Tolérance aux pannes pour un modèle à objets actifs

Christian Delbé
Projet OASIS, INRIA, CNRS-I3S, Université de Nice Sophia-Antipolis

Encadrants : Denis Caromel & Françoise Baude

Malgré la multitude de protocoles de tolérance aux pannes par points de reprise développés depuis une trentaine d'années, il n'en n'existe pas, à notre connaissance, adapté à la particularité suivante : la non-préemptivité sur l'exécution des processus. Cette caractéristique est pourtant intrinsèque à certains environnements comme la plate-forme Java, et se retrouve dans de nombreux intergiciels de communication.

Nous proposons donc, dans le cadre d'un modèle à objets actifs, un protocole de tolérance aux pannes par points de reprise induits par messages adapté à la non-préemptivité sur les processus. A la différence de nombreux protocoles existants, ce protocole assure la cohérence forte des lignes de recouvrement formées, et permet une reprise complétement asynchrone du système réparti en cas de panne.

DEA RSD

Mars-Juin 2003

Remerciements

Je remercie Denis Caromel et Françoise Baude qui m'ont encadré tout au long de ce stage et dont les conseils m'ont guidé dans ce travail.

Je tiens à remercier toute l'équipe OASIS pour l'accueil chaleureux qui m'a été fait, et en particulier Fabrice pour m'avoir "supporté" dans son bureau, et Romain pour les moments de décompression parfois nécessaires.

Je suis extrêmement reconnaissant envers Ludovic, sans qui ce travail n'aurait peut-être pas abouti.

Enfin, ma pensée la plus personnelle va à Tiffany, qui a su me soutenir durant cette période.

Table des matières

1 Introduction **6**
1.1 La tolérance aux pannes dans les systèmes distribués 6
1.2 Objectifs 6

I Contexte et analyse du problème **8**

1 Tolérance aux pannes par points de reprises **9**
1.1 Cohérence d'un état global 9
1.1.1 Définitions 9
1.1.2 Caractérisation de la cohérence 11
1.2 Cohérence forte d'un état global 13
1.3 Les différents protocoles 14
1.3.1 Indépendant 14
1.3.2 Synchronisé 14
1.3.3 Induit par message 14
1.4 Analyse et comparaison des différentes approches 15
1.4.1 Degré de tolérance aux fautes 15
1.4.2 Surcoût pendant l'exécution 16
1.4.3 Nombre de retours arrière moyen 16
1.4.4 Surcoût de stockage et ramassage 16

2 Tolérance aux pannes par journalisation des messages **17**
2.1 Déterminisme par morceaux 17
2.2 Les différents protocoles 18
2.2.1 Journalisation pessimiste 18
2.2.2 Journalisation optimiste 18
2.3 Analyse et comparaison des différentes approches 18
2.3.1 Nombre de processus à reprendre 19
2.3.2 Surcoût 19
2.3.3 Taille des messages 20

3 Synthèse **21**
3.1 Fréquence des pannes 21
3.2 Nombre de processus 21
3.3 Taux de communication, maillage 22
3.4 Non-déterminisme 22
3.5 Communication avec le monde extérieur 22

4 Objets actifs en Java : ProActive **24**
4.1 Présentation 24
4.2 Analyse 25
4.2.1 Java 25
4.2.2 Activité des objets 25
4.2.3 Protocole méta-objet et réification 26
4.2.4 Communications avec Rendez-Vous 26
4.2.5 Notations 27

II Solution proposée **29**

1 Principes **30**
1.1 Description 30
1.1.1 Modèle 30
1.1.2 Problème de la non-préemptivité 30
1.1.3 Cohérence forte 31
1.1.4 Points de reprise multiples et points de reprise impossibles 32
1.1.5 Equivalence des exécutions 33
1.1.6 Historique 34
1.1.7 Reprise après panne 36
1.2 Synthèse 38

2 Formalisation **40**
2.1 Définitions et notations 40
2.2 Protocoles 42
2.2.1 Protocole de points de reprise 42
2.2.2 Protocole de reprise 42
2.2.3 Clôture des historiques 44

3 Mise en oeuvre et expérimentations **46**
3.1 Eléments de mise en oeuvre 46
3.1.1 Support de stockage stable 46
3.1.2 Processus de reprise 47
3.1.3 Serveur de localisation 47
3.2 Implémentation et évaluation des performances 47
3.2.1 Choix d'implémentation 47

3.2.2 Evaluation du surcoût pendant une exécution sans panne 48

4 Conclusion et perspectives **52**

Table des figures

1.1 Etat global incohérent 10
1.2 Etat global cohérent 11
1.3 Effet domino 11
1.4 Chemin "zig-zag" entre C_P et C_R 13

2.1 Le processus Q doit lui aussi reprendre si $m4$ n'a pas été journalisé 19

4.1 Objets actifs communiquants 27
4.2 Objet actif prenant un point de reprise 28

1.1 La requête Q_A est exclue de la queue de requêtes lors de la prise du point $n+1$ sur j 31
1.2 La requête Q_A est enregistrée pour réémission dans le point de reprise $n+1$ de l'objet actif i 32
1.3 L'objet actif j doit faire 2 points de reprise d'affilé 33
1.4 Une deuxième exécution à partir de $n+1$ peut être différente 34
1.5 La réception sur j de Q_C doit précéder celle de Q_A 35
1.6 Les clôtures d'historique ($\diamondsuit$) sur j et k ne sont pas cohérentes entre elles 37

2.1 Protocole de points de reprise 43
2.2 Protocole de reprise 44

3.1 Application de test : le crible d'Erathostène 48
3.2 Surcoût à l'exécution (4 esclaves, $TTC\_INIT = 100$ sec) 50
3.3 Surcoût à l'exécution (Recherche du 3500ème nombre premier, $TTC\_INIT = 100$ sec) 51

Chapitre 1

Introduction

1.1 La tolérance aux pannes dans les systèmes distribués

La nature des ressources utilisées dans les systèmes distribués a considérablement changé depuis quelques années : on voit aujourd'hui des réseaux de stations de travail utilisés pour effectuer des calculs longs et complexes. Ce changement est la conséquence d'une réalité économique : la récupération des cycles CPU inutilisés sur les machines d'un réseau d'entreprise.

