ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

LSE

Travail pratique de diplôme

Adjonction d'un service de réplication aux SmartSockets

supervisé par le Prof. André Schiper

David Cavin
Février 2001

Résumé

Actuellement, il existe un nombre toujours plus élevé de services avec lesquels il est possible d'interagir à distance. Ces services doivent souvent tolérer la présence et l'accès concurrent d'un grand nombre de clients. Cela requiert la sécurité et la fiabilité des échanges. Un certain nombre de ses systèmes informatiques ne peuvent simplement pas tolérer l'arrêt ou la défaillance, même momentanés, des services qu'ils proposent (eBanking, systèmes temps réels, etc.).

C'est au centre de cette problématique que s'inscrit ce travail pratique de diplôme dont le but est d'implémenter un service répliqué. Le principe est de constituer un groupe de réplicats (p.ex.des ordinateurs) qui multiplient les supports où l'information est stockée de manière à pouvoir garantir sa persistance. Si l'un des réplicats, est défaillant, les autres peuvent continuer à assurer le bon fonctionnement du service. Mais il est impératif d'assurer que toutes les copies ont le même état.

Les possibilités déjà offertes par le logiciel `SmartSockets` sont étendues avec un service répliqué de type copie primaire. `SmartSockets` permet de concevoir des applications complexes qui communiquent dans une architecture de type *publish/subscribe*. C'est-à-dire, qu'un client va s'adresser au service répliqué en envoyant un message sur un certain sujet auquel la copie primaire a souscrit. Parce que `SmartSocket` n'est pas capable d'assurer à lui tout seul la cohérence du groupe de serveurs, il a été nécessaire d'implémenter une couche de communication supplémentaire qui propose un ensemble de primitives assurant un transport fiable et ordonné de l'information à l'intérieur du groupe.

Remerciements

Je tiens à remercier particulièrement le Prof. André Schiper pour son encadrement irréprochable, pour sa grande disponibilité et son intérêt pour ce projet. Je lui suis reconnaissant de m'avoir aidé à démarrer ce travail pratique de diplôme dans des conditions difficiles.

Mes remerciements vont aussi à Sébastien Baehni et Corine Hari qui m'ont soutenu durant tout ce projet, pour leur aide permanente et pour les fructueuses discussions techniques que nous avons eu ensembles.

Merci aussi à Allan Coignet qui a veillé durant quatre mois à maintenir un environnement de travail optimal en cédant à tous nos petits caprices.

Table des matières

1 Introduction 1
1.1 Objectifs 1
2 Les SmartSockets 3
2.1 Introduction 3
2.2 Les sujets 5
2.3 Les messages 6
2.4 Guaranteed Message Delivery 7
2.5 Load balancing 9
2.6 Les options 10
3 Le service répliqué 11
3.1 Introduction 11
3.1.1 Généralités 11
3.1.2 Architecture 11
3.2 L'architecture d'un réplicat 13
3.2.1 Application layer 14
3.2.2 Protocol layer 14
3.2.3 P/S layer 14
3.2.4 Failure layer 15
3.2.5 UDP transport layer 15
3.3 Les messages 16
3.3.1 Les types de messages 16
3.4 La primitive Reliable Broadcast 17
3.4.1 Description 17
3.4.2 L'architecture 18
3.4.3 L'algorithme 18
3.4.4 Justification de l'algorithme 20
Validité 20
Agrément 20
Intégrité 20
3.4.5 Exemples de chronogrammes 20
3.5 Algorithme du consensus 21
3.5.1 Description 21
Les détecteurs de fautes 22
Les « Stubborn buffers » 23
3.5.2 L'architecture 23
3.5.3 L'algorithme 24
3.5.4 Justification 26
3.5.5 Exemples de chronogrammes 26
3.6 La primitive Terminated Broadcast 27
3.6.1 Description 27
3.6.2 L'architecture 28

3.6.3 L'algorithme ... 28
3.6.4 Justification de l'algorithme ... 32
FIFO Order ... 32
3.6.5 Exemples de chronogrammes ... 32
3.7 La primitive Atomic Broadcast ... 33
3.7.1 Description ... 33
3.7.2 L'architecture ... 34
3.7.3 L'algorithme ... 36
4 L'implémentation ... 37
4.1 Environnement de développement ... 37
4.2 Préliminaires ... 37
4.3 Le paquetage client ... 38
4.4 Le paquetage server ... 39
4.4.1 Les interfaces ... 39
Les interfaces générales ... 39
Les controller ... 39
4.4.2 Les classes ... 40
Les classes générales ... 40
Les messages ... 42
Les buffers ... 42
Les managers ... 43
5 Mesures de performances ... 45
5.1 Situation ... 45
5.2 Performances d'Atomic Broadcast ... 45
6 Développements futurs ... 49
6.1 Vue dynamique ... 49
6.2 Clients répliqués ... 50
7 Conclusions ... 52
8 Abréviations ... 53
9 Bibliographie ... 54

Table des figures

Figure 1 – Les couches de développement 3
Figure 2 – Architecture des SmartSockets 4
Figure 3 – Structure arborescente des sujets 5
Figure 4 – Limites de la structure arborescente des sujets 6
Figure 5 – Structure d'un message 7
Figure 6 – Protocole GMD 8
Figure 7 – Load balancing 9
Figure 8 – Architecture générale 12
Figure 9 – Architecture du service répliqué 13
Figure 10 – Architecture d'un réplicat 14
Figure 11 – Message 16
Figure 12 – R-Broadcast Architecture 18
Figure 13 – Exécution normale de R-Broadcast 21
Figure 14 – Exécution de R-Broadcast avec retransmission 21
Figure 15 – Consensus architecture 24
Figure 16 – Exécution de l'algorithme du consensus 26
Figure 17 – Exécution de l'algorithme du consensus avec crash du coordinateur 27
Figure 18 – Terminated Broadcast architecture 28
Figure 19 – Exécution d'un T-Broadcast 33
Figure 20 – Exécution d'un T-Broadcast avec consensus 33
Figure 21 – Atomic Broadcast Architecture 35
Figure 22 – Interface graphique du client 38
Figure 23 – Informations sur l'environnement SmartSockets 39
Figure 24 – Gestion des "controller" 40
Figure 25 – Interface de déverminage 41
Figure 26 – Architecture du test 45
Figure 27 – Envoi de 100 messages avec intervalle de 100ms 46
Figure 28 – Envoi de 100 messages avec intervalle de 20ms 47
Figure 29 – A-Deliver / seconde 48
Figure 30 – Clients répliqués 50
Figure 31 – Exécution d'une requête avec un client répliqué 51

Table des algorithmes

Algorithme 1 – Reliable Broadcast 19
Algorithme 2 – Consensus 26
Algorithme 3 – Terminated Broadcast 30
Algorithme 4 – Atomic Broadcast 36

1 Introduction

Les systèmes distribués occupent une position clé dans le vaste univers de la gestion des informations électroniques. Les besoins en performances en persistance et en fiabilité sont au centre des préoccupations des gestionnaires de données informatiques. Un nombre croissant de services sont désormais accessibles via l'internet, par exemple, avec des exigences de temps réel très strictes (eBanking, eCommerce, bourse en ligne, bases de données, etc.). Ces services doivent tolérer la présence et donc l'accès simultané d'une grande quantité de clients qui souhaitent non seulement consulter, mais aussi modifier les informations qu'ils y trouvent. L'utilisateur est exigent et n'accepte pas volontiers qu'un service soit, même momentanément, inaccessible. Les récentes et nombreuses recherches dans le domaine des systèmes distribués tentent d'apporter des solutions en proposant notamment des architectures permettant de répliquer les services dans le but d'assurer leur persistance.

Le principe est de dupliquer l'information pour en garantir la disponibilité, par exemple, un groupe de plusieurs machines assurant le même service. Si l'un des membres de ce groupe, appelé aussi réplicat, tombe en panne, les autres sont capables de traiter les requêtes d'un client de manière transparente. Dès que le réplicat défaillant a retrouvé un état normal, il peut rejoindre le groupe. Si le service répliqué ne fournit que des informations non modifiables par les utilisateurs (p.ex. quotidien en ligne, indices boursiers, etc.), cela ne pose aucun problème. Par contre si les clients peuvent interagir, donc modifier les données à distances (p.ex. transmission d'un paiement, suppression d'une donnée, etc.), alors un problème de cohérence entre les réplicats peut se poser. En effet, si les requêtes des clients ne sont pas reçues ou pas dans le même ordre par toutes les copies, il va en découler une divergence d'états.

Un certain nombre d'architectures, d'algorithmes et de primitives de communication assurant le fonctionnement d'un service ont déjà été développés et sont présentés dans [Sch00] et [Mul93]. Toutes ces techniques reposent sur un protocole de transport spécifique de type TCP ou UDP, par exemple.

1.1 Objectifs

Le but de ce travail pratique de diplôme est, premièrement, de définir une architecture qui permette d'adjoindre un service répliqué basé sur un environnement de communication de type `publish/subscribe` mise en œuvre par `SmartSockets` (cf. chapitre 2). Dans cet environnement de communication un client ne s'adresse pas à un serveur pour lui demander une information mais c'est le serveur qui publie une information à laquelle certains clients peuvent être intéressés. Il faut donc définir la manière dont va être intégré le service répliqué dans cet environnement.

Dans un deuxième temps, il s'agit de développer, de mettre au point et d'implémenter un certain nombre de primitives de communication, c'est-à-dire de protocoles d'envoi de messages qui permettront d'assurer la cohérence des réplicats.

La dernière partie de ce projet consiste à mesurer les performances du service répliqué dans des conditions les plus réelles possibles et à réfléchir à de futures extensions.

La suite de ce document est découpée en cinq parties. La première d'entre elles est consacrée au logiciel `SmartSocket` et présente son fonctionnement et ses possibilités. Une grande partie est ensuite dédiée au service répliqué. Elle contient une description de son l'architecture, de la manière dont il est intégré aux `SmartSockets` ainsi que la formalisation en pseudo-code de toutes les primitives de communication. La troisième partie présente l'implémentation du service répliqué en Java qui doit permettre de faire le parallèle entre le code et l'architecture théorique. Ensuite se trouve un chapitre qui expose les tests réalisés et comments les résultats. Et, finalement une dernière partie est consacrée à deux développements futurs qui pourraient être rajoutés au service répliqué pour en généraliser son utilisation.

Ce travail pratique de diplôme qui s'inscrit dans le prolongement d'un projet de semestre, a été réalisé au Département de Systèmes de Communication (DSC) dans le Laboratoire de Système d'Exploitatoin (LSE) et supervisé par le Prof. André Schiper.

2 Les SmartSockets

2.1 Introduction

Il s'agit d'un logiciel qui permet de mettre en œuvre une communication de type `publish/subscribe` nécessaire, par exemple, pour une application répartie. Il offre une certaine tolérance aux pannes et garantit la livraison des messages qu'il route. SmartSockets fonctionne dans un réseau hétérogène qui peut être constitué de machines de types différents. Il permet aussi une communication point à point avec les mêmes propriétés. La figure ci-dessous (Figure 1) montre où s'insère SmartSockets dans la structure multi-couches d'une application

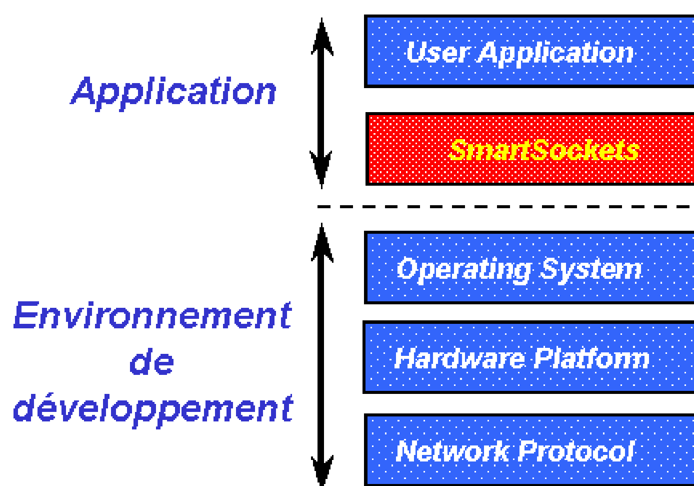

Figure 1 – Les couches de développement

Ce schéma montre que SmartSockets est indépendant de l'environnement puisqu'il se place à un haut niveau, au-dessus du système d'exploitation en offrant des interfaces de programmations variées.

Il faut maintenant décrire plus précisément comment fonctionne une application utilisant SmartSockets. Il faut distinguer deux acteurs : les RTClients qui regroupent tous les `publishers` et les `subscribers` et les RTServers, dont le rôle est d'acheminer les messages à leurs destinataires. L'architecture d'une application utilisant SmartSockets a typiquement l'allure suivante (Figure 2).

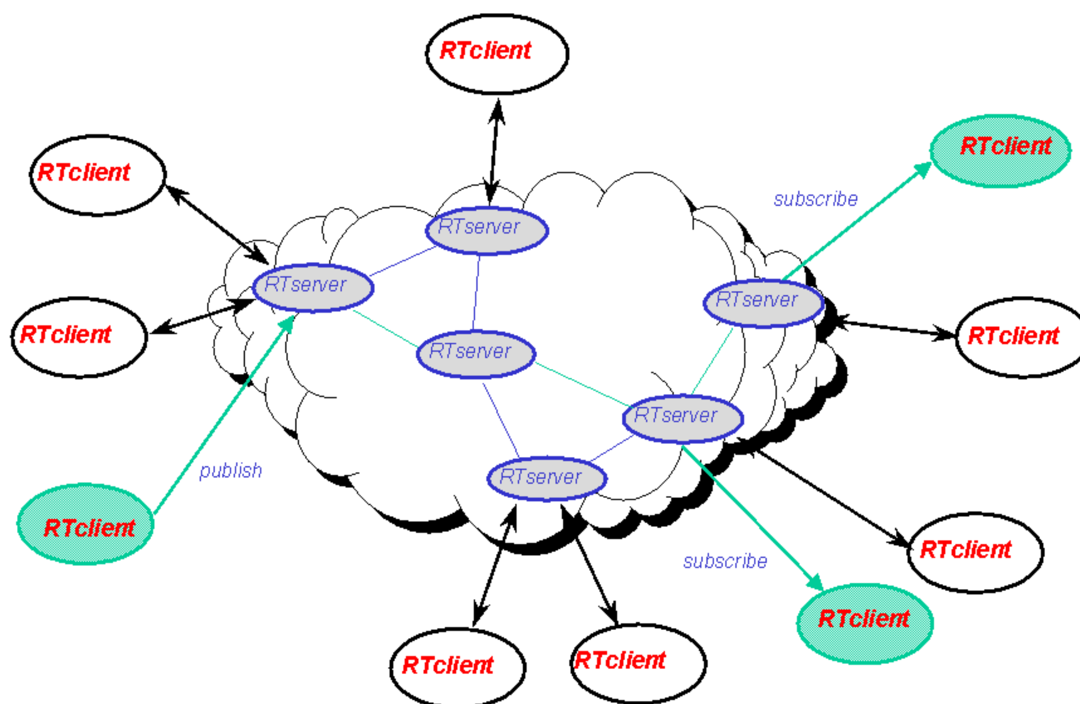

Figure 2 – Architecture des SmartSockets

Le nuage au centre représente un réseau dynamique de RTServers. Autour gravitent les RTClients, chacun connecté à un RTServer. Cette architecture est dynamique car il est possible d'ajouter ou de supprimer en tout temps des RTServers et des RTClients.

Le principe de fonctionnement est simple. Un RTClient qui publie un message sur le sujet `toto` le fait en l'annonçant au RTServer auquel il est connecté. Ensuite, le message va être routé vers tous les RTServers auxquels des RTClients, intéressés par le sujet `toto`, sont connectés.

Les messages utilisés sont structurés ce qui permet d'échanger facilement de l'information entre des machines de types différents. Le message lui-même est transparent et ce n'est qu'au moment de la réception qu'il est converti dans un format utilisable par le RTClient. Les messages peuvent être envoyés de manière synchrone (envoi bloquant) ou asynchrone (envoi non-bloquant).

SmartSockets existe pour la plupart des types de machine (Vax, Mac, PC, Unix, …), fonctionne de manière transparente sur des réseaux hétérogènes (LANs, WANs, Internet, ..) et met à disposition un support multi-platformes en proposant des APIs variées comme Java, C, C++ ou ActiveX.

Il est aussi possible d'utiliser des connections sécurisées qui garantissent la confidentialité des envois de messages. Cette possibilité ne sera pas présentée dans ce document.

Finalement, l'utilisateur dispose d'un outil graphique de maintenance et de configuration de l'application. Cette interface permet de visualiser tous les RTClients ainsi que les RTServers auxquels ils sont connectés. On peut voir aussi comment la charge du réseau est répartie, quels sont les RTServers les plus sollicités et le nombre de messages qui ont passé par chaque connexion. Cette interface ne sera pas décrite dans ce document.

2.2 Les sujets

Pour pouvoir organiser de manière performante les informations, SmartSockets propose une structure hiérarchique des sujets. Cette architecture arborescente est similaire à celle que l'on trouve pour contrôler un système de fichiers.

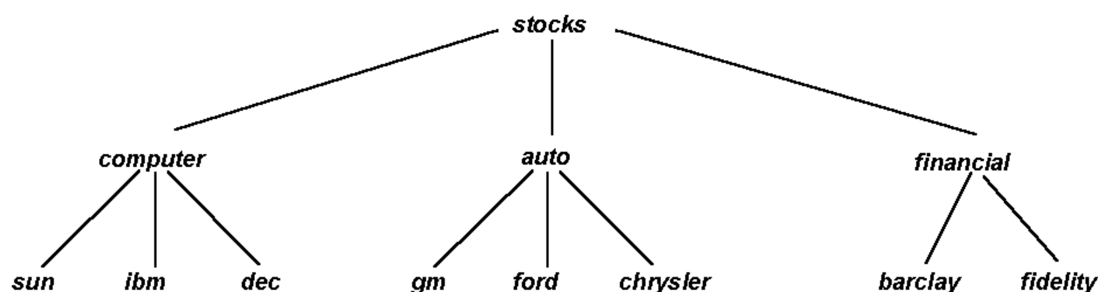

Figure 3 – Structure arborescente des sujets

Chaque RTServer conserve cette structure des sujets et la met à jour dynamiquement. Pour les RTClients, la désignation des sujets se fait comme pour un fichier.

/stocks/auto/ford

Cette structure permet aussi un accès généralisé aux sujets. Par exemple, si un RTClient est intéressé par toutes les actions dans le domaine de l'automobile sans préférences de marques, il peut simplement souscrire au sujet suivant :

/stocks/auto/*

Il recevra ainsi tous les messages publiés sur l'un des sous-sujets de *`/stocks/auto`*. Il est aussi possible d'être encore plus général en souscrivant par exemple aux sujets suivants :

/stocks//ibm*	Sujet « ibm » pour autant qu'il soit un sous-sujet de « stocks ».
//ibm*	N'importe quel sujet désigné par « ibm ».
/stocks/…/barclay	Sujet « barklays » s'il se trouve 2 niveaux en dessous de « stocks ».

Avec SmartSockets, chaque RTClient qui se connecte à un RTServer doit préciser le nom du projet auquel il participe. C'est-à-dire que l'espace de nom utilisé pour désigner les sujets est propre à un projet particulier. Un RTClient reçoit donc tous les messages publiés sur les

sujets auxquels il a souscrit, pour autant que lui et l'auteur du message appartiennent au même projet. Ce mécanisme permet d'avoir deux structures de sujets identiques mais de les séparer en les englobant dans deux projets différents. Par défaut tous les RTClients utilisent un même nom de projet.

