Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Communications de groupe modulaires: comparaison d'Appia et Cactus

Cuvellier, Xavier; Grégoire, Christophe

Award date:
2003

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 03. Jul. 2025

Facultés Universitaires Notre-Dame de la Paix, Namur

Institut d'Informatique

Année Académique 2002 - 2003

Communications de groupe modulaires

-

Comparaison d'Appia et Cactus

Xavier Cuvellier & Christophe Grégoire

Mémoire présenté en vue de l'obtention du grade de Maître et Licencié en Informatique.

LBS 10027140

Résumé

Les communications de groupe fournissent un ensemble de primitives permettant d'effectuer des broadcasts ou des multicasts avec certaines garanties. La fiabilité est la première a être demandée par des algorithmes distribués.

Travaillant dans un monde imparfait, les pannes au sein du système sont à prendre en compte. Nous parlons de pannes réseaux entraînant des retards importants dans la réception des messages, mais également de pannes au niveau des machines sur lesquelles s'exécute un processus prenant part à un algorithme distribué.

Les communications de groupe tolérantes aux pannes est un sujet très complexe. Il est donc préférable de décomposer ces primitives en sous-problèmes, ce qui facilitera leur conception, leur vérification, leur implémentation, leur maintenance et leur réutilisabilité.

Sergio Mena et d'autres membres du Laboratoire des Systèmes Répartis de l'Ecole Polytechnique Fédérale de Lausanne (Suisse), au sein du projet CRYSTALL, étudient cette approche modulaire des communications de groupe. Pour ce faire, ils utilisent des frameworks de composition de protocoles.

Notre mémoire présente une comparaison de deux de ces frameworks (Appia et Cactus) afin de déterminer le plus adéquat pour la suite du projet CRYSTALL. Pour ce faire, nous avons implémenté un service Atomic Broadcast avec une architecture originale.

Abstract

Group communications provide a set of primitives allowing broadcasts or multicasts with some guarantees. Reliability is the first guarantee asked by distributed algorithms.

Working in an imperfect world, failures within the systems must be taken into account. We speak about networks' failures involving important delays in the messages' reception, but also about failures on machines' level on which run processes take share in a distributed algorithm.

Fault-tolerant group communications is a very complex subject. It is thus preferable to break up these primitives into subproblems, which will facilitate their design, their checking, their implementation, their maintenance and their reusability.

Sergio Mena and the other members of the Distributed Systems Laboratory of the Ecole Polytechnique Fédérale de Lausanne (Swiss), within the CRYSTALL project, study this modular approach of group communications. To do that, they use frameworks for protocol composition.

Our master thesis will present a comparison of two of these frameworks (Appia and Cactus) in order to determine the most adequate for the continuation of the CRYSTALL project. With this intention, we have implemented an Atomic Broadcast service using an original architecture.

Remerciements :

Nous désirons remercier, notre promoteur, le Professeur Jean Ramaekers pour ses conseils avisés lors de la réalisation de ce mémoire.

Un merci tout particulier à André Schiper et Sergio Mena et tous les membres du Laboratorie des Systèmes Répartis de l'Ecole Polytechnique Fédérale de Lausanne (Suisse) pour leur acceuil chaleureux. Nous tenons à les remercier pour leur patience et leurs réponses de qualité à toutes nos questions sur ce sujet complexe.

Merci à nos parents, Maud et nos amis qui ont toujours été près de nous durant les moments les plus difficiles.

Table des matières

Table des matières 7

Introduction 11

I Présentation théorique 13

1 Les communications de groupe 15
1.1 Introduction 16
1.2 Définitions 16
1.3 Communications de groupe 18
1.3.1 Le modèle static/no-recovery 19
1.3.2 Consensus 22
1.3.3 Le modèle dynamic/no-recovery 27
1.4 La réplication 31
1.4.1 Introduction 31
1.4.2 Réplication passive 32
1.4.3 Réplication active 33
1.4.4 Comparaison 35
1.5 Conclusion 35

2 Les frameworks 37
2.1 Introduction aux frameworks 38
2.2 Canevas de description de frameworks 38
2.2.1 Modèle de composition 38
2.2.2 Modèle d'interaction 39
2.2.3 Modèle de concurrence 40
2.3 Framework abstrait 40
2.3.1 Modèle de composition 40
2.3.2 Modèle d'interaction 40
2.3.3 Modèle de concurrence 45
2.4 Appia 46
2.4.1 Modèle de composition 46
2.4.2 Modèle d'interaction 49

2.4.3 Modèle de concurrence 52
2.5 Cactus 52
2.5.1 Modèle de composition 53
2.5.2 Modèle d'interaction 54
2.5.3 Modèle de concurrence 56

3 Communications de Groupe modulaires 61
3.1 Introduction 62
3.2 Description du service implémenté 62
3.2.1 Pile implémentée 62
3.2.2 Présentation brève des modules 64
3.2.3 Types de données 65
3.2.4 Sémantique des évènements 66
3.3 Rappels théoriques 68
3.3.1 Rappels sur UDP 68
3.3.2 Rappels sur TCP 70
3.4 Choix conceptuels 77
3.4.1 Choix envisageables 77
3.4.2 La solution adoptée 80

4 Description des modules 81
4.1 Introduction 82
4.2 Transport 82
4.2.1 Rappels 82
4.2.2 Les objectifs 82
4.2.3 Machine à états du module Transport 84
4.2.4 Problèmes liés à l'utilisation de TCP/IP 84
4.2.5 Structures de données et fonctions 86
4.2.6 Machines à états de Transport sur TCP 89
4.2.7 Solution aux problèmes rencontrés 94
4.3 Unreliable 95
4.3.1 Rappels 95
4.3.2 Machine à états 96
4.4 Reliable Pt2Pt 97
4.4.1 Rappels 97
4.4.2 Objectifs 97
4.4.3 Structures de données et fonctions 98
4.4.4 Pseudo-Code de Reliable Pt2Pt 99
4.5 Failure Detector 100
4.5.1 Rappels 100
4.5.2 Structures de données et fonctions 100
4.5.3 Pseudo-code du Failure Detector. 101
4.6 Consensus 102
4.6.1 Rappels 102

4.6.2 Structures de données et fonctions 104
4.6.3 Pseudo-code du Consensus 106
4.7 Atomic Broadcast 109
4.7.1 Rappels 109
4.7.2 Pseudo-Code de Atomic Broadcast 109

II Implémentation **113**

5 Architecture globale **115**
5.1 Introduction 116
5.2 Conventions et classes de base 116
5.2.1 Conventions 116
5.2.2 Classes de base communes 116
5.3 Code commun 117
5.3.1 Architecture utilisée avec un code commun 118
5.4 Approche modulaire et paradigme évènementiel 119
5.4.1 Rappel sur le framework abstrait 119
5.4.2 Transposition en Appia 120
5.4.3 Transposition en Cactus 122

6 Implémentation des modules **127**
6.1 Introduction 128
6.2 Le module Transport 128
6.2.1 Rappels 128
6.2.2 Problèmes rencontrés 128
6.2.3 Architecture implémentée 129
6.3 Le module Unreliable 130
6.3.1 Rappels 130
6.3.2 Problèmes rencontrés 130
6.3.3 Architecture implémentée 130
6.4 Le module Reliable Pt2Pt 130
6.4.1 Rappels 130
6.4.2 Problèmes rencontrés 131
6.4.3 Architecture implémentée 138
6.5 Le module Failure Detector 139
6.5.1 Rappels 139
6.5.2 Problèmes rencontrés 139
6.5.3 Architecture implémentée 139
6.6 Le module Consensus 140
6.6.1 Rappels 140
6.6.2 Problèmes rencontrés 141
6.6.3 Architecture implémentée 142
6.7 Le module Atomic Broadcast 143

III Comparaison des frameworks **145**

7 Comparaison **147**
7.1 Introduction 148
7.2 Similitudes 148
7.3 Différences 149
7.3.1 Modèle de composition 149
7.3.2 Modèle d'interaction 149
7.3.3 Modèle de concurrence 150
7.4 Tests de performance 150
7.4.1 Configuration 151
7.4.2 Tests 151
7.4.3 Analyse 152
7.5 Conclusion 154
7.5.1 Modèle de composition 154
7.5.2 Modèle d'interaction 155
7.5.3 Modèle de concurrence 155

Conclusion **157**

Bibliographie **159**

A Conventions de représentation **163**
A.1 Conventions des machines à états 163
A.2 Conventions du pseudo-code 163
A.3 Conventions des diagrammes de séquence 165
A.3.1 Diagrammes de séquence des protocoles réseaux 165
A.3.2 Diagrammes de séquence de threads 165

B Méthodologie Appia **167**

C Arbres d'exécutions atomiques **169**

Introduction

Ce mémoire présente le travail que nous avons réalisé durant nos quatre mois de stage au Laboratoire des Systèmes Répartis (LSR) de l'Ecole Polytechnique Fédérale de Lausanne (EPFL), en Suisse. Cette expérience fut riche d'enseignements touchant de près ou de loin le sujet qui nous intéressait, à savoir l'approche modulaire des communications de groupe.

Tout d'abord, il nous semble intéressant de préciser que le LSR mène des recherches sur la tolérance aux pannes dans les systèmes distribués. Ses membres étudient des moyens logiciels afin de rendre des applications distribuées fiables, sûres et disponibles. Pour ce faire, différentes techniques de réplication ont été développées.

La réplication consiste à lancer plusieurs instances d'une application (réplicas) devant tolérer les pannes. Ce qui implique que si l'une d'entre elles venait à tomber en panne, les autres soient toujours disponibles avec un état cohérent pour assurer le service. Ce qui nous garantit qu'aucune donnée ne sera jamais perdue.

La difficulté majeure liée à la réplication est de garder les états internes de ces réplicas cohérents entre eux. Toute modification effectuée sur l'un devant l'être sur les autres. Pour ce faire, il est nécessaire de disposer de primitives de communication appropriées : les communications de groupe.

Les communications de groupe fournissent différentes primitives pour communiquer au sein d'un groupe. Cela inclut des broadcasts avec différentes propriétés telles que l'atomicité, l'ordre, etc., mais également la possibilité de travailler avec un groupe fixe ou un groupe dynamique. Le tout en sachant que les membres du groupe peuvent tomber en panne à tout moment.

Notre stage s'insérait dans un projet beaucoup plus vaste : le projet CRYSTALL (Correct Modular Group Communication Middleware). Ce dernier s'intéresse à la conception, la vérification et l'implémentation des communications de groupe en adoptant une approche modulaire.

Dans une approche modulaire, l'utilisation de frameworks s'impose. Ceux-ci nous permettent de créer des modules avec une interface claire et bien définie, et des les assembler entre eux afin de fournir un service complexe.

Notre attention se portera sur les frameworks de composition de protocoles. Ceux-ci sont destinés à faciliter la conception et l'implémentation de primitives de communications évoluées.

Le but d'une telle approche pour les communications de groupe est de trouver

et de spécifier les modules pertinents. Ces modules peuvent déjà avoir été décrits, ou restent encore à être découverts. La difficulté se situe dans les dépendances cachées pouvant exister entre les différents modules.

Notre tâche au sein du projet CRYSTALL a été de comparer deux frameworks (Appia et Cactus) afin de déterminer lequel des deux était le plus approprié pour la suite du projet. Pour ce faire, nous avons implémenté un même service des communications de groupe dans ces deux frameworks en respectant les mêmes contraintes.

Notre mémoire comportera donc une présentation rapide des communications de groupe (Chap. 1) ainsi que des différents frameworks que nous avons utilisés (Chap. 2). Nous décrirons le service que nous avons implémenté (Chap. 3 et 4), et présenterons les choix que nous avons posés pour réaliser cette implémentation (Chap. 5) ainsi que les diffilcultés auxquelles nous avons dû faire face (Chap. 6). Ce qui nous permettra de faire une comparaison d'Appia et Cactus (Chap. 7).

Première partie

Présentation théorique

Chapitre 1

Les communications de groupe

Sommaire

1.1 Introduction 16

1.2 Définitions 16

1.3 Communications de groupe 18

1.3.1 Le modèle static/no-recovery 19

1.3.2 Consensus 22

1.3.3 Le modèle dynamic/no-recovery 27

1.4 La réplication 31

1.4.1 Introduction 31

1.4.2 Réplication passive 32

1.4.3 Réplication active 33

1.4.4 Comparaison 35

1.5 Conclusion 35

1.1 Introduction

Les applications distribuées demandent des primitives de communication de plus en plus évoluées. Ces primitives sont soit point à point ou point à multi-points. Nous parlerons aussi couramment d'unicast et de multicast (ou broadcast). Alors que les premières sont simples, les secondes peuvent s'avérer très complexes.

En effet, il est souvent demandé d'avoir certaines garanties sur un broadcast. Il est tantôt nécessaire de faire un broadcast fiable, et tantôt, nous aimerions une autre primitive nous garantissant que tous les messages soient reçus dans le même ordre par toutes les destinations.

En outre, il devient indispensable d'envisager le comportement de ces primitives si l'une des destinations venait à tomber en panne. Et de manière plus générale, il faut prévoir leurs réactions face aux problèmes inhérents au réseau au-dessus duquel nous travaillons.

Les primitives que nous trouvons le plus souvent ne nous offrent pas ce genre de garanties évoluées, obligeant le développeur à les implémenter lui-même. Ce travail très laborieux nécessite cependant une somme de connaissances dont il ne dispose pas nécessairement, ce qui les rend beaucoup trop fragiles.

Des chercheurs étudient donc le problème et ont défini les communications de groupe. Cette théorie très vaste fournit un ensemble de primitives permettant de communiquer au sein d'un groupe, et d'autres pour gérer ce dernier. Les communications de groupe forment une couche située entre chacune des applications distribuées et le réseau et permettent de leur fournir des facilités de communication.

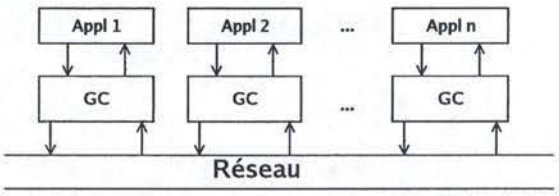


FIG. 1.1 – Les communications de groupe.

Nous allons, dans un premier temps, poser clairement tous les concepts que nous utiliserons dans la suite de ce mémoire (Sect. 1.2), ce qui nous permettra de décrire les communications de groupe et quelques primitives en faisant partie (Sect. 1.3). Enfin, nous montrerons un domaine dans lequel ces dernières peuvent être utilisées (Sect. 1.4).

1.2 Définitions

Processus : Nous allons dorénavant parler de *processus* pour désigner l'instance d'un programme manipulant un état interne.

Pannes : Un processus peut être sujet à des *défaillances* s'il dévie du comportement dicté par ses spécifications. Une défaillance est causée par une *erreur* dans l'état interne du système[1]. Une erreur est causée par une *faute* (qui peut elle-même avoir été causée par une défaillance). Il existe différents types de fautes :

1. les *pannes franches* (*crash faults*) sont les plus simples à gérer et consistent en l'arrêt brutal du processus. Le composant (processus) ne passe pas par des transitions incorrectes quand il tombe en panne.
2. les *ommissions* (*omission faults*) empêche un composant (processus) d'envoyer ou de recevoir un message.
3. les *fautes de timing* (*timing faults*) impliquent qu'un composant (processus) réponde trop tôt ou trop tard à un message.
4. les *comportements byzantins* (*byzantine faults*) sont des fautes qui causent un comportement complètement arbitraire du composant (processus) durant la défaillance.

Dans la suite de ce mémoire, nous ne nous intéresserons qu'aux pannes franches. Par simplicité, nous dirons qu'un processus tombe en panne s'il est sujet à une panne franche.

Processus correct et incorrect : Les processus pouvant tomber en panne, il est nécessaire de faire la distinction entre un processus *correct* et un processus *incorrect*.

- Un processus est dit correct s'il ne tombe pas en panne durant une exécution *infinie*.
- Un processus est dit incorrect s'il tombe en panne durant une exécution *infinie*.

Ces définitions peuvent facilement être généralisées aux autres types de fautes.

La notion de processus corrects et incorrects que nous venons de donner est très théorique, et malheureusement difficilement, voire impossible à vérifier dans la pratique. En effet, un processus est dit incorrect à son lancement dès lors qu'il sera sujet, à un moment ou à un autre, à une panne. Cette définition très forte n'est utile et utilisable que dans des démonstrations théoriques. Dans la pratique, une panne surviendra toujours lors de l'exécution d'un algorithme particulier. Quand nous aurons une approche plus pragmatique, nous remplacerons donc ces définitions par :

- Un processus est dit correct s'il ne tombe pas en panne durant *l'exécution de l'algorithme*.
- Un processus est dit incorrect s'il tombe en panne durant *l'exécution de l'algorithme*.

[1] Le système est l'ensemble des composants matériels (réseaux, machines) et logiciels permettant le bon fonctionnement des processus.

Canaux de communication Vu que les processus se situent dans un système distribué, ils doivent communiquer entre eux. Cette communication ne peut se faire que par l'*échange de messages*. Ces messages voyageront au sein de *canaux fiables*. Un canal fiable satisfait à la condition :

- Si un processus p envoie un message m à un processus q et que q est correct, alors q finira, tôt ou tard, par recevoir m.

Cette définition est de nouveau très contraignante et ne sera utilisée que pour des démonstrations théoriques se voulant générales. Dans la pratique, nous parlerons de *canaux quasi-fiables* se rapprochant très fortement du standard TCP [17]. Un canal quasi-fiable satisfait à la condition :

- Si un processus p envoie un message m à un processus q et que p et q sont *tous les deux* corrects, alors q finira, tôt ou tard, par recevoir m.

L'accès aux canaux de communication est réalisé par les deux primitives SEND(m) et RECV(m).

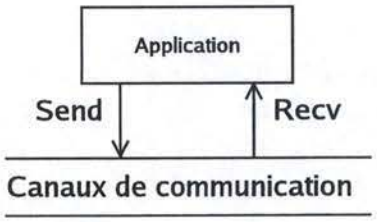


FIG. 1.2 – Pile classique.

Système asynchrone : Le système au-dessus duquel les communications de groupe ont été construites est dit *asynchrone*. Un système est asynchrone s'il n'existe pas de bornes supérieures aux temps transmission et de traitement d'un message (vitesse relative du réseau et des processus).

1.3 Communications de groupe

	Groupe statique	**Groupe dynamique**
Récupération après une panne possible	Modèle static/recovery	Modèle dynamic/recovery
Récupération après une panne impossible	Modèle static/no-recovery	Modèle dynamic/no-recovery

FIG. 1.3 – Les quatre modèles dans les communications de groupe.

Les communications de groupe peuvent être classées en quatre catégories selon deux grandes divisions, comme nous pouvons le voir à la figure 1.3. Le groupe peut être *statique* si après l'initialisation du groupe, il est impossible d'ajouter ou de retirer un processus du groupe. Il est dit *dynamique* si des processus peuvent rejoindre ou quitter le groupe tout au long de la vie de ce dernier. Le modèle peut également autoriser le retour d'un processus tombé en panne (*recovery*) ou non (*no-recovery*).

La différence entre les modèles n'autorisant pas le retour d'un processus tombé en panne et ceux le permettant ne nous intéresse pas, directement. Les nuances entre le modèles statiques et dynamiques sont plus intéressantes à étudier.

Nous aborderons, dans un premier temps, le modèle *static/no-recovery* (Sect. 1.3.1). Nous étudierons, ensuite, les différences avec le modèle *dynamic/no-recovery* (Sect. 1.3.3). Avant de traiter ce dernier, nous discuterons, plus profondément, des problèmes auxquels nous devons faire face dans un système distribué asynchrone en traitant le problème du consensus (Sect. 1.3.2).

1.3.1 Le modèle static/no-recovery

Le modèle *static/no-recovery* est certainement le plus simple à gérer. Les membres du groupe sont fixés au départ. Aucun processus ne peut y être ajouté. Si un membre tombe en panne, il sera exclu du groupe et ne pourra plus le rejoindre.

Abordons quelques primitives couramment utilisées. Nous commencerons par la plus simple, *Reliable Broadcast*, pour ensuite aborder *Atomic Broadcast* (ou *Total Order Broadcast*).

Reliable Broadcast

Reliable broadcast est la primitive de base des communications de groupe. Elle permet d'envoyer un message à l'ensemble des membres du groupe de manière fiable, même en présence de pannes franches des processus. Cette propriété n'est pas garantie par les primitives de plus bas niveau. En effet, un broadcast se fait soit à l'aide d'une adresse spéciale souscrite par tous les membres. Soit en envoyant individuellement le message à chaque membre. Dans le premier cas, toute la complexité se situe au niveau du réseau. Si un appareil tombe en panne, il se peut que seul une partie du groupe reçoive le message. Dans le second cas, le point critique se situe au niveau du processus qui envoie le message. S'il tombe en panne lors du $i^{\text{ième}}$ envoi, seul une partie du groupe recevra le message.

Définition : Reliable Broadcast est constitué par les primitives RBCAST(m) et R-DELIVER(m) qui satisfont aux propriétés suivantes [5] :

- *Validité :* si un processus correct exécute RBCAST(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer R-DELIVER(m).
- *Agrément :* si un processus correct effectue R-DELIVER(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer R-DELIVER(m).

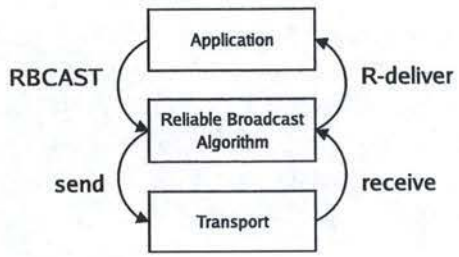


FIG. 1.4 – Pile Reliable Broadcast.

– *Intégrité :* pour tout message m, tous les processus corrects effectueront au plus une fois R-DELIVER(m) et cela seulement si un processus a effectué RBCAST(m).

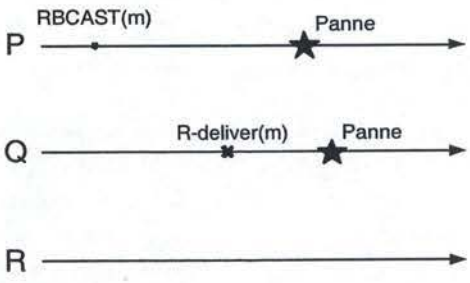


FIG. 1.5 – Exemple d'exécution de Reliable Broadcast.

Reliable Broadcast permet les exécutions dans lesquelles un processus incorrect effectue R-DELIVER(m), mais aucun processus correct n'effectue R-DELIVER(m). Considérons (Fig. 1.5), par exemple un processus qui effectue RBCAST(m) et tombe en panne. Un autre processus effectue R-DELIVER(m) et puis tombe en panne. Tous les autres processus sont corrects, mais aucun d'entre eux n'effectue R-DELIVER(m). Lors de cette exécution, aucune propriété de Reliable Broadcast n'a été violée.

Algorithme :

A titre d'exemple, nous allons donner un algorithme résolvant Reliable Broadcast avec des canaux quasi-fiables. Cet algorithme, très simple, ne se veut pas performant. L'algorithme consiste à envoyer un message m à tous les processus du groupe. Lors de la première réception de ce message, chaque processus le renvoie à tous les processus avant d'effectuer immédiatement un R-DELIVER(m) (Fig. 1.6). En pseudo-code[2], cela donne :

[2]Quelques conventions particulières utilisées dans le pseudo-code sont définies en annexe A.

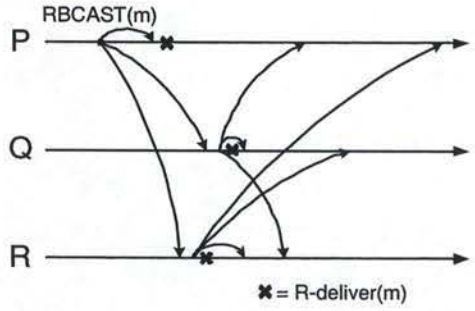
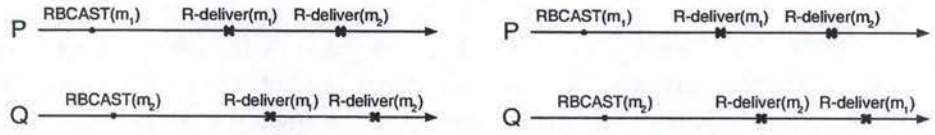


FIG. 1.6 – Reliable Broadcast.

```
Upon RBCAST (m) FROM p:
  SEND(m) TO all processes

Upon RECV(m) FROM p the first time:
  if (p != expediteur(m))
    SEND(m) TO all processes
  endif
  R-DELIVER(m)
```

Discution : Reliable broadcast permet l'envoi de messages de manière fiable mais non-ordonnée. Par exemple, comme illustré à la figure 1.7, deux processus P et Q effectuent, respectivement, RBCAST(m_1) et RBCAST(m_2). Ils peuvent délivrer[3] les messages dans le même ordre (Fig. 1.7(a)), ou dans un ordre différent (Fig. 1.7(b)).



(a) P et Q effectuent R-DELIVER(m_1) avant R-DELIVER(m_2)

(b) P effectue R-DELIVER(m_2) avant R-DELIVER(m_1) et Q l'inverse

FIG. 1.7 – L'ordre avec Reliable Broadcast.

[3]Nous devons ici faire la différence entre la réception d'un message et sa livraison. Nous parlons de réception quand le réseau délivre un message (à Reliable Broadcast, par exemple). La livraison est le moment où le message est délivré à l'application (à partir de Reliable Broadcast, par exemple).

Atomic Broadcast

Atomic Broadcast (ou Total Order Broadcast) ajoute une propriété d'ordre à Reliable Broadcast.

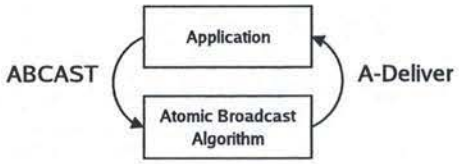


FIG. 1.8 – Pile Atomic Broadcast.

Définition : Atomic Broadcast est formellement défini par les primitives ABCAST(m) et A-DELIVER(m) qui satisfont aux propriétés suivantes [5] :

- *Validité :* si un processus correct exécute ABCAST(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer A-DELIVER(m).
- *Agrément uniforme :* si un processus (correct ou non) effectue A-DELIVER(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer A-DELIVER(m).
- *Intégrité :* pour tout message m, tous les processus corrects effectueront au plus une fois A-DELIVER(m) et cela seulement si un processus a effectué ABCAST(m).
- *Ordre total uniforme :* si un processus (correct ou non) effectue un A-DELIVER(m_1) avant un A-DELIVER(m_2), alors tous les processus effectueront A-DELIVER(m_2) seulement après avoir effectué A-DELIVER(m_1).

1.3.2 Consensus

Intuitivement, le consensus permet de décider une valeur parmi un ensemble de valeurs proposées par les membres d'un groupe. Le but étant, évidemment, que tout le monde décide la même valeur. Nous pouvons imaginer que cet algorithme pourra être utilisé tantôt pour définir l'ordre de livraison des messages (Atomic Broadcast, Sect. 1.3.1), tantôt pour décider qui fera partie du groupe à un moment donné (Group Membership, Sect. 1.3.3), ou alors pour résoudre bien d'autres problèmes dits d'agrément[4].

Le consensus n'a pas un grand intérêt pratique. Par contre, il est le bloc de base de nombre de primitives de plus haut niveau. Les résultats des études faites sur le consensus sont donc aussi applicables à ces primitives. Attardons nous particulièrement sur ce sujet. Cette partie sera fortement inspirée de [9].

[4]Problèmes où l'on doit se mettre d'accord sur quelque chose.

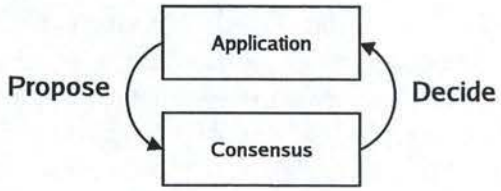


FIG. 1.9 – Pile Consensus.

Définition : Le consensus est défini formellement par les primitives PROPOSE(v_i)[5] et DECIDE(v)[6] qui satisfont aux propriétés suivantes [5] :

- *Terminaison :* tout processus correct finit, tôt ou tard, par décider une valeur.
- *Validité :* si un processus effectue DECIDE(v), cela veut dire que v a été proposé (PROPOSE(v)) par un processus.
- *Agrément :* deux processus corrects ne peuvent décider différemment.

La définition même du consensus n'est pas très compliquée, mais nous allons voir que sa mise en pratique peut s'avérer difficile.

L'impossibilité FLP (Fischer, Lynch et Patterson)

Fischer, Lynch et Patterson ont prouvé que dans un système asynchrone lorsqu'au moins un processus peut tomber en panne, il n'existe pas d'algorithme déterministe résolvant consensus [26]. Par soucis de généralité, cette impossibilité (*impossibilité FLP*) a été prouvée avec des canaux fiables.

Nous allons décortiquer l'impossibilité FLP afin de comprendre son fondement. Cela nous permettra d'en dégager plus facilement les conséquences.

Rappelons d'abord qu'un système est dit asynchrone quand il n'existe pas de bornes supérieures aux temps de transmission et de traitement d'un message. Dans ce système, lorsque nous attendons un message, nous ne pouvons donc pas savoir combien de temps il mettra pour nous parvenir. Si l'expéditeur est correct et que nous avons des canaux fiables, nous sommes certains de recevoir tôt ou tard ce message. Un message se faisant attendre peut donc être dû au caractère asynchrone du système, ou bien au fait que l'expéditeur soit en panne.

L'explication intuitive du résultat énoncé par Fischer Lynch et Patterson tient donc de l'impossibilité de distinguer un processus lent d'un processus tombé en panne. Si nous attendons un message d'un processus en panne, nous risquons de violer la propriété de terminaison. Mais si nous cessons d'attendre, alors que ce retard est dû au réseau particulièrement lent à ce moment, nous risquons de violer la propriété d'agrément[7].

[5] PROPOSE(v_i) : Le processus p_i propose sa valeur initiale (v_i).

[6] DECIDE(v) : Le processus décide la valeur v.

[7] Ou de *safety* en général. Ici, la propriété d'agrément est la seule propriété du consensus de type safety (rien de mal n'arrivera). Toutes les autres propriétés sont dites de *liveness* (ce qui est bien finira par arriver).

Certains critiquent l'impossibilité FLP en objectant qu'un système asynchrone est un objet purement théorique n'existant pas dans la pratique. En effet, si nous considérons un réseau local et que nous fixons un délai (timeout) de 30 secondes : si le processus destinataire ne répond pas dans ce délai, nous le considèrerons comme incorrect. Il est vrai qu'avec un tel délai, nous avons une probabilité très proche de 1 de suspecter correctement les processus incorrects. Dans ce cas, nous nous trouvons dans un système synchrone, et l'*impossibilité* perd tout son sens. Toutefois, cette solution n'est pas toujours satisfaisante. Prenons, par exemple, une application temps réel. Un tel délai n'est pas acceptable. Le réduire à 15 secondes, peut encore être acceptable en ce qui concerne la probabilité de fausses suspicions. Mais ce délai est toujours trop élevé pour garantir une bonne réactivité. Si nous passons à 0,5 seconde, ce délai devient intéressant. Par contre, nous obtenons une probabilité de fausses suspicions qui devient non négligeable. Nous retombons donc dans un système asynchrone et l'impossibilité reprend tout son sens[8].

Différentes solutions ont été proposées pour contourner cette impossibilité. Certaines portant sur des restrictions du modèle asynchrone (*partial synchrony* [8], *timed asynchronous* [12], ...), d'autres travaillant sur des algorithmes non déterministes ([18]). Finalement, la solution qui nous intéresse : la système asynchrone augmenté d'un *détecteur de pannes*[9] (*Failure Detector*), sorte d'oracle suspectant les processus.

Le système asynchrone avec détecteur de pannes

Un détecteur de pannes se veut une sorte d'oracle que l'on va greffer au système asynchrone. Le système peut se comporter tantôt de manière synchrone, tantôt de manière asynchrone. Cela rend toute affirmation à son sujet tout à fait illusoire. Nous allons définir un mécanisme permettant de combler ces aléas. Nous pourrons travailler dans un système asynchrone, sans faire d'hypothèses à son sujet. Par contre, nous ferons des hypothèses sur le détecteur. Connaissant le comportement attendu de ce dernier, nous pourrons construire et vérifier des algorithmes.

Nous parlerons souvent de détecteurs de pannes non fiables car, comme tout oracle, il peut se tromper. C'est pourquoi ils seront communément décrit par deux propriétés. La capacité à suspecter les processus incorrects (*complétude*) et la capacité à ne pas suspecter les processus corrects (*exactitude*). Ces deux propriétés peuvent être nuancées comme suit :

Complétude forte (Strong completeness)

Tout processus incorrect est finalement suspecté, pour toujours[10], par *tous* les processus corrects.

---

[8] Remarquons qu'un réseau local est un mauvais exemple de réseau asynchrone car il demeure relativement simple et n'est géré que par une entité. Si nous considérons Internet, nous pouvons réellement parler de réseau asynchrone car les temps de réponse varient fortement en fonction de la charge du réseau, et les pannes des composants du réseau des réseaux sont non négligeables.

[9] Nous parlerons aussi de détecteur de défaillances ou de fautes.

[10] La notion de « *pour toujours* » peut ici encore être traduite par « *jusqu'à la fin de l'exécution de l'algorithme* ».

Complétude faible (Weak completeness)
Tout processus incorrect est finalement suspecté, pour toujours, par *(au moins) un* processus correct.

Exactitude forte (Strong accuracy)
Aucun processus n'est suspecté avant qu'il ne soit tombé en panne.

Exactitude faible (Weak accuracy)
Au moins un processus correct n'est jamais suspecté.

Exactitude finalement forte (Eventual strong accuracy)
Il existe un temps t après lequel *tous* les processus corrects ne sont plus suspectés par *aucun* processus correct.

Exactitude finalement faible (Eventual weak accuracy)
Il existe un temps t après lequel *au moins un* processus correct n'est plus suspecté par *aucun* processus correct.

Chandra et Toueg ont défini les classes de détecteurs de pannes suivantes :
- $\mathcal{P}$ (perfect) : complétude forte et exactitude forte.
- $\diamond\mathcal{P}$ (eventually perfect) : complétude forte et exactitude finalement forte.
- $\diamond\mathcal{S}$ (eventually strong) : complétude forte et exactitude finalement faible.
- $\diamond\mathcal{W}$ (eventually weak) : complétude faible et exactitude finalement faible.

Ces mêmes auteurs ont prouvé qu'il existait une hiérarchie entre ces différents détecteurs de pannes. Le plus faible dont nous avons besoin pour résoudre le consensus est de type $\diamond\mathcal{W}$ [39]. Ils ont également prouvé que les détecteurs de fautes de type $\diamond\mathcal{S}$ sont équivalents à ceux de type $\diamond\mathcal{W}$[11] [38]. Comme les algorithmes se basant sur $\diamond\mathcal{S}$ sont plus simples, nous allons désormais travailler avec ce dernier.

Notons que les détecteurs de pannes sont souvent mal compris et considérés, à tort, comme des artefacts de théoriciens. Il est vrai que le détecteur de pannes $\diamond\mathcal{S}$ n'est pas implémentable dans un système asynchrone. L'implémentation ne peut qu'essayer de se rapprocher le plus possible de la spécification. Celle-ci rencontrera les spécifications quand le système sera stable, i.e. quand les délais de transmissions sont « *raisonnables* », ce qui est souvent le cas. Quand le réseau sera surchargé ou qu'un équipement du réseau tombera en panne, ces délais s'allongeront et feront croire, injustement, au détecteur que le processus que l'on surveille est tombé en panne. Nous savons, de par les spécifications, qu'un détecteur peut se tromper. C'est donc au concepteur de l'algorithme d'en tenir compte.

Cette approche a l'avantage de nous permettre de construire des algorithmes ne violant jamais les propriétés de type safety (l'agrément, dans le cas du consensus). Par contre, nous ne pouvons pas garantir les propriétés de type liveness (la terminaison, dans le cas du consensus) si le réseau se comporte indéfiniment de manière fortement asynchrone.

Les implémentations possibles d'un détecteur de pannes de type $\diamond\mathcal{S}$ peuvent être de type ping (requête/réponse) ou heartbeat (envoi périodique de messages). La difficulté réside dans le choix des timeouts. Trop petits, ils nous éloigneraient de

[11] $\diamond\mathcal{W}$ est réductible à $\diamond\mathcal{S}$ et vice versa.

la spécification, mais trop grands, ils induiraient une latence trop importante face à la panne d'un processus.

Un algorithme pour résoudre Consensus

De nombreux algorithmes existent pour résoudre le consensus dans un système asynchrone augmenté d'un détecteur de pannes [38, 23, 4]. Leurs fonctionnements sont légèrement différents, mais les concepts de base sont relativement similaires. Pour cette raison, nous choisirons celui de Chandra et Toueg [38].

Cet algorithme fonctionne dans un système asynchrone augmenté d'un détecteur de pannes de type $\diamond S$ pour autant qu'une majorité de processus soient corrects. Il utilise Reliable Broadcast (cfr. Sect. 1.3.1) et se base sur le paradigme du coordinateur tournant. L'algorithme se déroule en une succession de rondes durant lesquelles un nouveau coordinateur est élu[12].

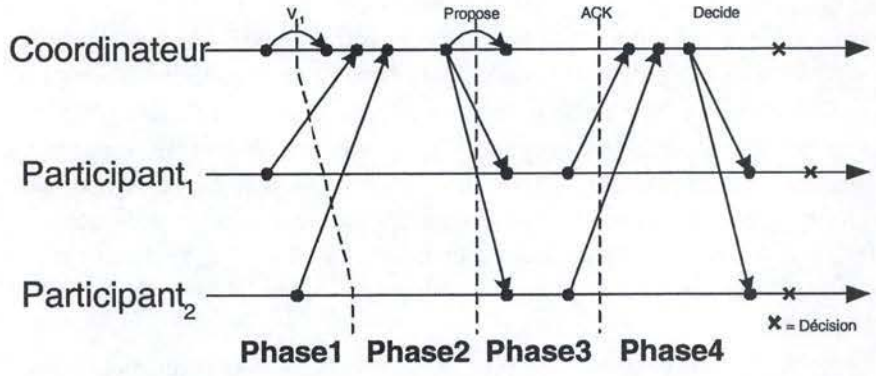


FIG. 1.10 – Exemple du déroulement d'une ronde du Consensus.

Lors d'une ronde, le coordinateur va essayer d'imposer une valeur parmi celles proposées par les participants. Une ronde est découpée en quatre phases (Fig. 1.10).

1. Dans la **première phase**, chaque processus envoie son estimation courante au coordinateur courant (c). Elle joint à cette estimation le numéro de la ronde durant laquelle celle-ci a été mise à jour pour la dernière fois.
2. Dans la **deuxième phase**, le coordinateur attend une majorité de ces estimations. Il sélectionne la plus récente et l'envoie à tous les processus.
3. Dans la **troisième phase** quand p_i reçoit l'estimation du coordinateur, il lui envoie un acquit (*ACK*). Cet acquit indique au coordinateur l'adoption de la nouvelle estimation. Si le détecteur de fautes de p_i suspecte le coordinateur avant de recevoir son estimation, p_i lui envoie un acquit négatif (*NACK*) et passera à la ronde suivante.

[12] Soient r, le numéro de la ronde et n le nombre de processus prenant part au consensus, le numéro du coordinateur de la ronde r est $(r \ mod \ n + 1)$ (pour autant que le numéro identifiant de chaque processus soit compris entre 1 et n).

4. Finallement, lors de la **quatrième phase**, le coordinateur attend une majorité d'ACKs ou un NACK. S'il reçoit une majorité d'ACKs, il décide l'estimation courante et effectue un Reliable Broadcast de cette décision. Quand les autres processus reçoivent cette décision, ils décident immédiatement. Par contre, si le coordinateur reçoit un NACK, il passe à la ronde suivante sans rien décider et sans envoyer de messages.

Applications pratiques

Comme nous l'avons déjà relevé, le consensus n'a pas un grand intérêt pratique. Par contre, il sert de bloc de base à un grand nombre de problèmes. Par exemple, Atomic Broadcast (cfr. Sect. 1.3.1) est réductible au consensus. En effet, nous pouvons lui trouver une solution simple en utilisant Reliable Broadcast (Sect. 1.3.1) pour envoyer les message et le consensus pour déterminer l'ordre dans lequel les messages seront délivrés à l'application [5].

1.3.3 Le modèle dynamic/no-recovery

Dans un premier temps, nous avons développé le modèle statique (Sect. 1.3.1), car il est plus simple pour introduire les concepts de base de communications de groupe. Nous allons, maintenant, voir brièvement comment gérer l'évolution dynamique du groupe. Ce qui nous permettra de présenter les primitives équivalentes à Reliable Broadcast et à AtomicBroadcast du modèle statique.

Nous nous attarderons moins sur cette partie car elle ne fera pas l'objet d'une étude plus approfondie dans la suite de ce mémoire.