Dans ce contexte, la tolérance aux pannes des systèmes répartis devient un élément incoutournable. En effet, les noeuds de ces systèmes peuvent donc être maintenant des stations de travail, dont les réactions sont dépendantes d'un matériel souvent peu fiable, et d'utilisateurs qui peuvent décider à tout moment de se déconnecter du réseau, ou bien d'éteindre leur machine. Un intergiciel proposant l'exécution de calculs sur ce type de système se doit alors d'intégrer, encore plus qu'un autre, un mécanisme de tolérance aux fautes.

Cependant, très peu d'intergiciels proposent cette fonctionnalité, en partie à cause du surcoût induit par les techniques usuelles, mais surtout à cause de la complexité de mise en oeuvre des techniques. En effet, une grande majorité des protocoles de tolérance aux pannes qui ont été développés depuis une trentaine d'années restent souvent théoriques, et ne tiennent pas forcément compte des caractéristiques des systèmes physiques, mais aussi des langages et des plate-formes utilisés.

1.2 Objectifs

Mon stage au sein de l'équipe OASIS (projet INRIA-CNRS/I3S-Université de Nice Sophia-Antipolis) a pour but de proposer un protocole de tolérance aux pannes pour un modèle à objets actifs. Nous avons developpé un *protocole de points de reprise induits par messages*, que nous avons implémenté dans ProActive, une bibliothèque Java de programmation parallèle et distribuée.

Ce rapport présente dans une première partie les deux approches fondamentales de la tolérance aux pannes pour les systèmes distribués : l'utilisation de points de reprise et la journalisation des messages. Une synthèse comparative de ces approches, par rapport aux caractéristiques des applications considérées, est proposée. Nous présentons ensuite le modèle à objets actifs de ProActive, et détaillons les propriétés qui auront un impact sur la conception d'un protocole de tolérance aux pannes adapté.

Cette analyse nous permet dans une deuxième partie de justifier le protocole proposé, puis de le formaliser. Enfin, après avoir décrit une architecture de mise en oeuvre du protocole et donné les premières évaluations de performances, nous concluons sur les perspectives d'évolution.

première partie

Contexte et analyse du problème

Chapitre 1

Tolérance aux pannes par points de reprises

La tolérance aux pannes par points de reprise (ou *checkpointing*) utilise le fait qu'un processus en cours d'exécution va être capable à tout moment de prendre un enregistrement complet de son état, ou *point de reprise*. Cet enregistrement sera utilisé en cas de panne comme point de reprise, ce qui évite au processus de repartir depuis le début de son exécution.

Cet enregistrement doit correspondre à un état cohérent [1], c'est à dire un état par lequel passe un processus lors de l'exécution normale. Si cette notion de cohérence paraît triviale lorsque l'on considère un seul processus qui enregistre régulièrement son état, le problème devient complexe lorsqu'on veut obtenir un état cohérent d'un *ensemble de processus communiquants*. C'est principalement cette problématique que tentent d'adresser les protocoles de tolérances aux pannes par points de reprise : assurer la cohérence de l'ensemble des états de chaque processus.

1.1 Cohérence d'un état global

1.1.1 Définitions

Considérons un système de n processus communiquants : nous allons montrer qu'un ensemble d'enregistrements locaux, représentant chacun un état cohérent d'un processus, ne représente pas toujours un état cohérent *de l'ensemble du système*.

Définition 1.1.1 *Un* ***état global*** *d'un système réparti est un ensemble d'états locaux dans lequel on trouve un enregistrement d'état local par processus.*

Dans la figure 1.1, nous montrons trois processus ayant chacun pris un enregistrement local de leur état ; pourtant, l'ensemble de ces trois points ne représente pas un état global cohérent. En effet, un état cohérent est un état par lequel peut passer une exécution normale. Or, on voit ici

1. nous considérerons toujours qu'un point de reprise pris par un processus unique représente un état cohérent de celui-ci

que si chaque processus repart depuis son point de reprise, le message m_3 a déjà été reçu par le processus R, *alors qu'il n'a pas encore été envoyé par Q !* On parle alors de message orphelin.

Définition 1.1.2 *Un **message orphelin** par rapport à un état global est un message qui a été envoyé aprés un enregistrement d'état local appartenant à cet état global et reçu avant un enregistrement d'état local appartenant à cet état global.*

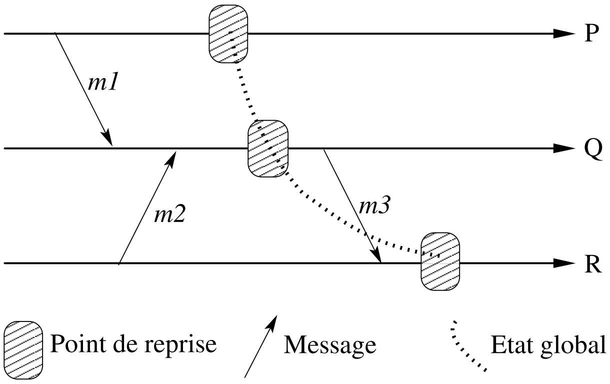

FIG. 1.1 – *Etat global incohérent*

Cette notion de cohérence a été formalisée par Chandy et Lamport dans [1] :

Définition 1.1.3 *Un état global est **cohérent** si il n'existe pas de message orphelin par rapport à cet état.*

En d'autres termes, si un point de reprise sur le processus P a enregistré la réception d'un message provenant du processus Q, alors il faut que le point de reprise sur Q appartenant au même état global ait enregistré l'emission de ce message. On note qu'un message peut avoir été envoyé mais non reçu (message m_3 sur la figure 1.2) ; l'état global reste cohérent, au sens de [1]. Ce message qui sera perdu en cas de reprise du système est un message *en transit*.

