ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Projet I de semestre

Protocol Composition Frameworks for Mobile Ad-Hoc Networks

Projet réalisé en juin 2003 au Laboratoire de systèmes répartis par
René Giller (rene.giller@epfl.ch)
Supervisé par David Cavin
Sous la direction de André Schiper

1. Tables des matières

2. Introduction

Dans tout ce document, les mots en **gras rose** représentent des classes, les mots en *italique bleu* (excepté les titres) représentent des événements, les mots en *italique vert* représentent des variables et les mots en *italique rouge* (suivis de parenthèses) représentent des méthodes.

2.1. Description du projet

Initialement, le projet consistait à explorer les trois frameworks que sont Cactus, Neko et Appia afin de définir si ils étaient ou non utilisables pour le développement d'applications avec des réseaux ad-hoc. Puis, venait le développement, dans les trois frameworks, d'une couche de communication permettant d'envoyer et recevoir des données en *UDP multicast*. Enfin, il fallait établir, toujours dans les trois frameworks, le schéma d'une pile incluant la couche de communication ainsi que des couches de flooding et de réseaux virtuels. L'interface pile – application n'étant pas définie. Mais, suite à quelques problèmes (mauvais release disponible, impossibilité de trouver les fichiers de configuration) lors de l'exploration de Neko, il a été décidé de laisser tomber ce framework. A la place, l'un des framework (Appia) a été développé selon le schéma déjà établi. Une fois cela terminé et fonctionnel, une couche de "reliable point to point" du travail de semestre de Jean Vaucher (LSR, décembre 2002) a été reprise et adaptée aux couches déjà existantes. Le but étant de pouvoir utiliser, avec ce programme, les applications déjà codées en Appia par le LSR. L'interaction avec ces applications étant déjà fixée, le code de cette nouvelle couche a dû être adapté en conséquence.

2.2. Méthode de travail

Le travail à effectuer a été séparé en phases décrites ci-après. Une nouvelle phase ne commençant qu'une fois la phase précédente terminée. Sauf indication contraire, les framework sont développés l'un après l'autre jusque au point 2.2.4. Les phases suivantes ne sont effectuées qu'avec Appia et ne suivent plus l'ordre décrit.

2.2.1. Exploration du framework

Dans cette phase, il s'agit de parcourir la documentation, le code source ainsi que quelques publications liées au framework afin de comprendre son fonctionnement, ses avantages et inconvénients. A la fin de cette phase, plus ou moins longue selon la quantité et la qualité des informations disponible, il s'agit de déterminer si l'utilisation avec les réseaux ad-hoc est possible. Le premier framework exploré (sommairement) est FRANC (anciennement MANET), développé justement pour les réseaux ad-hoc par le LSR en décembre 2002. Ce framework n'a jamais fait parti des tâches de ce projet, il a été étudié juste pour avoir faire une idée du produit à fournir.

2.2.2. Codage d'une couche

Il s'agit simplement de définir un cahier des charges, c'est-à-dire ce qui est demandé à la couche. Puis de coder, tout en respectant ce cahier des charges. Cette phase ne prend fin que lorsque le code compile.

2.2.3. Test d'une couche

En premier lieu, une couche d'application, en fait un chat qui a été réutilisé pendant tout le reste du projet, a été écrite. Celle-ci a quelque peu évoluée avec les différents frameworks, mais est restée basiquement la même. Elle permet d'envoyer un String en broadcast et d'afficher, avec leur expéditeur, les String reçus. Puis, les tests à effectuer afin de certifier la couche ont été déterminés. Lorsque les tests sont écrits, la couche est ajoutée à la pile constituée des couches déjà certifiées avec l'application au sommet. Puis les tests

préétablis sont lancés sur le réseau de l'EPFL. Une fois tous les tests positifs, la couche est certifiée et reçoit un numéro. A chaque modification d'une couche, elle repasse tous les tests et le numéro est incrémenté, ce qui permet de garder une trace des anciennes versions d'une couche. Cette technique garanti qu'une erreur, quelque soit l'endroit où elle est levée, provient de la seule couche en phase de test. Si une couche ne passe pas un test, des *System.out.println()* sont ajoutés aux endroits stratégiques du code afin d'imprimer les informations pertinentes, ce qui permet de localiser et corriger l'erreur assez rapidement.

2.2.4. Réalisation du schéma

Cette phase est purement théorique, elle consiste simplement a réaliser un schéma décrivant une pile contenant la couche déjà codée plus des couches de flooding (**Flooding**) et de réseaux virtuels (**VirtualNetworks**). Ce schéma n'est pas définitif, il évolue en fonction des améliorations apportées au programme. Il se peut aussi qu'une meilleure architecture soit découverte en cours de développement.

2.2.5. Codage du schéma

Une fois arrivé au point 2.2.4 avec Cactus et Appia (Neko étant déjà éliminé), il faut en choisir un qui sera développé selon le schéma. Les deux frameworks semblent fonctionner aussi bien l'un que l'autre, mais Appia, bien que moins commenté, est plus conventionnel, donc plus facile à maîtriser. Il a été choisit pour cette raison. Retour au point 2.2.2 puis 2.2.3 avec Appia afin d'intégrer ces deux nouvelles couches. Une fois tous les tests positifs pour toute la pile, passage au point 2.2.6.

2.2.6. Intégration d'une couche "reliable point to point" (RPt2Pt)

Cette couche est partiellement reprise du projet de semestre de Jean Vaucher (LSR, décembre 2002), supervisé par Sergio Mena. Son rôle est de mettre en place un système d'acquittement des messages afin de rendre la transmission fiable. En plus de cette tâche, elle s'occupe aussi de la traduction des événements que j'ai déjà codés avec ceux déjà développés et imposés par le LSR et vice-versa. Pour le développement de cette couche, le pseudo code ainsi qu'un code incomplet dont les méthodes utiles ont été extraites et remaniées, étaient disponibles. Une fois le codage terminé, retour au point 2.2.3.

2.2.7. Conclusion

Concrètement, en Appia, trois couches : **Flooding**, **VirtualNetworks** et **RPt2Pt** ainsi que l'application de test (**Application** et **Adaptor**) qui va avec ont été développées. La couche **UDPSimple**, fournie avec le framework, a été reprise sans lui apporter de modifications. Finalement, on obtient une pile, compatible avec les couches existantes du LSR, garantissant (dans la mesure des textes effectués) un transport point à point fiable des données avec *UDP multicast*, tout en simulant plusieurs réseaux.
En Cactus, une seule couche a été développée : **CommunicationLayer** plus l'application de test (**Application** et **Adaptor**) qui va avec. Cependant, le schéma permettant d'intégrer les couches **Flooding** et **VirtualNetworks** est déjà développé.
En Neko, rien n'a été codé, le point2.2.1 n'a pas été dépassé.

3. Les frameworks

3.1. Cactus

3.1.1. Introduction

Développé par le département d'informatique (Computer Science Department) de l'université d'Arizona, USA. Versions existantes en java, C et C++. Référence : http://www.cs.arizona.edu/cactus/. Cactus peut être vu comme une extension de x-kernel.

3.1.2. Architecture

Cactus a de multiples threads et dispose de deux niveaux de composition : coarse-grain et fine-grain.