Group Membership

Le *Group Membership* est le mécanisme permettant de gérer l'arrivée et le départ des processus au sein d'un groupe. Il va découper le temps en vues : ce sont des périodes durant lesquelles le groupe restera inchangé. Autrement dit, tant que nous ne changeons pas de vues, nous pouvons considérer que nous nous trouvons dans un modèle statique. La difficulté se trouve lors du changement de vue. Si un processus veut rejoindre le groupe ou le quitter, cela se fera en installant une nouvelle vue. Nous pouvons voir le lien de ce problème avec le consensus.

Celui-ci consiste à se mettre d'accord sur les membres faisant partie de la nouvelle vue.

La vue, notée $v_i(g)$, est l'ensemble des processus appartenants au groupe g au moment i[13]. Nous omettrons volontairement de mentionner le groupe à chaque fois, ce qui donnera la notation abrégée v_i.

[13] $v_i(g)$ représente en fait la i$^{\text{ième}}$ vue. Par abus de langage, nous pouvons utiliser ces numéros comme un repère temporel car ils croissent de manière monotone.

Définition : Le Group Membership se compose de trois primitives :

JOIN(p)

Exécuté par un processus actuellement membre du groupe, cette primitive demande l'ajout de p au sein du groupe.

LEAVE(p)

Exécuté par p ou tout autre processus membre du groupe, cette primitive demande l'exclusion de p du groupe.

INSTALL(v)

Appelé par le Group Membership, cette primitive signale à l'application que la vue courante a changé et est devenue v.



FIG. 1.11 – Pile Group Membership Problem.

Ces primitives satisfont aux propriétés suivantes [5] :

- *Validité :* Considérons deux vues consécutives v_i et v_{i+1}. Si $p \in v_i \setminus v_{i+1}$, alors un processus a exécuté LEAVE(p). Si $p \in v_{i+1} \setminus v_i$, alors un processus a exécuté JOIN(p).
- *Agrément :* Si un processus p dans la vue v_i installe la vue v_{i+1}, et qu'un processus q dans la vue v_i installe la vue v'_{i+1}, alors $v_{i+1} = v'_{i+1}$.
- *Terminaison :* Si un processus p dans la vue v, $p \in v$, exécute JOIN(q), alors, sauf si p tombe en panne, une vue v' telle que $q \in v'$ ou $p \notin v'$ sera finalement installée.

  Si un processus p dans la vue v, $p \in v$, exécute LEAVE(q), alors, sauf si p tombe en panne, une vue v' telle que $q \notin v'$ ou $p \notin v'$ sera finalement installée.

Notons que si un processus p du groupe est tombé en panne, un autre membre du groupe devra exécuter LEAVE(p) pour que les autres primitives des communications de groupe ne s'en préoccupent plus. De plus, tout processus ayant quitté le groupe ne pourra plus le rejoindre car nous travaillons dans le modèle no-recovery. Si un processus ayant quitté le groupe désire rejoindre le groupe, il est nécessaire que celui-ci change d'identifiant.

Exemple : La figure 1.12 représente une évolution possible de vues dans un système dynamique :

- La vue initiale v_0 se compose des processus P et Q ($v_0 = \{P, Q\}$)
- Le processus Q demande pour pour que le processus R rejoigne le groupe, la vue v_1 sera installée pour en tenir compte ($v_1 = \{P, Q, R\}$)

– Le processus P demande à quitter le groupe, ce qui entraînera l'installation de la vue v_2 ($v_2 = \{Q, R\}$

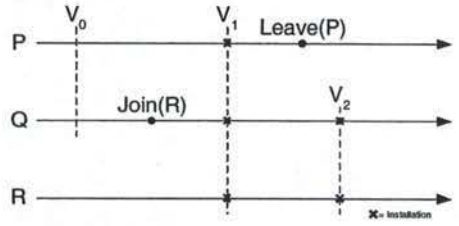


FIG. 1.12 – Evolution des vues d'un système dynamique.

Algorithme : Nous pouvons facilement résoudre le Group Memebership en le réduisant à Atomic Broadcast de la manière suivante :

```
Upon JOIN(p)
  ABCAST(add,p)

Upon A-DELIVER (add,p)
  v_{i+1} ← v_i U {p}
  INSTALL(v_{i+1})

Upon LEAVE(p)
  ABCAST(leave,p)

Upon A-DELIVER (leave,p)
  v_{i+1} ← v_i / {p}
  INSTALL(v_{i+1})
```

Discution : Un processus rejoignant le groupe a besoin d'initialiser son état interne. C'est ce que nous appelons le *transfert d'état*. Ce mécanisme est intégré dans la primitive JOIN. Lors de l'installation de la nouvelle vue, un processus qui fait partie de l'ancienne vue et qui fait toujours partie de la nouvelle vue enverra son état interne au(x) nouveau(x) membre(s) du groupe.

View Synchronous Broadcast

View Synchronous Broadcast peut être vu comme l'équivalent de Reliable Broadcast dans le modèle dynamique. Son rôle est donc d'envoyer un message à l'ensemble des membres d'une vue et ce d'une manière fiable.

Définition : View Synchronous Broadcast étend Group Membership et définit deux nouvelles primitives :

VSCAST(m)

Permet d'envoyer un message de manière fiable à l'ensemble des processus de la vue courante.

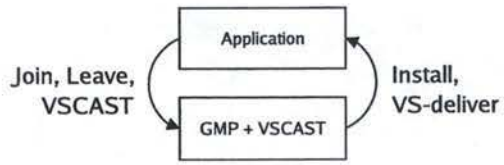


FIG. 1.13 – Pile View Synchronous Broadcast.

VS-DELIVER(m)
Délivre le message qui a été envoyé à l'aide de View Synchronous Broadcast.
La spécification informelle de VSCAST est la suivante :

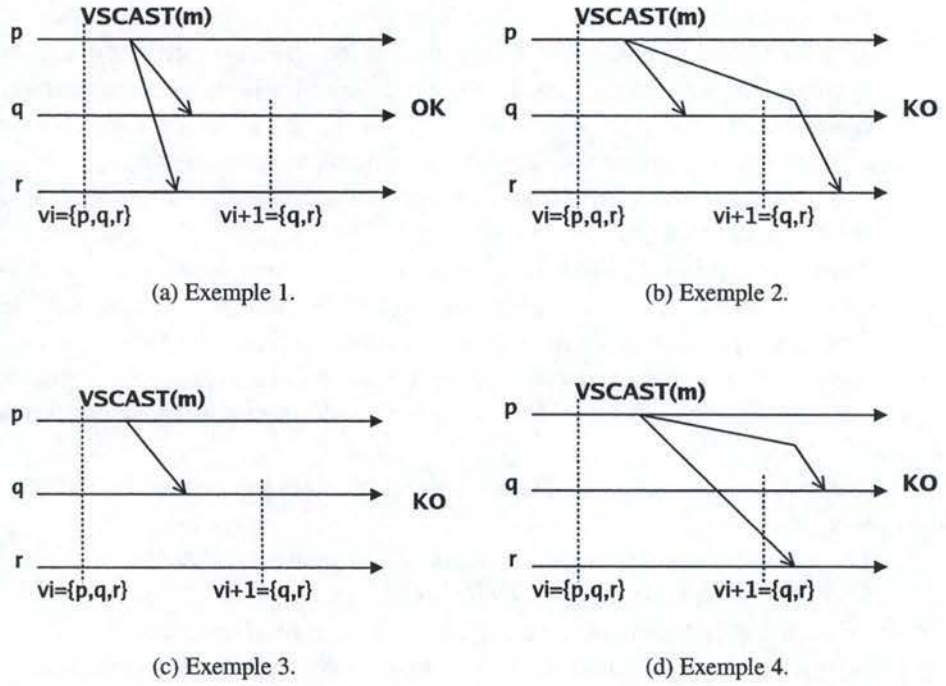
Si un processus p ($p \in v_i$) exécute VSCAST(m) dans la vue v_i :
- Soit tous les membres de v_i exécutent VS-Deliver(m),
- Soit une nouvelle vue v_{i+1} est installée, et si un processus a effectué VS-Deliver(m) avant d'installer v_{i+1}, alors chaque processus qui installe v_{i+1} a effectué VS-Deliver(m) avant d'installer la nouvelle vue.

Exemples : La figure 1.14 illustre l'interaction entre les vues et View Synchronous Broadcast, et met en évidence les propriétés de cette dernière. Seul le cas de la figure 1.14(a) représente une exécution correcte de View Synchronous Broadcast. p effectue VSCAST(m) dans la vue v_i et tous les autres membres de cette vue effectuent VS-DELIVER(m) avant le changement de vue. Le cas de la figure 1.14(d) est incorrect car p a effectué VSCAST(m) dans v_i et les autres processus effectuent VS-DELIVER(m) dans la vue v_{i+1}. Nous supposons implicitement que le processus effectuant VSCAST(m) effectue immédiatement VS-DELIVER(m), ce qui force les autres processus de la vue à effectuer VS-DELIVER(m) dans la même vue.

Algorithme : Nous ne présenterons pas ici un algorithme complet permettant de résoudre View Synchronous Broadcast. Il serait trop complexe tout en n'étant pas très intéressant pour la suite de l'exposé. Nous nous limiterons donc à en donner une idée générale et intuitive.

View Synchronous Broadcast ne peut pas être construit à partir du Group Membership. Au lieu d'utiliser une réduction à Atomic Broadcast, nous allons utiliser une réduction au consensus. La solution est basée sur la notion de *stabilité de message*.

Si un processus p_k envoie un message (m) à tous les processus de la vue v_i ($p_k \in v_i$), alors le prédicat $stable_k(m)$ est vérifié si et seulement si p_k sait que tous les processus appartenant à v_i ont reçu m.



(a) Exemple 1.

(b) Exemple 2.

(c) Exemple 3.

(d) Exemple 4.

FIG. 1.14 – Illustration de View Synchronous Broadcast.

Atomic Broadcast

Dans le modèle statique, Atomic Broadcast était défini par les propriétés de Reliable Broadcast en y ajoutant la propriété d'ordre. Dans le modèle dynamique, Atomic Broadcast est défini par les même propriétés que View Synchronous Broadcast en y ajoutant en plus cette même propriété d'ordre.

Nous ne nous attarderons pas d'avantage sur ce point qui ne sera pas approfondi dans la suite de ce mémoire.

1.4 La réplication

Nous allons montrer ici des applications pratiques des primitives des communications de groupe que nous venons de présenter.

1.4.1 Introduction

La réplication est une technique permettant de tolérer les pannes. La tolérance aux pannes fait partie d'un sujet plus large : la *sûreté de fonctionnement* (*dependability*) qui est définie par les attributs suivants :

- *Fiabilité* (*fiability*) : Probabilité que le système ne tombe pas en panne sur une période donnée.
- *Disponibilité* (*availability*) : Temps durant lequel le système est disponible sur une période donnée (une fois que l'on a retiré le temps nécessaire aux réparations).
- *Sûreté* (*safety*) : Aucun évènement catastrophique ne surviendra
- *Sécurité* (*security*) : Prévention des accès et des manipulations d'informations non-authorisées.
- *Maintenabilité* (*maintenability*) : Facilité de maintenir le système.

La réplication fait plusieurs copies (*réplicas*) des processus et maintient leur état interne cohérent. Cette technique permet de masquer les erreurs et de récupérer l'état interne d'une copie à partir des autres. Ce qui lui donne une grande disponibilité, contrairement à d'autres techniques telles que les transactions, le checkpointing, le journaling, etc..

Les deux techniques principalement utilisées pour assurer la cohérence d'objets distribués sont :

- La **réplication passive** (*passive replication / primary backup*).
- La **réplication active** (*active replication*).

Il existe d'autres techniques, mais elles ne sont pas intéressantes pour la suite de notre exposé. Pour information, nous pouvons citer la réplication semi-passive [40].

1.4.2 Réplication passive

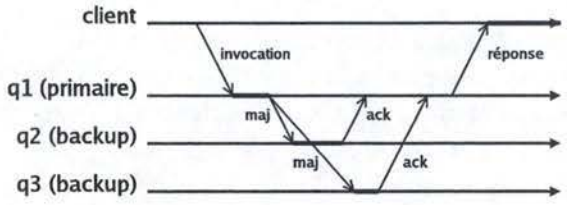


FIG. 1.15 – Réplication passive.

Dans le cadre de la réplication passive, nous distinguons trois acteurs. Nous avons tout d'abord un ou plusieurs *clients* qui envoient des requêtes à un processus que nous appellerons serveur[14]. Un *serveur primaire* qui reçoit les requêtes des clients. Un ou plusieurs *serveurs de backups* qui reçoivent les mises à jour du primaire.

Le protocole de mise à jour de l'état interne de ces différents serveurs est extrêmement simple. Le client envoie une *invocation* au primaire. Ce dernier va effectuer tous les calculs nécessaires afin de déterminer les mises à jour à effectuer.

[14] Afin de montrer la similitude avec le modèle client-serveur

Une fois cela fait, il enverra les modifications (*maj*) aux backups à l'aide de Reliable Broadcast ou de View Synchronous Broadcast (groupe statique ou dynamqie). Quand un serveur de backup aura effectué ces modifications, il enverra un acquit (*ack*) au primaire pour lui indiquer que tout s'est passé sans problèmes. Une fois que le primaire a reçu les acquits de tous les serveurs de backups, il enverra la *réponse* au client.

Si le primaire ne tombe pas en panne, alors l'*ordre*[15] et l'*atomicité*[16] sont assurés. L'ordre est défini par le primaire (coordinateur). L'atomicité est assurée car la mise à jour est envoyée à tous les backups par le primaire et qu'un acquit de leur part est attendu avant d'envoyer la réponse au client.

Si un serveur de backups venait à tomber en panne, il suffirait que le primaire s'en rende compte et qu'il cesse d'attendre les acquits de sa part.

La panne d'un serveur primaire est plus compliquée. Nous allons distinguer trois cas (Fig. 1.16) :

1. Le primaire tombe en panne avant d'envoyer la mise à jour (Fig. 1.16(a)) : il est convenu que le client attende un certain temps la réponse du primaire. Après ce temps, un nouveau primaire sera élu et le client renverra sa requête à ce dernier.
2. Le primaire tombe en panne pendant qu'il envoie les messages d'update ou avant d'envoyer la réponse au client (Fig. 1.16(b)) : le client va renvoyer sa requête au nouveau primaire. Il faut s'assurer que, dans le cas où tous les backups ont reçu le message d'update avant la panne du primaire, que la requête ne soit traitée une seconde fois. On peut aisément y arriver en ajoutant à chaque requête un numéro identifiant.
3. Le primaire tombe en panne après l'envoi de la réponse au client (Fig. 1.16(c)) : dans ce cas, un nouveau primaire doit être élu.

Notons que, généralement, la détection de la panne du primaire se fait à l'aide de timeouts. Ce mécanisme peut mener à de fausses suspicions (timeouts trop courts quand le réseau est fortement chargé). Nous avons vu (Sect. 1.3.2) comment gérer ces fausses suspicions en utilisant un détecteur de pannes.

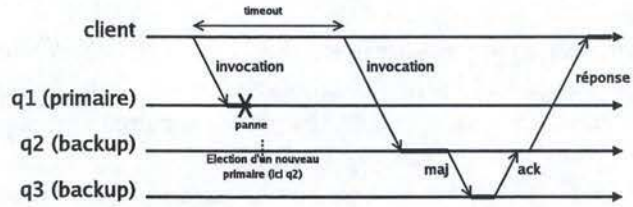
1.4.3 Réplication active

Dans le cadre de la réplication active, nous avons deux types d'acteurs : Un ou plusieurs *clients* qui envoient des *requêtes* et plusieurs *serveurs* traitant ces requêtes.

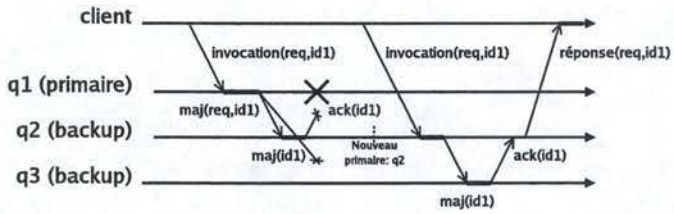
Le protocole de mise à jour de l'état interne des serveurs paraît encore plus simple que celui de la réplication passive. Un client envoie une *invocation* à tous les serveurs. Chaque serveur traite la requête du client et renvoie une *réponse* au client. Le client attend la première réponse et ignore les suivantes (qui sont identiques à la première si les traitements sont déterministes).

[15] Tous les serveurs exécutent les instructions dans le même ordre.

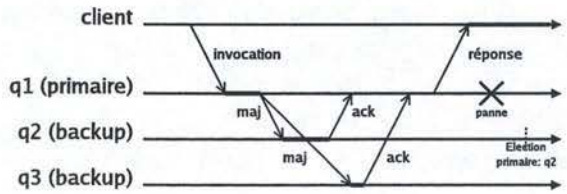
[16] La mise à jour est effectuée par tous les serveurs ou par aucun.



(a) Le primaire tombe en panne avant d'envoyer la mise à jour.



(b) Le primaire tombe en panne pendant qu'il envoie les messages d'update ou avant d'envoyer la réponse au client.



(c) Le primaire tombe en panne après l'envoi de la réponse au client.

FIG. 1.16 – Panne du primaire.

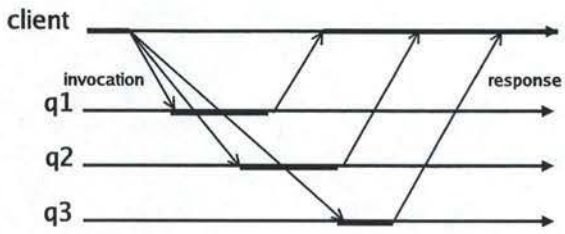


FIG. 1.17 – Réplication active.

Les propriétés d'ordre et d'atomicité n'étant plus gérées par un coordinateur, la réplication active nécessite une primitive de communication adéquate pour envoyer la requête à tous les serveurs et s'assurer que ces derniers traitent bien celles-ci dans le même ordre. Cette primitive est *Atomic Broadcast* (statique ou dynamique, selon les besoins).

La panne d'un serveur est totalement transparente pour le client car il n'attend que la première réponse.

1.4.4 Comparaison

Les deux techniques de réplication que nous venons de présenter sont très différentes. Nous ne pouvons pas dire que l'une soit meilleure que l'autre car tout dépend des priorités que l'on se fixe.

La réplication passive est faible consommatrice de temps calcul (seul le primaire est fortement sollicité), mais le temps de réaction suite à la panne du primaire peut être trop long pour une application temps réel.

La réplication active, quant à elle est tout à fait insensible à la panne d'un serveur. Par contre, elle exige que tous les serveurs soient performants et que les requêtes soient déterministes (contrairement à la réplication passive pour laquelle l'indéterminisme d'une requête ne pose aucun problème).

1.5 Conclusion

Comme nous venons de le voir tout au long de ce chapitre, les communications de groupe ne sont pas un sujet facile. De plus, elles font encore l'objet de nombreuses recherches afin de standardiser et clarifier les définitions pour les rendre plus accessibles aux personnes extérieures à la communauté de chercheurs dans le domaine. Ce ne sera que par cet intermédiaire que la tolérance aux pannes, en général, et les communications de groupe en particulier, pourront percer au grand jour.

Chapitre 2

Les frameworks

Sommaire

2.1 Introduction aux frameworks 38

2.2 Canevas de description de frameworks 38

2.2.1 Modèle de composition 38

2.2.2 Modèle d'interaction 39

2.2.3 Modèle de concurrence 40

2.3 Framework abstrait 40

2.3.1 Modèle de composition 40

2.3.2 Modèle d'interaction 40

2.3.3 Modèle de concurrence 45

2.4 Appia 46

2.4.1 Modèle de composition 46

2.4.2 Modèle d'interaction 49

2.4.3 Modèle de concurrence 52

2.5 Cactus 52

2.5.1 Modèle de composition 53

2.5.2 Modèle d'interaction 54

2.5.3 Modèle de concurrence 56

2.1 Introduction aux frameworks

Comme nous venons de le voir dans le chapitre 1, les applications distribuées demandent des primitives de communication de plus en plus complexes. Construire ces primitives de manière monolithique mène inévitablement à des codes complexes, peu flexibles et peu maintenables. Dès lors, il est souvent impossible de réutiliser les composants à cause des dépendances cachées. Et ce, bien qu'ils soient définis séparément.

L'utilisation d'un framework de composition de protocoles nous permet de construire séparément chaque bloc (ou module) et de les assembler en fonction du service que l'on désire rendre. Ceci rend le code plus flexible et maintenable pour autant que l'on respecte les règles fixées par le framework (ne pas introduire de dépendances cachées, ...).

Comme nous le verrons dans la suite de ce chapitre, un framework est conçu pour faciliter le travail d'un programmeur de protocoles. Mais avant de présenter les trois frameworks que nous allons utiliser, nous commencerons par un canevas permettant de décrire ces derniers (Sect. 2.2). Nous basant sur ce canevas, nous présenterons un framework abstrait (Sect. 2.3), Appia (Sect. 2.4) et Cactus (Sect. 2.5).

2.2 Canevas de description de frameworks

Vu que nous allons être amenés à présenter plusieurs frameworks, il nous semble intéressant de définir un canevas de description. En suivant ces lignes directrices, les trois présentations pourront être plus facilement comparées.

Les caractéristiques d'un framework peuvent être classées dans les trois rubriques (ou modèles) suivantes :

1. Modèle de composition.
2. Modèle d'interaction.
3. Modèle de concurrence.

Au vu de ces trois modèles, le framework peut offrir certains mécanismes de correction[1].

2.2.1 Modèle de composition

Le modèle de composition décrit la disposition voulue des modules afin de fournir le service attendu.

La composition peut être hiérarchique. Dans ce cas, les modules forment une pile. Ou elle peut être coopérative. Dans ce cas, aucune restriction n'est imposée quant à la structure.

[1] Mécanismes vérifiant si le protocole est correct selon un ou plusieurs critères.

La composition peut également comporter différents niveaux. Des modules peuvent être regroupés afin de fournir un service courant plus complexe que ceux founis par ses composants pris individuellement. Il peut donc être possible de créer des groupes de profondeur variable (groupes de groupes) et de les combiner ensemble (assembler un groupe avec un module).

Il est également envisageable que le framework vérifie que la composition soit correcte. Ceci est cependant difficile.

2.2.2 Modèle d'interaction

Le modèle d'interaction comporte deux aspects :
- Interaction interne
- Interaction externe

Modèle d'interaction interne

Le modèle d'interaction interne définit les moyens offerts par le framework aux modules pour interagir (communiquer) entre eux au sein d'une composition. La communication inter-modules est le point central de tout framework. Ce modèle déterminera fortement la modularité et la flexibilité du framework.

Les mécanismes de communication peuvent être, par exemple : les segments de mémoire partagée, les évènements synchrones ou asynchrones, l'échange de messages, etc. ou une combinaison de ces différents mécanismes. Notons que les mécanismes de communication entre modules et groupes de modules peuvent être différents.

Les mécanismes que nous venons de décrire offrent souvent différents types de services tels que le contrôle de flux des évènements, la synchronisation des segments de mémoire partagés, ...Ils offrent en outre des garanties telles que la garantire FIFO sur la livraison des évènements, ...

Modèle d'interaction externe

Le modèle d'interaction externe recouvre deux aspects bien distincts :
- L'interaction avec le système
- L'interaction avec le monde extérieur

L'interaction avec le système regroupe toutes les facilités offertes par le framework pour accéder de manière efficace aux ressources du système, à savoir les timers, l'accès disque, ...

L'interaction avec le monde extérieur nous montre les mécanismes fournis par le framework pour faire interagir une composition avec l'application et le réseau. L'interaction avec le réseau est souvent possible par l'interface d'un module écrit par les concepteurs. Celui-ci est livré avec le framework. Éventuellement, l'utilisateur peut réécrire lui-même un module d'accès réseau au cas où celui fournit ne donne pas satisfaction.

2.2.3 Modèle de concurrence

Le modèle de concurrence définit les parties de la composition pouvant s'exécuter en parallèle ainsi que les mécanismes de synchronisation offerts par le framework. Les extrêmes peuvent être vus comme le "thread par composition" (monothread) et le "thread par évènement". Un framework peut offrir plusieurs modèles de concurrences.

Il est également intéressant de décrire les mécanismes de correction offerts par le framework tels la prévention des interblocages, ...

2.3 Framework abstrait

Pour pouvoir raisonner dans le cadre d'une approche modulaire, nous allons définir un framework abstrait. Celui-ci nous permettra d'assembler les modules entre eux. Ce framework se veut indépendant de toute technologie et de tout langage de programmation. Il nous permettra de traduire, de manière plus ou moins automatique, les modules que nous aurons écrits pour ce framework dans un autre.

Nous ne nous sommes toutefois pas contentés d'un framework minimaliste. Comme nous décrivons un framework abstrait (i.e. n'étant pas destiné, dans un premier temps, à être implémenté), nous pouvons nous permettre de poser un certain nombre d'hypothèses facilitant son utilisation. Ces hypothèses doivent bien évidemment être réalistes. Nous devons pouvoir traduire les modules et les compositions définis par le framework abstrait dans un framework concret.

2.3.1 Modèle de composition

Le framework ne pose, a priori, aucune restriction sur les structures des compositions créées. Le modèle est alors dit coopératif (ou orienté graphe). Une composition est un ensemble de modules pouvant collaborer sans aucune restriction hiérarchique.

2.3.2 Modèle d'interaction

Nous allons étudier le coeur du framework, à savoir, son modèle d'interaction. Nous verrons dans un premier temps les mécanismes mis en place. Ceux-ci permettent aux modules de collaborer au sein d'une composition (interaction interne). Ensuite, nous décrirons les moyens dont dispose une composition ou un module pour interagir avec le système et le monde extérieur (interaction externe).

Modèle d'interaction interne

Le framework n'offre qu'un seul moyen d'interaction entre les différents modules d'une composition : le déclenchement d'évènements. Ces évènements auront les propriétés suivantes :

- point-à-point : un évènement qui a une destination unique et, par voix de conséquence, une seule source.
- asynchrones : il n'existe aucune garantie quant au délai après lequel un évènement sera traité.
- ouverts : il est possible de définir des nouveaux types d'évènements.
- avec garantie FIFO : si un module m_1 déclenche un évènement e_1 destiné à un module m_2, cet évènement sera délivré avant tout autre évènement destiné à m_2 que déclenchera m_1 après e_1.

Avant d'entrer dans les détails de fonctionnement des évènements, il est intéressant de fixer les structures de données qui nous serviront à les représenter. Nous pouvons ensuite décrire la manière d'offrir l'asynchronisme ainsi que la garantie FIFO.

Format des évènements Un évènement sera de la forme :

```
Event(Parameters)
```

Event est le type d'évènement permettant au framework d'exécuter le handler[2] correspondant dans le module de destination. Et *Parameters* sont les variables véhiculées par cet évènement.

Notons que certains paramètres peuvent être des messages. Les messages sont les seules structures de données pouvant transiter sur le réseau. Un message se compose d'une séquence d'en-têtes placée devant le message proprement dit :

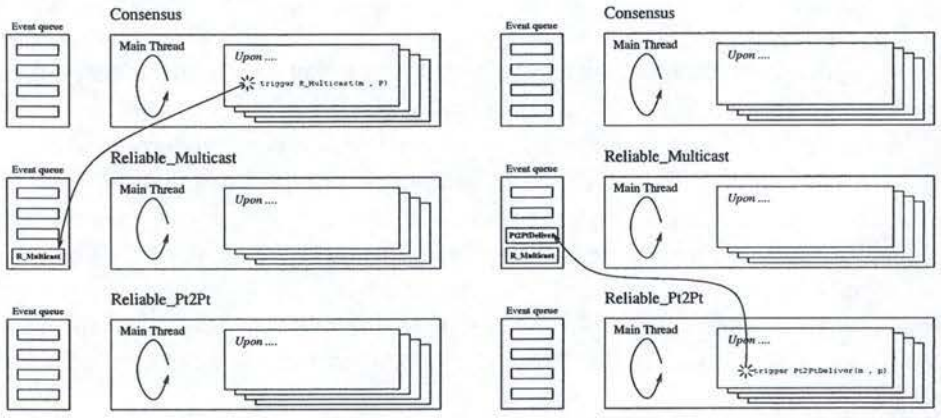
$h_1::h_2::...::h_n::m$
où h_i $(1 \leq i \leq n)$ sont des en-têtes
et m est un message pouvant comporter des en-têtes.

Tout module peut rajouter une en-tête au message. Ces en-têtes contiendront l'information nécessaire au processus destination afin de traiter correctement ce message. Ces en-têtes fonctionnent donc comme la pile de communication réseau (7 couches OSI). Cela s'effectuera pour autant qu'un message suive le même chemin en descendant (à la source) qu'en remontant (à la destination).

Evènements asynchrones et garantie FIFO Pour offrir l'asynchronisme des évènements échangés, chaque module dispose d'une queue FIFO centralisée comme décrite à la figure 2.1. A la figure 2.1(a) la couche du dessus déclenche un évènement lors de l'exécution d'un *Upon*. Après quoi (fig. 2.1(b)), le module du dessous déclenche un autre évènement qui est également mis dans la queue.

Les primitives que nous avons définies pour interagir avec la queue sont les classiques *push* et *pop*, que nous définirons comme suit :

[2]Un handler est une portion de code d'un module exécuté à la réception d'un évènement. Il existe un handler pour chaque type d'évènement auquel réagit le module. Dans la suite, nous noterons **Upon** $E(p)$ le handler associé à l'évènement de type E.



(a) Evènement du dessus (b) Evènement du dessous

FIG. 2.1 – Processus de déclenchement d'évènement.

Queue.push(Event(Parameters))

Ajoute *Event(Parameters)* à la fin de la queue *Queue*

Queue.pop()

Retire le premier élément non bloqué[3] de la queue *Queue* et le retourne. Si la queue est vide, le thread est bloqué jusqu'à ce qu'un évènement arrive dans la queue.

Ces primitives ne sont toutefois pas appropriées pour la conception de modules car elles s'appliquent à la queue d'un module particulier (que nous ne connaissons pas, et que nous ne voulons pas connaître). C'est pour cela que le framework offre les primitives **trigger** et **Upon** qui font abstraction de ces détails.

- **trigger** EVENT(parameters)

  Se traduisant par :

  ```
  m.Queue.push(EVENT(parameters))
  ```
  //En supposant que EVENT *soit pris en main par le module m.*

- **Upon** EVENT(parameters)
  //Code gérant le ***Upon***

Signifiant : « Lorsqu'un évènement de type *Event* est retiré de la queue, il faut exécuter le code qui suit en utilisant les paramètres de cet évènement. ». Cette primitive sera décrite plus formellement dans la section 2.3.3.

[3]Un évènement peut être bloqué dans la queue FIFO parce qu'il ne respecte pas la condition de la clause **when** associée au **Upon** devant traiter cet évènement (cfr. Sect. 2.3.3).

Comme nous manipulons des buffers (queues FIFO), nous devrions définir un mécanisme de contrôle de flux pour éviter que ces buffers n'explosent. Nous n'en ferons rien car nous décrivons un framework abstrait n'étant pas destiné à être implémenté (exécuté).

Evènements point à point Le fait d'utiliser des évènements *point-à-point* nous oblige à offrir à l'utilisateur un certain nombre de connecteurs. Un connecteur est un pseudo-module offert par le framework qui permet d'affiner le routage des évènements entre les différents modules.

Nous allons maintenant décrire les principaux connecteurs que nous avons dégagés[4].

Le multiplexeur total Le multiplexeur total permet d'envoyer un même évènement à deux modules. Dans l'exemple de la figure 2.2, le multiplexeur M copie les évènements provenant de $in1$ vers $out1$ et $out2$.

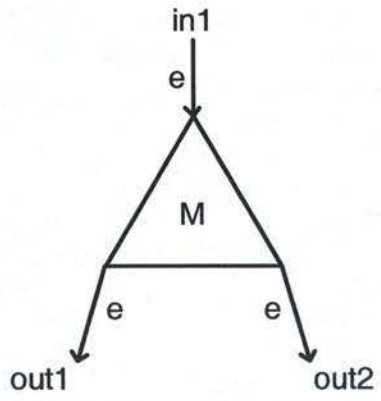


FIG. 2.2 – Le multiplexeur total.

De manière plus formelle, voici le pseudo-code de ce connecteur :

```
Upon Event(Parameters)
    //pour que les deux modules n'utilisent pas les mêmes variables
  Parameters' ← Parameters
  out1.Queue.push(Event(Parameters))
  out2.Queue.push(Event(Parameters'))
```

Le multiplexeur/démultiplexeur. Dans l'exemple de la figure 2.3, quand un évènement est envoyé vers le bas, D marque le message en y ajoutant une entête pour distinguer les évènements provenant de $in1$ ou de $in2$. En remontant, l'évènement transitera par le pseudo-module M qui retirera l'en-tête ajoutée par

[4]Cette liste n'est pas exhaustive, mais est suffisante pour l'utilisation que nous en ferons dans la suite de ce mémoire.

D. Avec l'information de cette en-tête, M sera capable de choisir vers quel module ($out1$ ou $out2$) il doit envoyer l'évènement .

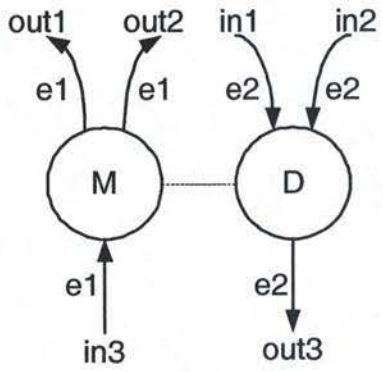


FIG. 2.3 – Le multiplexeur/démultiplexeur.

Supposons que $e1$ et $e2$ contiennent un message m auquel nous pouvons ajouter une en-tête[5]. Voici le pseudo-code des connecteurs :

- Pseudo-code du multiplexeur (M) :

```
Upon e1(..., inX::m, ...)
  switch(inX)
    'in1': out1.Queue.push(e1(..., m, ...))
           break
    'in2': out2.Queue.push(e1(..., m, ...))
  endswitch
```

- Pseudo-code du démultiplexeur (D) :

```
Upon e2(..., m, ...)
  trigger e2(..., e2.getSource()::m, ...)
    //getSource() Permet de connaître le module source de l'évènement.
    //getSource() n'est disponible que pour la gestion interne du framework.
```

Modèle d'interaction externe

Pour faciliter la tâche du programmeur, il est souhaitable qu'un framework offre un certain nombre d'abstractions. Celles-ci interagiront avec le système (accès disque, timers,...) et le monde extérieur (accès réseau, application, ...). Etant dans un framework abstrait, nous pouvons faire quelques hypothèses simplificatrices sachant qu'elles sont aisément implémentables (un module d'accès réseau basé sur TCP ou UDP, une gestion efficace des timers,...).

Une manière élégante de travailler avec des timers simples ou périodiques est d'utiliser des évènements destinés au framework. Par exemple, un module peut déclencher un évènement PERIODIC-TIMER(id, $periode$) afin de recevoir toutes les $periode$ millisecondes ce même évènement (le prévenant que le timer s'est écoulé).

[5]Ce connecteur ne peut fonctionner que si les évènements contiennent un message, dans le cas contraire, cela signifie qu'aucune information n'est destinée à voyager sur le réseau.

Déclencher PERIODIC-TIMER(id, 0) remet le timer à zéro. Nous ne recevrons donc plus d'évènement avant $periode$ millisecondes. Déclencher PERIODIC-TIMER(id, -1) annule ce timer périodique.

2.3.3 Modèle de concurrence

Le modèle de concurrence choisi est le "thread par module". Cette approche nous semble la plus intéressante car elle permet l'exécution concurrente d'un maximum de parties de la composition et cela en demandant un minimum d'efforts aux programmeurs de modules. Deux handlers d'un même module ne pouvant s'exécuter en même temps, il est superflu de synchroniser les structures internes de données. Cet aspect est important dans la mesure où le rôle d'un framework est de faciliter la vie des programmeurs.

Nous allons décrire le fonctionnement du thread principal d'un module[6]. Ensuite, nous analyserons les problèmes de synchronisation entre les modules.

Un thread par module

Notons qu'il existe deux types de handlers pouvant être exécutés par le thread principal. Cette distinction est introduite pour faciliter le travail du programmeur. Il est en effet possible de spécifier dans la signature du handler une clause (**when**) afin de retarder l'exécution de ce dernier. Un handler sans clause **when** peut être vu comme comportant une clause **when(true)** implicite.

Il est évident que la clause **when** va entraîner une violation de la garantie FIFO offerte par le framework. Le programmeur doit en être conscient pour utiliser les clauses à bon escient.

Dans un premier temps, nous verrons une version simplifiée du thread principal ne gérant pas les clauses **when**. Une fois les bases posées, nous verrons comment gérer la clause **when**.

Version simple, sans clause **when** :

```
// thread principal pour le module m
execute "Upon initialize" //Pour initialiser le module
while(true)
  (EVENT , parameters) ← m.Queue.pop()
  execute "Upon EVENT(parameters)"
endwhile
```

Version avec clause **when** [7] :

[6] Il serait envisageable qu'un module soit composé de plusieurs threads. Le thread principal est celui qui prend en charge les évènements entrants (exécute le handler).

[7] Ceci est un exemple d'implémentation. Cette version est très peu performante, mais permet au lecteur de prouver au lecteur que la gestion de la clause **when** est implémentable.

```
// thread principal pour le module m utilisant la clause when
execute "Upon initialize" //Pour initialiser le module
while(true)
  (EVENT , params) ← m.Queue.pop()
  let "Upon EVENT(params) when condition" existe dans m
  if(condition)
    execute "Upon EVENT(params)"
    somebody_executed ← true
    while(somebody_executed)
      //D'autres évènements retardés peuvent être exécutés
      somebody_executed ← false
      forall ((EVENT2 , pars2 , cond2) ∈ Buffer_When)
        //premier élément de la "queue"
        if(cond2)
          Buffer_When ← Buffer_When \ {(EVENT2 , pars2 , cond2)}
          execute "Upon EVENT2(pars2)"
          somebody_executed ← true
        endif
      endforall
    endwhile
  else
    Buffer_When ← Buffer_When ∪ {(EVENT , params , condition)}
  endif
endwhile
```

2.4 Appia

Appia est un « protocol kernel ». Il aide les programmeurs à réaliser des applications requérant plusieurs canaux de communication entre lesquels des contraintes peuvent exister. [15, 16, 14].

La seule implémentation existante d'Appia est écrite en JAVA. Nous décrirons donc Appia en utilisant les concepts des langages orientés objet.

2.4.1 Modèle de composition

Appia dispose d'un modèle de composition à la fois simple et sophistiqué. Simple car il n'autorise que des compositions hiérarchiques. Sophistiqué car il permet de mettre plusieurs modules à un même niveau hiérarchique. C'est la structure en forme de diamant. Mais avant d'en arriver là, nous allons d'abord poser les concepts introduits par Appia, ce qui nous permettra d'expliquer ladite structure.

Les concepts

Appia introduit un certain nombre de concepts représentés à la figure 2.4. Appia distingue la définition des modules (Protocol) de leur composition (Protocol Set). Il définit également les aspects statiques et dynamiques de ces deux concepts.

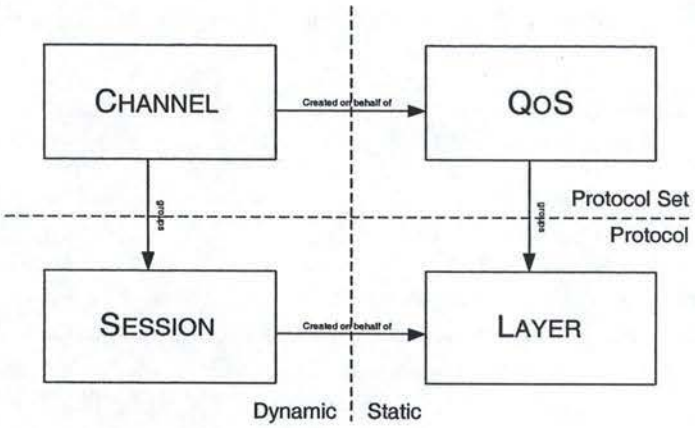


FIG. 2.4 – Relations entre LAYER, SESSION, QoS et CHANNEL.

LAYER Un LAYER (ou couche) définit le service fourni par le module (i.e. le « *Quoi* », ou plutôt le « *Avec quoi* »). Il n'implémente pas l'algorithme du module mais se limite aux aspects statiques.

Une couche définit trois ensembles d'évènements[8] :

- **Evènements fournis :** comprenant les évènements que le module fournira à d'autres modules.
- **Evènements acceptés :** comprenant les évènements que le module prendra en main. Si le code de l'algorithme s'attend à recevoir et prendre en main un type d'évènement qui n'est pas inclus ici, le noyau d'Appia ne délivrera pas ce type d'évènement à ce module.
- **Evènements Requis :** cet ensemble est un sous-ensemble des évènements acceptés. Les évènements déclarés ici sont ceux qui sont essentiels pour le fonctionnement de la couche.

