Titre: Title:	Collecte des données d'un réseau de capteurs sans fils en utilisant une surcouche réseau pair à pair
Auteur: Author:	David Rey
Date:	2010
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Rey, D. (2010). Collecte des données d'un réseau de capteurs sans fils en utilisant une surcouche réseau pair à pair [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/270/

UNIVERSITÉ DE MONTRÉAL

COLLECTE DES DONNÉES D'UN RÉSEAU DE CAPTEURS SANS FILS EN UTILISANT UNE SURCOUCHE RÉSEAU PAIR À PAIR

DAVID REY

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES

(GÉNIE INFORMATIQUE)

AVRIL 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

COLLECTE DES DONNÉES D'UN RÉSEAU DE CAPTEURS SANS FILS EN UTILISANT UNE SURCOUCHE RÉSEAU PAIR À PAIR

présenté par : REY David

en vue de l'obtention du diplôme de : Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de :

M. GALINIER Philippe, Doct., président

M. QUINTERO Alejandro, Doct., membre et directeur de recherche

M. GLITHO Roch, Ph.D., codirecteur de recherche

M. PIERRE Samuel, Ph.D., membre.

DÉDICACE

À Maël.

REMERCIEMENTS

Je tiens sincèrement à remercier mes directeurs de recherche, Messieurs Alejandro Quintero et Roch Glitho, pour leur encadrement, leurs précieux conseils ainsi que le support moral et financier qu'ils m'ont apportés tout au long de mes recherches. Ils ont toujours été patients et disponibles, et ont su me guider jusqu'à l'aboutissement de mon projet.

Je remercie également ma famille et mes amis qui m'ont encouragés et soutenu moralement pendant ma maîtrise.

Enfin je remercie les tous les membres du LARIM qui ont su apporter une bonne ambiance de travail dans un contexte de recherche universitaire.

RÉSUMÉ

Ces dernières années, on a vu exploser le nombre de dispositifs informatiques dans tous les domaines, et cela même chez les particuliers. De la voiture, au téléphone mobile, en passant par le pèse personne, l'informatique sous toutes ses formes et ses déclinaisons est omniprésente dans la vie courante, et ne cesse d'étendre encore son champ d'action.

Accompagnant cette expansion, les réseaux informatiques et plus particulièrement les réseaux sans fils sont eux aussi en pleine croissance. Il semble déjà lointain le temps où on ne téléphonait que de chez soi ou d'une cabine téléphonique, aujourd'hui tout le monde veut être connecté en tout temps. Et au-delà des personnes, maintenant la tendance est même à ce que tout dispositif soit relié au réseau, sans même aucune intervention de l'utilisateur.

À la croisée des chemins, on retrouve les réseaux de capteurs sans fils. Ces réseaux sont formés d'une multitude de petits dispositifs autonomes, capables de s'auto-organiser pour communiquer entre eux, et ainsi travailler de concert pour collecter des informations sur leur environnement, le tout sans intervention humaine. Ces dispositifs sont peu coûteux, mais peu performants, et peuvent être utilisés en masse sans prendre soin de s'occuper de chacun d'eux. Ils sont l'ultime signe actuel de l'informatisation de masse et du besoin absolu de réseautage.

Cependant dans une architecture traditionnelle actuelle, ces dispositifs ne sont pas totalement autonomes puisqu'ils nécessitent la présence de puits et de passerelles afin de pouvoir les utiliser avec le matériel informatique déjà présent.

Dans ce mémoire nous nous sommes attachés à présenter une architecture qui permette de s'affranchir de ces contraintes qui mettent à mal ce concept d'indépendance total des dispositifs, et qui plus est, portent préjudice aux performances globales des

réseaux de capteurs, en particulier dans certaines situations spécifiques. Pour cela nous avons proposé une architecture basée sur une sur couche pair à pair qui va nous permettre de collecter les données issues du réseau de capteurs. La collecte peut se faire soit grâce à un historique des valeurs récentes, qui de limiter la charge réseau en centralisant partiellement les données, soit par une connexion directe entre un utilisateur final et un capteur.

Nous avons aussi réalisé une implémentation partielle de cette architecture et avons fait quelques tests afin de démontrer la faisabilité de cette architecture. Si ces tests sont loin de fournir une conclusion définitive sur les performances de notre architecture, ils nous ont permis de conclure sur la faisabilité d'une telle architecture et l'absence de risque évident de défaillance généralisée.

ABSTRACT

Since few years we observed that number of computing devices increased quickly and found application in all domains and also among individuals. From car to cellphone, including bathroom scale, computing in all its forms and variations is ubiquitous in everyday life, and continues to expand its scope of action.

With this expansion, computer networks, especially wireless networks are also growing. It seems far the time where we have phone calls only from house or from phone booth, nowadays everybody wants to be connected at any time. And beyond people, now more and more devices are connected by themselves, without any user interaction.

At the crossroads, there are wireless sensor networks. These networks are composed of many small autonomous devices, which can self-organize them to communicate with each other and work together to capture information about their environment, without any human intervention. These devices are inexpensive, but inefficient, and can be deployed in mass without taking care of each one. They are the current ultimate stuff of mass computing deployment and absolute need of networks.

However in current traditional architecture, these devices are not really independent since they require the presence of sink and gateways in order to use them with present infrastructure.

In this paper we are committed to presenting an architecture that allows overcoming these constraints that undermine the concept of total independence of devices, and more, which can damage the overall performance of sensor networks, in particular in specific situations. For that we have proposed architecture based on a peer to payer overlay which will allow us to collect data from the sensor network. This collect can be done either through a history of recent values, which can limit the network

load by partially centralizing data, either through a direct connection between an end user and a sensor.

We also performed a partial implementation of this architecture and made some tests as a proof of concept of our architecture. If these tests cannot provide a definitive conclusion on the performance of our architecture, they have allowed us to conclude on the feasibility of such architecture and that there is not an obvious risk of widespread failure.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

3G	Réseau de troisième génération.
AD#	Agrégateur le dispositif # (*Aggregator on Device #*).
API	Interface de programmation (*Application Programming Interface*).
AWT	Bibliothèque graphique pour Java (*Abstract Window Toolkit*).
CDC	*Connected Device Configuration*.
CLDC	*Connected Limited Device Configuration*.
DHT	Table de hachage distribuée (*Distributed Hash Table*).
DNS	Système de noms de domaine (*Domain Name System*).
EUD#	Utilisateur final sur le dispositif # (*End-User on Device #*).
HKD#	Gestionnaire d'historique sur le dispositif # (*History Keeper on Device #*).
ID	Identifiant.
IMS	Sous système multimédia IP (*IP Multimedia Subsystem*).
IP	*Internet Protocol*.
Java EE	Java *Entreprise Edition*.
Java ME	Java *Micro Edition*.
Java SE	Java *Standard Edition*.
JSR	*Java Specification Requests*.
JVM	Machine virtuelle Java (*Java Virtual Machine*).

MANET	*Mobile Ad Hoc Network*.
NAT	Système de traduction d'adresse réseau (*Network Address Translation*).
NGN	Réseau de prochaine génération (*Next Generation Network*).
OS	Système d'exploitation (*Operating System*).
P2P	Pair à pair (*Peer to peer*).
RAM	Mémoire volatile à accès direct (*Random Access Memory*).
RC	Version stable d'un logiciel soumise à des « tests de dernière minute » (*Release Candidate*).
RISC	Type d'architecture matérielle de microprocesseur (*reduced instruction-set computer*).
RMI	*Remote method invocation*.
SD#	Capteur sur le dispositif # (*Sensor on Device #*).
SDRAM	Type de mémoire volatile (*Synchronous Dynamic Random Access Memory*).
SIP	*Session Initiation Protocol*.
TCP	Protocole de communication fiable par flux (*Transmission Control Protocol*).
UDP	Protocole de communication par datagrammes. (*User Datagram Protocol*).
URI	Identifiant unifié de ressource (*Uniform Resource Identifier*).
URL	Localisateur unifié de ressource ou « adresse universelle » (*Uniform Resource Locator*).

URN	Nom unifié de ressource (*Uniform Resource Name*).
WSN	Réseau de capteurs sans fils (*Wireless Sensor Network*).
XML	Langage extensible de balisage (*Extensible Markup Language*).

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION ET OBJECTIFS DE LA RECHERCHE

Depuis les débuts de l'informatique moderne, le besoin et l'envie récurrents de mettre en réseau les différents éléments informatisés que l'on peut avoir en sa possession se fait sentir. Cette mise en réseau peut se faire simplement pour échanger des messages (communication), pour échanger des données (stockage), ou encore pour partager la puissance de calcul. Les applications sont multiples et quasi illimitées. C'est dans cette optique qu'ont été déployés les premiers réseaux militaires, et universitaires, puis le plus grand d'entre eux : l'internet.

Internet, initialement déployé entre plusieurs universités américaines sous le nom d'ARPANET, n'est en fait qu'un réseau de réseaux. Internet ce n'est que l'interconnexion de plusieurs réseaux, soit à l'époque les modestes réseaux de ces universités. Cependant, notamment grâce à l'arrivée du *World Wide Web*, l'internet c'est ouvert au grand public, provoquant de réels bouleversements dans ses principes d'utilisation et dans les attentes qu'on a de lui, devenant ainsi un incontournable de la vie courante. On arrive d'ailleurs maintenant à ses limites, preuve en est le besoin de plus en plus pressant d'augmenter l'espace d'adressage (IPv6).

Aujourd'hui, accompagnée par la baisse massive des prix du matériel informatique, on observe une nouvelle évolution : les utilisateurs veulent pouvoir rester connectés en toutes circonstances et en tous lieux. Cette évolution est rendue possible grâce à la multiplication de nouveaux réseaux sans fil, comme le Wifi, les réseaux de troisième

génération (3G), le Bluetooth, le Wimax, Zigbee, etc. Tel un cercle vicieux, la multiplication de ces réseaux incite les usagers non seulement à rester eux-mêmes connectés en permanence, mais aussi à demander de plus en plus de capacités de communications à l'ensemble de leurs équipements. Nous ne nous étonnons plus maintenant de voir une console de jeu reliée à internet pour se mettre à jour (Wii, Xbox, PS3), un lapin capable de nous lire au réveil nos mails et les actualités récupérées sur l'internet (Nabaztag), ou encore un pèse-personne mettant directement en ligne notre poids sur Twitter ou par flux RSS, et permettant même d'afficher un historique en ligne ou depuis un iPhone (http://www.withings.com/).

Bref, tout doit être connecté tout le temps, personnes et accessoires. Cependant cela impose de nouvelles contraintes aux réseaux. Si le passage massif aux technologies sans-fil ces dernières années a permis de répondre à ces nouvelles attentes, ces technologies arrivent à leurs limites. En effet, elles requièrent encore l'installation préalable d'une architecture parfois lourde et complexe, notamment en cas de mobilité des nœuds dans le réseau. L'installation d'une telle architecture peut démultiplier les coûts d'installation et de maintenance. C'est dans cette optique que l'on a vu apparaître les réseaux ad hoc que nous allons vous présenter dans ce mémoire.

À la course à l'informatisation et au réseautage, si le grand public est devenu un acteur majeur, les entreprises et autres acteurs du marché ne sont pas en reste. Chacun dans son domaine respectif cherche à automatiser les tâches et informatiser la gestion des processus. C'est, pour le moment, dans ce domaine plus professionnel que nous allons retrouver les réseaux de capteurs que nous étudierons dans ce mémoire.

1.1 Les réseaux sans fils

De nos jours les technologies sans fils font parties de notre quotidien. Que ce soit des simples télécommandes infrarouge à l'internet sans fil en passant par les périphériques Bluetooth comme les téléphones portables, les souris, etc. aucune maison moderne n'échappe à la règle. On constate que de plus en plus de matériel électronique

devient communiquant, le plus souvent sans fils, et on ne voit pas ce qui pourrait freiner cette évolution.

Cependant, la majeure partie de ces technologies sans fils, à commencer par le Wifi, est basée sur des infrastructures fixes, limitant la mobilité des utilisateurs. Pour faciliter cette mobilité, il existe un autre type de réseau, de plus en plus courant, qui consiste à permettre aux nœuds du réseau de communiquer directement entre eux sans nécessiter d'infrastructure : ce sont les réseaux ad hoc.

On distingue donc deux principales classes de réseaux sans fils, les classiques structurés et les non structurés comme les réseaux ad hoc. Les réseaux ad hoc offrent la possibilité de connecter différents dispositifs sans avoir à préinstaller une infrastructure fixe comme dans les réseaux traditionnels.

1.1.1 Les réseaux sans fils classiques

La Figure 1.1 montre le fonctionnement classique des réseaux sans fils. Par exemple lorsque l'ordinateur de poche (PDA) veut communiquer avec le Tablet PC, les paquets envoyés par l'ordinateur portable sont acheminés au point d'accès 1, puis renvoyé au PDA. Pourtant le Tablet PC et le PDA pourraient communiquer directement puisqu'ils sont à portée l'un de l'autre. De même pour deux dispositifs qui ne sont pas à portée. Si l'ordinateur portable veut envoyer un message au Tablet PC, il va devoir contacter son point d'accès (soit le numéro 1), qui va faire suivre les paquets au point d'accès 2 via le réseau filaire, où ils seront enfin envoyés au Tablet PC. Dans tous les cas, tout le trafic doit passer par le point d'accès auquel est connecté le dispositif.

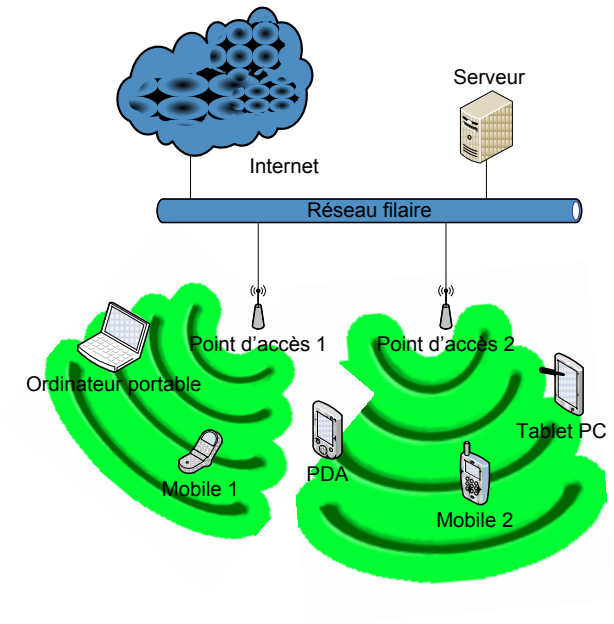

Figure 1.1 - Réseau sans fils traditionnel

De plus hormis l'envoi de paquets entre deux nœuds, les réseaux sans fils classiques nécessitent de nombreux autres services, par exemple pour gérer la mobilité des utilisateurs dans le réseau. Si une communication est en cours entre le mobile 1 et le mobile 2, que le mobile 1 se déplace et sort de la zone de couverture du point d'accès 1 (pour entrer dans la zone de couverture du point d'accès 2 par exemple), la communication sera coupée à moins de mettre en place une infrastructure spécifique nécessitant l'intervention d'un serveur.

Les réseaux ad hoc eux permettent d'établir des communications directement entre des nœuds du réseau et même d'assurer la mobilité des nœuds sans recourir à ces installations.

1.1.2 Les réseaux ad hoc

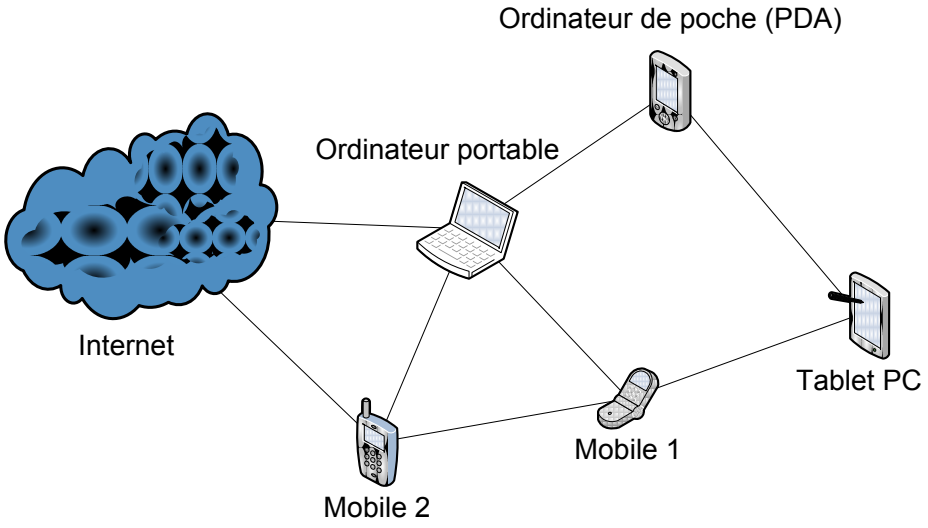

Figure 1.2 – Réseau ad hoc

Dans les réseaux ad hoc, l'ensemble des nœuds communiquent directement entre eux. C'est-à-dire que cette fois-ci, si l'ordinateur portable veut envoyer un message au Tablet PC, il ne va pas devoir contacter systématiquement un point d'accès, mais il va envoyer les paquets directement au destinataire. Pour cela, grâce à un protocole de routage, il va déterminer une route valide à un moment donné, puis envoyer ses paquets en suivant cette route. Dans notre cas une route valide serait : ordinateur portable, mobile 1, Tablet PC.

Bien sûr, dans ce cas, toute la difficulté va résider dans le fait de trouver une route valide. C'est dans ce but que de nombreux protocoles de routage ont été proposés. L'étude de ces protocoles reste d'ailleurs encore aujourd'hui un domaine de recherche très actif. Le but d'un tel protocole va être logiquement de découvrir une route valide et de la maintenir, mais aussi, de s'assurer de ne pas surcharger un nœud par rapport à un autre, ce qui causerait sa défaillance prématurée. La maintenance va inclure l'optimisation de la route en cas de mobilité des nœuds, la découverte d'une nouvelle route en cas de défaillance d'un nœud intermédiaire, etc.

Différentes solutions peuvent être adoptées pour la construction d'une route. Tout d'abord elle peut être construite à la volée, c'est-à-dire au moment d'envoyer un paquet [1], ou bien de manière pro active, c'est-à-dire en prévision d'un envoi possible. La première solution est plus basée sur une approche de découverte de nœuds tandis que la deuxième solution sera elle basée sur des tables recensant les nœuds du réseau. Les routes peuvent ensuite soit être établies à chaque paquet, c'est-à-dire que deux paquets d'une même communication peuvent emprunter deux routes totalement différentes, soit être la même pour toute la communication. D'autres solutions basées sur les coordonnées GPS (Global Positioning System) ont aussi été proposées.

1.2 Les réseaux de capteurs sans fils

Les réseaux de capteurs sont une application des réseaux ad hoc. Comme dit précédemment, de plus en plus de matériels électroniques sont maintenant communicants, et plus globalement on tend à tout automatiser, à commencer par les tâches répétitives, ingrates, dangereuses ou plus simplement ce que l'on a tendance à oublier. Ainsi on peut imaginer des stylos capables d'alerter un niveau d'encre faible et planifier automatiquement l'achat de nouvelles cartouches d'encre ; on peut imaginer automatiser le ramassage des déchets, leur tri et leur recyclage ; ou encore informatiser la surveillance des frontières. Dans tous ces cas on a besoin de nombreux capteurs, y compris pour l'automatisation des tâches, afin de s'assurer que tout se déroule correctement.

Cependant l'ajout et le câblage d'un trop grand nombre de capteurs peut couter très cher et donc rendre le projet irréalisable. Les réseaux de capteurs sans fils pallient ces difficultés grâce à l'utilisation de réseaux ad hoc.

1.2.1 Les capteurs

Les capteurs sont de simples petits modules électroniques à faible coût qui, grâce au principe d'auto-organisation des réseaux ad hoc, peuvent être déposés, voir même

largués depuis un avion, sans avoir à se soucier de leur connexion et sans avoir à déployer une infrastructure lourde.

Le principal problème de ces capteurs réside dans le fait que pour être produits et utilisés en grande quantité, ils doivent avoir un coût de production faible. Cela implique donc forcément des capacités réduites : capacité de calcul ; capacité de stockage ; portée radio.

Mais aussi, au-delà du problème de coût, le fait que les capteurs fonctionnent sur batterie, implique non seulement les mêmes restrictions sur la capacité, mais impose aussi de tout faire pour maximiser la durée de vie de la pile. Cela peut inclure de mettre en veille le capteur la majeure partie du temps, de limiter le trafic qui circule sur le réseau, etc.

1.2.2 Le réseau

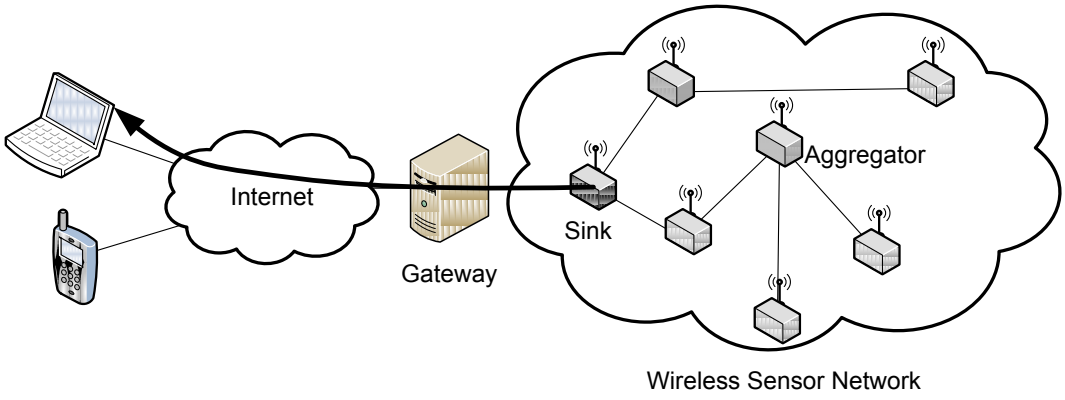

Figure 1.3 – Architecture classique d'un réseau de capteurs sans fils

La Figure 1.3 représente l'architecture habituelle des réseaux de capteurs sans fils. Ils sont construits autour des quatre principales entités suivantes :

- **Le capteur (sensor)**

  Comme le dit bien son nom, il est en charge de mesurer une valeur relative à son environnement (température, pression, luminosité, présence, etc.).

On peut parfois rencontrer des capteurs-actionneurs qui non seulement mesureront mais auront aussi pour rôle d'entreprendre une action en fonction de la valeur mesurée. L'intelligence nécessaire à la prise de décision quant à l'action à entreprendre peut alors être déportée sur un autre nœud du réseau.

- **L'agrégateur (aggregator)**

  Il est en charge d'agréger les messages qu'il reçoit de plusieurs capteurs puis de les envoyer en un seul message au puits (sink). Cette opération a pour principal but de limiter le trafic sur le réseau et donc de prolonger la durée de vie globale du réseau de capteur.

  Il correspond généralement à la tête d'une grappe (cluster head). L'utilisation de grappes offre de nombreux intérêts à tous les niveaux, notamment pour le routage.

- **Le puits (sink)**

  Le puits est le nœud final du réseau. C'est à lui qu'est envoyé l'ensemble des valeurs mesurées par le réseau. Il peut arriver qu'il y'ait plusieurs puits sur un même réseau de capteurs.

- **La passerelle (gateway)**

  La passerelle est un dispositif qui a la particularité d'avoir deux interfaces réseau. Il permet de relier le réseau de capteurs sans fils à un réseau plus traditionnel, typiquement l'internet.

  En effet, habituellement le réseau de capteurs ne sert qu'à faire remonter les mesures, les applications traitant ces informations étant exécutées sur la machine de l'utilisateur final.

Le fonctionnement global de cette architecture consiste donc à ce que les capteurs fassent des mesures qu'ils font remonter au puits via les agrégateurs. L'application finale tournant sur une machine se situant sur un autre réseau a ainsi accès aux valeurs via une passerelle.

À noter que les agrégateurs sont facultatifs, et que le puits et la passerelle sont généralement localisés dans un seul dispositif.

1.2.3 Problèmes et inconvénients

L'inconvénient majeur de cette architecture classique réside dans la faible fiabilité du puits et le goulot d'étranglement qu'il représente. En effet, une défaillance du puits provoque la mise hors service du réseau complet. Même si le puits est alimenté, le protégeant d'une défaillance due à une batterie vide, il provoque une surexploitation du réseau sur les nœuds alentours et donc fragilise le réseau.