- Coarse-grain (fig. 1): Les entités sont appelées *Protocol*. Elle sont hiérarchiques et communiquent par le moyen de messages grâce aux méthodes *sendUp()* / *fromBelow ()* et *sendDown()* / *fromAbove()*. Avant de commencer à transmettre des messages, un *Protocol* doit se coupler à celui d'en dessous. Ensuite, la communication a lieu dans les deux sens, c'est-à-dire que la méthode *sendUp()*, respectivement *sendDown()* de l'un appelle la méthode *fromBelow()*, respectivement *fromAbove()* de l'autre. Plusieurs *Protocol* peuvent se coupler au même et un *Protocol* peut se coupler à plusieurs autres. Les messages peuvent être envoyés sur le réseau.
- Fine-grain (fig. 2): Les entités sont appelées *Microprotocol*. Elle ne sont pas hiérarchiques et communiquent par le moyen d'événements et de variables partagées. Il peut y avoir plusieurs *Microprotocol* par *Protocol*. Les *Protocol* et *Microprotocol* doivent s'enregistrer auprès d'un événement pour le recevoir. Un événement ne peut être levé que si toutes les entités enregistrées chez lui ont appelés la méthode *signal()*. Les événements sont locaux à un *Protocol* et ne transportent aucune information, ils sont juste là pour informer les *Microprotocol* de ce qui ce passe. Les informations utiles sont stockées dans les variables partagées.

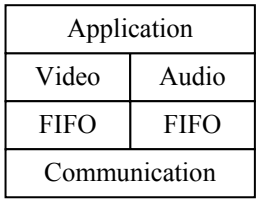


fig. 1. Cette disposition est tout à fait possible en Cactus. La couche de Communication transmet les messages soit vers Vidéo soit vers Audio selon le type des données

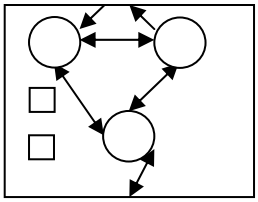


fig. 2. C'est un zoom de l'une des couches de la figure 1. Les cercles représentent les *Microprotocol*, les carrés des variables partagées et les flèches des événements. On voit ici que tous les *Microprotocol* communiquent entre eux et avec la couche elle-même.

3.1.3. Conclusion

Premièrement, Cactus a une Javadoc relativement complète et pertinente, ce qui en facilite grandement la compréhension. Cependant, il a été prévu uniquement pour des connexions TCP, non utilisable avec les réseaux ad-hoc. Cela implique le codage d'une couche de communication permettant de faire des connexions en *UDP multicast*. De plus, il est difficile de concevoir l'idée de travailler de manière non hiérarchique pour ce genre d'applications. Par ailleurs, on se demande aussi si Cactus n'utilise pas trop de threads pour ce qu'on lui demande de faire. Mais finalement, on peut conclure que Cactus est tout à fait utilisable pour les réseaux ad-hoc.

3.2. Neko

3.2.1. Introduction

Développé par le Laboratoire de Systèmes Répartis (LSR) de la faculté Informatique et Communication (I&C) de l'Ecole Polytechnique Fédérale de Lausanne (EPFL), Suisse.Versions existantes uniquement en Java. Référence : http://lsrwww.epfl.ch/neko/

3.2.2. Architecture

Neko propose une construction hiérarchique comportant des couches soit active (avec thread) soit passives (sans thread). Il fonctionne grâce à un fichier de configuration distribué à chaque hôte au lancement. Ce fichier spécifie le type de réseau utilisé, le nombre de processus ainsi que leur adresses, de quelles classes est constitué la pile du protocole et peut même contenir des paramètres spécifiques aux applications. Neko permet d'utiliser exactement le même code pour une utilisation sur un réseau réel ou simulé (JavaSim); il suffit de spécifier le type de réseau choisit dans le fichier de configuration.

3.2.3. Conclusion

L'étude de ce framework n'ayant pas été approfondie dans le cadre de ce projet (voir 2.1), les connaissances acquises sont purement théoriques et plutôt vagues. Toutefois, il semble, a priori, que Neko puisse être utilisable pour des réseaux ad-hoc.

3.3. Appia

3.3.1. Introduction

Développé par le département d'informatique (Departamento de Informática) de la faculté de science (Faculdade de Ciências) de l'université de Lisbonne (Universidade de Lisboa), Portugal. Versions existantes uniquement en Java. Référence : http://appia.di.fc.ul.pt/. Appia est une re-implémentation de Ensemble.

3.3.2. Architecture

Un seul thread gère toute la pile. Celle-ci est hiérarchique et communique par le moyen d'événements. Appia fait une distinction entre la définition statique (*Layer*) et dynamique (*Session*) d'une couche – dans la suite du paragraphe, le mot "couche" fera référence à une *Session* et le *Layer* correspondant. Les *Layer* sont uniques et ne font que définir les événements acceptés, fournis et requis par la couche. Ceci permet un contrôle partiel de la pile dans le sens où une erreur sera générée si un événement requis par un certain *Layer* n'est fournit par aucun autre. Une *Session* est une instance d'un *Layer* et décrit le comportement de la couche, c'est-à-dire, traite les données. Il peut y avoir plusieurs *Sessions* pour un seul *Layer*. Pour définir ses propres couches, l'utilisateur doit juste étendre les classes *Layer*, respectivement *Session*. De même, il y a une distinction entre *QoS* (Quality of Service) et *Channel*. Le premier est un tableau de *Layer* alors que le second est un tableau de *Session* et une instance du premier. Pour construire la pile, il faut définir un *QoS* auquel sont attribués des *Session*, puis un appel à la méthode *run()* envoi un événement *ChannelInit*, indiquant que le programme est prêt, dans chaque *Channel*. Une *Session* peut être commune à plusieurs *Channel*, c'est ce que les développeurs d'Appia ont appelé la structure en diamant (fig. 3). Une pile est donc composée de un ou plusieurs *Channel*. Les événements sont locaux à un *Channel* et peuvent soit monter soit descendre la pile. L'utilisateur peut écrire ses propres événements, il lui suffit pour cela d'étendre un événement déjà existant. Le nouvel événement sera alors considéré comme étant du super-type par le framework. La super-classe de tous les événements est *Event*. Ils peuvent contenir deux types d'information, des attributs ou des messages. Les attributs ne peuvent pas quitter le *Channel*, alors que les messages sont destinés à être envoyés sur le réseau. Il n'y a pas de contraintes sur les attributs, mais our transporter un message, un

événement doit étendre la classe *SendableEvent*. Chaque *Session* peut ajouter (*push()*) ou retirer (*pop()*) des headers du message, mais ne peut pas accéder aux headers des autres *Sessions* (sauf pour du cryptage, …). Les headers sont des tableaux de bytes, cependant, il existe des *ObjectsMessage*, une sous-classe de *Message*, qui eux permettent de travailler avec tous les types prédéfinit en Java (*Object* y comprit). Il y a possibilité d'insérer des événements, de manière asynchrone, depuis une application extérieure à la pile à l'aide de la méthode *asyngo()*. Un deuxième thread s'occupe d'écouter le socket et de mettre les messages reçus dans un événement avant de l'envoyer dans la pile.