SESSION Une SESSION (ou session) implémente le comportement du module (i.e.le « *Comment* »). Ce comportement est d'crit en fonction des évènements définis par la couche implémentée. Elle contient l'état du module (attributs de la classe), et des méthodes traitant les évènements entrants. Chaque session implémente une méthode $handle(\text{EVENT } e)$ servant de point d'entrée pour tous les évènements.

Cette définition autorise l'existence de plusieurs sessions qui implémente une même couche. Chaque session rend un service différent et utilise uniquement les évènements déclarés dans la couche.

QoS Une QoS est une séquence de couches (de bas en haut). Lors de la création de la QoS, le système vérifie partiellement la correction de la composition. Il vérifie

[8]Comme nous le verrons dans le modèle d'interaction interne, le seul moyen pour des modules de communiquer entre eux est le déclenchement d'évènements.

si les évènements requis par une couche sont bien fournis par une autre couche.

CHANNEL Un CHANNEL (ou canal) est une instanciation d'une QoS (instanciation de chaque couche de la QoS). Un canal est donc une séquence de sessions (de bas en haut).

Une pile Une pile est composée d'un certain nombre de canaux. Ces canaux peuvent avoir des sessions en commun pour autant qu'elles se trouvent au même niveau. Cette particularité donne lieu à ladite structure en diamant (illustrée par la figure 2.5). Nous pouvons y voir deux QoSs ($QoS1$ et $QoS2$) composées de trois couches (respectivement $[LS, LQ, LP]$ et $[LS, LR, LP]$). Chacunes des ces QoSs est instanciée par un canal (respectivement $Chan1$ et $Chan2$). Les deux canaux comportent deux sessions en commun (P et Q), alors qu'au niveau 2, ils comportent des sessions distinctes (respectivement Q et R).

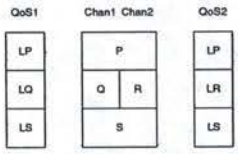


FIG. 2.5 – Pile Appia.

Structure en forme de diamant

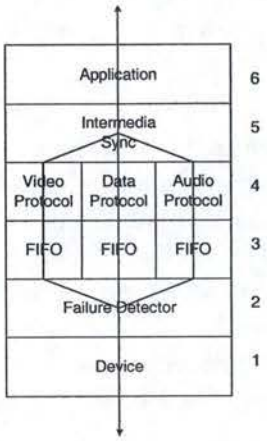


FIG. 2.6 – Exemple d'une pile Appia utilisant la structure en forme de diamant.

La figure 2.6 représente une pile qui permet la transmission de vidéo. Ce service est fourni par trois canaux : un pour la vidéo, un autre pour le son, et le dernier pour les méta-données. Ces canaux ne sont toutefois pas complètement séparés.

Dans l'exemple, il faut que le son soit synchronisé avec la vidéo. C'est pourquoi elles partagent toutes la même couche "Intermedia Sync".

Dans cet exemple, la pile sera réalisée très simplement en définissant trois QoS identiques au seul détail près de la quatrième couche. Lors de l'instanciation de ces trois QoS pour donner les trois canaux, il suffira de préciser que l'on souhaite la même session pour les couches 1, 2, 5 et 6, et laisser le framework instancier lui-même les couches 3 et 4.

2.4.2 Modèle d'interaction

Le modèle d'interaction bien que semblant restrictif à première vue est en réalité très puissant. Si le modèle d'interaction interne est bien défini, ce n'est pas le cas du modèle d'interaction externe.

Modèle d'interaction interne

Le modèle d'interaction interne est basé sur le déclenchement d'évènements. Ces évènements ont un certain nombre de propriétés qui font la force mais aussi la faiblesse d'Appia.

Un évènement Appia est dit :
- ouvert,
- asynchrone,
- suivant un canal,
- délivré de manière FIFO.

Nous allons décrire ces caractéristiques.

Evènements ouverts Les évènements sont dits ouverts car il est possible de définir de nouveaux types d'évènements. Appia définit un évènement de base (EVENT) qu'il est possible d'étendre par héritage. Il est évident qu'il est aussi possible d'étendre ses descendants directs ou indirects.

Un évènement de base est caractérisé par :
- une source : session ayant émis l'évènement.
- un canal : cheminement que suivra l'évènement.
- une direction : sens dans lequel l'évènement parcourra le canal (vers le bas ou vers le haut en partant de la source).

Nous pouvons relever deux descendants directs de EVENT prédéfinis par Appia :
- CHANNELEVENT
- SENDABLEEVENT

CHANNELEVENT Les CHANNELEVENT sont des évènements relatifs à la gestion du canal. Nous pouvons mettre en avant certains de ses descendants :
- CHANNELINIT qui permet d'ouvrir un canal.
- CHANNELCLOSE qui permet de fermer un canal.

- ECHOEVENT qui transporte un autre évènement en lui. Si une session au milieu d'un canal veut envoyer un message à toutes les sessions du canal, alors elle place cet évènement dans un ECHOEVENT et l'envoie vers le haut ou vers le bas. Quand l'ECHOEVENT arrive à une extrémité du canal, l'évènement qu'il contient est extrait et injecté dans le canal en sens inverse.
- DEBUG est utilisé à des fins de déboguage.
- TIMER (ou PERIODICTIMER) permet de lancer un timer simple ou périodique. Cet évènement est envoyé vers le bas. Quand il y arrive, le canal arme un timer. Lorsque le timer expire, un évènement de même type est envoyé vers le haut. C'est le seul moyen que les sessions ont pour utiliser les timers du système.

SENDABLEEVENT Les SENDABLEEVENT sont des événements contenant un message (MESSAGE).

Le message est la seule structure de donnée pouvant transiter sur le réseau. Un message est constitué d'une donnée unique aussi complexe que l'on veut. Les couches peuvent ajouter des en-têtes. Une en-tête ne peut être enlevée que par une couche soeur. Une en-tête peut également être une structures de données très complexe.

Evènements asynchrones Un évènement Appia est une structure de donnée transportant l'information nécessaire afin d'assurer la communication entre les différents modules de la pile.

Lancement des évènements Une couche lancera un évènement en appelant la méthode $go()$ sur ce dernier. Cette méthode utilise les champs source, direction et canal. afin de déterminer la couche destination.

Traitement des évènements Chaque session contient un handler unique : la méthode $handle$(EVENT e) définie dans SESSION. Appia appellera cette méthode lorsque la session traitera un évènement quelconque.

Le programmeur implémentera la méthode $handle$ de telle sorte qu'elle puisse distinguer le type exact de l'évènement afin de le traiter correctement. Ceci est possible grâce à l'opérateur $instanceof$ de JAVA. Cette technique permet à une vieille session de continuer à fonctionner, même si on n'utilise plus exactement l'évènement attendu. Un fils de l'évènement attendu sera vu par la session comme étant du même type que ce dernier.

Routage des évènements Appia optimise le routage des évènements en ne délivrant à une couche que les évènements nécessaires. Cela est rendu possible suite à la distinction faite entre les aspects statiques et dynamiques de la pile ainsi qu'au modèle d'évènements ouverts.

En effet, la couche définit les évènements fournis, acceptés et requis par les sessions l'implémentant. Ayant également fixé la QoS (séquence de couches), lors de la création du canal, Appia connaît déjà les évènements acceptés par chaque session. Cela lui permet de délivrer des évènements uniquement aux sessions qui en ont besoin.

Garantie FIFO Appia définit la garantie FIFO comme suit[9] :

Soit deux évènements A et B ayant la même direction. A est introduit avant B dans la pile. Si une session génère un nouvel évènement C (ayant la même direction que A) pendant le traitement de A, alors C sera traité avant B.

Contrôle de flux Dans la version que nous avons utilisée (kernel version 1.7, protocols version 0.9), le contrôle de flux n'était pas implémenté. Cela pouvait mener à des problèmes de saturation de la mémoire. Cet aspect est en cours d'étude.

Modèle d'interaction externe

L'interaction avec le système est une caractéristique importante qui permet de déterminer les facilités offertes par le framework. Nous distinguerons deux types d'interactions :

- L'utilisation au sein de la pile de services offerts par le système tel que les timers, le réseau, ...
- l'interfaçage de la pile avec l'application ou avec le réseau si nous n'utilisons pas les modules fournis.

Interaction avec le système Appia offre des timers simples et périodiques accessibles par le simple envoi d'un évènement (TIMER ou PERIODICTIMER, ou un de leur descendant). Une fois que le timer expire, cet évènement sera renvoyé en sens inverse pour l'indiquer à la couche[10].

Il existe également des couches d'accès réseau prédéfinies pour UDP et TCP (UDPSimple et TCPSimple). Ceci devrait éviter de devoir programmer à nouveau ses propres couches d'accès réseau.

Lors de l'envoi de messages sur le réseau, Appia ne veut pas utiliser la sérialisation JAVA car celle-ci est peu performante et peu portable (cfr. [14]). Il faut donc que le programmeur fasse lui-même la sérialisation « à la main ».

Toutefois, Appia offre au programmeur l'opportunité d'utiliser les OBJECTSMESSAGE[11] en lieu et place des traditionnels MESSAGE. Les OBJECTSMESSAGE

[9] Cette définition est tirée de [36] et ne figure dans aucun document. Elle a été trouvée en regardant le code source de l'EVENTSCHEDULER.

[10] Il est conseillé d'utiliser son propre type d'évènement descendant de TIMER ou PERIODICTIMER pour ne pas recevoir les timeouts des autres couches.

[11] Les OBJECTSMESSAGE sont définis dans le paquetage *appia.protocols.groups.events*. Notons également qu'ils étendent MESSAGE, ce qui garantit la compatibilité avec les vieilles couches utilisant les MESSAGE.

utilisent une sérialisation efficace pour certains types de données de base (String, Interger, ...). Pour les types plus compliqués, ils utilisent la sérialisation par défaut de JAVA.

Notons également que le fait d'envoyer un message sur le réseau n'est pas suffisant pour qu'il puisse être traité correctement à destination. En effet, lors de la réception d'un message, un évènement sera créé pour l'encapsuler et le faire voyager au sein de le pile. Malheureusement, ce message ne contient pas l'information nécessaire. Il ne comporte aucune indication ni sur le type exact de l'évènement à créer, ni sur le canal dans lequel il voyagera. Cette information devra figurer à côté du message.

Interfaçage L'interfaçage avec l'application et le réseau[12] n'est pas clairement défini.

L'application est une partie de code indépendante d'Appia. Elle peut insérer des évènements dans la pile[13]. Par contre celle-ci ne peut interagir avec l'application que de manière *ad hoc* (appel de méthode, queue de type producteur consommateur, ...). Cette caractéristique rend la frontière entre la pile et l'application très floue.

2.4.3 Modèle de concurrence

Le modèle de concurrence est très simple. Il n'existe qu'un seul thread principal pour gérer toute la pile. L'EVENTSCHEDULER est la partie la plus importante du thread Appia. C'est lui qui s'occupe du routage des évènements et de l'exécution du bon handler avec le bon évènement au bon moment.

Ce modèle garantit au programmeur la caractéristique suivante : le code d'une session ne sera exécuté que par un seul thread à la fois. Toute synchronisation au sein d'une session devient de ce fait tout à fait inutile.

2.5 Cactus

Cactus propose une approche différente de celle d'Appia. Appia était basé sur un modèle de composition unique en couches, Cactus est basé sur un double modèle de composition[6, 7] :

- un hiérarchique hérité d'x-Kernel[14][29].
- un sans hiérarchie qui communiquent par déclenchement d'évènements.

[12] Dans l'hypothèse où le programmeur souhaiterait redéfinir lui-même des couches d'accès réseau.

[13] L'insertion d'évènements dans une pile Appia par des threads extérieurs ne peut se faire qu'en utilisant la méthode $asyncGo$(CHANNEL $chan$,DIRECTION dir) définie dans EVENT. Cette méthode initialisera les champs canal et direction de l'évènement et l'insèrera au bas ou au sommet de la pile en fonction de la direction spécifiée (respectivement UP et DOWN).

[14] Une architecture pour implémenter des protocoles réseaux

La figure 2.7 est un exemple Cactus. Elle présente un service de sécurité afin d'envoyer des messages point à point à l'application (App). Elle permet de mettre en évidence les différents aspects de Catus. Ceux-ci seront présentés dans la suite de l'exposé.

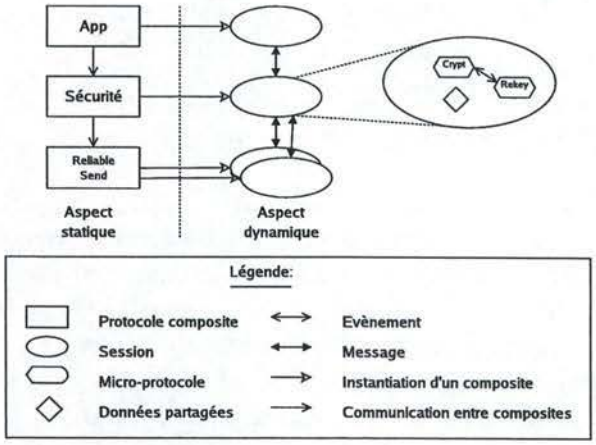


FIG. 2.7 – Exemples d'architecture Cactus.

2.5.1 Modèle de composition

Cactus repose sur un double niveau de composition. D'une part, nous avons un niveau macro composé de *protocoles composites* (ou *composites*), auxquelles vont être associé des *sessions*. D'autre part, nous avons un niveau micro, plus raffiné et inclus dans un composite. Ce niveau se compose de *micro-protocoles*. Ce double niveau donne à Cactus une grande flexibilité de composition. Nous allons maintenant définir et étudier les différents concepts énoncés.

Protocole composite : Défini de manière statique, il forme le bloc de base d'une composition. Son utilisation est double. D'une part, il peut être utilisé pour l'implémentation d'un service complexe (par exemple, un service sécurisé, un service multicast, un service de réplication, ...). D'autre part, il peut être utilisé pour encapsuler un acteur externe (l'application, la couche transport, etc...) et ainsi gérer la communication avec celui-ci. En effet, les acteurs externes n'étant pas construit selon le modèle Cactus, nous sommes obligé d'utiliser un composite pour interfacer ceux-ci avec le framework. L'architecture utilisée pour ordonancer les composites, est celle d'un graphe acyclique orientés comme illustré sur la figure 2.8. Un composite définit finalement les micro-protocoles inclus dans les sessions associées. Il définit également les événements qu'il est possible de déclencher en son sein.

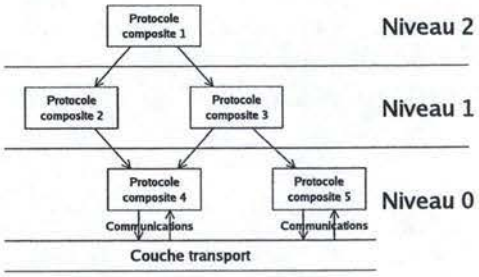


FIG. 2.8 – Exemple d'architecture de protocoles composites.

Session : Une session est la partie dynamique d'un protocole composite. En effet, le composite ne fait que définir des structures. Au cours de l'exécution, on associe à chaque composite zéro, une ou plusieurs sessions. Chaque session contient une instance des micro-protocoles définis dans le composite correspondant. Mais ceux-ci ne sont pas statiques, des micro-protocoles pourront être ajoutés ou retirés d'une session en cours d'exécution. L'utilisation de plusieurs sessions permet d'exécuter en parallèle plusieurs fois le service défini. Par exemple, si nous avons un composite dont le rôle est la gestion d'une connexion fiable unique, alors une session sera créée pour chaque destination. Deux sessions peuvent communiquer entre elles si les protocoles composites auxquelles elles se rapportent ont un arc les reliant. La communication a lieu par le déclenchement de messages.

Par la suite, nous utiliserons régulièrement le terme composite pour désigner la session attachée lorsque celle-ci est unique.

Micro-protocole : Les micro-protocoles, quant à eux, sont organisés sans aucune hiérarchie à l'intérieur d'un protocole composite. Ils coopèrent entre eux par l'échange d'évènements et l'utilisation de données partagées. Le rôle d'un micro-protocole est l'implémentation d'un service simple ou d'un algorithme. Un micro-protocole se définit comme une suite de handlers et d'un état interne privé. Un handler est une portion de code liée à un évènement et exécuté lorsque celui-ci est déclenché.

Pile : Le terme pile désigne l'ensemble des composites.

2.5.2 Modèle d'interaction

Cactus étant définit par un double modèle de composition, dans un premier temps, nous allons présenter les modèles d'interactions entre les sessions. Ensuite nous montrerons les possibilités d'interactions entre les micro-protocoles. Finalement, nous montrerons l'interaction entre micro-protoles et sessions.

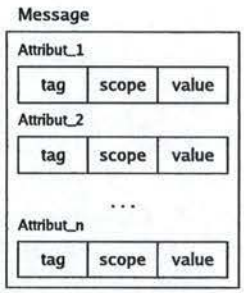


FIG. 2.9 – Format d'un message Cactus.

Interaction entre les sessions

Deux sessions communiquent entre elles par l'échange de messages. Un message est une structure de données, transmissible sur la couche transport et composée d'un ensemble d'attributs. La figure 2.9 représente le format général d'un message Cactus. Chaque attribut se présente sous la forme d'un tuple (*tag, scope, value*). Le *tag* (marqueur) permet d'identifier l'attribut dans l'ensemble. Le *scope* (étendue) indique la visibilité de l'attribut et peut prendre trois valeurs :

- il peut être visible uniquement dans le composite dans lequel il a été créé. L'étendue est dite *local*.
- il peut être visible uniquement dans la pile dans laquelle il a été créé. L'étendue est dite *stack*.
- il peut être visible uniquement dans les composites qui sont sur un même niveau quelque soit la pile. L'étendue est dite *peer*.

La *value* (valeur) contient la donnée du message, i.e., l'information qu'il véhicule. Les messages peuvent être envoyé vers le haut ou vers le bas. Lorsqu'un message est envoyé, si plusieurs sessions sont suceptibles de recevoir le message, alors l'utilisateur doit fournir une fonction de démultiplexage. Ceci afin de décider la session de destination. Cette décision est généralement basée sur certains attributs du message.

Interaction entre les micro-protocoles

Les micro-protocoles interagissent entre eux par des évènements. Lors de la définition d'un protocole composite, celui-ci définit un certain nombre d'évènements qui pourront être utilisés par les micro-protocoles le composant. La portée d'un évènement est donc celle du composite dans lequel il a été défini. Un évènement n'est défini que par son nom et ne comprend aucune autre information supplémentaire. On peut le voir comme un composant statique.

Au moment de l'initialisation, le micro-protocole lie ses handlers à des évènements définis dans le composite le contenant. Lorsque qu'un handler déclenche un évènement dans une session S, une occurrence d'évènement est créée. Elle est le pendant dynamique d'un évènement. Le handler peut y attacher une donnée quel-

conque. Ensuite, le framework exécutera tous les handlers des micro-protocoles de la session S liés à cet évènement. Il leur passera l'argument lié à l'occurence.

Jusqu'à présent, nous avons parlé du déclenchement d'un évènement par le code d'un seul handler. Une autre possibilité est de permettre à plusieurs handlers de déclencher un évènements lorsqu'ils ont chacun donné leur accord.

Une autre caractéristique de Cactus est la possibilité de lier et de délier des handlers en cours d'exécution. Celle-ci permet de modifier le comportement d'un micro-protocole en cours d'exécution.

Interaction entre micro-protocoles et sessions

Dans toute session, un certain nombre d'évènements sont prédéfinis afin d'interagir avec le framework. Nous nous attacherons aux trois évènements suivants : `sendUp`, `sendDown` et `msgRecv`. Ces évènements, une fois déclenchés, donnent lieu naturellement à une occurence d'évènement à laquelle un message sera attaché par le code du handler. Le message sera ensuite envoyé à une session supérieure (dans le cas d'un `sendUp`) ou à une session inférieure (dans le cas d'un `sendDown`). Lorsqu'un message arrive à la session de destination, une occurence de l'évènement `msgRecv` est créée à l'intérieur de celle-ci et prend comme paramètre le message.

Modèle d'interaction externe

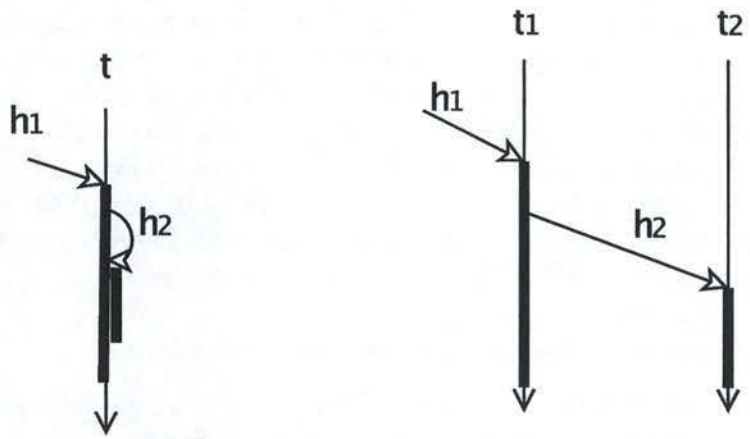
Au niveau de l'interaction externe, Cactus ne propose que peu de solution. En ce qui concerne l'interaction avec le système, rien est prévu par Cactus. Tout est à charge du programmeur. En ce qui concerne l'accès au réseau, UTP, un composite prédéfini permet l'envoi de messages de manière fiable (en utilisant TCP).

2.5.3 Modèle de concurrence

La communication entre micro-protocoles est réalisée par des évènements qui sont déclenchés de manière synchrone ou asynchrone. Cette dernière possibilité entraîne l'introduction de nombreux threads. Ceux-ci travaillant en parallèle sur des données partagées, nous devons étudier les problèmes engendrés.

Type de déclenchement d'évènement

Pour déclencher un évènement, un handler a deux possibilités. Il peut le déclencher de manière asynchrone. Nous parlerons d'un évènement levé. Dans ce cas, le thread qui exécute le code du handler crée un nouveau thread exécutera tous les handlers associés à cet évènement. Dans le second cas, le déclenchement a lieu de manière synchrone. Nous parlerons d'un évènement invoqué. Le thread qui exécute le code du handler s'interrompt et exécute le code des handlers associés à l'évènement invoqué. Les deux cas sont représentés sur la figure 2.10.



(a) Un évènement est levé par le handler h_1 et donne lieu à l'exécution du handler h_2 par le même thread t

(b) Un évènement est invoqué par le handler h_1 et donne lieu à la création d'un nouveau thread t_2 afin d'exécuter h_2

FIG. 2.10 – Déclenhements synchrones et asynchrones.

Ordre de déclenchement des handlers

Lorsqu'un évènement est déclenché, une occurrence est créée et un paramètre lui est éventuellement associé. Ensuite, le code de chaque handler associé à cet évènement est exécuté. Pour déterminer l'ordre d'exécution de ces handlers, lors de la liaison d'un évènement avec un handler, l'utilisateur peut lui donner un ordre de priorité. L'exécution a lieu de la priorité la plus élevée vers la priorité la plus basse. Lorsque plusieurs handlers ont la même priorité, l'ordre sera non-déterministe.

Concurrence pour l'envoi des messages

Un message est envoyé d'une session S_1 vers une session S_2 lorsque l'évènement `sendUp` ou `sendDown` est levé ou invoqué avec le message comme paramètre. Dans la session S_2 de destination, l'évènement `msgRecv` est levé de manière synchrone avec le message en paramètre. Comme l'envoi d'un message déclenche un évènement asynchrone dans la session de destination (S_2), nous pouvons dire que l'envoi des messages est asynchrone.

Gestion de la concurrence entre thread

Comme nous venons de le constater, plusieurs threads peuvent s'exécuter de manière concurrente à l'intérieur d'un protocole composite donné. Le problème

des appels synchrones est la création d'un grand nombre de threads différents. Une mauvaise gestion de la concurrence de ces threads peut entraîner une incohérence des données partagées et des interblocages. A priori, Cactus ne propose aucune solution pour gérer la concurrence. C'est à l'utilisateur de la prendre en compte.

Pour éviter ces problèmes, au lieu d'adopter une solution au cas par cas, il est assez intéressant de dégager des propriétés particulières. Nous allons étudier deux proriétés interessantes pour gérer la concurrence à l'intérieur d'un composite : les arbres d'exécution indépendante et la garantie FIFO. Ensuite, nous étudierons les difficultés pour garantir FIFO entre plusieurs composites.

Les arbres d'exécution indépendants dans un composite

Les arbres d'exécution exploitent une propriété interessante que nous avons dégagée au cours du stage. Ceux-ci définissent les handlers qui sont exécutés par un même thread.

Ensuite, nous avons défini une manière d'exécuter tous ces arbres de manière indépendante ; l'exécution d'un arbre dans le composite s'effectue comme s'il était le seul à s'exécuter dans celui-ci car il n'y a pas d'interférence nuisible avec les autres arbres.

En annexe C, nous trouvons une manière de représenter ces arbres et une façon d'implémenter la propriété d'indépendance en utilisant des mécanismes de synchronisation simples.

Garantie FIFO dans un composite

La garantie FIFO au niveau d'un protocole composite permet d'assurer la propriété suivante : "lorsqu'un message m_1 arrive à un protocole composite avant un message m_2, alors m_1 sera traité complètement dans ce composite avant m_2".

Cette propriété peut facilement être dérivée à partir des arbres d'exécution indépendants. Il suffit de ne créer qu'un seul arbre à l'intérieur du composite, i.e., utiliser que des appels synchrones. De cette manière, lorsqu'un message m_1 arrive, il déclenche un évènement asynchrone $msgRecv$ et y associe donc un nouveau thread. Ce thread s'active et s'exécute jusqu'à ce que tout l'arbre soit terminé, c-à-d jusqu'au traitement complet du message.

Garantie FIFO entre composites

La garantie FIFO entre composites permet d'assurer la propriété suivante : "lorsqu'un message m_1 est généré dans un composite quelconque avant un message m_2, alors m_1 sera traité entièrement avant m_2".

Pour assurer cette propriété, nous devons la garantir à l'intérieur des composites de la pile et entre ceux-ci. L'assurer à l'intérieur d'un composite a été etudié au point précédent. Voyons comment nous pouvons la garantir entre composites.

A priori, cette garantie ne peut être fournie par Cactus. Étudions le problème qui se pose sur un exemple simple. Supposons que nous ayons deux composites

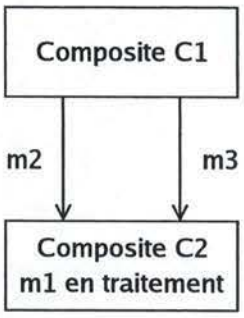


FIG. 2.11 – Garantie FIFO entre composites : exemple problématique

avec une session associée pour chacun. Ces deux composites sont illustrés à la figure 2.11.

Supposons que le composite C_2 soit en train de traiter un message m_1 par un thread t_1. Le composite C_1 génère un message m_2 et l'envoie à C_2. Au composite C_2, l'arrivée du message lève un évènement $msgRecv$ de manière asynchrone. Un nouveau thread t_2 est donc créé pour traiter ce message m_2 dans le composite. Comme t_1 associé à m_1 est en cours d'exécution, t_2 est suspendu. Le composite C_1 génère ensuite un message m_3 qui est envoyé au composite C_2. De la même façon que m_2, un thread t_3 y est associé et il est mis en attente. Lorsque C_1 termine l'exécution de m_1, un nouveau thread est choisi aléatoirement. Si t_3 est choisi, alors m_3 sera traité avant m_2 et la garantie FIFO ne sera pas respectée.

Pour résoudre ce problème, trois solutions s'offrent à nous. La première consiste à étendre le framework pour invoquer l'évènement $msgRecv$ de manière synchrone. De cette manière, un seul thread peut s'exécuter dans tout le framework à la fois. Par ce mécanisme, lorsqu'un message débute son traitement, aucun autre message ne pourra débuter le sien et ce aussi longtemps que le premier ne sera pas terminé. Cette solution limite le modèle de concurrence entre threads. La seconde solution consiste à ajouter des mécanismes pour numéroter les messages et demander aux composites de les traiter dans l'ordre de marquage. Cette solution augmente la complexité de conception des composites. La troisième solution consiste à n'utiliser qu'un composite unique. Cette solution entraine une réduction du modèle proposé par Cactus à un niveau de composition unique.

Chapitre 3

Communications de Groupe modulaires

Sommaire

3.1 Introduction 62

3.2 Description du service implémenté 62

3.2.1 Pile implémentée 62

3.2.2 Présentation brève des modules 64

3.2.3 Types de données 65

3.2.4 Sémantique des évènements 66

3.3 Rappels théoriques 68

3.3.1 Rappels sur UDP 68

3.3.2 Rappels sur TCP 70

3.4 Choix conceptuels 77

3.4.1 Choix envisageables 77

3.4.2 La solution adoptée 80

3.1 Introduction

Dans le chapitre 1, nous avons présenté la problématique des communications de groupe et quelques primitives de base. Dans le chapitre 2, nous avons présenté la programmation modulaire en définissant un framework abstrait et en présentant les caractéristiques de deux frameworks concrets, Appia et Cactus. Nous avons vu que ceux-ci simplifient la création de protocoles de communication.

Le but de notre travail au LSR[1] consistait à implémenter une pile de protocoles. Nous basant sur une approche modulaire, la description de celle-ci se base sur le framework abstrait. Cette pile fournit à l'application distribuée les primitives de base des communications de groupe dans le modèle dynamic/no recovery (Sect. 1.3.3). Ce service permet à l'application, par exemple, d'effectuer de la réplication active, Atomic Broadcast étant la primitive centrale.

Nous commencerons par décrire brièvement la pile implémentée (Sect. 3.2.1). Ensuite, nous rappelerons le fonctionnement des couches transport UDP et TCP et les formaliserons selon nos notations (Sect. 3.3). Finalement, nous présenterons plusieurs choix architecturaux possibles pour les couches transport (Sect. 3.4).

3.2 Description du service implémenté

Nous allons décrire la pile de protocoles que nous avons implémentée dans le cadre du projet (Sect. 3.2.1). Ensuite, nous décrirons brièvement chaque module la composant[2](Sect. 3.2.2), les types des données utilisés dans toute la composition (Sect. 3.2.3) et enfin les évènements déclenchés (Sect. 3.2.4).

3.2.1 Pile implémentée

En nous basant sur le framework abstrait, la figure 3.1 présente une pile implémentant un service Atomic Broadcast dans le modèle dynamic/no recovery (Sect. 1.3.1). Pour représenter la pile, nous utilisons les conventions définies dans le chapitre consacré au framework abstrait. Une boîte représente un module, les flèches représentent les évènements ; finalement, les cercles et les triangles sont utilisés dans le routage des évènements, ce sont les connecteurs.

Au plus bas niveau de la hiérarchie, nous avons les couches transport UDP et TCP sur lesquelles se basent la communication entre les piles. UDP (User Datagram Protocol) est un service sans connexion non-fiable avec détection d'erreur. TCP (Tranmission Control Protocol) est un service orienté connexion fiable avec contrôle de flux. Ces deux couches seront définies formellement aux sections 3.3.1 et 3.3.2. Au plus haut niveau, nous avons l'application distribuée qui communique avec d'autres applications en se basant sur notre service. Entre ces deux extrêmes, se trouve le corps de la pile : notre service Atomic Broadcast.

[1]Laboratoire des Systèmes Répartis, EPFL, lausanne

[2]Une description plus complète des modules sera faite au chapitre 4.

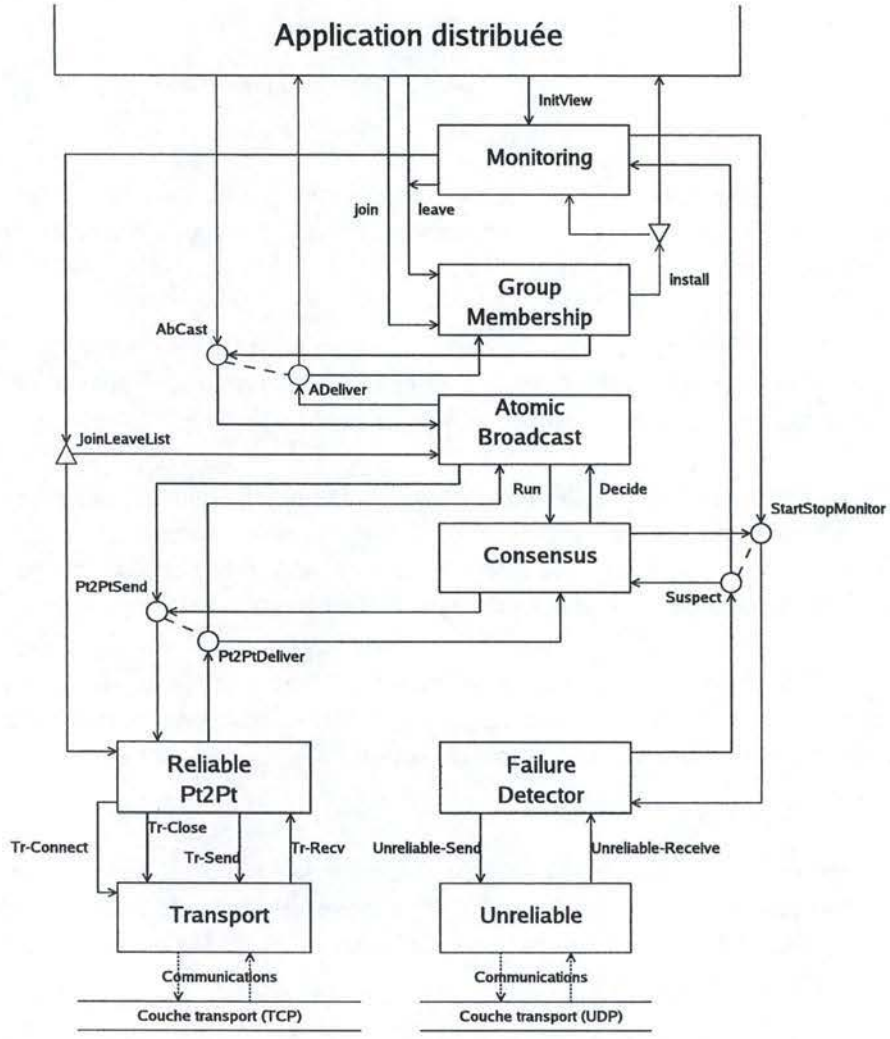


FIG. 3.1 – Modèle abstrait de la pile implémentée.

Le corps de la pile est composé d'un certain nombre de modules. Chacun d'entre eux implémente une primitive simple.

Dans le cadre de notre projet de mémoire, seule une partie de ces modules ont pu être implémentés. Nous nous sommes concentrés sur les modules transport (Unreliableet Transport), Reliable Pt2Pt, le détecteur de panne (Failure Detector), et Consensus(Consensus). De ce fait, nous présenterons brièvement le rôle de chaque module et nous étudierons en profondeur le fonctionnement de ceux que nous avons implémentés. Nous étudierons également le module Atomic Broadcast car il a été correctement défini et implémenté par un autre étudiant[20]. Les détails seront donnés dans le chapitre 4.

3.2.2 Présentation brève des modules

Nous décrivons les modules en commençant par les plus bas dans la hiérarchie jusqu'aux plus élevés.

Transport : Ce module sert d'interface entre la couche transport sous-jacente et le reste de la pile. Son objectif est l'implémentation des canaux de communication quasi-fiables unidirectionnels (Sect. 1.2).

Unreliable : Ce module sert d'interface entre une couche transport simple comme UDP et le reste de la pile. Il est utilisé lorsque des modèles de communication extrêmement simples sont nécessaires, i.e. sans aucune garantie.

Reliable Pt2Pt : Ce module implémente les communications fiables entre les processus. Basé sur le module Transport, il ajoute la notion de gestion du groupe. Un processus ne peut envoyer un message qu'à un autre processus du groupe. Un processus ne peut délivrer que des messages issus des processus du groupe.

Failure Detector : Ce module implémente un détecteur de pannes ayant un comportement proche de $\diamond\mathcal{S}$ (Sect. 1.3.2). Son fonctionnement ne recquiert aucune garantie en terme de communication. Il est donc construit au-dessus du module Unreliable.

Consensus : Ce module implémente l'algorithme Consensus basé sur un détecteur de pannes de type $\diamond\mathcal{S}$ (Sect. 1.3.2). Il se base également sur le module Reliable Pt2Pt pour communiquer avec les autres processus du groupe participant au consensus.

Atomic Broadcast : Ce module implémente l'algorithme Atomic Broadcast par réduction au Consensus (Sect. 1.3.1). De ce fait, ce module se base sur le module Consensus, mais également sur Reliable Pt2Pt pour la communication.

Group Membership : Ce module implémente une version de l'algorithme Group Membership en se basant sur le module Atomic Broadcast (Sect. 1.3.3).

Monitoring : Ce module sert de coordinateur entre les module Failure Detector, Group Membership et la couche applicative. Son rôle est d'utiliser l'algorithme Group Membership (module Group Membership) lorsque

- la couche applicative désire ajouter un processus (JOIN).
- la couche applicative désire retirer un processus (LEAVE).
- le module Failure Detector détecte un ou plusieurs processus en panne.

L'utilisation de Group Membership permet donc de changer la vue courante et donc les processus membres du groupe.

Application : Finalement, l'application peut ajouter (JOIN), retirer (LEAVE) des processus du groupe. Elle peut communiquer avec les autres processus du groupe grâce au module Atomic Broadcast. Elle est également avertie de chaque modification du groupe (INSTALL).

3.2.3 Types de données

Les messages : Un message correspond intuitivement à l'unité d'information qui transite sur la couche transport sous-jacente. Celui-ci a été formellement défini à la section 2.3.2.

Les évènement : Un événement correspond intuitivement à l'unité d'information qui transite entre deux modules. Celui-ci a été formellement défini à la section 2.3.2.

Les pID : Dans les communications de groupe, nous travaillons avec un groupe de processus. Le **pID** (ou process identifier) permet d'identifier de manière unique un processus de ce groupe. L'implémentation qui suit étant basée sur la couche réseau IP, le pID sera décrit par le tuple $(ip, port, inc)$:

- Le couple $(ip, port)$ désigne le socket d'une connexion, i.e., l'adresse réseau du processus. Cette structure sera définie dans la section 3.3.1.
- inc est le numéro d'incarnation. Il permet de différencier deux processus différents qui utilisent la même adresse réseau. En effet, comme nous travaillons dans le modèle no-recovery, lorsqu'un processus tombe en panne il l'est définitivement. Dès lors, il libère l'adresse réseau qu'il utilisait. De ce fait, si nous voulons lancer un nouveau processus sur la machine sur laquelle il tournait, sans le numéro d'incarnation nous serions obligé d'utiliser à chaque fois une adresse réseau différente. Grâce au numéro d'incarnation, nous pouvons lancé un processus différent qui utilise la même adresse réseau que celui tombé en panne en modifiant uniquement le numéro d'incarnation (par exemple, en incrémentant l'ancienne incarnation de 1).

De nombreux algorithmes doivent comparer et classer les processus. Dès lors, nous devons définir une relation d'ordre total sur l'ensemble des processus.
Pour rappel, une relation d'ordre total $\preceq$ sur un ensemble d'éléments $\mathcal{E}$ se définit par les quatre propriétés suivantes :

Réflexivité : $\forall x \in \mathcal{E} : x \preceq x$
Antisymétrie : $\forall x, y \in \mathcal{E} : x \preceq y \wedge y \preceq x \Leftrightarrow x = y$
Transitivité : $\forall x, y, z \in \mathcal{E} : x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$
Totalité : $\forall x, y \in \mathcal{E} : x \preceq y \vee y \preceq x$

3.2.4 Sémantique des évènements

Envoi et réception non fiables

UNRELIABLE-SEND(msg : **message** , dst : **pID**)
Envoie le message *msg* de manière non-fiable vers le processus *dst*. Ce message peut être perdu, dupliqué ou ne pas être dans la séquence.

UNRELIABLE-RECV(msg : **message** , src : **pID**)
Délivre le message *msg* issu du processus *src* (UNRELIABLE-SEND). Ce message peut déjà avoir été délivré en cas de duplication.

Gestion des connexions

TR-CONNECT(dest :**pID**)
Demande une ouverture de connexion vers le processus identifié par *dest*.

TR-CLOSE(dest :**pID**)
Demande la fermeture de la connexion établie avec le processus identifié par *dest*.

Envoi et réception fiables

TR-SEND(msg :**message** , dst :**pID**)
Envoie de manière fiable le message *msg* à *dst* en utilisant le module Transport. L'envoi n'est possible que si une connexion a préalablement été établie avec la destination *dst* (évènement TR-CONNECT).

TR-RECV(msg :**message** , src :**pID**)
Délivre le message *msg* envoyé par *src* de manière fiable (TR-SEND). L'ouverture préalable d'une connexion n'est pas nécessaire (évènement TR-CONNECT).