Ce n'est pas la seule manière de structurer les sujets utilisée par les logiciels de type *publish-subscribe*, mais elle a l'avantage d'ordonner de manière simple et rapide les différents thèmes. Néanmoins cette façon de faire a certains désavantages. Par exemple. il n'est pas possible de relier deux sujets s'il n'appartiennent pas au même sous-arbre (Figure 4).

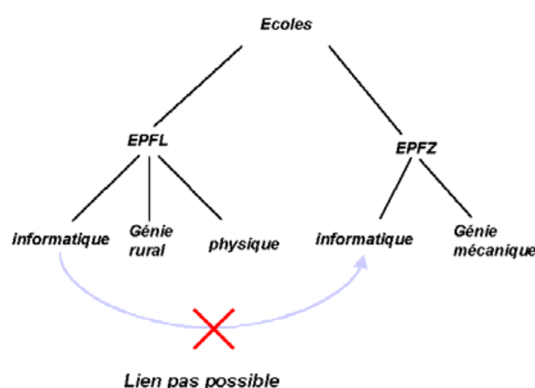

Figure 4 – Limites de la structure arborescente des sujets

2.3 Les messages

Pour pouvoir permettre d'échanger des messages dans un environnement hétérogène, SmartSockets utilise des messages structurés et typés. Chacun d'eux comporte deux parties. La première partie contient les propriétés du message comme son type, celui qui a publié l'information, le sujet auquel il est destiné, etc. La deuxième représente le contenu du message.

En-tête Propriétés	Type	QUOTE	
	Sender	/_lsesun13_5134	
	Destination	/stock/auto/ford	
	Priority	10	
	Read Only	FALSE	
	Delivery Mode	NONE	
	Delivery Timeout	30	
	Load Balance Mode	ROUND_ROBIN	
	Sequence Number	9842	
	Reference Count	1	
	User Property	0	
Contenu dépendant du type du message	Data	Str	ford
		Real4	133.000

```
TipcMtCreate("Quote", 500, "{str real4}");
```

Figure 5 - Structure d'un message

La notion de type de message est importante, car c'est avec ce mécanisme qu'il devient possible de créer et de lire des messages avec des machines différentes. En fait, le type décrit la grammaire du contenu du message. Ici, par exemple, le type est « QUOTE ». On le définit comme au bas de la Figure 5 en lui donnant un nom (« Quote »), un numéro (500) et en fixant le type des différents champs qu'il contiendra. Dans cet exemple les messages de type « QUOTE » contiendront un champ « chaîne de caractères » (*string*) et un champ « nombre réel » (*float*). De cette manière, le fait que la représentation des nombres à virgules flottantes soit différente d'une machine à l'autre n'a aucune importance. Un message peut être vu comme un paquet d'informations structuré envoyé d'un processus vers un autre. Cela facilite aussi le traitement des messages en permettant de les trier selon leur type.

SmartSockets met aussi à disposition un mécanisme de queue de messages. Cela permet de garantir que le premier message arrivé sera le premier délivré. Ce qui est intéressant c'est que la gestion de la queue tient compte de la priorité des messages. En effet, un message avec une plus grande priorité peut passer devant les autres et être délivré en premier.

2.4 Guaranteed Message Delivery

Le mécanisme *Guaranteed Message Delivery* (GMD) est une propriété que possède un message indiquant qu'un certain degré de tolérance aux pannes est souhaité. Avec SmartSockets, il est possible d'obtenir une confirmation qu'un message publié a bien été délivré aux destinataires intéressés. Cela exige, par exemple, qu'au moment d'être envoyés, les messages soient stockés provisoirement sur disque ou en mémoire. Le *publisher* reçoit un signal *ACK* lorsqu'il a été correctement délivré et est donc prévenu si son message doit être renvoyé.

Un message est envoyé avec l'un des trois types de GMD suivants :

1. **Sans GMD.** L'utilisateur n'a pas besoin de l'assurance que son message a été délivré
2. **GMD partielle.** Le message doit être délivré à certains de ses destinataires
3. **GMD standard.** Le message doit être délivré à tous les destinataires intéressés.

Ce mécanisme qui évite au programmeur de traiter des cas complexes de recouvrement nécessite néanmoins la mise en place d'un protocole particulier.

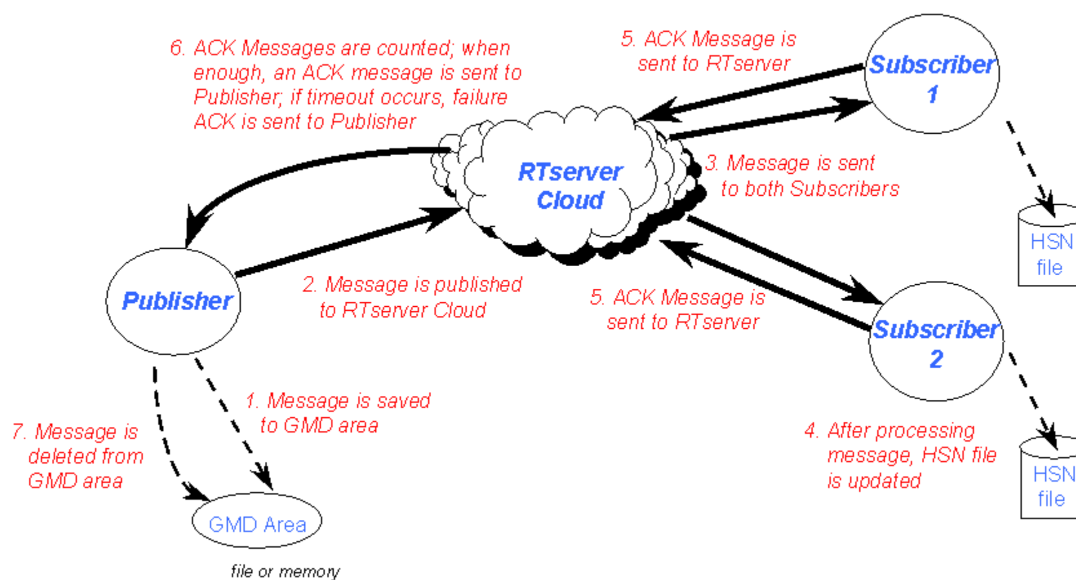

Figure 6 – Protocole GMD

1. Premièrement, le message à envoyer est stocké dans la `GMD area`. Il s'agit d'un espace sur disque ou en mémoire, selon les besoins de l'application, dans lequel les messages sont stockés au cas où ils devraient être réémis.
2. Le message est publié sur le nuage de RTServers. Il est routé vers tous ses destinataires.
3. Le message est délivré à tous les RTClients.
4. Chaque RTClient met à jour une table de hachage (`HSN`) qui conserve l'identificateur de chaque message délivré. Ceci évitera de traiter deux fois le même message.
5. Chaque RTClient envoie un message `ACK` ou confirmation de réception.
6. Ces messages `ACK` sont comptés et, en fonction du type de GMD utilisé (sans GMD, GMD partielle, GMD standard), un `ACK` global est envoyé au `publisher` qui a envoyé le message.
7. Ce dernier peut alors retirer le message de la `GMD area`.

Si le RTClient qui a publié le message ne reçoit pas de confirmation de livraison après un certain temps, une procédure d'erreur est exécutée qui peut, par exemple, décider de réémettre le message. Hélas, la documentation à disposition ne donne aucun détail sur la manière dont le GMD est réellement implémenté. L'information importante est que, si un message est publié, alors il sera reçu par tous les destinataires ou par aucun.

Finalement, pour le cas où le RTServer, auquel un RTClient est connecté, tombe en panne, il existe un mécanisme de *timeout* qui envoie régulièrement un type de message particulier appelé requête *keep alive*. Typiquement après 30 secondes d'inactivité, chaque RTClient envoie une requête *keep alive* à son RTServer pour vérifier s'il fonctionne encore. Si, après un certain délai, le RTServer n'a pas répondu, le RTClient le considère comme défaillant et rompt la connexion. Cette caractéristique permet de préserver l'intégrité de l'application en évitant que les RTClients ne dépendent que de leur RTServer actuel.

2.5 Load balancing

Une dernière caractéristique importante des SmartSockets est appelée *Load balancing*. Cela permet de répartir la charge du réseau en ne délivrant les messages qu'à un seul de ses destinataires. Cette propriété est très utile pour les applications réparties, comme nous le verrons plus loin.

Si l'on suppose que les trois RTClients de droite (Figure 7) constituent un service répliqué. La désignation de la copie primaire, quel que soit le nombre dynamique de copies, est faite automatiquement avec le *Load balancing*.

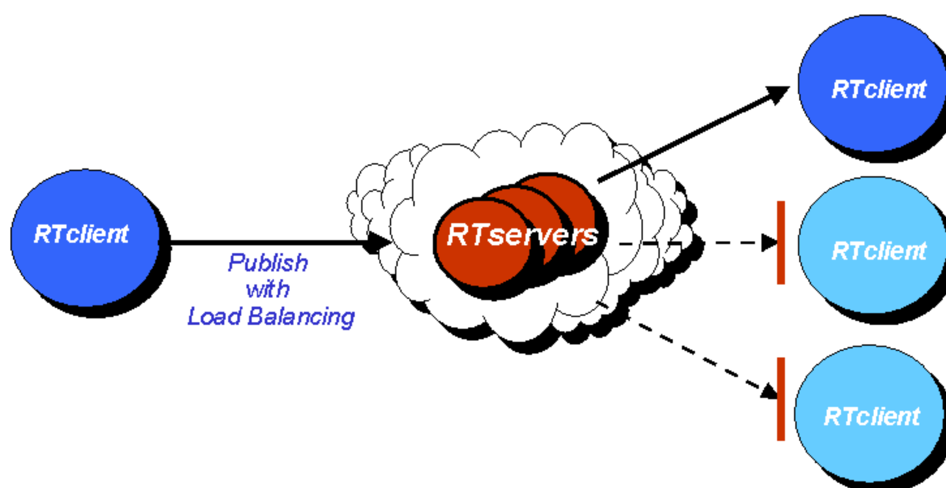

Figure 7 – Load balancing

Il y a trois manières de désigner le RTClient qui va recevoir le message :

1. **Round Robin.** Chaque message est délivré à un autre RTClient en parcourant successivement la liste des destinataires.
2. **Moindre coût.** Il est possible d'assigner un poids à chaque connexion, en fonction de la charge du réseau, par exemple. Ce mécanisme est utilisé pour trouver le RTClient qui recevra le message, situé le plus près de l'émetteur, au sens du moindre coût.
3. **Trié.** Chaque RTClient, lorsqu'il se connecte à un RTServer, reçoit un identificateur unique. Dans ce cas, le message est délivré au RTClient qui a l'identificateur le plus petit.

2.6 Les options

Cette section décrit brièvement les options qui peuvent être utilisées pour personnaliser le fonctionnement des RTClients. Elles sont importantes car elles peuvent largement influencer le comportement, la vitesse ou la réaction aux pannes de l'application. Elles seront présentées ici telles qu'on les trouve avec l'API Java.

- ss.default_msg_priority
  La priorité par défaut des messages
- ss.default_subject_prefix
  Le préfixe par défaut des sujets (par ex : /mon_application)
- ss_enable_control_msgs
  Plusieurs types de messages sont dits de contrôle car ils permettent, par exemple, de provoquer la terminaison d'un RTClient à distance. Cette option sert à définir les types de messages de contrôle qu'un RTClient accepte.
- ss.project
  Le nom du projet auquel appartient le RTClient.
- ss.server_auto_connect
  Une valeur booléenne qui indique qu'un RTClient doit automatiquement se connecter au RTServer dès qu'une opération de lecture ou d'écriture est entreprise.
- ss.server_auto_flush_size
  La taille du tampon (*buffer*) d'envoi.
- ss.server_keep_alive_timeout
  Le temps qu'un RTClient attend la réponse à une requête *keep alive* avant de déclencher une procédure d'erreur.
- ss.server_max_reconnect_delay
  Le délai maximum que va attendre le RTClient avant de se reconnecter au RTServer après avoir été déconnecté. Le vrai délai sera une valeur aléatoire comprise entre 0 et le délai maximal. Ceci permet de réduire la charge du RTServer en répartissant les reconnections dans le temps.
- ss-server_names
  Une liste de nom de RTServer auxquels un RTClient peut se connecter. Si une connexion est détruite, le RTClient essaie de se reconnecter au RTServer suivant dans cette liste.
- ss.server_read_timeout
  Le temps, sans trafic, qu'un RTClient attend avant d'envoyer une requête *keep_alive*.
- ss.server_connect_timeout
  Le temps qu'un RTClient attend avant que la réponse à sa demande de connexion n'arrive.
- ss.unique_subject
  Le *unique subject* du RTClient. Il s'agit d'un identificateur qui désigne de manière univoque un RTClient
- ss.user_name
  Le nom d'utilisateur associé au RTClient

3 Le service répliqué

3.1 Introduction

3.1.1 Généralités

Dans le but d'assurer la persistance d'un service, il est possible de constituer un groupe de réplicats (ie. copies identiques) qui multiplient les supports où l'information est stockée, de manière à pouvoir garantir sa persistance. C'est-à-dire que si l'un des réplicats est défaillant, les autres peuvent continuer à assurer le bon fonctionnement du service. Ce mécanisme introduit une certaine tolérance aux pannes, mais pose un problème supplémentaire qui consiste à assurer la cohérence entre les états respectifs des réplicats. Pour que les copies soient cohérentes, le groupe répliqué doit satisfaire les deux propriétés suivantes [Sch00] :

Ordre. Soit deux opérations o_1 et o_2 sur un objet x.
Si deux copies x_1 et x_2 effectuent les deux opérations o_1 et o_2, alors elles le font dans le même ordre.

Atomicité. Soit une opération o sur un objet x.
Si une copie de x effectue l'opération o, alors toutes les copies correctes (non en panne) du groupe effectuent également l'opération o.

Il existe deux techniques de réplication : la copie primaire (« primary backup », en anglais) et la réplication active [Sch00]. Avec la première, un client envoie sa requête à un réplicat particulier appelé copie primaire. Cette dernière traite la demande du client, envoie un message de mise à jour aux autres copies du groupe puis envoie sa réponse au client.
Avec la réplication active, toutes les copies reçoivent la requête du client, la traitent puis lui répondent. Le client poursuit son exécution dès qu'il a reçu la première réponse. La réplication active impose un traitement déterministe des requêtes.

3.1.2 Architecture

Le but de ce projet est de développer une architecture qui soit capable d'assurer une communication de groupe qui garantit la cohérence des états respectifs des réplicats. Le service répliqué, qui sera adjoint aux `SmartSockets`, doit pouvoir traiter une requête d'un client indépendamment du nombre de copies. La solution la mieux adaptée aux `SmartSockets` est la technique de la copie primaire car `SmartSockets` ne satisfait pas la propriété d'ordre énoncée au paragraphe 3.1.1. Le paradigme `publish/subscribe` peut, par contre, être utilisé pour l'échange de messages entre les clients et la copie primaire. Un client va publier sur un sujet particulier un message adressé au service, tandis que les copies auront toutes souscrit à ce même sujet. Le mécanisme du `Load balancing`, prévu initialement pour répartir la charge du réseau (cf §2.5), va être utilisé pour désigner la copie primaire. Il permet de ne délivrer les messages qu'à certains des `RTClients` qui ont souscrit au sujet. Pour le service répliqué, le `Load balancing` de type `round robin` est tout à fait

adapté car il permet de désigner automatiquement la copie primaire du groupe. Parallèlement, cela réparti les requêtes des clients entre tous les réplicats. Cette utilisation des `SmartSockets` rend la réplication du service totalement transparente aux `RTClient` qui y accèdent. Ils ignorent le nombre de copies et à laquelle ils s'adressent.

Pour la mise en œuvre de la communication intra-groupe, il n'est plus possible d'utiliser `SmartSocket` qui n'offre pas les garanties de fiabilité et d'ordonnancement nécessaires. Cette partie va devoir être assurée par une architecture encore à définir, basée sur UDP (cf. § 3.2.5), qui devra offrir des primitives assurant le transport fiable et ordonné de l'information.

Le but sous-jacent de cette implémentation est de pouvoir garantir la cohérence des copies constituant le service répliqué. Il faut absolument éviter que des messages ne soient délivrés qu'à une seule copie à l'exclusion des autres, ou qu'ils ne soient pas délivrés dans le même ordre.

La figure Figure 9 donne une vue très générale de l'architecture du service répliqué.

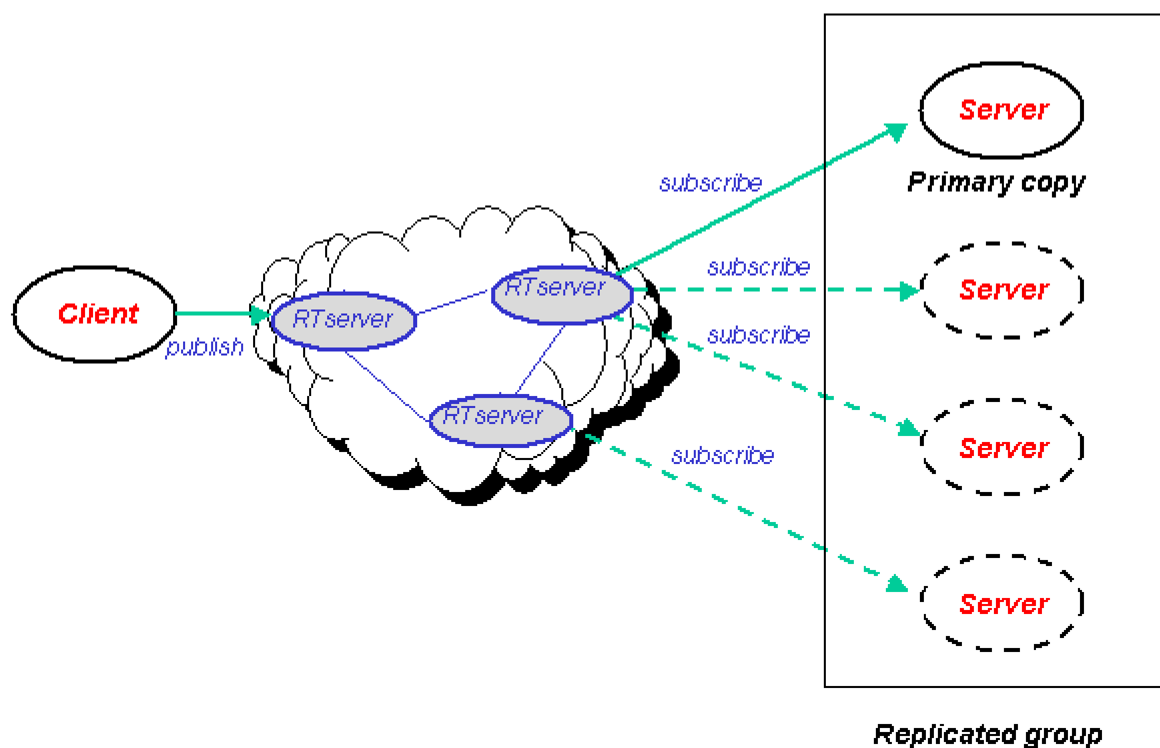

Figure 8 – Architecture générale

Les `RTClients`, à droite, qui constituent le service répliqué sont à partir de maintenant appelés `Server`. Ce sont naturellement des `RTClient` de `SmartSocket` qui joue le rôle de fournisseur d'information, par conséquent de serveur.