L'utilisation de puits multiple corrige partiellement ce problème, mais n'offre qu'une fiabilité imparfaite, impose de planifier le positionnement de ces puits et pose la question de leur connexion à une passerelle. Une bonne solution serait l'utilisation de réseaux de capteurs sans avoir recours à un puits.

1.3 Scénarios d'utilisation

Si dans de nombreux cas les réseaux de capteurs sans fils traditionnels sont très efficaces, il peut exister des scénarios dans lesquels leurs performances sont faibles, voire des cas où ils sont tout simplement contre-productifs.

Nous allons donc présenter ici des scénarios où ces faiblesses se révèlent être de réels handicaps, motivation initiale à la proposition d'une nouvelle architecture sans puits pouvant répondre à ces différents cas.

1.3.1 Système de détection d'incendie

Un domaine où les réseaux de capteurs peuvent s'avérer très utiles, c'est pour la surveillance de bâtiments, par exemple pour la détection d'incendie. En effet, le câblage d'un bâtiment peut s'avérer être fastidieux et très onéreux. L'installation d'équipements

sans fils permet d'obtenir, à moindre coût, des systèmes plus complets et plus performants.

Supposons alors le scénario suivant : un bâtiment est protégé par un système de détection d'incendie constitué de capteurs situé dans chacune des pièces de l'immeuble. Le puits se trouve à l'entrée du bâtiment dans un local technique. Deux problèmes majeurs se posent à l'utilisation d'un réseau de capteurs traditionnel dans ce contexte :

- **Risque d'isolement de capteurs ou du puits dû au feu**
  Un des intérêts de l'utilisation d'un réseau de capteurs par rapport à un l'utilisation de simples détecteurs de fumée réside aussi dans le fait de pouvoir suivre en continu l'évolution du feu en cas d'incendie. Cependant, si le puits est mis hors service à cause du feu, c'est l'ensemble du système qui est mis hors d'usage.

- **Manque de réactivité du système :**
  Lorsqu'un pompier est dans le bâtiment pour lutter contre le feu, il veut connaître la valeur des capteurs qui sont proches de lui (derrière une porte qu'il veut ouvrir). L'utilisation d'un puits situé à l'autre bout du bâtiment et le passage par des réseaux différents, risquent de causer des ralentissements et de rendre l'utilisation peu efficace.

1.3.2 Système de prévention d'accident entre véhicules

Une autre application possible des réseaux de capteurs sans fils est pour un système de prévention d'accident entre véhicules. En effet, on peut imaginer intégrer à chaque voiture un capteur de vitesse, de telle sorte que l'ensemble des voitures sur une route forme un réseau de capteurs sans fils. Tous les nœuds seront donc en mouvement. Tout d'abord les capteurs les uns par rapport aux autres, mais aussi et surtout le réseau complet (comprendre l'ensemble des nœuds qui le forme) sera en mouvement par rapport au sol, et donc par rapport à des installations fixes. Dans ce cas, l'utilisation d'un

puits n'est absolument pas envisageable puisque la topologie du réseau est trop dynamique. Il en sera de même dans nombre de systèmes basés sur un réseau de capteurs en mouvement.

L'idée serait alors que chaque véhicule ait à bord un capteur mesurant la vitesse, laquelle serait partagée, permettant aux véhicules qui le suivent de connaître la vitesse des voitures les précédant. Ainsi, si la voiture détecte que les voitures qui la précèdent sont en forte décélération, l'ordinateur de bord pourra prendre la décision de freiner, évitant la collision.

L'avantage d'une telle solution par rapport à un système complexe comme on peut le voir sur certaines voitures futuristes, est la simplicité et le faible coût d'installation sur des voitures anciennes. Il est cependant certain qu'il n'est en réalité pas possible de déployer cela à grande échelle puisque des standards existent déjà quant aux communications entre les voitures, et de lourdes contraintes de sécurité sont à prendre en considération dans ce domaine. Cependant, cette solution serait envisageable et peut nous servir de base de réflexion, notamment pour ce qui a trait à la mobilité des nœuds.

1.4 Objectifs de la recherche

L'objectif principal de ce mémoire est de concevoir une architecture généraliste permettant de collecter les données issues d'un réseau de capteurs sans fils à l'aide d'une surcouche pair à pair, tout en respectant l'ensemble des contraintes fixées. De manière plus spécifique ce mémoire vise à :

- Analyser la problématique de l'architecture classique des réseaux de capteurs sans fils, ainsi que d'autres architectures déjà proposées dans le domaine, pour en extraire les contraintes auxquelles une nouvelle architecture devra se soumettre.
- Proposer une nouvelle architecture visant à pallier les inconvénients remarqués, et répondant aux contraintes que nous avons fixées.

- Rechercher des solutions pour rendre une implémentation possible sur des capteurs dont les performances sont réduites.
- Réaliser une implémentation de l'architecture de collecte des données afin non seulement de valider la faisabilité de l'architecture proposée, mais aussi pour tester sommairement les conditions d'utilisation de l'architecture.

CHAPITRE 2 REVUE DE LITTÉRATURE

2.1 Revue de littérature

Pour bien comprendre les tenants et les aboutissants des réseaux de capteurs sans fils, nous avons étudié la littérature portant sur ce type de réseau, et plus globalement sur les réseaux ad hoc et les réseaux sans fils. C'est ce que nous allons sommairement détailler ici. Nous n'aborderons pas l'ensemble du sujet, compte tenu de la quantité d'information disponible.

2.1.1 Le routage dans les réseaux ad hoc

L'un des plus gros problèmes des réseaux ad hoc réside dans le choix d'un protocole de routage efficace. En effet, comme ces réseaux sont sans infrastructure, il est impossible de faire du routage basé sur les adresses IP comme cela peut être fait dans des réseaux tels l'internet [2]. Il n'est pas possible non plus de planifier le positionnement des nœuds ou autre, tout doit se faire dynamiquement, y compris le routage.

Pour permettre à deux nœuds de communiquer entre eux, il faut donc réussir à trouver une route valide. Cette recherche de route peut être faite avant d'envoyer un paquet et maintenue à jour en prévision (*proactive*), ou bien juste au moment d'envoyer des informations (*on-demand*) comme [1], ou encore en envoyant les paquets qui seront acheminés automatiquement par les nœuds intermédiaires [3].

Pour le choix de la route, de nombreux critères de sélections peuvent être utilisés. Les plus courants sont :

- la qualité du service à offrir [4] (les critères de définition de la QoS étant eux aussi nombreux et variés) ;
- le coût associé à une route [5] (coût en termes de nombre de nœuds traversés, de temps de latence, etc.) ;
- le positionnement géographique du nœud [6] (grâce à l'utilisation des coordonnés GPS par exemple) ; …

Les réseaux de capteurs sans fils étant basés sur les réseaux ad hoc, on retrouve le même problème, avec cependant quelques contraintes différentes. En effet, on connaît généralement le type de données qui vont circuler, et dans une architecture traditionnelle des réseaux de capteurs, les données sont toutes renvoyées vers un puits. [2] donne un large aperçu de cette question, dont le Tableau 1.1 résume la complexité et le large choix de solutions possibles.

On constate sur ce tableau qu'une des solutions très prisées est la mise en grappes (ou cluster) avec agrégation des données. En effet, elle permet non seulement de simplifier le problème, mais aussi de réduire la charge réseau par l'agrégation. Une solution prisée aussi est par positionnement géographique. De manière général il est plus simple et moins couteux d'adopter une solution consistant à organiser les nœuds, plutôt que de chercher à optimiser seulement à partir des données.

Tableau 1.1 – Classification des protocoles de routages dans les réseaux de capteurs sans fils [2]

Routing protocol	Data-centric	Hierarchical	Location-based	QoS	Network-flow	Data aggregation
SPIN	✓					✓
Directed Diffusion	✓					✓
Rumor routing	✓					✓
Shah and Rabaey	✓		✓			
GBR	✓					✓
CADR	✓					
COUGAR	✓					✓
ACQUIRE	✓					
Fe et al.					✓	
LEACH		✓				✓
TEEN and APTEEN	✓	✓				✓
PEGASIS		✓				✓
Younis et al.		✓	✓			
Subramanian and Katz		✓				✓
MECN and SMECN			✓			
GAF		✓	✓			
GEAR			✓			
Chang and Tassiulas		✓			✓	
Kalpakis et al.			✓		✓	
Akkaya et al.		✓		✓		
SAR				✓		
SPEED			✓	✓		

2.1.2 L'utilisation de TCP dans les MANET

Un autre problème majeur des réseaux ad hoc et donc des réseaux de capteurs sans fil, c'est la très faible performance de TCP.

Sans entrer dans les détails, le problème est que TCP utilise une fenêtre de transmission pour maximiser la vitesse de transfert. Cependant, lorsqu'il détecte une congestion du réseau, il réduit automatiquement cette fenêtre. L'inconvénient est que la fiabilité des routes sur les réseaux ad hoc est faible, ce qui fait que TCP détecte une congestion alors qu'il s'agit en fait d'une rupture d'une route. Il va donc réduire la fenêtre, alors qu'il devrait simplement attendre l'ouverture d'une nouvelle route.

En effet, TCP a été conçu pour fournir une communication fiable entre deux nœuds et le plus souvent déployé sur des systèmes filaires. Mais ce déploiement ne prend pas en compte les spécificités des réseaux ad hoc, en particulier les problèmes de confusion entre congestion et défaillances d'une route, ou les déconnexions fréquentes

des nœuds accompagnées de rupture de routes [7]. Il en résulte des performances souvent mauvaises sur les réseaux ad hoc.

De nombreuses solutions ont été proposées pour palier à ce problème. Ainsi [8] expose un certain nombre de ces solutions, dont le Tableau 1.2 présente une comparaison des solutions vis-à-vis des différents problèmes que rencontre TCP sur les réseaux ad hoc. Une des difficultés consiste à proposer une solution qui ne remette pas en cause les réseaux actuels, c'est pour cette raison qu'on peut voir que des solutions sont proposées à tous les niveaux du modèle OSI. On peut d'ailleurs remarquer qu'un certain nombre des solutions proposées est basée sur le « cross layer », technologie que nous allons présenter au point suivant.

Tableau 1.2 – Comparaison générale des différentes solutions proposées pour l'utilisation de TCP sur les MANET [8]

Problem	Proposal	Solution type	Complexity degree	Network type	TCP connections load/type	Evaluation
TCP is unable to distinguish between losses due to route failures and congestion	ELFN	TCP-net. cross layer	Medium	Random mobile	One persistent	Simulation no routing
	ATCP	TCP-net. cross layer	High	Random mobile	One persistent	Experimental no routing
	TCP-BuS	TCP-net. cross layer	High	Random mobile	Multiple peristent	Simulation no routing
	TCP-F	TCP-net. cross layer	Medium	Random mobile	One persistent	Emulation no routing
	TCP-DOOR	TCP layer	Low	Random mobile	One persistent	Simulation
	TCP-RTO	TCP layer	Low	Random mobile	One persistent	Simulation
Frequent route failures	Split TCP	TCP-net. cross layer	Medium	Random mobile	One persistent	Simulation
	Preemptive routing	Network-physical cross layer	Medium	Random mobile	No TCP load	Simulation
	Signal strengh based link	Network-physical cross layer	High	Random mobile	Two persistent	Simulation
	Backup routing	Network layer	Low	Random mobile	One persistent	Simulation

Problem	Proposal	Solution type	Complexity degree	Network type	TCP connections load/type	Evaluation
Contention on wireless channel	Dynamic delayed ACK	TCP layer	Low	Static chain	One persistent/short	Simulation
	COPAS	Network layer	Medium	Static random	Multiple persistent	Simulation
	LRED/adaptative pacing	Link layer	Low	Static	Multiple persistent	Simulation
TCP unfairness	Non work conserving	Link layer	Low	Static chain	Multiple persistent	Simulation
	Neighborhood RED	Link layer	Low	Static / mobile	Multiple persistent	Simulation

2.1.3 Cross-layer

Le *cross-layer* est une technique très utilisée pour résoudre les problèmes des réseaux ad hoc. Elle consiste à faire circuler des informations entre les différentes couches du modèle OSI.

Comme le montre la Figure 1.4, l'idée de base est de permettre aux couches supérieures d'obtenir des informations de niveau matériel pour améliorer leur comportement, ou inversement de diffuser de l'information des couches supérieures aux couches inférieures. Par exemple pour la gestion du TCP, cela peut permettre de détecter une défaillance de la route plutôt que de croire à une congestion [9]. Cela pourrait aussi aider à mieux gérer la batterie par exemple, en optimisant les interfaces radios.

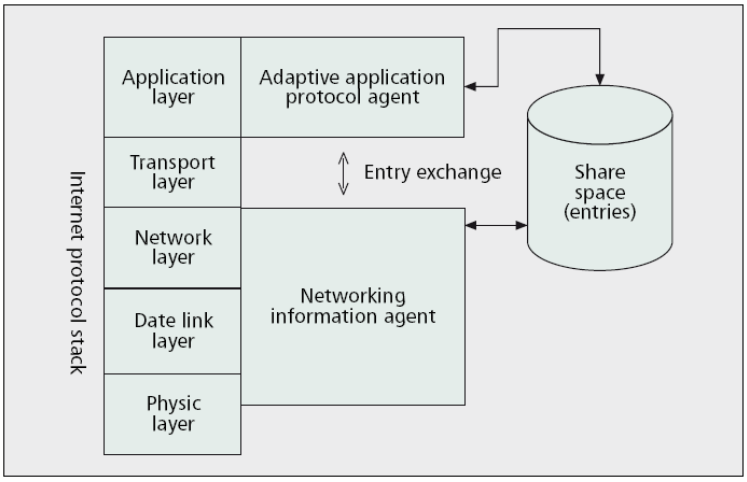

Figure 1.4 – Exemple d'architecture cross-layer proposée par [10]

2.2 Travaux similaires

On ne trouve dans la littérature qu'assez peu de solutions conçues comme nous l'envisageons, à savoir sans recourir à un puits (SinkLess). Les principales sont les suivantes :

2.2.1 TinyLIME

Tout d'abord une solution intéressante, la plus proche de notre conception, est LIME et son extension TinyLIME. Cette solution décrite dans [11] offre un système complet d'accès aux données sans puits via des nœuds mobiles qui évoluent entre les capteurs.

Dans cette solution les capteurs ne sont accessibles que par les dispositifs à portée. C'est-à-dire que si on suppose une série de capteurs répartis sur une zone géographique, seuls les capteurs à proximité de dispositifs plus puissants, tels que des ordinateurs portables, seront accessibles. Une surcouche réseau est ensuite proposée entre les dispositifs les plus puissants, pour permettre d'accéder aux capteurs à portée des autres dispositifs.

Un inconvénient majeur de cette solution est qu'elle force à n'utiliser qu'un seul intergiciel (*middleware*), celui de TinyLIME. Pourtant on trouve dans la littérature de nombreux intergiciels pair à pair, chacun ayant des avantages et des inconvénients par rapport à un autre, ou en fonction du type de réseau sous-jacent. Nous souhaitons proposer une solution plus générale, qui soit indépendante de l'intergiciel choisi.

2.2.2 Architecture basés sur les services

D'autres solutions basées sur des services ont été proposées et décrites dans [12] et [13]. Il s'agit dans ces cas d'utiliser des capteurs comme des fournisseurs de services auxquels l'utilisateur final souscrit. Cette approche peut-être particulièrement intéressante pour une intégration des réseaux de capteurs sans fils dans le sous-système multimédia IP (*IP Multimedia Subsystem – IMS*) des réseaux de prochaine génération (*Next Generation Network – NGN*), puisque l'ensemble d'IMS est orienté sur les services.

Dans cette topologie, lorsque qu'un utilisateur souhaite communiquer avec un capteur, il doit rechercher, sur un serveur, l'adresse d'un fournisseur de services (un capteur) qu'il va ensuite interroger.

Cependant étant donné que cette architecture nécessite l'utilisation d'un serveur de mise en relation des fournisseurs et des utilisateurs de services, elle est encore tributaire d'une infrastructure et on revient sur le même genre de problématique qu'avec le puits.

2.2.3 Puits multiples

Il existe aussi des solutions proposant une architecture à puits multiples. Cette solution classique va simplement consister à mettre plusieurs puits dans un seul réseau de capteurs sans fils.

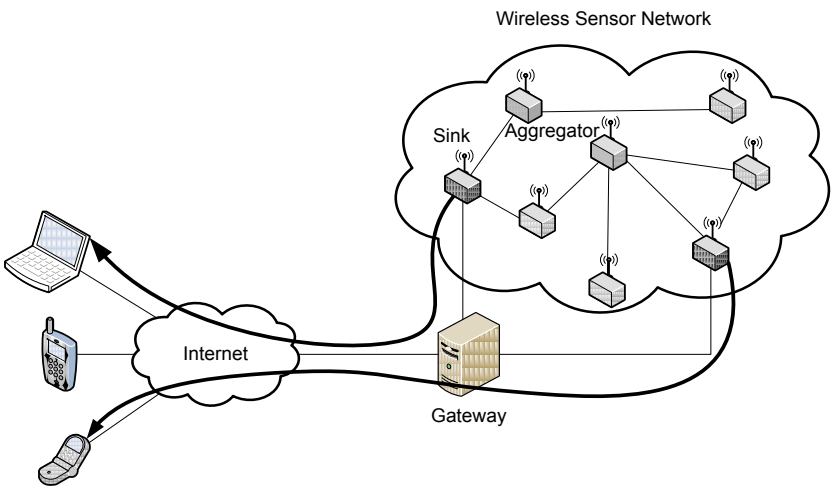

Figure 1.5- Réseau de capteurs sans fils à puits multiples

Dans cette topologie, les données sont transmises aux utilisateurs finaux en utilisant l'un ou l'autre des puits du réseau. Le choix du puits est délégué au protocole de routage du réseau de capteurs.

Un gros problème de cette solution réside dans la difficulté du routage des valeurs mesurées. On peut d'ailleurs percevoir la complexité d'une telle solution dans l'article [14], qui propose une solution d'optimisation du routage dans une architecture du type plusieurs-à-plusieurs comme celle-là.

Cette proposition souffre aussi d'un problème évident de manque d'évolutivité. En effet si le réseau de capteurs grossit, afin de ne pas retomber dans les mêmes problèmes qu'une architecture à puits unique, il va falloir rajouter des puits, ce qui ne peut pas se faire automatiquement. De plus, cette solution ne permet toujours pas une connexion directe entre l'utilisateur final et le capteur.

2.3 Intergiciels pair à pair

Dans la littérature, on trouve de nombreux intergiciels pair à pair. Il est hors de propos de tenter d'en faire une liste exhaustive. Ils se différencient principalement selon

leur but et leur architecture. Parmi les plus connus on notera Kadmelia [15], Gnutella [16], BitTorrent, ou encore Jxta [17].

En effet, si dans la vie courante le terme de pair à pair fait avant tout référence aux échanges de fichiers entre utilisateurs, les intergiciels pair à pair ne se limitent heureusement pas à ça. Ils permettent d'échanger d'autres ressources comme par exemple de la puissance de calcul. Ils peuvent servir aussi pour la propagation de flux audio / vidéo, ou encore être utilisés pour la création de base de données distribuées.

Mais au-delà de cette différenciation par le but de l'intergiciel, on distingue deux principales architectures de réseau pair à pair, que nous allons décrire ci-dessous : les réseaux structurés et les réseaux non structurés.

2.3.1 Les réseaux pair à pair structurés

Les réseaux pairs à pairs structurés se distinguent des réseaux non structurés par la nécessité de serveurs. Ces serveurs serviront principalement à mettre en relation les différents pairs entre eux.

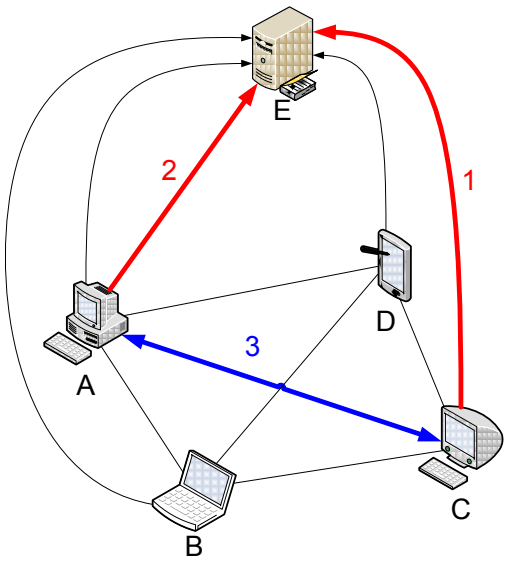

Figure 1.6 – Réseau P2P structuré

Sur la Figure 1.6, pour que les échanges puissent se faire entre les nœuds A et C, C commence par indiquer au serveur E ce qu'il met à disposition des autres nœuds (1), ensuite A demande au serveur qui à l'information dont il a besoin (2), et enfin les échanges se font de pair à pair directement entre A et C (3).

Si cette architecture est clairement la plus simple à mettre en œuvre, elle présente l'inconvénient évident de toujours être dépendante d'un serveur, avec tout ce que cela implique notamment au niveau de la tolérance aux fautes.

2.3.2 Les réseaux pair à pair non structurés, ou totalement décentralisés

Les réseaux pairs à pairs totalement décentralisés se différencient par rapport aux réseaux structurés de par leur capacité à s'affranchir totalement d'un quelconque serveur centralisé.

Dans les protocoles de partage de fichier, pour arriver à cet affranchissement total, on utilise généralement des tables de hachages distribuées DHT (*Distributed Hash Table*). Le *hash* d'un fichier, ou empreinte, c'est une suite de caractères qui caractérise de manière quasi unique un fichier. Les fonctions de hachage, très utilisées en cryptographie pour s'assurer que des données n'ont pas été altérées, sont conçues de telle manière que deux sources proches l'une de l'autre donnent des empreintes totalement différentes. Dans ces conditions la probabilité d'avoir deux fichiers ayant la même empreinte est infime, et on peut donc admettre, pour un système de partage de fichier, que ce risque est nul. Des fonctions de hachage bien connues sont SHA, MD5 et Whirlpool.

On définit ainsi une série de clefs indexables et stockables dans des bases de données. Ces bases de données sont alors distribuées entre les pairs du réseau, généralement avec redondance, prévenant ainsi tout problème lié à la défaillance d'un nœud. Si on ajoute par-dessus cela un protocole pair à pair spécifique permettant

l'interrogation de cette base de données, on arrive à une architecture totalement décentralisée.

Cependant ce modèle n'est souvent que très théorique, puisque pour être parfait il faudrait s'affranchir de tout système centralisé, y compris les serveurs DNS. À noter que pour pouvoir joindre un réseau P2P de ce type, on doit forcément avoir un point d'entrée. Ce point d'entrée ne doit donc par exemple pas être basé sur un nom de domaine.

2.4 Jxta

Parmi tous les intergiciels pair à pair disponibles, nous avons choisi d'utiliser Jxta (prononcer « juxta », comme *juxtapose*) pour notre implémentation et nos essais. Notre choix a principalement été motivé par la fiabilité reconnue de Jxta et sa simplicité de mise en œuvre. Il a par exemple déjà été utilisé avec succès dans différents projets comme [18] . De plus, comme nous le verrons au point 2.6, Jxta a été développé dans une version spécialement conçue pour les appareils mobiles.

Une introduction plus complète à Jxta a été jointe à ce mémoire en Annexe 1.

2.4.1 Présentation de Jxta

Jxta est un projet open source développée par Sun, qui définit un certain nombre d'APIs permettant de facilement construire un réseau pair à pair. Ces APIs étant disponibles en Java, C++ et C#, les différentes versions sont bien entendu compatibles entre elles.

À noter que Jxta est basé sur les standards du web comme TCP/IP, HTTP, et XML, et offre l'énorme d'avantage d'être conçu pour passer outre les restrictions qu'on peut retrouver sur de nombreux réseaux. Il permet notamment de passer outre les pare-feu (firewall) et les NAT (*Network Address Translation* – dispositif permettant de partager une seule adresse IP avec plusieurs machines).

Jxta est composé d'une suite de six protocoles, certains fondamentaux et d'autres facultatifs, qui vont travailler conjointement à :

- permettre aux pairs de se découvrir entre eux ;
- auto-organiser les pairs en groupes ;
- annoncer et découvrir des ressources ;
- permettre les communications entre les pairs.