Gestion des processus du groupe

JOINLEAVELIST(New :**Set[pID]** , Left :**Set[pID]**)
Indique les processus à ajouter au groupe (*New*) ou à enlever de celui-ci (*Left*).

Envoi et réception aux processus du groupe

PT2PTSEND(msg :**message** , dst :**pID** , promisc :**boolean**)
Envoie de manière fiable le message *msg* à *dst*, où *dst* est un processus du groupe.
Si *promisc* vaut **true**, l'expéditeur est instantanément ajouté aux processus du groupe de *p* (à condition qu'il n'en fasse pas déjà partie). Ce mécanisme permet à un processus d'envoyer des messages à un processus qui n'appartient pas encore au groupe.

PT2PTDELIVER(msg :**message** , src :**pID**)
Délivre le message *msg*. Le message *msg* a été envoyé de manière fiable (PT2PTSEND) par *src* appartenant au groupe.

Exécution d'un consensus.

RUN(msg :**message** , P :**Set[pID]**)
Débute un consensus avec les processus contenus dans *P*, en proposant *msg* comme valeur initiale.
DECIDE(msg :**message**)
Délivre la décision *msg* ressortant du consensus ayant commencé précédemment (RUN). Cette décision a été prise selon les règles du consensus décrites dans la section 1.3.2.

Surveillance de processus.

STARTSTOPMONITOR(Start :**Set[pID]** , Stop :**Set[pId]**)
Commence la surveillance des processus repris dans *Start* et arrête la surveillance de ceux dont l'identifiant est contenu dans *Stop*.
SUSPECT(Suspected :**Set[pID]**)
Signale que les processus dont l'identifiant est repris dans *Suspected* sont suspectés d'être tombés en panne. Un processus qui disparaît de Suspected d'un évènement Suspect à l'autre signifie que ce processus n'est plus suspecté.

Exécution d'un Atomic Broadcast.

ABCAST(msg :**message**)
Demande l'exécution d'un Atomic Broadcast avec *msg* comme message.
A-DELIVER(msg :**message**)
Délivre le message *msg* envoyé par un Atomic Broadcast. Cette réception est conforme aux règles relatives à Atomic Broadcast definies dans la section 1.3.1.

Gestion du groupe par Group Membership

JOIN(process :**pID**)
Indique que le processus *process* rejoint le groupe.
LEAVE(process :**pID**)
Indique que le processus *process* quitte le groupe.
INSTALL(P :**Set[pID]**)
Indique qu'une nouvelle vue contenant les processus de l'ensemble *P* est installée en respectant les propriétés relatives au Group Membership (Sect. 1.3.3).

Initialisation de la pile

INITVIEW(P :**Set[pID]**)
Initialise la pile avec le groupe de processus P.

3.3 Rappels théoriques

Les communications de groupe se basent sur les couches transport existantes afin de fournir leurs services. Avant d'entrer dans l'étude de la conception de chaque module, il nous semble opportun de décrire les couches transport qui nous intéressent et de les formaliser selon nos conventions.

Pour décrire le fonctionnement des couches TCP et UDP, nous avons utilisé le schéma suivant :

- une définition intuitive de la couche.
- la description des types de données nécessaires à son fonctionnement.
- le modèle de communication (i.e., la façon dont la communication peut être établie entre deux applications).
- une formalisation des opérations permises par celles-ci sous la forme d'évènements.
- les machines à états représentant le fonctionnement de la couche selon le formalisme défini.

Cette formalisation des couches transport en terme d'évènements nous permet de simplifier l'exposé des module Transport et Unreliable. En effet, de cette manière, nous ne travaillons plus qu'avec un modèle d'interaction unique : évènementiel.

Les deux couches transport qui vont nous intéresser sont UDP et TCP. Nous allons décrire respectivement le fonctionnement d'UDP puis celui de TCP.

3.3.1 Rappels sur UDP

Définition

UDP est un service sans connexion, non-fiable avec détection d'erreurs. Le principe est extrêmement simple. Supposons qu'une application A désire communiquer avec B. Il suffit que A connaisse l'adresse de B et lui envoie des messages à cette adresse. Un message envoyé de cette manière peut être perdu, dupliqué ou désséquencé.

Identification d'une application

Pour pouvoir envoyer un message vers une application, celle-ci doit être facilement localisable et doit être identifiée de manière unique. L'identification est réalisée par un couple (adresse-ip, port), appelé *socket* :

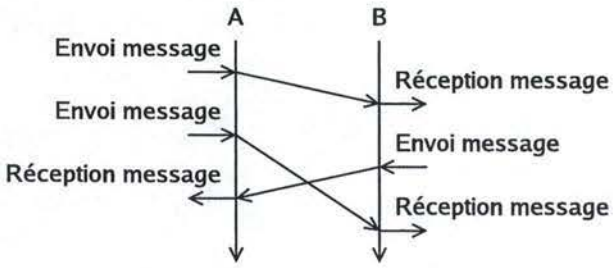


FIG. 3.2 – Fonctionnement intuitif de UDP.

adresse-ip
L'adresse IP permet d'identifier de manière unique une machine sur un réseau IP[3].

port
Le numéro de port permet de déterminer à quelle application se rapporte ce socket sur la machine.

Dans la suite, nous utiliserons le type de données **socket** pour désigner un socket.

Nous utiliserons les notations suivantes pour décrire la structure de données d'un socket. Soit S, un socket,

`S.ip` désigne l'adresse IP.

`S.port` désigne le numéro de port.

Nous définirons une relation d'ordre total $\preceq_S$ sur les sockets de la manière suivante : soit S_1 et S_2, deux sockets,

$$S_1 \preceq_S S_2 \Leftrightarrow (S_1.ip < S2.ip) \vee (S_1.ip = S2.ip \wedge S_1.port \leq S_2.port)$$

Modèle de communication

UDP est basé sur un modèle client-serveur. L'application serveur, pour recevoir des messages, doit choisir un socket dédié à la réception. Pour envoyer un message à l'application serveur, l'application cliente doit fournir un socket libre (inutilisé par aucune autre application de la machine) et connaître le socket de l'application serveur.

Évènements d'UDP

Les évènements déclenchés et reçus relatifs à la couche transport UDP seront utilisés par la suite pour définir le module Unreliable.

UDP-SEND(msg : **message**, dst : **socket**)

Envoie le message *msg* de manière non fiable à l'application serveur qui

[3]IP étant la couche réseau sur laquelle se base les couches transport TCP et UDP pour fournir leur service.

écoute sur son socket *dst*. Ce message peut être perdu, dupliqué ou désséquencé.

UDP-RECV(msg : **message**, src : **socket**)

Délivre le message *msg* de l'application cliente identifiée par *src*.

La figure 3.3 illustre les quatre cas possibles lors de l'envoi d'un message.

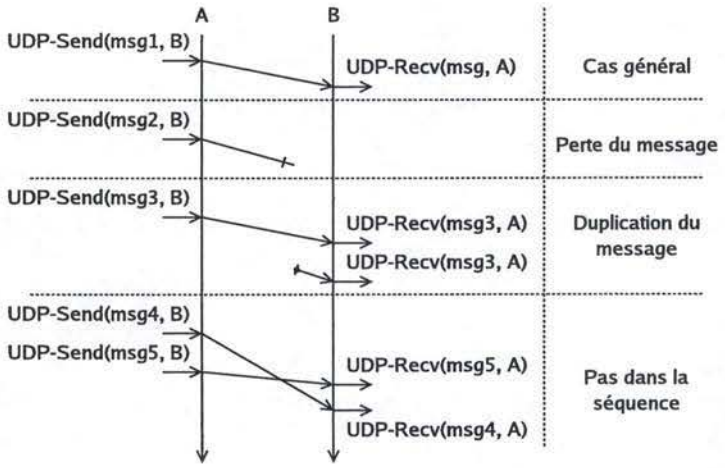


FIG. 3.3 – Envoi et réception non fiables par UDP.

Machines à états relatives à UDP

Les deux machines à états[4] de la figure 3.4 décrivent les évènements qui peuvent être déclenchés ou reçus au niveau de l'application cliente (Fig. 3.4(a)) et de l'application serveur (Fig. 3.4(b)).

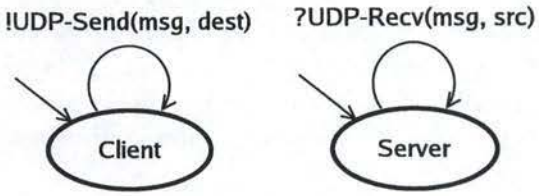
L'application cliente, connaissant le socket de l'application serveur, peut lui envoyer des messages par l'évènement UDP-SEND. L'application serveur écoute les messages que les clients lui envoient (évènement UDP-RECV).

3.3.2 Rappels sur TCP

Définition

TCP est un service orienté connexion qui permet l'envoi de messages entre deux applications de manière fiable : pas de perte, ni de duplication, ni de désséquencement de messages. Le principe est simple et se passe en trois phases. Supposons que nous ayons deux applications A et B qui désirent communiquer entre elles. L'application A ouvre une connexion vers l'application B, c'est la phase de connexion. Ensuite, lorsque cette connexion est établie, les deux applications A et

[4]Les conventions utilisées dans nos machines à états sont définies en annexe A.



(a) Machine à états de l'application cliente.

(b) Machine à états de l'application serveur.

FIG. 3.4 – Machines à états d'UDP.

B s'échangent des messages entre elles. C'est la phase de communication. Finalement, à n'importe quel moment, chaque application (soit A, soit B) peut décider de fermer la connexion dans une direction unique (de A vers B si A décide de fermer la connexion, et de B vers A dans le cas contraire), c'est la phase de déconnexion.

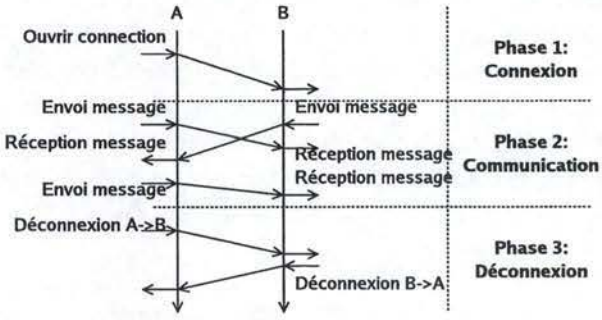


FIG. 3.5 – Les trois phases d'une connexion TCP.

Identification d'une connexion

Dans la description d'UDP, nous avons vu comment identifier une application à l'aide du concept de socket. L'identification d'une connexion TCP est réalisée par un couple de deux sockets. Le socket de l'application serveur sur lequel elle écoute les demandes de connexion et le socket de l'application cliente choisi arbitrairement.

Nous définissons le type **connexion** comme ce couple de deux sockets. Nous utilisons la notation suivante pour décrire les structures de données d'une connexion C :

$C.S_1$ désigne le premier socket du couple.

$C.S_2$ désigne le second socket du couple.

Nous définissons également une relation d'ordre total $\preceq_C$ sur les connexions. Nous commencerons par réorganiser les sockets à l'intérieur de la connexion de façon à ce qu'une connexion soit définie par le même couple chez le serveur et chez le client. Soit C, une connexion, C' désigne cette réorganisation :

$$C' = (S_1, S_2)\ si\ S_1 \preceq_S S_2$$

$$C' = (S_2, S_1)\ sinon$$

La relation d'ordre total entre deux connexions C_1 et C_2 se définit comme suit :

$$C_1 \preceq_C C_2 \Leftrightarrow (C_1'.S_1 \prec_S C_2'.S_1) \vee (C_1'.S_1 =_S C_2'.S_1 \wedge C_1'.S_2 \preceq_S C_2'.S_2)$$

Modèle de communication

Le principe de communication se base également sur le modèle client-serveur. L'application serveur va définir un socket composé de son adresse IP et d'un numéro de port choisi arbitrairement. L'application cliente est celle qui s'occupe d'ouvrir une connexion vers l'application serveur. Elle doit fournir deux informations au protocole TCP, son socket (composé de son adresse IP et un numéro de port encore inutilisé par une autre application) et le socket de l'application serveur. Les deux sockets (socket-client, socket-serveur) permettent d'identifier de manière unique la connexion TCP. Dans le cas de l'application cliente, la connexion est identifiée par le couple (socket-client, socket-serveur). Chez l'application serveur, elle est identifiée par le couple (socket-serveur, socket-client).

Évènements relatifs à TCP

Nous allons maintenant définir les évènements déclenchés et reçus qui permettent l'accès au réseau. Ils seront utilisés par la suite dans les machines à états qui définissent TCP et le module Transport. Dans la suite de la présentation, nous utiliserons les conventions suivantes pour désigner les connexions. Une connexion que nous désirons établir entre deux application A et B sera notée $(A - B)$. Nous utiliserons également les notations suivantes pour ajouter une information supplémentaire sur la direction de la connexion. $(A \leftrightarrow B)$ sera utilisé si elle est bidirectionnelle, $(A \rightarrow B)$ si elle est ouverte de A vers B et $(A \leftarrow B)$ si elle est ouverte de B vers A.

Évènements liés à l'ouverture d'une connexion TCP (Fig. 3.6) :

TCP-CONNECT(con : **connexion**)

Permet à l'application cliente de tenter d'ouvrir une connexion TCP avec l'application serveur. Le déclenchement de cet évènement donne toujours lieu à la réception d'un des deux évènements suivants : TCP-CONNECT-ACCEPTED si la connexion est établie, TCP-CONNECT-REFUSED si la connexion échoue.

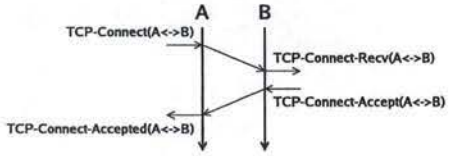
TCP-CONNECT-ACCEPTED(con : **connexion**)
Indique la réussite de la tentative d'ouverture de la connexion *con*. Il fait suite à la demande d'ouverture d'une connexion (TCP-CONNECT).

TCP-CONNECT-REFUSED(con : **connexion**)
Indique l'échec de la tentative d'ouverture de la connexion *con*. Le refus d'une connexion peut être lié à une coupure réseau, un refus ou une panne de l'application serveur. Il fait suite à la demande d'ouverture d'une connexion (TCP-CONNECT).

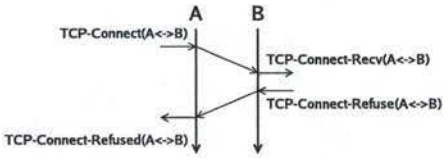
TCP-CONNECT-RECV(con : **connexion**)
Est déclenché lorsque l'application serveur reçoit une demande de connexion par une application cliente (TCP-CONNECT).

TCP-CONNECT-ACCEPT(con : **connexion**)
Permet d'accepter une demande de connexion de l'application cliente (appelé suite à la réception d'un TCP-CONNECT-RECV).

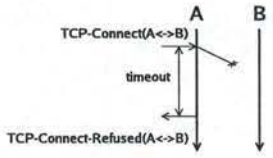
TCP-CONNECT-REFUSE(con : **connexion**)
Permet de refuser une demande de connexion d'une application cliente (appelé suite à la réception d'un TCP-CONNECT-RECV).



(a) Réussite de la connexion.



(b) Refus de la connexion.



(c) Echec de la connexion suite à un problème.

FIG. 3.6 – Connexion TCP.

Évènements liés à une déconnexion (Fig. 3.7) :
TCP-CLOSE(con : **connexion**)

Permet de fermer la connexion identifiée par *con* dans la direction issue de cette application. Les messages en provenance de l'autre application continuent d'arriver.

TCP-CLOSED(con : **connexion**)

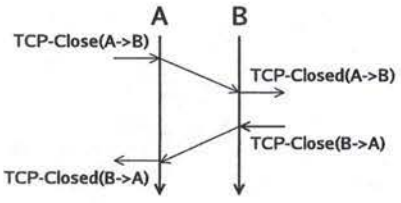
Indique que la connexion identifiée par *con* a été fermée dans la direction de cette application. Aucun message ne sera plus reçu sur cette connexion mais cette application peut continuer à envoyer des messages.

TCP-ABRUPT-CLOSE(con : **connexion**)

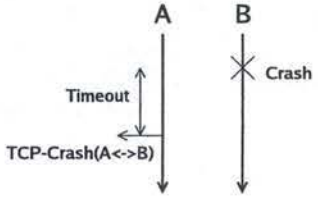
Permet de fermer la connexion dans les deux sens de manière abrupte. Une fois déclenché, la connexion se termine et aucun message ne peut plus transiter dessus. Les messages en cours de transmission peuvent être perdus.

TCP-CRASH(con : **connexion**)

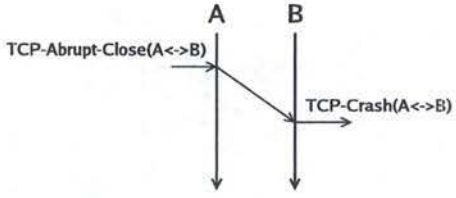
Indique la fermeture abrupte de la connexion. La connexion se termine et aucun message ne peut plus transiter. Cet évènement est déclenché suite à l'envoi d'un TCP-ABRUPT-CLOSE, suite à la panne de l'application distante, ou suite à une coupure réseau.



(a) Demande de fermeture de la connexion.



(b) Fermeture abrupte de connexion après une panne.



(c) Fermeture abrupte de la connexion.

FIG. 3.7 – Déconnexion TCP.

Évènements liés à la communication :

TCP-SEND(msg : **message**, con : **connexion**)

Permet d'envoyer le messages *msg* sur la connexion *con* de manière fiable.

TCP-RECV(msg : **message**, con : **connexion**)

Délivre le message *msg* issu de la connexion *con*.

Machine à états d'une connexion TCP.

La machine à états de la figure 3.8 explique l'enchaînement des différents évènements définis ci-dessus afin de gérer une connexion entre deux applications.

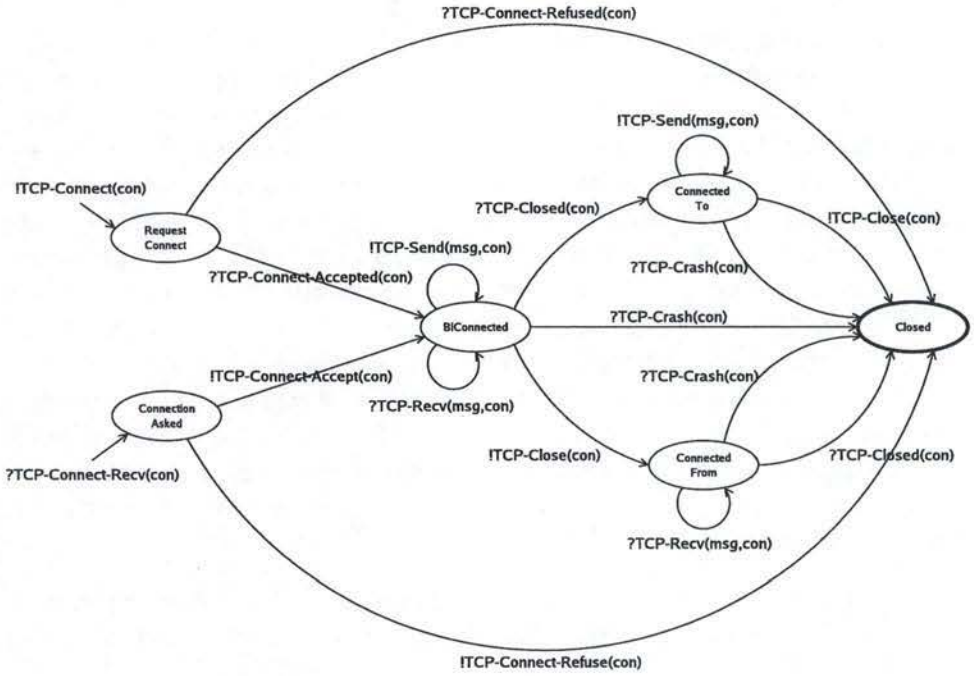


FIG. 3.8 – Machine à états d'une connexion TCP.

Dans les explications qui vont suivre, nous allons considérer deux applications A et B travaillant chacune avec leur machine à états propre.

Elle comporte les six états suivants :

`RequestConnect` :
L'application cliente A désire se connecter à l'application serveur B. Elle est dans l'attente d'une réponse.

`ConnectionAsked` :
L'application serveur B désire se connecter à A.

`BiConnected` :
Une connexion bidirectionnelle existe entre les deux applications A et B.

`ConnectedTo` :
Une connexion existe entre les deux applications A et B. La direction est unidirectionnelle de A vers B.

`ConnectedFrom` :
Cet état indique qu'une connexion existe entre les deux applications A et B.

La direction est unidirectionnelle de B vers A.

`Closed` :

État final, la connexion est terminée.

Fonctionnement d'une connexion :
Lorsqu'une application A désire ouvrir une connexion vers l'application B, B doit tout d'abord avoir défini un socket sur lequel il attend les demandes de connexions. A doit ensuite choisir un socket encore inutilisé par aucune autre application. A va déclencher l'évènement TCP-CONNECT($A \leftrightarrow B$) pour indiquer son désir de se connecter à B, A entre dans l'état `RequestConnect`. Si aucun problème réseau n'est présent, B va recevoir la demande par l'intermédiaire de l'évènement TCP-CONNECT-RECV($A \leftrightarrow B$), il peut soit accepter la connexion en émettant l'évènement TCP-CONNECT-ACCEPT($A \leftrightarrow B$), soit la refuser grâce à l'évènement TCP-CONNECT-REFUSE($A \leftrightarrow B$). S'il l'accepte, A passe dans l'état `BiConnected`, sinon il passe dans l'état final `Closed` (aucune connexion établie). Dès que B reçoit la réponse de A (TCP-CONNECT-ACCEPTED), il entre soit dans l'état `BiConnected` si elle est affirmative, sinon il entre dans l'état `Closed`. Il est à noter que tout problème, soit une panne d'une application soit une coupure réseau, se traduit par un refus de connexion (TCP-CONNECT-REFUSE).

Dans l'état `BiConnected`, les deux applications peuvent communiquer librement grâce aux deux évènements TCP-SEND(msg, $A \leftrightarrow B$) et TCP-RECV(msg, $A \leftrightarrow B$).

Si une des deux applications désire fermer la connexion, deux choix s'offrent à elle. La première solution, assez radicale, consiste à fermer la connexion dans les deux sens de manière abrupte. Le désavantage de cette solution est la perte des messages qui transitteraient sur le réseau lors de la déconnexion. La seconde solution, préférable à la précédente, consiste à fermer la communication de manière ordonnée dans un sens unique. Lorsque les deux applications ont fermé leur sens de communication, la connexion se termine. Supposons que A désire fermer la connexion vers B. A, actuellement dans l'état `BiConnected`, déclenche l'évènement TCP-CLOSE($A \rightarrow B$) et entre dans `ConnectedTo`. A ce moment A ne peut plus envoyer de messages vers B mais il est toujours capable d'en recevoir. B reçoit du réseau l'évènement TCP-CLOSED($A \rightarrow B$), il passe dans l'état `ConnectedFrom` et il ne recevra plus de message de A mais il peut toujours en envoyer. Pour passer dans l'état `Closed`, il faut que B émette l'évènement TCP-CLOSE($A \leftarrow B$). Ce dernier donnera lieu à l'évènement TCP-CLOSED($A \leftarrow B$) pour A. Dans le cas d'une déconnexion abrupte (TCP-ABRUPT-CLOSE($A \leftrightarrow B$)) ou lors d'une panne, quelque soit l'état dans lequel nous nous trouvons (`BiConnected`, `ConnectedTo` ou `ConnectedFrom`). Il donnera lieu à l'évènement TCP-CRASH($A \leftrightarrow B$) qui mène directement à l'état `Closed`.

3.4 Choix conceptuels

Dans la section précédente, nous avons défini formellement les couches transport sur lesquelles vont se baser notre implémentation. Bien que certains frameworks disposent de leurs propres modules d'accès au réseau (TCPSimple et UDPSimple dans le cas d'Appia, UTP dans le cas de Cactus), nous allons présenter dans cette section les choix d'architectures qui nous étaient offerts pour interfacer Appia et Cactus avec TCP et UDP.

3.4.1 Choix envisageables

Trois choix nous étaient offerts, chacun avec ses avantages propres et ses inconvénients. Le premier consiste à réutiliser les modules prédéfinis par les frameworks. Le second choix porte sur l'utilisation d'un module unique qui gère tout. Le troisième choix allie les deux premiers, un module qui sert d'interface avec la couche transport et un module qui implémente les fonctions plus éloignées mais nécessaires par les communications de groupe (i.e., la gestion du groupe). Ces trois choix sont présentés à la figure 3.9.

Nous allons maintenant décrire chaque solution ainsi que leurs avantages et leurs inconvénients respectifs. Pour ce faire, nous allons nous baser sur quatre critères :

Clarté : Permet de juger la lisibilité de l'architecture et la facilité de la comprendre.

Réutilisabilité : Indique si les modules implémentés sont facilement réutilisables en cas de changement de la couche transport sous-jacente.

Adaptation : Indique si les modules sont facilement adaptables face à une autre pile de protocoles.

Performance : Estime les performances liées à la solution adoptée.

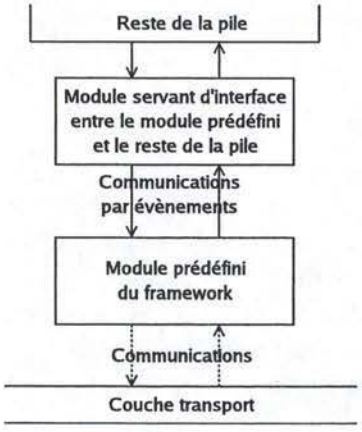
Choix 1 : Réutilisation des modules prédéfinis

Cette solution consiste à réutiliser les modules prédéfinis par le framework réel (Cactus ou Appia). La communication se base sur deux modules (Fig. 3.9(a)). Le premier est un module prédéfini par le framework concret. Le second module, de notre conception, sert à interfacer celui prédéfini avec le reste de la pile. Nous devons également ajouter dans ce module les particularités propres au service implémenté (dans notre cas, la gestion du groupe).

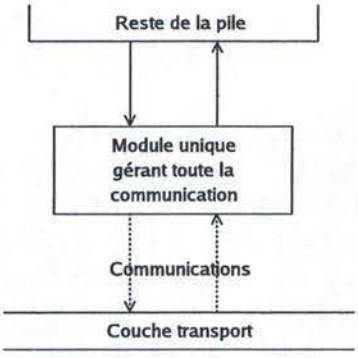
Avantages et inconvénients :

Clarté : Faible.

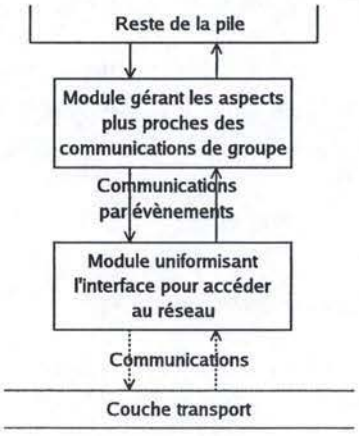
Un même module gère deux fonctions : l'interfaçage avec le module prédéfini et les particularités du service implémenté. Il est préférable d'associer une fonction unique à un module.



(a) Réutilisation des modules prédéfinis.



(b) Utilisation d'un module unique.



(c) Séparations des rôles.

FIG. 3.9 – Architecture des modules de communication.

Réutilisabilité : Importante.
Lorsque la couche transport change, le module prédéfini pour la nouvelle couche devrait présenter la même interface.
Adaptation : Faible.
Le module construit devra être revu lors du passage à une autre pile.
Performance : Faible.
Deux modules impliquent une interaction plus importante et donc une perte de performances.

Choix 2 : Utilisation d'un module unique

Cette solution consiste à implémenter un module de communication unique monolithique tel que présenté sur la figure 3.9(b).

Avantages et inconvénients :
Clarté : Faible.
Nous avons un module unique qui a deux fonctions : interfaçage avec la couche transport et implémentation des particularités de la pile.
Réutilisabilité : Faible.
Ce module devra être adapté en cas de changement de la couche transport.
Adaptation : Faible.
Ce module sera difficilement adaptable à une autre pile de protocole.
Performance : Importante.
Un seul module implique une interaction moindre et donc de meilleures performances.

Choix 3 : Séparation des rôles

Cette solution se base également sur deux modules (Fig. 3.9(c)). Le premier module consiste à standardiser les concepts relatifs à la couche transport, il permet de proposer une interface commune et bien définie pour y accéder quelque soit la couche transport considérée. Le second module, construit au-dessus du premier, implémente les concepts inhérents au service implémenté par la pile mais nécessaire à ce niveau (dans notre cas, la gestion du groupe).

Avantages et inconvénients :
Clarté : Importante.
Chaque module implémente une fonction bien spécifique. Cela facilite la compréhension de leur rôle.
Réutilisabilité : Moyenne.
En cas de modification de la couche transport, seul le module inférieur devra être modifié.

Adaptation : Moyenne.
En cas d'utilisation d'une autre pile, seul le module supérieur devra être adapté.

Performance : Faible.
Deux modules impliquent une interaction plus importante et donc une perte de performances.

3.4.2 La solution adoptée

Nous avons finalement opté pour le troisième choix (Fig. 3.9(c)), et ce pour plusieurs raisons. Nous raisonnons à ce niveau à partir d'un framework abstrait qui ne possède pas de module prédéfini. De plus Appia et Cactus possèdent des modules d'accès ayant une interface fort différente l'une de l'autre. Dès lors, nous avons préféré écarter la première solution trop dépendante du framework concret.

Dans les communications de groupe, la gestion du groupe est une notion qui apparaît également dans les modules de communication. Ce concept n'existant pas dans d'autres applications distribuées, il nous semble opportun de séparer les rôles relatifs à la communication et les rôles relatifs à la gestion du groupe. Dès lors, nous écartons le second choix.

Chapitre 4

Description des modules

Sommaire

4.1 Introduction 82
4.2 Transport 82
4.2.1 Rappels 82
4.2.2 Les objectifs 82
4.2.3 Machine à états du module Transport 84
4.2.4 Problèmes liés à l'utilisation de TCP/IP 84
4.2.5 Structures de données et fonctions 86
4.2.6 Machines à états de Transport sur TCP 89
4.2.7 Solution aux problèmes rencontrés 94
4.3 Unreliable 95
4.3.1 Rappels 95
4.3.2 Machine à états 96
4.4 Reliable Pt2Pt 97
4.4.1 Rappels 97
4.4.2 Objectifs 97
4.4.3 Structures de données et fonctions 98
4.4.4 Pseudo-Code de Reliable Pt2Pt 99
4.5 Failure Detector 100
4.5.1 Rappels 100
4.5.2 Structures de données et fonctions 100
4.5.3 Pseudo-code du Failure Detector. 101
4.6 Consensus 102
4.6.1 Rappels 102
4.6.2 Structures de données et fonctions 104
4.6.3 Pseudo-code du Consensus 106
4.7 Atomic Broadcast 109
4.7.1 Rappels 109
4.7.2 Pseudo-Code de Atomic Broadcast 109

4.1 Introduction

Dans ce chapitre, nous allons décrire de manière précise chaque module implémenté. Les pseudo-codes présentés dans ce chapitre se basent sur ceux réalisés par les membres du LSR et adaptés en cours de projet. La présentation de chaque module va s'articuler selon trois grands axes :

- Une partie théorique où nous allons rappeler le rôle du module et certaines hypothèses sur son fonctionnement.
- Une description des structures de données et des fonctions manipulées par le module.
- Le pseudo-code ou les machines à états qui décrivent le fonctionnement du module.

4.2 Transport

4.2.1 Rappels

Le module Transport gère la communication fiable point à point entre deux processus distants ou non. Il sert d'interface entre la couche transport utilisée (TCP) et le reste de la pile.

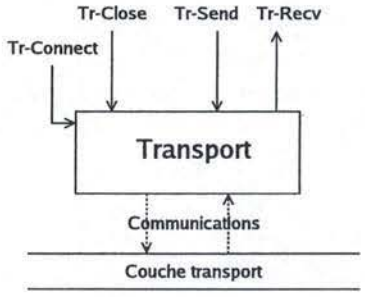


FIG. 4.1 – Interface du module Transport.

Nous allons définir les objectifs que nous nous sommes fixés pour ce module (Sect. 4.2.2). Ensuite, nous présenterons les machines à états et leur fonctionnement attendu (Sect. 4.2.3). Finalement, nous nous intéresserons aux problèmes d'implémentation soulevés par ce module lié à sa construction sur TCP/IP (Sect. 4.2.4 à 4.2.7).

4.2.2 Les objectifs

Les services fournis par ce module sont multiples :

Modèle orienté connexion : Il permet d'ouvrir des canaux de communication quasi-fiables unidirectionnels entre deux processus et de les fermer (Sect. 1.2).

Le choix de gérer un modèle orienté connexion permet d'utiliser les avantages proposés par la couche transport sous-jacente comme la transmission point à point fiable, orientée connexion, avec un contrôle de flux, ...

Succès automatique des connexions : La majeure partie des algorithmes dans les communications de groupe suppose l'existence d'un groupe de processus. Le rôle de savoir si un processus doit appartenir au groupe ou non n'est pas du ressort des modules de communication. Ce rôle est assumé par un module dédié appelé le détecteur de pannes. Dès lors, nous avons voulu que l'ouverture d'un canal de communication vers un processus soit toujours possible. Ce canal reste toujours ouvert tant que l'ordre explicite de fermeture n'a pas été reçu. L'ouverture d'un canal de communication est donc toujours possible vers un processus qui n'a pas encore été initialisé.

Par exemple, supposons deux processus A et B comme illustré sur la figure 4.2. A a été lancé et non B. Si A veut établir une connexion vers B, elle sera considérée comme établie. De ce fait, A pourra envoyer des messages vers B qui seront mis en attente. Lorsque le processus B sera lancé, A entrera en communication avec B et lui enverra tous les messages mis en attente. Ce mécanisme évite des problèmes de synchronisation au moment du lancement et de l'initialisation des piles.

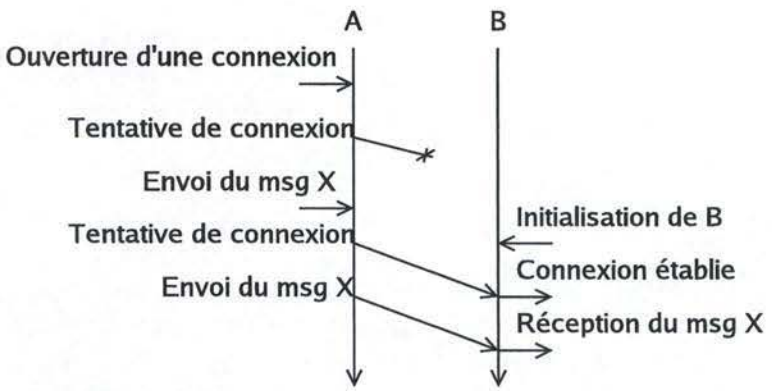
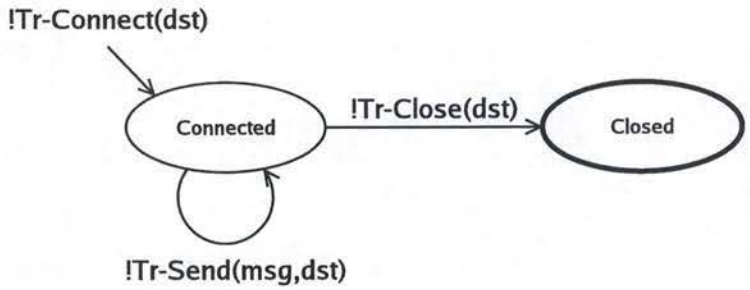


FIG. 4.2 – Connexion vers B, un processus non initialisé.

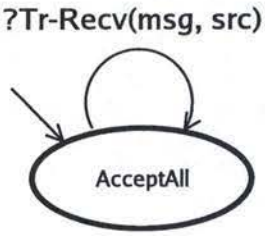
Robustesse des connexions : Un canal de communication établi doit le rester tant que l'ordre de fermeture n'a pas été reçu. Et ce, même lors de coupures réseau, lors d'une panne d'un autre processus, ... La possibilité d'envoyer des messages dans un canal de communication doit toujours rester possible. Les canaux établis doivent donc respecter la propriété de robustesse.

4.2.3 Machine à états du module Transport

La figure 4.3 présente le fonctionnement d'un canal de communication quasi-fiable unidirectionnel.



(a) Connexion à un autre processus.



(b) Réception d'une connexion d'un autre processus.

FIG. 4.3 – Machines à états du module Transport.

Dès la réception d'une demande d'ouverture d'un canal de communication unidirectionnel vers un processus B (évènement TR-CONNECT), A rentre dans l'état `Connected`. Dans cet état, A peut envoyer des messages vers B via l'évènement TR-SEND. Lors de la réception d'un TR-CLOSE, le canal est fermé et A entre dans l'état final `Closed`.

Selon ce modèle, nous pouvons recevoir à n'importe quel moment des messages issu d'un processus distant. En effet, les évènements, tels que définis en 3.2.4, ne permettent pas à un processus A de détecter l'ouverture d'une connexion transport de B vers A.

4.2.4 Problèmes liés à l'utilisation de TCP/IP

Pour implémenter ce module, nous nous sommes basés sur le protocole transport TCP. Son utilisation nous a simplifié grandement la tâche.

Pour éviter toute confusion entre le concept de connexion TCP et le concept de connexion défini par notre module, nous utiliserons deux termes distincts. Le terme connexion-transport($A \rightarrow B$) désigne le concept de connexion de notre module entre deux processus A et B. Le terme connexion-TCP($A \leftrightarrow B$) désigne une connexion TCP entre les processus A et B. Nous utiliserons également les deux formes abrégées suivantes : connexion-transport et connexion-TCP.

Bien que le concept de connexion-transport semble fort proche de celui de connexion-TCP, l'utilisation de TCP sans ajout n'est pas possible pour atteindre les objectifs fixés. De nombreux problèmes restent à résoudre.

Problème de l'initialisation : L'ouverture d'une connexion-TCP n'est possible que vers un processus qui a été préalablement initialisé. Par contre, l'ouverture d'une connexion-transport peut être établie à tout moment.

Problème des déconnexions non-désirées : Toute panne d'un processus ayant une connexion-TCP ouverte ou toute coupure dans le réseau entraîne la fermeture abrupte de celle-ci. Les connexions-transports doivent, par contre, rester ouvertes quelque soit les évènements externes. Nous voulons assurer la propriété de robustesse à nos connexions-transports.

Problème de l'identification : Lorsqu'une connexion-transport est ouverte entre deux processus, chacun doit connaître l'identifiant de l'autre processus. L'identifiant d'une connexion-TCP (**connexion**) étant différent de celui d'une connexion-transport (**pID**), il doit exister un mécanisme permettant aux deux processus d'échanger leur identifiant avant tout autre message.

Problème de la double connexion-transport : Nous ne voulons pas qu'un même processus puisse ouvrir une connexion-transport vers un processus avec lequel il en a déjà ouverte une.

Problème de la réutilisation des connexions-TCP : Supposons qu'un processus A a ouvert une connexion-transport($A \rightarrow B$) comme illustré à la figure 4.4. L'ouverture d'une telle connexion a donné lieu à l'ouverture d'une connexion-TCP($A \leftrightarrow B$). Dès lors, si le processus B désire ouvrir une connexion-transport($B \rightarrow A$), il serait intéressant de réutiliser la connexion-TCP($A \leftrightarrow B$) déjà établie. Ce procédé permet de diminuer le nombre de connexions-TCP à gérer.

Problème de la double connexion-TCP : Supposons que A décide d'ouvrir un canal de communication vers B (connexion-transport($A \rightarrow B$)). Au même moment B veut ouvrir un canal de communication de B vers A (connexion-transport($B \rightarrow A$)) comme décrit sur la figure 4.5.

Dans ce cas, A constate qu'aucune connexion-TCP n'est établie avec B et va donc en ouvrir une. B va effectuer la même constatation et en ouvrir une de B vers

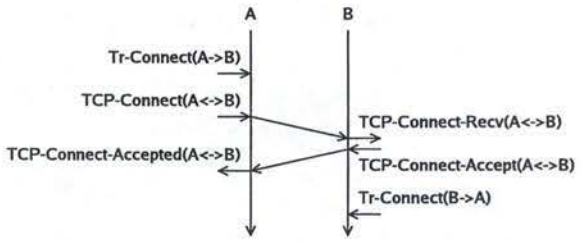


FIG. 4.4 – Exemple de réutilisation de la connexion TCP.

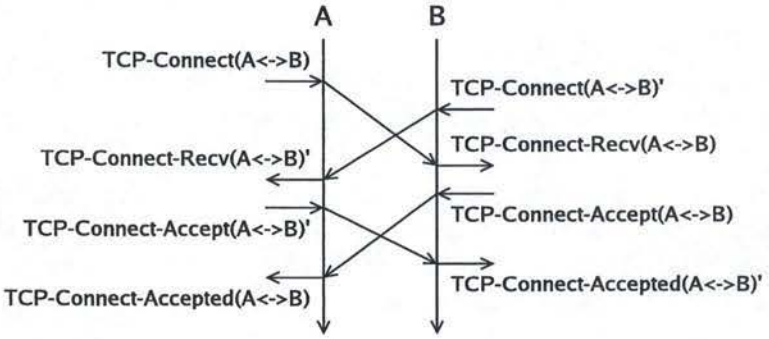


FIG. 4.5 – Problème de la double connexion.