Un client, à gauche, est connecté à un RTServer. Il peut publier un message à l'intention du service répliqué sur un sujet connu, défini à l'avance. Le message est routé à travers le nuage de RTServer. L'utilisation du `load balancing` (cf. §2.5) permet aisément de ne délivrer le message qu'à un seul des serveurs intéressés par le sujet. L'application développée utilise un `load balancing` de type `round robin` ayant pour effet de changer de copie primaire à chaque réception de message. Cela permet de répartir de manière équitable la charge du réseau entre les différentes copies. Le message reçu par la copie primaire doit encore être propagé de manière fiable et ordonnée dans le groupe. C'est dans ce but qu'a

été définie une architecture munie de plusieurs primitives de communication assurant la communication intra-groupe.

En partant de la Figure 2, on peut insérer les couche nécessaire au service répliqué comme dans la figure Figure 9 dans laquelle on trouve, à côté de la couche SmartSockets, une couche de protocole et une couche d'erreur. Une application qui souhaite utiliser un service répliqué dispose maintenant des SmartSockets pour la communication client-groupe et d'une nouvelle partie (`protocol + failure layer`) qui assurera la communication à l'intérieur du groupe répliqué en offrant plusieurs primitives et mécanismes assurant un transport fiable de l'information. Les deux étant bien entendu encore basées sur un environnement de développement particulier.

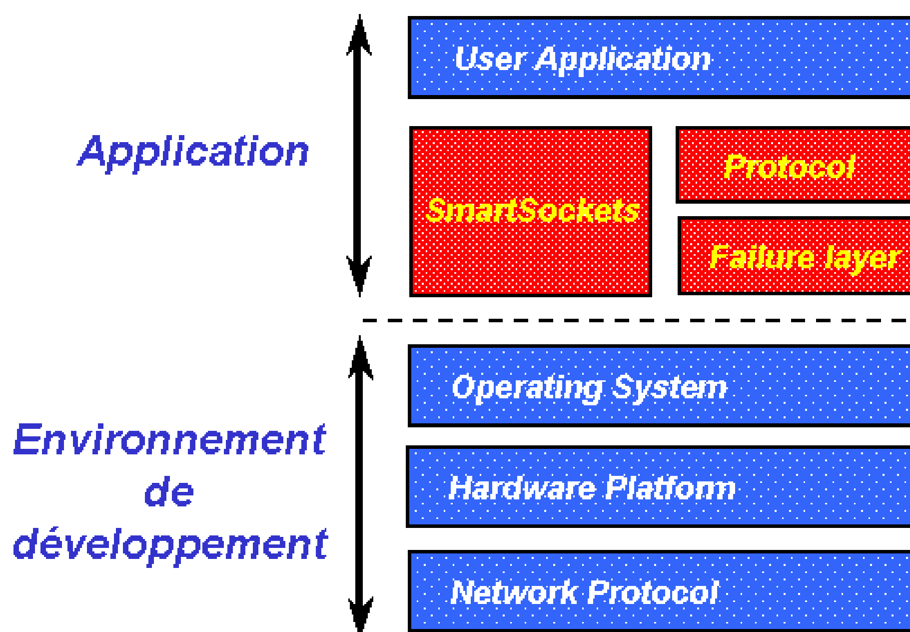

Figure 9 – Architecture du service répliqué

3.2 L'architecture d'un réplicat

La Figure 10 illustre de manière plus détaillée les deux nouveaux blocs `protocol` et `failure layer`. On retrouve sur la figure la couche applicative tout en haut ainsi qu'une couche appelée couche UDP qui regroupe les trois blocs « Environnement de développement » présents dans la Figure 9.
Chacun des réplicats, membres du groupe répliqué, possède une telle architecture. Elle lui permet de communiquer aussi bien avec les clients du groupe qu'avec les autres membres.

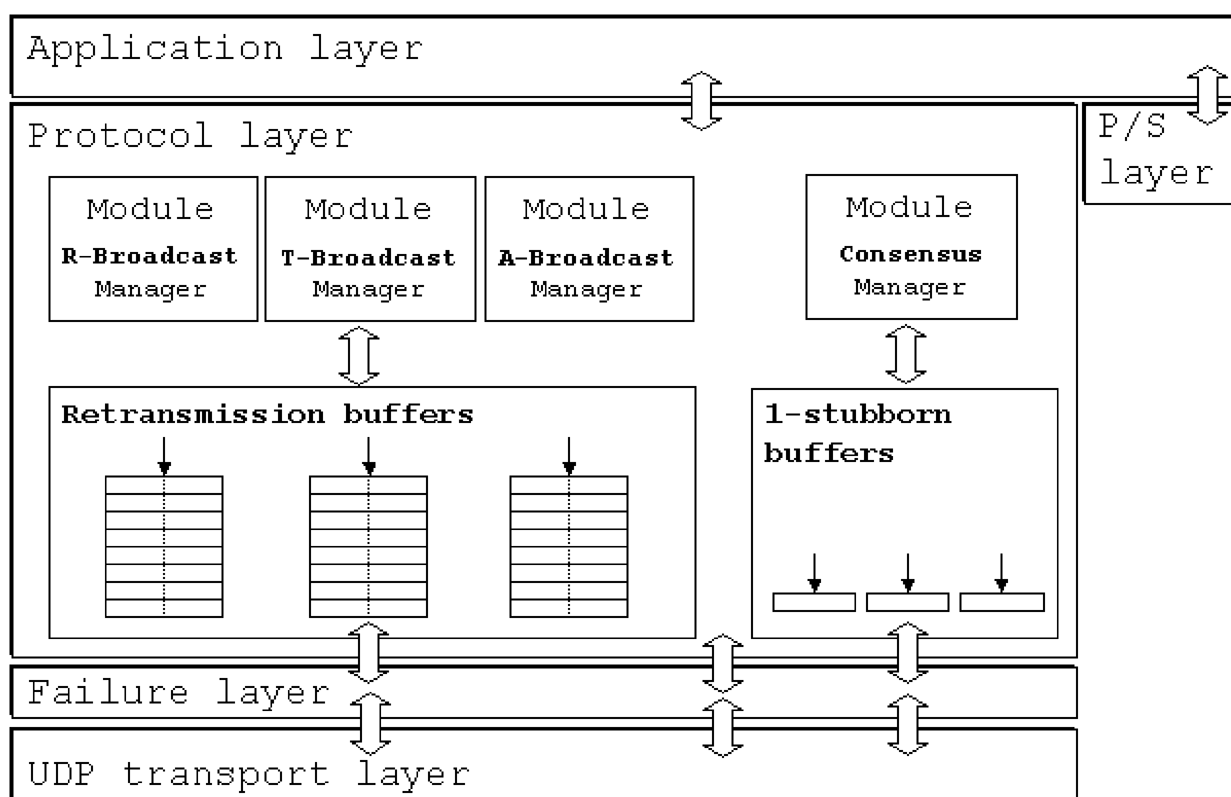

Figure 10 – Architecture d'un réplicat

3.2.1 Application layer

La couche supérieure représente la couche applicative. C'est dans cette dernière que sera exécuté le code qui souhaite utiliser le service répliqué. Cette couche offre deux moyens de communication différents : la couche de protocole (« `protocol layer` ») pour l'échange de messages à destination des autres membres du groupe, et l'accès au RTServer, et par conséquent à tous les `RTClients` de SmartSocket grâce à la couche nommée « `P/S layer` ».

3.2.2 Protocol layer

La couche de protocole est le centre du service répliqué. C'est elle qui doit offrir à la couche applicative les différentes primitives de communication. Chacune de ces primitives est traitée par un module portant son nom. Chacun des modules contient un algorithme Cette couche contient aussi deux types de `buffers`, utilisés pour la retransmission des messages. Une description détaillée de chacune des primitives figure plus loin dans ce document (§3.4, 3.6 et 3.7).

3.2.3 P/S layer

Cette couche offre à l'application la possibilité d'accéder au RTServer auquel elle est connectée. Cela lui permet de recevoir et de publier des messages à destination des clients du service répliqué.

3.2.4 Failure layer

Cette couche n'a pas de raison d'être à terme car elle n'est là que pour simuler la présence de pannes ou d'erreurs de transmission. Elle est néanmoins très utile au moment de vérifier la fiabilité de l'application.

3.2.5 UDP transport layer

Cette couche regroupe toutes celles, dans la Figure 9 de plus bas niveau que la couche de protocole, jusqu'au matériel. Elle offre un moyen simple d'envoyer des paquets de données vers des machines distantes. Le protocole UDP qui est mis en œuvre par cette couche signifie « User Datagram Protocol ». Il est complètement décrit dans la RFC 768 (i.e « Request For Comment » 768) [RFC768]. Dans les grandes lignes, il s'agit d'un protocole basé sur IP (i.e. « Internet Protocol ») qui permet d'envoyer des paquets structurés, appelés « datagram packets ». Ces paquets contiennent cinq champs, en plus des en-têtes IP : le port de l'émetteur, le port du destinataire, la longueur totale du paquet, une somme de contrôle (vérification partielle de l'intégrité des données) et les données proprement dites. Le protocole UDP n'assure ni la protection contre la duplication des paquets, ni leur livraison. C'est donc pour cette raison, qu'il est nécessaire d'ajouter une couche de protocole, au-dessus d'UDP, pour assurer le transport fiable et ordonné des messages.
Pourquoi alors, ne pas avoir utilisé TCP (i.e. « Tranmission Control Protocol ») qui, lui, assure une livraison point-à-point ordonnée et fiable des messages ? Il y a globalement quatre raisons. Premièrement, une architecture basée sur TCP est beaucoup plus complexe à mettre en œuvre. En effet, avec UDP, chaque processus ne possède qu'un seul `socket` qu'il peut utiliser pour l'émission et la réception de messages à destination de plusieurs processus. L'information de destination est propre à chaque paquet. Avec TCP, il est nécessaire de connecter entièrement les processus entre eux avec des `sockets`. Cela signifie que pour un groupe répliqué de n processus, il faut n `sockets` avec UDP et n^2 avec TCP. Cela rend la mise en place de l'architecture beaucoup plus complexe.
Un deuxième désavantage de TCP est lié à la gestion des erreurs. La défaillance d'un processus et donc la rupture d'un `socket` lève une exception rendant le recouvrement excessivement complexe. De plus, elle exige la création d'un nouveau `socket` ; opération inutile avec UDP. Mais le plus ennuyeux est qu'il est impossible de savoir quels messages ont été reçus par le processus défaillant avant la rupture du `socket` , ce qui nécessite donc la mise en œuvre d'un mécanisme d'acquittement similaire à celui qui doit de toute façon exister avec UDP.
Troisièmement, les `sockets` qualifiés de `multicast` (i.e. à destinataires multiples) qui existent avec UDP sont très bien adaptés à la communication de groupe. En effet, il est possible de n'envoyer qu'un seul message dont l'adresse de destination est celle d'un groupe de processus. Cette fonctionnalité n'existe pas avec TCP.
Finalement, le protocole TCP est beaucoup plus lent que UDP et n'assure de pas l'ordre total qui est souhaité d'une manière ou d'une autre pour un groupe répliqué.

3.3 Les messages

Les messages échangés entre les clients et les serveurs, de même qu'à l'intérieur du groupe sont des objets complexes. Cela permet une plus grande flexibilité de contenu et offre un moyen plus générique d'échange d'informations. Globalement, les messages sont composés de trois parties, le type, un en-tête et un contenu.

1. Type. Pour pouvoir distinguer les messages entre eux, et donc savoir à qui en incombe le traitement, les messages sont typés. Le type d'un message définit sa structure, c'est-à-dire le nombre de champs qu'il contient et leur signification.
2. En-tête. Il s'agit d'un nombre variable de champs, comme l'identificateur du message, son émetteur, ainsi que plusieurs champs dépendant de son type.
3. Contenu. Il représente ce que le message contient effectivement en tant qu'information à transmettre (requête, réponse, fichier, données, etc.).

Pour être envoyés, les messages sont sérialisés, c'est-à-dire transformés en un tableau d'octets aisément transmissible grâce à la couche UDP. A l'inverse, lors de sa réception, un nouvel objet message est construit et initialisé avec un tableau d'octets. La figure Figure 11 schématise cette transformation.

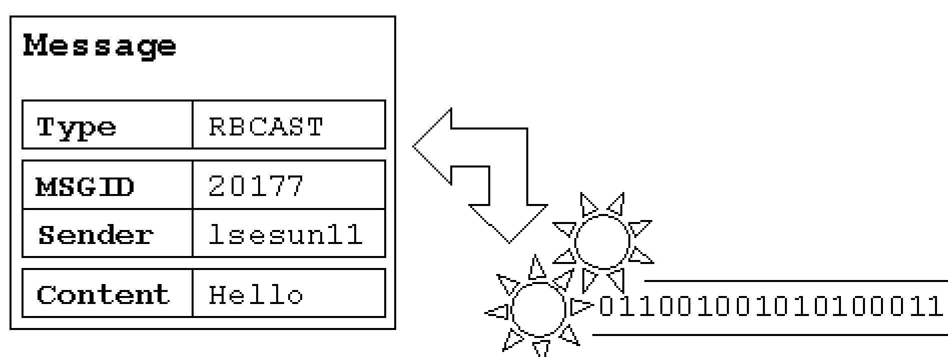

Figure 11 - Message

Tous les champs d'un message (en-tête et contenu), sont nommés. C'est-à-dire que leur ordre n'a aucune importance puisqu'il n'est possible d'y accéder qu'en les nommant. Il suffit donc de stocker un ensemble de couples (clé, valeur) dans lesquels la clé désigne, sous la forme d'une chaîne de caractères, le nom du champ auquel on souhaite accéder.

3.3.1 Les types de messages

Comme dit ci-dessus, les messages sont typés. Cela permet de pouvoir effectuer un traitement particulier lors de l'arrivée d'un nouveau message en provenance de la couche de transport UDP. En fait, il existe dans l'architecture un composant spécifique, appelé « trieur » auprès duquel chacun des modules peut s'inscrire comme destinataire privilégié d'un type de message. Il faut remarquer que le trieur n'apparaît pas explicitement dans les schémas de l'architecture, étant donné qu'il n'a aucun sens algorithmique et ne viendrait que surcharger la figure. Néanmoins, son rôle est très important. La couche UDP s'adresse à lui chaque fois qu'un nouveau message est reçu. Le trieur va ensuite extraire le type du message et

transmettre ce dernier au module intéressé. Cela permet de tolérer qu'un grand nombre de messages indépendants les uns des autres soient transmis et reçus simultanément. Les types de message définis sont :

CLIENT_MSG	Messagse échangés entre les clients et le groupe répliqué
RBCAST	Messages échangés durant l'exécution d'un `Reliable Broadcast`.
RBCAST_ACK	Messages d'acquittement pour un RBCAST
TBCAST	Messages échangés durant l'exécution d'un `Terminated Broadcast`.
TBCAST _ACK	Messages d'acquittement pour un TBCAST.
SP	Messages S_p échangés durant l'exécution d'un `Terminated Broadcast`.
ABCAST	Messages échangés durant l'exécution d'un `Atomic Broadcast`.
UDP_ACK	Messages d'acquittement standard
START_CONSENSUS	Messages échangés au démarrage d'un consensus
CONSENSUS	Messages échangés durant l'exécution d'un consensus.
CONSENSUS_ACK	Messages d'acquittement pour un CONSENSUS.

3.4 La primitive Reliable Broadcast

3.4.1 Description

La première étape du projet consiste à implémenter la primitive de communication de groupe *R-BroadCast* qui doit assurer la livraison des messages à tous les destinataires.
Plus formellement, le `Reliable Broadcast` d'un message *m* à un ensemble de processus *P* (avec *émetteur(m)* $\in$ *P*) doit satisfaire les trois conditions suivantes [Mul93]:

1. **Validité.** Si un processus correct exécute *R-Broadcast(m)*, alors tous les processus corrects de `P` finiront par délivrer `m`.
2. **Agrément.** Si un processus correct délivre un message *m*, alors tous les processus corrects de *P* finiront par délivrer *m*.
3. **Intégrité.** Pour tout message *m*, tous les processus corrects de *P* délivrent *m* au plus une fois, et seulement si *émetteur(m)* a auparavant exécuté *R-Broadcast(m)*.

L'implémentation du *Reliable Broadcast* ne représente pas une grande difficulté. Il suffit d'assurer que les messages envoyés sont systématiquement réémis jusqu'à la réception d'un acquittement. Cela suppose que les messages émis avec *R-Broadcast* sont, en plus d'être envoyés à tous les membres du groupe (*Broadcast*), placés dans un *buffer* de retransmission, dont le contenu est réenvoyé périodiquement. Lorsqu'un acquittement *ack(m)* pour un message *m* est reçu, il suffit de retirer m du *buffer* de retransmission.

3.4.2 L'architecture

Ce paragraphe présente l'architecture de communication utilisée pour la réalisation du *R-Broadcast*. La Figure 12 met en évidence les principaux éléments nécessaires. Il s'agit d'une version simplifiée de l'architecture de base.

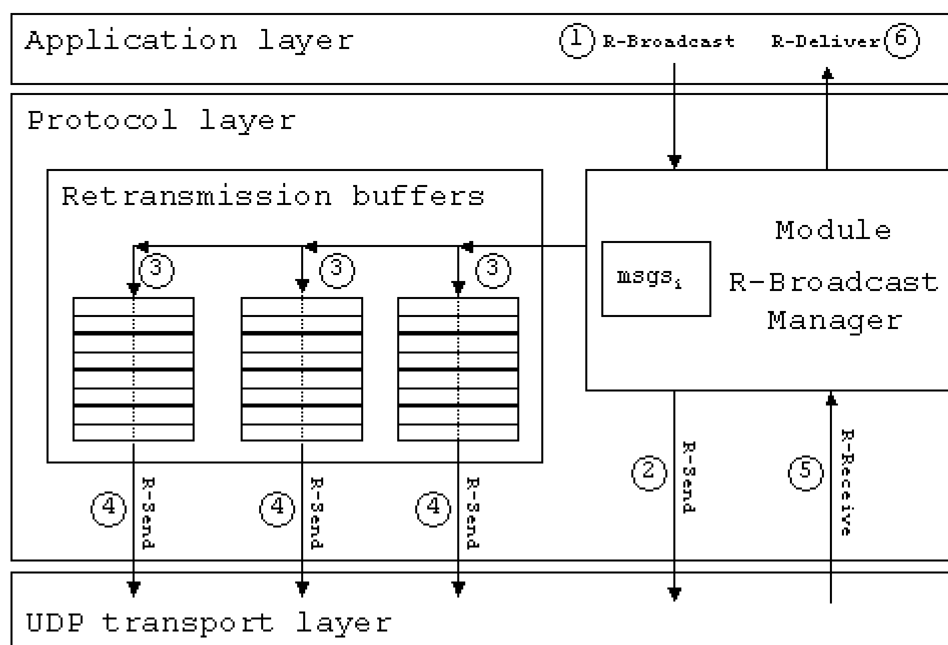

Figure 12 – R-Broadcast Architecture

A droite dans la Figure 12, se trouve le module R-Broadcast Manager qui met en œuvre le protocole d'échange de messages satisfaisant les propriétés de la primitive *R-Broadcast* (cf. paragraphe 3.4.1). Le module *R-Broadcast Manager* « s'inscrit » auprès de la couche de transport UDP pour recevoir tous les messages de type *R-Broadcast*. Cela signifie qu'à la réception d'un message de ce type, il est automatiquement (c-à-d. sans traitement) transmis au module. Sur la gauche du schéma, on trouve les *buffers* de retransmission que le module utilise pour retransmettre les messages non acquittés.
Le protocole fonctionne de la manière suivante. Les messages envoyés avec *R-Broadcast* (point 1) à partir de la couche applicative sont diffusés à tous les membres du groupe (point 2) en utilisant la couche de transport UDP en même temps qu'ils sont placés dans les *buffers* de retransmission (point 3). Ces derniers, stockent provisoirement les messages en les retransmettant périodiquement (point 4).
A la réception d'un message de la couche UDP (point 5), il est délivré à la couche applicative (point 6). Parallèlement, un acquittement contenant un identificateur du message est envoyé à l'émetteur qui pourra retirer le message du *buffer* de retransmission.