Définition 1.1.4 *Un **message en transit** par rapport à un état global est un message qui a été envoyé avant un enregistrement d'état local appartenant à cet état globalet reçu après un enregistrement d'état local appartenant à cet état global.*

Nous verrons par la suite une notion de cohérence plus stricte qui interdit aussi ce type de message.

De manière générale, le but des protocoles de tolérance aux pannes par points de reprise est de s'assurer que les processus prennent régulièrement des points de reprises locaux, et que ces points forment des états globaux cohérents.

Définition 1.1.5 *La **ligne de recouvrement** est l'état global cohérent le plus récent.*

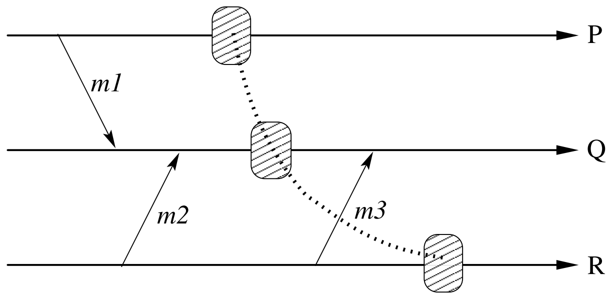

FIG. 1.2 – *Etat global cohérent*

Cette ligne de recouvrement doit toujours être assez récente, de façon à minimiser le *retour arrière* en cas de reprise, c'est à dire le nombre de points de reprise pris entre le dernier pris et celui utilisé pour la reprise. Nous verrons cependant que certains protocoles ne poussent pas à la création d'une ligne de recouvrement durant l'exécution, mais recherchent cette ligne dans l'ensemble des états globaux formés, après qu'une panne soit survenue dans le système. Ces protocoles sont fortement sujets à ce qu'on appelle *l'effet domino* [2].

L'effet domino est caractérisé par une cascade de retours arrière lors de la reprise du système après une panne. Considérons la figure 1.3 : lors de la panne du processus P (survenant au point $\times$), le système va tenter de reprendre depuis l'état global le plus récent, c'est à dire celui formé par les points de reprise C_P^3,C_Q^3 et C_R^3. A cause du message orphelin m_6, cet état n'est pas cohérent : on va donc faire un retour arrière sur R, jusqu'au point C_R^2. Mais l'état global ainsi formé n'est toujours pas cohérent. De la même manière, le système va devoir reculer jusqu'à la ligne de recouvrement (formée de C_P^1,C_Q^1 et C_R^1) et donc perdre une grande quantité de travail.

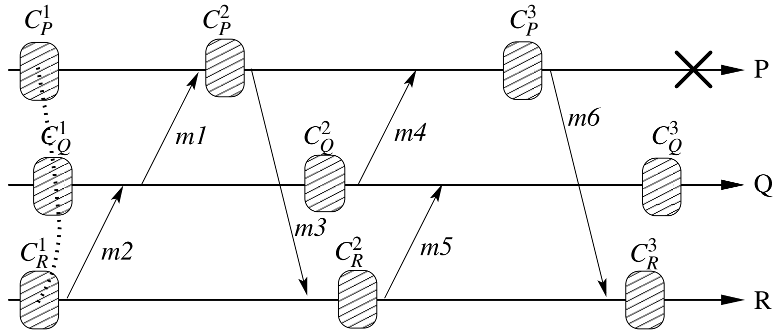

FIG. 1.3 – *Effet domino*

1.1.2 Caractérisation de la cohérence

Nous donnons ici certaines conditions pour qu'un état global soit cohérent au sens de [1]. On distingue deux manières de caractériser la cohérence. La première est basée sur la relation *happened-before* de Lamport [3], que l'on définit ainsi :

Définition 1.1.6 *L'événement a précède l'événement b (ou b dépend causalement de a) si*
- *si a et b surviennent sur le même processus, alors a survient avant b,*
- *si a est l'envoi d'un message par un processus, et que b est la réception de ce message,*
- *si a précède l'évenement e et e précède b.*

Cette relation de causalité entre évenements permet de définir une condition nécessaire pour qu'un état global soit cohérent.

Propriété 1.1.1 *Si deux points de reprise appartiennent tous les deux à un état global cohérent, alors il n'existe pas de dépendance causale entre eux.*

En effet, si il existe une dépendance causale entre deux points de reprise du même état global, alors il existe dans cet état global deux point de reprise C_P sur le processus P et C_Q sur le processus Q qui sont en dépendance causale *directe*, c'est à dire qu'un message a été envoyé *après* la prise de C_P et reçu *avant* la prise de C_Q. Ce message est donc un message orphelin, et l'état global considéré n'est pas cohérent.

De cette manière, on peut vérifier que chaque communication directe (point-à-point) n'entraîne pas une dépendance causale entre un point de reprise de l'émetteur et un point de reprise du récepteur, et prendre les mesures nécessaires si c'est le cas, comme par exemple prendre un point de reprise avant l'émission ou la réception.

Cependant, cette règle n'est utilisable que si on considère les communications directes entre processus : il faut qu'elle soit vraie deux à deux pour tous les processus ayant eu une communication directe. Il existe cependant une autre méthode pour caractériser la cohérence d'un état global, qui considère aussi les relations indirectes entre processus. Elle a été proposée par Netzer et Xu dans [4] et est basée sur la notion de chemin "zig-zag", une généralisation de la relation de causalité de Lamport.

Comme on peut le voir dans la figure 1.4, C_P^1 n'est pas en relation causale avec C_R^1, pourtant ils ne peuvent pas appartenir tous deux à un même état global cohérent, puisque soit le message $m1$ dans l'état global $[C_P^1, C_Q^2, C_R^1]$, soit le message $m2$ dans l'état global $[C_P^1, C_Q^1, C_R^1]$ serait orphelin.