A. Nous nous retrouvons alors avec l'établissement de deux connexions-TCP entre A et B. Cette situation doit être gérée car nous ne voulons qu'une connexion-TCP unique entre les deux processus.

4.2.5 Structures de données et fonctions

Voici les structures de données et les fonctions qui nous sont utiles pour la description du module Transport.

Structures de données

local_pid
Cette constante identifie notre processus.

SetConnections
Cette variable décrit un ensemble de connexions-TCP vers des processus distants. L'identifiant de cet ensemble est un élément de type *connexion*. Un élément de l'ensemble est un tuple de la forme :

con
Il est l'identifiant de l'ensemble. Il identifie la connexion-TCP établie ou en cours d'établissement.

distant_pid
Il identifie du processus distant avec lequel la connexion est établie.

state_TCP
Il indique l'état de la connexion-TCP proprement dite. La mise à jour de cet état est présentée dans la section suivante par la machine à états de la figure 4.6.

state_Transport
indique l'état de la connexion-transport liée à la connexion-TCP. Elle est décrite par la machine à états de la figure 4.7 à la section suivante.

buf
Il est une file FIFO qui mémorise les messages à envoyer.

Fonctions

Signature	GetCon(pid :**pID**) :**connexion**
Description	Pré : pid initialisé Post : $local_pid = (ip_1, port_1, inc_1) \wedge$ $pid = (ip_2, port_2, inc_2) \Rightarrow ((ip_1, port_1), (ip_2, port_2))$
Remarques	Cette fonction construit l'identifiant d'une connexion-TCP à partir du pID d'un processus distant et du pID de notre processus ($local_pid$).

Signature	CheckPID(pid :**pID**) :**booléen**
Description	Pré : pid initialisé Post : $SetConnections(getCon(pid)) = \perp \Rightarrow$ **true**$\vee$ $SetConnections(getCon(pid)) = (p, sTCP, sTr, buf) \Rightarrow$ $(p = pid)$
Remarques	Cette fonction vérifie si le pID du processus distant (pid) auquel nous voulons nous connecter est le processus voulu. En effet, lorsque nous nous connectons à un processus distant, nous le faisons en utilisant un socket $(ip, port)$. Or, le pID comprend trois informations : $(ip, port, inc)$. De ce fait, lorsque la connexion est établie, nous devons encore vérifier le numéro d'incarnation.

Signature	CheckConnection(pid :**pID**) :**boolean**
Description	Pré : pid initialisé Post : $SetConnections_0(getCon(pid)) = \perp \Rightarrow$ **false** $SetConnections_0(getCon(pid)) = (p, s_1, s_2, buf) \Rightarrow$ $((p \preceq pid) \wedge$!TCP-CLOSE$(p) \wedge$ $SetConnection(getCon(pid)) =$ $(pid, Connected, Connected, \emptyset) \wedge$ **true**$) \vee$ $(!(p \preceq pid) \wedge$ **false**$)$
Remarques	L'objectif de cette fonction est double. Le premier objectif est de vérifier si une connexion-TCP existe déjà vers le processus *pid*. Si une telle connexion est détectée, nous sommes dans le cas d'une double connexion-TCP. Dès lors, son second objectif est de déterminer laquelle des deux doit être fermée et il la ferme. En effet, nous ne pouvons fermer aucune des deux connexions de manière non-déterministe. Pour comprendre le problème, prenons l'exemple suivant, soit deux processus A et B. Une double connexion-TCP a été établie (connexion-TCP($A \leftrightarrow B$) et connexion-TCP($A \leftrightarrow B$)'). Si A ferme de manière non-déterministe la connexion-TCP($A \leftrightarrow B$) et que B ferme la connexion-TCP($A \leftrightarrow B$)', nous nous retrouvons avec deux connexions-transport ouvertes sans aucune connexion-TCP sous-jacente, soit un état d'erreur à éviter. Nous utilisons la relation d'ordre total définie sur les connexions (cfr. Sect. 3.2.3) pour décider celle à fermer.

Signature	addBuffer(msg :**message**, pid :**pID**) :**void**
Description	Pré : $(getCon(pid) \neq \perp) \wedge (msg\ initialis)$ Post : $Setconnections_0(pid) = (p, s_1, s_2, buf_0) \wedge$ $SetConnections = (p, sTCP, sTr, buf) \wedge$ $buf = msg :: (buf_0)$
Remarques	Cette fonction ajoute *msg* au buffer de la connexion-TCP liée au processus *pid*.

Signature	flushBuffer(pid :**pID**) :**void**
Description	Pré : $getCon(pid) \neq \perp$ Post : $SetConnections_0(pid) = (p, s_1, s_2, (m_1, \ldots, m_n)) \wedge$ $SetConnection(pid) = (p, s_1, s_2, \emptyset) \wedge$!TCP-SEND$(m_1) \wedge \ldots \wedge$!TCP-SEND$(m_n)$
Remarques	Cette fonction envoie tous les messages mis en attente dans le buffer de la connexion-TCP liée au pID *pid*.

Pseudo-évènements

Les trois évènements présentés ci-dessous n'existent pas mais ils sont utilisés afin de synchroniser les machine à états qui seront présentées sur les figures 4.6 et 4.7.

Notify-Open(pid : **pID**)
Indique la réussite de l'ouverture de la connexion-TCP liée au processus *pid*.

Notify-Close(pid : **pID**)
Indique la fermeture de la connexion-TCP liée au processus *pid*.

Notify-Recv(msg : **message**, pid : **pID**)
Indique la réception du message msg sur la connexion-TCP liée au processus *pid*.

4.2.6 Machines à états de Transport sur TCP

Par souci de clarté, nous allons décrire le module Transport à l'aide de deux machines à états.

Les sockets robustes[32]

Pour régler le problème des déconnexions non-désirées et ainsi ajouter aux connexions-TCP la propriété de robustesse, nous avons utiliser les sockets robustes. Ceux-ci remplacent les sockets standards. Une connexion-TCP peut être coupée dans les deux cas suivant : suite à la panne d'un processus ou lors d'une coupure réseau. Les Reliable Socket permettent d'éliminer ces deux cas.

Machines à états

Les deux machines à états des figures 4.6 et 4.7 présentent le fonctionnement d'une connexion-transport basé sur le mécanisme des connexions-TCP décrit à la section 3.3.2. La première machine à états est responsable de gérer une connexion-TCP unique. La seconde, fortement liée à la première, modélise la connexion-transport construite au-dessus.

Machine à états d'une connexion-TCP

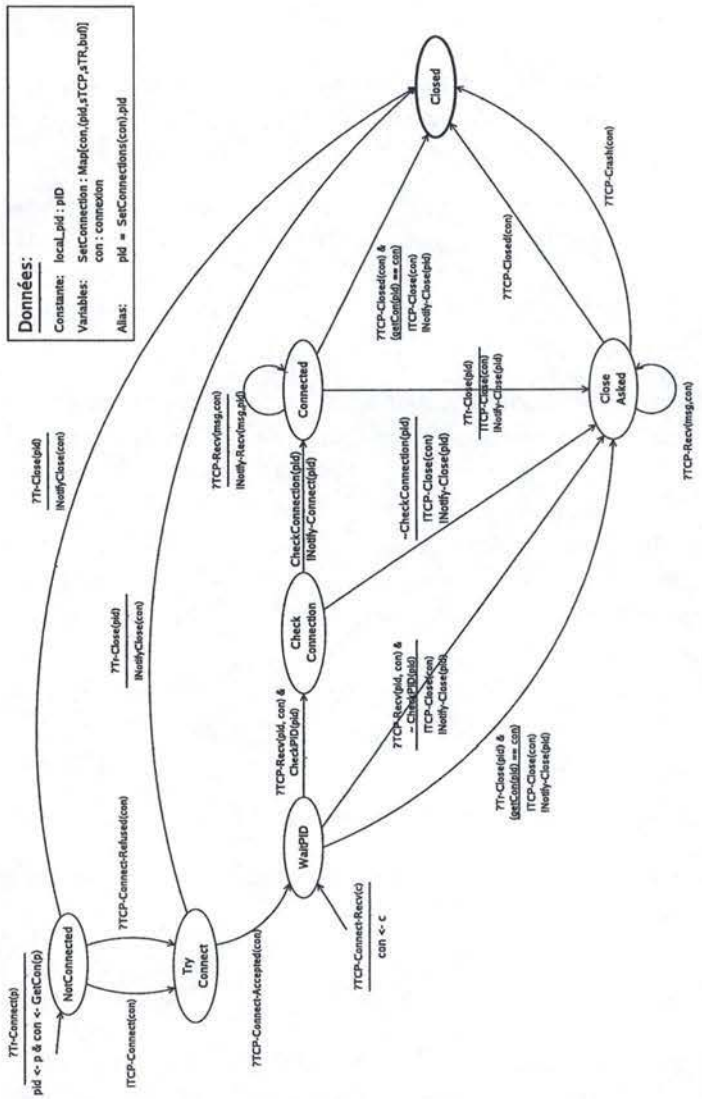


FIG. 4.6 – Machine à états d'une connexion-TCP.

La figure 4.6 représente la machine à états d'une connexion-TCP. Ses objectifs sont multiples. Le principal est la gestion d'une connexion-TCP depuis l'ouverture de celle-ci jusqu'à sa clôture.

Elle avertit également la seconde machine à états qui gère les relations entre les connexions-transport et les connexions-TCP (Fig. 4.7) des évènements importants :

- l'ouverture réussie d'une connexion-TCP (NOTIFY-OPEN).
- la fermeture d'une connexion-TCP (NOTIFY-CLOSE).
- la réception d'un message (NOTIFY-RECV).

Elle effectue finalement quelques vérifications sur le processus distant. Elle

vérife si une double connexion-TCP n'a pas été ouverte et si le processus distant est le bon.

Description des états :

`NotConnected` :
Aucune connexion-TCP n'a encore pu être établie.

`TryConnect` :
Une demande de connexion-TCP est en cours. Nous attendons la réponse du processus distant.

`WaitPID` :
La connexion-TCP correspondante est établie. Nous attendons l'identifiant du processus distant.

`CheckConnection` :
La connexion-TCP correspondante est établie. Nous vérifions qu'une double connexion-TCP n'a pas déjà été établie.

`Connected` :
La connexion-TCP est établie. Le processus distant est correctement identifié.

`CloseAsked` :
La connexion-TCP est en cours de fermeture.

`Closed` :
La connexion-TCP est fermée.

Description du fonctionnement : Pour décrire l'ouverture d'une connexion-TCP, deux cas de figures peuvent se présenter : soit nous tentons d'établir une connexion suite à la demande d'ouverture d'une connexion-transport (évènement TR-CONNECT), soit elle est établie suite à la réception d'une demande de connexion-TCP d'un autre processus (évènement TCP-CONNECT-RECV).

Considérons le premier cas. La demande d'ouverture d'une connexion-transport donne lieu au passage dans l'état initial `NotConnected`. Aucune connexion-TCP n'ayant encore été établie, notre processus tente d'en créer une (état `TryConnect`). Il réitère la procédure jusqu'à la réussite de la connexion (passage dans l'état `WaitPID`). Ensuite, il envoie son pID (*local_pid*) à l'autre processus et attend l'identification du processus distant. Dès qu'il le reçoit, la fonction `CheckPID` renvoie *vrai* s'il est bien identique au pID de la connexion-transport que nous voulons ouvrir. La vérification du pID est ici nécessaire car nous sommes dans le modèle no-recovery. Un processus qui est tombé en panne, ne peut pas être réparé ni relancé. Si nous voulons lancer à nouveau le processus sur la machine tombée en panne, celui-ci aura obligatoirement un identifiant différent. Rappelons que le pID est défini par le tuple (ip, port, incarnation). Par contre, la connexion-TCP établie est identifiée chez le processus distant par le couple (ip, port). Ainsi, lorsque la connexion-TCP est établie, il nous faut encore vérifier que le numéro d'incarnation soit correct. Dans le cas contraire, la connexion-TCP sera rejetée (passage dans

l'état `CloseAsked`). S'il est correct, nous passons dans l'état `CheckConnection`.

Considérons le deuxième cas, celui de la réception d'une demande d'ouverture d'une connexion-TCP (évènement TCP-CONNECT-RECV). Nous acceptons immédiatement la demande et nous envoyons son identification (*local_pid*). Nous passons dans l'état `WaitPID` et nous attendons l'identification de l'autre processus. Cette attente est nécessaire car la connexion-TCP ne nous donne qu'une partie du pID du processus distant $(ip, port)$. Dès que nous la recevons, nous passons dans l'état `Connected`. Il est à noter que la fonction CheckPID renvoie *vrai* dans ce cas-ci car *distant_pid* n'a pas encore été initialisé.

Une fois dans l'état `CheckConnection`, nous devons vérifier qu'une connexion-TCP n'a pas déjà été établie, par exemple dans le cas d'une double connexion-TCP. C'est le rôle de la fonction CheckConnection de vérifier cette condition. Si une connexion-TCP a déjà été établie, une des deux sera fermée (cfr. la description de CheckConnection). Si cette fonction renvoie *vrai* la connexion est établie et nous passons dans l'état `Connected`, sinon nous passons dans l'état `ClosedAsked`.

Dans l'état `Connected`, tous les messages reçus du réseau (évènement TCP-RECV) sont envoyés à l'automate gérant les connexions-transport via l'événement NOTIFY-RECV.

Concernant la fermeture d'une connexion-transport par l'évènement TCP-CLOSE, deux cas sont à considérer : le cas où une connexion-TCP n'a pas encore été établie (état `NotConnected` et `TryConnect`) et le cas où elle a été établie (état `WaitPID`, `Connected` et `CloseAsked`). Dans le premier cas, il suffit de passer dans l'état `Closed` et d'avertir l'autre machine la fermeture de la connexion-TCP (évènement NOTIFY-CLOSE). Dans le second cas, nous allons fermer la connexion-TCP progressivement. Nous déclenchons d'abord l'évènement TCP-CLOSE et passons dans l'état `CloseAsked`. Ensuite, nous attendons un TCP-CLOSED indiquant la fermeture de la connexion-TCP dans les deux sens. Enfin nous arrivons dans l'état final `Closed`.

Machine à état d'une connexion-transport

L'objectif de cette machine à état est de gérer la connexion-transport liée à une connexion-TCP sous-jacente. En effet, plusieurs cas peuvent se présenter. Le premier est celui où nous avons ouvert une connexion-transport vers un processus distant. Le second est celui où un processus distant a ouvert une connexion-transport vers notre pile. Dans ce cas, une connexion-TCP est établie mais aucune connexion-transport ne s'y rapporte. Le troisième cas est celui où les deux processus ont ouverts une connexion-transport l'un vers l'autre.

Description des états :

`NotConnected`

Une connexion-transport est établie mais aucune connexion-TCP n'a encore pu être établie.

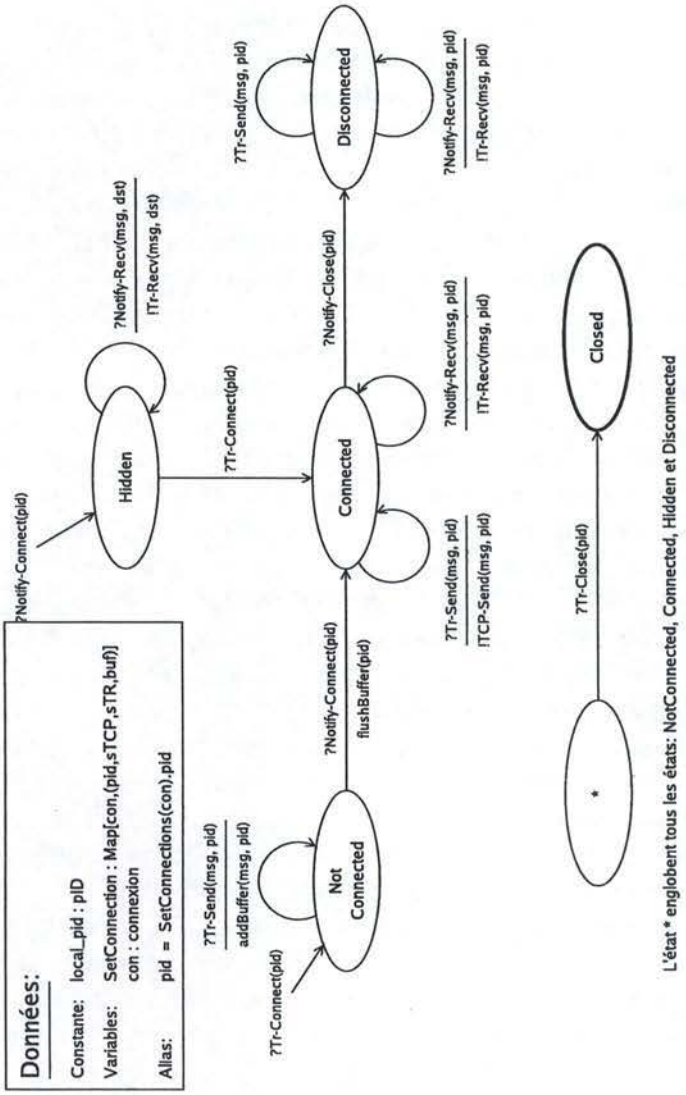


FIG. 4.7 – Machine à états d'une connexion-transport.

`Hidden`
Une connexion-TCP est établie. Aucune connexion-transport n'y correspond.

`Connected`
La connexion-transport et la connexion-TCP correspondante ont été établies.

`Disconnected`
La connexion-TCP est fermée mais la connexion-transport reste établie.

`Closed`
La connexion-transport et la connexion-TCP correspondante sont fermées.

Description du fonctionnement : Cette machine à états décrit les interactions entre une connexion-transport et la connexion-TCP correspondante. Deux cas de figures sont également possibles : soit nous voulons nous connecter à un autre processus, soit nous recevons une notification de connexion pour un processus pour lequel aucune machine à états n'a été initialisée. Ce dernier cas correspond à la réception d'une demande d'une connexion-TCP venant du processus distant.

Dans le premier cas, la demande de connexion-transport (TR-CONNECT) nous fait entrer dans l'état `NotConnected` : une connexion-transport est établie mais aucune connexion-TCP sous-jacente ne l'est. Tous les messages que nous désirons envoyer sont mis en attente dans le buffer associé à cette connexion. Lorsque la connexion-TCP est établie (évènement NOTIFY-OPEN), nous passons dans l'état `Connected` et nous envoyons tous les messages mis en attente dans le buffer. Dans le second cas, nous passons dans l'état `Hidden` : celui où un processus a ouvert une connexion-transport vers notre processus mais pas inversément. Nous pouvons donc recevoir des messages, mais pas en envoyer. Si nous voulons ouvrir une connexion vers le processus distant (évènement TR-CONNECT), il nous suffit de passer dans l'état `Connected` et ainsi de profiter de la connexion déjà établie.

Concernant la fermeture de la connexion, nous considérons deux cas : soit nous recevons un TR-CLOSE indiquant la fermeture de la connexion-transport vers notre processus, soit nous recevons un NOTIFY-CLOSE.

Dans le premier cas de figure, si nous sommes dans l'état `NotConnected`, nous passons dans l'état `Closed` (notons que tous les messages placés dans le buffer sont perdus[1]). Si nous sommes dans l'état `Connected`, nous passons dans l'état `Closed` et la connexion-TCP liée sera fermée. Si nous sommes dans l'état `Disconnected`, nous passons dans l'état `Closed`.

Dans le deuxième cas de figure, si nous sommes dans l'état `Hidden`, l'autre processus a fermé sa connexion-transport et nous pouvons passer dans l'état `Closed`. Si nous sommes dans l'état `Connected`, le processus distant vient de fermer sa connexion-transport vers notre processus. Il a également fermé la connexion-TCP, nous passons dans l'état `Disconnected`[2].

4.2.7 Solution aux problèmes rencontrés

Pour conclure cette section, nous allons reprendre les problèmes posés ci-dessus (Sect. 4.2.4) et montrer comment nos deux machines à états réussissent à les résoudre.

Problème de l'initialisation : Au niveau de la première machine à états, nous essayons d'établir une connexion-TCP juqu'à sa réussite (états `NotConnected` et `TryConnect`). Lorsqu'elle réussit, une notification est envoyée à la seconde

[1] Ce comportement nous éloigne quelque peu de la définition d'un canal de communication quasi-fiable.

[2] Dans cet état, les messages envoyés sur la connexion-transport sont jetés. Ce cas ne correspond donc pas au canaux quasi-fiables mais ne posait pas de problème dans le cas de notre implémentation.

machine à états (évènement NOTIFY-OPEN). Au niveau de la seconde machine, tant que la connexion n'a pas réussi, nous restons dans l'état `NotConnected` et les messages sont mis en attente dans un buffer. Lorsqu'elle réussit (évènement NOTIFY-OPEN), nous passons dans l'état `Connected` et tous les messages mis en attente sont envoyés. De cette manière, aucune perte de message n'a eu lieu.

Problème des déconnexions non-désirées : Ce problème est dû au manque de robustesse des connexions-TCP. En effet, en cas de panne, la connexion-TCP se termine de manière abrupte suite à un timeout. L'utilisation des sockets robustes permet de régler ce problème.

Problème de l'identification : L'ajout de l'état `WaitPID` dans la première machine à états permet de résoudre ce problème. Lorsque nous y entrons suite à la réussite de la connexion-TCP, nous envoyons comme premier message sur celle-ci notre identifiant (*local_pid*). Lorsque nous recevons l'identification de l'autre processus, nous passons dans l'état suivant (`CheckConnection`), ce qui est un pas supplémentaire vers le passage dans l'état `Connected`.

Problème de la double connexion-transport : Dans cette implémentation, une telle demande sera ignorée par les deux machines à états.

Problème de réutilisation des connexions-TCP : Ce problème est géré par l'introduction de l'état `Hidden` dans la seconde machine à états. Nous entrons dans cet état lorsqu'une connexion-TCP vers un processus A a été établie. Tandis que la seconde machine à états de ce processus ($SetConnections(con).stateTr$) n'a pas encore été initialisée. Nous sommes dans ce cas lorsqu'aucune connexion-transport ne correspond à la connexion-TCP établie. La demande d'une connexion-transport nous fait passer dans l'état `Connected` sans création d'une nouvelle connexion-TCP. Nous utilisons donc bien la connexion-TCP déjà établie.

Problème des doubles connexions-TCP : Ce problème est réglé grâce à l'introduction de l'état `CheckConnection` et de la fonction homonyme. Par sa spécification, cette fonction permet de détecter les doubles connexions, d'en sélectionner une et de la fermer.

4.3 Unreliable

4.3.1 Rappels

Le module Unreliable gère la communication non-fiable point à point entre deux processus. Ce module est utilisé en lieu et place du module Transport lorsque

les besoins de communication sont simples, sans garantie comme par exemple réaliser de simples *ping*, ...Nous allons définir ce module au-dessus de la couche transport UDP (cfr. section 3.3.1).

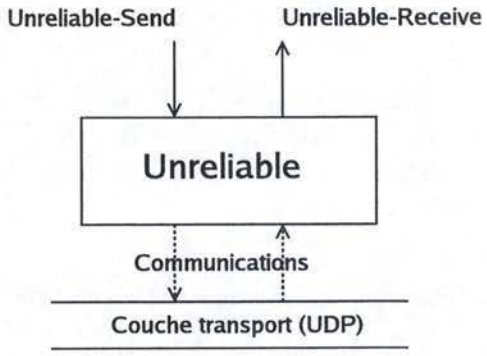


FIG. 4.8 – Interface du module Unreliable.

4.3.2 Machine à états

La machine à états de la figure 4.9 explique le fonctionnement de ce module. Le fonctionnement de ce module est trivial. Un évènement UNRELIABLE-SEND

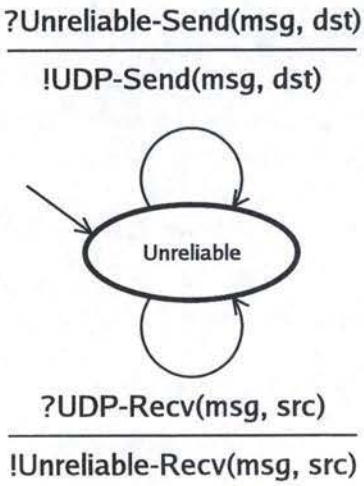


FIG. 4.9 – Machine à états du module Unreliable.

donne lieu à l'envoi d'un message UDP (évènement UDP-SEND) avec les mêmes paramètres. La réception d'un évènement UDP-RECV donne lieu au déclenchement d'un évènement UNRELIABLE-RECV avec les mêmes paramètres.

4.4 Reliable Pt2Pt

4.4.1 Rappels

Le but du module Reliable Pt2Pt, implémenté au-dessus du module Transport, est de gérer les concepts relatifs aux particularités des communications de groupe : la gestion d'un groupe de processus.

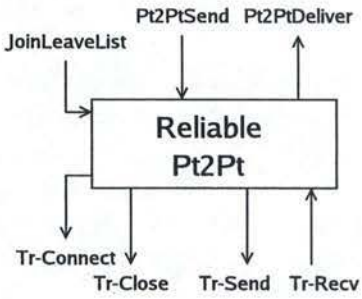


FIG. 4.10 – Interface du module Reliable Pt2Pt.

4.4.2 Objectifs

Dû à la définition du groupe de processus dans les communications de groupe, un processus A peut envoyer un message à un autre processus B, si et seulement si A et B appartiennent au groupe. De la même manière, ce processus A ne peut délivrer un message issu de B que dans les mêmes conditions.

Le premier cas ne devrait manifestement jamais se produire car c'est le processus lui-même qui décide quels processus il connaît. Mais nous ne pouvons par contre rien dire concernant les messages reçus. En effet, par la définition de la couche transport, il n'existe pour le moment aucun moyen de savoir si un message reçu vient d'un processus du groupe ou pas. Un processus A peut avoir été présenté à un processus B, alors que B n'a pas encore été présenté à A. B peut donc envoyer des messages à A, alors que ce dernier ne connaît pas B.

Le module Reliable Pt2Pt aura donc une liste de processus connus, et une liste de processus masqués. A chaque processus masqué sera associé un buffer chargé de stocker les messages que ce processus nous aura envoyé[3]. Tous les messages contenus dans un buffer sera évidemment délivré quand le processus sera connu.

Remarquons une dernière particularité du module Reliable Pt2Pt, la gestion de deux types de messages : les messages normaux, et les messages *promisc*. Les messages normaux se comportent comme nous l'avons décrit précédemment, alors que si un message *promisc* devait aller dans un buffer parce que le processus l'ayant

[3]Ce buffer peut exploser. Aucun mécanisme n'a été prévu pour gérer ce problème. Nous ne pouvons pas limiter la taille de ce dernier et jeter les messages en trop car le module nous délivrant ce message est fiable et ne dispose pas d'un moyen pour demander à l'expéditeur de renvoyer son message ultérieurement.

émis est masqué, il aura pour effet de faire passer le processus émetteur dans le groupe. Les messages du buffer sont alors délivrés ainsi que ce message *promisc*.

4.4.3 Structures de données et fonctions

Structures de données

- *Initiliazed* : Booléen utilisé afin d'éviter que le protocole ne soit initialisé deux fois.
- *Known* : Ensemble de processus qui représente le groupe de processus. Chaque élément de cet ensemble est un singleton {p :**pID**}, l'identifiant du processus.
- *Masked* : Ensemble de processus inconnus (ce processus n'appartient pas encore au groupe) pour lesquels on a déjà reçu un(des) message(s). Chaque élément de cet ensemble est identifié par un **pID**, et contient une liste de messages masqués.

Fonctions

Signature	joinKnown(New :**Set[pID]**)
Description	Précondition : $Known \cap New = \emptyset$ Ajoute les processus de *New* dans *Known* et ouvre une connexion-transport vers chacun de ces processus.
Remarques	A la fin de cette fonction, il peut exister des processus connus qui soient également masqués. Il faudra exécuter un flush(p) pour chaque processus dans cette situation.

Signature	leaveKnown(Left :**Set[pID]**)
Description	Précondition : $Left \subseteq Known$ Retire de *Known* les processus de *Left* (ferme les connexions transport vers les processus de *Left*).
Remarques	Nous ne pouvons rien dire concernant les connexions-TCP car il se peut qu'elles soient déjà fermées, ou que le processus avec lequel nous sommes connecté nous connaisse toujours.

Signature	put(p :**pID** , m :**message**)
Description	Masque le message *m* provenant du processus identifié par *p*.
Remarques	Cette fonction ajoute *p* à *Masked* s'il n'y est pas déjà.

Signature	flush(P :**Set[pID]**)
Description	Délivre tous les messages masqués pour tous les processus, dont l'identifiant est contenu dans *P* et dans *Masked* et les enlève de *Masked*.
Remarques	Si il existe des processus appartenant à *P* qui ne se trouvaient pas dans *Masked*, rien n'est fait pour ces processus.

4.4.4 Pseudo-Code de Reliable Pt2Pt

Le pseudo-code ci-dessous représente les traitements à effectuer lors de la réception des évènements afin de garantir les propriétés énoncées précédemment.

```
Upon initialize
  assert(¬ Initialized) //Il est interdit d'initialiser deux fois le module.
  Initialized ← true
  Known ← ∅
  Masked ← ∅

Upon Pt2PtSend(m , p , promisc)
  assert(Known(p) ≠ ⊥) //Il est interdit d'envoyer un message à un proccessus inconnu.
  if(promisc)
    trigger Tr-Send(m::promisc , p)
  else
    trigger Tr-Send(m::normal , p)
  endif

Upon Tr-Recv(m::type , p)
  switch(type)
    promisc:
      if(Known(p) = ⊥) //Je connais p
        //Include p in set Known
        Known(p) ← p
        trigger Pt2PtDeliver(m , p)
        flush({p})
      else //Je ne connais pas p
        trigger Pt2PtDeliver(m , p)
      endif
      break
    normal:
      if(Known(p) ≠ ⊥) //Je connais p
         trigger Pt2PtDeliver(m , p)
      else //Je ne connais pas p
         put(p , m::type)
      endif
      break
  endswitch
```

```
Upon JoinLeaveList(Left , New)
  assert(Left ∩ New = Known ∩ New = ∅)
  Left ← Left ∩ Known //Car nous autorisons Left ⊈ Known
  joinKnown(New)
  flush(New)
  leaveKnown(Left)
```

4.5 Failure Detector

4.5.1 Rappels

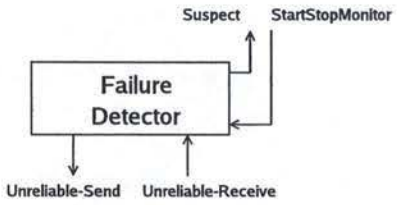


FIG. 4.11 – Interface du module Failure Detector.

Comme nous l'avons déjà expliqué dans la section 1.3.2, un détecteur de pannes de type $\diamond\mathcal{S}$ n'est pas implémentable dans un réseau asynchrone, mais nous pouvons trouver une implémentation se rapprochant le plus possible de sa spécification en choisissant judicieusement les timeouts.

Nous avons choisi de construire un détecteur de type ping (requête/réponse) au lieu d'un heartbeat (envoi périodique de messages), par simplicité.

Gestion des timers

Pour construire le module Failure Detector, nous avons besoin de timers périodiques. Autrement dit, un mécanisme qui nous fait signe toutes les X millisecondes. Ceci nous permet de déterminer l'instant auquel nous devons effectuer un ping.

Ceux-ci nous sont fournis par le framework abstrait au travers des évènements PERIODIC-TIMER (Sect. 2.3.2 (p. 44)).

4.5.2 Structures de données et fonctions

Structures de données

- *Monitored* : Ensemble de processus surveillés par le détecteur de pannes. Chaque élément de cet ensemble est un singleton {p :**pID**} identifiant un processus.

- *Suspects* : Ensemble de processus suspectés par le détecteur de pannes. Chaque élément de cet ensemble est un singleton {p :**pID**} identifiant un processus.
- *timeOut* : Timeout après lequel envoyer un ping (en millisecondes).
- *process* : Notre propre identifiant.
- *nbTimeOut* : nombre de pings restés sans réponse par un processus surveillé.
- *n* : Nombre maximum de ping pouvant rester sans réponse avant de suspecter un processus

Fonctions

Aucune fonction particulière n'est nécessaire pour ce module.

4.5.3 Pseudo-code du Failure Detector.

Nous avons construit ce module dans l'optique de générer le moins de trafic possible. C'est pourquoi, nous utilisons tous les messages (destinés au Failure Detector) comme signes de vie. Généralement, un processus en surveillant un autre sera également surveillé par ce dernier. Fonctionnant sur le principe requête/réponse, les requêtes, au même titre que les réponses peuvent être considérées comme des signes de vie. Le timer lié à l'envoi d'une nouvelle requête sera donc remis à zéro si nous recevons une requête ou une réponse.

```
Upon initialize (p, t, max)
  process ← p
  timeOut ← t
  n ← max

Upon StartStopMonitor (Start, Stop)
  assert(Stop ⊆ Monitored)
  assert(Start ∩ Stop = ∅)

  forall p ∈ Stop
    trigger periodic-timer(p, -1)
    Monitored(p) ← ⊥
    Suspects(p) ← ⊥
    nbTimeOut(p) ← ⊥
  endforall

  trigger Suspect(Suspects)

  forall p ∈ Start
    trigger Unreliable-Send(process::true::p, p)
    trigger periodic-timer(p, timeOut)
    Monitored(p) ← p
    nbTimeOut(p) ← 0
  endforall
```

```
Upon PERIODIC-TIMER(pid, p)
  trigger UNRELIABLE-SEND(process::true::p, p)
  nbTimeOut(p) ← nbTimeOut(p) + 1
  if(nbTimeOut(p) ≥ n)
    nbTimeOut(p) ← 0
    if(p ∉ Suspects)
      Suspects(p) ← p
      trigger SUSPECT(Suspects)
    endif
    trigger UNRELIABLE-SEND(process::true::p, p)
  endif

Upon UNRELIABLE-RECV (src::original::dest, dest)
  assert(dest = process)
  if(original)
    //C'est un ping,Je réponds pong
    trigger UNRELIABLE-SEND (process::false::src, src)
  endif
  if(src ∈ Monitored)
    //Je surveille src
    if(src ∈ Suspects)
      //J'arrête de le suspecter
      Suspects(src) ← ⊥
      trigger SUSPECT(Suspects)
    endif
    trigger PERIODIC-TIMER(src, 0)
    nbTimeOut(src) ← 0
  endif
```

4.6 Consensus

4.6.1 Rappels

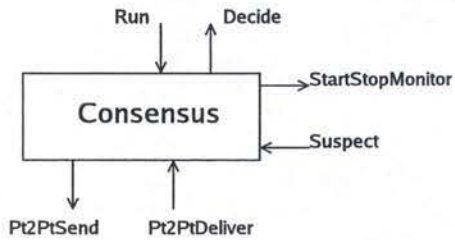


FIG. 4.12 – Interface du module Consensus.

Nous allons maintenant décrire de manière formelle l'implémentation du module Consensus. Celle-ci se basera sur l'algorithme de Chandra et Toueg présenté dans la section 1.3.2.

Ce module est construit au-dessus des modules Reliable Pt2Pt et Failure Detector. Par simplicité, nous avons choisi d'incorporer une version très simple de Reliable Broadcast au sein du module Consensus.

Nous allons réécrire ici l'algorithme résolvant le consensus car la description donnée à la section 1.3.2 est trop informelle et laisse beaucoup de détails d'implémentation dans l'ombre. Nous allons montrer comment gérer des exécutions simultanées du consensus, comment optimiser la première ronde et finalement comment nettoyer la mémoire des structures de données inutilisées (garbage collection).

Afin de pouvoir exécuter plusieurs consensus simultanément, nous avons besoin de conserver les états de ces différentes exécutions et de leur affecter un identifiant pour pouvoir les distinguer. Tous ces états seront conservés dans un ensemble (*S*) composé de tuples de la forme (*k, round, phase, estimate, lastUpdated, nbAck, nbNack, numEstimate, Group*) qui seront détaillés dans le point suivant.

Notons que nous avons laissé le consensus attribuer lui-même l'identifiant des exécutions. Ceci pourrait poser problème si plusieurs modules utilisent celui-ci. En effet, si les exécutions ne commencent pas dans le même ordre sur toutes les compositions participantes, les décisions renvoyées pourraient n'avoir aucun sens pour certains participants. L'attribution de l'identifiant devrait donc être le résultat d'une opération prenant en compte le module à l'origine de l'exécution de ce consensus. Bien que cela soit envisageable dans le framework abstrait, en supposant qu'il existe une primitive *sender(e)* permettant de déterminer la source d'un évènement, nous devons garder en tête que ce module est destiné a être implémenté dans le plus grand nombre de frameworks concrets possible. Nous laisseront donc ce problème en suspens, tout en notant qu'il suffirait de modifier l'évènement RUN pour le solutionner[4].

En plus des informations contenues dans *S*, nous avons besoin de trois variables communes à tous les consensus. La première étant l'ensemble des processus suspectés (*Suspected*) fourni par le détecteur de pannes. La seconde est un ensemble (*Already*) permettant de gérer le Reliable Broadcast effectué pour diffuser la décision. Il est composé de couples (*nummer*, *nbAck*) où *nummer* est l'identifiant du consensus diffusant sa décision pour lequel il reste *nbAck* « confirmations » à recevoir avant que le Reliable Broadcast ne soit complètement terminé. La troisième (*finishedUpTo*) nous indique le numéro du consensus en dessous duquel nous sommes certains que tous les consensus soient finis[5].

[4] Introduction d'un paramètre permettant de distinguer de manière non équivoque le module émetteur.

[5] Nous ne recevrons plus de messages envoyés par Reliable Broadcast en relation avec ce consensus.

4.6.2 Structures de données et fonctions

Structures de données

- *Initilized* : Booléen est utilisé pour éviter que le protocole ne soit initialisé deux fois.
- *S* : Ensemble d'états (State) de consensus qui ont la forme :
  - *k* : Numéro du consensus (permet l'exécution simultanée de plusieurs consensus)
  - *round* : Numéro de la ronde courante.
    -1 signifie que le consensus n'a pas encore commencé.
    ∞ signifie que le consensus est terminé.
  - *phase* : Numéro de la phase courante. Peut être 2 ou 4 (-1 lors de l'initialisation)[6].
  - *estimate* : Estimation courante de la valeur du consensus.
  - *lastUpdated* : Ronde de la dernière modification de l'état du consensus.
  - *nbAck* : Nombre d'acquits positifs reçus.
  - *nbNack* : Nombre d'acquits négatifs reçus.
  - *numEstimate* : Nombre d'estimations reçues. N'a de sens que si le processus est le coordinateur pour cette ronde et est dans la phase 2.
  - *Group* : Liste triée des identifiants des processus participant à ce consensus. Le premier élément se trouvant à l'indice 0.
- *next* : Numéro du prochain consensus qui sera exécuté.
- *process* : Identifiant du processus.
- *Suspects* : Ensemble d'identifiants de processus suspectés par le détecteur de pannes (Failure Detector).
- *Already* : Ensemble de la forme :
  - number : Numéro du consensus (identifiant).
  - nbAck : Nombre d'acquits restants à recevoir.
- *finishedUpTo* : Numéro de consensus en dessous duquel nous sommes sûrs que tous les consensus soient finis.

Fonctions

Signature	getCoord(s :**State**) :**pID**
Description	Retourne l'identifiant du coordinateur en fonction de l'état du consensus. Il est donné par la formule : $S.Group(S.round$ **modulo** $card(S.Group))$
Remarques	/

[6]Les phases 1 et 3 ne seront jamais mentionnées explicitement.

Signature	getLimit(g :**Group**) :**int**
Description	Retourne le nombre minimal de processus à atteindre si l'on veut une majorité.
Remarques	/

Signature	getOthers(g :**Group**) :**Group**
Description	Retourne une liste triée égale à *g* d'où l'on a retiré *process* (nous-même).
Remarques	/

Signature	send (type :**int** , nummer :**int** , round :**int** , content :**message** , dest :**Set[pID]**)
Description	Envoie le message à tous les processus dans *dest* sous la forme (*type : :nummer : :round : :content*) où – *type* est le type du message – *nummer* est l'identifiant du consensus correspondant – *round* est le numéro de la ronde à laquelle ce message a été envoyé – *content* est le contenu du message
Remarques	Le champ *type* peut prendre les valeurs : – *RBCAST* : pour un Reliable Broadcast – *PROPOSE* : pour une proposition – *ESTIMATE* : pour une estimation – *ACK* : pour un acquit En fonction de son type, le contenu d'un message peut prendre différentes formes : – *RBCAST* : contient une décision – *PROPOSE* : contient une proposition – *ESTIMATE* : contient une estimation (*content.estimate*) et la ronde de sa plus récente mise à jour (*content.lastUpdate*) – *ACK* : contient un booléen indiquant qu'il représente un acquit positif (**true**) ou négatif (**false**).