3.4.3 L'algorithme

Voici l'algorithme exécuté par le module R-Broadcast Manager. Les deux événements à décrire sont l'appel de R-Broadcast par la couche applicative (point 1 de la Figure 12) ainsi que la réception d'un message transporté par la couche UDP (point 5 de la Figure 12).

```
1   module R-Broadcast manager for a process pi
2   {
3     buffersi[] ;   //buffersi[j] ≡ retransmission buffer of pi to pj.
4     msgsi;         //messages received.
5
6     function R-Broadcast(msg) {
7       R-Send(msg) to all processes pj ≠ pi
8       msgsi = msgsi ∪ {(pi, msg)};
9       R-Devlier(msg) ;
10      ∀j, buffersi[j] = buffersi[j] ∪ {(msg)};
11    }
12
13    function main {
14      while(true) {
15        select {
16          upon R-Receive(pj, msg):
17            R-Send(ack(msg)) to pj;
18            if(msgsi ∩ {(pj, msg)} = ∅) {
19              msgsi = msgsi ∪ {(pj, msg)};
20              R-Deliver(msg);
21            }
22          upon R-Receive(pj, ack(msg)):
23            buffersi[j] = buffersi[j] / {msg};
24        }
25      }
26    }
27  }
```

Algorithme 1 – Reliable Broadcast

Les *buffers* de retransmission situés dans la couche de protocole sont appelés *buffers* dans l'algorithme (lignes 3, 8, 21 et 24).

Aux lignes 6 à 9, on trouve le pseudo-code de la fonction *R-Broadcast*, qui prend, en paramètre le message à envoyer. Son implémentation est relativement simple, il suffit de le transmettre, via la couche de transport UDP, à tous les processus du groupe (ligne 7) et de le placer dans les *buffers* de retransmission (ligne 9).

La fonction `main`, décrite aux lignes 11 à 27, est démarrée lors de la création du module. Son rôle est d'attendre la réception d'un message en provenance de la couche UDP et de transmettre périodiquement le contenu des `buffers` de retransmission (lignes 23 et 24). Lors de la réception d'un message envoyé avec `R-Broadcast` (lignes14 à 19), le processus envoie un acquittement contenant un unique identificateur du message reçu (ligne 15). S'il est reçu pour la première fois, il est ajouté à l'ensemble des messages reçus (ligne 17) et délivré à la couche applicative (ligne 18). Si le nouveau message est un acquittement (lignes 20 et 21), on ne fait que retirer le message acquitté du *buffer* de retransmission de l'émetteur (ligne 21).

3.4.4 Justification de l'algorithme

Validité

Si un processus correct exécute `R-Broadcast(m)`, alors il finira par délivrer `m`.

En effet, supposons que p_i, processus correct, exécute R-Broadcast(m) (ligne 6). Il va placer `m` dans les `buffers` de retransmission de chacun des processus de la vue courante (ligne 8), notamment le sien. Il finira donc par délivrer `m` (ligne 18), qui sera retransmis jusqu'à ce qu'il soit acquitté par p_i (ligne 15).

Agrément

Si un processus correct délivre un message `m`, alors tous les processus corrects de `P` finiront par délivrer `m`.

Supposons que p_i, un processus correct de la vue, délivre `m`. Cela signifie que `m` a été envoyé par un processus p_j (ligne 7) ou retransmis par p_j (ligne 24). Lorsque p_j a exécuté `R-Broadcast(m)`, `m` a été envoyé à tous les processus appartenant à `P` (ligne 7) et placé dans chacun des `buffers` de retransmission à destination des processus de `P` (ligne 8). Donc si p_i a délivré le message `m`, alors tous les processus de `P` finiront par délivrer `m`.

Intégrité

Pour tout message `m`, tous les processus corrects de `P` délivrent `m` au plus une fois, et seulement si `émetteur(m)` a auparavant exécuté `R-Broadcast(m)`.

Supposons que p_i, processus correct de `P`, reçoit un message `m` pour la première fois (ligne 14). Seul un message reçu pour la première fois, c'est-à-dire pas encore contenu dans le `buffer` de réception $msgs_i$ (ligne 16), est délivré (ligne 18). Et si `m` est reçu par p_i, c'est qu'il a été envoyé par un processus p_j qui a exécuté `R-Broadcast(m)` (lignes 6 à 9).

3.4.5 Exemples de chronogrammes

Ce paragraphe présente, sous forme de chronogrammes, le comportement de l'algorithme de *`R-Broadcast`*. Cela doit permettre de mieux comprendre son fonctionnement dans différentes situations.
Le cas le plus standard est celui ou le message n'a pas besoin d'être réémis. Tous les processus de la vue courante ont reçu le message envoyé avec *`R-Broadcast`*. La Figure 13, dans laquelle on suppose l'existence de trois réplicats illustre cette situation. Le *`R-Deliver(m)`* a lieu dès que *`m`* est reçu. Immédiatement après, un acquittement est envoyé à l'émetteur qui peut alors retirer *`m`* de son *`buffer`* de retransmission.

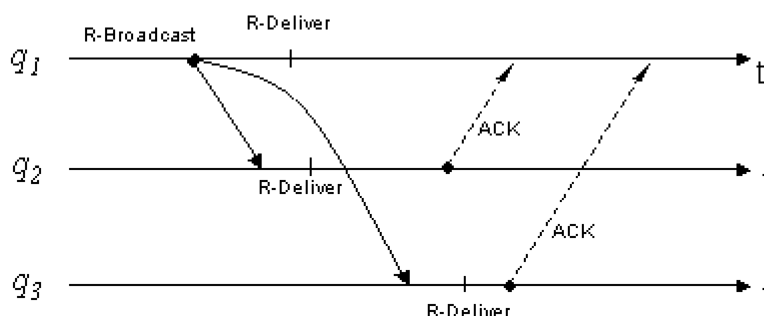

Figure 13 – Exécution normale de R-Broadcast

Si le message d'acquittement est perdu ou envoyé trop tard, il est simplement réémis par l'émetteur à intervalles réguliers. Par exemple, les messages contenus dans les `buffers` de retransmission sont envoyés toutes les 500 millisecondes. Si un message est reçu deux fois par un processus de la vue, le deuxième est simplement ignoré. Cela nécessite de conserver une trace (par ex. un identificateur) de tous les messages reçus. La figure ci-dessous illustre cette situation.

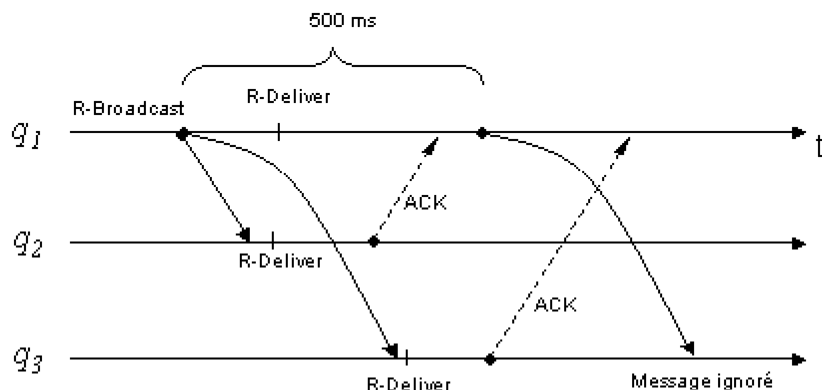

Figure 14 – Exécution de R-Broadcast avec retransmission

3.5 Algorithme du consensus

3.5.1 Description

Pour de l'implémentation de primitives de communication dans un groupe répliqué et pour en assurer la cohérence, on est bien souvent confronté au problème de l'agrément. Le but est de pouvoir assurer qu'un ensemble de processus s'est mis d'accord sur une même propriété, un même ensemble de messages ou un même état. L'agrément permet une sorte de synchronisation entre les processus. Dit de manière simplifiée, chaque processus participant au consensus propose une valeur (par ex. un message ou une suspicion de panne) et ont dit que tous les processus corrects « décideront » de manière irrévocable une même valeur parmi celles initialement proposées. Plus formellement, le consensus doit satisfaire les trois propriétés suivantes :

1. **Terminaison.** Tout processus correct finira par décider une valeur.

2. **Agrément uniforme.** Deux processus ne peuvent pas décider deux valeurs différentes.
3. **Validité uniforme.** Si un processus décide v, alors v était une proposition de l'un des processus.

Ces trois propriétés dépendent du sens que l'on donne au qualificatif correct. Dans notre cas, un processus correct est un processus qui ne tombe pas en panne, qui ne « crashe » jamais.
De plus, le problème du consensus peut être plus ou moins difficile à résoudre en fonction des hypothèses émises à propos des possibilités de défaillances. En effet, dans un système synchrone, où les temps de calcul et les délais de communication sont bornés et connus, la résolution du problème du consensus est facile [Oli00]. A l'opposé, dans un système totalement asynchrone dans lequel la détection de pannes est très complexe, le problème du consensus devient très difficile à résoudre ou même impossible.
En 1985, Fischer, Lynch et Paterson ont prouvé qu'il était impossible de résoudre le problème du consensus dans un système asynchrone dès qu'un seul processus est défaillant (i.e. pas correct). Ce résultat est connu sous le nom de l'impossibilité FLP.
Dans notre cas, cela pose un important problème, car le système est asynchrone et le transformer en un système synchrone serait bien trop contraignant.
Heureusement, il existe une autre option qui consiste à renforcer le modèle asynchrone de manière à pouvoir résoudre le consensus. C'est l'approche qui a été proposée par Chandra et Toueg en 1991. Ils ont proposé d'étendre le modèle asynchrone en y ajoutant la notion de détecteur de fautes. Ils ont montré que si le détecteur de fautes satisfait certaines propriétés, alors le problème du consensus peut être résolu dans un contexte asynchrone.

Les détecteurs de fautes

Le but de cette section n'est pas de détailler les possibilités et les avantages des différents types de détecteurs de fautes, mais de présenter celui qui a été utilisé pour l'implémentation de notre algorithme du consensus.
De manière générale, un détecteur de fautes est un module attaché à chacun des processus qui maintient à jour une liste des processus suspectés. Un détecteur de faute peut se tromper en soupçonnant à tort un processus correct. Il peut aussi changer d'avis, c'est-à-dire suspecter un processus au temps `t` et ne plus le suspecter au temps `t+1`. Finalement, les détecteurs de fautes peuvent être en désaccord et donc ne pas suspecter le même ensemble de processus au même moment. Un détecteur de fautes est caractérisé par deux propriétés : le complétude et la justesse (« completeness » et « accuracy », en anglais).

- **Complétude.**
  - **Complétude forte.** Tout processus défaillant finira par être suspecté pour toujours par tous les processus corrects (i.e. non défaillants).
  - **Complétude faible.** Tout processus défaillant finira par être suspecté pour toujours par au moins un processus correct.
- **Justesse.**
  - **Justesse forte.** Aucun processus correct n'est jamais suspecté.
  - **Justesse faible.** Au moins un processus correct n'est jamais suspecté.

- **Justesse forte inéluctable.** Il existe un instant t à partir duquel aucun processus correct n'est pas suspecté.
- **Justesse faible inéluctable.** Il existe un instant t à partir duquel au moins un processus correct n'est pas suspecté.

On peut alors définir plusieurs classes de détecteurs de fautes qui sont fréquemment utilisées.

$\diamond\mathcal{S}$: satisfait la complétude forte et la justesse faible inéluctable.
$\diamond\mathcal{W}$: satisfait la complétude faible et la justesse faible inéluctable.
$\diamond\mathcal{P}$: satisfait la complétude forte et la justesse forte inéluctable.
$\mathcal{P}$ (parfait) : satisfait la complétude forte et la justesse forte.

Les « Stubborn buffers »

Une autre caractéristique novatrice de cet algorithme est l'utilisation de canaux de communication « bornés » (« stubborn », en anglais).
En fait, le temps nécessaire à la terminaison des algorithmes du consensus utilisant des détecteurs de fautes basés sur la justesse faible, comme $\diamond\mathcal{S}$, ne peut pas être borné. Il en découle que le nombre de messages échangés dans ce type d'algorithmes peut être très grand. En plus, on souhaite aussi que les messages soient envoyés de manière fiable ce qui nécessite l'utilisation de tampons dont la taille ne fait que croître linéairement.
Les réflexions menées sur ce sujet montrent, en fait, qu'il n'en est rien. En effet, il semble plutôt qu'un nouveau message envoyé par un processus rend le plus souvent la livraison du précédant inutile [Oli00]. En partant de cette observation, il devient possible de définir un nouveau type de canaux de communication qui sont appelés `stubborn buffers`. Un algorithme qui utilise le protocole `stubborn`, limite à k le nombre de messages qui peuvent être mis dans un `buffer`. On dit alors que l'on utilise des `k-stubborn buffers`. L'algorithme qui utilise ce protocole a la responsabilité de décider ce qu'il faut faire si un ancien message doit être écrasé ou non par l'arrivée d'un nouveau.
L'algorithme du consensus, implémenté dans le service répliqué, utilise un protocole `1-stubborn`. Pour résumer, il s'agit de `buffers` de retransmission de taille 1. Si un nouveau message est envoyé avant que le précédent ait été acquitté, il le remplace simplement.

3.5.2 L'architecture

Comme pour le `R-Broadcast`, il a été défini un nouveau module situé dans la couche de protocole, responsable de l'exécution de l'algorithme du consensus. Deux procédures d'envoi et de réception de messages ont été définies. La primitive `S-Send` permet d'envoyer un message selon le protocole `1-stubborn` et `S-Receive` permet la réception d'un message en provenance de la couche UDP. La Figure 15, extraite de l'architecture de base, illustre les principaux éléments.

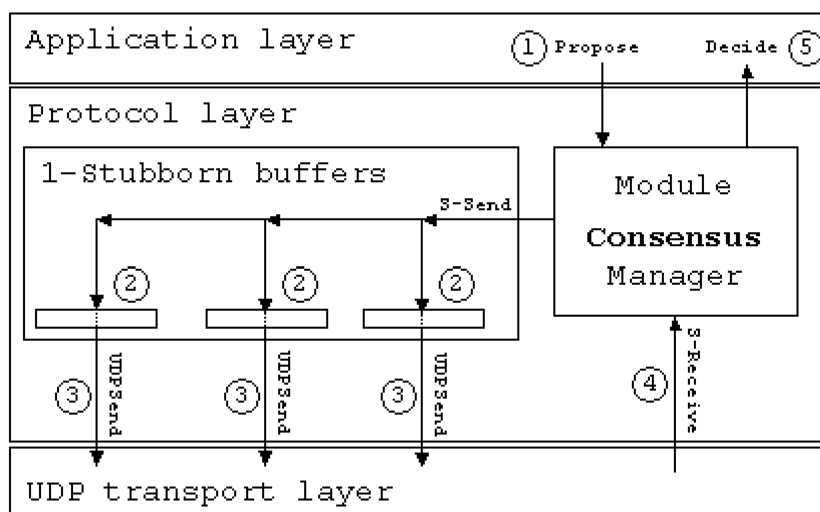

Figure 15 – Consensus architecture

Un consensus est démarré avec l'appel de la procédure *propose* (point 1) à qui il faut donner une valeur initiale en paramètre. Tous les messages échangés durant le consensus sont envoyés à l'aide des *1-stubborn buffers* (point 2) dont le contenu, de même que celui des buffers de retransmission, est envoyé périodiquement (point 3). L'exécution de l'algorithme est structurée en rondes. Pour chaque ronde, parmi tous les processus de la vue courante, un coordinateur est élu de manière déterministe. Le but est de faire accepter (i.e. décider) la valeur initiale du coordinateur par tous les autres participants au consensus. Si ce coordinateur est défaillant, il sera suspecté par un mécanisme de *timeout*. Si tel est le cas, tous les processus changent de ronde et élisent un nouveau coordinateur. Finalement, la décision est retournée à la couche applicative avec l'appel de la procédure *decide*, qui contient la décision.

3.5.3 L'algorithme

Chaque processus invoque la fonction propose de l'Algorithme 2 avec sa propre valeur initiale v_i. La fonction se termine dès que p_i exécute l'instruction *return* (lignes 24 et 49). On dit alors que p_i décide. Cet algorithme suppose l'existence d'un détecteur de faut appartenant à la classe $\diamond\mathcal{S}$.
Chaque ronde de l'algorithme est divisée en deux phases, *PHASE1* et *PHASE2*. Dans la *PHASE1* de chaque ronde *r*, tous les processus essaient d'adopter la proposition du coordinateur. Si le coordinateur de la ronde *r* est suspecté, alors tous les processus tentent de le destituer en entrant dans la *PHASE2*. Cette dernière sert à définir un nouveau consensus qui aura lieu dans la ronde *r + 1*. Il faut encore préciser que la valeur initiale d'un processus p_i dans la ronde *r + 1* est la valeur d' *estimate* que p_i avait à la fin de la *PHASE2*.
Tous les messages échangés durant l'exécution d'un consensus sont constitués de quatre champs :

p_i	L'identité de l'émetteur.
$estimate_i$	Une estimation de la décision.

r_r	La ronde d'émission du message.
$phase_i$	Le numéro de la phase (PHASE1 ou PHASE2)

Voici le pseudo-code de l'algorithme utilisé. Une description plus complète se trouve dans [Oli00].

```
1   module Consensus manager
2   {
3     function propose(v_i) {
4       estimate_i = (i,v_i);
5       r_i = 0;
6       phase_i = PHASE1;
7       while(true) {
8         roundTerminated_i = false;
9         coord_i = (r_i mod n) + 1;
10        if(i = coord_i) {
11          S-Send(p_i,estimate_i,r_i,PHASE1);
12        }
13        while(roundTerminated = false)       {
14          select {
15            upon S-Receive(p_j,estimate_j,r_j = r_i,PHASE1) and phase_i = PHASE1:
16              if(msgs_i = ∅ and p_j ≠ coord_i){
17                estimate_i = estimate_j;
18                S-Send(p_i,estimate_i,r_i,PHASE1);
19              }
20              msgs1_i = msgs1_i ∪ {(p_j,r_i,PHASE1,estimate_j)};
21              if(|msgs1_i| = ⌈(n+1)/2⌉){
22                S-Send(p_i,estimate_i,r_i,DECIDE);
23                decide(esitmate_i.value);
24                return;
25              }
26            upon coord_i ∈ D_i and phase_i = PHASE1
27              phase_i = PHASE2;
28              S-Send(p_i,estimate_i,r_i,PHASE2);
29            upon S-Receive(p_j,estimate_j,r_j=r_i,PHASE2):
30              if(phase_i = PHASE1){
31                phase_i = PHASE2;
32                S-Send(p_i,estimate_i,r_i,PHASE2);
33              }
34              msgs2_i = msgs2_i ∪ {(p_j,r_i,PHASE2,estimate_j)};
35              if(estimate_i.process = coord_i) {
36                estimate_i.value = estimate_j.value;
37              }
38              if(|msgs2_i| = ⌈(n+1)/2⌉){
39                estimate_i.process = I; r_i = r_i + 1; phase_i = PHASE1;
40                roundTerminated_i = true;
41              }
42            upon S-Receive(p_j,estimate_j,r_j>r_i,PHASE1 | PHASE2):
43              estimate_i = (i,estimate_j.value);
44              r_i = r_j; phase_i = PHASE1;
45              roundTerminated_i = true;
46            upon S-Receive(p_j,estimate_j,r_j,DECIDE):
47              S-Send(p_i,estimate_j,r_j,DECIDE);
48              decide(estimate_j.value);
49              return;
50          }
```

```
51              }
52          }
53      }
54  }
```

Algorithme 2 – Consensus

3.5.4 Justification

Une justification complète de cet algorithme se trouve dans [Oli00].