Cette relation qui existe entre C_P et C_R se nomme chemin "zig-zag". C'est une extension de la relation de causalité de Lamport et peut se définir comme :

Définition 1.1.7 *Il existe un chemin "zig-zag" entre deux points de reprise C_P et C_Q si et seulement si il existe des messages m_1, m_2, ..., m_n tels que :*
- *m_1 est émis par P après la prise de C_P*
- *si m_k ($1 \leq k \leq n$) est reçu par le processus R, alors le message m_{k+1} est envoyé par R dans le même intervalle de points de reprise (ou dans un intervalle suivant). m_{k+1} peut être envoyé aussi bien avant que après la réception de m_k*
- *m_n est reçu par Q avant la prise de C_Q*

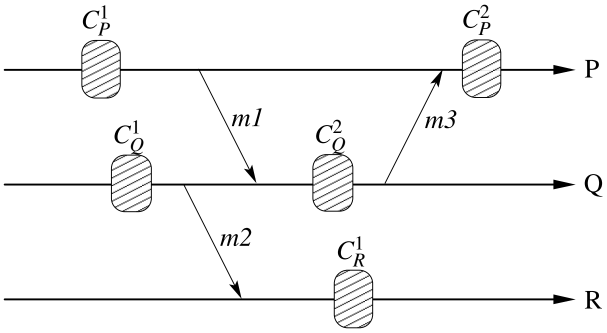

FIG. 1.4 – *Chemin "zig-zag" entre C_P et C_R*

Lorsqu'il existe un chemin "zig-zag" entre un point de reprise et lui-même, on parle alors d'un cycle "zig-zag". De fait, un point de reprise pris dans un cycle "zig-zag" ne peut bien-sûr appartenir à aucun état global cohérent, et est donc inutile. Par exemple, dans la figure 1.4, le point C_Q^2 est pris dans un cycle "zig-zag" à cause des messages m_1 et m_3.

Un grand nombre de protocoles de tolérance aux pannes par points de reprise utilisent ces notions de chemin et de cycle "zig-zag" pour déterminer les lignes de recouvrement, ou pour savoir si il est utile à un instant donné de prendre un point de reprise [5, 6].

1.2 Cohérence forte d'un état global

Comme on l'a vu dans 1.1.1, la cohérence au sens de [1] n'interdit pas les messages en transit : on parle alors de cohérence *simple*. De nombreux protocoles assurent des reprises depuis des états dont la cohérence est simple ; la gestion des messages en transit est alors laissée à un autre protocole[2], ou alors l'application doit pouvoir supporter la perte de message. Ce sont cependant des hypothèses très contraignantes, et c'est pour cela que l'on introduit la notion de cohérence *forte*.

Définition 1.2.1 *La* ***cohérence forte*** *étend la cohérence simple, avec la condition supplémentaire qu'il n'existe pas de messages en transit dans l'état global considéré.*

Hélary, Netzer et Raynal montrent dans [7] que si l'absence de chemin "zig-zag" entre les points de reprise assure l'absence de message orphelins, alors l'absence de chemin "zig-zag" *inverse* assure l'absence de messages en transit. Un chemin "zig-zag" inverse est un chemin "zig-zag" que l'on trouverait dans l'exécution si on inversait tout les messages de sorte que les émetteurs deviennent les récepteurs, et réciproquement. En effet, on peut constater qu'un message en transit est le cas inverse, ou *dual*, d'un message orphelin : l'un n'a plus d'émetteur et

2. on peut par exemple enregistrer tous les messages en transit de manière à les réémettre en cas de reprise du système [8]

l'autre n'a plus de récepteur.

Cette propriété nous permet d'assurer une cohérence forte à partir d'un protocole assurant la cohérence faible : si on est capable d'appliquer dans le même temps ce protocole *et* son dual, on empêche la présence de message orphelin *et* de message en transit : on assure alors la cohérence forte des états globaux formés [8].

1.3 Les différents protocoles

Nous présentons ici les différentes familles de protocoles de tolérance aux pannes par points de reprise : indépendants, synchronisés et induits par messages.

1.3.1 Indépendant

Cette méthode consiste à faire prendre aux différents processus du système des points de reprise de manière totalement indépendante. La cohérence des états globaux formés n'est donc pas assurée et c'est lors d'une reprise que le protocole recherche une ligne de recouvrement parmi tous les états globaux formés. Cette recherche s'effectue par la création d'un graphe de dépendance entres les points de reprise locaux [9], idée basée sur la dépendance causale de Lamport. Cette méthode est donc fortement sujette à l'effet domino, puisque rien n'assure lors de la prise d'un point de reprise que celui-ci pourra faire partie d'un état global cohérent.

Elle peut aussi être utilisée avec un protocole supplémentaire : on verra par exemple dans le chapitre 2 que les protocoles de journalisation de messages utilisent en plus des points de reprise indépendants, pour limiter le retour arrière en cas de panne ; dans ce cas, la cohérence des états globaux n'est pas demandée, et ce type de protocole est donc bien adapté.

1.3.2 Synchronisé

Les points de reprises sont réalisés de manière coordonnée sur tous les processus de l´exécution, de façon à assurer que l'état global résultant soit cohérent. Cette synchronisation peut être :

– bloquante : la cohérence de l'état global est assurée par le fait que tous les processus sont stoppés lors de la prise de cet état global.
– non bloquante : la cohérence est assurée par des messages "balais", qui vident les canaux de communication afin d'éviter les messages orphelins. Ce protocole requiert des canaux de communications FIFO (*First In First Out*).

Dans les deux cas, le protocole nécessite l'utilisation de messages additionnels.