2.4.2 Les éléments de Jxta

Pour utiliser efficacement Jxta nous avons accès à un certain nombre d'éléments inclus dans l'intergiciel dont voici les principaux avec la terminologie issue de Jxta :

- **ID :** Les ID sont construits selon le principe des URN, ils servent à tout identifier dans le réseau (pairs, pipes, etc.)
- **Pairs (*Peers*) :** C'est un nœud du réseau P2P possédant un ID. Il existe des pairs particuliers appelés *superPeer* qui ont des fonctionnalités avancées, comme les relais ou les rendez-vous.
- **Groupes de pairs :** Un groupe de pairs représente un regroupement de plusieurs pairs qui offrent un même service. Un pair peut faire partie de plusieurs groupes à la fois.
- **Annonces (*Advertisements*) :** Les annonces sont au cœur du fonctionnement de Jxta. Elles permettent à un nœud de s'annoncer à ses pairs, mais aussi d'annoncer des ressources comme un groupe, un service ou encore un canal de communication.

- **Canaux de communications (*Pipe*) :** Ils permettent la communication entre deux pairs. Ils peuvent être unicast ou multicast et sont accessibles entre autre via des APIs implémentant l'interface des sockets.

2.5 Machine virtuelle Java

La machine virtuelle Java présente l'intérêt majeur d'être multiplateforme, c'est-à-dire qu'elle peut être installée sur différents systèmes d'exploitation (OS). On peut donc exécuter le même code java quel que soit le type de dispositif. C'est un atout essentiel dans le cas d'un réseau de capteurs sans fils où on peut facilement arriver à avoir de nombreux types de matériels. Un autre intérêt est que de nombreux travaux sont disponibles sur Java, y compris des intergiciels pair à pair, parfois encore expérimentaux.

Il existe différents types de machines virtuelles en fonction du type de dispositif sur lesquelles elles se trouvent. L'intérêt de multiplier les machines virtuelles java (JVM) est de mieux les faire coller aux performances du dispositif.

Cela ne remet que très peu en cause la portabilité du code, puisqu'une application développée pour un type de machine virtuelle restera valide pour tous les dispositifs utilisant cette machine. De plus, dépendamment du type de machine virtuelle et des librairies utilisées, le code pourra être totalement valide sur plusieurs machines virtuelles différentes.

Les trois principaux types de machines virtuelles sont Java EE (*Java Entreprise Edition*), Java SE (*Java Standard Edition*), et Java ME (*Java Micro Edition*). Il en existe d'autres dont les spécifications peuvent être trouvées sur [19].

2.5.1 Java SE

Cette JVM est définie pour être utilisée sur des postes de travail classiques. Elle contient toutes les libraires de base ainsi que les librairies spécifiques à un poste de

travail, par exemple les librairies d'interface graphique comme Swing et AWT, des parseurs XML, ou encore RMI [19].

2.5.2 Java EE

Java Enterprise Edition est une JVM destinée aux entreprises et aux serveurs. Elle reprend l'ensemble des APIs de J2SE auxquelles elle ajoute des librairies spécifiques, notamment des APIs pour le réseau, pour des applications réparties ou encore pour la tolérance aux fautes [19].

2.5.3 Java ME

C'est cette version de la plateforme Java qui va nous intéresser pour notre architecture. En effet c'est ce JRE qui est destiné aux dispositifs mobiles qui ont peu de ressources. On la retrouve par exemple dans de nombreux téléphones ou ordinateurs de poche.

Java ME est une machine virtuelle capable d'exécuter une application java, les APIs de base regroupées en configuration et des APIs spécialisées regroupée en profils. Il existe plusieurs configurations standardisées, les plus courantes sont :

- **CLDC** (*Connected Limited Device Configuration*)
  Cette configuration concerne des appareils ayant de très faibles performances. Elle est le plus souvent associée au profil MIDP.

- **CDC** (*Connected Device Configuration*)
  Cette configuration concerne des dispositifs ayant des capacités légèrement plus élevées. Elle est le plus souvent associée au profil « *foundation* ».

2.6 Jxme, une implémentation de Jxta pour l'environnement mobile Java ME

Un problème de Jxta, comme tout intergiciel pair à pair, est qu'il nécessite que les dispositifs aient des ressources relativement importantes, ce qui n'est pas forcément compatible avec des dispositifs Java ME. On peut ainsi relever plusieurs éléments principaux rendant impossible l'utilisation de Jxta sur des appareils ayant de faibles ressources :

- **La mise en cache des *advertisments* :**
  Comme dit au point 1.3.4, l'intergiciel Jxta est basé sur des annonces (*advertisments*) permettant d'annoncer des nœuds, des groupes de nœuds, des canaux de communications, des services, etc. Ils sont omniprésents dans Jxta et sont mis en cache par tous les nœuds du réseau. Malheureusement cette mise en cache peut se relever trop gourmande en espace de stockage pour les dispositifs utilisant Java ME.

- **L'utilisation massive du XML :**
  Les *advertisments* sont en fait des messages XML. Si l'efficacité de XML n'est plus à prouver, il n'en reste pas moins que les parseurs XML sont généralement très gourmands en puissance de calcul. Si aujourd'hui on trouve des parseurs XML utilisables sur des dispositifs Java ME comme NanoXML [20], ils ne sont pas nécessairement compatibles avec tous les matériels, notamment ceux basés sur CLDC.

- **L'utilisation de sockets dans Jxta :**
  Jxta utilise des sockets pour communiquer avec les autres nœuds du réseau. Or les dispositifs utilisant la configuration CLDC ne supportent pas de sockets, il nous est donc impossible de faire tourner Jxta sur ce type de dispositif.

Partant de ce constat, l'équipe de développement de Jxta a décidé de développer une version de Jxta qui soit utilisable sur les dispositifs Java ME : JXME [21].

2.6.1 Jxme avec relai (CLDC)

Dans un premier temps, pour porter Jxta sur les dispositifs Java ME la décision a été prise d'utiliser un relai. Le relai étant chargé de faire l'intermédiaire entre les dispositifs Java ME et le réseau classique Jxta.

Dans cette architecture le relai sert en fait à prendre en charge toutes les fonctionnalités de base qu'on attend de tout nœud Jxta mais qu'un appareil Java ME n'arrivera pas forcément à gérer. Le relai n'est qu'un simple nœud Jxta fonctionnant sur Java SE (JXTA-J2SE).

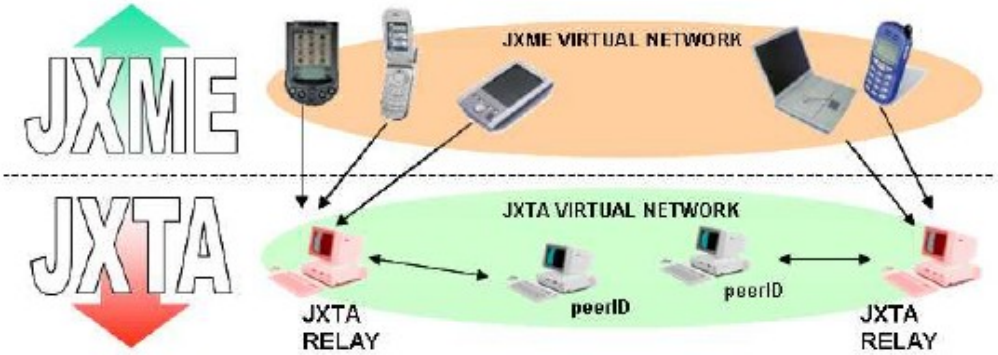

Figure 1.7 – Principe de fonctionnement de JXME avec relais [22].

Le problème est que ce principe d'utilisation de Jxme est sans intérêt dans notre architecture. En effet l'utilisation de relais extérieurs reviendrait à introduire un puits dans le réseau de capteurs sans fil.

2.6.2 Jxme sans relai (CDC)

Dans un second temps l'équipe de développeurs de Jxta s'est attachée à offrir la possibilité d'utiliser Jxme sans avoir recours à un nœud relai. C'est ce qui a été fait avec « *Jxme-proxyless* ».

Jxme sans relai, c'est l'implémentation complète de l'intergiciel Jxta, mais en n'utilisant que des librairies accessibles depuis un dispositif Java ME utilisant la configuration CDC.

Ainsi les nœuds Jxme sans relai sont des nœuds à part entière dans le réseau Jxta et peuvent former un réseau entier sans avoir besoin d'un seul nœud JXTA-J2SE. C'est donc logiquement cette implémentation de Jxta que nous avons utilisée pour implémenter notre architecture.

2.7 Interface SIP

Étant donné que nous avons décidé d'utiliser des messages SIP pour faire passer un certain nombre d'informations entre les nœuds, nous avons cherché une librairie permettant de facilement gérer des transactions.

Le JSR 180 [23], aussi appelé API SIP, définit un ensemble d'interfaces à implémenter pour fournir des services SIP avec des dispositifs Java ME, CDC comme CLDC. C'est cette API qui est implémentée sur de nombreux appareils, notamment les téléphones. Son fonctionnement est décrit à la Figure 1.8. Elle est basée sur 3 entités : les `SipClientConnection` et `SipServerConnection` qui représentent les connexions à proprement parler, et le `SipConnectionNotifier` qui permet d'écouter les connexions entrantes et qui notifie le serveur d'une nouvelle connexion de manière asynchrone.

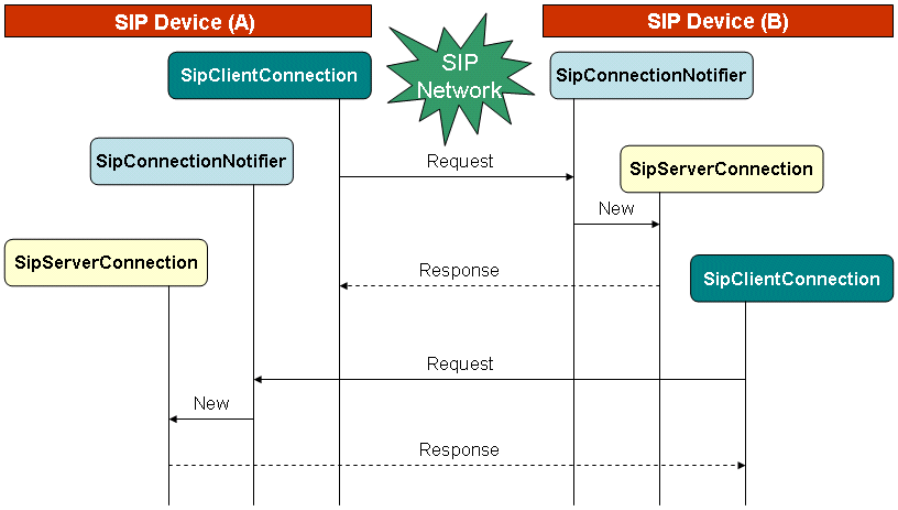

Figure 1.8 – Interaction entre les interfaces de l'API SIP [24]

Cependant, nous n'avons trouvé qu'une seule implémentation du JSR 180, celle de Nokia disponible sur le forum Nokia [25]. Malheureusement, cette implémentation ne fonctionne qu'avec des nœuds Java ME CLDC, ce qui ne convient pas pour notre utilisation.

D'autres solutions comme MjSip [26] fournissent les méthodes nécessaires pour faire des dialogues SIP, et est compatible avec CLDC et CDC. Cependant, elles ne respectent pas le JSR 180, n'offrant donc pas de réelle portabilité. De plus, cette solution nous aurait obligés à modifier profondément notre code, nous avons donc pris le parti d'implémenter partiellement l'API SIP dont les grandes lignes peuvent être retrouvées dans l'Annexe 2.

2.8 Interactions avec l'intergiciel

Comme précisé dans les exigences, nous avons cherché à faire une architecture indépendante de l'intergiciel. Cependant, nous avons recours à l'utilisation de services offerts par l'intergiciel. Ces services sont communs et fournis par tous les intergiciels pair-à-pair.

Nous allons utiliser les annonces de l'intergiciel et leur mise en cache, pour former l'historique réparti entre les gestionnaires d'historique. Nous allons aussi nous servir des sockets multicast des intergiciels comme base de communication pour les quatre groupes définis au point 3.2.4.1.

CHAPITRE 3 PROPOSITION D'UNE NOUVELLE ARCHITECTURE

Afin de pouvoir proposer une nouvelle architecture sans puits pour l'accès aux données d'un réseau de capteurs, nous avons besoin de déterminer avec précision quelles sont les exigences auxquelles elle devra répondre, et quelles sont les hypothèses de travail que nous allons préalablement poser. Pour cela nous allons partir des scénarios d'utilisations présentés au chapitre précédent pour en extraire les exigences qu'ils imposent à notre architecture. Dans une seconde partie nous allons décrire une nouvelle approche permettant d'accéder aux données d'un réseau de capteurs sans fils sans avoir recours à un puits ou une passerelle en utilisant une surcouche réseau pair à pair, ce qui permettra une connexion directe entre l'application et un capteur.

3.1 Hypothèses et exigences

Afin de déterminer le travail que nous avons à faire, il est important de fixer le cadre dans lequel nous allons évoluer. Nous avons donc posé certaines hypothèses de travail.

3.1.1 Constat

Contrairement aux réseaux de capteurs sans fils traditionnels, dans notre architecture nous souhaitons que les dispositifs des utilisateurs finaux puissent se

connecter directement aux capteurs. Dans cette optique nous avons défini quatre rôles différents :

- **Rôle de capteur :** qui définit la capacité à faire des mesures ;
- **Rôle d'agrégateur :** qui définit la capacité à agréger du contenu ;
- **Rôle de gestionnaire d'historique :** qui définit la capacité à sauvegarder des valeurs ;
- **L'application de l'utilisateur final :** qui caractérise l'application qui a la capacité d'utiliser les valeurs captées.

Par rapport à une architecture classique, seul le gestionnaire d'historique est ajouté, les autres jouant des rôles similaires à ce qu'on peut voir habituellement. Une description complète de ces rôles et faites au point 3.2.1. Ces rôles seront implémentés par deux types de dispositifs :

- **Dispositif dit capteur / agrégateur :** cela correspond aux capteurs physiques qu'on trouve sur le marché. Ils vont jouer au choix un ou plusieurs rôles parmi ceux de capteurs, d'agrégateur, ou de gestionnaire d'historique.
- **Dispositif de l'utilisateur final :** c'est le dispositif sur lequel l'application de l'utilisateur final tourne. Il pourra aussi éventuellement implémenter le rôle de gestionnaire d'historique.

3.1.2 Hypothèses de travail

À partir de ces considérations, et compte tenu de nos objectifs, nous allons supposer différentes hypothèses.

Tout d'abord, nous supposerons que tous les dispositifs ont la même pile de protocole. En effet, compte tenu du caractère totalement décentralisé de notre

architecture et des faibles ressources des nœuds, il n'est pas acceptable d'utiliser certains nœuds comme relais de communication.

À la vue des scénarios présentés précédemment, nous partirons aussi du principe que chacun des nœuds peut être soit fixe soit mobile. En effet, dans le scénario du système de prévention des accidents tous les nœuds sont mobiles, tandis que dans le scénario du système de prévention d'incendie, seuls les dispositifs des utilisateurs finaux sont mobiles. Il est donc important de couvrir le maximum de cas, c'est pourquoi nous devons concevoir notre architecture en prenant en compte toutes les possibilités de mobilité ou non des nœuds.

Enfin, nous allons supposer que l'ensemble des nœuds du réseau sont adressables. C'est-à-dire qu'on pourra établir une communication directe entre deux nœuds. C'est un élément très important dans l'architecture décentralisée que nous voulons mettre en place, puisque cela va nous permettre d'assurer la fiabilité du réseau quel que soit l'état de celui-ci.

3.1.3 Exigences

À partir des hypothèses que nous avons formulés au point précédent, et au regard des scénarios que nous avons présentés au point 1.3, nous allons ici définir une série d'exigences auxquelles notre architecture devra répondre.

Tout d'abord la plus importante, nous souhaitons que notre architecture soit indépendante de l'intergiciel. En effet de nombreux intergiciels pair à pair sont disponibles, comme par exemple, parmi les plus connus Kadmelia [15] et Gnutella [16]. Chaque intergiciel apportant son lot d'avantages et d'inconvénients, certains étant plus performants sur certains types de réseaux, ou dans un domaine spécifique, d'autres nécessitant moins de ressources par exemple. Il est donc important de définir une architecture indépendante de l'intergiciel, cela permettra d'offrir une solution plus

généraliste et laissera libre de choisir au mieux l'intergiciel le mieux adapté aux circonstances.

D'un point de vue plus général, nous devrons assurer la fiabilité du réseau. Nous devrons notamment pouvoir ajouter ou retrier des nœuds du réseau à tout moment. Nous devrons aussi assurer une optimisation de la charge sur le réseau de manière à limiter la consommation des ressources et par conséquent prolonger la durée de vie du réseau. Enfin, dans la même optique que l'indépendance de l'intergiciel, nous nous attacherons à offrir une architecture la plus générale et évolutive possible afin qu'elle puisse s'adapter à un maximum de cas.

Dans le cas du système de détection d'incendie, par exemple, il pourrait être utile de conserver quelques valeurs passées des capteurs, permettant ainsi de remonter facilement à la source du feu. De manière plus pratique, stocker dans une sorte de cache les dernières valeurs, pourrait également permettre de limiter l'utilisation des ressources des capteurs. Nous allons donc chercher un moyen de fournir un petit historique des valeurs récentes. Nous devrons donc aussi nous assurer d'une relative fiabilité de cet historique.

Enfin, nous avons une exigence plus spécifique à certains nœuds. Les dispositifs des utilisateurs finaux doivent pouvoir communiquer directement avec un capteur / agrégateur, y compris, et surtout, si ce capteur / agrégateur est isolé du réseau. En effet, pour le système de détection d'incendie et en cas d'incendie déclaré, il se peut qu'un certain nombre de capteurs / agrégateurs soient mis hors d'usage. Pour autant quelques-uns peuvent être encore actifs, mais isolés. Une communication directe permettra de quand même tirer profit de ces capteurs / agrégateurs.

3.2 Présentation de l'architecture

Dans cette partie nous allons présenter une architecture répondant à l'ensemble des exigences que nous avons présentées au point 3.1.

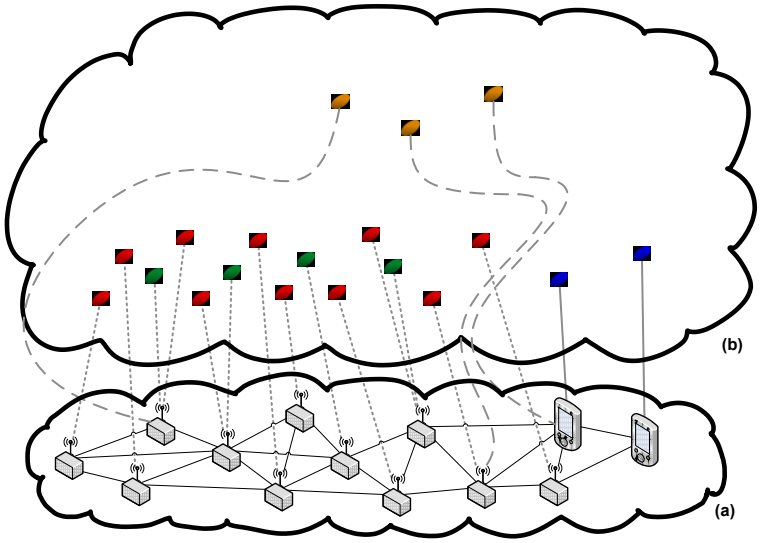

Figure 3.1 – Réseaux réels et surcouche pair à pair

Notre architecture est composée de deux principales entités représentées sur la Figure 3.1. Tout d'abord le nuage (a) qui représente le réseau réel, constitué des différents dispositifs utilisés, à savoir les capteurs / agrégateurs et les équipements des utilisateurs finaux. Le nuage (b) représente la surcouche pair à pair réalisée à l'aide de l'intergiciel. Les traits gris représentent le fait que les équipements se connectent à l'intergiciel pair à pair.

3.2.1 Les différentes types de nœuds de la surcouche pair à pair

Nous avons introduit dans l'architecture quatre entités différentes. Chaque entité définit des capacités très précises. Certaines entités sont communes à toutes les architectures de réseau de capteurs sans fils, d'autres comme le gestionnaire

d'historique, sont plus spécifiques à notre architecture. Voici une liste exhaustive des compétences de chaque entité.

À noter qu'étant donné que les agrégateurs et les gestionnaires d'historique ont des comportements similaires, une bonne solution pour l'implémentation consiste à localiser ces deux entités sur un même dispositif.

3.2.1.1 Le capteur (*sensor*)

Le rôle de capteur, comme le dit bien son nom, définit la capacité d'un nœud à effectuer des mesures pour ensuite les envoyer soit à un agrégateur, soit à un utilisateur final en réponse à une requête. Cela implique aussi la capacité de gérer la procédure de mise en grappe (*clustering*).

Les capteurs apparaissent en rouge sur la Figure 3.7.

3.2.1.2 L'agrégateur (*aggregator*)

Les agrégateurs vont être en charge d'agréger les valeurs reçues de plusieurs capteurs pour ensuite les sauvegarder grâce à un nœud de sauvegarde (*History Keeper*).

Les agrégateurs correspondent aux têtes des grappes (*Cluster Head*). Ils ne sont pas utiles si on choisit de ne pas utiliser de procédure de mise en grappe. Dans ce cas, les données seront directement envoyées par les capteurs aux gestionnaires d'historique sans agrégation. Nous n'allons pas nous intéresser à cette disposition qui ne présente aucun intérêt particulier.

Les agrégateurs apparaissent en vert sur la Figure 3.7.

3.2.1.3 Le gestionnaire d'historique (*History Keeper*)

Le rôle de gestionnaire d'historique définit la capacité d'un nœud à construire un historique de valeurs de manière distribuée avec les autres gestionnaires d'historique présents sur le réseau. Cela implique donc la capacité à conserver, durant une période

limitée, les valeurs qu'il reçoit, et éventuellement à la propager à d'autres gestionnaires pour assurer une redondance. Le but n'est pas de conserver un historique complet de toutes les mesures effectuées durant la vie du réseau, mais juste de fournir un cache des dernières valeurs, permettant ainsi un accès facile à ces valeurs et un accès à quelques valeurs dans le passé. Les valeurs sont supprimées de l'historique selon une date de validité prédéfinie ou pour faire de la place à des valeurs plus récentes. Le rôle de gestionnaire d'historique implique également la faculté de répondre aux requêtes des utilisateurs finaux.

Les gestionnaires d'historique apparaissent en orange sur la Figure 3.7.

3.2.1.4 L'utilisateur final (*End-User*)

Enfin, la capacité d'un nœud à agir en tant qu'utilisateur final définit le fait qu'il va faire tourner l'application qui va avoir à utiliser les données issues du réseau de capteurs, quelles soient stockées dans l'historique ou que ce soit des valeurs instantanées. Cela implique donc la capacité à envoyer des requêtes à un gestionnaire d'historique ou directement à un capteur.

Les utilisateurs finaux apparaissent en bleu sur la Figure 3.7.

3.2.2 Les éléments de l'architecture

Dans notre architecture différents éléments sont remarquables. Nous allons essayer d'en dresser ici une liste exhaustive, et d'en expliquer les principes, et leur utilité.

3.2.2.1 Les strates de la surcouche pair à pair

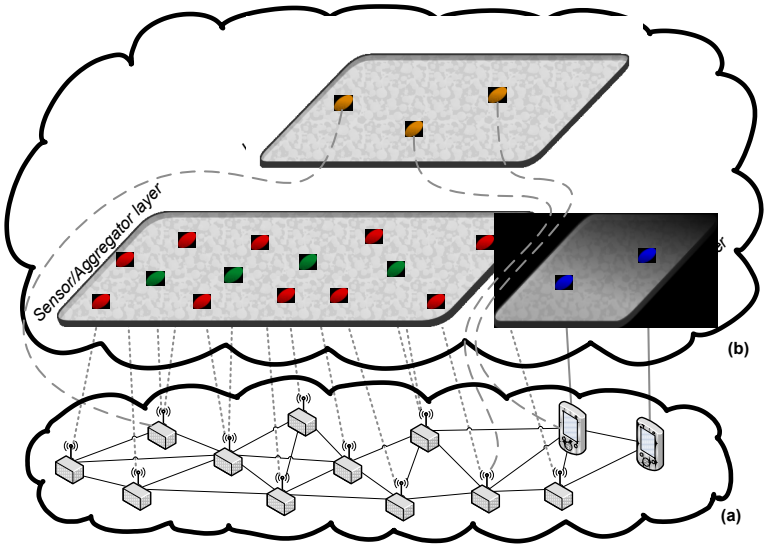

Figure 3.2 – Les trois strates de la surcouche pair à pair

La Figure 3.2 montre comment la surcouche pair à pair est découpée en trois strates, correspondant aux différentes entités que nous avons introduites précédemment. Les capteurs et les agrégateurs étant regroupés au sein d'une seule strate.