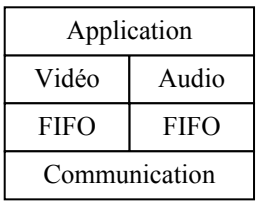


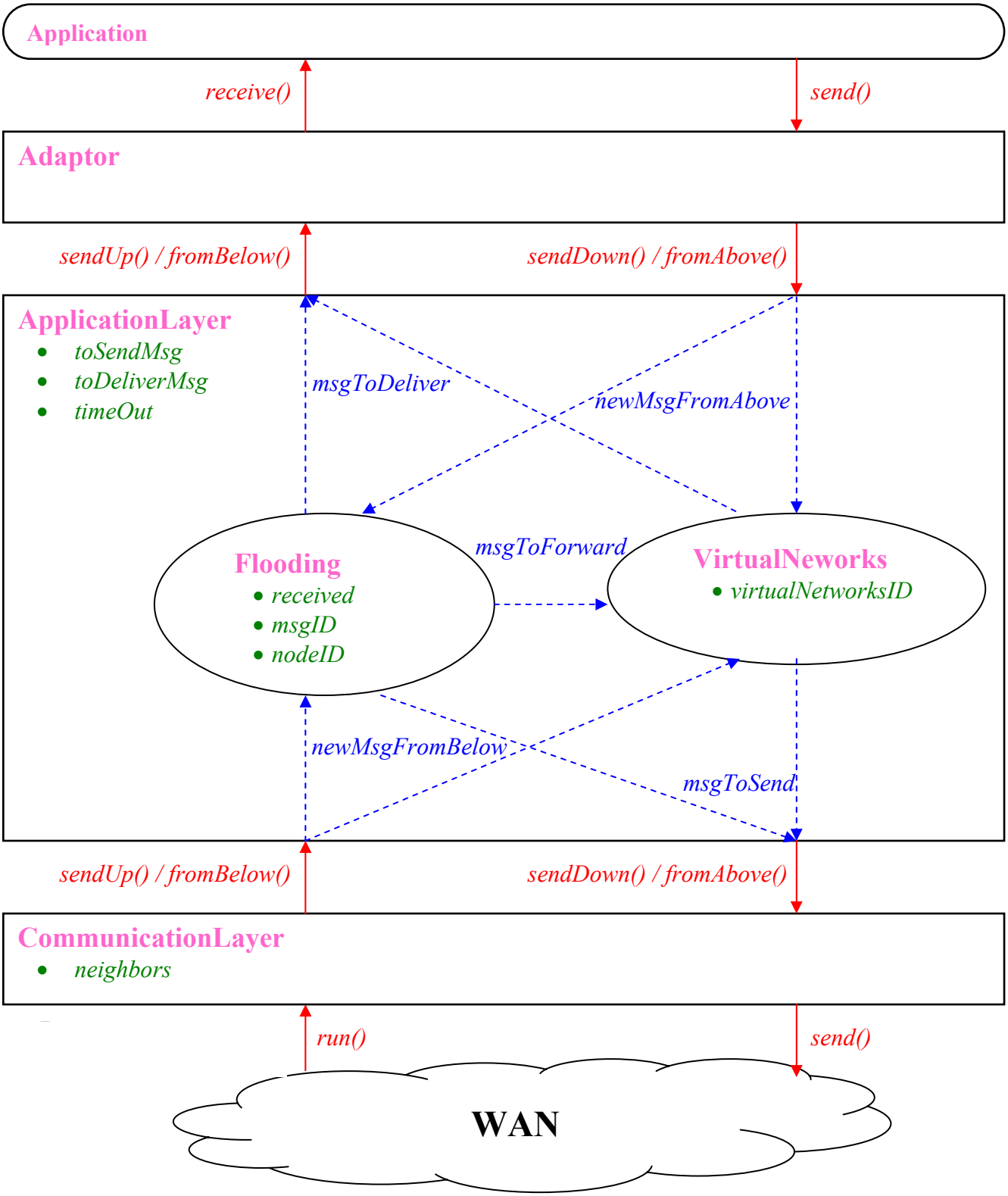
fig. 3. Un exemple de la structure en diamant. La pile est composée de deux *Channel*, le premier comprend les couches Application, Vidéo, FIFO et Communication, le second, les couches Application, Audio, FIFO et Communication. Les messages arrivant à communication contiennent aussi le *Channel* dans lequel ils doivent être envoyés. Il n'a pas moyen d'établir une communication directe entre les couches Vidéo et Audio, il faut passer par Application ou Communication.

3.3.3. Conclusion

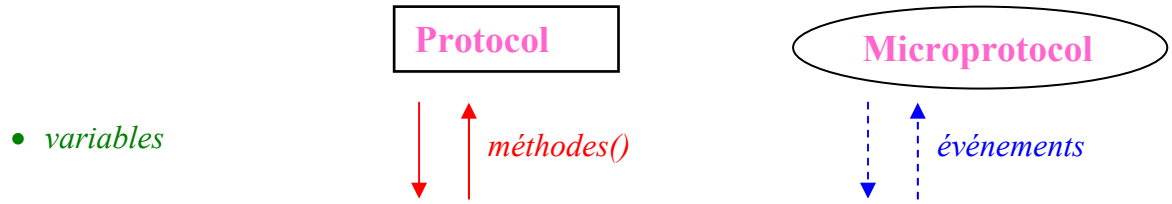
Contrairement à Cactus, la documentation d'Appia est quasi-inexistante. La Javadoc ne fournit que la signature des méthodes et parfois une légère explication sur l'utilisation d'une classe. Cela se révèle totalement insuffisant pour comprendre a quoi servent les nombreuses classes déjà codées. On est donc obligé de se plonger dans le code source, et là, de nouveau, il y a absence de commentaires. Tout cela a rendu l'exploration de ce framework très ardue, si bien qu'il m'a fallut deux semaines, passées à déchiffrer le code source, pour en arriver à la conclusion que toutes les couches, déjà présentes dans le framework, servant à la communication de groupe sont inutilisables. En effet, elles supposent la présence d'un serveur fixe qui gère les arrivées et départ des membres du groupe, ce qui est incompatible avec un réseau ad-hoc. La seule couche réutilisable est la couche **UDPSimple** qui s'occupe de l'interaction avec le réseau. Pour conclure, Appia est tout à fait utilisable avec des réseaux ad-hoc, moyennant le codage de couches spécifiques.

4. Le travail fait en Cactus

4.1. Schéma



Légende :



4.2. Détail des couches

4.2.1. Définitions

Les termes définis ci-après sont valables pour tout le chapitre 4.

- noeud : n'importe quoi (ordinateur, palm, portable, ...) relié à un réseau ad-hoc et accueillant ce programme.
- message : message au sens de cactus, c'est-à-dire un objet de type *Message* transportant des informations d'une couche à une autre. Un message peut traverser le réseau.
- événement : *Event* au sens de Cactus, c'est-à-dire un objet sans information et local à un *Protocol*. Un événement sert juste à avertir les *Microprotocol* de ce qui se passe.
- couche : c'est un *Protocol* ou un *Microprotocol.*
- couche supérieure : c'est le *Protocol* atteignable par appel de *sendUp()*.
- couche inférieure : c'est le *Protocol* atteignable par appel de *sendDown()*.
- paquet : paquet au sens UDP, c'est-à-dire un objet de type *DatagramPacket* transportant des informations d'une noeud à l'autre.
- texte : un String à envoyer à une autre noeud, utilisé par **Application**.
- voisin : noeud atteignable en un hop.
- signaler : quand toutes les entités (*Protocol* et *Microprotocol*) enregistrées pour un événement ont signalé celui-ci, l'événement est levé.