3.5.5 Exemples de chronogrammes

Dans l'exemple de la Figure 16, on trouve une exécution sans problème de l'algorithme du consensus. Au départ, tous les processus démarrent un consensus en proposant leur propre valeur initiale. Ensuite, q_3, le coordinateur, envoie sa proposition à tous puis, q_1 et q_2 essaient de le soutenir en renvoyant sa proposition. Dès qu'un processus a reçu une majorité de messages, il peut décider et terminer l'algorithme.

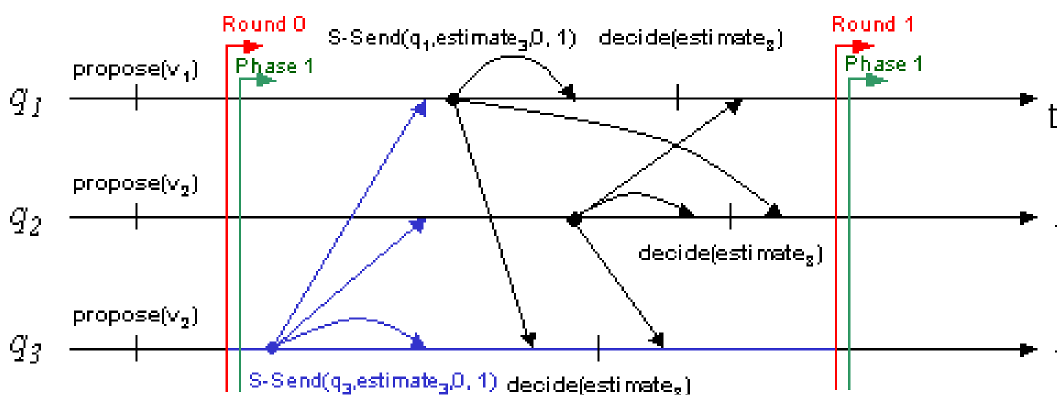

Figure 16 – Exécution de l'algorithme du consensus

Le chronogramme suivant montre une exécution où le coordinateur est défaillant et devient suspecté après un certain laps de temps. Le réplicat q_2 est le premier à suspecter q_3, et envoie un message invitant tous les processus à passer dans la phase 2. Le nouveau coordinateur q_2 émet sa proposition, soutenue par q_1 et tous deux décident.

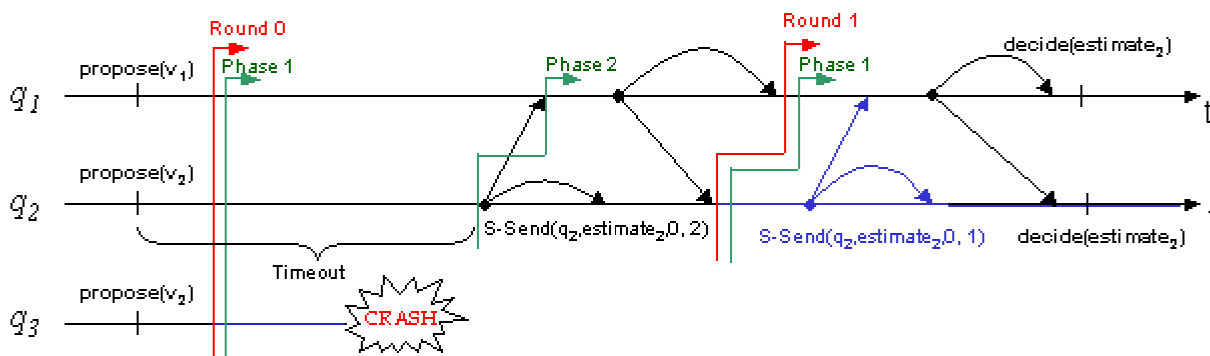

Figure 17 - Exécution de l'algorithme du consensus avec crash du coordinateur

3.6 La primitive Terminated Broadcast

3.6.1 Description

La primitive *Terminated Broadcast*, plus tard appelée *T-Broadcast*, permet d'envoyer un message dans un environnement où le temps de livraison des messages n'est pas borné. Cela est typiquement le cas avec l'architecture de communication basée sur UPD dont nous disposons. L'envoi de message avec T-Broadcast est très similaire au mécanisme utilisé à celui du R-Broadcast. Mais le problème est qu'il est impossible de prédire quand l'exécution du R-Deliver correspondant pourra avoir lieu.
Actuellement, cette nouvelle primitive ne dispose pas d'une spécification formelle. Néanmoins, son principe de fonctionnement peut tout de même être clairement décrit.
Premièrement, les messages envoyés avec T-Broadcast sont délivrés dans l'ordre FIFO (« First In First Out »). Cela signifie que tous les messages en provenance d'un même processus du groupe sont délivrés dans le même ordre. Plus formellement T-Broadcast satisfait la propriété suivante [Mul93].

FIFO order. Si un processus exécute T-Broadcast(m_1) avant T-Broadcast(m_2), alors aucun processus correct de P ne délivre m_2 avant d'avoir délivré m_1.

De plus, lorsqu'un message m est envoyé avec T-Broadcast, tous les membres du groupe, indépendamment les uns des autres, décident d'attendre le T-Deliver(m) correspondant pendant un certain temps Δt_i. Il y a maintenant deux solutions. Soit tous les processus de P ont exécuté T-Deliver(m), soit l'intervalle de temps Δt_i s'est écoulé pour un processus p_i. Ce dernier va démarrer un consensus après avoir construit sa valeur initiale. Dans ce but, il va envoyer son ensemble $unstable_i$, défini comme ci-dessous, à tous et va attendre une majorité de messages similaires en réponse des autres processus du groupe.

stable$_i$. Un message m appartient à $stable_i$ si et seulement si p_i sait que $\forall p_k \in P$ p_k a reçu m.
unstable$_i$. Un message m appartient à $unstable_i$ si et seulement si p_i a reçu m et m n'appartient pas à $stable_i$.

Finalement, p_i va démarrer un consensus en proposant, comme valeur initiale, la réunion des ensembles $unstable_k$ reçus. La décision du consensus sera donc un ensemble de messages que p_i va parcourir pour délivrer ceux qui ne le sont pas encore. Et ce n'est qu'après cette étape que p_i, et les autres processus de P, vont délivrer un message particulier, appelé S_p, qui signifie que l'émetteur du message m est suspecté. L'éventuel traitement particulier que nécessiterait la livraison de S_p est laissé aux soins du module qui exécute un T-Broadcast. Aucun traitement n'est fait dans le T-Broadcast Manager. Cet

algorithme permet de s'assurer, qu'à partir de la livraison de S_p, tous les processus de P ont délivré exactement les mêmes messages.

3.6.2 L'architecture

La figure ci-dessous, illustre l'architecture d'un réplicat pour l'exécution d'un Terminated Broadcast, ds laquelle n'apparaîssent que les éléments nécessaires. On trouve le module T-Broadcast Manager qui pour assurer l'envoi des messages utilise les buffers de retransmission. De plus, s'il a besoin de démarrer un consensus, il peut interagir avec le module Consensus Manager à qui il faut les 1-stubborn buffers pour l'émission de ses propres messages.

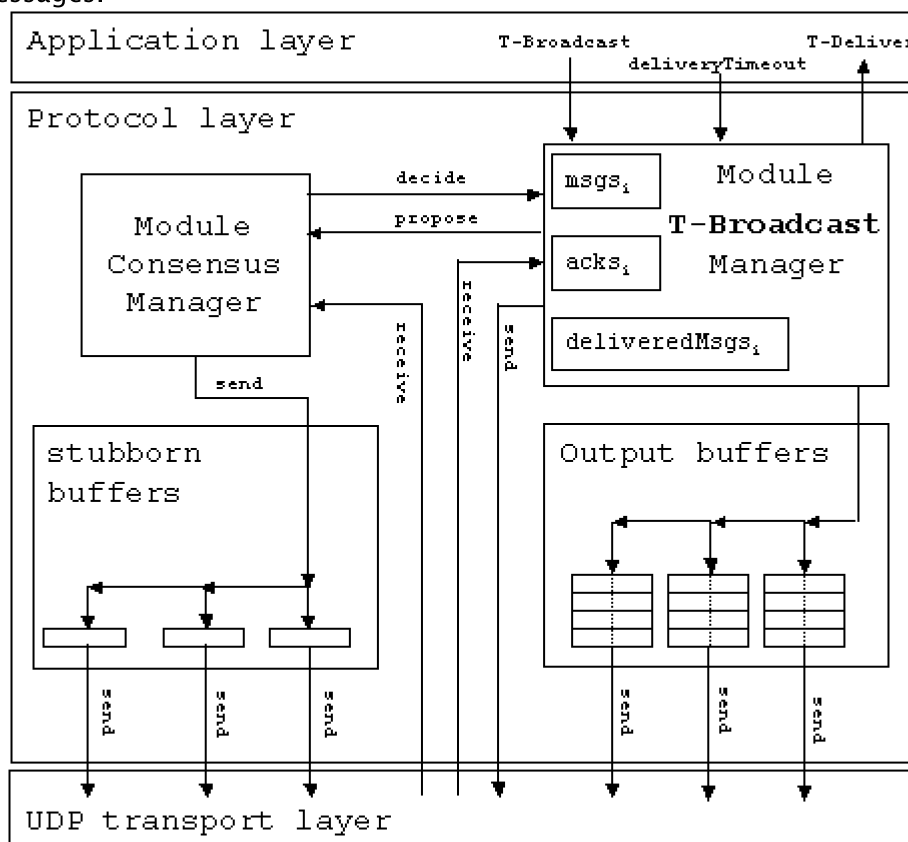

Figure 18 – Terminated Broadcast architecture

3.6.3 L'algorithme

Voici l'algorithme exécuté dans le module T-Broadcast Manager en pseudo-code. Ce dernier contient quatre fonctions. La première, T-Broadcast, est appelée par la couche applicative ou un autre module pour exécuter un Terminated Broadcast. La deuxième, main, est la fonction qui est démarrée lors de la création du module T-Broadcast Manager. Il s'agit d'une boucle infinie qui réceptionne et traite les différents messages en provenance de la couche de transport UDP. La troisième, decide, est utilisée par le module consensus Manager pour communiquer la décision d'un consensus. Et finalement, la méthode deliveryTimeout, appelée si le laps de temps fixé pour l'attente de l'exécution du T-Deliver est écoulé. Le module à besoin de sept variables pour fonctionner.

`buffers`$_i$`[k]`	Référence sur le `buffer` de retransmission à destination de $p_k \in P$
`unstable`$_i$	Ensemble de messages tel que décrit au paragraphe 3.6.1
`TdeliveredMsgs`$_i$	Ensemble des messages délivrés dans le module `T-Broadcast Manager`
`acks`$_i$`(id(m))`	Ensemble des acquittements pour un message `m` particulier.
`proposition`$_i$	La valeur initiale proposée lorsqu'un consensus est démarré.
`lastMsgID`$_i$`[k]`	Identificateur de dernier message délivré et émis par p_k.
`nextConsensusID`$_i$	Identificateur de prochain consensus à démarrer.

```
1   module T-Broadcast manager for a process pi
2   {
3     buffersi[] ;               //buffersi[j] ≡ retransmission buffer of pi to pj.
4     unstablei= ∅ ;             //unstable messages.
5     TdeliveredMsgsi = ∅ ;      //T-Delivered messages.
6     acksi = ∅ ;                //acknowledgments received.
7     propositioni= ∅ ;          //consensus initial value.
8     lastMsgIDi[pk] = 0;        //last message ID T-Delivered by pi sent by pk.
9     nextCOnsensusID = 0;       //id of the next consensus.
10
11    function T-Broadcast(msg) {
12      lastMsgIDi[pi] = lastMsgIDi[pi] + 1 ;
13      send(TBCAST,lastMsgIDi[pi],msg) to all processes pj ≠ pi ;
14      ∀j, buffersi[j] = buffersi[j] ∪ {(msg)};
15      unstablei = unstablei ∪ {(pi, msg)};
16      send(TBCAST_ACK,pi, id(msg)) to all processes pj ≠ pi;
17      acksi[id(msg)] = acksi[id(msg)] ∪ {pi};
18    }
19
20    function main {
21      while(true) {
22        select {
23          upon receive(TBCAST,id(msg),msg) from pj:
24            send(TBCAST_ACK,pj, id(msg)) to all processes pj ≠ pi ;
25            acksi[id(msg)] = acksi[id(msg)] ∪ {pj};
26            if(unstablei ∩ {(pj, msg)} = ∅) {
27              unstablei = unstablei ∪ {(pj, msg)};
28            }
29          upon receive(TBCAST_ACK,pk, id(msg)) from pj:
30            if(pk = pi) {
31              buffersi[j] = buffersi[j] / {msg};
32            }
33            if(acksi(id(msg)) ∩ {pj} = ∅) {
34              acksi[id(msg)] = acksi[id(msg)] ∪ {pj};
35              if(|acksi[id(msg)]| = n) {
36                acksi[id(msg)] = ∅ ;
37                unstablei = unstablei / {msg} ;
38              }
39            }
40            while( ∃ m ∈ unstablei | (id(m) = lastMsgIDi[sender(m)] + 1
41                  and |acksi[id(m)]| >= ⌈(n+1)/2⌉) {
42              T-Deliver(m) ;
43              lastMsgIDi[sender(m)] = lastMsgIDi[sender(m)] + 1 ;
44              TdeliveredMsgsi = TdeliveredMsgsi ∪ {m} ;
45            }
46          upon receive(START_CONSENSUS,consensusID, unstablej) from pj:
47            if(consensusID >= nextConsensusID) {
```

```
48                proposition_i = proposition_i ∪ unstable_j ;
49                if(proposition_i = ∅) {
50                  send(START_CONSENSUS,nextConsensusID,unstable_i)
51                    to all processes p_j ≠ p_i ;
52                  proposition_i = proposition_i ∪ unstable_i ;
53                }
54                if(|proposition_i| >= ⌈(n+1)/2⌉) {
55                  propose(proposition_i,lastMsgID_i) ;
56                }
57              }
58          }
59        }
60      }
61      function decide(CONSENSUS_DECIDE,consensusID,lastMsgID_j, decision_j) :
62              if(consensusID >= nextConsensusID) {
63                nextConsensusID = consensusID + 1 ;
64                while( ∃ m ∈ decision_j | id(m) =  lastMsgID_i[sender(m)] + 1) or
65                     (∃ m ∈ unstable_i | id(m) =  lastMsgID_i[sender(m)] + 1 and
66                      id(m) ≤ lastMsgID_j[sender(m)])) {
67                  T-Deliver(m) ;
68                  lastMsgID_i[sender(m)] = lastMsgID_i[sender(m)] + 1 ;
69                  TdeliveredMsgs_i = TdeliveredMsgs_i ∪ {m} ;
70                }
71                proposition_i = ∅ ;
72              }
73          }
74        }
75      }
76
77      function deliveryTimeout() {
78        send(START_CONSENSUS,nextConsensusID,unstable_i) to all processes p_j ≠ p_i ;
79        proposition_i = proposition_i ∪ unstable_i ;
80      }
81  }
```

Algorithme 3 – Terminated Broadcast

Fonction `T-Broadcast` (lignes 11 à 18). Cette fonction est appelée par la couche applicative ou un module de p_i qui souhaite envoyer un message `msg` avec `T-Broadcast`. Le processus p_i envoie un message de type `TBCAST` contenant `msg` à tous les processus, excepté lui-même (ligne 13). Il place aussi `msg` dans les `buffers` de retransmission à destination des membres du groupe (ligne 14) pour en assurer la livraison. Finalement, puisqu'il ne se l'envoie pas à lui-même, il stocke `msg` dans sont ensemble $unstable_i$ (ligne 15) et envoie un message d'acquittement à tous les autres membres du groupe (ligne16).

Réception d'un message de type `TBCAST` (lignes 23 à 28). Lorsqu'un processus p_i reçoit un message de type `TBCAST` de p_j, il envoie aussitôt un acquittement de type `TBCAST_ACK` à tous les processus du groupe (ligne 24). Si le message reçu n'est pas encore contenu dans l'ensemble $unstable_i$ (ligne 26), p_i l'y ajoute (ligne 27).

Réception d'un acquittement de type `TBCAST_ACK` (lignes 29 à 45). Si p_i est l'émetteur du message `msg` acquitté par p_j (ligne 30), alors il le retire du `buffer` de retransmission à destination de p_j (ligne 31). Ensuite, si l'acquittement est reçu pour le première fois (ligne

33), p_i l'insère dans son ensemble $acks_i$ en créant une entrée correspondant à id(m) (ligne 34). Pour p_i si, avec cet acquittement, tous les processus du groupe ont acquitté msg (ligne 35), il peut alors effacer l'entrée id(msg) dans l'ensemble $acks_i$ (ligne 36) et le retirer de $unstable_i$ (ligne 37). Il ne reste plus qu'à délivrer les messages qui peuvent l'être. C'est-à-dire tous les messages appartenant à $unstable_i$ qui n'ont pas été acquittés par tous les processus (lignes 40 et 41). Les variables lastMsgID et $TdeliveredMsgs_i$ sont mises à jour à chaque fois que T-Delvier est exécuté. A la ligne 43, on trouve l'appel d'une fonction « sender » avec un message m en paramètre. Il s'agit d'un raccourci pour extraire l'expéditeur de m de l'ensemble $unstable_i$ qui est un ensemble de couples du type (sender(m),m).

Réception d'un message de type START_CONSENSUS (lignes 46 à 57). Si l'identificateur du consensus que p_j souhaite démarrer est valide (ligne 47), p_i ajoute à sont ensemble $proposition_i$, tous les messages de $unstable_j$ qui n'y sont pas déjà (ligne 48). Si p_i reçoit pour la première fois un message de type START_CONSENSUS pour le consensus n° consensusID (ligne 49), alors il envoie à tous les processus sont propre ensemble $unstable_i$ (lignes 50 et 51) et l'ajoute à $proposition_i$ (ligne 52). Finalement, si p_i a reçu une majorité de messages START_CONSENSUS (ligne 54), alors il décide de démarrer un nouveau consensus (ligne 55) en faisant appel au module Consensus Manager à qui il transmet l'identificateur du consensus, son ensemble $proposition_i$ et sa variable $lastMsgID_i$.

Fonction decide (lignes 61 à 75). Cette fonction est appelée par le module Consensus Manager lorsque p_i à pu décider un ensemble de messages. De nouveau, il faut vérifier que la décision est bien celle d'un consensus valide (ligne 62). Si tel est le cas, p_i peut mettre à jour sa variable nextConsensusID (ligne 63). Ensuite, tant que l'ensemble $decision_j$ contient un message m qui peut être délivré en respectant l'ordre FIFO (lignes 64), alors m est délivré (ligne 67), inséré dans l'ensemble des messages délivrés (ligne 69) et la variable $lastMsgID_i$ est mise à jour (ligne 68). Si $decision_j$ ne contient pas le prochain message à délivrer, p_i utilise la variable $lastMsgID_j$ qu'il a reçu dans la décision. Elle correspond à l'identificateur du dernier message délivré par le coordinateur. Si p_i possède le prochain message à délivrer dans son ensemble $unstable_i$ p_i, il peut le délivrer pour autant que son identificateur soit inférieur $lastMsgID_j$ (lignes 65 et 66). Cela permet à p_i de délivrer des messages qui l'ont été par les autres processus mais qui ne sont pas contenus dans la décision du consensus. Finalement, l'ensemble $proposition_i$ est vidé pour permettre l'exécution d'un nouveau consensus (ligne 71).

