db"

# Additional system properties which are special for this application:
SYSTEM_PROPERTIES=""

# Allows overriding the VM by setting the JAVA environment variable.
if [ x${JAVA} = x ];
then
                                        JAVA=/usr/bin/cacao
fi

echo ${JAVA} ${CLASSPATH_ARG} ${MAIN_CLASS}
exec ${JAVA} ${CLASSPATH_ARG} ${MAIN_CLASS}
```

L'information concernant le nom (DNS) et l'adresse IP utilisé est alors stockée dans le fichier `network_info.txt`. On peut aussi directement le modifier sans passé par l'interface graphique.

Ensuite vient le lancement de Zeroconf sur SLSF qui se fait par l'exécution du script suivant :

```
#!/bin/sh
# This file is autogenerated by the maven-pkg-plugin. The source
responsible
# for this script can be found in class:
de.tarent.maven.plugins.pkg.WrapperScriptGenerator

# You can provide additional VM arguments by setting the VMARGS
environment variable.
CLASSPATH_ARG="-cp /usr/share/java/swt-gtk-
3.4.M3.jar:/javan800/ALLJANE.jar:/javan800/slsfjmdns.jar:/javan800/"

MAIN_CLASS=platform.PlatformClient

# Path of the shared libraries (e.g. for SWT
LIBRARY_PATH_ARG="-Djava.library.path=/usr/lib/jni"

# Path to BC-ABI compiled classes. Has no effect on runtimes other than
GCJ.
DB_PATH_ARG="-Dgnu.gcj.precompiled.db.path=/var/lib/gcj-4.1/classmap.db"

# Additional system properties which are special for this application:
SYSTEM_PROPERTIES=""

# Allows overriding the VM by setting the JAVA environment variable.
if [ x${JAVA} = x ];
then
                                        JAVA=/usr/bin/cacao
fi

echo ${JAVA} ${CLASSPATH_ARG} ${MAIN_CLASS}
exec ${JAVA} ${CLASSPATH_ARG} ${MAIN_CLASS}
```

Selon la version d'OS du nokia il faudra remplacer « `/usr/share/java/swt-gtk-3.4.M3.jar:` » par « `/usr/share/java/swt.jar:` ». Le lancement se fait alors avec la commande `./nom_du_script` et on obtient l'interface suivante :



4. Conclusion

Ce document a permis de décrire les différentes manipulations d'installation et de mise en œuvre associées à chacun des composants utilisés pour les expérimentations respectivement dans les locaux d'Ucopia (composants logiciels d'Ucopia) ainsi qu'à la Cité des Télécommunications (composants logiciels d'Orange Labs R&D et de l'INRIA Madynes.)

5. Gestion de Document

Adresser toute remarque sur ce document au responsable de sa gestion.

Laurent Reynaud
Orange Labs R&D
Laurent.reynaud@orange-ftgroup.com