Cette mise en strate correspond en fait à une classification des nœuds selon leur centre d'intérêt et des communications dont ils vont avoir besoin pour fonctionner. Ainsi les agrégateurs et les capteurs, qui constituent un sous-ensemble des réseaux de capteurs sans fils traditionnels, se retrouvent ensemble puisqu'ils réalisent conjointement la même tâche qui est de faire des mesures (capteurs) et de les rendre accessibles sur le réseau (agrégation). On aurait pu différencier selon deux strates mais cela n'aurait pas bien fait ressortir le lien fort qui existe entre les capteurs et les agrégateurs, notamment vis-à-vis des clusters présentés au point suivant.

Les gestionnaires d'historique sont sur une strate différente puisqu'ils jouent un rôle complétement différent, à savoir celui de gérer l'historique.

Enfin les utilisateurs finaux sont eux aussi sur une autre strate, puisqu'ils jouent un rôle différent et très particulier dans le réseau.

3.2.2.2 Les clusters (grappes)

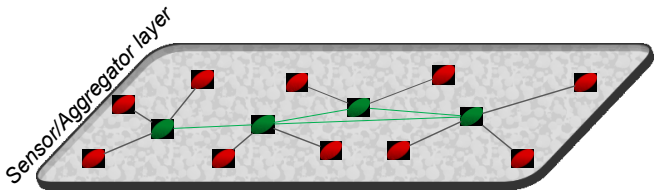

Figure 3.3 – Utilisation des clusters

Pour l'agrégation des données, nous avons pris le parti de définir des clusters. Les têtes de grappes (*cluster head*) sont alors les agrégateurs. Cette disposition va surtout nous permettre de limiter le trafic dans le réseau. En effet, le processus de mise en grappes va avoir pour rôle de s'assurer qu'en cas de déplacement des nœuds, les capteurs soient toujours connectés à l'agrégateur le plus proche. Le choix de l'algorithme de mise en grappe est laissé à l'utilisateur.

3.2.2.3 Utilisation des gestionnaires d'historique

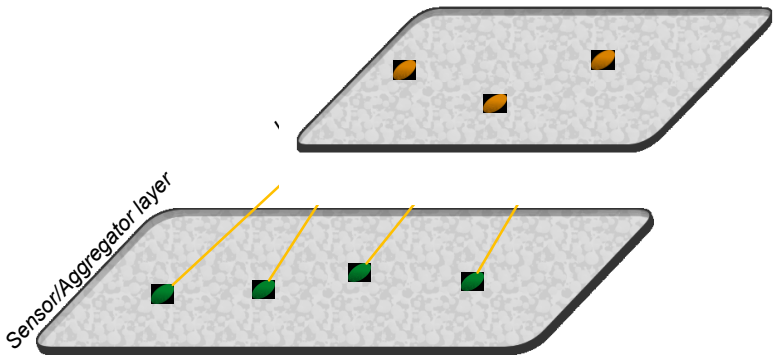

Figure 3.4 – Principe d'utilisation des gestionnaires d'historique.

Les agrégateurs sont enregistrés et connectés aux gestionnaires d'historique. Il y a forcément un gestionnaire d'historique pour chaque agrégateur. Les agrégateurs envoient les valeurs reçues de tous les capteurs de leur cluster à leur gestionnaire d'historique, lesquels stockeront ces valeurs dans l'historique distribué pour une période prédéfinie, ou jusqu'à ce qu'il faille de la place pour des valeurs plus récentes.

Ce système permet de limiter les requêtes faites directement aux capteurs. En effet, si elles sont disponibles, les utilisateurs pourront accéder aux valeurs à partir de l'historique sans avoir à interroger systématiquement les capteurs. De plus, cela permet de conserver un certain nombre de valeurs passées des capteurs, et ainsi offre aux utilisateurs la possibilité d'accéder à des valeurs mesurées alors qu'ils n'étaient même pas encore dans le réseau.

Dans cette architecture les agrégateurs permettent donc de limiter les ressources réseau nécessaires à l'enregistrement des données, mais aussi de limiter le nombre de connexions et donc la charge des gestionnaires d'historique.

3.2.2.4 Historique distribué

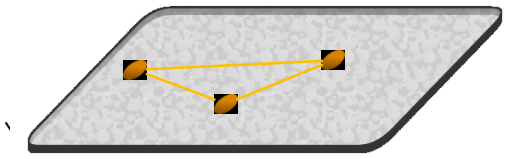

Figure 3.5 – Historique distribué sur la strate des gestionnaires d'historique.

L'historique sera donc formé par l'ensemble des gestionnaires d'historique. Lesquels ont la charge d'assurer la distribution de cet historique sur l'ensemble des gestionnaires afin de répartir la charge et d'en assurer la fiabilité. Il n'est pas question de

conserver les données générées sur l'ensemble de la durée de vie du réseau, mais juste les dernières valeurs sur une période limitée.

3.2.2.5 Accès aux données

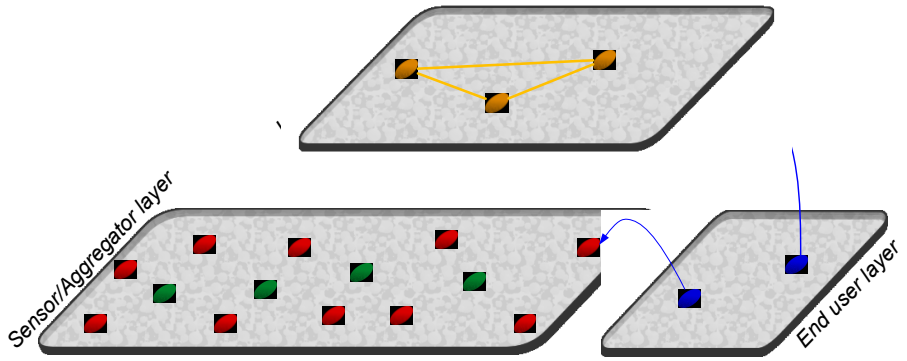

Figure 3.6 – Principe d'accés aux données

Une fois l'ensemble de l'architecture mis en place, les utilisateurs finaux peuvent accéder aux données soit en interrogeant l'historique via un gestionnaire d'historique, soient en interrogeant directement un capteur par connexion directe sur celui-ci.

3.2.3 Schéma global de l'architecture

Voici donc un schéma représentant l'ensemble de l'architecture que nous proposons. Elle est sans puits et permet une connexion directe entre un agrégateur et un capteur.

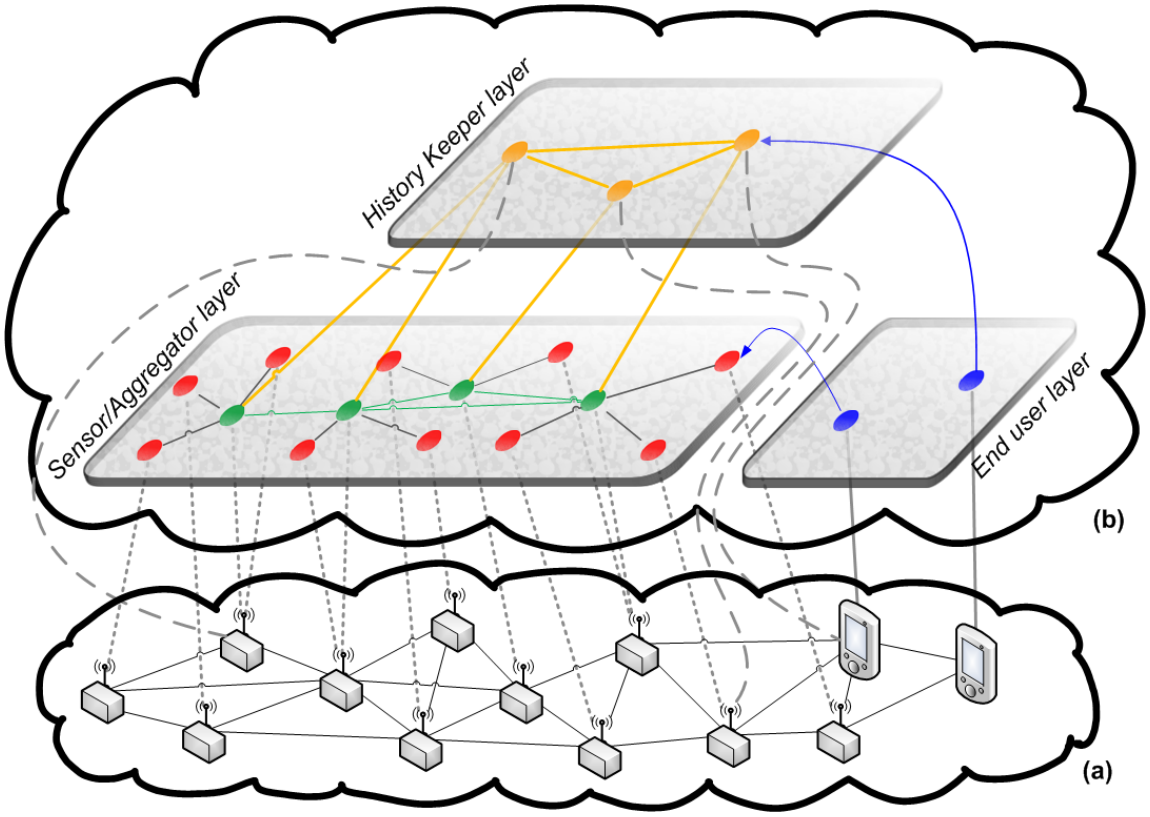

Figure 3.7 – Architecture sans puits basée sur une surcouche pair à pair pour réseau de capteurs sans fils
(a) réseau réel des dispositifs; (b) surcouche pair à pair

Le principe général de fonctionnement est le suivant : les capteurs envoient leurs mesures à la tête de la grappe qui va ensuite s'occuper de les faire sauvegarder par un gestionnaire d'historique. Les utilisateurs finaux peuvent alors accéder aux données via l'historique ou par connexion directe avec un capteur.

3.2.4 Procédures et protocoles

Nous allons ici voir en détail l'ensemble des procédures et des cas qui peuvent se présenter lors de l'utilisation de notre architecture.

Nous avons donc pris le parti d'utiliser des messages SIP pour les communications entre les nœuds puisque c'est un protocole standard, personnalisable, et qu'il nous permet d'exprimer aisément la sémantique dont nous avons besoin.

3.2.4.1 Découverte des nœuds

Pour la découverte des nœuds dans le réseau, afin de garder une totale indépendance vis-à-vis de l'intergiciel pair à pair nous avons pris le parti d'utiliser le multicast.

Nous aurions en effet pu utiliser les primitives d'accès offertes par des intergiciels pour la découverte de pairs sur le réseau, mais cela aurait été en contradiction avec notre exigence d'indépendance par rapport à l'intergiciel. Si l'utilisation du multicast est probablement moins performante que ces processus de découvertes intégrés, il offre quand même un bon compromis entre indépendance et performances.

Pour limiter la surcharge de trafic sur le réseau, nous avons défini quatre groupes multicast :

- Un groupe pour les capteurs ;
- Deux groupes pour les agrégateurs :
    - Un général, pour tous les agrégateurs,
    - Un réservé aux agrégateurs actifs (voir point 3.2.4.2) ;
- Un groupe pour les gestionnaires d'historique.

Pour découvrir un capteur, un agrégateur ou un bien un gestionnaire d'historique, il suffira alors d'envoyer un message SIP INFO avec un petit arbre XML contenant les informations sur la requête. Par exemple la Figure 3.8 signifie que nous recherchons un agrégateur actif.

```
<request>
  <limit>5</limit>
  <role name="aggregator" status="active" />
</request>
```

Figure 3.8 – Exemple de requête de découverte d'un agrégateur actif

Tel que décrit dans le RFC 3261, les nœuds devront alors répondre sur le même groupe multicast. Seuls les nœuds correspondant aux critères de recherche définis dans la requête devront répondre. Les réponses devront inclure les informations sur tous les rôles que joue le nœud.

```
<reply>
  <role name="sensor">
    <field name="position" value="room1" />
  </role>
  <role name="aggregator" status="active" />
  <role name="historykeeper" status="down" />
</reply>
```

Figure 3.9 – Exemple de réponse de découverte

Si l'utilisation du multicast pour la réponse n'est pas non plus forcément optimale pour les réseaux de capteurs, cela va nous permettre d'assurer une certaine gestion du nombre de réponses, et au besoin, les autres nœuds pourront garder eux aussi en mémoire les informations contenues dans la réponse. À noter que, principalement en raison de la latence du réseau, il est parfaitement possible de recevoir plus de réponses que le maximum fixé.

3.2.4.2 Connexion d'un agrégateur

Quand un agrégateur / capteur rejoint le réseau pair à pair, il doit se faire connaître sur le réseau et remplir son rôle en prenant en compte l'état actuel du réseau. Sa première action va donc être de rechercher un agrégateur et de déterminer dans quel état il doit se mettre. Le processus de connexion va se dérouler selon l'arbre de décision présenté sur la Figure 3.10.

Il est important de remarquer que pour différentes raisons, et principalement pour prendre en compte l'exigence de mobilité de nœuds, il est possible d'avoir plus d'agrégateurs dans le réseau que nécessaire. Pour optimiser les performances, certains de ces agrégateurs vont rester inactifs, et donc ne vont pas jouer leur rôle d'agrégateur. Nous allons donc parler « d'agrégateur actifs » pour distinguer ceux dont le rôle est activé et « d'agrégateur » pour parler de l'ensemble des agrégateur, actifs ou non.

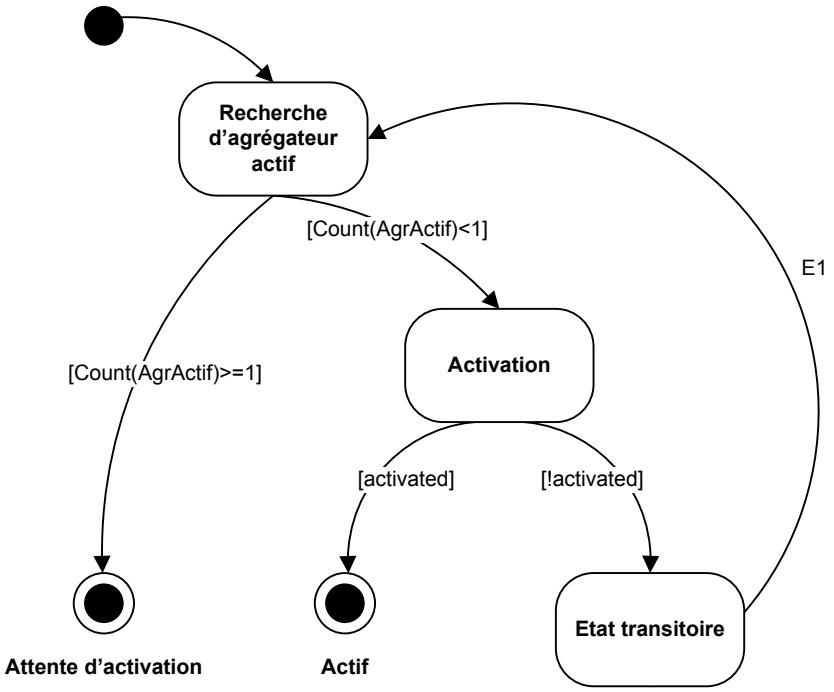

Figure 3.10 – Processus à la connexion d'un agrégateur

- **États finaux**

  Ces états signifient que l'agrégateur a atteint un état stable, et qu'il ne va pas chercher à changer d'état à moins d'y être invité.

  - **Attente d'activation :** Cet état signifie qu'il y'a déjà au moins un agrégateur actif dans le réseau. Le nouvel agrégateur va donc simplement

attendre qu'un agrégateur actif lui demande de s'activer à son tour. Il est accessible en utilisant le groupe multicast des agrégateurs.

- **Actif :** Il n'y avait aucun agrégateur dans le réseau, il s'est donc auto activé. Des capteurs peuvent donc s'enregistrer auprès de lui pour lui envoyer leurs données. L'agrégateur sera joignable en utilisant les groupes multicast des agrégateurs, des agrégateurs actifs ou par unicast.

- **État transitoire**

L'agrégateur arrive dans l'état transitoire quand il n'arrive pas à s'auto-déclarer. Cela peut se produire s'il ne trouve pas de gestionnaire d'historique. Régulièrement ou suite à des évènements comme la connexion d'un gestionnaire d'historique, l'agrégateur va relancer le processus afin de voir si un autre agrégateur a réussi à s'activer ou s'il parvient à s'activer lui-même.

3.2.4.3 Connexion d'un capteur

Quand un capteur rejoint le réseau, il va suivre le processus décrit pas la Figure 3.11.

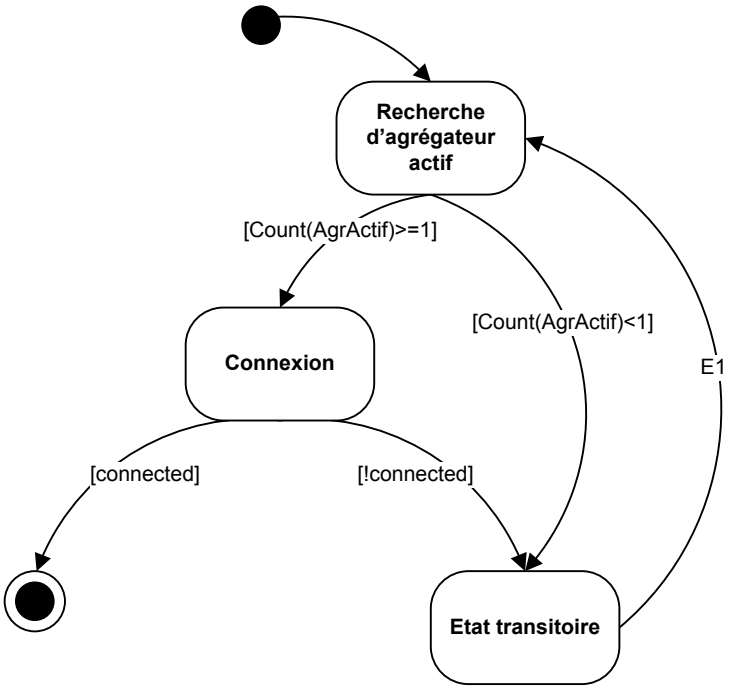

Figure 3.11 – Processus à la connexion d'un capteur.

- **État transitoire :** Dans cet état le capteur peut accepter les requêtes provenant d'un utilisateur final, mais comme il n'a pas réussi à s'enregistrer auprès d'un agrégateur, il ne peut pas envoyer de lui-même ses valeurs. Suite à certains événements comme la connexion d'un agrégateur ou simplement de manière régulière, le capteur va relancer le processus de découverte d'agrégateur afin de tenter de quitter cet état transitoire.

- **État final :** Le capteur peut recevoir des requêtes d'un utilisateur final et a trouvé un agrégateur après duquel il a réussi à s'enregistrer, il lui envoie donc régulièrement les mesures qu'il effectue.

3.2.4.4 Connexion d'un gestionnaire d'historique

Comme pour les agrégateurs nous pouvons avoir plus de gestionnaires d'historique que nécessaire, nous allons donc en garder certains inactifs. Ainsi lorsqu'un

gestionnaires d'historique rejoint le réseau, il va rejoindre le groupe multicast des gestionnaire d'historique, puis va se mettre en attente (i.e. écoute de requêtes SIP) qu'un agrégateur l'active.

3.2.4.5 Connexion d'un utilisateur final

Quand un utilisateur final se connecte, il va commencer par rechercher un gestionnaire d'historique. S'il en trouve un, il pourra lui envoyer des requêtes pour obtenir des données issues de l'historique. S'il n'en trouve pas, il ne pourra qu'interroger des capteurs sur leur état actuel par communication directe.

3.2.4.6 Activation d'un agrégateur

Lorsqu'un agrégateur s'active, c'est-à-dire qu'il se met à réellement jouer son rôle d'agrégateur, il va chercher à se connecter à un gestionnaire d'historique. S'il n'en trouve aucun d'actif, il va tenter d'en activer un à l'aide d'une requête INVITE. Ensuite il va s'enregistrer auprès de ce gestionnaire d'historique. S'il n'y arrive pas, l'agrégateur refusera de s'activer. L'action peut être demandée soit par un agrégateur déjà actif soit par auto-activation tel que décrit dans le processus de connexion d'un agrégateur.

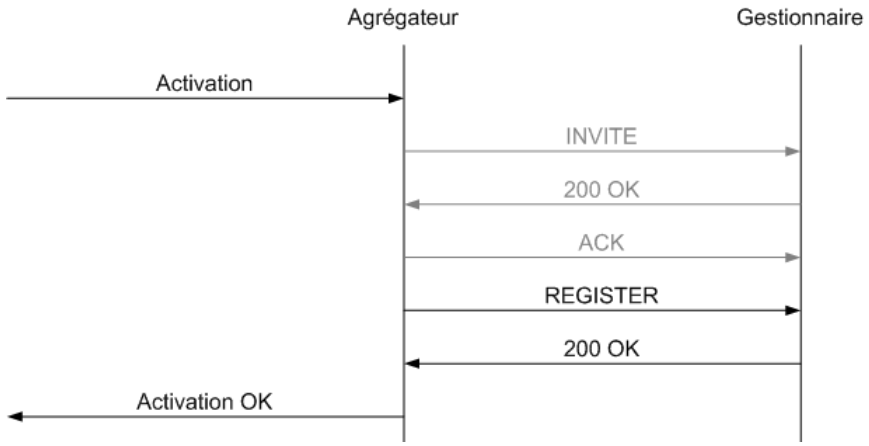

Figure 3.12 – Diagramme de séquence, activation d'un agrégateur avec activation d'un gestionnaire d'historique.

Le coût de communication *C* d'une telle procédure, va être :

Tableau 3.1 – Coût de communication de l'activation d'un agrégateur

	Auto activation de l'agrégateur	Activation par un autre agrégateur
Gestionnaire d'historique déjà actif	Cas impossible	$C = 2C_d + 2c$
Nécessité d'activer un nouveau gestionnaire d'historique	$C = C_d + 5c$	$C = 2C_d + 5c$

Où C_d représente le coût de découverte d'un nœud, qui est faite sur un groupe multicast et dont le coût va dépendre de la topologie du réseau et de l'intergiciel. c étant le coût d'une communication entre deux nœuds du réseau en unicast, il va lui aussi dépendre de la topologie du réseau. À noter que C_d est largement supérieur à c.

3.2.4.7 Mise en grappe

Durant le processus de connexion, le capteur est amené à tenter de s'enregistrer sur un agrégateur. Pour cela, le capteur va envoyer une requête REGISTER à l'agrégateur qui pourra répondre soit un succès, soit répondre un refus, selon les règles de l'algorithme de formation des grappes. Dans ce cas-là l'agrégateur devra joindre à la réponse les informations nécessaires à la connexion à un autre agrégateur capable de gérer le capteur. Ce nouvel agrégateur pourra avoir été activé pour l'occasion par l'ancien agrégateur, comme décrit dans la Figure 3.13. Le but est de limiter le travail des capteurs dont les ressources sont les plus limitées.