Signature	incRound(s :**State**)
Description	Passe à la ronde suivante.
Remarques	Pseudo-code avec optimisation de la première ronde :

```
incRound(s)
```

```
s.round ← s.round + 1
if(process ≠ getCoord(s))
  //Je ne suis pas le coordinateur
  if(s.round > 0)
    //J'envoie mon estimation au coordinateur
    send(ESTIMATE, s.k, s.round, (s.estimate , s.lastUpdated),
          {getCoord(s)})
  endif
  if(s.coord ∈ Suspects)
    //Le coordinateur est suspecté
    send(ACK, s.k, s.round, false, {getCoord(s)})
    incround(s)
  endif
else
  //Je suis le coordinateur
  if(s.round > 0)
    //J'attends les estimations des autres processus
    s.nbAck ← 0
    s.nbNack ← 0
    s.numEstimate ← 0
    s.phase ← 2
  else
    send(PROPOSE, s.k, s.round, s.estimate, getOthers(s.Group))
    s.phase ← 4
  endif
endif
```

4.6.3 Pseudo-code du Consensus

```
Upon INITIALIZE(p)
  assert(¬Initialized)
  Initialized ← true
  S ← ∅
  next ← 0
  process ← p
  Susptects ← ∅
  Already ← ∅
  finishedUpTo ← -1

Upon RUN(m, P)
  //Je dois faire partie du groupe
  assert(process ∈ P)
  S(next) ← {(next, -1, -1, m, -1, 0, 0, 0, P)}
  next ← next + 1
  //Je commence le consensus
  incRound(S(next-1))
```

```
Upon PT2PTDELIVER(type::k::round::content, src)
    when (k ≤ finishedUpTo
          ∨ (S(k) ≠ ⊥
               ∧ round ≤ S(k).round))
  if (k ≤ finishedUpTo ∨ round < S(k).round)
    //Le consensus est déjà terminé OU c'est un vieux message
    return
  endif
  switch(type)
    RBCAST :
      if (Already(k) = ⊥)
        //C'est la première fois que je reçois ce message
        Already(k) ← card(S(k).Group) - 1   //Ceci n'est pas optimal
        send(type, k, round, content, dest)
        //Je décide.
        s.round ← ∞
        //Je délivre la décision
        trigger DECIDE(s.estimate)
        //Garbage collection
        while S(finishedUpTo + 1).round < ∞
          finishedUpTo ← finishedUpTo + 1
          S(finishedUpTo) ← ⊥
        endwhile
      else
        //J'ai déjà reçu ce message
        Already(k) ← Already(k) - 1
        if (Already(k) ≤ 0)
          //J'ai reçu le message au moins card(dest) fois
          Already(k) ← ⊥
        endif
      endif
      break
    PROPOSE :
      //Je ne peux pas être le coordinateur
      assert(process ≠ getCoord(S(k)))
      S(k).estimate ← content
      S(k).lastUpdated ← S(k).round
      send(ACK, k, S(k).round, true, {getCoord(S(k))})
      incRound(S(k))
      break
    ESTIMATE :
      //Je dois être le coordinateur
      assert (process = getCoord(S(k)))
```

```
    //Je dois être dans la phase 2
    if (S(k).phase ≠ 2)
      //J'ai déjà reçu suffisamment d'estimations
      //J'attends les acquits pour ma proposition
      break
    endif

    if (content.lastUpdated < S(k).lastUpdated)
      //Le message contient une estimation plus ancienne que la mienne
      S(k).lastUpdated ← content.lastUpdated
      S(k).estimate ← content.estimate
    endif
    S(k).numEstimate ← S(k).numEstimate + 1

    if (S(k).numEstimate ≥ getLimit(S(k)))
      //J'ai reçu un nombre d'estimations suffisant
      send(PROPOSE, k, S(k).round, S(k).estimate,
           getOthers(S(k).Group))
      S(k).phase ← 4

      if (S(k).nbNack ≠ 0)
        //Je ai reçu des NACK's
        //Je ne peux pas décider
        incRound(S(k))
      endif
    endif
    break

  ACK :
    //Je dois être le coordinateur
    assert(process = getCoord(S(k)))

    if (content)
      //C'est un ACK
      S(k).nbAck ← S(k).nbAck + 1
    else
      //C'est un NACK
      S(k).nbNack ← S(k).nbNack +1
    endif

    if (S(k).phase = 2)
      //Les ACK's/NACK's ne sont traités qu'en phase 4
      break
    endif

    if (S(k).nbNack = 0)
      if (S(k).nbAck ≥ getLimit(S(k)))
        //Le nombre de NACK est nul est celui d'ACK est max.
        //Je décide et broadcaste la décision.
        send(RBCAST, S(k).k, -∞, S(k).estimate,
             getOthers(S(k).Group))
        trigger DECIDE(s.estimate)
        Already(k) ← card(S(k).Group) - 1
```

```
            endif
          else
              //Le nombre de NACK est non nul. ⇒ Je ne peux pas décider.
              incround(S(k))
          endif
          break
    endswitch

Upon SUSPECT(Suspected)
    forall s ∈ S
      if (getCoord(s) ≠ process ∧ getCoord() ∈ Suspected)
        //Je ne suis pas le coordinateur et le coordinateur est suspecté
        //J'envoie au coordinateur un NACK et je change de round
        send (ACK, s.k, s.round, false, {getCoord(s)})
        incRound(s)
      endif
    endforall
    Suspects ← Suspected
```

4.7 Atomic Broadcast

4.7.1 Rappels

Le module Atomic Broadcast permet d'effectuer un broadcast fiable et ordonné comme nous l'avons présenté à la section 1.3.1.

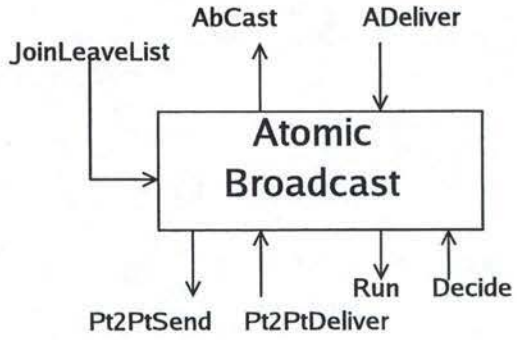


FIG. 4.13 – Interface du Atomic Broadcast.

Atomic Broadcast se base sur le module Reliable Pt2Pt qui assure la communication fiable et sur Consensus que nous utiliserons pour que les membres du groupe se mettent d'accord sur l'ordre dans lequel délivrer les messages.

4.7.2 Pseudo-Code de Atomic Broadcast

Nous n'avons pu parvenir à ce niveau du développement de la composition, cette partie a été réalisée par Jean Vaucher [20] lors de son projet de semestre au

sein du même département que nous. Ce travail a débouché sur le pseudo-code qui suit, ainsi qu'une implémentation se basant sur Appia.

L'idée principale de ce pseudo-code est d'utiliser Reliable Broadcast pour envoyer les messages, et ensuite de lancer un consensus afin de déterminer l'ordre dans lequel les messages reçus par Reliable Broadcast doivent être délivrés. Le pseudo-code qui suit incorpore un mécanisme de ramasse miette afin d'éviter que la mémoire ne soit saturée.

```
Upon INITIALIZE(P)
  assert(initialized ≠ true)
  initialized ← true
  A_delivered ← ∅
  A_Undelivered ← ∅
  k ← 0
  Processes ← P
  forall p ∈ Processes
    maxIdProProc(p) ← -1
  endforall
  running_consensus ← false
  trigger JOINLEAVELIST(∅, P)

Upon ABCAST(m)
  id ← abcast_ID(m)
  forall p ∈ Processes
    trigger PT2PTSEND(id::m, p, false)
  endforall

Upon PT2PTDELIVER
  if ((A_delivered(id) = ⊥)
      ∧ (id.number > maxIdProProc(id.proc))
      ∧ (A_Undelivered(id) = ⊥))
    forall p ∈ Processes
      trigger PT2PTSEND(id::m, p, false)
    endforall
    A_Undelivered(id) ← (id,m)
    if ((A_Undelivered ≠ ∅) ∧ ¬ running_consensus)
      trigger RUN(Processes, k::A_Undelivered)
      running_consensus ← true
      k ← k + 1
    endif
  endif
```

```
Upon DECIDE(k_decision::Decision)
  assert (k_decision + 1 = k)
  forall (id,m) ∈ Decision
    if ((A_Delivered(id) = ⊥) ∧ (id.number > maxIdProProc(id.proc))
      trigger A-DELIVER(m)
      A_Undelivered(id) ← ⊥
      A_Delivered(id) ← (id, m)
      while (A_Delivered(next(id)) ≠ ⊥)
        A_Delivered(next(id)) ← ⊥
        maxIdProProc(id.proc) ← maxIdProProc(id.proc) + 1
      endwhile
    endif
  endforall
  running_consensus ← false
  if ((A_Undelivered ≠ ∅) ∧ (¬ running_consensus))
    trigger RUN(Processes, k::A_Undelivered)
    running_consensus ← true
    k ← k + 1
  endif
```

Deuxième partie

Implémentation

Chapitre 5

Architecture globale

Sommaire

5.1 Introduction 116

5.2 Conventions et classes de base 116

5.2.1 Conventions 116

5.2.2 Classes de base communes 116

5.3 Code commun 117

5.3.1 Architecture utilisée avec un code commun 118

5.4 Approche modulaire et paradigme évènementiel 119

5.4.1 Rappel sur le framework abstrait 119

5.4.2 Transposition en Appia 120

5.4.3 Transposition en Cactus 122

5.1 Introduction

Avant d'entrer dans l'implémentation des modules, nous allons présenter ce qu'il nous a été demandé de réaliser par le LSR. Cette présentation devrait permettre au lecteur de mieux comprendre les choix architecturaux posés.

Le LSR souhaitait, à l'issue de notre travail, avoir suffisemment d'informations pour pouvoir comparer Appia et Cactus. Cette comparaison concerne la facilité d'utilisation et les performances des deux implémentations.

L'approche suivie est une composition modulaire s'insérant dans le paradigme évènementiel.

Pour faciliter les comparaisons des performances des deux implémentations, nous avons construit la pile abstraite en Appia et en Cactus, et ce en partageant toute la partie algorithmique. Dès lors, nous avonc développé un code commun pouvant être utilisé à la fois par Appia et Cactus. Ce code se veut le plus générique possible, c'est pourquoi nous avons tenté de le faire le plus proche possible du pseudo-code tout en répondant aux exigences d'Appia et de Cactus.

Dans cette optique, nous avons développé un certain nombre de classes et d'interfaces qui représentent les différents concepts rencontrés dans le pseudo-code.

Dans ce chapitre, nous allons commencer par définir les classes de base communes (pID, message, ...) implémentées ainsi que les conventions utilisées (Sect. 5.2). Ensuite, nous présenterons la technique et les motivations relatives à l'utilisation d'un code commun pour les différents frameworks (Sect. 5.3). Finalement, nous présenterons les aspects d'Appia et de Cactus utilisés pour achever les objectifs de programmation modulaire et évènementielle (Sect. 5.4).

5.2 Conventions et classes de base

5.2.1 Conventions

L'implémentation réalisée dans le cadre du projet se base sur la version JAVA des deux frameworks. Dans la suite, les notations, les types de données et les diagrammes de classes seront illustrés en utilisant les conventions JAVA.

5.2.2 Classes de base communes

La figure 5.1 illustre les classes de base utilisées par notre architecture. Celles-ci se basent principalement sur les types définis dans la section 3.2.3.

PID : La classe **PID** définit le format de l'identifiant d'un processus. Pour rappel, elle contient trois informations :

`ip` - L'adresse IP de la machine sur laquelle tourne le processus.

`port` - Le numéro de port sur lequel le processus écoute les demande de connexions.

`incarnation` - Le numéro d'incarnation du processus.

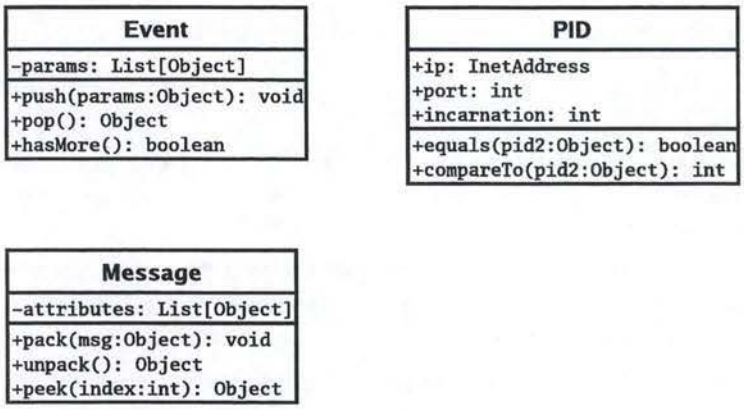


FIG. 5.1 – Classes de bases communes

Deux méthodes sont ajoutées pour implémenter la relation d'ordre totale définie sur les pIDs : *equals* et *compareTo*.

Message : La classe **Message** définit le format d'un message. Elle contient une liste d'objets que l'on peut transmettre sur le réseau. Un message est géré comme une pile LIFO. Nous pouvons y ajouter une en-tête par la méthode *pack* et en retirer une par la méthode *unpack*. Enfin, nous pouvons regarder la i^{eme} en-tête du message par la méthode *peek*.

Event : La classe **Event**[1] définit le format des paramètres d'un évènement. Elle est constituée d'une liste d'objets. Nous pouvons ajouter un élément en début de liste avec *push*, retirer le premier élément avec *pop* et vérifier si la file contient des éléments avec *hasMore*.

5.3 Code commun

L'objectif premier du LSR était de pouvoir comparer Appia et Cactus en terme de fonctionnalités et de performances. Pour comparer les performances, un point essentiel était l'utilisation de la même algorithmique et des mêmes fonctions. De cette façon, la différence était due au surcoût ajouté par les frameworks respectifs et non pas à une algorithmique différente. Grâce aux similitudes entre les frameworks, nous avons opté pour un code commun facilement interfaçable avec chacun des frameworks.

[1] Dans le code source, un choix malheureux en début d'implémentation nous a fait appeler cette classe VSCASTEvent.

5.3.1 Architecture utilisée avec un code commun

Dans le chapitre 3, nous avons vu que les deux primitives utilisées pour gérer les évènements sont :

- **trigger**(evt) : cette primitive permet de déclencher l'évènement *evt*.
- **Upon**(evt) : encore appelé handler, cette portion de code est exécutée lorsque l'évènement *evt* est reçu.

Ces deux primitives représentent les briques de base de la programmation évènementielle et sont disponibles dans nos trois frameworks.

Le but recherché est de définir ces deux primitives de manière générique au niveau du code commun. Ensuite, pour chaque framework, nous définirons une façon d'interfacer les deux primitives du framework avec celles du code commun. Les évènements déclenchés et reçus par un module variant de l'un à l'autre, la définition de ces interfaces sera différente. Mais elle se base sur un principe commun expliqué à la figure 5.2 qui illustre l'architecture adoptée. Sur ce diagramme de

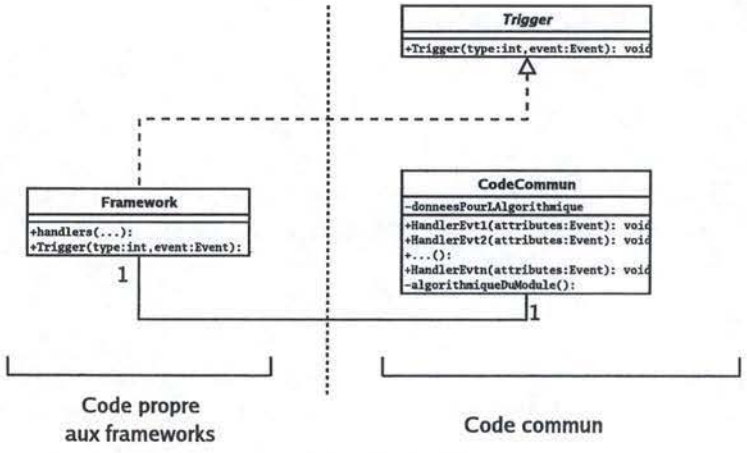


FIG. 5.2 – Architecture commune

classes, nous distinguons deux classes et une interface. La classe *Framework* représente la partie de code propre à chaque framework. Nous pouvons distinguer la pseudo-méthode *handlers(...)* qui désigne les handlers d'évènements et la méthode *Trigger* qui permet de déclencher des évènements propres au framework. La classe *CodeCommun* illustre le code commun. Elle contient deux parties importantes : la partie responsable de l'algorithmique du module (pseudo-méthode algorithmiqueDuModule()) et une primitive pour chaque handler d'évènement (HandlerEvt1, HandlerEvt2, ...). Finalement l'interface *Trigger* définit la signature de la méthode commune pour déclencher des évènements.

Pour gérer les primitives **trigger** et **Upon**, nous avons utilisé les mécanismes suivants :

trigger

Lorsque le code commun désire déclencher un évènement, celui-ci appele

la méthode générique *Trigger* avec comme paramètre un entier désignant le type de l'évènement et la liste de paramètres de cet évènement (Event). Cette primitive sera implémentée différemment par chaque framework. En fonction du type et des paramètres, elle va créer un évènement propre au framework considéré et va ensuite déclencher l'évènement correspondant. Nous avons donc un moyen de déclencher un évènement à partir du code commun.

Upon

Lorsque le framework reçoit un évènement représenté sur le schéma par la pseudo-primitive *handlers*, le code associé va analyser l'évènement reçu et créer une liste d'attributs de type Event. Ensuite, il appellera une primitive prédéfinie du code commun (HandlerEvt1(att :Event), ...). Nous avons donc un moyen de déclencher des portions du code commun en fonction des évènements reçus par le framework.

5.4 Approche modulaire et paradigme évènementiel

Pour l'implémentation de la pile, une des exigences était l'utilisation d'une approche modulaire. Pour cela, nous avons utilisé la version JAVA des deux frameworks Appia et Cactus (présentés aux sections 2.4 et 2.5) afin d'implémenter les modules définis dans le chapitre consacré à la définition de la pile Atomic Broadcast (Chap. 3 et 4).

Chaque framework ayant ses caractéristiques propres, ses avantages et ses inconvénients, la transposition du framework abstrait en Appia et en Cactus ne s'est pas fait sans mal. De nombreux problèmes se sont présentés et des choix architecturaux ont dû être posés.

Dans cette section, nous allons montrer comment transposer les principaux concepts du framework abstrait en Appia et Cactus.

5.4.1 Rappel sur le framework abstrait

Dans le chapitre 2 consacré à la description des frameworks, nous avons vu qu'un service complexe tel que Atomic Broadcast allait être représenté sous la forme d'une *pile*. Chaque fonction ou algorithme de ce service est encapsulé dans des containers indépendants, appelés *modules*. La communication entre modules est réalisée par le déclenchement d'*évènements* point à point. Des pseudo-modules appelés *connecteurs* permettent d'affiner le routage des évènements. Nous en avons étudié deux types : le multiplexeur/démultiplexeur et le multiplexeur total. Finalement, les deux acteurs externes restent à prendre en compte :

- la communication avec la *couche applicative*.
- la communication avec la *couche transport* sous-jacente.

En effet, pour accéder à ces couches transport, il doit exister un mécanisme d'interfaçage de façon à ne pas rendre ces deux couches dépendantes du framework

utilisé. Il serait en effet malheureux d'imposer l'utilisation du paradigme évènementiel au concepteur de l'application. Ces différents concepts sont illustrés sur un exemple simple à la figure 5.3.

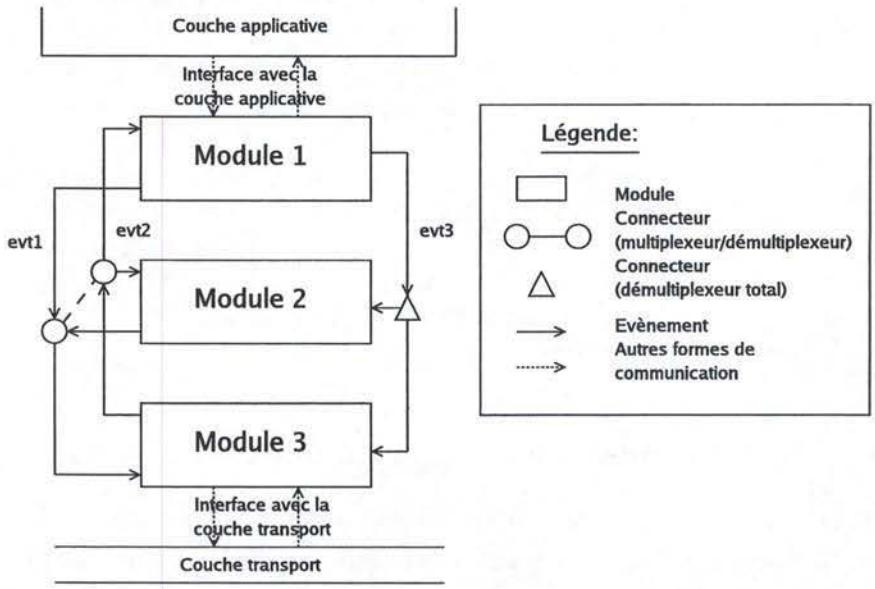


FIG. 5.3 – Un exemple de pile basé sur le framework abstrait

Cette figure présente trois modules qui communiquent entre eux à l'aide de trois évènements. Les deux premiers évènements (evt_1 et evt_2) sont routés par un connecteur de type multiplexeur/démultiplexeur du module 1 ou 2 vers le module 3. Le troisième évènement (evt_3) est routé par un connecteur de type multiplexeur total du module 1 vers les modules 2 et 3. Le module 1 communique avec la couche applicative. Le module 3 communique avec la couche transport.

En nous basant sur cet exemple qui présente toutes les structures fondamentales du framework abstrait, nous allons maintenant montrer comment implémenter ces concepts dans les deux frameworks concrets que sont Appia et Cactus.

5.4.2 Transposition en Appia

La figure 5.4 est l'équivalent Appia de la figure 5.3. Tous les concepts y sont repris.

Les modules et les évènements trouvent une correspondance immédiate en Appia. Seul le routage de ces derniers diffère. L'aspect le plus compliqué est sans nul doute l'introduction des connecteurs. Fournis dans le framework abstrait, ils sont inexistants en Appia. En effet, Appia a des évènements voyageant au sein de canaux. Les couches désireuses de recevoir un évènement le déclareront dans leurs évènements acceptés (ou requis). Un évènement, une fois lancé, sera attrapé par la première session sur son chemin. Vu que dans notre code, nous ne faisons pas

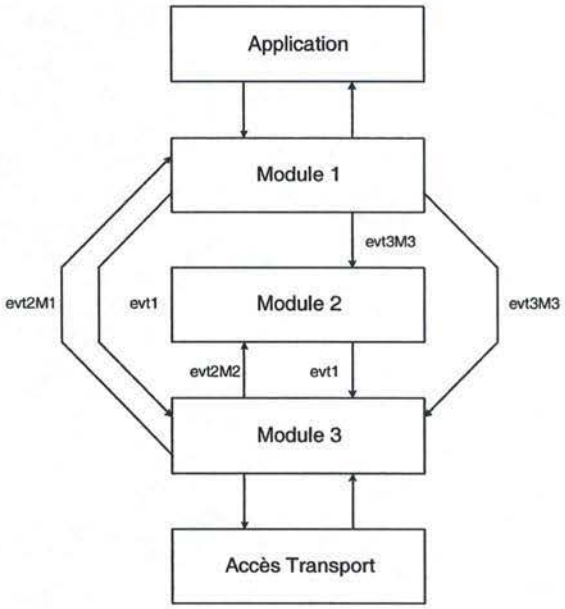


FIG. 5.4 – Pile Appia correspondant à la figure 5.3

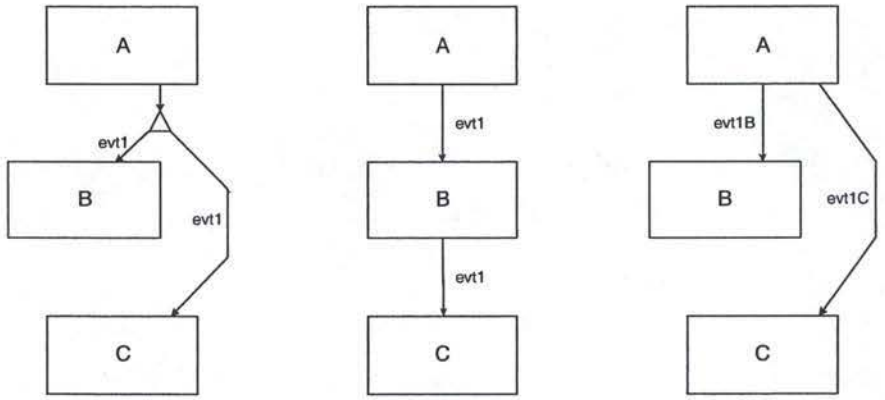
suivre les évènements[2], les autres sessions ne recevront jamais cet évènement. Le modèle traditionnel de routage des évènements Appia (Fig. 5.5(b)) n'est donc pas appriopriété pour traduire le multiplexeur total (Fig. 5.5(a)). Nous devons utiliser des évènements de types différents comme à la figure 5.5(c)[3].

Cette solution n'est cependant pas évidente à mettre en oeuvre car les couches B et C acceptent l'évènement evt_1 et non pas les évènements evt_1B et evt_1C. De plus la couche A ne fournit que l'évènement evt_1. Etant donné que nous sommes désireux de garder un maximum de flexibilité, nous ne pouvons pas nous permettre de modifier le code des couches, et encore moins des sessions. Nous nous sommes donc tout de suite intéressé aux solutions utilisant l'héritage offert par JAVA.

Nous avons déjà constaté que nous devions créer des sous-types de evt_1 (evt_1B et evt_1C) afin de laisser Appia s'occuper du routage des évènements. Pour cela, nous devons aussi faire en sorte que B et C acceptent, respectivement, evt_1B et evt_1C au lieu de evt_1. Il faut aussi que A fournisse ces deux évènements au lieu de l'unique evt_1. Ceci peut se faire facilement en étendant les couches concernées pour modifier les évènements admis, requis et fournis. Cette technique ne pose aucun problème pour le traitement des nouveaux évènements par les sessions car elles considèreront ces derniers comme un simple évènement evt_1. Par contre, le déclenchement des évènements est plus délicat. Heureusement, la structure du code

[2]Ils meurent dans la session.

[3]L'utilisation de plusieur canaux est possible, mais obligerait la session A a être au courant de l'existance de ces différents canaux. De plus, le déclenchement de evt_1 fait suite à la réception d'un autre évènement voyageant au sein d'un canal. evt_1 devrait donc voyager dans le même canal, ce qui ne serait plus le cas si nous utilisons plusieurs canaux pour implémenter le multiplexeur total.



(a) Routage dans le framework abstrait.

(b) Routage traditionnel Appia.

(c) Routage Appia lié au framework abstrait.

FIG. 5.5 – Implication sur le routage des évènements en Appia de la transposition du framework abstrait.

commun et l'interfaçage avec Appia nous aide grandement. En effet, toute la complexité du déclenchement se situe au niveau de la méthode *Trigger*. Il nous suffit donc d'étendre cette méthode.

Cette solution est valable pour le multiplexeur total, mais est facilement généralisable pour le multiplexeur/démultiplexeur (Fig. 5.6). Dans ce cas, nous devons, en plus, ajouter une en-tête au message contenu dans l'évènement transitant dans le démultiplexeur et l'enlever lors de son passage dans le multiplexeur afin de déterminer à quelle couche il est destiné. Ceci nous oblige à étendre, outre la méthode *Trigger* de A et B pour ajouter une en-tête au message contenu dans l'évènement evt_1[4], la méthode *Trigger* de C afin d'enlever l'en-tête du message contenu dans evt_2[5] et, en fonction de cette en-tête, déclencher un evt_2A ou evt_2B.

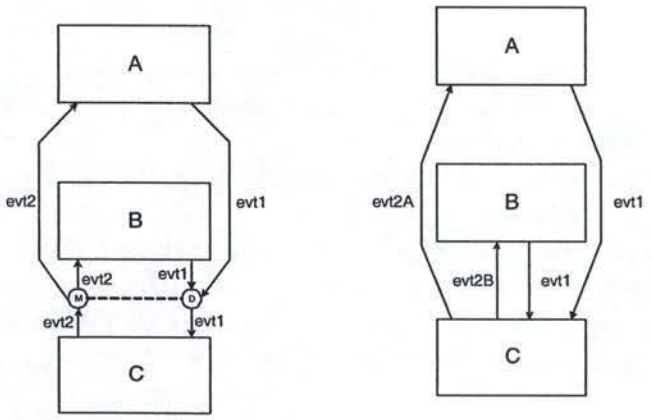
5.4.3 Transposition en Cactus

Pour expliquer les transformations, nous allons nous baser sur la figure 5.7. Celle-ci présente la transposition du framework abstrait en Cactus. Chaque concept va ensuite être expliqué au cours de l'exposé.

Sur cette figure, nous constatons que la pile est représentée par un protocole composite instancié par une session unique contenant quatre micro-protocoles. Trois de ceux-ci représentent les modules du framework abstrait. Le dernier sert

[4] Afin de pouvoir déterminer la destination de l'évènement evt_2 associé.

[5] evt_2 est l'évènement montant généré suite au traitement de evt_1. Généralement, le message contenu dans evt_1 transitera sur le réseau et lors de sa réception par une composition distante, sera encapsulé dans evt_2.



(a) Multiplexeur-Démultiplexeur dans le framework abstrait.

(b) Multiplexeur-Démultiplexeur en Appia.

FIG. 5.6 – Multiplexeur/Démultiplexeur.

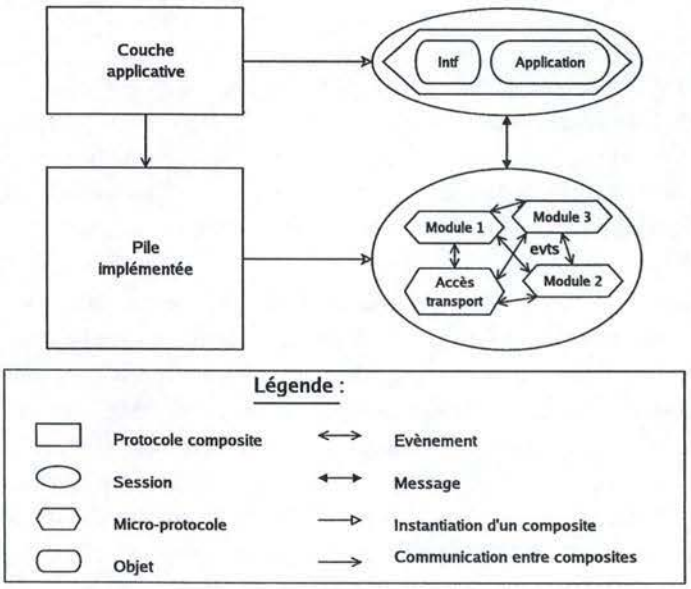


FIG. 5.7 – Un exemple de pile basé sur le framework abstrait

d'intermédiaire pour l'accès à la couche transport. La couche applicative est représentée par un protocole composite contenant un micro-protocole unique. Celui-ci contient l'application proprement dite et un module qui sert d'interface entre l'application et les concepts de Cactus.

Étudions maintenant en détail les choix d'implémentation pour chaque concept

ainsi que les motivations qui nous ont poussé à les faire.

Pile : Cactus repose sur un double niveau de composition :

- Le niveau des protocoles composites qui implémentent un service complexe ou encapsulent un acteur externe.
- Le niveau des micro-protocoles qui implémentent une fonction unique, spécifique et indépendante.

Dès lors, nous avons choisi d'encapsuler la pile à l'intérieur d'un composite unique afin d'utiliser le niveau micro pour les modules. La possibilité d'associer plus d'une session au composite permettrait de construire plusieurs piles différentes. Dans le cas de notre service, cela s'avère inutile. Dès lors, une session unique est associée à ce composite.

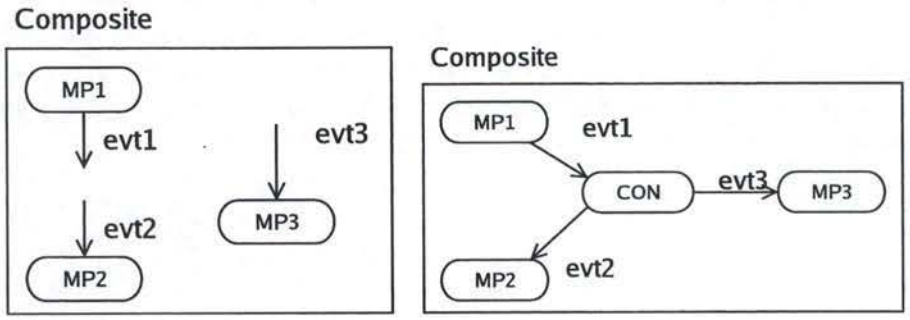
Module : Le niveau macro de Cactus ayant été utilisé pour construire la pile, nous optons pour l'utilisation des micro-protocoles pour représenter les modules.

Évènements : Pour illustrer le concepts d'évènements du frameworks abstrait, l'utilisation des évènements déclenchés entre micro-protocoles semble appropriée.

Connecteurs : La grande différence entre les évènements du framework abstrait et de ceux de Cactus est la possibilité de les router dans l'un mais pas dans l'autre. Lorsqu'un évènement e est déclenché en Cactus, tout handler enregistré pour cet évènement e est automatiquement exécuté. Cactus, de base, n'offre aucun moyen pour modifier leur route. Pour faire face à ce problème, plusieurs solutions sont envisageables.

La première solution consite à implémenter ce connecteur à l'intérieur même du micro-protocole. Lorsqu'un évènement est déclenché, les handlers des micro-protocoles qui écoutent celui-ci sont exécutés. Ils vérifient en premier lieu si le message passé en paramètre de cet évènement lui est destiné, par une en-tête particulière par exemple. Dans l'affirmative, il le traite sinon il l'ignore. Cette première solution, la plus simple, nécessite l'introduction d'une grande interdépendance entre les micro-protocoles. Chaque micro-protocole implémente des connecteurs et doit donc connaître en partie l'architecture utilisée. L'ajout d'une en-tête au message pour le distinguer des autres introduit également de la complexité. En programmation modulaire, cette solution est inacceptable.

La seconde solution consiste à utiliser des noms d'évènements différents pour chaque micro-protocole. Ainsi, un évènement déclenché ne donnera lieu à aucune suite car aucun handler n'est enregistré pour cet évènement. Lorsque deux ou plusieurs micro-protocoles veulent communiquer, nous introduirons un nouveau micro-protocole qui servira de connecteur entre eux. Ce connecteur définira un handler pour l'évènement qu'il désire router. Ce handler déclenchera à son tour les évènements destinés aux modules destinataires.



(a) Situation sans connecteur.

(b) Situation avec connecteur.

FIG. 5.8 – Implémentation possible d'un connecteur Cactus.

L'exemple de la figure 5.8 illustre le fonctionnement de ce connecteur. Nous avons trois micro-protocoles : $MP1$, $MP2$ et $MP3$. Le premier exporte l'évènement evt_1. Le second et le troisième importent respectivement evt_2 et evt_3. Ces trois évènements evt_i prennent les mêmes paramètres lorsqu'ils sont déclenchés. Dans la situation de départ, ces modules ne communiquent pas entre eux (Fig. 5.8(a)). Ensuite, pour les faire communiquer, nous introduisons un nouveau micro-protocole nommé CON. Rappelons que l'ajout d'un micro-protocole Cactus peut se faire à l'initialisation de la pile mais également en cours d'exécution. Ce micro-protocole ne contiendra qu'un seul handler. À l'écoute de l'évènement evt_1, il déclenchera les évènements evt_2 et evt_3. Cette situation est illustrée à la figure 5.8(b). Cette solution a comme inconvénient l'introduction de nombreux évènements et connecteurs. En effet selon cette description, pour chaque évènement point à point, un connecteur devra être défini.

La troisième solution consiste à étendre les fonctionnalités du framework par les mécanismes d'héritage de JAVA. Cette solution ne sera pas décrite dans ce mémoire car l'idée de connecteur pour Cactus n'était encore qu'à l'état embryonnaire. Il nécessiterait, en outre, l'exposé de nombreuse particularité de Cactus et de JAVA.

La solution finalement choisie pour implémenter les connecteurs Cactus est la troisième. En effet, la première solution introduit une trop grande dépendance entre les modules. La deuxième solution, quant à elle, introduit un trop grand nombre de micro-protocoles différents.

Couche applicative : A priori, Cactus ne possède aucun moyen de communication avec la couche applicative. Il encourage à encapsuler celle-ci dans un protocole composite. C'est ce que nous avons fait. Nous avons défini un composite et nous y avons associé une session unique dans laquelle nous avons enregistré un micro-protocole unique. Celui-ci contient deux grandes structures : l'application proprement dites écrites en JAVA et un module d'interfaçage. Ce dernier transforme les

appels de méthodes de communication par la création d'évènements Cactus correspondants. De manière inverse, cette couche d'interface transforme les évènements Cactus destinés à l'application en appel de méthodes. Un tel mécanisme d'interfaçage étant utilisé et décrit dans la section 5.3 (code commun), nous y renvoyons le lecteur sur la possibilité de créer une telle interface.

Chapitre 6

Implémentation des modules

Sommaire

6.1 Introduction 128
6.2 Le module Transport 128
6.2.1 Rappels 128
6.2.2 Problèmes rencontrés 128
6.2.3 Architecture implémentée 129
6.3 Le module Unreliable 130
6.3.1 Rappels 130
6.3.2 Problèmes rencontrés 130
6.3.3 Architecture implémentée 130
6.4 Le module Reliable Pt2Pt 130
6.4.1 Rappels 130
6.4.2 Problèmes rencontrés 131
6.4.3 Architecture implémentée 138
6.5 Le module Failure Detector 139
6.5.1 Rappels 139
6.5.2 Problèmes rencontrés 139
6.5.3 Architecture implémentée 139
6.6 Le module Consensus 140
6.6.1 Rappels 140
6.6.2 Problèmes rencontrés 141
6.6.3 Architecture implémentée 142
6.7 Le module Atomic Broadcast 143

6.1 Introduction

Ce chapitre présente l'implémentation de chaque module. Son but n'est pas de les décrire de manière détaillée, mais bien de montrer les problèmes rencontrés durant le projet. Pour chaque module, l'exposé se découpera en trois parties :

- *Rappels :* un bref rappel du rôle du module et des évènements qu'il utilise.
- *Problèmes rencontrés :* les problèmes rencontrés pendant l'implémentation.
- *Architecture implémentée :* une brève esquisse des classes du code commun du module.

Cette présentation, qui ne se veut pas exhaustive, permet de mettre en évidence certains problèmes qui serviront dans la comparaison des frameworks.

6.2 Le module Transport

6.2.1 Rappels

Le module Transport implémente des canaux de communication quasi-fiables au-dessus de la couche transport TCP (cfr. Sect. 4.2).

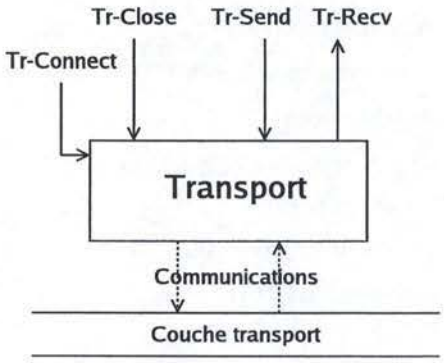


FIG. 6.1 – Interface du module Transport.

6.2.2 Problèmes rencontrés

L'implémentation n'a pas pu se baser sur un code commun, et ce, pour deux raisons. Tout d'abord, chaque framework utilise une technique particulière pour communiquer avec un acteur externe tel que la couche réseau. Appia utilise une solution ad hoc pour y arriver. Cactus encapsule la couche dans un micro-protocole.

La seconde raison concerne la sérialisation[1] des données. Appia autorise seulement l'envoi de *byte[]* sur le réseau, tandis que Cactus permet l'envoi de tout type d'objets.

[1] La manière dont les données sont envoyées sur le réseau.

Pour ces raisons nous avons été obligé d'utiliser deux codes différents. Mais l'architecture et l'algorithmique dans les deux codes sont les mêmes.

6.2.3 Architecture implémentée

La figure 6.2 illustre les classes qui composent le module Transport.