1.3.3 Induit par message

La synchronisation se fait ici de manière "paresseuse", en utilisant les messages de l'application. Chaque processus prend régulièrement des points de reprise de manière indépendante. Cependant, en fonction des messages reçus et envoyés et des informations qui sont transportées sur ces messages, le processus devra peut-être prendre un point de reprise additionel ; on dit

que cette prise est *forcée*. L'algorithme de décision assure (ou maximise la probabilité) que le point pris fasse partie d'un état global cohérent, et donc qu'il existe une ligne de recouvrement toujours *assez* récente.
On distingue deux familles de protocole :

– les protocoles *model-based* : les communications et les prises de points de reprise doivent respecter un certain motif : par exemple, si tout envoi et toute réception de message est précédé d'un point de reprise, alors tous les points au moins à *un* état global cohérent, et le dernier point pris fait toujours partie de la ligne de recouvrement.
  Ces protocoles sont généralement basés sur les notions de chemins et de cycles "zig-zag" pour déterminer les états globaux cohérents.
– les protocoles *index-based* [10, 11] : les points de reprise sont indexés, chaque message porte l'index du dernier point de l'émetteur. Sur réception d'un message indiquant un index supérieur au sien, le récepteur doit prendre un point de reprise *avant* la prise en compte du message : l'incohérence est evitée "au dernier moment".
  Ce type de protocole utilise donc une méthode basée sur la suppression des dépendances causales entre les points de reprise des processus qui ont des communications directes.

1.4 Analyse et comparaison des différentes approches

Nous allons comparer ici les approches présentées dans la partie précédente. Nous ne considérerons que les approches synchronisées et induites par messages ; en effet, l'approche indépendante n'implique pas réellement l'utilisation d'un protocole : la recherche d'un état global cohérent est faite *après* une panne, au moment de la reprise. Nous comparerons sur les points suivants [12, 9] :

1. le degré de tolérance aux fautes, c'est à dire le nombre de fautes simultanées tolérées, pendant l'éxecution normale ou pendant la reprise,
2. le surcoût pendant l'exécution, en terme de CPU et de réseau,
3. le nombre de retours arrière moyen,
4. le surcoût de stockage, et la facilité du ramassage des points de reprises stockés devenus inutiles.

1.4.1 Degré de tolérance aux fautes

Les protocoles synchronisés sont particulièrement sensibles aux pannes des noeuds coordinateurs de la création d'une ligne de recouvrement. Ces noeuds contrôlent le début et la terminaison du protocole de constitution de la ligne, et leur panne doit être gérée comme un cas particulier dans le protocole, si on veut pouvoir les tolérer.

Dans les deux approches, la panne simultanée de plusieurs processus est supportée (en dehors de la création de la ligne de recouvrement) : il existe un état global antérieur à partir duquel tous les processus du système *doivent* de toutes façons reprendre, en cas de panne d'un ou de plusieurs processus.

1.4.2 Surcoût pendant l'exécution

L'utilisation de messages additionnels rend les protocoles synchronisés gourmands en ressources durant une exécution normale, que ce soit en terme de réseau ou en terme de temps d'exécution, particulièrement pour les protocoles bloquants qui stoppent les processus au cours de la synchronisation. C'est principalement ce surcoût qui rend les protocoles synchronisés en général incapables de passer à l'échelle, même dans le cas d'approches hybrides comme [13] où la synchronisation est rare.

L'approche induite par messages ne rajoute pas de messages additionnels durant l'exécution : le surcoût principal se trouve dans le traitement des messages applicatifs. En effet, il faut filtrer tout message entrant pour récuperer les informations utiles au protocole. Il y a aussi un surcoût du à la *taille des messages* qui peut être très différent d'un protocole à l'autre, allant de sans surcoût à un vecteur d'entiers de la taille du nombre de processus dans le système [14].

Bien-sûr, le coût de la prise d'un point de reprise par un processus est la même dans les deux approches.

1.4.3 Nombre de retours arrière moyen

Les protocoles synchronisés ont ici un avantage certain : *tous* les points de reprise sont utiles, et lorsqu'un processus prend un point de reprise, il est assuré de ne pas devoir retourner plus loin que ce point en cas de panne.

Dans le cas des protocoles induits par messages, on trouve deux politiques [5]: certains choisissent d'assurer que tout point de reprise pris est *utile*, c'est à dire qu'il fait partie d'un état global cohérent, alors que d'autres protocoles tentent de maximiser la probabilité que ce point soit utile, sans pour autant l'assurer. Cependant, pour les deux politiques, le temps de création de la ligne de recouvrement n'est pas borné (contrairement à l'approche synchronisée) puisqu'il depend de la propagation des messages applicatifs. La ligne de recouvrement peut s'étaler sur une grande période (le dernier point la constituant a été pris longtemps après le premier point) : on peut ainsi avoir une perte considérable de travail effectué pour certains processus.

1.4.4 Surcoût de stockage et ramassage

Sur ce point aussi, les protocoles synchronisés sont avantageux : puisque tout état global nouvellement créé est forcement un état cohérent, alors l'état global précédent peut être effacé de la mémoire. Le surcoût est donc minimal puisqu'il n'est nécessaire de conserver qu'un seul état global (plus bien-sûr celui en construction), et le ramassage des états devenus inutiles devient trivial.
Le ramassage des points dans le cas des protocoles induits par messages sera dépendant de la politique choisie, et nécessitera le plus souvent un processus chargé de déterminer les points devenus inutiles.

Chapitre 2

Tolérance aux pannes par journalisation des messages

La tolérance aux fautes par journalisation des messages (ou *message logging*) est une approche dans laquelle on enregistre systématiquement les messages qui circulent dans le système pour pouvoir “rejouer” les communications en cas de panne d’un processus. Ainsi, un processus enregistre sur un support stable tous les messages entrant et, occasionnellement et de façon totalement indépendante [1], prend un point de reprise. En cas de panne de ce processus, il sera relancé depuis son point de reprise le plus récent, puis tous les messages entrant enregistrés depuis la prise de ce point lui seront renvoyés. Cette approche est nommée *journalisation basée sur la réception*

L’alternative est la *journalisation basée sur l’émission* [16] : dans ce cas, c’est l’émetteur qui enregistre tous les messages sortant. En cas de panne, un processus devra demander à tous les processus avec qui il a communiqué la réémission de ces messages.