Fonction deliveryTimeout (lignes 77 à 80). Cette fonction est appelée par la couche applicative ou le module qui est en train d'exécuter un T-Broadcast. Elle signifie que le temps maximal fixé pour la réception d'au moins un message est dépassé. Dans ce cas, p_i envoie à tous les processus du groupe, excepté lui-même, un message de type START_CONSENSUS contenant l'identificateur du consensus qu'il souhaite démarrer ainsi que son ensemble de messages $unstable_i$ (ligne 78). Et puisqu'il ne se l'envoie pas à lui-même, p_i ajoute les messages $unstable_i$ à son propre ensemble $proposition_i$ (ligne 79).

3.6.4 Justification de l'algorithme

Comme il n'existe pas de spécification formelle de la primitive Terminated Broadcast, une justification rigoureuse n'est pas envisageable. Le but de ce paragraphe est plutôt de se convaincre que l'Algorithme 3 satisfait les exigences présentées dans la section 3.6.1.

FIFO Order

Si un processus exécute T-Broadcast(m_1) avant T-Broadcast(m_2), alors aucun processus correct de P ne délivre m_2 avant d'avoir délivré m_1.

Supposons qu'un processus p_i exécute T-Broadcast(m_1). Il attribue à m_1 unique identificateur, en incrémentant sa variable $lastMsgID_i[p_i]$ (ligne 12). Par conséquent p_i ne peut pas envoyer deux messages avec le même identificateur. De plus les identificateurs des messages émis par p_i sont strictement croissants et incrémentés de 1 à chaque envoi. Donc si p_i exécute ensuite T-Broadcast(m_2), l'identificateur de m_2 est forcément égal à id(m_1) + 1.
Supposons maintenant qu'un processus p_j correct appartenant à P reçoit le message m_2 en premier. L'identificateur du dernier message reçu par p_j en provenance de p_i, stocké dans la variable $lastMsgID_j[p_i]$ est égal à id(m_2) – 2 puisqu'il n'a pas encore reçu m_1. Le processus p_j ne va donc pas délivrer m_2 avant d'avoir délivré m_1 puisqu'il délivre toujours les messages dans l'ordre de leur identificateur (lignes 37-38 et 61-62). Il en découle donc qu'aucun processus ne peut délivrer m_2 avant m_1 et l'ordre FIFO est garanti.

Supposons maintenant que l'intervalle de temps Δt_i s'est écoulé pour p_i car il n'a pas pu délivrer le message m émis par p_j. Il ne va plus délivrer aucun message et va vouloir démarrer un consensus. Pour cela, p_i va envoyer son ensemble $unstable_i$, qui peut contenir le message m, à tous les autres processus. On suppose, pour simplifier, que le coordinateur p_k du consensus n'est pas défaillant et donc pas suspecté. Si m se trouve dans la décision alors p_i pourra exécuter T-Delvier(m). Si ce n'est pas le cas, p_i va consulter la variable $lastMsgID_k$ qu'il a reçu avec la décision du consensus. Si m est contenu dans $unstable_i$ et que l'identificateur de m est inférieur à $lastMsgID_k$[sender(m)](ie. p_k a déjà exécuté T-Deliver(m)) alors p_i peut aussi exécuter T-Deliver(m). Finalement, si m n'est ni contenu dans la décision ni dans l'ensemble $unstable_i$ de p_i, cela signifie qu'aucun processus n'a exécuté T-Deliver(m) et la cohérence est donc assurée.

3.6.5 Exemples de chronogrammes

La Figure 19 illustre une exécution d'un T-Broadcast sans exécution d'un consensus. Les trois processus du groupe ont pu délivrer m avant que le l'intervalle de temps Δt ne soit écoulé. On constate que l'exécution du T-Deliver(m) a lieu dès la réception du premier acquittement. En fait chaque processus est en possession d'une majorité d'acquittements (ici 2), le sien (implicite) ainsi qu'un autre.

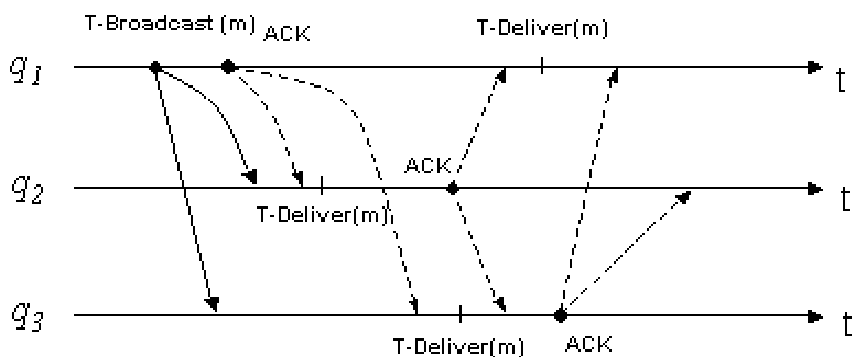

Figure 19 – Exécution d'un T-Broadcast

Dans la Figure 20, le processus q_3 ne reçoit pas les deux acquittements en provenance de q_1 et q_2. Par conséquent il ne peut pas délivrer le message m. Une fois que l'intervalle de temps Δt_3 est écoulé, q_3 envoie son ensemble $unstable_3$ aux autres processus qui répondent de la même manière. Le consensus démarré, où on peut supposer que q_1 est coordinateur, va aboutir sur une décision qui va contenir le message m qui manque à q_1.

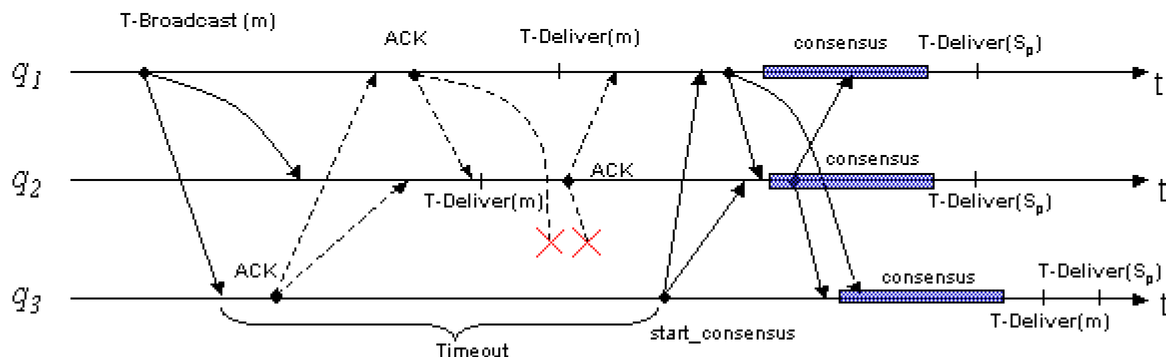

Figure 20 – Exécution d'un T-Broadcast avec consensus

3.7 La primitive Atomic Broadcast

3.7.1 Description

La primitive de communication de groupe Atomic Broadcast, appelée plus tard `A-Broadcast`, est la dernière brique nécessaire au fonctionnement du service répliqué et constitue le but de la démarche menée jusqu'à maintenant. Utilisée par les clients pour s'adresser au service répliqué, elle assure la diffusion atomique des messages. C'est-à-dire qu'elle garantit la réception des requêtes des clients par tous les réplicats corrects et dans le même ordre et par conséquent, la cohérence des états de tous les processus du groupe.
Plus formellement, la primitive A-Broadcast exécutée par un client s'adressant au groupe réplique P doit satisfaire les trois conditions suivantes [Sch00]:

Ordre. Soit ABCAST(m_1,P_x), ABCAST(m_2,P_x) et les réplicats x_j et x_k dans P_x. Si x_j et x_k exécutent A-Deliver(m_1) et A-Deliver(m_2), alors ils le font dans le même ordre.

Atomicité. Soit ABCAST(m,P). Si x, appartenant à P, exécute A-Deliver(m), alors tout réplicat correct de P finira aussi par exécuter A-Deliver(m).

Terminaison. Supposons qu'un processus p_i exécute ABCAST(m,P_x). Si p_i est correct, alors tout réplicat correct appartenant à P finira par exécuter A-Deliver(m).

Cette primitive s'appuie sur les deux précédentes (`R-Broadcast et T-Broadcast`). En fait, `R-Broadcast` assure la livraison des messages à tous les processus corrects et `T-Broadcast` les ordonne. L'ordre FIFO assuré par cette dernière peut facilement être transformé en ordre total si un unique processus dans le groupe effectue les `T-Broadcast`. C'est de cette manière que fonctionne A-Broadcast. Pour cela, on définit la notion d'époque (« `epoch` », an anglais), similaire à celle de ronde dans l'algorithme du consensus. On commence à l'époque 0 en élisant, de manière déterministe un processus particulier : le séquenceur. Lorsqu'un client exécute un `A-Broadcast(m)`, le processus qui reçoit la requête diffuse `m` en utilisant `R-Broadcast(m)`. Cela assure la livraison de `m` à tous les réplicats corrects. Après avoir exécuté `R-Deliver(m)`, le séquenceur, dont le rôle est d'ordonner les messages, exécute `T-Broadcast(m)`. Finalement, dès qu'un processus exécute `T-deliver(m)`, il peut délivrer le message et effectuer le traitement correspondant à la requête. Si le message S_p est délivré à la place de `m`, cela signifie que l'émetteur, c'est-à-dire le séquenceur, est suspecté. Le consensus qui vient d'avoir lieu assure que tous les processus du groupe ont exactement délivré les mêmes messages et tous décident de changer d'époque et donc d'élire un nouveau séquenceur.

3.7.2 L'architecture

La Figure 21 donne une représentation complète de la couche de protocole dans laquelle tous les modules sont présents. Le module `A-Broadcast Manager` (en haut à droite sur la figure) est responsable de la mise en œuvre de la primitive `Atomic Broadcast`.

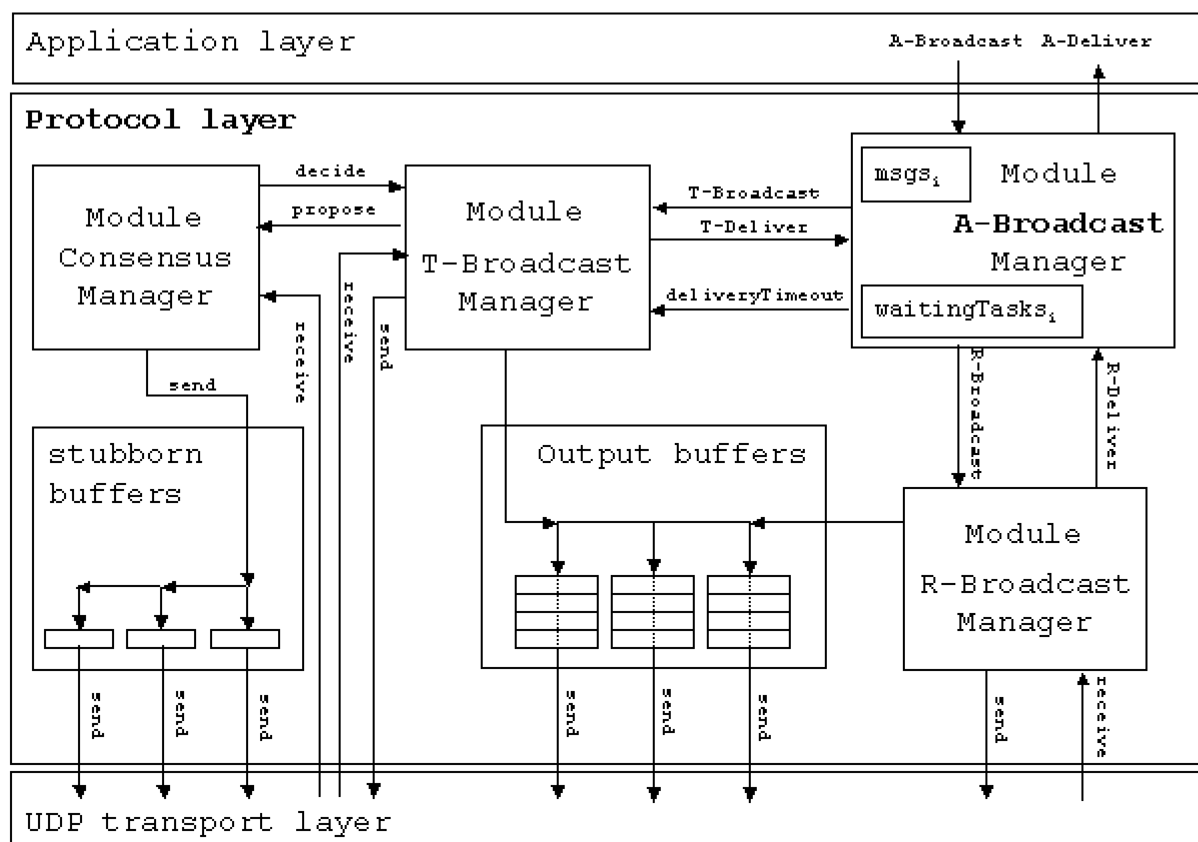

Figure 21 – Atomic Broadcast Architecture

Le module `A-Broadcast Manager` possède bien entendu une fonction A-Broadcast qui permet à la couche applicative (ou éventuellement à un autre module) d'envoyer un message avec `Atomic Broadcast`. Mais aussi une fonction `R-Deliver` qui permet au module R-Broadcast Manager de lui signifier la livraison d'un message envoyé avec `R-Broadcast`. Et finalement il existe une fonction `T-Deliver` similaire qui lui notifie la livraison d'un message envoyé avec `T-Broadcast`.

3.7.3 L'algorithme

```
1   module A-Broadcast manager for a process pi
2   {
3     msgsi[] ;                    //set of messages initialized to ∅.
4     epochi ;                     //current epoch initialized to 0 ;
5     Δti ;                        //amount a time that pi waits for a T-Deliver.
6     sequenceri ;                 //actual sequencer initialized to p1.
7     waitingTasksi ;              //set of waiting tasks.
8
9     function A-Broadcast(msg) {
10      R-Broadcast(epoch,msg) ;
11    }
12
13    function R-Deliver(epoch,msg) {
14      if(epochi = epoch) {
15        if(msgsi ∩ {(msg,_)} ≠ ∅) {
16          msgsi = msgsi / {(msg,_)} ;
17        }
18        else {
19          msgsi = msgsi ∪ {(msg,RDELIVER)} ;
20          waitingTasksi = waitingTasksi ∪ {(id(msg),Δti)} ;
21        }
22        if(pi = sequenceri) {
23          T-Broadcast(msg) ;
24        }
25      }
26    }
27
28    function T-Deliver(msg) {
29      if(msgsi ∩ {(msg,_)} ≠ ∅) {
30        waitingTasksi = waitingTasksi / {(id(msg),Δti)} ;
31        msgsi = msgsi / {(msg,_)} ;
32      }
33      else {
34        msgsi = msgsi ∪ {(msg,TDELIVER)} ;
35      }
36      A-Deliver(msg) ;
37    }
38
39    function T-Deliver(Sp) {
40      epochi = epochi + 1 ;
41      sequencer = p(epochi mod n)+1 ;
42      if(pi = sequenceri) {
43        ∀(m,type) ∈ msgsi | type = RDELIVER, T-Broadcast(m) ;
44      }
45    }
46
47    function waitingTimeout(id(msg)) {
48      deliveryTimeout() ;
49    }
50  }
```

Algorithme 4 – Atomic Broadcast

4 L'implémentation

Parallèlement à la phase théorique présentée au chapitre 3, une réalisation concrète du service répliqué a été développée en Java. Le but de ce chapitre est de présenter de manière générale son organisation avec le souci de mettre en évidence les points communs et les différences avec le modèle théorique.
Deux applications différentes ont été implémentées. Une partie client et une autre serveur. Elles ont été réalisées de manière totalement indépendante de manière à pouvoir donner le code que de l'une des deux.

4.1 Environnement de développement

Toute la partie de développement a été effectuée dans un environnement de type SUN sur les machines appartenant au réseau `lsesuns` du Laboratoire de Systèmes d'Exploitations. La liste ci-dessous donne une description résumée de chacune d'elles.

```
SunOS lsesun1 5.6 Generic_105181 15 sun4u sparc SUNW,Ultra-60
SunOS lsesun2 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun4 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-60
SunOS lsesun5 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun6 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-60
SunOS lsesun7 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun8 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun9 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun10 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-30
SunOS lsesun11 5.6 Generic_105181-16 sun4u sparc SUNW,Ultra-60
SunOS lsesun12 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-30
SunOS lsesun13 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-30
SunOS lsesun14 5.6 Generic_105181-15 sun4u sparc SUNW,Ultra-1
SunOS lsesun15 5.6 Generic_105181-16 sun4u sparc SUNW,Ultra-30
```

L'implémentation du service répliqué est programmée avec le langage Java et plus précisément la version J2SE 1.3.0 dans le but d'assurer une portabilité maximale du code. Il s'agit de la dernière version de Java 2. Une description complète de cet environnement se trouve à l'adresse suivante : http://java.sun.com/j2se/1.3/docs/index.html.
Finalement, le code a été écrit avec l'éditeur XEmacs version 20.4.
Les paragraphes 4.4.1 et 4.4.2 contiennent une brève description de chacune des classes et interfaces en faisant, si possible, correspondre sa fonction avec un élément de l'architecture (cf.Figure 10).

4.2 Préliminaires

Lors du lancement d'un réplicat, un certain nombre de paramètres sont à fixer par l'utilisateur. Afin d'éviter de devoir les spécifier sur la ligne de commande, ils sont lus dans un fichier par le code d'initialisation du réplicat. Il en va de même pour démarrer un client.
Les paramètres pour un réplicat sont :

`server.view` Liste (adresses) statique des membres du groupe

`server.view_group`	Adresse IPMulticast du groupe répliqué
`server.rtservers`	Liste des RTServer auxquels le réplicat peut se connecter
`server.tbcast_timeout`	Intervalle de temps qu'un processus attend un T-Deliver.
`server.consensus_timeout`	Intervalle de temps qu'un processus attend la proposition du coordinateur (ie. délai de suspicion)
`server.udp_port`	Port UDP d'émission et de réception

L'unique paramètre nécessaire au démarrage d'un client est :

`server.rtservers`	Liste des RTServer auxquels le client peut se connecter

4.3 Le paquetage client

Il n'est pas nécessaire de décrire les classes qui composent la partie client du service répliqué. A l'aide d'une interface graphique, un client peut demander à la copie primaire qui reçoit son message d'exécuter l'une des trois primitives à disposition (R-Broadcast, T-Broadscast et A-Broadcast). L'utilisateur peut choisir d'envoyer un seul message textuel ou un ensemble de messages dont il peut fixer la taille (cf. Figure 22).

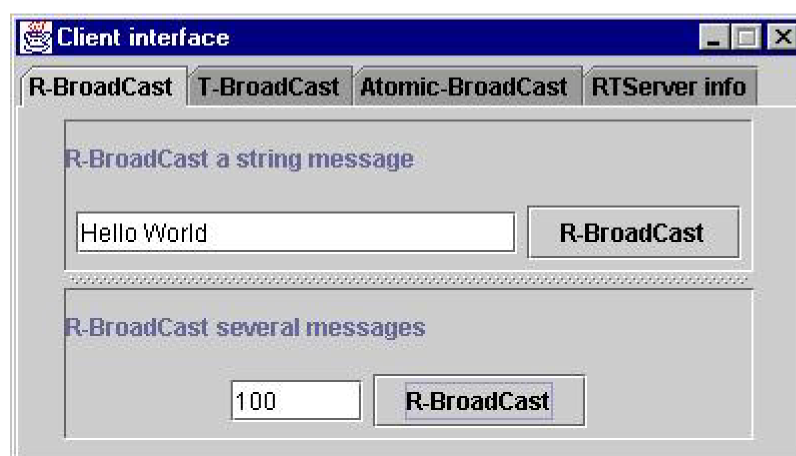

Figure 22 – Interface graphique du client

Finalement, l'utilisateur a aussi la possibilité d'obtenir quelques informations basiques à propos de son environnement `SmartSockets` (Figure 23).