4.2.2. CommunicationLayer

Dans l'état actuel du code, cette couche ne permet de travailler qu'avec des String, de plus est communique directement avec **Adaptor** plutôt qu'avec **ApplicationLayer**. Cependant, les explications suivantes correspondent au schéma du point 4.1.

Cette couche envoie sur une adresse *multicast* tous les messages venant de **ApplicationLayer** et transmet à cette même couche tous les messages reçus du réseau sauf ceux dont le texte est "HELLO". Ceux-ci sont jetés. De plus, cette couche s'occupe de tenir plus ou moins à jour une liste des voisins et avertir ceux-ci de sa présence périodiquement. Toutes les 5 secondes, si aucun message n'est partit, un message "HELLO" est envoyé. A la réception d'un message ("HELLO" ou non) l'adresse de l'expéditeur est insérée dans une liste et tous les voisins qui n'ont pas envoyé de message depuis plus de 10 secondes sont virés de cette liste. La réception des messages se fait par la classe *SocketListener*, qui est gérée par un thread séparé. Cette classe ne fait que lire le socket de manière bloquante et transmet à **CommunicationLayer** tous les paquets reçus.

4.2.3. ApplicationLayer

Dans l'état actuel du code, cette classe ainsi que les deux suivantes n'existent pas. Cependant, les explications suivantes correspondent au schéma du point 4.1.

Cette couche transmet à **CommunicationLayer** tous les messages venant des applications. Les messages venant de **CommunicationLayer** sont traités en quatre catégories : (1) la destination est ce noeud et le

message est reçu pour la 1ère fois; (2) la destination n'est pas ce noeud et le message est reçu pour la 1ère fois; (3) le message a déjà été reçu une fois; (4) l'expéditeur est sur un réseau virtuel incompatible.
Les messages de la catégorie (1) sont transmis à l'application, ceux de la catégorie (2) sont retransmis à **CommunicationLayer** et ceux des catégories (3) et (4) sont simplement jetés.
Pratiquement, lors d'un appel a *fromBelow()*, *newMsgFromBelow* est levé et le message est stocké dans *toDeliverMsg*. Quand *msgToDeliver* est levé par les deux *Microprotocol*, la méthode *sendUp()* est appelée avec le premier message de *toDeliverMsg* en paramètre. Dans l'autre sens, lors d'un appel à *fromAbove()*, un *newMsgFromAbove* est levé et le message est stocké dans *toSendMsg*. Quand *msgToSend* est levé par les deux *Microprotocol*, la méthode *sendDown()* est appelée avec le premier message de *toSendMsg* en paramètre. Si un événement n'est signalé que par un *Microprotocol*, on attend le signal de l'autre. S'il n'arrive pas après *timeOut*, le signal du premier *Microprotocol* est annulé et le message correspondant est jeté.

4.2.4. Flooding

Cette couche s'occupe du routage des messages. En fait, elle réexpédie tout les messages reçus pour la première fois et dont la destination n'est pas ce nœud. Pratiquement, après réception de *newMsgFromAbove*, elle signale un *msgToSend* et ajoute un identificateur unique de message (*msgID*) au premier message de *toSendMsg*. Celui-ci est inséré dans la liste des messages reçus (*received*), cela permet de ne pas envoyer le message deux fois : à sa création et à sa première réception. Quand *newMsgFromBelow* est reçu, *msgID* est retiré du premier message de *toDeliverMsg* et inséré dans *received*. Si il y était déjà, le message est jeté, sinon la destination du message est comparée avec *nodeID*. Si c'est la même, un *msgToDeliver* est signalé, sinon *msgID* est remis dans le message et celui-ci est retiré de *toDeliverMsg* pour être inséré en tête de *toSendMsg*. Ensuite, un *msgToForward* est levé et un *msgToSend* est signalé.

4.2.5. VirtualNetworks

Cette couche sert juste à modéliser un réseau virtuel, c'est-à-dire un environnement où deux noeuds doivent passer par une troisième pour pouvoir communiquer (fig. 4.). Pour implémenter ce fonctionnement, on définit des réseaux virtuels auxquels on attribue un numéro d'identification. Chaque noeud possède un tableau d'identificateurs de réseaux virtuels (*virtualNetworksID*) qui représente tout les réseaux auxquels ce noeud est reliée directement. Après réception de *newMsgFromAbove* ou de *msgToForward*, *virtualNetworksID* est inséré dans le premier message de *toSendMsg* et un *msgToSend* est signalé. A la réception de *newMsgFromBelow*, le tableau d'identificateurs de réseaux virtuels est retiré du premier message de *toDeliverMsg* et comparé à *virtualNetworksID*. Si au moins l'un des identificateurs se trouve dans les deux tableaux, un *msgToDeliver* est signalé, sinon le message est jeté.
A noter que cette couche est facultative, sa présence (ou son absence) ne change rien au fonctionnement du reste de la pile. Si elle n'est pas là, l'événement *msgToForward* devient orphelin, mais ne gène pas le programme.

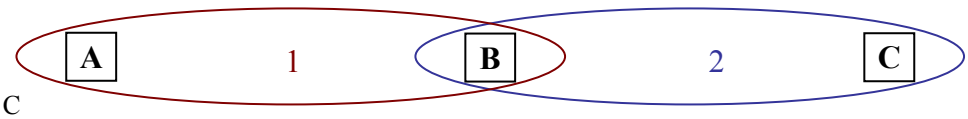


fig. 4. Il y a trois noeuds : A, B et C ainsi que deux réseaux virtuels : 1 et 2. Le nœud A ne connaît que le réseau 1 et C que le réseau 2. B connaît les deux réseaux. Donc si A et C veulent communiquer, ils sont obligés de passer par B.

4.2.6. Adaptor

Cette couche sert de transition entre la pile Cactus et l'application. Son seul rôle est de simuler la présence de couches au-dessus de **ApplicationLayer**. Elle est inutilisable sans **Application** et sert uniquement à tester le programme. Elle est destinée à être remplacée par des couches plus élaborées, déjà développées par le LSR.
Pratiquement, elle ne fait rien d'autre que d'appeler la méthode *receive()* de **Application** sur appel de sa méthode *fromBelow()*. Elle appelle *sendDown()* sur appel de sa méthode *send()* par **Application**. Elle doit aussi transformer les données de *String* à *Message* et vice-versa.

4.2.7. Application

C'est une application de chat très rudimentaire, elle ne fait que d'envoyer un String en broadcast et d'afficher tous les String reçus avec leur expéditeur. Comme est doit s'occuper d'appels au clavier, elle est gérée par un autre thread que ceux de cactus, c'est pourquoi elle est indépendante des autres couches. Elle est inutilisable sans **ApplicationLayer** et sert uniquement à tester le programme.
Pratiquement, elle demande à l'utilisateur d'entrer un message à envoyer ou de taper "exit" pour quitter le système. Puis elle appelle la méthode *send()* de **ApplicationLayer**. Cette dernière couche fait appel à la méthode *receive()* de **Application** pour lui transmettre des données.