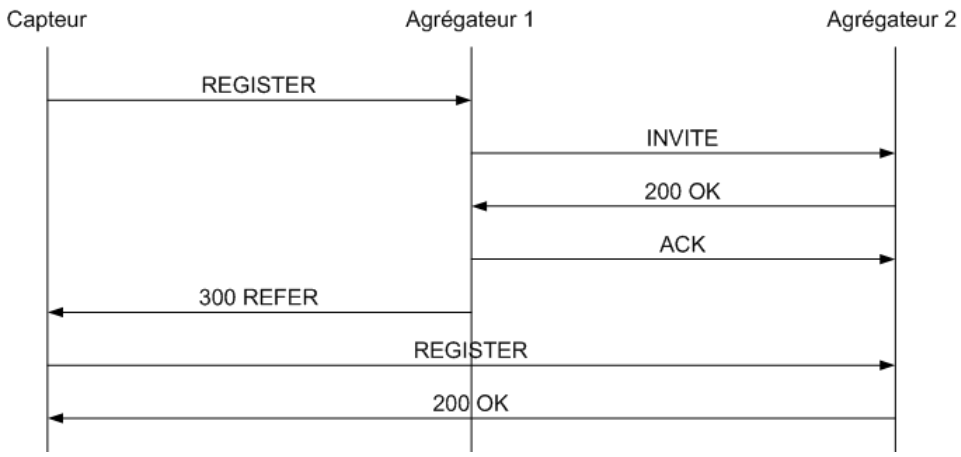

Figure 3.13 – Diagramme de séquence, enregistrement d'un capteur auprès d'un agrégateur, avec report

Le coût de communication *C* de cette procédure sera :

$$C = C_d + 7c$$

3.2.4.8 Optimisation des grappes

Suite à la création d'une nouvelle grappe, ou bien encore compte tenu de la mobilité des nœuds du réseau, un algorithme d'optimisation des grappes peut être mis en œuvre. Le but est de minimiser le coût des communications entre le capteur et son agrégateur.

Pour ce faire les agrégateurs doivent partager un moyen de communication commun efficace. Une bonne solution serait l'utilisation de messages multicast comme le proposent les intergiciels pair à pair.

Une fois la décision prise de changer l'agrégateur d'un capteur, il sera averti au moyen d'une requête UPDATE contenant les informations concernant le nouvel agrégateur qui lui a été assigné. Le coût de communication de cette procédure, sans prendre en compte les coûts relatifs au fonctionnement de l'algorithme d'optimisation, sera $C = C_d + 4c$. Il sera donc intéressant de faire cette procédure si *C* est inférieur à la

somme, sur l'ensemble des publications futures, de la différence entre le coût sans et avec optimisation :

$$C < \sum c_{non\ opt} - c_{opt}$$

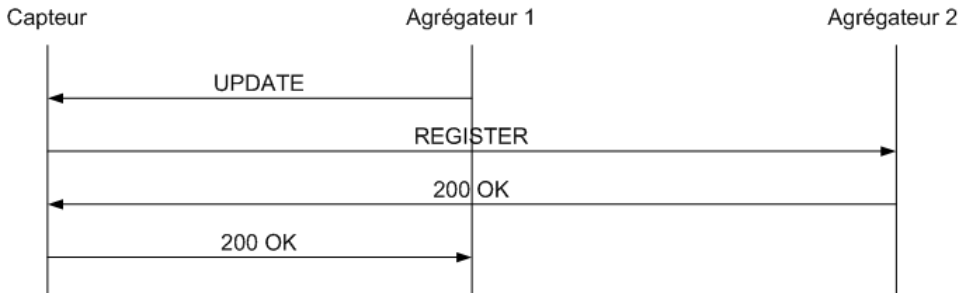

Figure 3.14 – Diagramme de séquence, mise à jour de l'agrégateur d'un capteur

L'optimisation des coûts s'applique aussi entre un agrégateur et son gestionnaire d'historique. Pour cela, l'agrégateur va simplement de lui-même s'enregistrer auprès d'un autre gestionnaire d'historique. Le coût de communication sera aussi $C = C_d + 4c$ (sans tenir compte des coûts de fonctionnement de l'algorithme).

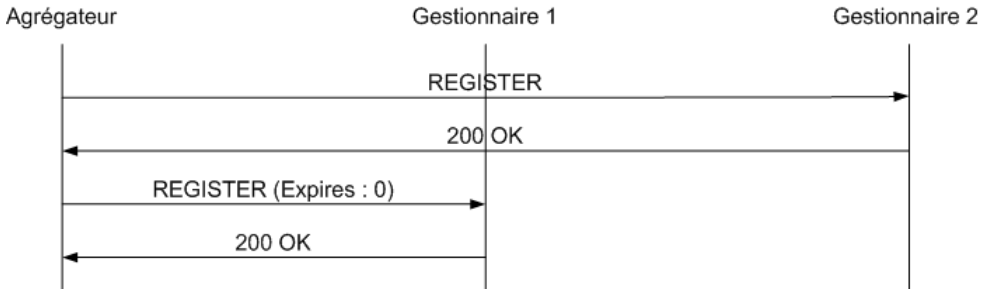

Figure 3.15 – Diagramme de séquence, changement de gestionnaire d'historique

3.2.4.9 Agrégation des données

Une fois les capteurs connectés à un agrégateur, les capteurs vont devoir faire des mesures régulières qu'ils vont publier vers cet agrégateur via des requêtes PUBLISH.

3.2.4.10 Sauvegarde des données dans l'historique

L'agrégateur va alors à son tour renvoyer ces valeurs vers un gestionnaire d'historique, lui aussi avec une requête PUBLISH.

3.2.4.11 Récupération de données de l'historique

Une fois publiées et sauvegardés par les gestionnaires d'historique, les données sont accessibles aux utilisateurs finaux via une simple requête SIP INFO sur le réseau standard. Cela permet de ne pas avoir à interroger plusieurs capteurs pour obtenir de chacun sa valeur, ce qui limite la charge réseau, et permet aussi d'accéder à d'anciennes valeurs d'un capteur.

3.2.4.12 Communication directe entre un utilisateur final et un capteur

L'utilisateur final peut découvrir les capteurs présents sur le réseau grâce au socket multicast des capteurs, pour ensuite leur envoyer directement des requêtes INFO pour connaître la valeur courante du capteur. Seule la valeur courante est accessible par ce moyen.

3.2.4.13 Déconnexion d'un nœud

- **Capteur :** Dans le cas d'une déconnexion normale, il doit se déconnecter de son agrégateur. Dans le cas d'une défaillance, l'agrégateur libérera tout seul les ressources occupées par ce capteur au bout d'un certain temps sans action (*timeout*).

- **Agrégateur :** Dans le cas d'une déconnexion normale il devra élire un nouvel agrégateur (INVITE) et alerter tous les capteurs du changement via un message UPDATE comme dans l'optimisation des grappes. En cas de déconnexion non planifiée, les capteurs n'arriveront plus publier les valeurs et vont d'eux même changer d'agrégateur. Le gestionnaire d'historique, va lui détecter la défaillance au bout d'un *timeout* et va libérer les ressources dédiés à cet agrégateur.

- **Gestionnaire d'historique :** Lorsque un gestionnaire d'historique quitte le réseau correctement, il doit simplement avertir les agrégateurs qui sont connectés qu'il quitte le réseau grâce à une requête INFO. Les agrégateurs devront retrouver un autre gestionnaire d'historique ou revenir à un état transitoire. En cas de défaillance, les agrégateurs ne pourront plus faire de PUBLISH et vont d'eux-mêmes trouver un nouveau gestionnaire d'historique.

- **Utilisateur final :** La déconnexion d'un utilisateur final n'aura aucune incidence sur le réseau.

3.3 Exemple de fonctionnement

Afin de bien voir les interactions entre les nœuds et voir comment notre architecture permet de répondre à l'ensemble des exigences que nous avions défini, nous allons voir en détails le scénario d'utilisation présenté au point 1.3.1, à savoir le système de détection d'incendie.

3.3.1 Présentation du scénario

Dans ce scénario, dans un but de clarté, nous allons nous limiter à un petit réseau composé de seulement quelques dispositifs :

- quatre capteurs fixes
- un dispositif d'utilisateur final (type PDA) mobile qui sera manipulé par un pompier.

Tableau 3.2 - Rôles implémentés par les dispositifs pour le scénario de test

Dispositif	Capteur	Agrégateur	Gest. d'historique	Utilisateur final
Capteur 1	SD1			
Capteur 2	SD2	AD2	HKD2	
Capteur 3	SD3	AD3		
Capteur 4	SD4			
PDA 5				EUD5

Le Tableau 3.2 indique quels rôles sont implémentés par chacun des dispositifs, ainsi que le nom attribué à chaque nœud correspondant dans le réseau pair à pair. Les dispositifs vont rejoindre un à un le réseau, dans l'ordre du Tableau 3.2.

Pour ce qui est de la mise en grappes, nous allons considérer qu'un agrégateur ne peut gérer que trois capteurs à la fois. Et pour les gestionnaires d'historique nous considèrerons qu'un gestionnaire peut s'occuper de trois agrégateurs.

3.3.2 Description des échanges

Dans un premier temps, SD1 va rejoindre le réseau. Étant seul sur le réseau, il ne va pas trouver d'agrégateur et va donc se mettre en état transitoire tel que décrit au point 3.2.4.2. À ce moment il ne peut que répondre à des requêtes reçues depuis un utilisateur final.

Quand le dispositif 2 va rejoindre le réseau réel, trois nœuds se connectent au réseau pair à pair : HKD2, AD2 et SD2. Au démarrage de l'agrégateur AD2, il ne va pas trouver d'agrégateur sur le réseau, il va donc s'auto-déclarer agrégateur et au passage activer le gestionnaire d'historique HKD2. Ensuite SD1 et SD2 vont pouvoir le détecter, s'enregistrer auprès de lui (REGISTER) et commencer à publier des mesures (PUBLISH). SD1 passe alors de l'état transitoire à l'état normal.

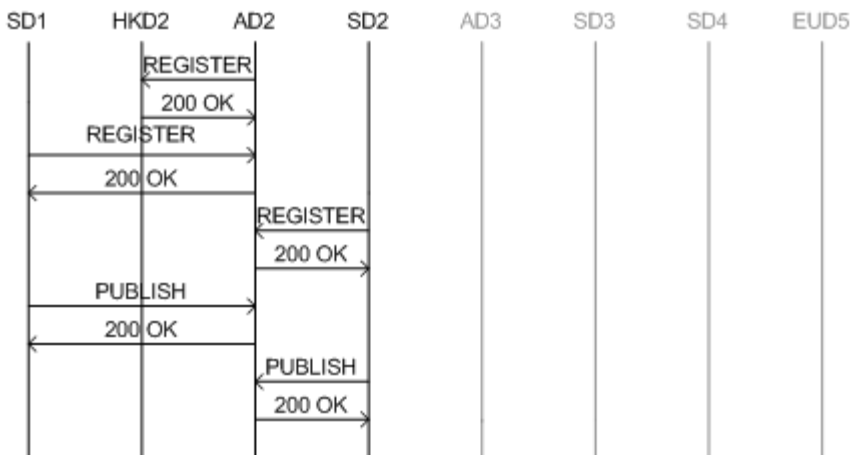

Figure 3.16 - Diagramme de séquence, connexion du premier agrégateur

Lorsque SD3 et AD3 se connectent au réseau P2P, AD3 détecte la présence d'un agrégateur, il contrôle que l'agrégateur est encore valide en lui envoyant une requête REGISTER dont le header « expires » est défini à zéro et le header « user-agent » est défini à « aggregator ». Cette opération a pour but d'éviter que la présence d'un agrégateur fantôme dans le cache de l'intergiciel ne bloque tout le réseau. L'agrégateur est considéré comme non fantôme à partir du moment où il envoie une réponse, que ce soit un « succès » ou un « échec ». AD3 se contente alors de s'annoncer sur le réseau mais ne s'active pas comme un agrégateur, tandis que SD3 lui va s'enregistrer après de l'agrégateur AD2.

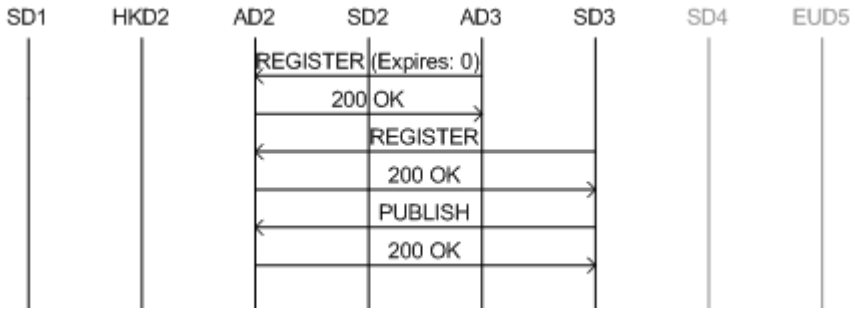

Figure 3.17 - Diagramme de séquence, connexion du deuxième agrégateur

Au moment où SD4 va rejoindre le réseau, il va chercher un agrégateur et ne va trouver que AD2. Il va donc tenter de s'enregistrer auprès de lui. Seulement AD2 gère déjà trois capteurs : SD1, SD2 et SD3, soit la limite de capteurs pour un seul agrégateur, défini dans nos hypothèses de travail au 3.3.1. AD2 va donc refuser SD4, mais il va cependant devoir envoyer à SD4 les coordonnées d'un autre agrégateur qui sera capable de s'occuper du capteur. Cette procédure est ainsi conçue dans le but de limiter les échanges que le capteur doit faire et permet donc de préserver ses ressources.

Pour le moment, il n'y a qu'un seul agrégateur actif dans le réseau : AD2. Il va donc falloir activer AD3 en lui envoyant un message SIP INVITE. Si AD2 n'avait pas trouvé d'agrégateur à activer, il aurait simplement refusé le capteur, lequel serait alors entré dans un état transitoire (c'est-à-dire qu'il n'aurait fait que répondre aux requêtes d'un utilisateur final). Une fois l'agrégateur AD3 activé, SD4 peut s'y connecter.

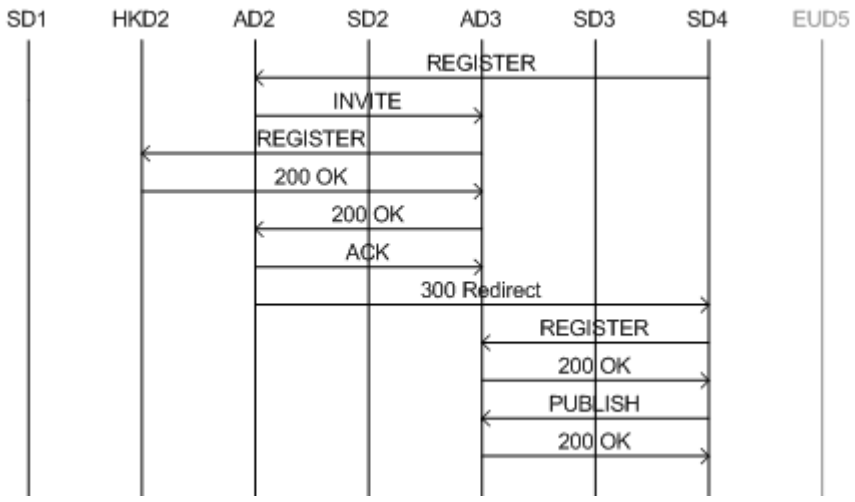

Figure 3.18 - Diagramme de séquence, connexion d'un nouveau capteur avec activation d'un agrégateur à la volée

Comme SD3 est plus proche d'AD3 que d'AD2 (son agrégateur actuel), l'algorithme d'optimisation des grappes va décider de déplacer SD3 vers l'agrégateur récemment activé. L'algorithme d'optimisation des grappes se déroule entre les

agrégateurs via, par exemple, un socket multicast, et n'est pas représenté ici. La mise à jour de la route va être faite par un message UPDATE de l'agrégateur au capteur.

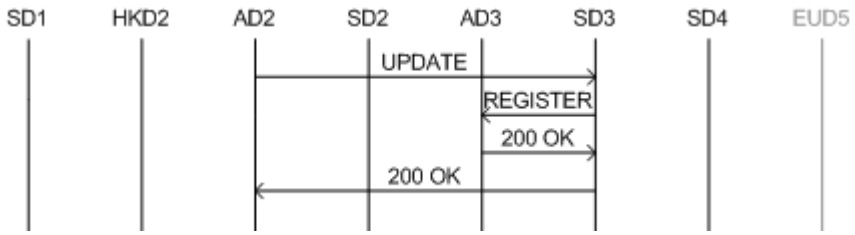

Figure 3.19 - Diagramme de séquence, optimisation des grappes

Enfin quand EUD5 se connecte il peut dorénavant interroger les capteurs et les gestionnaires d'historiques. En cas de défaillance d'un grand nombre de capteurs, par exemple à cause du feu, les capteurs demeurent ainsi accessibles par connexion directe par des messages INFO. Cette connexion directe permet aussi d'avoir une vue rapide de l'environnement proche de l'utilisateur.

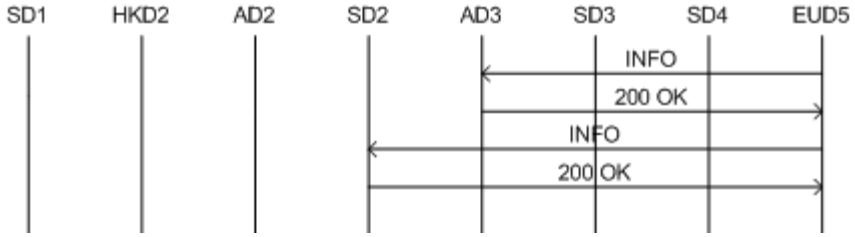

Figure 3.20 - Diagramme de séquence, utilisation du réseau

CHAPITRE 4 VALIDATION DE L'ARCHITECTURE

Afin de tester notre architecture nous avons simulé une utilisation basique de notre architecture. C'est cette simulation et les résultats obtenus que nous allons présenter dans cette partie.

4.1 Implémentation

Nous avons implémenté les éléments basiques de notre architecture et nous l'avons testée afin d'en démontrer la faisabilité. Nous nous contenterons d'implémenter les éléments essentiels à notre architecture, les éléments optionnels seront omis. De plus nous nous limiterons juste à avoir une implémentation fonctionnelle, sans rechercher à avoir une implémentation la plus performante possible.

4.1.1 Présentation de l'architecture

La Figure 4.1 représente sommairement les principales classes et leurs interactions.

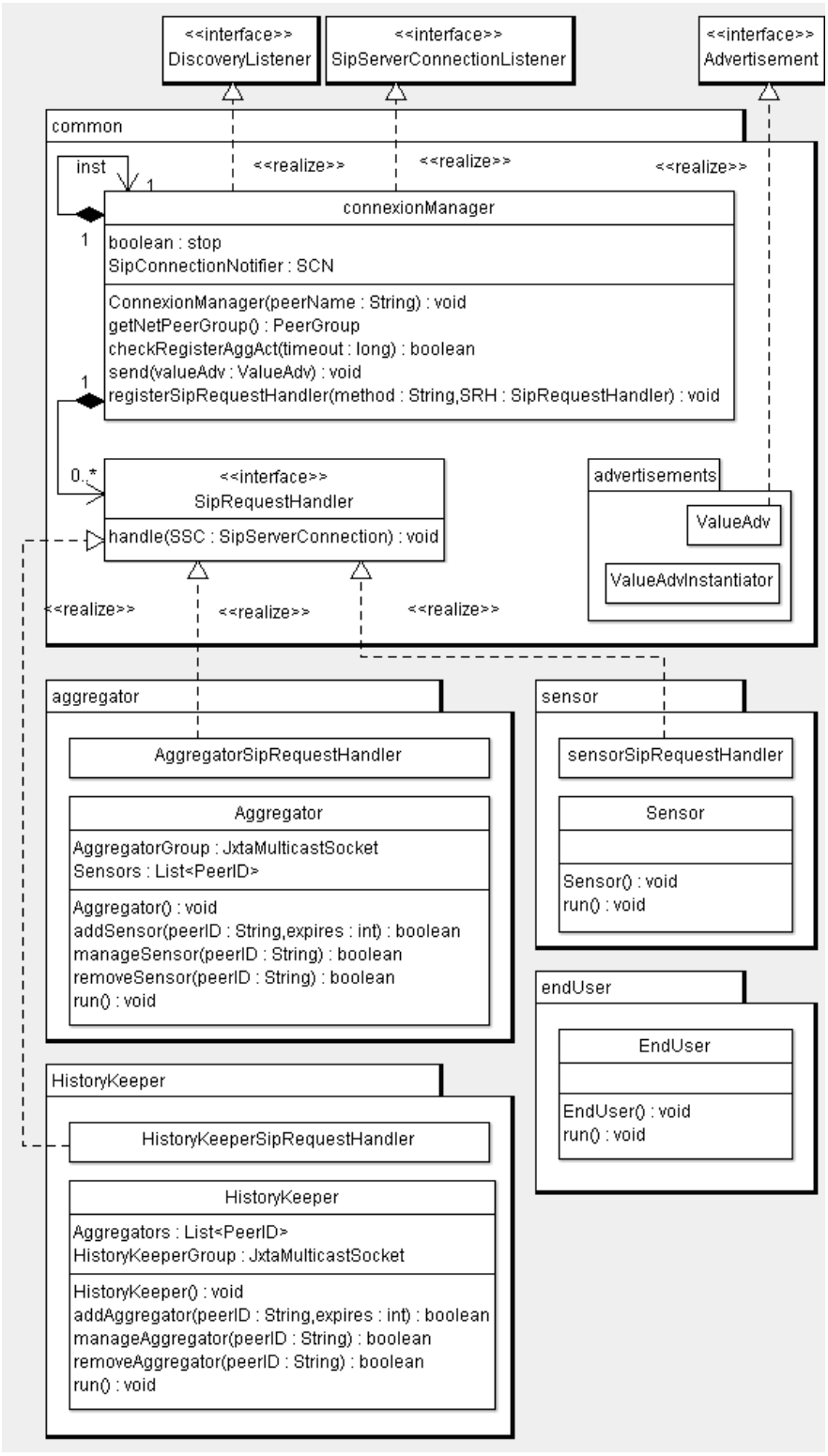

Figure 4.1 – Diagramme de classes des principales classes de notre implémentation.

Notre implémentation est construite autour de quatre packages : `common`, `aggregatorHistoryKeeper`, `sensor`, et `endUser`. Le package `common` doit être inclus sur tous les nœuds du réseau, les autres packages sont ajoutés ou non en fonction des rôles joués par un dispositif.

- **common :**
  Le package `common` contient l'ensemble des classes qui seront utilisées par tous les types de nœuds. C'est lui qui va contenir les primitives d'accès au réseau pair à pair, la gestion des requêtes SIP, ainsi que les divers outils. Il contient aussi le point d'entrée du programme qui va charger les comportements.

- **Aggregator :**
  Contient les classes utiles au fonctionnement du nœud en tant qu'agrégateur.

- **HistoryKeeper :**
  Ce package contient tous les éléments utiles à un nœud pour fonctionner en tant que gestionnaire d'historique.

- **sensor :**
  Contient le comportement de capteur. Dans notre implémentation les valeurs sont en fait générées de manière aléatoire.

- **endUser :**
  Pour implémenter la fonction d'utilisateur final.

La classe la plus importante de notre implémentation est indéniablement le `ConnexionManager` du package `common`. C'est cette classe qui offre toutes les primitives d'accès au réseau. Elle implémente notamment les classes `DiscoveryListener` et `SipServerConnectionListener`.

- **`DiscoveryListener` :**

  Permet de recevoir des événements à la réception d'une réponse à un *discover* préalablement envoyé avec le `getRemoteAdvertisements`. L'évènement reçu permettra d'avoir un accès direct aux *Advertisments* nouvellement reçus.

- **`SipServerConnectionListener` :**

  Permet d'être averti de la réception de nouvelles requêtes SIP, pour la traiter et envoyer une réponse.

Ainsi l'instanciation du `ConnexionManager` ouvre le socket pour la réception des requêtes SIP. À la réception d'une requête, un événement est levé et la méthode du `ConnexionManager` est appelée pour assurer la gestion. Cette gestion se fait alors grâce à l'enregistrement préalable de composants de gestion de requêtes (`SipRequestHandler`) sur le `ConnexionManager`. Cette architecture permet aisément de définir des comportements en fonction des rôles joués.

Étant donné que le `ConnexionManager` offre toutes les primitives d'accès au réseau, il doit être accessible à toutes les autres classes qui doivent travailler avec le réseau, c'est-à-dire presque toutes. Comme on ne peut pas définir de variable à portée globale en Java, on a utilisé un subterfuge consistant à ce que le `ConnexionManager` se référence lui-même. Ainsi, à l'instanciation d'un `ConnexionManager`, il va s'enregistrer lui-même dans une variable statique et publique `inst`. De cette façon, depuis n'importe où dans le programme, on pourra accéder à l'instance courante du `ConnexionManager` simplement en faisant `ConnexionManager.inst`.

4.1.2 Création d'une annonce personnalisée

Comme dit dans la présentation, Jxta est basé sur les annonces qui sont stockées dans les caches des pairs du réseau. Chaque annonce consiste en fait en un petit fichier XML dans lequel est stocké notamment l'ID du nœud qui a créé l'*advertisment*, un certain nombre de valeurs indexables par les pairs, une durée de validité, et diverses autres informations.