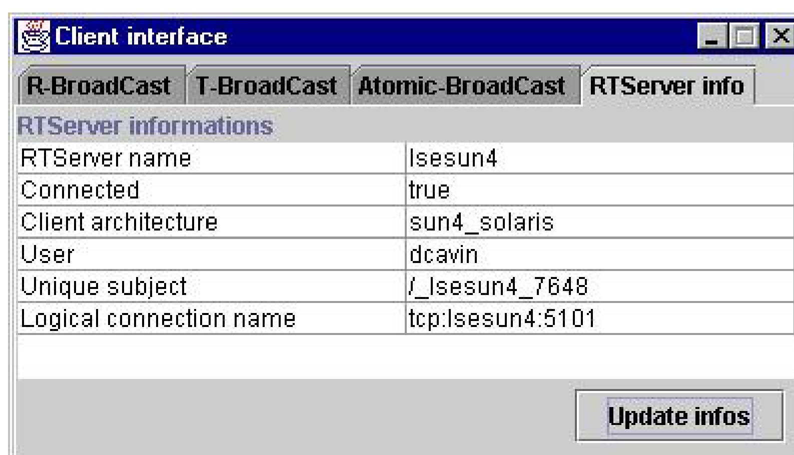

Figure 23 - Informations sur l'environnement SmartSockets

4.4 Le paquetage server

4.4.1 Les interfaces

Les interfaces générales

Interface UDPMessagesListener
Cette interface est implémentée par toutes les classes qui souhaitent pouvoir s'inscrire auprès de la couche UDP pour recevoir un certain type de messages. Elle fait le lien entre la couche UDP et la couche de protocole.

Interface MessagesListener
Cette interface est implémentée par toutes les classes qui souhaitent pouvoir s'inscrire auprès du `DeliverDispatcher` comme destinataire final de la livraison d'un message. Elle fait le lien entre la couche de protocole et la couche applicative.

Interface DecideListener
Cette interface est implémentée par toutes les classes qui souhaitent pouvoir s'inscrire auprès du module `Consensus Manager` pour être informé de la décision d'un consensus.

Les controller

Toutes les interfaces dont le nom se termine par « controller » sont destinées à offrir aux classes qui les implémentent une sorte de contrat d'utilisation. C'est le cas pour tous les modules de la couche de protocole. Chacun implémente sa propre interface « controller » et n'offre qu'un ensemble choisi de méthodes qui pourront être appelées. Toutes ces interfaces sont enregistrées auprès d'une classe particulière (`ControllerRegistry`) à qui un module peut demander une référence sur un « controller » spécifique. Cela permet d'éviter un grand nombre de problèmes liés à la présence de références croisées et n'impose pas de prévoir à l'avance quels seront les modules nécessaires.

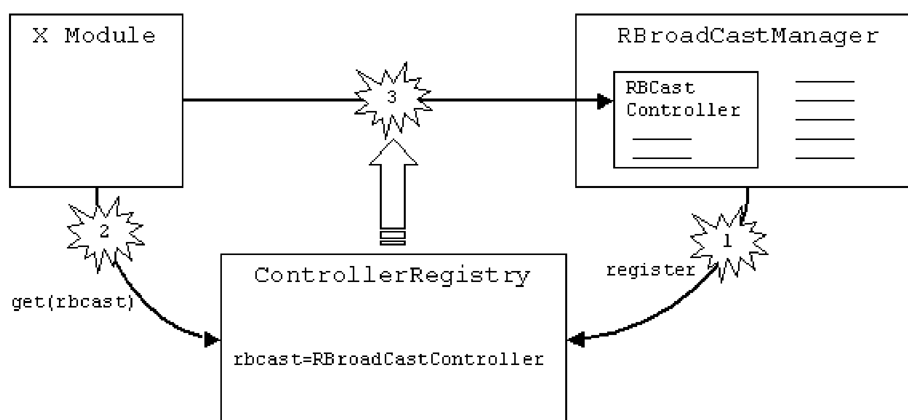

Figure 24 - Gestion des "controller"

La Figure 24 illustre le mécanisme d'inscription des `controller`. A l'étape 1, le module `RBroadCastManager` inscrit son `controller`, c'est-à-dire un ensemble choisi de méthodes qu'il implémente et qu'il souhaite mettre à disposition d'autres objets, auprès du `ControllerRegistry`. A l'étape 2, un module X (i.e. quelconque) souhaite utiliser le module `RBroadCastManager` sur lequel il n'a aucune référence. Pour cela, il s'adresse au `ControllerRegistry` qui lui retourne, à l'étape 3, une référence sur le `RBroadCastManager` via son interface `RBCastController`.

Interface RBCastController
Cette interface n'est implémentée que par la classe `RBroadCastManager` et offre un ensemble fixé de méthodes qui permettent de contrôler le module `R-Broadcast Manager`.

Interface ConsensusController
Cette interface n'est implémentée que par la classe `ConsensusManager` et offre un ensemble fixé de méthodes qui permettent de contrôler le module `Consensus Manager`.

Interface TBCastController
Cette interface n'est implémentée que par la classe `TBroadCastManager` et offre un ensemble fixé de méthodes qui permettent de contrôler le module `T-Broadcast Manager`.

Interface ABCastController
Cette interface n'est implémentée que par la classe `ABroadCastManager` et offre un ensemble fixé de méthodes qui permettent de contrôler le module `A-Broadcast Manager`.

4.4.2 Les classes

Les classes générales

Classe Server
Il s'agit de la classe principale du projet et la seule exécutable. Un objet de la classe `Server` représente un réplicat (ou serveur). En l'instanciant, un nouveau réplicat est créé en même

temps que tous les éléments nécessaires au fonctionnement du réplicat (modules, buffers, couche UDP, etc.) sont créés, initialisés et connectés.

Classe UDPLayer

Cette classe représente la couche de transport UDP. Elle offre plusieurs méthodes qui permettent d'envoyer des objets `Message` soit en mode destinataire simple, soit en mode `multicast` (envoi à tous les membres du groupe, cf. 3.2.5). Il est aussi possible d'enregistrer un objet du type `UDPMessagesListener` qui sera prévenu lors de la réception d'un type de message particulier. Globalement, il s'agit d'un `Thread` qui attend indéfiniment l'arrivée d'un nouveau message qui, après en avoir extrait son type, le délivre aux `UDPMessagesListener` intéressés. C'est dans cette classe que les objets `Message` sont transformés en tableaux d'octets et vice-versa.

Classe FailureUDPLayer

Il s'agit d'une sous-classe directe de UDPLayer. Elle offre par conséquent les mêmes fonctionnalités mais il est possible, en plus, de redéfinir à souhait les méthodes d'envoi et de réception de messages dans le but de simuler la présence d'erreurs de transmission/réception. Elle dispose aussi d'une interface graphique simple qui permet de visualiser un certain nombre d'informations importantes, comme une description des `Threads` actifs, le nombre de messages reçus et émis par seconde ainsi que l'occupation instantanée des `buffers` de retransmission (cf. Figure 25).

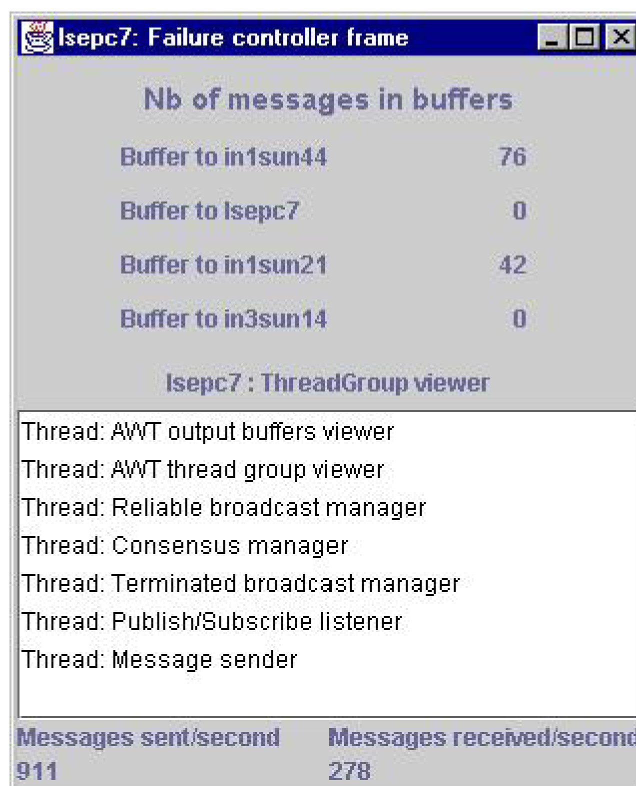

Figure 25 – Interface de déverminage

Classe View

Cette classe représente la vue qu'un processus a du groupe répliqué. Elle contient une liste d'adresses IP. Pour l'instant, il n'est pas possible d'ajouter ou de retirer un processus de la vue initiale.

Classe ControllerRegistry

Cette classe permet aux différents modules d'inscrire leur interface « `controller` » (cf. §4.4.1).

Les messages

Cette section présente les deux classes qui sont en rapport avec les messages. Elle est à mettre en relation avec les paragraphes 2.3 et 3.3.

Classe MessagesTypes

Cette classe ne représente pas un objet particulier, mais contient l'ensemble des identificateurs (nombres entiers) de messages qui sont définis dans le projet (cf §3.3.1). De plus elle permet d'obtenir une chaîne de caractères décrivant chacun des types de messages.

Classe Message

Cette classe représente les messages qui sont échangés entre les clients et le groupe répliqué ainsi qu'à l'intérieur du groupe. Elle offre un certain nombre de méthodes qui permettent d'ajouter un nouveau champ d'un type particulier au message (p. ex. `addNamedInt`) ; symétriquement, des méthodes qui permettent de retourner la valeur d'un champ (p. ex. `getNamedObject`). Elle offre aussi la possibilité à la couche UDP de transformer les messages en tableaux d'octets et vice-versa.

Les buffers

Cette section regroupe toutes les classes qui sont liées au fonctionnement des différents types de `buffers`.

Classe OutputBuffer

Un objet de cette classe représente un `buffer` de retransmission. Il s'agit d'un tableau dynamique d'objets `Message` auquel est attaché une adresse de destination. La classe Server crée et initialise exactement `n-1` objets `OutputBuffer` si `n` est le nombre de processus dans la vue.

Classe StubBornBuffer

Il s'agit d'une sous-classe directe de OutputBuffer qui modélise les `stubborn buffers` nécessaires dans l'algorithme du consensus (cf §3.5.1). Cette fois-ci le tableau de messages n'est plus dynamique, mais borné par un entier fourni en paramètre.

Classe MessageSender

Il s'agit d'un `Thread` responsable de l'envoi périodique du contenu des `buffers` de retransmission. Son constructeur accepte en paramètre un tableau d'objets `OutputBuffer` ou, par héritage, de `StubBornBuffer`.

Classe InputBuffer

Cette classe offre un moyen de stocker avant leur traitement des messages qui sont reçus de la couche UDP. A chaque fois qu'un module de la couche de protocole reçoit un message transmis par la couche UDP, il le place dans un `buffer` instance de cette classe. Cela permet de réduire le temps nécessaire à la réception d'un message. Cette classe possède une méthode qui retourne le prochain message et qui bloque le `Thread` qui le demande si le `buffer` est vide.

Les managers

Le dernier groupe de classe est celui des « managers ». Il s'agit des quatre modules qui mettent en œuvre les primitives de communication et les algorithmes. Cette section est à mettre en rapport avec les chapitres 3.4, 3.5, 3.6 et 3.7.

Classe PrimitiveManager

Il s'agit de la super classe de toutes les primitives implémentées, c'est-à-dire de tous les modules. Cette classe est un `Thread` dont la méthode principale est abstraite. Cette dernière devra être implémentée dans chacune des sous-classes et contiendra l'algorithme mis en œuvre par la primitive. Elle offre aussi un certain nombre de méthodes qui pourront être hérités et utilisées par toutes les sous-classes (p.ex. envoi de messages avec retransmission, réception, etc.).
Elle possède aussi un `buffer` de réception, de type `InputBuffer`, qui lui permet de stocker provisoirement les messages reçus de la couche UDP. Tant que le `buffer` n'est pas vide, le `Thread` traite les messages et sinon il s'interrompt.

Classe RBroadCastManager

Cette classe, sous-classe de `PrimitiveManager`, représente le module `R-Broadcast Manager` (cf.Figure 12). Elle implémente l'interface `UDPMessageListener` qui lui permet de recevoir les messages de type `RBCAST` et `RBCAST_ACK`, ainsi que `RBCastController` qui permet aux autres modules d'y accéder.

Classe ConsensusManager

Cette classe, sous-classe de `PrimitiveManager`, représente le module `Consensus Manager` (cf. Figure 15). Elle implémente l'interface `UDPMessageListener` qui lui permet de recevoir les messages de type `CONSENSUS`, `CONSENSUS_ACK` et `CONSENSUS_DECIDE`, ainsi que `ConsensusController` qui permet aux autres modules d'y accéder.

Classe Estimate

Cette classe n'est utile que pour le module `Consensus Manager`. Elle représente le champ $estimate_i$ utilisé dans l'algorithme du consensus (cf §3.5.3).

Classe TBroadCastManager

Cette classe, sous-classe de `PrimitiveManager`, représente le module `T-Broadcast Manager` (cf.Figure 18). Elle implémente l'interface `UDPMessageListener` qui lui permet de recevoir les messages de type `TBCAST` et `TBCAST_ACK`, ainsi que `TBCastController` qui permet aux autres modules d'y accéder.

Classe ABroadCastManager

Cette classe, sous-classe de `PrimitiveManager`, représente le module `A-Broadcast Manager` (cf.Figure 21). Elle implémente l'interface `ABCastController` qui permet aux autres modules d'y accéder.

Classe DebugMessageWrapper

Cette classe, sous classe de `PrimitiveManager`, est un manager un peu particulier. Il s'agit d'une sorte de module qui intercepte tous les messages en provenance des clients (type `CLIENT_MSG`). Ces derniers contiennent, comme champ, le type de la requête qu'ils souhaitent exécuter (i.e. `R-Broadcast, Consensus, T-Broadcast, A-Broadcast`). En fonction de son type, le `DebugMessageWrapper`, envoie le message avec la bonne primitive. Cela permet de facilement tester le fonctionnent de chacune d'elles. A l'avenir, cette classe devrait disparaître car les clients ne pourront plus qu'envoyer une requête au service répliqué sans pouvoir choisir le type de primitive qui va être utilisé.

5 Mesures de performances

5.1 Situation

Après avoir complètement implémenté le service répliqué et un client qui peut y accéder, des tests de performances ont été réalisés. Ils avaient pour but principal d'évaluer le temps nécessaire à l'exécution d'une requête `Atomic Broadcast`.

Parallèlement, il s'agissait aussi d'une phase de tests de fonctionnement, qui permettrait de valider l'approche et de s'assurer de la conformité des primitives. Un grand nombre de corrections ont dû être apportées à l'implémentation et aux algorithmes, suffisantes pour justifier une nouvelle version du service répliqué. Par exemple, la taille maximale des paquets `UDP` qui peuvent être envoyés était bien souvent dépassée. Un nouvel objet `Message`, plus performant, a donc été créé. Ces modifications ont certainement changé les performances de communication. Donc, les résultats présentés dans ce chapitre sont ceux obtenus avec une version plus ancienne. Il est néanmoins possible de les interpréter qualitativement.

5.2 Performances d'Atomic Broadcast

Les mesures ont été réalisées en utilisant des machines appartenant au sous-réseau du LSE (`lsesuns`). Un client envoie des requêtes avec diverses fréquences que la copie primaire va transmettre aux autres membres du groupe en exécutant un `Atomic Broadcast` (Figure 26).

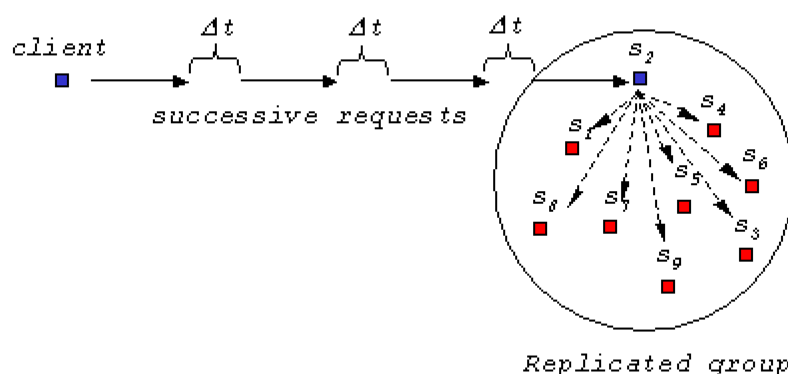

Figure 26 – Architecture du test

Le client fixe un intervalle Δt entre deux envois successifs de messages. Lorsque la copie primaire du groupe reçoit une requête `m` du client, elle l'estampille avec l'heure actuelle. Ensuite elle exécute `A-Broadcast(m)`. Dès qu'elle exécute `T-Deliver(m)`, il regarde si c'est lui qui l'a estampillé et si tel est le cas, alors il stocke dans une collection la différence entre l'heure actuelle et l'estampille. Comme les messages du client sont envoyés avec un `Load balancing` de type `round robin` (cf §2.5), chacun des `n` membres du groupe vont estampiller une proportion égale à 1/n des messages envoyés. Il ne reste plus qu'à les remettre dans l'ordre. Finalement, avec la version de Java utilisée, il existe un mécanisme qui

permet à l'application qui se termine, de faire du « rangement » (p.ex. sauvegarde de l'état, envoi d'un message, etc.). Cela est utile car il est possible de stocker les mesures de performances dans la mémoire et de les écrire dans un fichier (ou à l'écran) que tout à la fin du test sans influencer leur qualité.
Voici les résultats obtenus avec un envoi de 100 messages avec un intervalle de 100ms.