4.3. Evénements

4.3.1. msgToDeliver

Cet événement, accepté par **ApplicationLayer**, est signalé par **Flooding** si le premier message de *toDeliverMsg* est reçu pour la première fois et sa destination est ce noeud. Il est signalé par **VirtualNetworks** si ce même message est sur un réseau virtuel compatible. Lorsque il est signalé par les deux *Microprotocol*, le message en question est retiré de *toDeliverMsg* et transmis à la couche supérieure via *sendUp()*. Si il n'est signalé que par un seul pendent *timeOut* secondes, le signal est annulé et le message est retiré de *toDeliverMsg* et jeté.

4.3.2. msgToForward

Cet événement, accepté par **VirtualNetworks**, est levé par **Flooding** si le premier message de *toDeliverMsg* est reçu pour la première fois et n'est pas pour ce noeud. Ce message est alors retiré de *toDeliverMsg* et inséré en tête de *toSendMsg*. **VirtualNetworks** le traite comme si il avait reçu un *msgToSend*. Si **VirtualNetworks** est supprimé, cet événement devient orphelin, mais ne gène pas le bon fonctionnement du reste du programme.

4.3.3. msgToSend

Cet événement, accepté par **ApplicationLayer**, est signalé par **VirtualNetworks** après que *virtualNetworksID* aie été ajouté au premier message de *toSendMsg*. Il est signalé par **Flooding** dès que *msgID* a été ajouté à ce même message ou juste après avoir levé un *msgToForward*. Lorsque il est signalé par les deux *Microprotocol*, le message en question est transmis à la couche inférieure. Si il n'est signalé que par un seul pendent *timeOut* secondes, le signal est annulé et le message est retiré de *toSendMsg* et jeté.

4.3.4. newMsgFromBelow

Cet événement, accepté par **Flooding** et **VirtualNetworks**, est levé par **ApplicationLayer** sur appel de *fromBelow()* juste après avoir inséré le nouveau message en queue de *toDeliverMsg*. Il indique aux deux *Microprotocol* qu'un nouveau message est arrivé du réseau.

4.3.5. newMsgFromAbove

Cet événement, accepté par **Flooding** et **VirtualNetworks**, est levé par **ApplicationLayer** sur appel de *fromAbove()* juste après avoir inséré le nouveau message en queue de *toSendMsg*. Il indique aux deux *Microprotocol* qu'un nouveau message est arrivé de l'application.

4.4. Variables

4.4.1. msgID *[Identifier]*

Cette variable représente un identificateur unique de message pour tout le réseau au niveau de **Flooding**. Elle est insérée dans le premier message de *toSendMsg* et retiré du premier message de *toDeliverMsg* si celui-ci doit être transmis à la couche supérieure (elle reste dans les messages retransmis à la couche inférieure). Le type *Identifier* contient l'adresse (*InetAddress*) de l'expéditeur plus un numéro, incrémenté à chaque message.

4.4.2. neighbors *[ArrayList(InetAddress)]*

Cette variable est une liste contenant les adresses de tous les voisins. Elle est liée à un mécanisme asynchrone (chaque appel à *sendUp()*) supprimant toutes les adresses vieilles de plus de 10 secondes. Cette liste n'est pas parfaitement à jour, il y a un léger décalage entre le moment où un voisin change d'état (connecté / non connecté) et le moment où la liste est mise à jour. Cependant, ce délai n'est pas jugé important.

4.4.3. nodeID *[InetAddress]*

Cette variable représente l'adresse de ce noeud. Si elle est égale à la destination du message, alors un *msgToDeliver* est signalé. Sinon, un *msgToSend* est signalé et un *msgToForward* est levé.

4.4.4. received *[HashSet(Identifier)]*

Cette variable est une liste contenant les identificateurs de tous les messages déjà reçus par **Flooding**. On y ajoute les identificateurs de chaque message (*msgID*) arrivant jusque là, si il était déjà présent, le message est jeté.

4.4.5. timeOut *[int]*

Cette variable représente le délai à attendre avant d'annuler un signal si le deuxième n'arrive pas.

4.4.6. toDeliverMsg *[LinkedList(Message)]*

Cette variable est une liste dans laquelle les nouveaux messages arrivant de **CommunicationLayer** sont insérés en queue. Le premier message en est retiré, soit pour être traité, auquel cas il est réinséré en tête après, soit pour être transmis, soit à la couche inférieure, auquel cas il est inséré en tête de *toSendMsg* soit supérieure. Comme deux *Microprotocol* vont accéder à cette variable en même temps, il est nécessaire de la synchroniser.

4.4.7. toSendMsg *[LinkedList(Message)]*

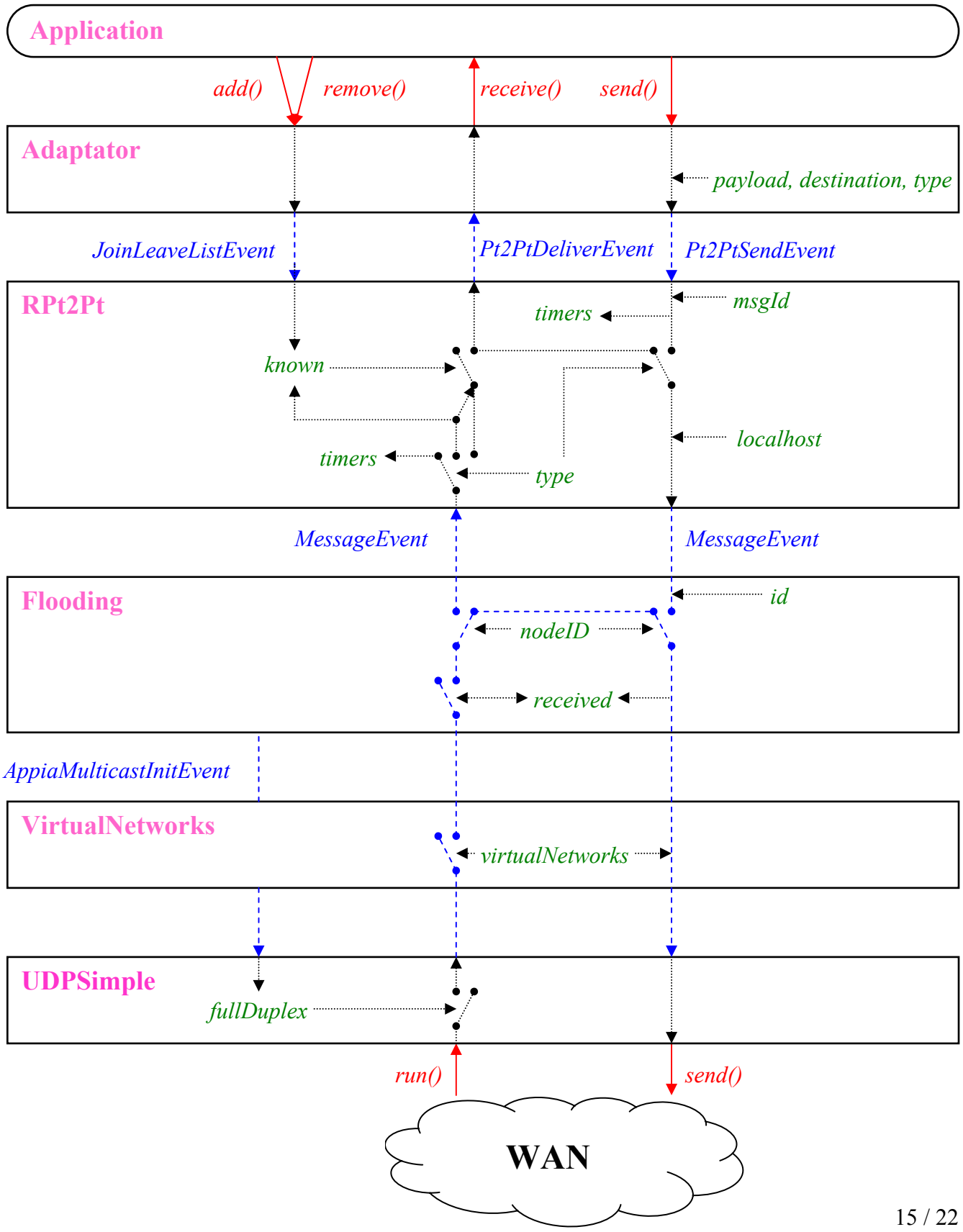
Cette variable est une liste dans laquelle les nouveaux messages arrivant de **Application** sont insérés en queue. Le premier message en est retiré soit pour être traité, auquel cas il est réinséré en tête après, soit pour être transmis à la couche inférieure. Comme deux *Microprotocol* vont accéder à cette variable en même temps, il est nécessaire de la synchroniser.