Cela correspond parfaitement à ce dont nous avons besoin pour sauvegarder les informations sur les gestionnaires d'historique.

Nous avons donc défini une classe `ValueAdv` qui implémente l'interface `Advertisment` et qui représente l'annonce d'une valeur d'un capteur. Les annonces de valeurs sont instanciées par les capteurs, sérialisées en XML et envoyées aux agrégateurs dans le contenu des messages PUBLISH. Si on avait dissocié agrégateur et gestionnaire d'historique, les agrégateurs auraient eu pour mission d'agréger plusieurs annonces de valeurs pour les envoyer un seul message PUBLISH au gestionnaire. Ensuite le gestionnaire d'historique a juste à publier l'annonce grâce à la méthode `remotePublish()` sur le groupe de pairs des gestionnaires d'historique.

Cependant, si l'implémentation des méthodes de l'interface `Advertisment` est assez aisée, l'utilisation d'une annonce non standard n'est pas triviale car assez peu documentée et risque de faire perdre beaucoup de temps. En effet il ne suffit pas qu'une classe implémente l'interface pour pouvoir être publiée et reçue. Il faut aussi créer une autre classe (`ValueAdvInstantiator`) implémentant l'interface `Instantiator` qui va être chargée d'instancier un nouveau `ValueAdv` à la réception d'une annonce du type `ValueAdv`.

Enfin, il ne faudra pas oublier d'enregistrer l'`instantiator` auprès de l'`AdvertisementFactory` pour que l'ensemble fonctionne.

La Figure 4.2 représente l'ensemble du processus nécessaire pour envoyer une valeur à un agrégateur en utilisant une annonce personnalisée. Les tableaux Tableau 4.1 et Tableau 4.2 contiennent les légendes des différents éléments numérotés de la figure.

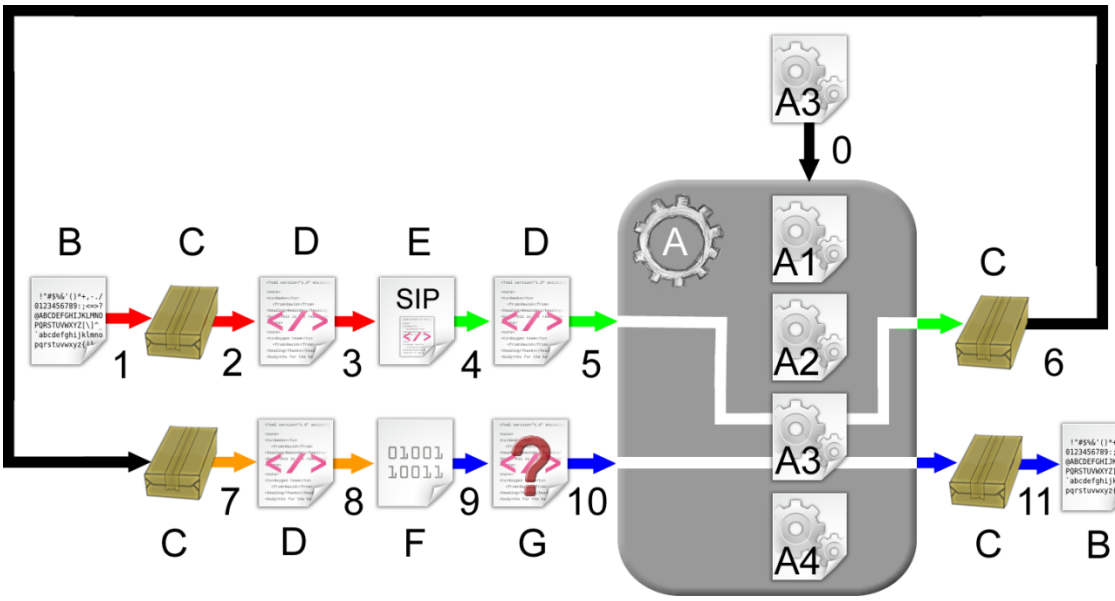

Figure 4.2 – Processus d'envoi d'une valeur via une annonce personnalisée sérialisée et désérialisée en utilisant l'*advertisment factory*

Légende des couleurs des flèches de la Figure 4.2 :

- **Rouge :** Actions effectuées par le capteur ;
- **Vert :** Actions effectuées par l'agrégateur ;
- **Orange :** Actions effectuées par le gestionnaire d'historique ;
- **Bleu :** Actions effectuées par l'utilisateur final.

Tableau 4.1 - Description des différents éléments de la Figure 4.2

Élément	Description
A	`AdvertisementFactory`.
A#	Objets implémentant l'interface `Instantiator`. A3 permettant d'instancier des objets du type de A.
B	Valeur mesurée par le capteur.
C	Objet implémentant l'interface `Advertisement`.
D	Document XML contenant la sérialisation de C.
E	Document XML D encapsulé dans une requête SIP qui est envoyé sur le réseau vers l'agrégateur.
F	Message géré par Jxta (annonce publiée sur le réseau).
G	Document XML reçu contenant la sérialisation d'une annonce. À ce stade on ne sait cependant pas de quel type d'annonce il s'agit.

Tableau 4.2 – Description des différentes actions de la Figure 4.2

Action	Description
0	Enregistrement de notre *instantiator* en appelant la méthode `AdvertisementFactory.registerAdvertisementInstance()`. C'est l'action à faire avant toute autre chose.
1	Création d'une annonce pour une valeur.
2 & 7	Sérialisation XML en appelant la méthode `Advertisement.getDocument()`.
3	Encapsulation du XML dans un message SIP.
4	L'agrégateur lit du content du message SIP reçus et extrait un fichier XML.

Action	Description
5	L'agrégateur ré-instancie l'objet A en appelant la méthode `AdvertisementFactory.newAdvertisement()`. L'`AdvertisementFactory` va sélectionner tout seul le bon *instantiator*. Ou pourra alors tester le type d'*Advertisment* dont il s'agit en comparant la classe de l'objet retourné.
6	Passage de l'annonce au gestionnaire d'historique. Correspond à répéter les étapes 2, 3, 4 et 5 en incluant plusieurs annonces de valeur dans un seul message SIP.
8	Publication de l'annonce sur le groupe des gestionnaires d'historique grâce à la méthode `remotePublish()` ou `publish()`. L'annonce est alors disponible pour tous les gestionnaires via les caches.
9	L'utilisateur final découvre les valeurs grâce aux méthodes `getRemoteAdvertisements()` et `getLocalAdvertisements()`. En réalité le *discover* sera envoyé par un gestionnaire d'historique qui transmettra ensuite les réponses à l'utilisateur final.
10	À la réception d'une annonce, on ne sait pas de quel type d'annonce il s'agit, on ne peut pas l'instancier aisément. C'est ce à quoi l'`AdvertismentFactory` est dédié. On passe donc le fichier XML contenant une annonce de type inconnu avec la méthode `newAdvertisement()` et on récupère en retour une annonce instanciée qu'on pourra traiter facilement, à condition toutefois qu'il ait un *instantiator* capable d'interpréter ce fichier XML.
11	L'utilisateur final peut maintenant récupérer les valeurs du capteur.

4.1.3 Fonctionnement global de notre implémentation

Lorsqu'on démarre notre implémentation, les rôles sont exécutés successivement en fonction de la configuration du nœud. En premier le rôle de gestionnaire d'historique, puis celui d'agrégateur, et enfin ceux de capteur ou d'utilisateur final.

- **Gestionnaire d'historique :**
  Le démarrage du rôle de gestionnaire d'historique est le plus simple. Il se contente d'ouvrir le socket multicast du groupe des gestionnaires, puis d'enregistrer le `ConnexionManager` pour recevoir les requêtes SIP entrantes. Il devra bien entendu auparavant enregistrer le `HistoryKeeperSipRequestHandler` auprès du `ConnexionManager` pour définir les requêtes acceptées et le comportement lié à la réception de ces requêtes.

- **Agrégateur :**
  Au démarrage du rôle d'agrégateur, le nœud commence par lancer le processus de découverte d'un agrégateur actif (processus commun qui est implémenté par le `ConnexionManager`). Il enregistre aussi l'`AggregatorSipRequestHandler` auprès du `ConnexionManager`. Au moment de l'activation de l'agrégateur il enregistre un second `AggregatorSipRequestHandler` sur de nouvelles requêtes SIP. L'activation de l'agrégateur provoque le démarrage d'un nouveau thread.

- **Capteurs :**
  Lorsqu'un capteur démarre, sa première action va être d'enregistrer le `SensorSipRequestHandler` afin de pouvoir recevoir les requêtes des utilisateurs finaux, sur un socket unicast ainsi que sur le groupe multicast des capteurs. Ensuite lui aussi provoque le démarrage du processus de découverte d'un agrégateur actif du `ConnexionManager`. Cependant à

l'inverse du rôle agrégateur, le processus reste actif tant qu'il ne trouve pas d'agrégateur actif. Dès qu'il en trouve un, il tente de s'y enregistrer, s'il y parvient il commence à envoyer des valeurs.

- **Utilisateur final :**
  Au démarrage du rôle de gestionnaire d'historique il n'y pas de procédure automatiquement exécutée. Il ne commence à communiquer sur le réseau que s'il a besoin d'accéder à une valeur (action de l'utilisateur). Il interroge alors l'historique via le groupe multicast des gestionnaires d'historique, ou directement un capteur par connexion unicast, après découverte via le groupe multicast des capteurs.

4.2 Environnement de test

Notre test n'a pas pour but de comparer les performances de notre architecture avec d'autres architectures traditionnelles ou sans puits. Nous allons juste chercher à évaluer sommairement la stabilité de notre architecture ainsi que son comportement en situation semblable à la réalité.

Ainsi pour notre test nous allons nous contenter de quelques dizaines de capteurs. Cependant comme malheureusement nous n'avons pas suffisamment de capteurs au laboratoire, nous allons simuler une partie de ces capteurs.

Nous allons effectuer deux séries de tests. La première série, servira à tester le fonctionnement global de l'architecture, tandis que la seconde série sera plus orientée vers la mesure de l'impact de l'architecture sur les ressources des nœuds.

4.2.1 Environnement des tests de la série 1

Comme annoncé, les tests de la première série sont avant tout faits pour analyser le comportement global du réseau. L'environnement de test est représenté sur la Figure 4.3.

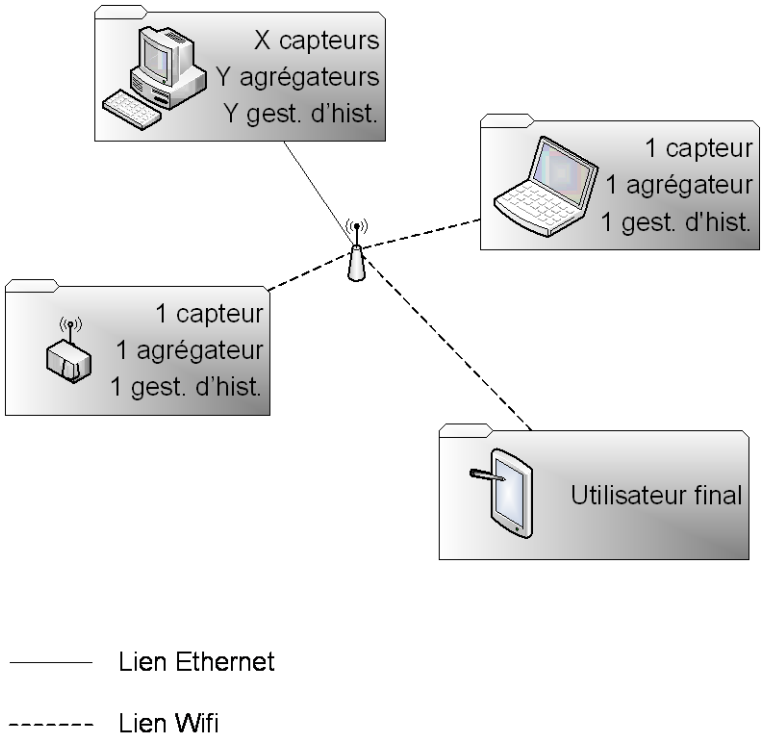

Figure 4.3 – Environnement de test

L'ordinateur de bureau est là pour simuler tous les nœuds du réseau qui n'ont pour but que de peupler le réseau et provoquer du trafic. Il s'agit d'un simple ordinateur de bureau :

- OS : Microsoft Windows XP Professionnel.
- Matériel : AMD Athlon XP 2600 + avec 2 Go de RAM.
- Connexion au point d'accès : Câble Ethernet à 100 Mbits/s.

L'ordinateur portable va nous permettre de faire tourner un Wireshark afin de mesurer la quantité d'information échangée entre les nœuds du réseau. Nous prendrons évidement bien soin de couper tout service pouvant provoquer du trafic réseau, de la découverte du réseau, aux mises à jour automatiques. N'ayant pas un contrôle total sur Jxta, la mise ne place de filtre de capture peut ne pas s'avérer optimal. Sa configuration est la suivante :

- OS : Microsoft Windows XP Professionnel installé sur disque externe USB 2.0.
- Matériel : Acer Travelmate 4100 contenant un Intel Pentium M 725 avec 2 Go de RAM.
- Connexion au point d'accès : Wifi G @ 54 Mbits/s.

Le capteur est un Stargate fabriqué par Crossbow. Il va nous servir principalement à évaluer le comportement d'un capteur dans un petit réseau. Nous allons principalement surveiller la consommation de la mémoire et mesurer le temps de réponse. Sa configuration est :

- OS : Linux
- Matériel : Stargate contenant un processeur 32 bits, 400 Mhz Intel PXA 255 XScale RISC avec 64 Mo de SDRAM, 32 Mo de mémoire Flash, et une clef USB de 4 Go.
- Connexion au point d'accès : Wifi B @ 11 Mbits/s.

Le Tablet PC va s'occuper de l'application de l'utilisateur final. C'est depuis cet ordinateur que nous interrogerons le réseau pour obtenir les valeurs. Le principe de fonctionnement de notre application de test sera décrit au point 4.3. La configuration de cet ordinateur est la suivante :

- OS : Windows Seven Intégral RC 32 bits
- Matériel : Tablet PC Asus R1E contenant un Intel Core 2 Duo Mobile T8300 avec 3 Go de RAM.
- Connexion au point d'accès : Wifi G @ 54 Mbits/s.

Enfin, dans un souci de simplification de configuration, nous avons pris le parti d'utiliser un point d'accès pour relier tous les éléments du réseau. Il est évident qu'une configuration en mode ad hoc aurait été plus appropriée au concept de notre architecture, cependant étant donné que nous travaillons au niveau application du modèle OSI, cela n'aurait rien apporté. Nous allons donc utiliser un routeur Netgear WGR615V. Il ne sera cependant utilisé que comme point d'accès et commutateur, les fonctionnalités de routeur ne serviront à rien puisque tous les nœuds du réseau seront dans le même sous-réseau.

4.2.2 Environnement des tests de la série 2

Les tests de la seconde série sont vont essentiellement nous permettre d'évaluer l'impact d'un de notre architecture en présence d'un certain nombre de nœuds. L'environnement de ces tests est décrit par la Figure 4.4.

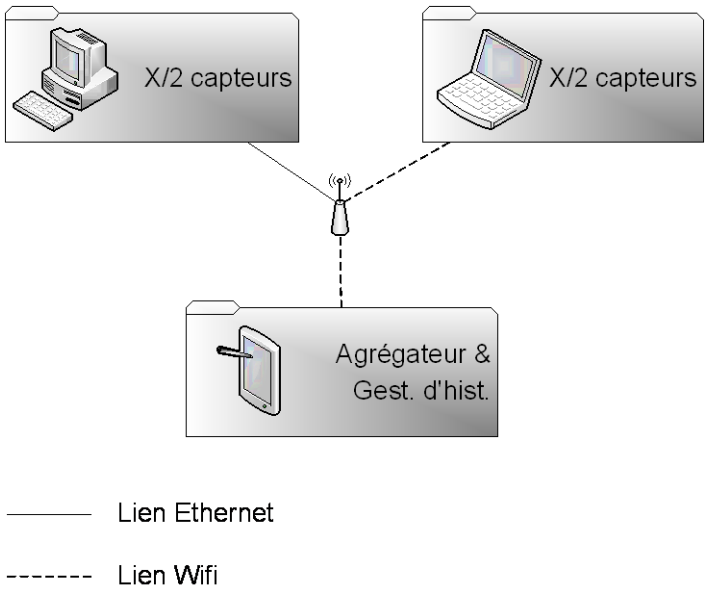

Figure 4.4 – Environnement des tests de la série 2

Le matériel utilisé est le même que pour la série 1.

4.3 Configuration de l'application et scénarios de tests

Dans notre implémentation les nœuds jouant le rôle de capteur ne vont pas faire de vraies mesures mais vont se limiter à générer aléatoirement des valeurs, y compris celui étant sur le capteur physique. Ils enverront ces valeurs toutes les 2 minutes, c'est à dire à une fréquence supérieure à ce qui est généralement configuré dans les réseaux de capteurs sans fils.

Durant nos tests, nous nous assurerons de toujours avoir suffisamment d'agrégateurs et de gestionnaires d'historique. Pour cela, nous adapterons le nombre de

capteurs dont un agrégateur peut s'occuper. De plus nous co-localiserons les agrégateurs et les gestionnaires d'historique sur des même nœuds afin de s'assurer de ne pas manquer de gestionnaire d'historique. À un agrégateur correspondra donc un gestionnaire d'historique.

Enfin, nous n'utiliserons pas d'algorithme d'optimisation des grappes. La sélection de l'agrégateur ou du gestionnaire d'historique ne sera basée que sur le succès ou l'échec de l'enregistrement.

L'application de l'utilisateur final, lorsqu'elle sera présente (première série de tests) sera configurée de manière à récupérer l'ensemble des valeurs actuelles des capteurs dès sa connexion au réseau, puis à récupérer les mises à jour toutes les 5 minutes. De plus l'utilisateur final établira une connexion directe avec les capteurs / agrégateurs du Stargate et du PC portable.

4.4 Résultats des tests

Comme nous l'avons annoncé, l'ensemble des tests que nous avons effectués n'ont pas pour but de comparer les performances entre notre architecture et une autre. Elle a juste pour but de tester succinctement le comportement de notre architecture et de son implémentation dans un petit réseau de capteurs sans fils.

4.4.1 Temps de réponse

Nous avons voulus mesurer le temps de réponse à une requête émise par l'utilisateur final, vers un gestionnaire d'historique, ou envoyée directement à un capteur. Cependant compte tenu de l'architecture le temps de réponse peut varier de quelques milli secondes à plusieurs secondes pour une même requête. Ces tests ne nous permettent pas de tirer de conclusion intéressante, nous ne les présenterons donc pas.

En effet, les requêtes sont envoyées par SIP au gestionnaire d'historique qui va alors se charger de faire la recherche dans base de données distribué formée par les

caches des différents gestionnaires d'historique. Mais cette recherche est effectuée au moyen de la découverte des annonces distantes décrites dans l'Annexe 1, cependant cette découverte dépend exclusivement de l'intergiciel et il nous est impossible de le contrôler. Ainsi si la réponse est « proche » du gestionnaire d'historique utilisé pour la recherche, la réponse sera rapide, et inversement.

Dans le cas d'une requête faite directement à un capteur, le problème est le même, sauf que le temps dévolu à la détection du capteur (*discover* de *peer advertisement*) est incontrôlable.

La seule conclusion que nous pouvons tirer de nos essais, est que nous n'avons pas relevé d'augmentation notable du temps de réponse quand il y'avait plus de nœuds dans le réseau.

4.4.2 Trafic réseau

Nous avons voulu observer le trafic réseau généré par notre architecture. En effet les échanges pairs à pairs ne sont généralement pas connus pour être optimaux, tout au moins sur l'internet. De plus comme les interfaces réseau sont gourmandes en énergie par rapport aux ressources d'un capteur, il est important de garder un œil sur le trafic réseau.

Pour cela nous avons fait tourner un Wireshark sur l'ordinateur portable, qui hébergeait un capteur, un agrégateur et un gestionnaire d'historique. La Figure 4.5 montre le nombre de paquets échangés. La Figure 4.6 représente le débit moyen généré par les échanges durant les tests. Enfin les figures Figure 4.7, Figure 4.8 et Figure 4.9 permettent de visualiser le trafic en fonction du temps durant les trois tests.

Afin d'obtenir des résultats plus significatifs, nous avons activé tous les gestionnaires d'historique, sans tenir compte du nombre de gestionnaires déjà actifs dans le réseau. En effet, cette façon de faire va nous permettre de mieux répartir la charge entre les gestionnaires. Cette gestion serait normalement dévolue au gestionnaire de

grappes, mais nous n'en avons pas mis en place. Nous n'observerons cependant pas le trafic entre un agrégateur et un gestionnaire d'historique puisqu'ils sont sur la même machine.

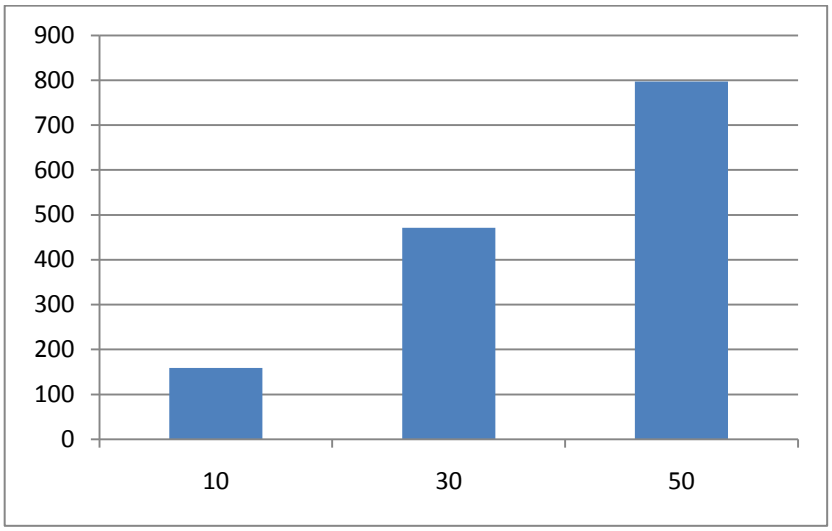

Figure 4.5 – Nombre de paquets échangés en 10 minutes par l'ordinateur portable en fonction du nombre de capteurs présent dans le réseau

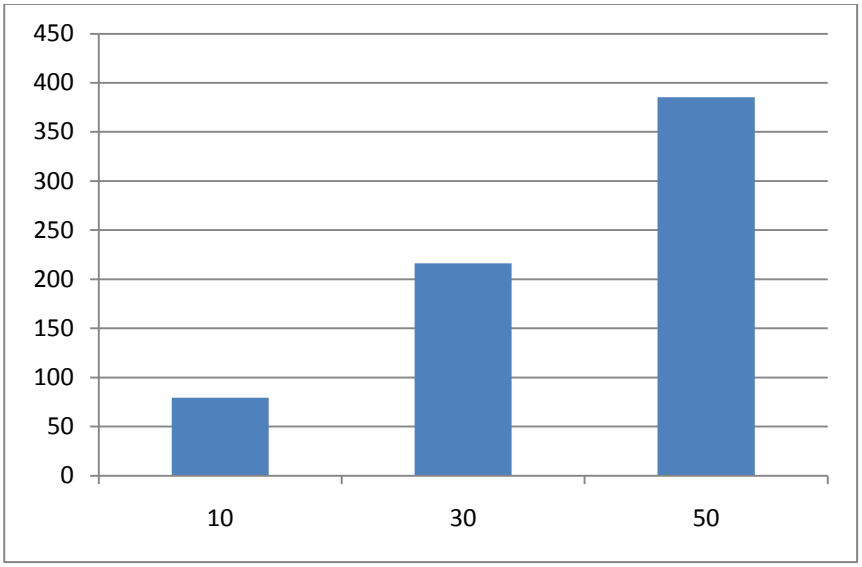

Figure 4.6 – Débit moyen en octets par secondes en fonction du nombre de capteurs

C’est donc sans surprise que nous voyons que le trafic généré sur un nœud qui joue les rôles d’agrégateur et de gestionnaire d’historique est directement proportionnel au nombre de capteurs présents dans le réseau. Cela est logique puisque nous avons cherché à répartir les capteurs sur l’ensemble des agrégateurs. Or le volume des données générées est intrinsèquement lié au nombre de capteurs dans le réseau. Nous pouvons cependant apprécier le fait que la quantité de données échangées n’augmente pas de manière exponentielle avec le nombre de capteurs présents.