2.1 Déterminisme par morceaux

Avec les protocoles de journalisation, quand un processus tombe en panne, il sera parfois le seul à reprendre dans le système : les messages lui seront renvoyés pour qu’il “rattrape” les autres processus du système. Mais cette approche suppose alors que la deuxième exécution de ce processus soit *strictement équivalente* à la première pour que l’état global reste cohérent, puisque les autres processus ne reprennent pas.

Les processus du système doivent alors être partiellement déterministes ; on dit qu’ils sont déterministes par morceaux (*piecewise deterministic*) [17]. Un processus déterministe par morceaux est un processus dans lequel les seuls évenements non-déterministes sont les réceptions de message : à partir d’un état précis et d’une histoire de messages, toute exécution est équivalente.

1. Certains protocoles proposent cependant l’utilisation d’un protocole de points de reprise synchronisé, en plus de la journalisation des messages [15].

2.2 Les différents protocoles

On peut distinguer trois approches différentes dans la tolérance aux pannes par journalisation : pessimiste, optimiste et causale [18]. Nous n'aborderons pas ici la journalisation causale, qui est une extension de la journalisation optimiste.

2.2.1 Journalisation pessimiste

La journalisation pessimiste [19] est une approche qui enregistre de façon synchrone et sur un support stable tous les messages en transit dans le système. Tout message, une fois arrivé dans le contexte du récepteur est forcément enregistré, et pourra donc être rejoué en cas de panne : ainsi, tous les points de reprise du processus sont utilisables pour une reprise, puisque *tous* les évenements ont été enregistrés depuis la prise de ce point.

On note que l'ordre de ces messages doit lui aussi être conservé, de manière à recréer lors de la reprise la même histoire de message.

2.2.2 Journalisation optimiste

Dans le cas de la journalisation optimiste [17], les messages reçus ne sont pas forcément enregistrés immédiatement sur une mémoire stable. L'enregistrement des messages en transit dans le système se fait de façon asynchrone, de manière à grouper les enregistrements sur mémoire stable qui représentent un surcoût considérable. Les protocoles font donc l'hypothèse que ces enregistrements seront terminés avant qu'une panne n'arrive, de façon à ce que tous les évenements puissent être rejoués en cas de panne d'un processus.

Ils doivent cependant gérer le cas où une panne survient avant que tous les messages soient enregistrés : les informations enregistrées en mémoire volatile du processus tombé en panne sont alors perdues. Dans ce cas, certains processus seront *orphelins*, c'est à dire qu'un message qui leur était destiné a été perdu ; en cas de reprise, ce processus rendra l'état global incohérent. Mais si ce processus, une fois orphelin, a communiqué avec d'autres processus, alors eux-aussi seront incohérents en cas de reprise, puisque que leur état dépend de l'état d'un processus orphelin.

Regardons par exemple la figure 2.1, et supposons que l'on utilise un protocole optimiste basé sur la réception. Si R tombe en panne avant que $m4$ soit enregistré (mais aprés émission du message $m5$) puis reprend depuis le point de reprise C_R, Q devient alors orphelin et doit reprendre en C_Q pour défaire la réception de $m5$.

Les processus doivent alors maintenir des informations sur les relations causales entre eux, pour pouvoir déterminer, *au moment de la reprise*, quels sont les processus qui doivent reprendre pour que l'état global reste cohérent.

2.3 Analyse et comparaison des différentes approches

Nous allons comparer ici les deux approches de la journalisation selon les critères suivants :

1. le nombre de processus devant reprendre en cas de panne,
2. le surcoût en terme de temps d'exécution,

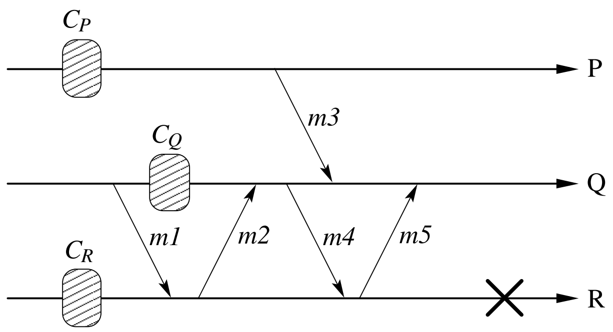

FIG. 2.1 – *Le processus Q doit lui aussi reprendre si* m4 *n'a pas été journalisé*

3. la taille des messages.

2.3.1 Nombre de processus à reprendre

C'est là le point fort des protocoles pessimistes : lors de la panne d'un processus, ces protocoles assurent que seul le processus fautif devra reprendre depuis son dernier état enregistré. Les messages étant réémis, soit par un processus extérieur dans le cas de l'approche basée sur la réception, soit par les autres processus de l'application dans le cas de celle basée sur l'émission, les autres processus n'ont pas à recouvrir un état antérieur.

Dans le cas des procotoles optimistes, le nombre de processus qui reprennent après une panne peut être égal dans le pire des cas au nombre de processus du système. En fait, ce nombre dépend du maillage et du taux de communication de l'application : dans le cas d'une application fortement maillée où les processus communiquent très souvent, la panne d'un processus entraînera certainement la reprise de tout le système.

2.3.2 Surcoût

Les protocoles pessimistes ont bien-sûr un coût très élevé à l'exécution. Tout message envoyé doit être journalisé de façon synchrone avant d'être reçu, ce qui en moyenne va doubler les temps de communication [20].

Les protocoles optimistes ont eux un surcoût à l'exécution "réglable". En effet, il est possible d'adapter la fréquence des enregistrements sur support stable en fonction de la fréquence des pannes et du surcoût maximal accepté durant une exécution sans panne. Bien-sûr, des enregistrements peu fréquents entraînent un temps de reprise en cas de panne important, puisque plus les intervalles entre enregistrements sont grands, plus les processus ont le temps de communiquer ; dans ce cas, un processus orphelin peut rendre orphelin un grand nombre d'autres processus.