4.4.8. virtualNetworksID *[int]*

Cette variable contient tous les identificateurs de réseaux virtuels acceptés par ce noeud. Elle est insérée par **VirtualNetworks** dans le premier message de *toSendMsg*.

5. Le travail fait en Appia

5.1. Schéma



5.2. Détail des couches

5.2.1. Définitions

Les termes définis ci-après sont valables pour tout le chapitre 5.

- couche : une *Session* et le *Layer* associé.
- noeud : n'importe quoi (ordinateur, palm, portable, ...) relié à un réseau ad-hoc et accueillant ce programme.
- message : structure dans laquelle on peut insérer des données sous forme de header. Un message est encapsulé dans un événement (qui doit obligatoirement étendre *SendableEvent*), jusqu'à ce qu'il soit envoyé sur le réseau. Un message peut passer d'une noeud à l'autre, mais ne peut changer de couche qu'à l'intérieur d'un événement.
- événement : *Event* au sens de Appia (voir 3.3.2), c'est-à-dire un objet pouvant passer d'une *Session* à une autre sans pouvoir changer de *Channel*. Moyen de transport des informations, un événement ne peut pas changer de noeud, mais peut monter ou descendre la pile.
- message montant : message venant du réseau
- message descendant : message à destination du réseau

5.2.2. UDPSimple

Cette session, fournie avec le framework Appia, gère le socket. Elle envoie sur une adresse *multicast* tous les messages descendants. Les messages montants sont transmis à la couche supérieure selon la valeur de *fullDuplex*. Cette variable indique simplement si la couche doit recevoir ses propres messages. **UDPSimple** est initialisée par un *AppiaMulticastInitEvent* qui spécifie le port et l'adresse *multicast* à utiliser.

5.2.3. VirtualNetworks

Cette couche sert juste à modéliser un réseau virtuel, c'est-à-dire un environnement où deux noeuds doivent passer par un troisième pour pouvoir communiquer (fig. 4 page 12). Pour implémenter ce fonctionnement, on définit des réseaux virtuels auxquels on attribue un numéro d'identification. Chaque noeud possède un tableau d'identificateurs de réseaux virtuels (*virtualNetworks*) qui représente tout les réseaux auxquels ce noeud est reliée directement. Ce tableau est inséré dans chaque message descendant et remplace le tableau déjà présent le cas échéant. Les messages montants, sont transmis à la couche supérieure si son expéditeur est sur un réseau virtuel compatible, c'est-à-dire que l'un, au moins, des réseaux virtuels affectés au message correspondent à l'un des réseaux virtuels acceptés par ce noeud. Les messages ne remplissant pas cette condition sont jetés. A noter que cette couche est facultative, sa présence (ou son absence) ne change rien au fonctionnement du reste de la pile.

5.2.4. Flooding

Cette couche a deux fonctions, premièrement, elle s'occupe du routage des messages et deuxièmement, elle initialise **UDPSimple** grâce à un *AppiaMulticastInitEvent* qui spécifie le port et l'adresse à utiliser pour le *multicastSocket*. Cette même adresse *multicast* est ajoutée dans chaque *MessageEvent* descendent. Puis, un identificateur unique de message (*id*) est inséré dans chaque message descendant. Celui-ci est aussi inséré

dans la liste des messages reçus (*received*), cela permet de ne pas envoyer le message deux fois : à sa création et à sa première réception. Les messages montant ayant déjà été reçus sont jetés, les autres sont ajoutés à *received*. Puis, les messages dont le destinataire est *nodeID* sont transmis à la couche supérieure, les autres sont retransmis à la couche inférieure.

5.2.5. RPt2Pt

Cette couche est inspirée du projet de semestre de Jean Vaucher (LSR, décembre 2002). Sa tâche principale est de fournir un système d'acquittement. A chaque message descendent, si le destinataire est connu, elle ajoute un identificateur de message (*msgID*) et l'adresse de l'expéditeur (*localhost*) puis elle stocke une copie du message et lance un timer périodique avec *msgID* comme identificateur. Si le destinataire est inconnu, un texte d'erreur est imprimé et le message est ignoré. Pour chaque message montant, elle arrête le timer correspondant, si c'est un acquittement, ou envoie un acquittement contenant *msgID* autrement. Puis, si le message est reçu pour la première fois, il est transmis à la couche supérieure, sinon il est jeté. Si un timer expire avant la réception de l'acquittement, le message est renvoyé. Cette couche ne permet pas de recevoir un message d'une adresse inconnue sauf si le type est "promisc", auquel cas on ajoute l'expéditeur à la liste des processus connus (*known*) avant de traiter le message.
La deuxième tâche de cette couche est de rendre compatible la pile déjà codée avec les applications déjà développées au LSR. Pratiquement, un *MessageEvent* est créé avec les données reçues dans le *Pt2PTSendEvent* (*payload*, *type* et *destination*) à la réception de celui-ci. Et un *Pt2PTDeliverEvent* contenant le *payload* et l'expéditeur est créé lors de la réception d'un *MessageEvent*. Si la destination d'un message est "0.0.0.0", cela indique un broadcast, une copie du message est alors envoyée, en unicast, à chaque processus présent dans *known*. Les acquittements sont des messages sans *payload*, mais avec le même *msgID* que le message à acquitter, ce qui permet de retrouver le timer associé. Attention à ne pas confondre *msgID*, qui identifie un message localement au niveau de **RPt2PtSession** et *id*, qui identifie un message globalement au niveau de **Flooding**. C'est-à-dire que plusieurs messages pris sur différents noeuds peuvent avoir le même *msgID*, alors qu'un *id* est unique sur tout le réseau.