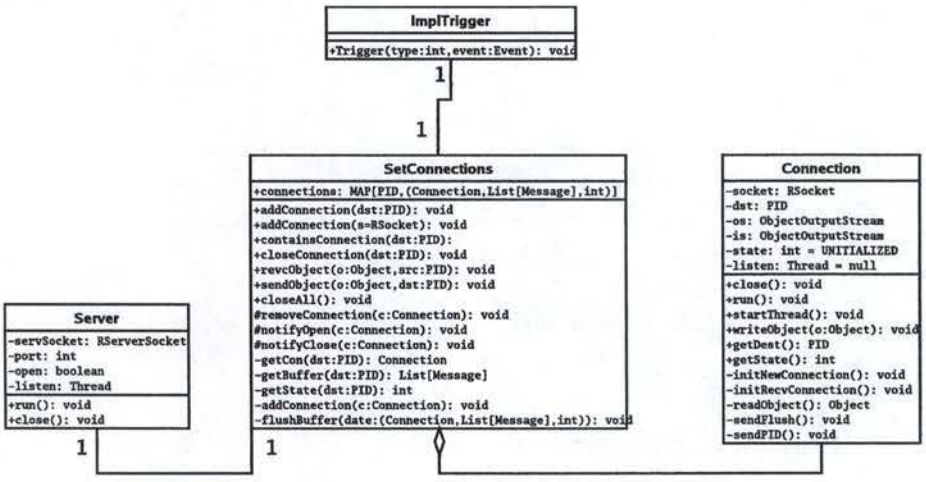


FIG. 6.2 – Diagramme des classes du module Transport.

Ce diagramme diffère légèrement des classes JAVA qui ont été implémentées pour Appia et Cactus. Ce choix a été réalisé afin de masquer les différences entre l'implémentation Appia et celle de Cactus. De cette manière, nous obtenons un diagramme de classes proche de celui d'un code commun. Il donne une idée générale de l'architecture. Il se compose de quatre classes :

Connection
Cette classe gère une connexion TCP selon la machine à états décrite sur la figure 4.6 (page 90).

SetConnections
Cette classe gère l'ensemble des connexions-TCP et transport établies. A chaque élément de cette ensemble est associé un objet **Connection** pour gérer la connexion-TCP correspondante et une machine à états dont le rôle est la gestion de la connexion-transport correspondante (cfr. Fig. 4.7 page 93).

Server
Cette classe écoute les demandes de connexions.

ImplTrigger
Cette classe désigne la classe du framework concret qui implémente l'interface *Trigger* (Sect. 5.3).

6.3 Le module Unreliable

6.3.1 Rappels

Le but du module Unreliable est de permettre l'accès à la couche transport UDP (cfr. Sect. 4.3).

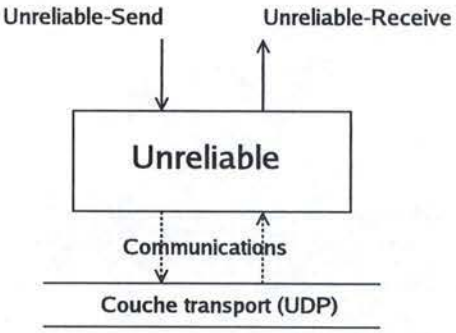


FIG. 6.3 – Interface du module Unreliable.

6.3.2 Problèmes rencontrés

Le module Unreliable étant également à la frontière du réseau, les mêmes problèmes que le module Transport se sont posés. De ce fait, nous avons aussi opté pour la séparation des deux codes. Dans le cas d'Appia, un module d'accès à UDP (UDPSimple) existait déjà et nous l'avons réutilisé. Dans le cas de Cactus, un module spécifique a été implémenté.

6.3.3 Architecture implémentée

Seul Cactus a nécessité une implémentation. Celle-ci était triviale et ne sera pas présentée.

6.4 Le module Reliable Pt2Pt

6.4.1 Rappels

Le but du module Reliable Pt2Pt, implémenté au-dessus du module Transport, est la gestion du groupe de processus lors de l'envoi fiable de messages d'un processus vers un autre. Plus précisément, son rôle est double. Il est responsable de transmettre un message sur le réseau uniquement si le destinataire est un processus connu. De manière similaire, il accepte les messages du réseau uniquement issus d'un processus connu (cfr. Sect. 4.4).

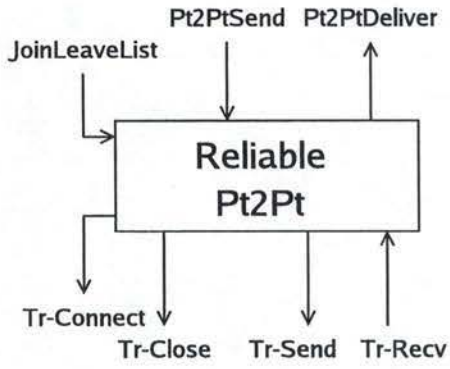


FIG. 6.4 – Interface du module Reliable Pt2Pt.

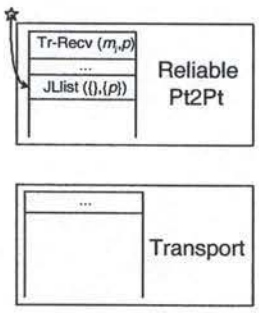
6.4.2 Problèmes rencontrés

Nous n'avons pas encore abordé le problème le plus important, celui lié au choix architectural que nous avons fait à la section 3.4 : l'éclatement de la partie communication en deux modules. Ces modules, bien que jouant des rôles distincts, manipulent une structure de données très similaire. Le module Transport gère les connexions entre différents processus, alors que le module Reliable Pt2Pt place ces processus dans un groupe. Une connexion ne devant être ouverte que lorsque le processus appartient au groupe et devant être fermée lorsque celui-ci le quitte. Le module Reliable Pt2Pt qui demande au module Transport de fermer une connexion s'attend donc à ne plus recevoir de messages en provenance de cette connexion.

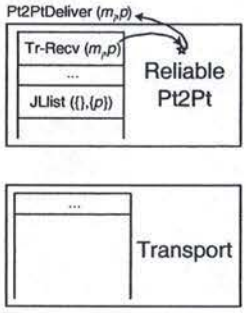
Les différentes étapes de l'exemple de la figure 6.5 illustre le problème auquel nous devons faire face. Un module demande au module Reliable Pt2Pt d'exclure le processus p du groupe en déclenchant un évènement JOINLEAVELIST. Ce dernier sera stocké dans le buffer du module Reliable Pt2Pt en attendant d'être traité (Fig. 6.5(a)). Les évènements se trouvant devant lui seront traités normalement. Par exemple, un évènement TR-RECV contenant un message émis par le processus p sera délivré vers le module supérieur (Fig. 6.5(b)). Pendant ce temps[2], le module Transport peut encore recevoir des messages provenant de p (Fig. 6.5(c)) et même les traiter et les faire suivre au module Reliable Pt2Pt ((Fig. 6.5(d))). Quand viendra enfin le moment de traiter l'évènement JOINLEAVELIST, ce dernier déclenchera un évènement TR-CLOSE pour fermer la connexion-TCP avec p ((Fig. 6.5(e))). Cet évènement ne sera, de nouveau, pas traité immédiatement[3] Nous constatons que le buffer du module Reliable Pt2Pt peut encore contenir des évènements de type TR-RECV véhiculant un message en provenance de p alors que l'évènement TR-CLOSE a été traité (Fig. 6.5(f)). Finalement, ces évènements seront traités par le module ne se rappelant plus que p est un processus venant juste

[2]Chaque module possède son propre thread.

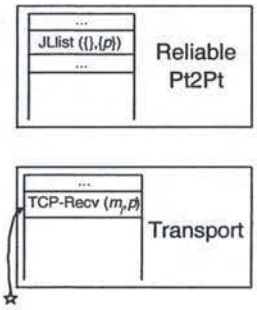
[3]Le module Transport pourrait encore délivrer des évènements TR-RECV transportant un message de p.



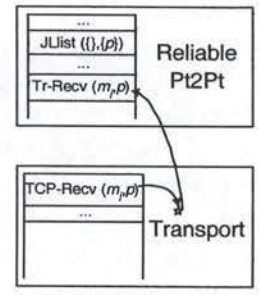
(a) Mise dans le buffer de JOINLEAVE-LIST({},{p}).



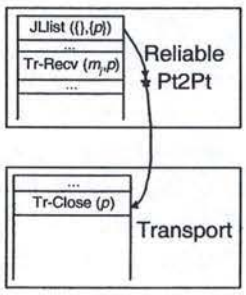
(b) Traitement de TR-RECV(m_i, p).



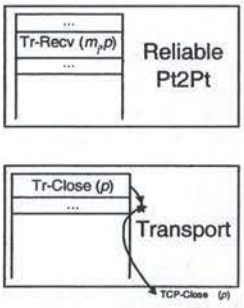
(c) Mise dans le buffer de TCP-RECV(m_j, p).



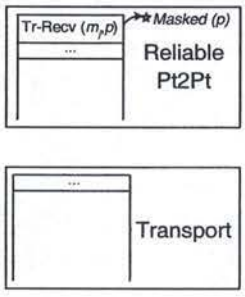
(d) Traitement de TCP-RECV(m_j, p).



(e) Traitement de JOINLEAVE-LIST({},{p}).



(f) Traitement de TR-CLOSE(p).



(g) Traitement de TR-RECV(m_j, p).

FIG. 6.5 – Comparaison des performances.

de quitter le groupe. De ce fait, il le masquera en pensant qu'il sera bientôt ajouté

aux processus connus. Toutes ces données resteront indéfiniment en mémoire car le processus p ne se manifestera plus jamais.

Plusieurs solutions sont envisageables. La première consiste à mémoriser les processus qui ont quitté le groupe. Celle-ci doit être écartée car elle ne fait que déplacer le problème. En effet, nous mémorisons des identifiants de processus au lieu de messages. Cela prend peut-être moins de place, mais demeure inacceptable.

Une seconde solution peut se situer au niveau du framework abstrait. Il suffit de n'avoir qu'un seul thread pour les modules Reliable Pt2Pt et Transport[4]. Malheureusement, vu que chaque module dispose d'un buffer d'évènements entrants, cette solution limite le problème, mais ne le résout pas. Il faudrait donc court-circuiter ce buffer afin d'obtenir un modèle complètement synchrone entre les deux modules posant problèmes.

Une troisième solution basée sur le principe des connecteurs est en cours d'étude. Nous ne développerons donc pas plus en profondeur ce point problématique au sein du framework abstrait.

Ce problème est également présent dans nos implémentations en Appia et Cactus. Nous allons étudier en profondeur chacun de ces deux cas, car les problèmes et les solutions apportées sont fort différents.

Problème Appia

Le problème que nous avons rencontré avec le framework abstrait est également présent dans notre implémentation Appia. Ceci est dû au caractère limitrophe des deux couches incriminées. Comme indiqué à la figure 6.6, nos deux couches ont leurs threads propres. Elles interagissent entre elles par le déclenchements d'évènements qui sont stockés dans un buffer avant traitement.

Nous avons déjà relevé toutes les difficultés inhérentes à cette architecture, et ce lors de la présentation du problème dans le framework abstrait. De ce fait, nous avons entamé une réflexion sur les solutions pouvant être apportées au niveau du framework abstrait. Nous irons donc droit au but en expliquant la solution implémentée.

Appia ne nous permet malheureusement pas de résoudre ce problème de manière très élégante car nous devrions pouvoir manipuler les évènements au sein du buffer Appia. La solution consiste à enlever tous les évènements TR-RECV provenant du processus quittant le groupe[5]. Nous devons donc contourner cette faille du framework.

Appia ne permet pas de partager des structures de données entre différentes sessions. Or, notre situation ne correspond pas tout à fait à cette restriction. En effet,

[4]Nous avons immédiatement exclus toutes solutions entraînant une modification de l'interface des modules et/ou de leur code afin d'établir un moyen de communication entre Reliable Pt2Pt et Transport.

[5]La session Reliable Pt2Pt démarrerait ce drainage lors du traitement de l'évènement JOIN-LEAVELIST, et la session Transport le stopperait lors du traitement de l'évènement TR-CLOSE correspondant.

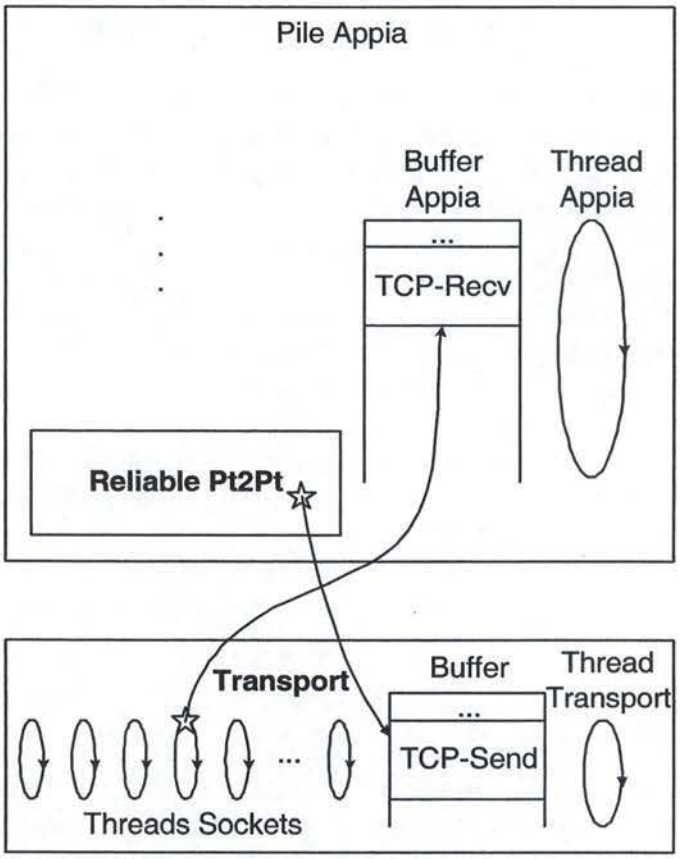


FIG. 6.6 – Architecture de notre pile Appia.

la couche Transport n'en est pas vraiment une car elle comporte son propre thread et son propre buffer pour les évènements entrants. Nous pouvons donc partager une structure de données nous permettant de résoudre notre problème.

Nous avons choisi de créer un buffer partagé pour chaque processus auquel nous sommes connectés. Ces buffers contiennent les évènements véhiculant les messages en provenance du réseau. Lors de la réception d'un message devant être délivré à la couche Reliable Pt2Pt, la couche Transport insère un évènement TR-RECV dans la pile. Cet évènement ne contient toutefois pas le message en question. Il sert juste de signal pour indiquer à la session Reliable Pt2Pt qu'un message est arrivé. Cette session peut le trouver dans le buffer associé au processus expéditeur. Evidemment, la couche Transport insère l'évènement correspondant dans ce buffer. Cet évènement est ensuite transmis au code commun.

Dans le cas qui nous intéresse, si un processus vient à quitter le groupe, lors du traitement de l'évènement JOINLEAVELIST, la session Reliable Pt2Pt supprime simplement le buffer[6]. Tous les messages qui se trouve dans ce buffer ne sont donc

[6]Ce buffer sera déréférencé et donc inaccessible, aussi bien par la couche Reliable Pt2Pt que par

jamais traités. Il est impossible d'en insérer des nouveaux. Donc, si la session Reliable Pt2Pt reçoit un évènement lui indiquant l'arrivée d'un message, elle vérifie d'abord que le buffer existe toujours, avant d'aller le chercher. S'il n'y a plus de buffer, la couche a fini son travail. De même pour la couche Transport qui, voulant insérer l'évènement contenant le message dans le buffer, ne trouve pas le buffer.

Cette solution fonctionne très bien pour autant que nous nous trouvons en bordure de pile. Si ce problème se pose au sein même de la pile[7], cette solution n'est plus valable car nous ne pouvons pas partager de structures de données. A ce niveau, nous sommes obligés de modifier l'interaction entre les sessions afin de résoudre le problème.

Problème Cactus

L'optique de Cactus est l'utilisation d'un protocole composite différent pour le module gérant la communication avec un acteur externe. Selon cette optique, le module Transport est encapsulé dans un protocole composite et non pas dans un micro-protocole. La figure 6.7 présente cette composition.

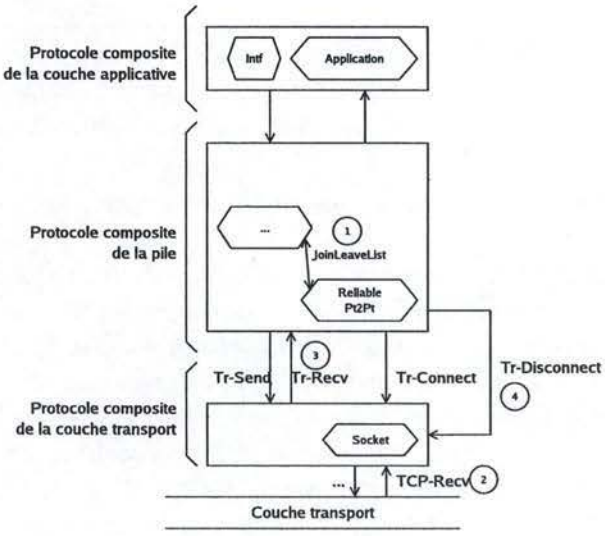


FIG. 6.7 – Architecture Cactus conseillée.

Pour comprendre le problème de Cactus, rappelons-nous quelques points sur la gestion de la concurrence. Nous avons présenté quelques propriétés dont une permettant d'ajouter la propriété FIFO à l'intérieur d'un composite (Sect 2.5.3). Selon cette propriété, lorsqu'un thread débute le traitement d'un message dans un composite, aucun autre ne peut débuter le sien et ce tant que le premier n'a pas fini.

---

la couche Transport.

[7]Nous avons vu, dans l'étude du problème avec le framework abstrait, que ce conflit existait même en présence d'un seul thread, pour autant qu'il y ait des buffers.

Par contre, en ce qui concerne la gestion des messages concurrents entre protocoles composites, nous n'avons pu tirer aucune propriété intéressante.

Le problème qui se pose chez Cactus se réduit à l'impossibilité de garantir la propriété FIFO entre protocoles composites. Voyons le phénomène qui se produit sur la figure 6.7.

Le diagramme de séquence de la figure 6.8 présente le problème qui se pose. Supposons, pour simplifier, que le groupe de processus connu se limite à un pro-

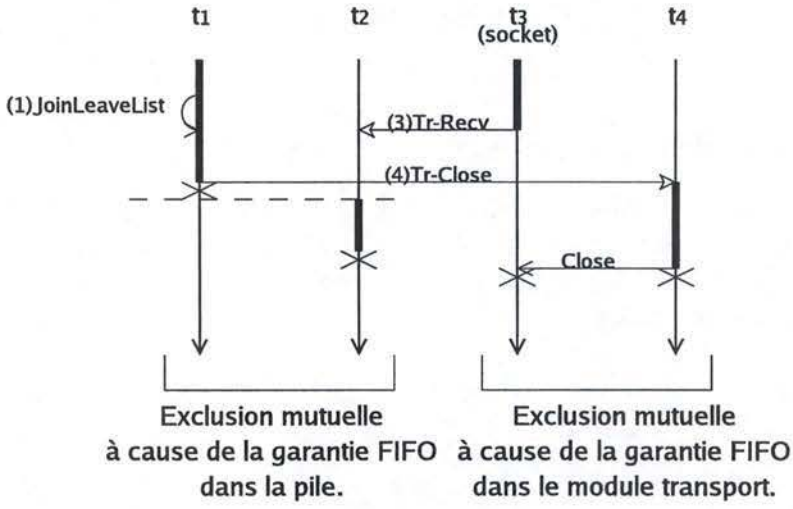


FIG. 6.8 – Diagramme de séquence du problème Leave/Recv en Cactus

cessus unique. Supposons également que le thread t_1 qui s'exécute dans le composite pile génère un évènement JOINLEAVELIST afin d'exclure ce processus (1). Ensuite, le thread t_3 associé au socket dans le composite Transport reçoit un évènement TCP-Recv du réseau (2). Un nouveau message Tr-Recv est créé et le thread t_2 y est associé (3). Ce message est envoyé vers la pile. t_3 est ensuite suspendu. Le message TR-RECV est mis en attente car le thread t_1 continue à s'éxécuter. t_1 se termine finalement par l'envoi d'un message TR-CLOSE vers le composite Transport. Un thread t_4 y est associé (4). A ce moment le processus est exclu du groupe. Le thread t_2 commence son exécution. Il constate que le message TR-RECV est issu d'un processus qui n'appartient pas au groupe. Il est mis dans un buffer et le thread se termine. Finalement, le thread t_4 termine son exécution en fermant le socket. Nous avons donc un message mis en attente dans un buffer destiné à un processus qui a quitté le groupe.

Solution Cactus

La question posée est de savoir ce qui se passe si la garantie FIFO est également assurée entre les protocoles composites. Dans la section dédiée à Cactus (Sect. 2.5), nous avons vu plusieurs mécanismes qui ajoutent cette propriété. Par exemple,

choisissons la première. Elle consiste à ajouter la garantie FIFO également entre les protocoles composites par extension du framework. Sur tout le framework, un seul thread s'exécute à la fois et ce depuis la prise en charge d'un message jusqu'à la fin de son traitement. Dans ce cas, les diagrammes de séquence des figures 6.9 et 6.10 présentent les deux scénarios possibles.

Le premier scénario, illustré à la figure 6.9, est le cas où le thread t_1 s'exécute. Dans ce cas, le thread du socket ne peut prendre la main et ce jusqu'à la fin du traitement du JOINLEAVELIST et du TR-CLOSE, i.e., jusqu'à ce que le socket soit fermé et le thread associé fini.

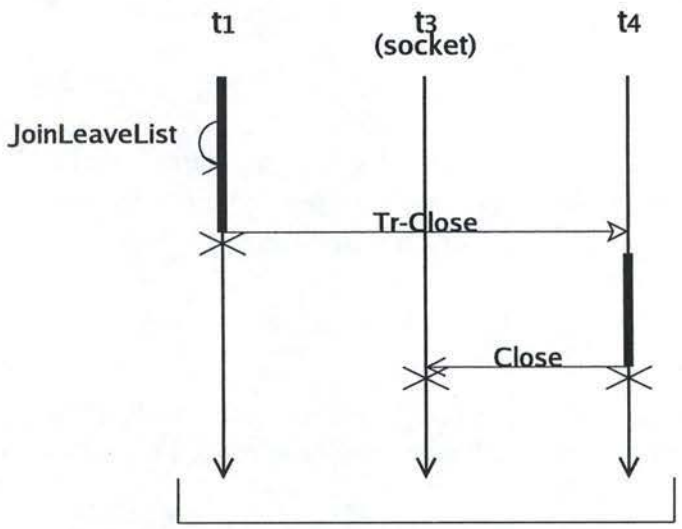


FIG. 6.9 – Scénario 1.

Le second scénario, illustré à la figure 6.10, est celui où le thread du socket (t_3) s'exécute. Dans ce cas, aucun autre message/évènement ne peut s'exécuter à cause de la garantie FIFO. Le message TR-RECV ainsi reçu continue son exécution jusqu'à son traitement complet. Ensuite le thread t_1 peut commencer son exécution et fermer le socket.

Comme nous pouvons le constater sur ces deux scénarios, le problème ne se produit plus. L'ajout de la garantie FIFO à l'intérieur et entre les composites permet d'éliminer ce problème. Pour ajouter cette garantie FIFO, nous avons étendu le framework.

Une autre solution, équivalente, consiste à placer la couche transport dans un micro-protocole. De cette manière, la pile et le module Transport se trouvent dans un même protocole composite dans lequel la garantie FIFO est assurée. C'est cette

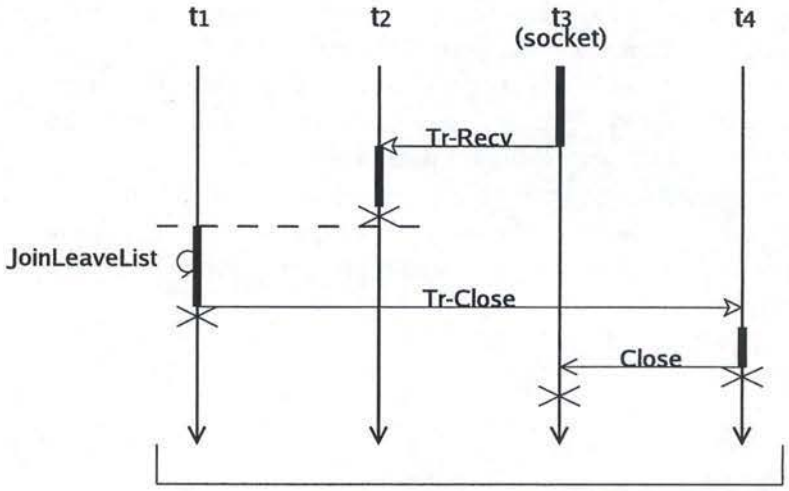


FIG. 6.10 – Scénario 2.

dernière solution qui a été retenue pour l'implémentation. Nous n'avons toutefois pas voulu présenter cette solution car elle aurait compliqué l'exposé.

6.4.3 Architecture implémentée

La figure 6.11 présente les classes du code commun utilisées pour implémenter le module Reliable Pt2Pt.

Ce diagramme se compose de quatre classes :

KnownProcesses
Cette classe définit le groupe de processus connus.

MaskedProcesses
Cette classe contient les processus qui ont ouvert un canal de communication vers notre processus mais que nous ne connaissons pas.

Reliable_Pt2Pt_Impl
Cette classe contient le code des handlers des évènements reçus par ce module : PT2PTSEND, TR-RECV et JOINLEAVELIST.

ImplTrigger
Cette classe désigne la classe du framework concret qui implémente l'interface *Trigger* (Sect. 5.3).

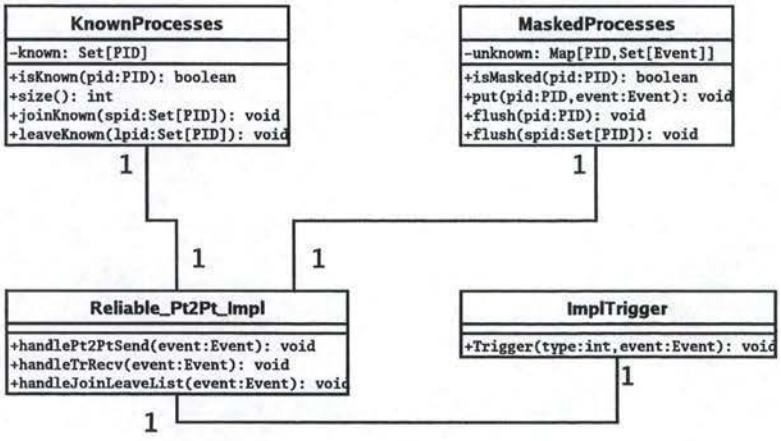


FIG. 6.11 – Diagramme des classes du code commun du module Reliable Pt2Pt.

6.5 Le module Failure Detector

6.5.1 Rappels

L'objectif du module Failure Detector est de suspecter les processus susceptibles d'être tombés en panne (cfr. Fig. 4.5).

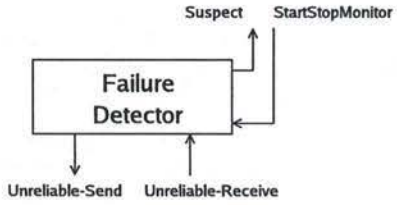


FIG. 6.12 – Interface du module Failure Detector.

6.5.2 Problèmes rencontrés

Ce module implémente un détecteur de type *ping*. Pour le construire, nous avons besoin de timers périodiques. Dans son modèle d'interaction externe, Appia définit des timers qui ont pu être réutilisés pour implémenter ce module. Par contre, Cactus, ne proposant rien, des timers spécifiques ont dû être implémentés.

6.5.3 Architecture implémentée

La figure 6.11 présente les classes du code commun utilisées pour implémenter ce module.

Ce diagramme présente trois classes :

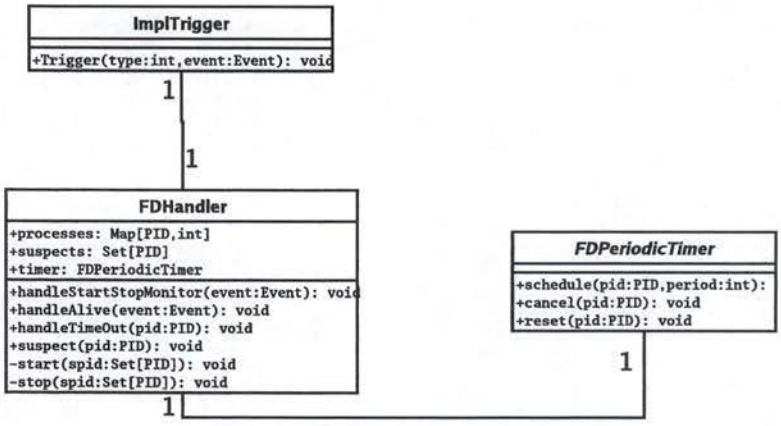


FIG. 6.13 – Diagramme des classes du code commun du module Failure Detector.

FDHandler
Cette classe contient le code des handlers des évènements reçus par ce module : STARTSTOPMONITOR, UNRELIABLE-RECV(handleAlive). Il contient également le handler qui gère les timeouts périodiques.

FDPeriodicTimer
Cette classe abstraite sert d'interface entre le module et les timers.

ImplTrigger
Cette classe désigne la classe du framework concret qui implémente l'interface *Trigger* (Sect. 5.3).

6.6 Le module Consensus

6.6.1 Rappels

Le module Consensus implémente un algorithme permettant à tous les membres d'un groupe de se mettre d'accord sur une valeur préalablement proposée (cfr. Sect. 4.6).

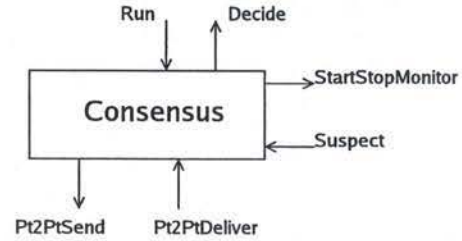


FIG. 6.14 – Interface du module Consensus.

6.6.2 Problèmes rencontrés

Adpatation du code Neko

L'implémentation du module Consensus s'est basée sur un code préexistant écrit dans le cadre d'un autre projet. Celui-ci consiste en l'implémentation des communications de groupe sur un autre framework : Neko [30]. L'algorithme Consensus implémenté se base sur celui présenté dans la section 4.6. L'objectif de notre travail sur ce module n'est pas l'implémentation de l'algorithme, mais bien un travail de rétro-ingénierie : comprendre et adapter le code Neko à nos deux frameworks. Les principaux problèmes posés sont donc des problèmes d'interfacage et ce à deux niveaux.

Le premier problème concerne l'interfaçage de nos évènements RUN, DECIDE, PT2PTSEND, PT2PTDELIVER, SUSPECT et STARTSTOPMONITOR. Ceux-ci étant définis dans Neko comme des appels de méthode JAVA, il est aisé de faire appeler ces méthodes par les handlers de nos frameworks respectifs et ce pour les évènements entrants. Pour les évènements sortants, nous avons modifié les méthodes correspondantes afin d'appeler la méthode *Trigger* de notre interface commune.

Le second problème concerne l'utilisation de certaines structures propres à Neko. La gestion de l'identifiant du processus (pID) et le format des messages dans Neko sont différents. Pour les pIDs, nous avons adapté tout le code à notre propre méthode d'identification. Pour le format des messages, nous avons utilisé les mécanismes d'héritage de JAVA ; nous avons implémenté au-dessus de nos messages standards une classe *ConsensusMsg* qui implémente de la manière la plus efficiente possible le format des messages utilisés par Neko. Finalement, le code Neko suppose l'existence de quelques structures sous-jacentes propres à lui-même. Celles-ci ne sont pas fort complexes et nous avons décidé de les simuler.

Reliable Broadcast

Le code de Consensus présenté se base sur deux types d'envoi :

- L'envoi point à point fiable.
- L'envoi en utilisant l'algorithme Reliable Broadcast (Sect. 1.3.1).

Pour l'envoi point à point, nous avons utilisé le module Reliable Pt2Pt. Par contre, en ce qui concerne Reliable Broadcast, l'implémentation de cet algorithme a été réalisée à l'intérieur du module Consensus. Cette solution n'est pas la meilleure à notre point de vue. En effet, un module est utilisé pour implémenter une fonction ou un algorithme unique. Notre module, par contre, en implémente deux : l'algorithme consensus et l'algorithme Reliable Broadcast[8].

[8]Toutefois, un module séparé pour Reliable Broadcast aurit pu introduire un problème du même genre que celui présenté à la section 6.4.2 (Leave/Recv), qui aurait pu poser des problèmes supplémentaires.

6.6.3 Architecture implémentée

Le code commun implémenté est celui représenté par le diagramme de classe de la figure 6.15[9].

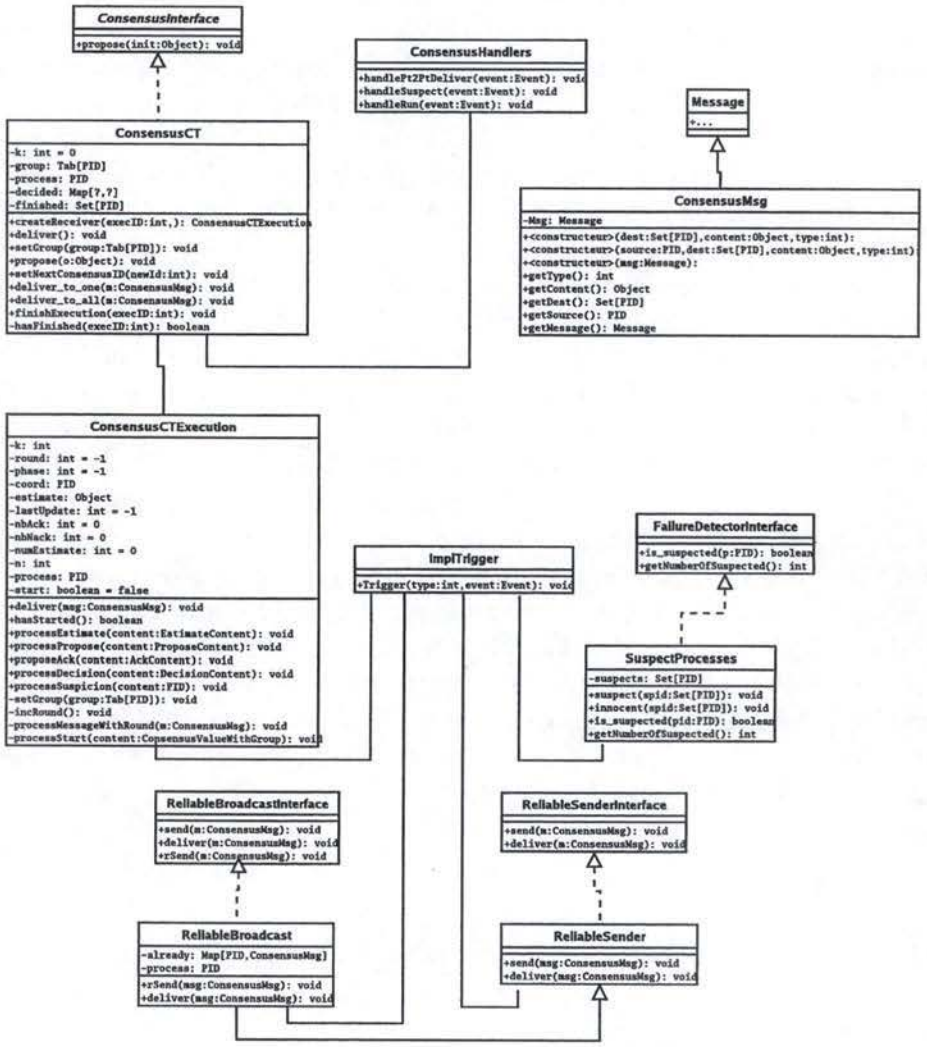


FIG. 6.15 – Diagramme des classes du code commun du module Consensus.

Nous distinguons quatre parties.

La première partie est consacrée à l'algorithme Consensus. Les classes et les interfaces propres au consensus sont les suivantes :

[9]Pour alléger le schéma, ni les relations vers la classe ConsensusMessage, ni les relations entre les classes relatives à consensus et les classes d'interfaçage avec les autres modules de notre pile ne sont représentés.

ConsensusInterface
Cette interface définit la méthode `propose` qui doit être implémentée par chaque implémentation de consensus (au cas où nous décidions d'implémenter différentes versions de l'algorithme).

ConsensusCT
Cette classe permet d'exécuter et de gérer plusieurs consensus en parallèle.

ConsensusCTExecution
Cette classe gère une exécution unique d'un consensus.

La deuxième partie est consacrée à l'algorithme Reliable Broadcast :

ReliableBroadcastInterface
Cette interface définit les méthodes que toutes les versions de Reliable Broadcast doivent implémenter : rSend et deliver.

ReliableBroadcast
Cette classe implémente l'algorithme Reliable Broadcast (Sect. 1.3.1).

La troisième partie contient les classes qui servent d'interface avec les autres modules utilisés :

ReliableSenderInterface
Cette interface définit les méthodes d'accès au module Reliable Pt2Pt.

ReliableSender
Cette classe implémente l'accès à notre module Reliable Pt2Pt.

FailureDetectorInterface
Cette interface définit les méthode d'accès au module Failure Detector.

SuspectProcess
Cette classe implémente l'accès à notre module Failure Detector.

Finalement, la dernière partie est consacrée à l'interfaçage avec le framework :

ImplTrigger
Cette classe désigne la classe du framework concret qui implémente l'interface *Trigger* (Sect. 5.3).

6.7 Le module Atomic Broadcast

Les détails d'implémentation de ce module ne seront pas donnés ici. Ceux-ci ont été traités dans le cadre d'un autre projet. Le problème principal relève de l'utilisation d'un multiplexeur/démultiplexeur défini aux sections 5.4.2 (pour Appia) et 5.4.3 (pour Cactus).

Troisième partie

Comparaison des frameworks

Chapitre 7

Comparaison

Sommaire

7.1 Introduction 148

7.2 Similitudes 148

7.3 Différences 149

7.3.1 Modèle de composition 149

7.3.2 Modèle d'interaction 149

7.3.3 Modèle de concurrence 150

7.4 Tests de performance 150

7.4.1 Configuration 151

7.4.2 Tests 151

7.4.3 Analyse 152

7.5 Conclusion 154

7.5.1 Modèle de composition 154

7.5.2 Modèle d'interaction 155

7.5.3 Modèle de concurrence 155

7.1 Introduction

Nous avons, tout au long de ce mémoire, montré une manière de faire des communications de groupe modulaires et en avons réalisé deux implémentations en nous basant sur deux frameworks de composition de protocoles : Appia et Cactus. Nous allons maintenant comparer ces derniers afin de déterminer leurs points forts et leurs points faibles face à l'usage que nous en avons fait.

Nous commencerons par relever les similitudes (Sect. 7.2) et enchaînerons avec les différences (Sect. 7.3) entre Appia et Cactus. Après cette comparaison qualitative, nous analyserons les performances des deux frameworks (Sect. 7.4). Finalement, nos conclurons en émettant suggestions d'améliorations (Sect. 7.5).

7.2 Similitudes

Nous avons, durant notre travail, travaillé au départ d'un framework abstrait. Nous avons été jusqu'à écrire le code des modules en utilisant les caractéristiques de ce dernier. Cette prouesse n'a pu avoir lieu que grâce à un certain nombre de similitudes entre Appia et Cactus.

Tout d'abord, nous pouvons constater que les deux frameworks ont adopté un modèle d'interaction évènementiel. Une session Appia ou un micro-protocole Cactus reçoivent des évènements causant l'exécution d'un handler. Ce handler modifiera l'état interne du module et pouvant déclencher des évènements qui seront à leur tour reçus par d'autres modules.

Un autre point commun se situe au niveau de la communication entre applications distantes. Celle-ci ne peut se faire que par l'échange de messages. Bien que la structure des messages diffère, le concept d'en-tête est présent dans les deux frameworks[1]. Si un module donné ajoute une en-tête à un message, seul le module correspondant dans la composition de destination est capable d'accéder et d'enlever cette en-tête. Quand un message descend vers le réseau, les modules de niveaux inférieurs ne voient pas les en-têtes des niveaux supérieurs et considèrent le tout (messages et en-têtes) comme un simple message. Cette caractéristique nous a permis de traiter de manière homogène les messages au sein du code commun.

De même, nous avons constaté avec notre pile que dans le cadre des communications de groupe, nous ne travaillions qu'avec des évènements point à point[2]. Cette particularité nous ayant forcé à introduire le concept de connecteur. Les modèles d'interactions de Cactus et d'Appia ont ici montré leurs limites.

---

[1] Cactus ne définit pas explicitement les en-têtes, mais les attributs avec une portée de type *peer* se comportent de la même manière.

[2] Un évènement entrant dans un module n'en sort jamais, mais donnera lieu au déclenchement de nouveaux évènements.

7.3 Différences

Pour exposer les différences entre Appia et Cactus, nous allons réutiliser la décomposition en trois modèles que nous avons décrites à la section 2.2.