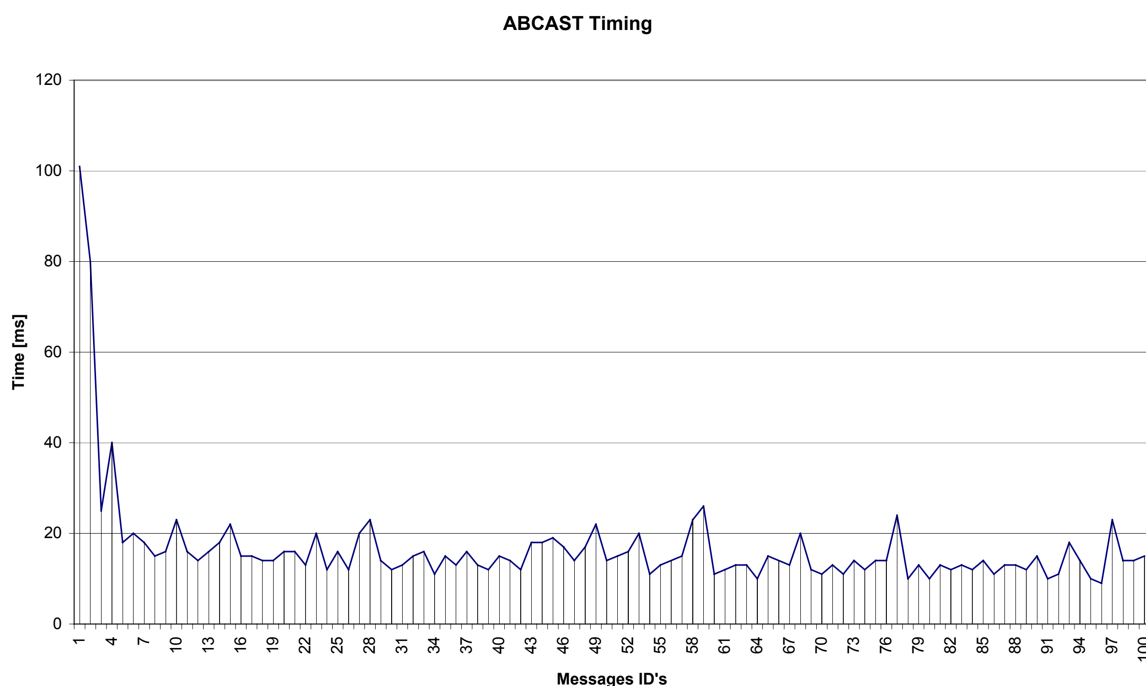

Figure 27 – Envoi de 100 messages avec intervalle de 100ms

Sur l'axe des abscisses se trouvent les identificateurs de messages ordonnés de 0 à 100 et sur l'axe des ordonnées les temps en millisecondes mesurés pour chacun des messages.
La première exécution de `A-Broadcast` prend beaucoup de temps (~ 100 ms) car elle nécessite une phase d'initialisation. Pour le reste, les valeurs se trouvent, en moyenne, à 16 ms.
Lorsqu'on réduit l'intervalle d'émission des messages du client. La courbe est modifiée et ressemble à celle de la Figure 28. On constate que le temps de livraison des messages croit rapidement pour atteindre 5000 ms. Cela peut s'expliquer par le fait que tous les messages, bien que répartis entre tous les membres du groupe, doivent être ordonnés par un unique processus, le séquenceur. Ce dernier joue le rôle de goulet d'étranglement. En effet, tous les processus du groupe estampillent en parallèle les messages des clients qui vont ensuite être ordonnés les uns après les autres par le séquenceur.

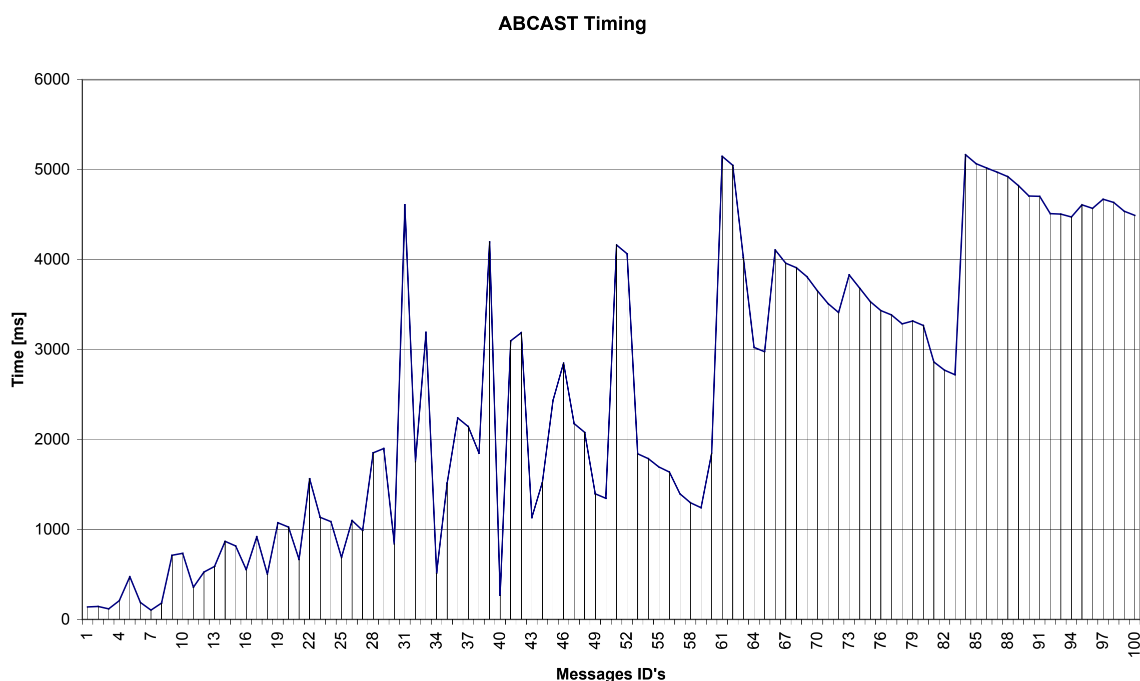

Figure 28 – Envoi de 100 messages avec intervalle de 20ms

Si on considère maintenant le nombre d'exécution de `A-Deliver` par seconde, on peut déduire des observations qui précèdent que ce nombre doit d'abord croître lorsque la fréquence d'envoi diminue, puis atteindre un maximum et finalement diminuer au moment où le séquenceur est surchargé. C'est effectivement ce qui se produit sur la Figure 29. L'axe des abscisses représente l'intervalle de temps d'émission des messages du client qui augmente progressivement. Sur l'axe des ordonnées se trouve une moyenne sur tous les processus du nombre d'exécution de `A-Delvier` par seconde. Les mesures apparaissent sous la forme de points bleus. Une courbe de tendance polynomiale d'ordre 3 donne l'allure générale. On remarque qu'elle atteint son maximum lorsque l'intervalle d'émission est égal à environs 32 ms.
Cela confirme donc que le service répliqué peut être saturé si la fréquence des requêtes devient trop élevé.

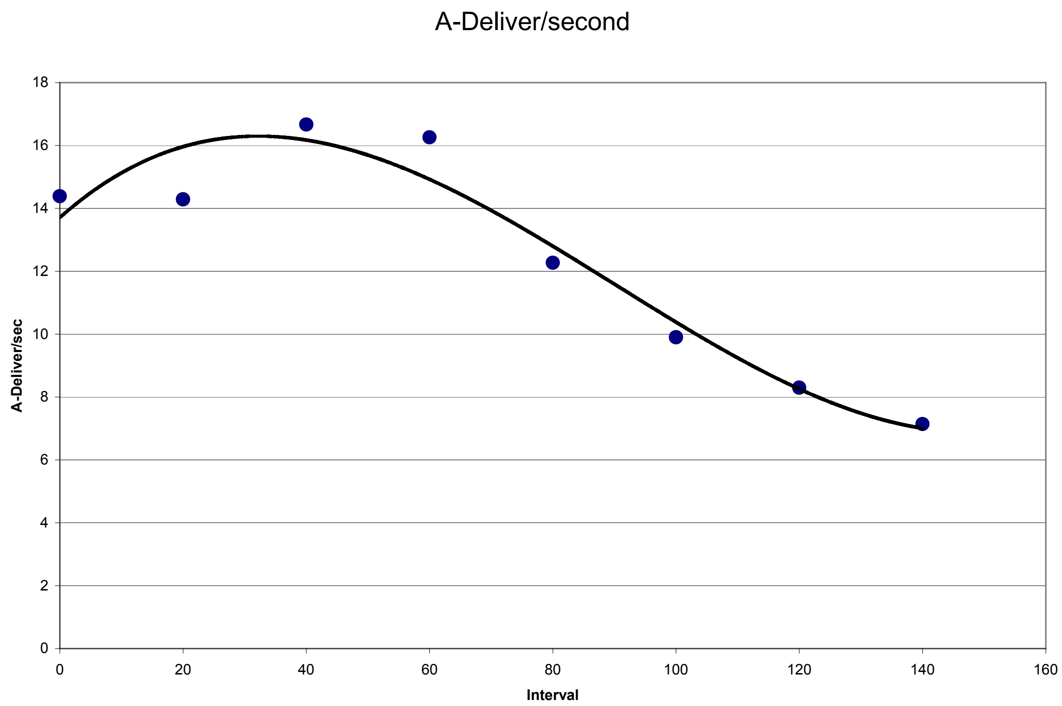

Figure 29 - A-Deliver / seconde

6 Développements futurs

Deux extensions importantes ont été prévues. Il s'agit d'améliorations et de généralisations qui permettent une plus grande souplesse du service répliqué. La première a pour but de rendre la vue dynamique, c'est-à-dire de permettre d'ajouter ou de retirer à tout moment un réplicat du groupe. Actuellement, la vue est statique et, par exemple, la défaillance d'un processus n'a aucun effet sur elle.
La deuxième extension envisagée, doit permettre à un groupe répliqué d'être lui-même client d'un deuxième service répliqué. Cela permettra de généraliser ses conditions d'utilisation, sans à priori sur ce que vont faire les processus du groupe.
Même si le temps n'a pas permis de réaliser une implémentation de ces deux fonctionnalités, un début de réflexion a été menée.

6.1 Vue dynamique

Actuellement, la vue que chaque processus a du groupe, est statique, c'est-à-dire qu'elle est fixée lorsque le réplicat est créé (cf chapitre 4.2). Si un réplicat « crashe » il ne peut donc plus ni envoyer, ni recevoir de messages et les autres membres du groupe n'ont aucun moyen de s'en rendre compte. Ils vont continuer, à retransmettre le contenu de leur `buffer` à destination du processus défaillant.
Il serait bien plus intéressant de pouvoir exclure ce processus de la vue courante et même accepter l'arrivée d'un nouveau réplicat. Ceci d'autant plus que l'environnement `SmartSocket` n'est absolument pas concerné par ce changement. En effet, l'arrivée et le départ dynamique de `RTClients` est tout à fait possible.
Parallèlement au mécanisme de changement de vue, il doit exister un moyen de suspecter un processus qui pourra être retire de la vue. Pour cela il est possible de rajouter un nouveau module dont le rôle est de s'assurer que les autres membres du groupe ne sont pas défaillants. Il suffit qu'il envoie régulièrement des requêtes, appelées « `keep alive` » que les destinataires doivent simplement acquitter. Si un acquittement en provenance d'un processus p_i n'arrive pas avant un certain délai, alors p_i sera suspecté. Il est peut-être aussi envisageable de se passer d'un tel module est de déduire la défaillance d'un processus de l'architecture actuelle. En effet, il existe déjà un mécanisme qui permet de suspecter le coordinateur dans l'algorithme du consensus. De même, la livraison du message S_p durant l'exécution d'un `Atomic Broadcast` signifie que le séquenceur est suspecté. Ce serait plus judicieux de tirer parti au maximum de ces éléments qui existent déjà.
Il faut aussi définir deux fonctions `join` et `leave` qui permettent de rejoindre et de quitter la vue volontairement, indépendamment du mécanisme de suspicion.
Finalement, il faut mettre en oeuvre un protocole de changement de vue qui doit garantir que tous les processus qui feront partie de la nouvelle vue sont d'accord sur sa composition. Ce protocole doit satisfaire les trois conditions suivantes [Sch00]:

On suppose un groupe P de processus

Terminaison. La défaillance d'un processus ou l'exécution de `join` ou `leave` finira par conduire à un changement de vue.

Agrément. Si un processus p dans la vue $v_i(P)$ installe la nouvelle vue $v_{i+1}(P)$ et un processus q dans la vue $v_i(P)$ installe la nouvelle vue $v'_{i+1}(P)$, alors $v_{i+1}(P) = v'_{i+1}(P)$.

Validité. Supposons deux vues consécutives $v_i(P)$ et $v_{i+1}(P)$. Si $p \in v_i(P) \setminus v_{i+1}(P)$, alors p est défaillant (ou suspecté) ou a exécuté `leave`. Si $p \in v_{i+1}(P) \setminus v_i(P)$, alors p a exécuté `join`.

Un algorithme comme `VSCAST` (`View Synchronous Broadcast`) met en œuvre un mécanisme de changement de vue qui satisfait ces trois propriétés [Sch00]. Il nécessite l'utilisation du consensus et ressemble au comportement de la primitive `T-Broadcast`.

6.2 Clients répliqués

La situation actuelle n'empêche pas le service répliqué, en réponse à la requête d'un client, de s'adresser à un second groupe répliqué. En effet, pendant la phase de traitement de la requête, la copie primaire est libre d'envoyer des messages à l'extérieur du groupe. Mais cette situation n'est pas satisfaisante car elle peut poser des problèmes de cohérences entre les réplicats si la copie primaire est défaillante.
La Figure 30 illustre les différentes étapes durant lesquelles le `Group 1` va s'adresser au `Groupe 2`.

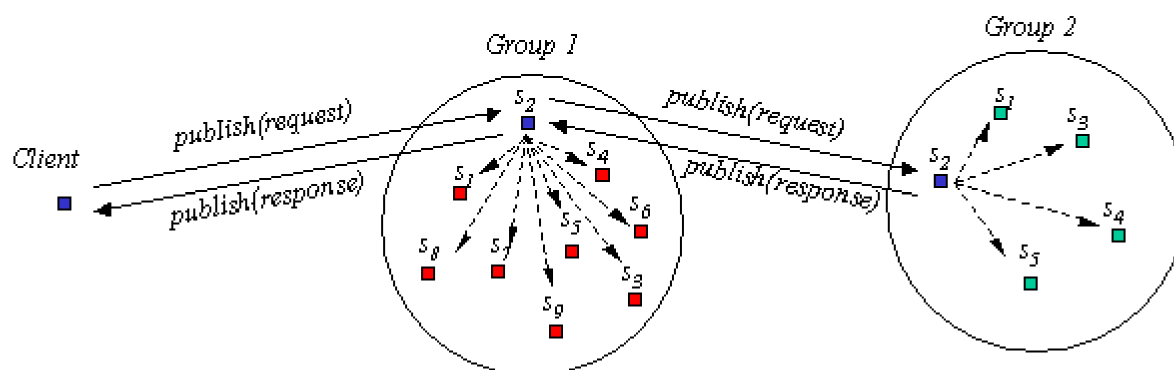

Figure 30 – Clients répliqués

Un client formule sa requête en la publiant sur un sujet auquel tous les membres du `Group 1` ont souscrit. Seule la copie primaire reçoit son message. Durant sa phase de traitement, elle décide de contacter un second service répliqué (`Group 2`) en publiant, elle aussi, sa requête sur un sujet auquel tous les membres du `Group 2` ont souscrit. Après avoir traité la requête, la copie primaire du `Group 2` va envoyer un message de mise à jour au membre de son groupe et envoyer sa réponse au `Group 1`. La copie primaire du `Group 1` fait de même et publie sa réponse au client.
Cette exécution peut poser un problème. En effet, si la copie primaire du Group 1 « crashe » avant d'avoir reçu la réponse du `Group 2`, cette dernière va être reçue par un autre réplicat

du Group 1 qui ne saura pas comment la traiter. Il est donc nécessaire de définir un bref protocole d'échange de messages qui permet d'éviter la situation décrite ci-dessus (Figure 31).

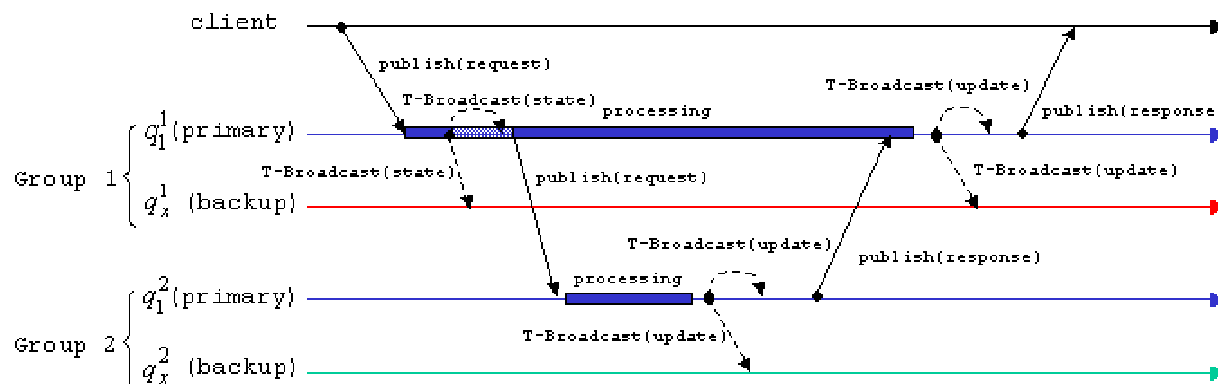

Figure 31 – Exécution d'une requête avec un client répliqué

La copie primaire q_1^1 du `Group 1`, après la réception de la requête du client, commence son traitement et juste avant de contacter le second service répliqué, elle envoie son état avec message `T-Broadcast`. De cette manière, elle s'assure que toutes les copies du `Group 1` ont pu se mettre à jour en tenant compte du début du traitement de q_1^1. Ce n'est qu'après avoir exécuté `R-Deliver(state)` que q_1 peut contacter le `Group 2`. La copie primaire q_1^2 du Group 2 traite la requête et envoie un message de mise à jour au reste du groupe avec T-Broadcast. La copie q_1^1 fait exactement de même et après avoir exécuté `T-Deliver(update)`, elle peut transmettre sa réponse au client.
Avec cette approche, si la copie primaire « `crashe` » juste après avoir envoyé sa requête au `Group 2`, une autre copie est capable de traiter la réponse car elle est dans le même état que l'était la copie primaire.

7 Conclusions

Le service répliqué qui a été réalisé permet à un groupe de `RTClients SmartSockets` de répartir une prestation quelconque de manière transparente et fiable. La primitive de communication `Atomic Broadcast` offre un moyen sûr de propager une information au sein du groupe et garantit sa cohérence. Son implémentation est originale car elle réussit à mettre en œuvre l'ordre total à l'aide d'une primitive (`Terminated Broadcast`) qui n'assure que l'ordre FIFO. L'algorithme du consensus lui aussi, offre une solution novatrice au problème de l'agrément en bornant le nombre de messages à stocker. Ce projet a aussi permis d'implémenter complètement et de tester cet algorithme ainsi que toutes les autres primitives et particulièrement la plus récente, `Terminated Broadcast`. Les nombreux tests, à défaut d'avoir une réelle importance quantitative, ont surtout servi à corriger et finalement valider les algorithmes.

La réalisation en Java permet au service répliqué de parfaitement s'intégrer aux `SmartSockets` et de conserver une portabilité maximale. Le code a été écrit de manière extrêmement modulaire dans le but de le faire correspondre le plus précisément possible avec l'architecture théorique. Il est donc possible, d'ajouter facilement de nouvelles primitives de communication. De plus, la présence d'une couche permettant de simuler des erreurs de transmission ou des défaillances, a permis de tester le service répliqué dans des conditions proches de la réalité.

Il reste néanmoins un certain nombre de choses à faire pour s'assurer du comportement correct du service répliqué dans toutes les situations. Pour achever de formaliser et de valider l'architecture développée, il est nécessaire de réaliser des tests plus complets. En effet, les exécutions de programmes répartis sont extrêmement complexes à interpréter car cela impose d'ordonner un grand nombre d'événements concurrents.

8 Abbréviations

API	Application Programming Interface
FIFO	First In First Out
FLP	Fischer Lynch Patterson (impossibilité FLP)
GMD	Guaranteed Message Delivery
IP	Internet Protocol
J 2SE	Java 2 Standard Edition
LAN	Local Area Network
RFC	Request For Comment
TCP	Transport Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

9 Bibliographie

[Mul93] S. Mullender, *Distributed Systems*, second edition, éd. Addinson-Wesley, 1993.

[Sch00] A. Schiper, *Distributed Algorithms*, éd P. Urbán and A. Schiper, 2000.

[Oli00] R. Oliveira, *Solving consensus : from fair-lossy channels to crash-recovery of processes*, éd. EPFL, 2000.