2.3.3 Taille des messages

Les protocoles pessimistes n'ajoutent aucune information supplémentaire sur les messages, et n'ont donc aucun impact sur la taille des messages.

Les protocoles optimistes doivent conserver assez d'information pour pouvoir identifier les processus qui ont communiqué avec le processus fautif lors d'une panne : la plupart des propositions utilisent des vecteurs dont la taille est égale au nombre de processus du système (dans le meilleur des cas, ces vecteurs sont des vecteurs d'entiers). Ils doivent être ajoutés à tous les messages en transit dans le système, ce qui peut augmenter considérablement le temps d'émission mais aussi de traitement des messages.

Chapitre 3

Synthèse

Nous proposons ici une analyse comparative des protocoles de points de reprise et des protocoles de journalisation en fonction de certaines caractéristiques du système et de l'application. Nous détaillons ici cinq critères qui nous semblent déterminants et qui sont, ormis la fréquence des pannes, dépendants de l'application.

3.1 Fréquence des pannes

La valeur caractéristique de la fréquence des pannes dans un système distribué est le MTBF (*Mean Time Between Failures*), c'est à dire le temps moyen entre deux pannes consécutives de n'importe quel noeud du système. Cette valeur varie d'une minute pour les systèmes constitués de stations de travail [21], à deux semaines pour une grappe de 1024 machines dédiée au calcul haute-performance [22].

Dans le cas d'un système où le MTBF est très petit, de l'ordre de quelques minutes, il faut privilégier la rapidité de la reprise. On choisira alors une approche par journalisation, dans laquelle généralement [1] seul le processus fautif doit reprendre. Les protocoles de points de reprise nécessiteraient une reprise de tout le système, qui serait plus longue : si le temps moyen entre deux pannes est inférieur au temps de reprise globale, le système pourrait alors rester indéfiniment dans l'état de reprise !

Il faut cependant exclure les protocoles ne supportant pas les pannes simultanées (ou seulement *k* pannes simultanées, où *k* est un paramètre du protocole), comme certaines approches par journalisation optimiste.

3.2 Nombre de processus

Les protocoles qui nécessitent la reprise des tous les processus en cas de panne de l'un d'entre eux sont mal adaptés aux systèmes comportant un grand nombre de noeuds. Les pro-

1. dans le cas des protocoles optimistes, on a vu que la panne d'un processus pouvait entraîner la reprise de plusieurs processus

tocoles de journalisation des messages sont donc plus adaptés aux applications de grande taille que les protocoles de points de reprise.

Cependant, c'est l'approche qui dégrade le plus les perfomances des exécutions sans panne ; le choix sera donc aussi guidé par le taux de panne moyen des processus du système. Pour un système où la fréquence des panne est faible, on choisira plutôt une journalisation optimiste, et si on veut optimiser le temps d'exécution sans panne, on se penchera alors vers un protocole de points de reprise induits par messages.

3.3 Taux de communication, maillage

Un taux de communication élevé va rendre l'utilisation de protocoles de journalisation très coûteuse pendant une exécution sans panne, surtoût dans le cas d'une approche pessimiste. Ces protocoles nécessitent des accès fréquents à une mémoire stable, ce qui a pour effet, en plus du coût de la communication, de créer de la contention au niveau de cette mémoire. De plus, l'espace de stockage nécessaire pour les points de reprise et les journaux de messages est important, en particulier pour les approches optimistes, où le ramassage des points et des journaux devenus inutiles est complexe et nécessite souvent de conserver plusieurs points consécutifs par processus.

A l'opposé, un taux de message élevé va aider au bon déroulement des protocoles de points de reprise induits par messages. En effet, plus les processus communiquent entre-eux, plus ils diffusent rapidement les informations sur la ligne de recouvrement en cours. La formation des lignes de recouvrement est alors plus rapide, donc le nombre de points de reprise potentiellement inutiles diminue. A l'inverse, plus la constitution de la ligne de recouvrement est longue, plus une panne a de risque de survenir pendant cette constitution, rendant ainsi les points déjà pris inutiles.

3.4 Non-déterminisme

Si le système contient des processus non-déterministes, alors les protocoles de journalisation de messages ne peuvent pas être utilisés. En effet, si le comportement de certains processus est la conséquence d'actions indéterministes internes, ils ne sont alors plus déteministes par morceaux, et deux exécutions avec la même histoire de messages peuvent être différentes. L'état global du système après la reprise d'un processus peut donc être incohérent. On choisira dans ce cas un protocole de points de reprise.

3.5 Communication avec le monde extérieur

Si le système doit communiquer avec des processus du monde extérieur, c'est à dire des processus qui ne peuvent pas être contrôlés par le protocole, ou dont le comportemant à des effets de bord sur le monde reél (affichage, impression,...), alors les protocoles de points de reprise ne sont pas adaptés. En effet, on ne peut pas faire reprendre ces processus dans un état antérieur en cas de panne.

Si le choix se porte quand même sur un protocole de ce type, il doit alors être modifié pour prendre en compte ce monde extérieur, par exemple en journalisant les messages entrant et sortant du système.

TAB. 3.1 – *Adéquation des différentes techniques en fonction des caractéristiques de l'application*

	POINTS DE REPRISE		JOURNALISATION	
	Synchronisé	**Induit par messages**	**Pessimiste**	**Optimiste**
MTBF petit	$+$	$+$	$++$	$++$
Nombre de processus élevé	$-$	$+$	$++$	$++$
Taux de communication élevé	$++$	$++$	$-$	$+$
Non-déterminisme des processus	$++$	$++$	$--$	$--$
Communication avec le monde extérieur	$+$	$+$	$++$	$++$

Le tableau 3.1 résume les choix adaptés de protocoles de tolérance aux pannes en fonction des caractéristiques de l'application. On note :

- $++$, quand l'utilisation de ce protocole est idéale dans ces conditions,
- $+$, quand le protocole utilisé doit être légerement modifié pour prendre en compte les caractéristiques de l'application,
- $-$, quand le protocole est utilisable, mais peut conduire à des surcoûts inacceptables,
- $--$, lorsque le protocole ne peut pas fonctionner dans ces conditions.