Observons plus précisément les données échangées durant le période du test :

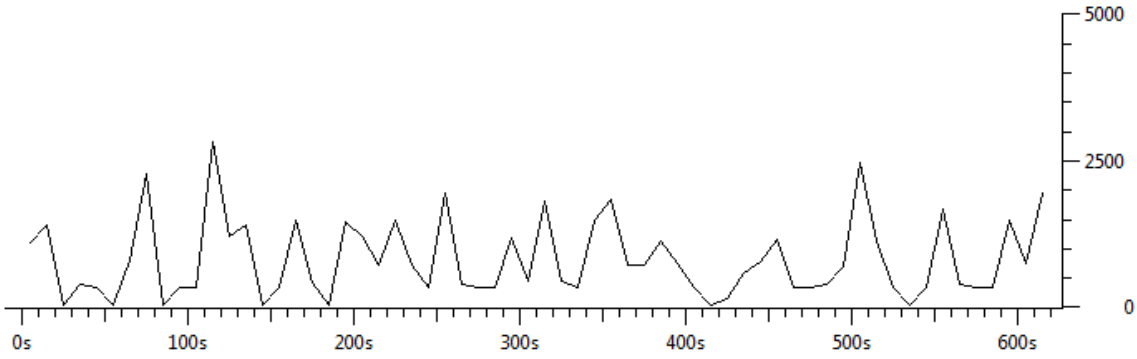

Figure 4.7 – Octets échangés par l’ordinateur portable en fonction du temps avec 10 capteurs (échantillonnage : 10 secondes)

Le taux d’échantillonnage trop important fait apparaître des sommets et des creux, mais globalement la moyenne des octets échangés reste relativement constante. Ainsi une fois que tous les capteurs ont rejoint le réseau, on n’a pas de croissance significative de l’activité sur le réseau, ce qui est encourageant.

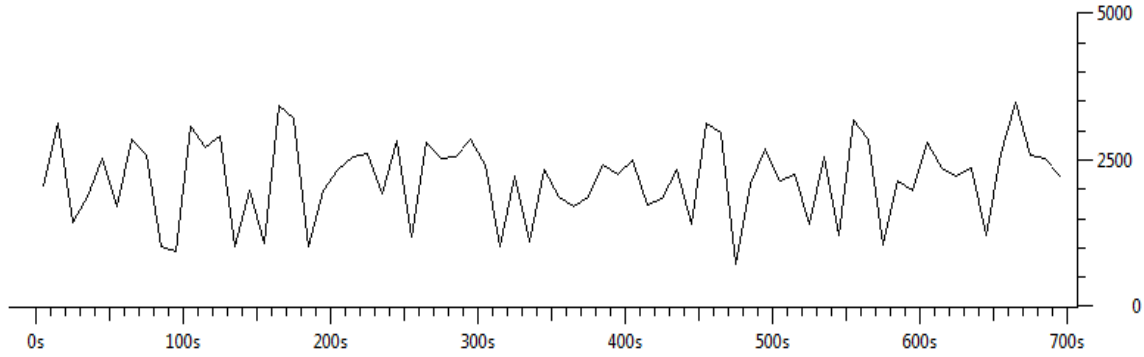

Figure 4.8 - Octets échangés par l'ordinateur portable en fonction du temps avec 30 capteurs (échantillonnage : 10 secondes)

Ici encore le taux d'échantillonnage trop grand fait apparaître des pics et des creux, mais globalement le nombre d'octets reste constant. La moyenne est logiquement plus élevée que pour les 10 capteurs, comme l'avait fait apparaître la Figure 4.6.

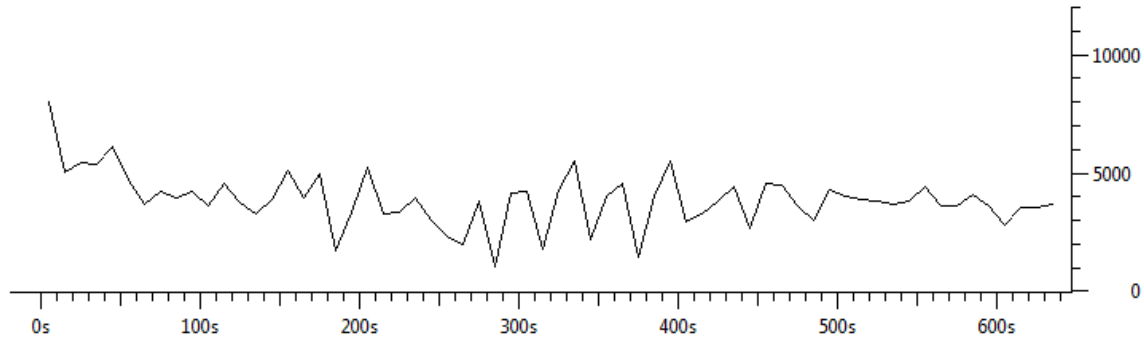

Figure 4.9 – Octets échangés par l'ordinateur portable en fonction du temps avec 50 capteurs (échantillonnage : 10 secondes)

Nous constatons donc que le trafic est globalement stable dans les trois cas. À noter tout de même un léger pic au démarrage des mesures sur la Figure 4.9. Cela est dû au fait que lorsqu'un nœud rejoint le réseau il provoque du trafic supplémentaire pour se déclarer (au niveau de l'intergiciel, et au niveau de notre architecture). Bien que les conditions dans lesquelles nous avons réalisé les deux graphiques précédent soient les mêmes, on ne retrouve pas de pic semblable. On peut supposer que cela est dû au fait

que s'annoncer sur un réseau de 50 capteurs prend plus de temps et génère plus de trafic que sur un plus petit réseau. On notera tout de même qu'une fois cette phase passée, dans les trois configurations, le trafic généré reste relativement constant : les autres pics et creux ne sont dûs qu'à un taux d'échantillonnage trop élevé que nous n'avons pas pu réduire à cause d'une limitation de Wireshark. Nous noterons d'ailleurs que chaque pic est accompagné d'un creux.

On se doit cependant de nuancer ces résultats en notant le fait que dans ces mesures n'est pas pris en compte le trafic dû à une l'utilisation d'un réseau ad hoc. De plus nous n'avons déplacé aucun nœud dans le réseau Or le mouvement d'un nœud provoque du trafic à tous les niveaux du réseau.

4.4.3 Utilisation de la mémoire

Une des ressources limités sur les capteurs est l'espace de stockage Il est donc important de surveiller la place nécessaire à la sauvegarde de l'historique. La Figure 4.10 représente la taille de l'historique dans trois configurations.

Pour tracer ce graphique, nous avons fait trois tests. L'un où nous avons simulé 10 capteurs, un avec 30 et un avec 50 capteurs. Ils sont tous enregistrés auprès d'un seul et unique agrégateur, lui-même connecté à un seul gestionnaire d'historique. Cette configuration est évidement une mauvaise utilisation de l'architecture, mais elle va nous permettre d'estimer la taille de l'historique en fonction de la taille du cache du nœud jouant le rôle de gestionnaire d'historique.

Bien entendu la taille de ce cache / historique va dépendre de la durée de vie accordée à chaque valeur publiée par un capteur, du contenu d'une valeur, etc. Dans notre cas, les valeurs sont renouvelées toutes les deux minutes et l'historique ne se compose que d'une seule annonce de valeur pour chaque capteur.

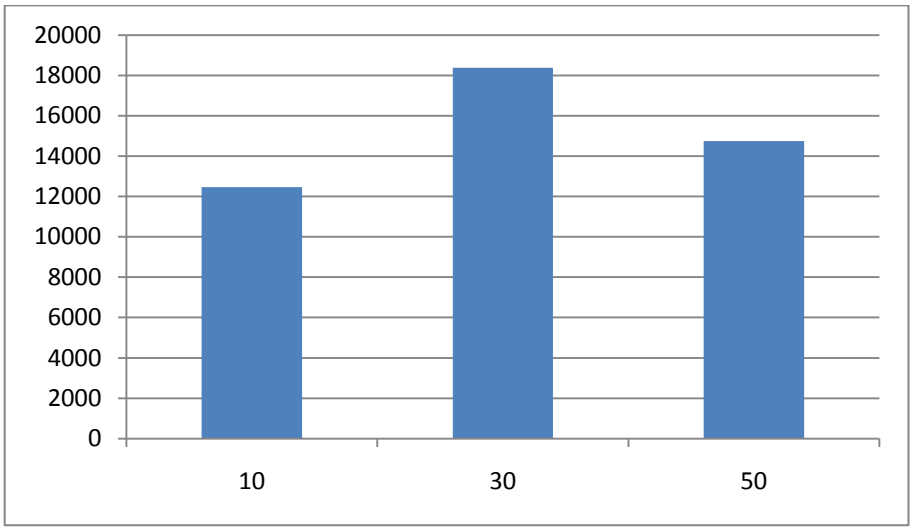

Figure 4.10 – Taille de l'historique en octet
en fonction du nombre de capteur dans le réseau

C'est avec une certaine surprise que nous avons constaté une diminution de la taille de ce cache lors de simulation de 50 capteurs. Il est d'ailleurs intéressant d'étudier la courbe d'évolution de la taille du cache durant ce test. C'est ce qui est représenté sur la Figure 4.11.

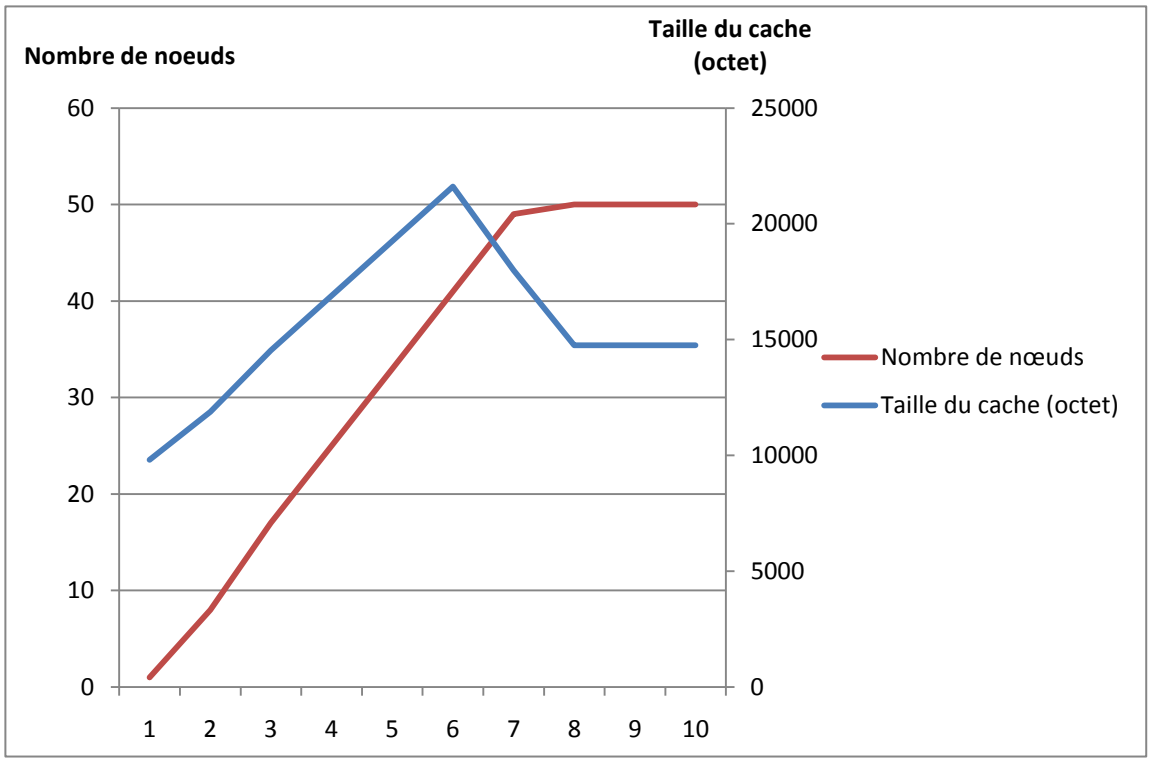

Figure 4.11 – Evolution de la taille du cache et du nombre de nœuds dans le réseau en fonction du temps en minutes

On remarque qu'une fois les 20 Ko atteints, la taille diminue pour se stabiliser à 14,4 Ko. En étudiant le contenu on remarque que des annonces utilisées par l'intergiciel Jxta disparaissent, alors que dans les simulations avec moins de capteurs, qui ont durées pourtant plus longtemps, il n'y a pas eu de diminution. L'intergiciel a semble-t-il volontairement supprimé certaines annonces inutiles afin de limité la taille du cache. Il est cependant évident que la taille du cache va recommencer à croître si on ajoute encore plus de capteurs. En effet s'il a supprimé une partie des annonces nécessaires à son propre fonctionnement, il ne pourra pas prendre moins de place que les valeurs que nous stockons dans ce gestionnaire d'historique.

4.5 Interprétation des résultats et conclusion

L'ensemble de nos tests ne nous permet pas de tirer des conclusions précises et définitives. Ils nous permettent cependant de tirer quelques conclusions d'ordre général à propos de notre architecture et des considérations à prendre en compte lors d'une implémentation éventuelle.

Tout d'abord, on remarque que la présence d'un certain nombre de capteurs dans le réseau ne semble pas provoquer un trafic de signalisation nettement supérieur, excepté la séquence d'initialisation. En effet l'augmentation du trafic est proportionnelle au nombre de nœuds que nous ajoutons dans le réseau. Cela signifie que notre architecture est utilisable aussi pour un grand nombre de nœuds.

Il semble cependant primordial de choisir un bon algorithme de mise en grappe. En effet, sur nos résultats n'apparaît pas le trafic généré par l'utilisation d'une architecture ad hoc au lieu d'une architecture avec infrastructure. Si les grappes ne sont pas optimales et maintenues à jour correctement, on va avoir un trafic inutile qui va dégrader les performances. Dans ces conditions, comme les valeurs n'auront plus besoin de traverser tout ou partie du réseau pour revenir à un puits, il est raisonnable de penser que le trafic réseau sera équivalent voire moindre que dans une architecture plus traditionnelle.

On notera aussi que des optimisations sont possibles côté réseau. En effet nous avons fait le choix d'utiliser SIP sur TCP en créant une connexion pour chaque message envoyé. C'est loin d'être optimal. L'utilisation d'UDP réduirait probablement considérablement le nombre de messages circulant sur le réseau.

Pour la gestion de l'historique nous pouvons conclure qu'il ne devrait pas être excessif, s'il est bien configuré, puisque les données nécessaires à la gestion de l'intergiciel ne prennent pas une place trop importante et sont parfois même éliminées pour laisser de la place pour les autres données.

Ainsi si l'ensemble du réseau est bien configuré (notamment choix judicieux de l'algorithme de mise en grappes, et bonne gestion de la durée de vie et de la taille des données), notre architecture semble parfaitement à même de fonctionner, y compris sur des réseaux comportant un nombre important de capteurs.

CHAPITRE 5 CONCLUSION

Dans ce mémoire nous avons proposé une solution pour accéder aux données d'un réseau de capteurs grâce à une surcouche réseau pair à pair. Cette architecture permet notamment de se passer de puits et permet une connexion directe entre les capteurs et les utilisateurs finaux qui utilisent les valeurs collectée par les capteurs. Cette architecture offre de grands avantages notamment dans des contextes de mobilité des utilisateurs et des capteurs.

5.1 Synthèse des travaux

La première partie de ce mémoire a été consacrée à l'étude de l'état actuel des choses, de ce qui fait la force des réseaux de capteurs sans fils, de leurs inconvénients. Nous avons aussi cherché à voir dans quel monde évoluent les réseaux de capteurs sans fils, et quelles sont les solutions qui ont déjà été proposées pour améliorer leurs performances.

Nous avons ensuite proposé une nouvelle architecture sans puits qui permet d'utiliser les réseaux de capteurs sans fils grâce à une surcouche pair à pair permettant, entre autre, un accès direct entre les utilisateurs finaux et les capteurs, tout en offrant la possibilité de préserver les ressources des réseaux capteurs grâce à un système d'historique. Nous nous sommes basés sur des analyses de situations concrètes dans lesquelles les réseaux de capteurs organisés selon une architecture traditionnelle arrivaient mal à remplir leurs missions. De ces analyses nous avons extrait les exigences

auxquelles notre architecture devrait se conformer. Enfin, à partir des ces exigences et d'hypothèses réalistes, nous avons construit notre architecture.

Nous avons alors implémenté et testé notre architecture afin d'en vérifier la faisabilité et le bon fonctionnement. Nous n'avons implémenté que partiellement notre architecture puisque une implémentation complète aurait demandée énormément de temps. Nous nous sommes donc limités aux fonctionnalités essentielles et importantes de l'architecture, aux concepts originaux tels la surcouche pair à pair. Nous n'avons pas non plus cherché à faire la meilleure implémentation possible, mais juste une implémentation fonctionnelle nous permettant de vérifier les principaux concepts de notre architecture.

Enfin, nous avons profité de cette implémentation partielle pour faire des tests sommaires de performances de l'architecture. En effet les capteurs ont généralement des capacités très limitées, notamment au niveau mémoire, des capacités de calcul, de la portée radio et surtout de la durée de vie de la batterie. Donc même si nous n'avons pas couvert tous ces domaines, nous avons cherché à estimer sommairement la qualité de l'architecture, et nous avons pu conclure que l'utilisation de notre architecture ne provoque pas de phénomène menant inévitablement à une défaillance généralisée du réseau.

Notre travail à permis de proposer une nouvelle approche pour la collecte des données sur les réseaux de capteurs. Ils ne sont plus réduits à un regroupement de capteurs individuels couvrant une zone, mais chaque capteur devient une entité du réseau, un acteur du réseau. Notre travaille ouvre la voie à de nouvelles possibilités et de nouveaux domaines d'applications pour les réseaux de capteurs. De plus, en se basant sur une architecture pair à pair, notre travaille libère les réseaux de capteurs sans fils de toute infrastructure, rendant totalement mobile l'ensemble du réseau. Cela ouvre la porte à des utilisations 100% mobiles, ce qui était encore impossible jusqu'à présent, à moins de recourir à des super-nœuds nécessitants des dispositions particulières.

5.2 Limitations des travaux

Parmi les limitations notables de nos travaux nous pouvons distinguer deux principales catégories. Tout d'abord les limitations d'ordre architectural, puis les limitations d'ordre plus technique.

D'un point de vue architectural, nous noterons notamment l'hypothèse que nous avons faite selon laquelle l'ensemble des nœuds du réseau devront partager la même pile réseau. Si cela est généralement vrai pour un seul réseau de capteurs, il existe une multitude de type de réseaux de capteurs, chacun adoptant la pile réseau de son choix, rendant donc impossible toute collaboration entre des capteurs de plusieurs fabricants. Mais surtout cela est n'est généralement pas le cas entre un capteur et un utilisateur final. Cela peut être corrigé simplement avec une normalisation des réseaux de capteurs.

Pour les limitations d'ordre technique on notera principalement le fait que pour notre implémentation nous avons dû avoir recours à un capteur des plus puissants du marché. En effet pour des capteurs plus petits, même si cela ne semble pas totalement inenvisageable, l'implémentation demanderait tout de même un effort extrêmement important. Enfin, pour ce qui est des tests que nous avons fait, nous n'avons pas testé l'impact sur la batterie, ce qui est souvent l'élément le plus crucial des réseaux de capteurs. Cependant ces aspects devraient être résolus dans les années à venir grâce à l'émergence de nouvelles technologies, à commencer par les nanotechnologies.

5.3 Orientation de recherches futures

Ce travail nous a permis de mettre en place une architecture pair à pair, et de faire certains tests, cependant fautes de moyens et de temps, nous n'avons pas pris soin d'analyser les performances par rapport à d'autres architectures ou encore dans différentes conditions. Il reste donc encore un travail d'analyse et de tests avant de pouvoir conclure que l'architecture est parfaitement utilisable. Parmi les tests indispensables on pourra noter :

- Dans un réseau avec de nombreux capteurs ;
- En provoquant la mobilité des nœuds ;
- Sur une longue durée.

Un autre domaine de recherche possible serait l'intégration de notre architecture au domaine particulier des réseaux véhiculaires. En effet, nous avons proposé comme scénario d'utilisation un système de prévention d'accident entre véhicules, cependant il existe des normes très strictes régissant les réseaux véhiculaires, normes que nous n'avons pas prises en compte. La question est donc de savoir s'il est possible d'utiliser notre architecture sur des réseaux de ce type, si notre architecture doit être modifiée pour y parvenir, ou bien si cela est tout simplement impossible.

Enfin, notre architecture ouvre la voie à un vaste domaine de recherche sur des systèmes d'accès pair à pair, et plus globalement à des solutions alternatives aux puits qui sont néfastes aux réseaux de capteurs sans fils.

RÉFÉRENCES

[1] Y. C. Hu, A. Perrig, and D. B. Johnson, "Ariadne: A secure on-demand routing protocol for ad hoc networks," *Wireless Networks,* vol. 11, pp. 21-38, 2005.

[2] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," *Ad Hoc Networks,* vol. 3, pp. 325-349, 2005.

[3] H. Füßler, J. Widmer, M. Käsemann, M. Mauve, and H. Hartenstein, "Contention-based forwarding for mobile ad hoc networks," *Ad Hoc Networks,* vol. 1, pp. 351-369, 2003.

[4] L. Hanzo and R. Tafazolli, "A survey of QoS routing solutions for mobile ad hoc networks," *IEEE Communications Surveys and Tutorials,* vol. 9, pp. 50–70, 2007.

[5] V. Pandey, D. Ghosal, and B. Mukherjee, "Pricing-based Approaches in the Design of Next-Generation Wireless Networks: A review and a unified proposal," *IEEE Communications Surveys & Tutorials,* vol. 9, pp. 88-101, 2007.

[6] M. Mauve, A. Widmer, and H. Hartenstein, "A survey on position-based routing in mobile ad hoc networks," *IEEE network,* vol. 15, pp. 30-39, 2001.

[7] G. Xylomenos, G. C. Polyzos, P. Mahonen, and M. Saaranen, "TCP performance issues over wireless links," *IEEE Communications Magazine,* vol. 39, pp. 52-58, 2001.

[8] A. Al Hanbali, E. Altman, and P. Nain, "A survey of TCP over ad hoc networks," *IEEE Communications Surveys & Tutorials,* vol. 7, pp. 22–36, 2005.

[9] D. Kliazovich and F. Granelli, "Cross-layer congestion control in ad hoc wireless networks," *Ad Hoc Networks,* vol. 4, pp. 687-708, 2006.

[10] C. Fu, R. Glitho, and F. Khendek, "A cross-layer architecture for signaling in multihop cellular networks," *IEEE Communications Magazine,* vol. 46, pp. 174-182, 2008.

[11] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "TinyLIME: bridging mobile and sensor networks through middleware," *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on,* pp. 61-72, 2005.

[12] R. Glitho, F. Khendek, N. Y. Othman, and S. Chebbine, "Web Services-Based Architecture for the Interactions between End-User Applications and Sink-Less Wireless Sensor Networks," *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE,* pp. 865-869, 2007.

[13] F. C. Delicato, P. F. Pires, L. Pirmez, and L. F. R. da Costa Carmo, "A flexible web service based architecture for wireless sensor networks," pp. 730–735, 2003.

[14] P. Ciciriello, L. Mottola, and G. P. Picco, "Efficient routing from multiple sources to multiple sinks in wireless sensor networks," *Lecture Notes in Computer Science,* vol. 4373, p. 34, 2007.

[15] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," *Proceedings of IPTPS02, Cambridge, USA,* vol. 1, pp. 2-2, 2002.

[16] "the_gdf : The Gnutella Developer Forum (GDF)," http://groups.yahoo.com/group/the_gdf/, 2009.

[17] B. Traversat, M. Abdelaziz, M. Duigou, J. C. Hugly, E. Pouyoul, and B. Yeager, "Project JXTA Virtual Network," *Sun Microsystems, Oktober,* 2002.

[18] A. Jere, M. Meza, B. Marusic, S. Dobravec, T. Finkst, and J. F. Tasic, "Peer to peer search engine and collaboration platform based on JXTA protocol," *The IEEE Region 8 EUROCON 2003. Computer as a Tool,* vol. 1, 2003.

[19] I. Sun Microsystems, "API Specifications," http://java.sun.com/reference/api/, 2009.