7.3.1 Modèle de composition

Le modèle de composition d'Appia est hiérarchique, alors que celui de Cactus (entre micro-protocoles) est coopératif. Cet aspect pourrait rendre Appia moins flexible que Cactus, mais nous n'avons, pour l'instant, relevé aucune structure réalisable avec Cactus qui ne le soit pas avec Appia.

Cactus, contrairement à Appia, permet de construire des compositions sur deux niveaux : les micro-protocoles et les protocoles composites. Ces niveaux, bien qu'intéressants d'un point de vue conceptuel, sont peu utilisés car l'interaction entre composites est trop restrictive.

Notons qu'Appia effectue un contrôle partiel de la correction de la composition en vérifiant que tous les évènements requis par une couche soient bien fournis par une autre. Ce qui est totalement inexistant avec Cactus.

7.3.2 Modèle d'interaction

Modèle d'interaction interne

Appia ne permet une interaction interne que par le déclenchement d'évènements, alors que Cactus autorise également le partage de structures de données (entre micro-protocoles). Cet ajout ne devant, à notre avis, n'être utilisé que comme moyen de synchronisation entre micro-protocoles.

Le déclenchement d'évènements n'a cependant pas les mêmes caractéristiques dans les deux frameworks. Cela étant dû, en grande partie, au modèle de composition. En effet, Appia, avec sa structure hiérarchique, définit des canaux que les évènements doivent suivre ; alors que Cactus lance un évènement pouvant être reçus par n'importe quel micro-protocole (écoutant cet évènement). Les conséquences de ces modèles peuvent être vues à la figure 7.1 où la structure hiérarchique oblige Appia à utiliser les ECHOEVENTs dans la figure 7.1(b).

Modèle d'interaction externe

L'interaction entre la composition et l'application ou le réseau revêtent des approches bien distinctes. Avec Cactus, l'application et le réseau doivent adopter l'interface des protocoles composites (interface de type x-kernel), ce qui nous semble fort restrictif. Avec Appia, l'environnement (application ou réseau) peut insérer n'importe quel évènement dans la pile grâce à la primitive *asyncGo*. Malheureusement, l'interface dans l'autre sens doit se faire de manière ad-hoc (par exemple à l'aide d'une queue producteur-consommateur.).



(a) Routage traditionnel Appia.

(b) Routage particulier oblige Appia à utiliser les ECHOEVENTs.

FIG. 7.1 – Deux flux d'évènements différents.

Appia, interdisant tout accès aux ressources système au sein d'une session, il est normal qu'un certain nombre de couches et de mécanismes soient offerts (timers, timers périodiques, couches UDP et TCP). Cactus n'offre quant à lui qu'un protocole composite (UTP) fonctionnant au-dessus de TCP.

7.3.3 Modèle de concurrence

Appia est mono-thread, alors que Cactus autorise énormément de concurrence (jusqu'au thread par évènement). Nous avons vu dans notre composition que même si Cactus autorise autant de concurrence, cette dernière n'est pas toujours une bonne amie. Nous l'avons d'ailleurs complètement supprimée. Ces exécutions en parallèle n'ont de sens que si la composition s'exécute sur une machine multi-processeurs ou si nous voulons pouvoir donner une plus grande priorité au traitement d'un évènement par rapport à un autre (ou un module par rapport à un autre).

La synchronisation entre micro-protocoles Cactus se fait en utilisant des structures de données partagée comme sémaphore. La synchronisation entre protocoles composites est impossible sans modifier la manière dont ils interagissent. Avec Appia, la synchronisation de la pile avec l'environnement peut se faire en utilisant des queues qu'il est possible de drainer (supprimer certains évènements). Malheureusement, cette technique ne peut pas fonctionner au sein de la pile[3] où, malgré l'absence de concurrence, les problèmes de synchronisation sont bien présents (à cause du caractère asynchrone des évènements).

7.4 Tests de performance

Ayant implémenté deux services identiques de manières différentes, il est intéressant de mesurer et comparer leurs performances. Ce qui n'est cependant pas

[3] Interdiction de partager des structures de données.

la chose la plus aisée. Il est en effet difficile de déterminer les critères que l'on souhaite mesurer et de trouver une manière de le faire.

N'ayant pas implémenté le module Atomic Broadcast, nous pouvions, tout au plus, tester le module Consensus. Or, mesurer les performances d'un algorithme distribué n'est pas une chose aisée, surtout quand nous voulons tolérer les pannes. Le nombre de paramètres à considérer est tellement grand que ces tests auraient pris à eux seuls un mois ; mois que nous n'avions pas. Nous nous sommes donc contentés de tester le module Reliable Pt2Pt[4], qui, faute de comporter de nombreux modules, nous a tout de même permis de révéler quelques points intéressants.

7.4.1 Configuration

Pour tester le module Reliable Pt2Pt, nous avons utilisé SockPerf 1.2, un outil développé par IBM pour mesurer la performance des sockets JAVA [10]. Nous avons donc dû construire une interface, entre la pile et SockPerf, similaire à celle des sockets JAVA (Fig. 7.2).

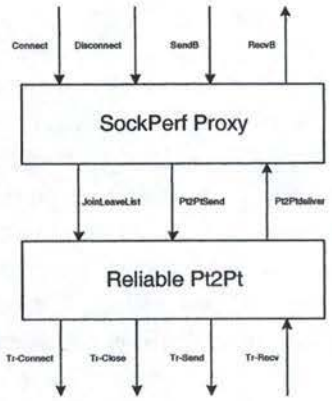


FIG. 7.2 – Interface entre SockPerf et Reliable Pt2Pt.

Le matériel utilisé pour les mesures est un réseau Ethernet 100 Base-TX, sans trafic d'une tierce partie, sur lequel nous avions deux PCs sous Linux Red Had 7.2 (kernel 2.4.18-19) avec un processeur Pentium III 766MHz et 128MB de RAM. La machine virtuelle JAVA utilisée est celle de Sun (JDK 1.4.0).

7.4.2 Tests

Nous avons testé le débit et le temps de réponse du module Reliable Pt2Pt.

[4]Nous n'avons pu effectuer que des tests préliminaires car nous avons été amenés à faire des modifications importantes sur le module Transport après notre départ du LSR. Les résultats exposés dans la suite de cette section sont ceux obtenus par Sergio Mena et exposés dans [37].

Débit

Nous avons effectué un test de débit en observant sa variation en fonction de la taille des messages transmis.

Le test se déroule comme suit. D'abord, le client se connecte en utilisant l'évènement CONNECT qui cause le déclenchement d'un JOINLEAVELIST qui ouvrira un socket au niveau TCP. Quand le socket est ouvert, le client envoi des messages d'une taille donnée en utilisant l'évènement SEND-BYTES. Le traitement de cet évènement déclenchera un TR-SEND qui causera le déclenchement d'un TCP-SEND. Le client envoie les messages le plus vite possible, et le débit est mesuré.

Rappelons que le thread de SockPerf exécutera toute la composition implémentée en Cactus, alors que le thread Appia se chargera d'exécuter la sienne. Dans un premier temps, nous avons dû implémenter un contrôle de flux pour Appia. Nous avons donc dû implémenter nous-même un contrôle en lmimitant le nombre d'évènements entrant par le dessus de la pile. Le thread SockPerf étant bloqué lorsque le buffer de l'EVENTSCHEDULER est plein, pour Appia, alors qu'il est bloqué lorsque le buffer TCP est plein, pour Cactus.

Temps de réponse

Le temps de réponse est mesuré en faisant une série de requête-réponse. Ici encore, nous avons fait varier la taille de la requête (qui est la même que la réponse).

Le test se déroule comme suit. Le client se connecte au serveur de la même manière que pour le test de débit. Une fois que la connexion est établie (le socket est ouvert), le client envoie un message au serveur et attend sa réponse. Lorsque le serveur reçoit la requête, il la renvoie immédiatement en guise de réponse. Le temps écoulé chez le client entre l'envoi de la requête (SEND-BYTES) et la réception de la réponse (RECEIVE-BYTES) est mesuré. Nous considèrerons le temps moyen pris pour plusieurs rêquetes-réponses consécutives.

7.4.3 Analyse

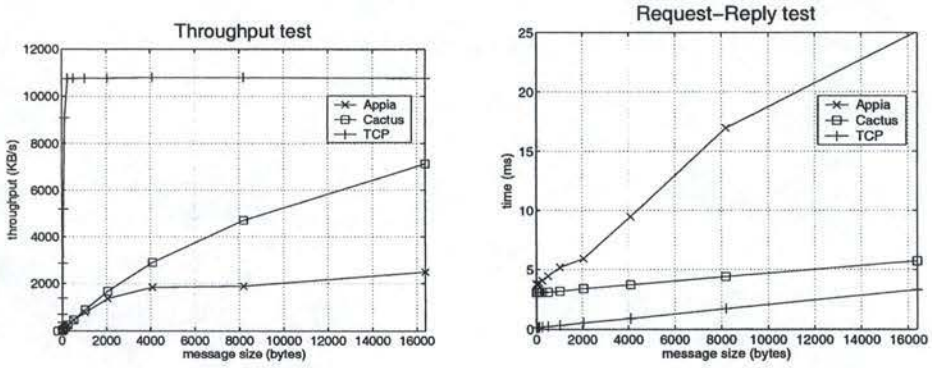
A titre de comparaison, nous avons également effectués les tests de débit et de temps de réponse sur les sockets JAVA, sur lesquels nous nous basons[5].

Les résultats de nos tests sont repris à la figure 7.3[6]. La figure 7.3(a) nous montre le débit d'Appia, Cactus et des sockets JAVA en fonction de la taille des messages, alors que la figure 7.3(b) représente l'évolution du temps de réponse de ces trois éléments en fonction du même paramètre.

Nous constatons que les débits obtenus avec Appia et Cactus, pour des messages inférieurs à 2000 octets sont forts proches et croissent linéairement, mais

[5]Notre implémentation se base en fait sur les sockets robustes ([32]), une extensions de sockets JAVA. La différence de performance est faible, voire insignifiante dans notre cas.

[6]Ces résultats sont tirés de [37].



(a) Débit.

(b) Temps de réponse.

FIG. 7.3 – Comparaison des performances.

sont dérisoires par rapport aux performances d'un simple socket[7]. Il est évident que plus un message est grand, plus le temps de transmission prendra de l'importance sur le temps de calcul induit par le traitement de ce dernier. Or, nous pouvons remarquer que pour des messages de 16Ko, Cactus est plus de trois fois plus performant qu'Appia. Ceci semblerait s'expliquer par la manière dont Appia gère les messages. En effet, Appia, voulant être indépendante de la sérialisation JAVA, fournit la classe OBJECTSMESSAGE pour éviter que le programmeur ne doive tout sérialiser à la main. Cette classe offre des primitives pour sérialiser efficacement quelques types de base, mais utilise la sérialisation JAVA pour les types complexes. Dans notre test, nous n'avons utilisé que cette dernière. Un message subira donc un certain nombre de traitements inutiles qui grèvent sérieusement les performances.

Le temps de réponse pour Appia et Cactus est, lui aussi, largement supérieur que celui des sockets JAVA. Appia affiche plus ou moins le même temps de réponse pour les messages de très petite taille que Cactus, mais est 5 fois plus lent pour des messages de grande taille. De nouveau, ce phénomène s'explique par le surcoût engendré par les frameworks face au temps de transfert des messages sur le réseau.

L'intuition que nous avons eue est confirmée par un profilage[8]. La figure 7.4 nous montre les résultats du profilage du client[9] lors d'un test du temps de réponse avec des messages de 16Ko. Nous avons découpé le temps d'exécution en cinq catégories :

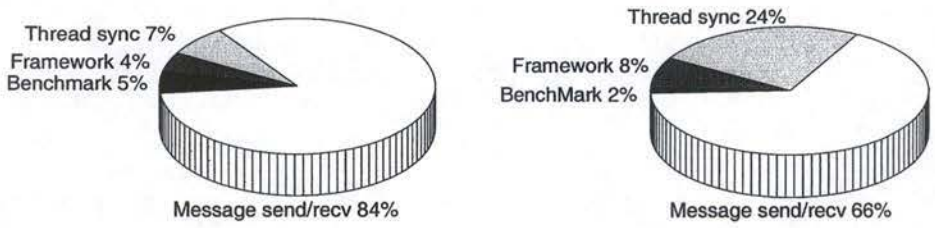
1. Attente sur le socket (non repris sur le graphique),
2. Méthodes liées à SockPerf,
3. Méthodes liées au framework (y compris l'exécution du protocole),

[7]Le débit d'un socket est principalement lié aux performances du réseau car il effectue peu de traitements sur le message

[8]Ce profilage, réalisé par Sergio Mena, est tiré de [37]

[9]Le serveur se comportant de la même manière.

4. Méthodes pour la synchronisation des threads,
5. Méthodes de sérialisation et de transmission des messages.



(a) Appia. (b) Cactus.

FIG. 7.4 – Profilage du test du temps de réponse avec des messages de 16Ko (côté client).

Nous pouvons constater que pour les deux frameworks, la plupart du temps est passée dans le traitement de messages (84% pour Appia et 66% pour Cactus). La différence entre Appia et Cactus semble donc bien venir du coût supplémentaire qu'entraîne l'utilisations des OBJECTSMESSAGES en Appia. Cela étant principalement dû au fait que nous n'utilisons pas les primitives de sérialisation efficaces que cette classe propose car la manipulation des messages se fait dans le code commun[10].

Dans les deux cas, il semble évident que le goulot d'étranglement vienne de la sérialisation JAVA qui est bien trop lourde et destinée au transfert de gros volumes d'information et non pas au transfert de simples messages.

7.5 Conclusion

Comme nous venons de le montrer, Appia et Cactus sont deux frameworks très intéressants. Ils ont chacun leurs spécificités, ce qui les rend plus ou moins adaptés à la construction de protocoles de communications de groupe modulaires.

Suite à notre expérience, nous allons proposer les bases pour la construction d'un nouveau framework plus adapté à nos besoins. Ce dernier repred les caractéristiques les plus intéressantes d'Appia et de Cactus, mais en ajoute d'autres, absentes de ces deux frameworks.

7.5.1 Modèle de composition

Nous avons vu à la section 7.3.2 qu'une composition hiérarchique entraînait quelques restrictions (difficultés d'utilisation) par rapport à un modèle coopératif.

[10]Il serait donc intéressant de modifier la manière de traiter les messgaes dans le code commun afin de déterminer le gain que cela pourrait entraîner.

Nous sommes donc enclin à opter pour un modèle de composition coopératif. Il serait tout de même intéressant de disposer, comme dans Appia, d'un moyen de vérifier la correction d'une composition.

La découpe en différents niveaux d'une composition proposée par Cactus est intéressante, mais trop restrictive. Les différents niveaux peuvent être vus comme des services de plus en plus complexes utilisables pour faciliter le travail ultérieur des programmeurs.

Notons, finalement, que la séparation entre les aspects statiques et dynamiques d'une composition est une point important pour faciliter différents contrôles de correction et optimisations.

7.5.2 Modèle d'interaction

Modèle d'interaction interne

Notre expérience nous a montré que dans les communications de groupe modulaires, les évènements point à point semblent les plus appropriés. Ce modèle d'interaction interne peut être vu comme une version simplifiée de la communication entre micro-protocoles. Il nous semble que, bien que simplifiée, cette version est tout aussi puissante que Cactus, et plus adaptée, une fois combinée aux connecteurs.

Le modèle de composition autorisant différents niveaux de composition, il serait intéressant de disposer du même modèle d'interaction entre des groupes de modules qu'entre les modules eux-mêmes. Cet aspect nous permettant de travailler indifféremment avec un module ou avec un groupe.

Avec un tel modèle d'interaction interne, nous pouvons facilement transposer le mécanisme de contrôle de correction d'Appia à ce nouveau framework.

Modèle d'interaction externe

L'interaction entre la composition et l'application ou le réseau devrait ressembler à ce que propose Appia. L'interaction entre l'application et la composition est, quant à elle, plus délicate car elle dépend du modèle d'application. Une solution serait de fournir un certain nombre d'interfaces recouvrant les modèles d'application les plus courants. Une bonne solution concernant l'interaction avec le réseau serait d'utiliser un standard comme APEX [27].

7.5.3 Modèle de concurrence

Un framework devrait offrir plusieurs modèles de concurrence pour s'adapter le mieux possible à l'environnement (un ou plusieurs processeurs). La concurrence irait d'un thread pour tout la composition (comme Appia) à un thread par module. Si nous avons des groupes de modules, il serait également envisageable de n'avoir qu'un seul thread par groupe.

Tous les modèles de concurrence proposés doivent éviter au programmeur de devoir s'occuper de tous les problèmes de synchronisation, aussi bien au sein d'un module (garanti car nous aurons au plus un thread par module), mais également entre modules (ce qui est moins évident).

De plus, une garantie FIFO sur le traitement des évènements devrait être fournie. Cet aspect s'est montré très intéressant pour notre service.

Finalement, le changement du modèle de concurrence devrait être tout à fait transparent pour les modules.

Conclusion

Tout au long de ce mémoire, nous avons montré une manière originale de faire des communications de groupe modulaires. Nous avons découpé un service complexe en modules indépendants dont toutes les interactions sont clairement définies. Ce sujet d'étude est complexe et est traité au sein du projet CRYSTALL du Laboratoire des Systèmes Répartis (LSR) de l'Ecole Polytechnique Fédérale de Lausanne (EPFL).

Pour aider les concepteurs de protocoles de communications, tels que les communications de groupe, il existe des outils spécifiques : les frameworks de composition de protocoles. Ceux-ci permettent de définir des modules et de les faire interagir entre eux pour fournir le service attendu. Il existe un grand nombre de tels frameworks, plus ou moins adaptés aux besoins du projet CRYSTALL.

Il nous a été demandé de comparer deux d'entre eux : Appia et Cactus. Ces deux frameworks sont assez représentatifs de ce qui se fait pour le moment dans le domaine. Appia peut être qualifié d'assez restrictif dans le sens où il définit de manière très stricte ce qu'il est autorisé de faire lors de la construction d'une composition. Au contraire, Cactus laisse faire beaucoup de choses en posant très peu de contraintes. Appia rend parfois le travail du programmeur assez ardu quand il désire s'éloigner des sentiers battus, tandis que Cactus l'oblige à implémenter certains services de base que l'on est enclin à attendre de tout framework.

Pour effectuer cette comparaison, nous avons implémenté un service effectuant des broadcasts fiables et ordonnés : Atomic Broadcast. Pour ce faire, nous avons défini un module pour implémenter des canaux (quasi-)fiables, un autre pour le consensus et un dernier pour le détecteur de pannes.

Désireux d'effectuer des tests de performance afin de déterminer le surcoût de chaque framework, nous avons choisi de partager toute les parties algorithmiques afin de n'avoir, comme parties individuelles, que l'interfaçage propre à chaque framework. Nous avons donc construit le code commun en nous basant sur le framework abstrait utilisé pour la description théorique de la composition.

Nous avons donc défini une architecture type pour que le code commun puisse facilement être interfacé avec Appia et Cactus. Nous avons aussi étudié les problèmes liés au passage du framework abstrait à nos deux frameworks concrets. Le concept de connecteur est en effet problèmatique, de même que le modèle de concurrence de Cactus.

Au terme de notre travail, nous sommes parvenus à mettre en évidence les

concepts clés que devraient offrir tout framework de composition de protocoles destinés aux communications de groupe. En résumé, certains points d'Appia sont plus adaptés que leur équivalent Cactus, et vice versa, mais certaines de nos attentes ne furent rencontrées par aucun de ces deux frameworks.

Notre travail n'est que le commencement. D'autres modules et d'autres compositions sont en cours de développement en se basant sur nos modules, notre architecture et les autres concepts dégagés.

Nous sommes toutefois conscients que notre travail n'est pas absent de lacunes. La plus importante se situe certainement dans notre manière assez naïve de définir et d'implémenter notre format de message dans le code commun. Nous avons pu constater que les performances médiocres de nos canaux (quasi-)fiables venaient dans l'inadéquation de la sérialisation JAVA, ce qui pourrait être en partie comblé si nos messages offraient des primitives efficaces pour sérialiser les types de base.

Bibliographie

[1] Basu A., Charron-Bost B., and Toueg S. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, Computer Science Department, October 1996.

[2] Mestafaoui A., Rajsbaum S., and Raynal M. A versatile and modular consensus protocol. Technical Report 1427, Institut en Recherche Informatique et Systèmes Aléatoires (IRISA), December 2001.

[3] Pinto A. *Appia group communication manual*, February 2001.

[4] Schiper A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3) :149–157, April 1997.

[5] Schiper A. Lecture notes for the graduate school I&C (EPFL – LSR), October 2002.

[6] Bhatti and Schlichting. A system for constructing configurable high-level protocols. aug 1995.

[7] Hiltunen Bhatti, Chiu and Schlichting. Coyote : A system for constructing fine-grain configurable communications services. *ACM Transactions on Computer Systems*, 16(4) :321–366, nov 1998.

[8] Dwork C., Lynch N., and Stockmeyer L. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2) :288–323, April 1988.

[9] Grégoire C. Consensus : Un rapide tour d'horizon. Papier écrit dans le cadre du cours d'Ingénierie des Systèmes d'Information Distribués Matière Approfondie (http ://www.info.fundp.ac.be/ ven/CISma/), Juin 2003.

[10] IBM Corporation. SockPerf : A Peer-to-Peer Socket Benchmark Used for Comparing and Measurong Java Socket Performance, 2000.

[11] Dolev D., Dwork C., and Stockmeyer L. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1) :77–97, January 1987.

[12] Cristian F. and Fetzer C. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :642–657, June 1999.

[13] Attiya H., Dwork C., Lynch N., and Stockmeyer L. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1) :122–152, January 1994.

[14] Miranda H., Pinto A., and Rodrigues L. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of The 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, Arizona, USA, April16–19 2001. IEEE Computer Society.

[15] Miranda H. and Rodrigues L. Communication support for multiple QoS requirements. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira Island, Portugal, April 1999.

[16] Miranda H. and Rodrigues L. Flexible communication support for CSCW applications. In *5th Internation Workshop on Groupware - CRIWG'99*, pages 338–342, Cacún, México, September 1999. IEEE.

[17] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http ://rfc.sunsite.dk/rfc/rfc793.html.

[18] Aspnes J. Randomized protocols for asynchronous consensus, September 2002.

[19] Hickey J., Lynch N., and van Renesse R. Specifications and proofs for ensemble layers. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1579, pages 119–133. Springer-Verlag, Berlin Germany, 1999.

[20] Vaucher J. Atomic broadcast. Projet de semestre, February 2003.

[21] Aguilera M. K. and Toueg S. Failure detection and randomization : A hybrid approach to solve consensus. *SIAM J. Comput.*, 28(3) :890–903, 1998.

[22] Kihlstrom K., Moser L, and Melliar-Smith P. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4) :371–406, November 2001.

[23] Lamport L. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2) :133–169, 1998.

[24] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press.* 1995.

[25] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.

[26] Fischer M., Lynch N., and Patterson M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32 :374–382, April 1985.

[27] Rose M., Klyne G., and Crocker D. The application exchange core. RFC 3340. The Internet Society (IETF), 2002.

[28] Hayashibara N., Urbán P., Schiper A., and Katayama T. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. Technical Report IC-2002-61, Ecole polytechnique Fédérale de Lausanne (EPFL), Faculté d'Informatique et Communications (I&C), 2002.

[29] Hutchinson N. and Peterson L. The x-kernel : An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1) :64–76, 1991.

[30] Urbán P., Défago X., and Schiper A. Neko : A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6) :981–997, November 2002.

[31] Verissimo P. and Almeida C. Quasi-synchronism : a step away from the traditional fault-tolerant real-time system models. *IEEE TCOS bulletin*, 7(4) :35–39, December 1995.

[32] Ekwall R., Urbán P., and Schiper A. Robust TCP Connections for Fault Tolerant Computing. *Journal of Information Science and Engineering*, 19(3) :503–516, May 2003.

[33] Guerraoui R. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 87–100, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.

[34] Guerraoui R. and Schiper A. Consensus : the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, October 1997. IEEE Computer Society Press.

[35] Oliviera R., Guerraoui R., and Schiper A. Consensus in the crash-recovery model. Technical Report TR-97/239, EPFL, Dept d'Informatique, July 1997.

[36] Mena S. Configuration and Extension of Group Communication Protocols, July 2001.

[37] Mena S., Cuvellier X., Grégoire C., and Schiper A. Appia *vs.* Cactus : Comparing protocol composition frameworks. In *22nd Symposium on Reliable Distributed Systems. Florence, Italy*, October 2003.

[38] Chandra T. and Toueg S. Unreliable failure detector for reliable distributed systems. *Communications of the ACM*, 43(2) :225–267, 1996.

[39] Chandra T. and Toueg S. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4) :685–722, 1996.

[40] Défago X., Schiper A., and Sergent N. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.

Annexe A

Conventions de représentation

A.1 Conventions des machines à états

Les opérateurs logiques :

&	La conjonction
|	La disjonction
!	La négation
=	L'égalité
!=	L'inégalité

Notation des évènements :

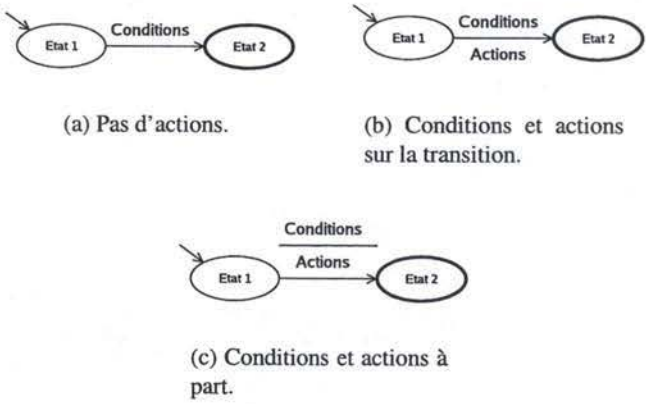
- $!evt(p_1, p_2, \ldots, p_n)$ indique le déclenchement d'un évènement de type evt avec la séquence de paramètres $(p_1, p_2, \ldots, p_n)$.
- $?evt(p_1, p_2, \ldots, p_n)$ indique la réception d'un évènement de type evt avec la séquence de paramètres $(p_1, p_2, \ldots, p_n)$.

États et transitions :

- Les transitions comportent deux informations : la(les) condition(s) nécessaire(s) pour passer de l'état actuel au suivant et la(les) action(s) à accomplir lors de ce passage. Lorsque aucune action n'est présente, nous utilisons la notation de la figure A.1(a). Dans le cas contraire, soit nous utilisons la notation de la figure A.1(b), soit la notation de la figure A.1(c).
- Lorsque plusieurs états initiaux sont présents, nous rentrons dans la machine à états lorsque les conditions d'une des transitions initiales sont obtenues.
- Les états dessinés en gras représentent les états finaux.

A.2 Conventions du pseudo-code

Les conventions utilisées dans les pseudo-codes sont standards pour la plupart. Quelques conventions le sont moins et vont être expliquées.



(a) Pas d'actions.

(b) Conditions et actions sur la transition.

(c) Conditions et actions à part.

FIG. A.1 – Conventions de représentation des machines à états.

Paradigmes utilisés

Dans le cadre du projet, nous avons travaillé dans deux paradigmes : le paradigme orienté objet et le paradigme évènementiel. Le paradigme orienté objet étant bien connu, nous ne le décrirons pas d'avantage. Nous allons nous concentrer sur le paradigme évènementiel.

Dans ce dernier, nous avons deux entités importantes :

- **Les évènements** : Indiquent les évènements qui peuvent être déclenchés.
- **Les handlers (Upon)** : Portion de code éxécutée lors de la réception d'un évènement.

Dans ce paradigme, une méthode spéciale est disponible : la méthode **trigger**. Elle permet de déclencher un évènement et de lui associé des paramètres éventuels. Une fois déclenché, tous les handlers associé à cet évènement seront exécutés dans un ordre non-déterministe et les paramètres éventuels sont passés en argument. Nous pouvons comparer le déclenchement d'un évènements comme un appel de plusieurs méthodes : nous appelons chaque handlers associés au handler déclenché.

Le pseudo-code utilisé va se présenter comme une succession de handlers. Le code de chacun de ces handlers pourra utiliser tous les concepts de la programmation orientée objet mais également déclencher de nouveaux évènements par la méthode **trigger**.

Manipulation des ensembles

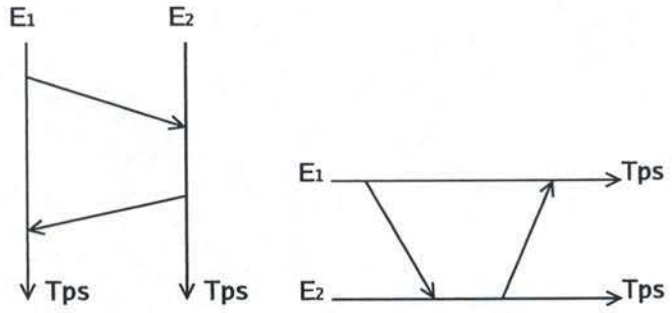
Dans le pseudo-code, un ensemble est représenté par un tuple. Seules quelques éléments du tuple identifient celui-ci dans l'ensemble.
Supposons un ensemble E contenant trois éléments (el_1, el_2, el_3) et identifié par el_1.

- $E(e_1) = \perp$ indique qu'aucun tuple identifié par el n'est présent dans l'ensemble.

- $E(e_1) = (e_2, e_3)$ indique que l'ensemble possède un tuple identifié par e_1. Le couple e_2 et e_3 désignent respectivement la valeur de el_2 et de el_3.

A.3 Conventions des diagrammes de séquence

Les diagrammes de séquence seront représentés soit de manière horizontale, soit de manière verticale. Ils décrivent l'évolution des interactions entre plusieurs entités au cours du temps. La figure A.2 présente les deux cas possibles.



(a) Diagramme de séquence horizontal.

(b) Diagramme de séquence vertical.

FIG. A.2 – Diagramme de séquence.

E_i désigne une entité et une flèche représente une interaction entre les deux entités.

A.3.1 Diagrammes de séquence des protocoles réseaux

Ils décrivent l'évolution des messages échangés entre deux applications ou processus au cours du temps. Sur la figure A.3, nous distinguons trois types de flèches :

1. Ce type de flèche fait suite au déclenchement d'un évènement.
2. Ce type de flèche désigne un message transitant sur le réseau.
3. Ce type de flèche désigne un évènement déclenché.

En règle général, le déclenchement d'un évènement d'une application A donne lieu à un ou plusieurs échanges de messages sur le réseau entre A et B. Cet échange déclenchera un évènement chez l'application B.

A.3.2 Diagrammes de séquence de threads

Ils décrivent l'évolution de plusieurs threads et leurs interactions au cours du temps. La figure A.4 illustre un exemple et décrit dans la légende les concepts

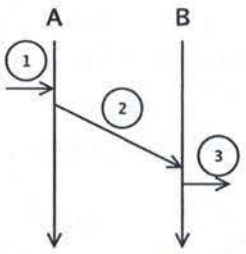


FIG. A.3 – Conventions des diagrammes de séquence pour protocoles réseaux.

utilisés.

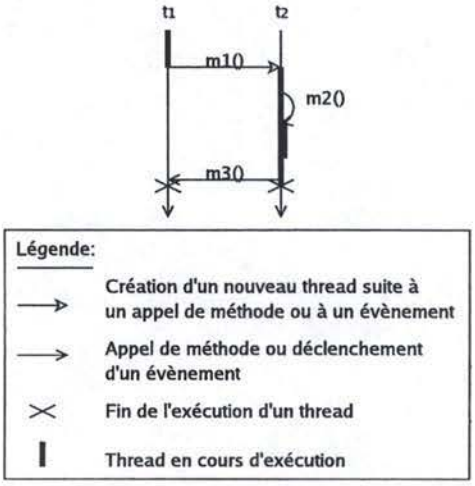


FIG. A.4 – Conventions des diagrammes de séquence entre thread.

L'exemple de la figure décrit deux thread t_1 et t_2. t_1 en cours d'exécution déclenche une méthode m_1 qui crée un nouveau thread t_2. t_1 suspend son exécution et t_2 débute la sienne. Ensuite t_2 appelle une méthode m_2 qui est exécutée par lui-même. Finalement t_2 appelle une méthode m_3 qui provoque la terminaison du thread t_1. t_2 arrête finalement son exécution. s

Annexe B

Méthodologie Appia

En nous basant sur [36], nous allons présenter une méthodologie pour développer des piles de communication avec Appia.

Dans un premier temps, il faut définir le service que la pile doit rendre. Il faudra ensuite choisir les couches composant cette pile. Il est conseillé d'essayer de réutiliser au maximum les modules déjà existants, car la réutilisation est l'essence même de tout framework. Il ne sera cependant certainement pas possible de créer un nouveau service uniquement avec d'anciens modules, ne serait-ce que pour l'interfaçage avec l'application. Nous allons donc décrire une manière de concevoir des modules, ensuite nous verrons les étapes nécessaires pour assembler tous ces éléments pour avoir une pile fonctionnelle.

1. ***Conception d'un module :*** Tout module se compose de deux aspects : statique et dynamique. Il faudra donc écrire deux classes JAVA. L'une étendant LAYER et l'autre SESSION.

   Pour la couche, il suffit juste de spécifier trois tableaux pour décrire les évènements fournis, acceptés et requis, respectivement $evProvide$, $evAccept$, $evRequire$. Ces tableaux sont de type CLASS[]. Il ne faut pas oublier de préciser que la couche accepte les évènements CHANNELINIT, CHANNELCLOSE (et DEBUG si vous le gérez).

   En ce qui concerne la session, Il "suffit" juste d'implémenter la méthode $handle$(EVENT e) par laquelle passera tout évènement déclaré comme accepté. Il faudra faire soi-même la distinction des évènements selon leur type. Pour ce faire, il est courant d'utiliser l'opérateur $instanceof$ fournit par JAVA. La structure des conditions utilisées pour ce faire devra être pensée soigneusement pour optimiser le code et ne pas avoir de conflit de types[1].

   Notons qu'il n'est pas nécessaire de synchroniser les structures de donnée créées par la session car elle ne sera exécutée que par un seul thread.

[1] Un conflit de type peut survenir car une couche peut accepté des évènements dont l'un est descendant de l'autre. L'opérateur $instanceof$ appliqué sur le fils en comparaison avec le type du père renverra **true**.

2. ***Définition et création d'une QoS :*** Une QoS est représentée par un tableau de LAYER. La premier élément de la QoS contenant la couche la plus basse, et le dernier élément, la couche la plus haute.

   Cette QoS ne restera pas un simple tableau. Appia va l'incorporer dans un objet de type QOS comportant un nom et contenant le tableau de couches. En faisant cela, Appia va vérifier que la QoS que l'on tente de construire est correcte, i.e., tous les évènements acceptés par les couches sont bien fournis par d'autres couches.

3. ***Instantiation de la QoS :*** Chaque QoS doit maintenant être instanciée pour donner un canal. Cela se fait simplement en appelant la méthode $createUnboundChannel$ (STRING $channelId$). L'argument permettra de distinguer les différents canaux d'une pile.

   Le canal ainsi créé est non lié, i.e., aucune session n'est affectée à aucun niveau. Ceci est fait pour permettre de lier à la main les sessions de manière très flexible afin de former la structure en forme de diamant.

4. ***Liaison des Sessions :*** Une fois que nous disposons de tous les canaux devant constituer notre pile, nous pouvons créer les sessions qui nous intéressent et les lier au bon endroit. Pour ce faire, nous disposons de CHANNELCURSOR, une sorte de pointeur nous permettant de parcourir un canal pour désigner l'endroit où nous désirons lier une session. Une même session peut évidemment être liée à plusieurs canaux pour former la structure en forme de diamant.

   Cette méthode étant relativement lourde à mettre en oeuvre, nous ne l'utiliserons que pour les sessions "particulières", i.e., celles qui seront liées à plusieurs canaux, ou celles nécessitant des paramètres pour les instancier. Appia se chargera du reste lors de l'appel de la méthode $start()$ sur les différents canaux. Cette méthode a un rôle important. En effet, elle va vérifier que toutes les sessions liées le soient avec la bonne couche, créer et lier les sessions manquantes et envoyer l'évènement d'initialisation (CHANNELINIT). Dès que la méthode $start()$ a été appelée, le canal est donc prêt à accepter et traiter des évènements.

5. ***Démarrage de l'*EVENTSCHEDULER *:*** Une fois que les différents canaux de la pile sont initialisés, il ne reste plus qu'à "lancer" la pile en démarrant un thread qui se chargera du traitement des évènements (via l'EVENTSCHEDULER) et de l'exécution des handlers.

Annexe C

Arbres d'exécutions atomiques

Les arbres d'exécution indépendants permettent d'assurer l'exécution indépendante d'un enchaînement de handlers et d'évènements prédéfinis.

Convention de représentation

Nous allons montrer une manière de représenter de manière schématique ces arbres d'exécution en se basant sur les handlers et les évènements contenus dans un protocole composite. Nous avons dit qu'un micro-protocole est composé de plusieurs handlers. Nous prenons tous les handlers de tous les micro-protocoles d'un composite, soit h_1, h_2, ..., h_n. Ces handlers vont former les noeuds d'un graphe. Si un handler h_i déclenche un évènement e quelconque qui est lié à un handler h_j, nous construisons un arc entre les deux noeud h_i et h_j du graphe. Si l'évènement déclenché est synchrone, le trait de la flèche est pleine. S'il est asynchrone, le trait est pointillé. La figure C.1 présente un tel exemple de graphe.

Pour décrire ce schéma, prenons par exemple, le cas des arbre A_1 et A_2. Lorsque le code du handle h_1 est exécuté, celui-ci invoque un évènement e_1. Deux handlers sont associés à cet évènement, h_2 et h_3 (car l'appel est synchrone). Lors de l'invocation de e_1 le thread de h_1 interrompt son exécution et exécute le code de h_2 et de h_3. L'exécution de h_3 donne lieu à h_4, h_5 et ensuite h_6 par le même procédé. Pendant l'exécution de h_2, celui-ci lève de manière asynchrone l'évènement e_2. Un nouveau thread est dès lors créé et il va exécuter le seul handler associé à e_2 (h_7) ; un nouveau thread est créé, un nouvel arbre d'exécution indépendant débute. Finalement h_7 déclenche un évènement e_7 de manière synchrone. Nous avons donc deux arbres d'exécution indépendants, l'arbre A_1, composé des handlers h_1 à h_6, et l'arbre A_2 composé de h_7, h_8 et h_9.

Objectif

L'objectif va être d'exécuter l'ensemble de ces arbres de manière indépendants, lorsqu'un arbre commencera son exécution, aucun autre ne pourra commencer son exécution avant la fin du premier.

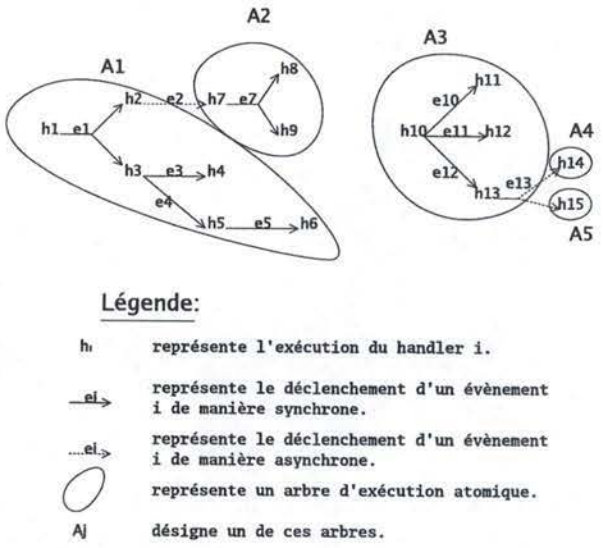


FIG. C.1 – Exemple d'arbres d'exécution indépendants.

Définition

Un arbre d'exécution se définit comme un enchaînement d'handlers et d'évènements exécuté par un thread unique.

Lorsqu'un handler déclenche un évènement synchrone, c'est le même thread qui déclenchera les handlers à l'écoute de cet évènement. Tous ces évènements et handlers exécutés par le même thread font donc partie du même arbre d'exécution.

Lorsqu'un handler déclenche un évènement asynchrone, un nouveau thread est créé qui exécutera les handlers à l'écoute de cet évènement. Ce n'est plus le même thread qui exécutera ces handlers, ceux-ci feront dès lors partie d'un nouvel arbre d'exécution.

Exécution indépendante de tous les arbres

Nous avons vu que chaque arbre d'exécution indépendant était exécuté par un thread unique. Pour garantir l'indépendance, il nous suffit de bloquer l'exécution de tous les autres arbres pendant l'exécution d'un arbre.

Nous pouvons utiliser des mécanismes de synchronisation. Il suffit de synchroniser les threads associés à chacun des arbres entre-eux. De cette manière lorsqu'un thread t_i associé à un arbre A_i commence son exécution, aucun autre thread t_j associé à un autre arbre A_j ne peut débuter son exécution tant que A_i n'est pas terminé.

Cette solution permet de limiter complètement les accès concurrents en ne permettant que l'exécution d'un thread à la fois.