5.2.6. Adaptor

Cette couche sert de transition entre la pile Appia et l'application. Son seul rôle est de simuler la présence de couches au-dessus de **RPt2Pt** en lui fournissant les événements nécessaires. Elle est inutilisable sans **Application** et sert uniquement à tester le programme. Elle est destinée à être remplacée par des couches plus élaborées, déjà développées par le LSR.
Pratiquement, elle ne fait rien d'autre que créer un *JoinLeaveListEvent* sur appel de ses méthodes *add()* ou *remove()* et un *Pt2PTSendEvent* sur appel de sa méthode *send()*. Elle appelle la méthode *receive()* de **Application** lorsque elle reçoit un *Pt2PtDeliverEvent*. Elle doit aussi transformer le format des données : de *String* à *VSCASTMessage* et vice-versa.

5.2.7. Application

C'est une application de chat très rudimentaire, elle ne fait qu'envoyer un *String* au *Channel* soit vers une destination particulière soit en broadcast (tous les processus connus). Tous les String reçus sont affichés avec leur expéditeur. Elle permet aussi d'ajouter ou de retirer des processus. Comme est doit s'occuper d'appels au clavier, elle est gérée par un autre thread que celui d'Appia, c'est pourquoi elle est indépendante de la pile. Elle est inutilisable sans **Adaptor** et sert uniquement à tester le programme.
Pratiquement, elle demande à l'utilisateur ce qu'il veut faire : ajouter ou supprimer un processus, envoyer un message en mode "normal" ou "promisc", quitter le système. Selon la réponse, elle demande les informations nécessaires (destination ou message) et appelle les méthodes correspondantes de **Adaptor** : *add()* pour un ajout, *remove()* pour une suppression et *send()* pour un envoi (quelque soit le type).

5.3. Evénements

5.3.1. AppiaMulticastInitEvent

Cet événement, fournit avec le framework Appia, sert à initialiser **UDPSimple**, c'est-à-dire, lui dire de créer un *multicastSocket* avec un port et une adresse déterminés par l'événement. Il fixe aussi la valeur de *fullDuplex*.

5.3.2. JoinLeaveListEvent

Cet événement, fournit par le LSR, sert à ajouter ou supprimer des processus de la liste des processus connus (*known*). Il contient deux *Set* : l'un pour les nouveaux processus, l'autre pour les processus à supprimer. Cet événement permet donc d'ajouter et de supprimer plusieurs processus en même temps.

5.3.3. MessageEvent

Cet événement, définit uniquement pour les couches inférieures à **RPt2Pt**, est utilisé pour la transmission de messages d'une couche à l'autre. Il étend *SendableEvent* afin de pouvoir envoyer son message sur le réseau. La seule différence entre ces deux événements est que *MessageEvent* contient un *ObjectsMessage* plutôt qu'un *Message*, ce qui lui permet de manipuler tous les types prédéfinis en Java (*Object* y comprit) plutôt que des tableaux de *byte*. *ObjectsMessage* est un type prédéfinit en Appia et sert normalement à la communication de groupe (voir 3.2.2).

5.3.4. Pt2PtDeliverEvent

Cet événement, fournit par le LSR, est définit uniquement pour les couches supérieures à **RPt2Pt**. Il est utilisé pour la transmission de données vers le haut. Il contient une liste dans laquelle se trouve l'adresse de l'expéditeur et le *payload*, une liste contenant tout les headers ajoutés par les couches supérieures à **RPt2Pt** du processus expéditeur.

5.3.5. Pt2PtSendEvent

Cet événement, fournit par le LSR, est définit uniquement pour les couches supérieures à **RPt2Pt**. Il est utilisé pour la transmission de données vers le bas. Il contient une liste dans laquelle se trouve le *type* ("normal" ou "promisc"), la *destination* et le *payload*, une liste contenant tout les headers ajoutés par les sessions supérieures à **RPt2Pt**.

5.4. Variables

5.4.1. destination *[PID]*

Cette variable représente l'adresse du processus cible. Elle est insérée dans chaque message descendant. Elle est utilisée par **Flooding** pour déterminer si le message est arrivé à bon port ou non, donc s'il doit être transmis à la couche supérieure ou inférieure. Le type *PID* est un identificateur de processus (soit l'expéditeur, soit le destinataire), il contient une *InetAddress*, un port (*int*) et une incarnation (*int*). Le port sert à différencier plusieurs processus sur la même noeud. Mais comme on ne peut pas avoir plusieurs processus par noeud en faisant du *UDP multicast*, ce champ ne signifie rien et un numéro arbitraire lui est attribué. La classe *PID* est fournit et imposée par le LSR.

5.4.2. fullDuplex *[boolean]*
Cette variable, dont la valeur est fixée par un *AppiaMulticastInitEvent*, indique simplement si le *multicastSocket* doit recevoir ses propres messages.

5.4.3. id *[Identifier]*
Cette variable représente un identificateur unique de message pour tout le réseau au niveau de **Flooding**. Elle est insérée dans chaque message descendant et supprimée de chaque message montant transmis à la couche supérieure (elle reste dans le message retransmis à la couche inférieure). Le type *Identifier* contient l'adresse (*InetAddress*) de l'expéditeur plus un numéro, incrémenté à chaque message.

5.4.4. known *[Hashtable(PID, KnownPID)]*
Cette variable est une liste contenant tous les processus connus. Elle permet de retrouver un processus grâce à son adresse (*PID*). Un processus doit être dans cette liste pour pouvoir lui envoyer ou en recevoir un message. *KnownPID* est un type permettant d'avoir un seul objet représentant un processus, on y trouve aussi tous les messages de ce processus en attente d'acquittement ainsi que tous les messages reçus de ce processus. La classe *KnownPID* est fournit et imposée par le LSR.

5.4.5. localhost *[PID]*
Cette variable représente l'adresse du processus expéditeur. Elle est insérée dans chaque message descendant.

5.4.6. msgId *[int]*
Cette variable identifie un message au niveau de **RPt2Pt**, elle est insérée dans tous les messages arrivant de la couche supérieure. Elle est utilisée pour relier un acquittement à un message et un timer. Elle sert aussi à virer les messages reçus plusieurs fois suite à, par exemple, une perte de l'acquittement.

5.4.7. nodeID *[InetAddress]*
Cette variable représente l'adresse de ce noeud. Si elle est égale à la destination du message, alors celui-ci est transmis à la session supérieure, sinon, il est retransmis à la session inférieure.

5.4.8. payload *[VSCASTMessage]*
Cette variable représente toutes les données qui ne sont pas utilisées pour la communication. Le type *VSCASTMessage* est simplement une *linkedList* dans laquelle se trouvent tous les headers ajoutés par les couches au-dessus de **RPt2Pt**. Les données contenues dans cette variable peuvent être de n'importe quel type. Elle est insérée dans tous les messages descendants mais est absente des acquittements. Ce type est fournit par le LSR.

5.4.9. received *[HashSet(Identifier)]*
Cette variable est une liste contenant les identificateurs de tous les messages déjà reçus par **Flooding**. On y ajoute les identificateurs de chaque message (*id*) arrivant jusque là, si il était déjà présent, le message est jeté.