[20] M. D. Scheemaecker, "NanoXML - DEVKIX," http://devkix.com/nanoxml.php, 2009.

[21] I. Sun Microsystems, "JXTA Java Micro Edition Project," https://jxta-jxme.dev.java.net/, 2009.

[22] C. Blundo and E. De Cristofaro, "A bluetooth-based JXME infrastructure," *Lecture Notes in Computer Science,* vol. 4803, p. 667, 2007.

[23] I. Sun Microsystems, "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 180," http://jcp.org/en/jsr/detail?id=180, 2009.

[24] I. Sun Microsystems, "Getting Started with SIP API for J2ME (JSR 180)," http://developers.sun.com/mobility/apis/articles/sip/, 2009.

[25] Nokia, "Forum Nokia," http://www.forum.nokia.com/, 2009.

[26] "MjSip," http://www.mjsip.org/mjsipME.html, 2009.

[27] B. Traversat, A. Arora, M. Abdelaziz, M. Duigou, C. Haywood, J. C. Hugly, E. Pouyoul, and B. Yeager, "Project JXTA 2.0 super-peer virtual network," *Sun Microsystem White Paper. Available at www. jxta. org/project/www/docs,* 2003.

[28] I. Sun Microsystems, "JXTA v2.0 Protocols Specification," 2007.

[29] A. Heukmes, "2dconcept - JXTA," http://www.2dconcept.com/jxta.html, 2006.

ANNEXES

ANNEXE 1 INTRODUCTION À JXTA

1.1 Présentation de Jxta

Jxta est un projet open source développé par Sun, qui définit un certain nombre d'APIs permettant de facilement construire un réseau pair à pair. Ces APIs étant disponibles en Java, C++ et C#, les différentes versions étant bien entendu compatibles en elles.

À noter que Jxta est basé sur les standards du web comme TCP/IP, HTTP, et XML, et offre l'énorme d'avantage d'être conçu pour passer outre les restrictions qu'on peut retrouver sur de nombreux réseaux. Il permet notamment de passer outre les pare-feu (firewall) et les NAT (*Network Address Translation* – dispositif permettant de partager une seule adresse IP avec plusieurs machines).

1.2 Principe de fonctionnement de Jxta

Pour son fonctionnement Jxta définit pas moins de six protocoles différents, indépendants du langage de programmation, chacun ayant un rôle précis. Les principaux buts de ces protocoles sont :

- de permettre aux pairs de se découvrir entre eux ;
- d'auto organiser les pairs en groupes ;
- d'annoncer et découvrir des ressources ;
- de permettre les communications entre les pairs.

On distingue deux types de protocoles : d'un côté les protocoles qui servent pour le fonctionnement de base de Jxta et qui sont indispensables pour toute implémentation de Jxta ; et d'un autre coté les protocoles permettant d'offrir un certain nombre de services supplémentaires :

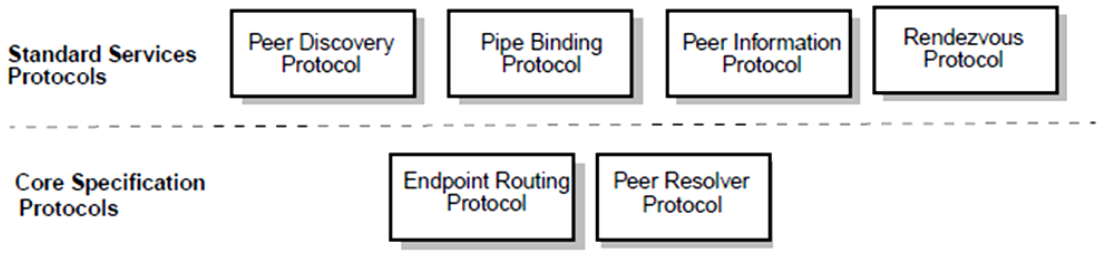

Figure A1.1 – Les protocoles du projet Jxta [27]

1.2.1 Protocoles à la base du réseau (*Core Specification protocols*)

Ils doivent être présents dans toute implémentation de Jxta, ils vont former la base du réseau.

- ***Endpoint Routing Protocol*** **(ERP) :**
  Ce protocole va servir à découvrir une route entre deux pairs du réseau.

- ***Peer Resolver Protocol*** **(PRP) :**
  Ce protocole va être utilisé pour envoyer des requêtes génériques à un ou plusieurs pairs

1.2.2 Protocoles additionnels (Standard Services protocols)

Ils ne sont pas indispensables au bon fonctionnement de Jxta et pourrait être omis pour une intégration dans un nœud ayant de très faibles ressources. Cependant il est très fortement recommandé de les implémenter pour assurer une certaine interopérabilité et des fonctionnalités minimales.

- ***Rendezvous Protocol*** **(RVP) :**
  Permet de propager les messages, il est notamment utilisé pour propager des messages au sein d'un groupe de pairs.

- ***Peer Discovery Protocol*** **(PDP) :**
  Il est utilisé pour publier ou recevoir des annonces (*advertisement*) de contenu.

- ***Peer Information Protocol*** **(PIP) :**
  Ce protocole sert à obtenir des informations à propos de l'état d'un pair du réseau.

- ***Pipe Binding Protocol*** **(PBP) :**
  Il permet de créer des canaux de communication entre les nœuds.

On peut alors représenter les interactions entre les protocoles par la Figure A1.2. À noter que les spécifications complètes des différents protocoles de Jxta sont décrites dans [28].

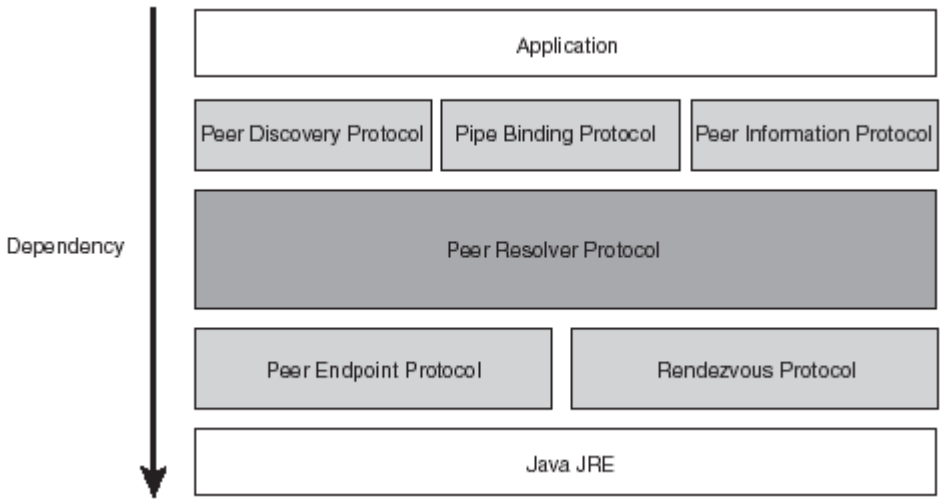

Figure A1.2 – Interactions entre les protocoles de Jxta [29]

1.3 Les principaux éléments de Jxta

Pour utiliser Jxta il est important de bien comprendre toute la terminologie utilisée. Voici donc un rapide aperçu des principaux éléments qu'on peut retrouver dans Jxta.

1.3.1 ID

Comme souvent en informatique, le simple fait de donner des noms aux objets n'est pas suffisant, qui plus est dans un réseau pair à pair où le nombre de nœud n'est pas préalablement connu. Jxta utilise donc des ID uniques pour tout identifier dans le réseau : pairs ; canaux de communications ; contenu ; spécifications de module ; classes de modules.

Ces ID sont construits selon le principe des URN (*Unified Resource Name*) défini par le RFC 2141. Les URN se différencient des URL (*Unified Resource Locator*) par le fait d'identifier une ressource indépendamment de la méthode pour y accéder, tandis que l'URL identifie un moyen d'accéder à une ressource. Les URN et les URL sont des URI (*Unified Resource Identifier*), c'est-à-dire une chaine de caractères permettant d'identifier une ressource.

Ainsi un ID dans Jxta ressemblera à çà :

```
urn:jxta:uuid-59616261646162614E504720503250337BFCFB5FB0884B4C8
827FDB1C5E13EA404
```

Figure A1.3 – Exemple d'UUID utilisé par Jxta

À noter qu'il y'a trois ID réservés : le NULL ID, l'ID du World Peer Group, et l'ID du Net Peer Group (groupes par défaut).

1.3.2 Les pairs (*peers*)

C'est un nœud du réseau P2P. Chacun aura son propre ID, on parlera alors de *peerID*. Associés à ces *PeerID* on aura des *endPoint* correspondant à des interfaces de communications pour un pair.

Il existe des pairs particuliers, appelés *super peer*, qui ont des fonctionnalités avancés. Ce sont les relais (*relay*) et rendez-vous. Ils sont notamment utilisés pour résoudre les problèmes réseaux comme les NAT, les pare-feu et les proxys.

1.3.3 Les groupes de pairs

Un groupe de pairs représente un regroupement de plusieurs pairs qui offrent un même service. A la connexion d'un pair sur le réseau, il rejoint automatiquement le *Net Peer Group*. Un pair peut faire partie de plusieurs groupes à la fois.

Pour les identifier un groupe on parlera de *peerGroupID*. Remarquons que chaque groupe peut définir sa propre politique d'accès. Elle peut prendre de multiples formes, d'aucune exigence particulière à l'utilisation de certificats.

Tous les groupes de pairs offrent un certain nombre de services de base, d'autres pouvant être ajoutés. Les services de base [29] :

- **Discovery Service :** Il servira à la recherche du contenu dans un groupe ;
- **Membership service :** Ce service va permettre de définir les options de sécurité du groupe et d'ensuite se connecter à un groupe sécurisé ;
- **Access Service :** Permet de valider l'accès d'un pair à un groupe ;
- **Pipe Service :** Ce service sera utilisé pour la création de canaux de communication ;

- **Resolver Service :** Il permet d'effectuer des requêtes pour les services de pairs ;
- **Monitoring Service :** Ce service permet aux pairs de surveiller d'autres pairs ou groupes de pairs.

1.3.4 Les annonces (*Advertisements*)

Les annonces sont au cœur du fonctionnement de Jxta. Elles permettent à un nœud de s'annoncer à ses pairs, mais aussi d'annoncer des ressources comme un groupe, un service ou encore un canal de communication. Elles sont publiées sur le réseau puis mises en caches par les pairs pour une durée pouvant être définie dans le corps de l'annonce.

Elles sont massivement utilisées par l'ensemble des protocoles de Jxta, on retrouve par exemple une large gamme d'*Avertissements* dans l'implémentation de Jxta (*Peer Advertisement, Peer Group Advertisement, Pipe Advertisement, Rendezvous Advertisement, Peer Info Advertisement, ...*) chacune permettant d'annoncer aux autres pairs une fonctionnalité particulière.

Les annonces dans Jxta, ce sont des petits arbres XML (ce qui impose l'utilisation d'un parseur XML pour utiliser Jxta) :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:PeerAdvertisement>
<jxta:PeerAdvertisement xmlns:jxta="http://jxta.org">
  <Name>David@CRONOS</Name>
  <PID>urn:jxta:uuid-
    59616261646162614E504720503250337BFCFB5FB0884B4C8</PID>
  <GID>urn:jxta:jxta-NetGroupId</GID>
  <Parm>
    <Addr> tcp://88.160.211.251:9721/ </Addr>
  </Parm>
```

```
</jxta:PeerAdvertisement>
```

Figure A1.4 – Exemple de contenu d'une annonce utilisée par Jxta

Ainsi après la création d'une entité (pair, canal de communication, etc.), la marche à suivre pour la rendre accessible aux autres nœuds du réseau, va être tout d'abord de créer une annonce à propos de cette entité, puis la publier soit dans le cache local, soit dans les caches distants (i.e. les caches des autres pairs). C'est sur ce processus de base que repose tout le réseau. Cela va être réalisé grâce au *Discovery Service* des groupes de pairs (voir 1.3.3).

Une fois publiée, les autres pairs peuvent retrouver cette annonce grâce au service de découverte (*discovery*). On va distinguer deux types de découvertes en fonction du domaine de recherche :

- **Local (`getLocalAdvertisements()`):**
  La découverte locale va consister à rechercher dans le cache local du nœud si une annonce correspond aux critères de recherche. Ce cache est rempli au fur et à mesure par les annonces reçus des autres pairs.
  Ce type de découverte va aussi rechercher sur les nœuds du groupe en *multicast* ou *broadcast* (donc limité au réseau local).

- **Distante (`getRemoteAvdertisements()`):**
  Chaque groupe a au moins un rendez-vous (le premier nœud qui se connecte à un groupe en devient automatiquement le rendez-vous). Le fait de faire une découverte distante va notamment consister à envoyer la requête de découverte à ces rendez-vous, permettant d'accéder à des pairs jusqu'alors inconnus, y compris sur l'internet.

Grâce à ce système d'annonces et de découvertes, on peut alors découvrir tous les pairs et les services qu'ils offrent. C'est la pierre angulaire du réseau pair à pair.

1.3.5 Les canaux de communication (*pipes*)

Pour la communication entre deux nœuds, Jxta utilise des tubes de communication similaires à ce qu'on peut retourner dans des systèmes UNIX. L'idée est comme dans le modèle OSI de permettre une abstraction des couches sous-jacentes lors de l'envoi de données. L'utilisateur envoie ses informations sans se soucier des contraintes des couches de plus bas niveau.

Les pipes utilisent la notion d'*endpoint* pour désigner les points d'entrée et de sortie d'un canal de communication. Il existe trois types de pipes :

- **Unicast :** unidirectionnel, non sécurisé et non fiable.
- **Unicast sécurisé :** unidirectionnel, sécurisé et non fiable.
- **Propagés :** tube de propagation, non sécurisé et non fiable.

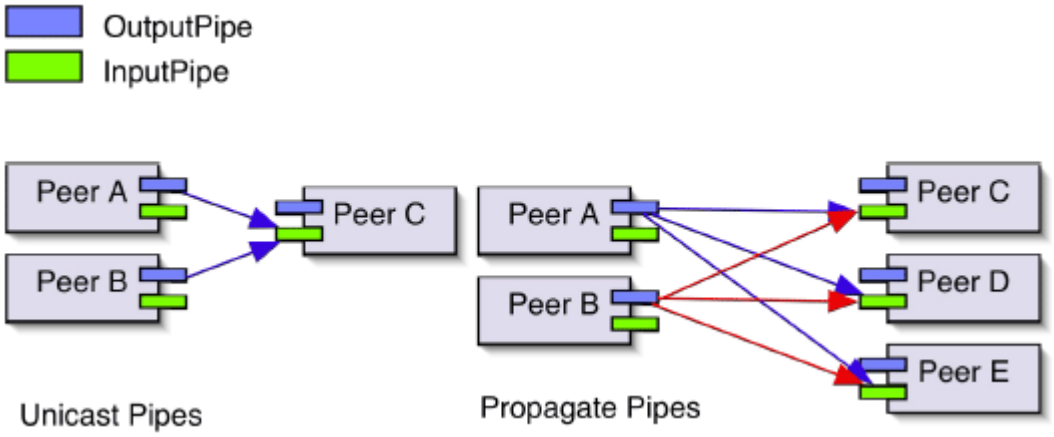

Figure A1.5 – Les tubes de communication de Jxta [29]

L'inconvénient de ces tubes est qu'ils sont unidirectionnels et non fiables, comparable à des datagrammes UDP. Cependant si on prend en compte qu'une très grande majorité du trafic sur l'internet se fait grâce à des flux TCP, il serait agréable de retrouver quelque chose de similaire avec Jxta, à savoir des canaux de communications

bidirectionnels et fiables. C'est dans cette optique qu'ont été développés les `JxtaServerSocket`, `JxtaSocket`, `JxtaMulticastSocket`, `JxtaServerPipe` et `JxtaBiDiPipe`. La Figure A1.6 représente le fonctionnement de ces entités. Les sockets présentent de plus l'avantage d'implémenter les interfaces classiques des sockets, et permettent donc de se retrouver dans un environnement familier.

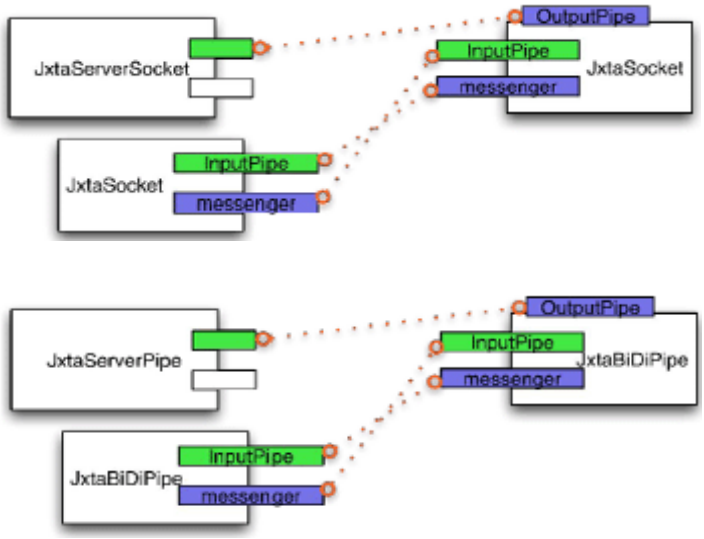

Figure A1.6 – Les canaux de communications bidirectionnels et fiables de Jxta [29].

ANNEXE 2 INTERFACE SIP

2.1 SipConnection

```
package common.sip;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.util.HashMap;
import java.util.Iterator;

import javax.microedition.io.StreamConnection;
import javax.microedition.sip.SipConnection;
import javax.microedition.sip.SipDialog;
import javax.microedition.sip.SipException;

public abstract class SipConnectionGlue implements SipConnection {

        protected StreamConnection SC;
        protected HashMap headers;
        protected String method;
        protected int statusCode = 0;
        protected int responseCode = 0;

        private OutputStream OS = null;
        private InputStream IS = null;

        public static final byte CREATED = 1;
        public static final byte REQUESTRECEIVED = 2;
        public static final byte INITIALIZED = 3;
        public static final byte STREAMOPEN = 4;
        public static final byte PROCEEDING = 5;
        public static final byte COMPLETED = 6;
        public static final byte UNAUTHORIZED = 7;
```

```
	public static final byte TERMINATED = 8;

	protected SipConnectionGlue() {
		this.headers = new HashMap(5);
	}

	public void addHeader(String key, String value) throws SipException,
IllegalArgumentException { /* .. */ }
	public String getHeader(String key) { /* .. */ }
	public String getMethod() { /* ... */ }
	public int getStatusCode() { /* ... */ }
	public void removeHeader(String key) throws SipException { /* ... */ }

	public InputStream openContentInputStream() throws IOException, SipException {
		if (statusCode != PROCEEDING && statusCode != COMPLETED && statusCode !=
REQUESTRECEIVED)
			throw new SipException(SipException.INVALID_STATE);
		return this.getInputStream();
	}

	public OutputStream openContentOutputStream() throws IOException, SipException {
		if (statusCode != INITIALIZED)
			throw new SipException(SipException.INVALID_STATE);
		send();
		this.statusCode = STREAMOPEN;
		return this.getOutputStream();
	}

	public void send() throws IOException, SipException {
/*	WARNING	!	contrairement	à	ce	qui	est	défini	ici	:
http://mobilezoo.biz/jsr/180/javax/microedition/sip/SipClientConnection.html	j'accepte
aussi l'état STREAMOPEN pour le receive() et je le bloque dans le send() ! Sinon on ne
peut pas coller à l'exemple de code donné sur la même page qui consiste à envoyer sur le
stream et directement faire un receive ! */
		if (statusCode != INITIALIZED /*&& statusCode != STREAMOPEN*/ &&
(statusCode!=COMPLETED && responseCode >= 200 && responseCode <300))
			throw new SipException(SipException.INVALID_STATE);
		System.out.println("Envoi d'un message SIP method = "+method+" /
responseCode = "+responseCode);
		OutputStreamWriter OSW = new OutputStreamWriter(this.getOutputStream());
		OSW.write(this.toString());
		OSW.flush();
		this.statusCode = PROCEEDING;
	}
```

```
	public void setHeader(String key, String value) throws SipException,
IllegalArgumentException { /* ... */ }
	public void close() throws IOException {
		this.statusCode = TERMINATED;
		System.out.println("Fermeture d'une connexion pour un "+method+" /
"+responseCode);
		SC.close();
	}
}
```

2.2 SipServerConnection

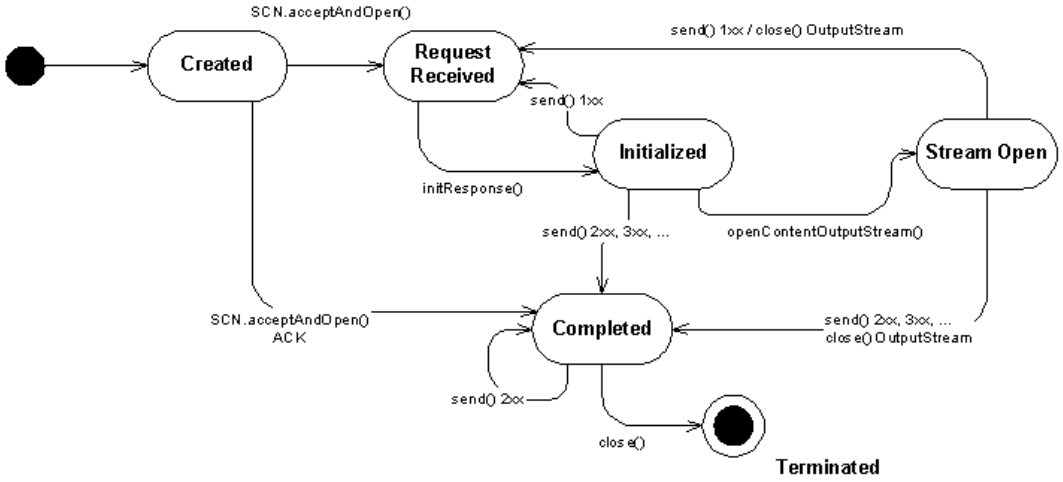

Figure A2.1 – Diagramme d'état du fonctionnement de la classe SipServerConnection [23]

2.3 SipClientConnection

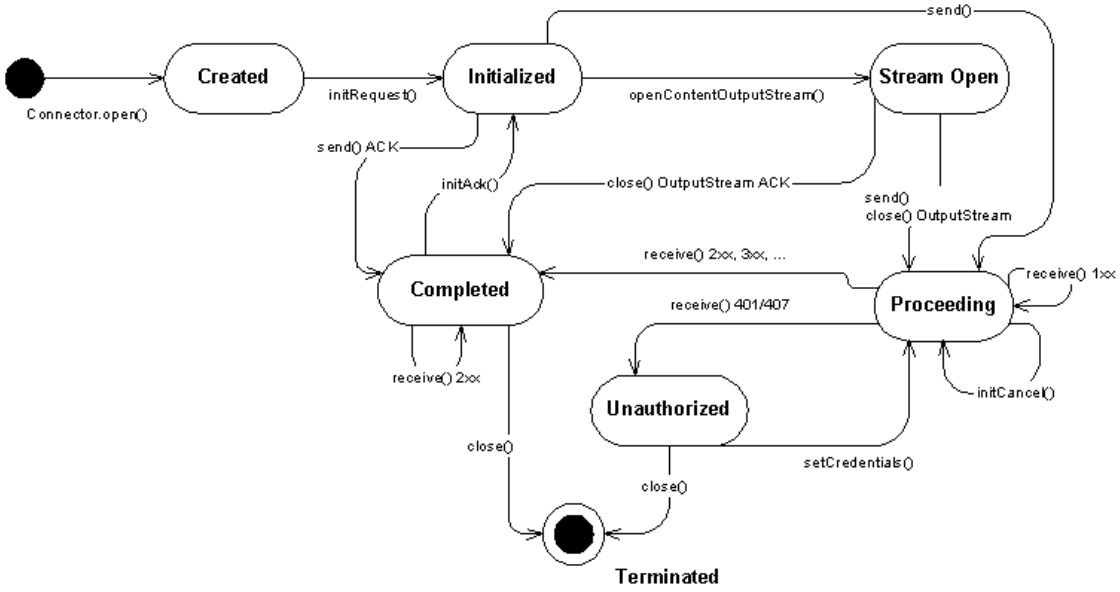

Figure A2.2 – Diagramme d'état du fonctionnement de la classe SipClientConnection [23]