5.4.10. timers *[Hashtable (String, KnownPID)]*
Cette variable est une liste contenant tous les timers en cours utilisé par **RPt2Pt**. Elle permet, premièrement de savoir si le timer reçu doit être traité ou ignoré (il peut être destiné à une autre couche). Deuxièmement de retrouver, grâce à l'identificateur du timer, un *String* représentant le *PID* du destinataire et *msgID*, le processus auquel il faut renvoyer un message. Ce même *String* est encore utilisé pour déterminer quel

message il faut renvoyer, ceux-ci étant stockés dans une *Hashtable(String, ObjectsMessage)*. Dès qu'un acquittement est reçu, le timer correspondant est viré de cette liste.

5.4.11. type *[int]*

Cette variable peut prendre trois valeurs : "normal", pour envoyer un message, "promisc" pour signaler en plus l'identité de l'expéditeur et "ack" pour indiquer un acquittement. Si le *type* du message montant est "normal", le *payload* et l'expéditeur (*PID*) sont transmis à la couche supérieure dans un *Pt2PtDeliverEvent*. De plus, on envoie un acquittement (un message de *type* "ack" avec le même *msgID*). Si c'est "promisc", l'expéditeur est ajouté à la liste des processus connus, puis le message est traité comme le cas "normal". Enfin, si le *type* est "ack", on arrête simplement le timer correspondant. Quand le *type* est fournit par *Pt2PtSendEvent*, il ne peut prendre que l'une des deux premières valeurs.

5.4.12. virtualNetworks *[int[]]*

Cette variable contient tous les identificateurs de réseaux virtuels acceptés par ce noeud. Elle est insérée par **VirtualNetworks** dans chaque message descendant et remplace l'ancienne liste le cas échéant.

6. Améliorations possibles

6.1. En Cactus

1) La première amélioration à apporter en Cactus est de coder les classes **Flooding**, **VirtualNetworks** et **ApplicationLayer**. Ce ne devrait pas poser de problèmes majeurs puisque tout le cheminement est déjà expliqué (4.2).
2) Si l'amélioration 1) est déjà faite, le prochain pas consiste à coder le *Microprotocol* **RPt2Pt**. Le codage de cette entité est plus ardu, mais en s'inspirant de ce qui est déjà fait en Appia et en demandant le pseudo code correspondant au LSR, ce ne devrait pas être trop difficile.
3) L'une des faiblesses du code actuel est qu'il ne permet de traiter que des *String*. Afin d'intégrer tous les types courants, il faut écrire une méthode permettant de transformer ces types en tableaux de bytes et vice-versa. En effet, le constructeur de *DatagramPacket* impose ce format de données.
4) Comme Cactus permet d'avoir plusieurs sous-couches pour une couche, il est possible de déclarer la classe *SocketListener* comme une deuxième sous-couche de **ApplicationLayer** (fig. 5.) tout en restant gérée par un autre thread. A ce moment, il n'y aurait plus de communication entre **CommunicationLayer** et **SocketListener**. Il faut alors déplacer la méthode de mise à jour des voisins ainsi que la liste des voisins dans la nouvelle couche. Le problème est que maintenant **SocketListener** doit aussi avoir une méthode d'envoi pour les messages "HELLO". Cependant, si, cette fonctionnalité est déplacée dans une autre couche ou, comme en Appia, carrément supprimée, le problème est résolu.
5) Dans le même ordre d'idée, on pourrait ajouter une couche **Dispatcher** permettant à plusieurs applications ayant chacune leurs propres couches de tourner sur le même noeud. A ce moment, l'adresse d'un noeud n'est plus suffisante pour identifier une destination, il faut utiliser la classe *PID* (voir 5.4.1) dont le port identifierait un processus. Il faut aussi remplacer la variable *nodeID* de **Flooding** par une *Hashtable(PID, Protocol)* dont la clé est l'adresse du processus et l'objet, une référence sur la couche correspondante. Ensuite, il ne reste plus qu'à faire appel à la bonne méthode *sendUp()*. Le problème est qu'il est obligatoire le connaître le nom de chaque application au démarrage afin de pouvoir coupler leur couches au dispatcher.

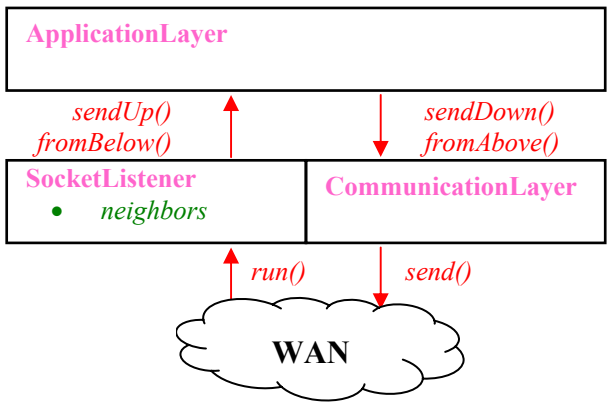


fig. 5. C'est une reprise simplifiée du schéma 4.1. La couche **ApplicationLayer** reste la même. **CommunicationLayer** ne peut plus envoyer de données à la couche supérieure de même que **SocketListener** ne peut rien recevoir cette même couche. **ApplicationLayer** envoie ses messages seulement à **CommunicationLayer** et en reçoit uniquement de **SocketListener**.

6.2. En Appia

1) Comme en Cactus, on peut ajouter une couche **Dispatcher** permettant à plusieurs applications ayant chacune leurs propres couches de tourner sur un même noeud. En Appia, la classe *PID* est déjà utilisée, donc le seul changement à effectuer est de remplacer la variable *nodeID* de **Flooding** par une *Hashtable(PID, Channel)* dont la clé est l'adresse du processus et l'objet une référence sur le *Channel* correspondant. Ensuite, il ne reste plus qu'à envoyer le message dans le bon C*hannel*. Le même problème qu'en Cactus se pose, il est obligatoire le connaître le nom de chaque application au démarrage afin de pouvoir leur attribuer un *Channel* chacun.
2) L'application de test peut être amélioré en incluant la possibilité d'ajouter et de supprimer plusieurs processus en même temps. Cela est permit par le *JoinLeaveListEvent*, mais pas par **Application** qui ne peut gérer qu'un ajout ou une suppression à la fois.
3) Si cela se révélait utile, il est toujours possible de remettre les méthodes de mise à jour des voisins et d'envoi des identités dans la couche **Flooding** ou n'importe quelle autre, tant qu'elle connaît l'expéditeur du message. La marche à suivre est décrite dans le fichier Neighbors.java.

6.3. En général

1) On pourrait créer des exceptions et les lever quand le programme fait une erreur, plutôt que de quitter le système, comme c'est le cas actuellement. Il faut pour cela définir les exceptions utiles pour le reste de la pile.
2) Une autre amélioration possible est l'optimisation. En Java, un appel à *new* est une action très coûteuse en temps et beaucoup de ces appels peuvent être évités. Par exemple, on peut réutiliser les événements et les messages au lieu d'en créer chaque fois de nouveaux. Cela implique de mettre un point un système de stockage d'objet avec le moyen d'en créer de nouveaux si le stock est épuisé. Il faut aussi être sûr de ne plus avoir besoin d'un objet avant de le stocker. Une solution serait de créer une *LinkedList* pour chaque type d'objet réutilisable et de sortir le premier en cas de besoin. Si la liste est vide, une exception est levée, il faut alors faire un appel à new. Lorsque un objet a accomplit la tâche pour laquelle il a été créé, il est réinitialisé et inséré dans cette liste.