Chapitre 4

Objets actifs en Java : ProActive

Notre travail se place dans le cadre du projet OASIS qui développe la bibliothèque ProActive. Nous proposons ici une rapide présentation de cette bibliothèque, et une analyse des différentes caractéristiques qui auront un impact fort dans la conception d'un protocole de tolérance aux pannes adaptés.

4.1 Présentation

ProActive est une bibliothèque Java qui permet la programmation d'applications parallèles et distribuées. Elle repose sur un modèle MIMD (*Mutliple Instructions, Multiple Data*) et sur la notion d'objet actif [23], c'est à dire un objet qui possède une activité propre. Elle fournit en particulier :

- des appels de méthodes aynchrones avec futurs transparents, c'est à dire une communication par messages (requêtes et réponse),
- de la mobilité faible,
- des communications de groupes.

ProActive utilise un protocole méta-objet pour réifier les objets et les appels de méthodes entre objets actifs [24]. On appellera "objet réifié" l'objet Java standard rendu actif par l'ajout de méta-objets, et "objet actif" l'ensemble objet réifié et méta-objets [1].

La réification des communications permet d'avoir des communications asynchrones : lorsqu'un objet actif appelle une méthode sur un autre objet actif, cet appel est réfié en une requête qui est émise au récepteur pour y être stockée dans sa *queue de requêtes*.

Lors de cette émission, l'objet émetteur reçoit un *futur* pour lui permettre de continuer son exécution. Ce futur correpond à une promesse de réponse, qui sera mise à jour de façon transparente lorsque le résultat sera disponible. Cependant, si un objet tente d'utiliser un futur avant qu'il soit mis à jour, alors il entre dans un état d'*attente par nécessité*, c'est à dire que son fil d'exécution est suspendu jusqu'à ce que le futur soit mis à jour.

1. Le terme d'objet réifié est un abus de language, puisque c'est la *référence* sur l'objet qui est réifiée.

Cette réification permet d'avoir des appels de méthodes asynchrones, mais un *rendez-vous* est conservé : lors de l'appel d'une méthode d'un objet vers un autre, l'émetteur ne peut continuer son exécution que lorsque la requête créée est effectivement deposée dans la queue de requêtes du récpeteur.

4.2 Analyse

Nous allons analyser ici les propriétés de ProActive, pour déterminer les caractéristiques qui auront un impact important sur la conception du procotole, que ce soit un impact positif comme la sérialisation de Java ou le rendez-vous, ou négatif comme l'impossibilité de serialiser l'état d'un fil d'exécution.

4.2.1 Java

Tout d'abord, ProActive est entiérement écrit en Java : un objet actif est en fait un ensemble d'objets Java standards. Nous allons donc pouvoir utiliser la sérialisation [25] proposée par Java pour avoir une représentation des objets qui puisse être stockée et récupérée en cas de reprise du système.

Pourtant, si l'utilisation de Java nous apporte cette fonctionnalité intéressante, nous allons voir que ce language pose un problème majeur dans la conception d'un protocole de tolérance aux pannes pour ProActive, mais aussi pour toutes les bibliothèques du même type *écrites en Java*.

4.2.2 Activité des objets

On a vu dans les chapitres précédents que la majorité des protocoles de tolérance aux pannes nécessitent la capacité de prendre un enregistrement de l'état d'un processus *à tout moment*, puisqu'on peut par exemple devoir prendre un point de reprise sur réception d'un message. C'est là un point critique dans la conception du protocole : cette *préemptivité* n'est pas possible avec Java.

En effet, un objet actif possède une activité propre, ce qui en Java s'exprime par le fait que cet objet est associé à un objet de la classe `Thread`. Lorsqu'on va vouloir prendre un enregistrement d'un objet, on va le serialiser, mais les objets de la classe `Thread` *ne sont pas serialisables*, et ne peuvent donc pas faire partie de l'enregistrement[26]. De plus, on ne peut pas, sur la plate-forme Java standard, contrôler de façon préémptive le comportement de ces *threads* : en particulier, on ne peut pas stopper l'exécution d'un *thread* par une action extérieure. La conséquence est que les prises de points de reprise ne seront possibles qu'à certains moments de l'exécution, moments que l'on appelera *états stables*. Ces moments sont ceux pendant lesquels l'objet actif ne sert pas de requête, comme par exemple entre deux services consécutifs.

Notre protocole va donc devoir tenir compte de cette contrainte, et nous verrons que nous allons pour cela utiliser les propriétés offertes par l'utilisation dans ProActive d'un protocole méta-objet.

4.2.3 Protocole méta-objet et réification

ProActive réifie les communications en objets de type requêtes ou réponses, et utilise des futurs du coté appelant, et une queue de requêtes du coté appelé. Cette matérialisation des appels va nous permettre dans le protocole de facilement les manipuler pour, par exemple, enregistrer certaines communications qui devraient être rejouées en cas de panne du système, ou encore modifier la queue de requête d'un objet pour influer sur l'ordre de réception des requêtes pour conserver l'ordre global.

La réception d'une requête et d'une réponse par un objet actif est différente au niveau de son impact sur les objets eux-mêmes : la réception d'une requête n'a d'effet sur les objets qu'au niveau méta, c'est à dire que la queue de requêtes va être modifiée, mais pas l'état de l'objet au niveau applicatif.

La réception d'une réponse va mettre à jour le futur correspondant, et, en particulier si l'objet était en attente sur ce futur, certainement modifier l'état de l'objet au niveau applicatif, comme par exemple mettre à jour une variable d'instance de cet objet. On note qu'une réponse reçue correspond forcément à un futur chez le récepteur et, si ce n'était pas le cas, cette réponse serait ignorée par ce récepteur.

Le fait que la réception d'une requête n'a pas d'effet sur l'objet réfié a une conséquence importante au niveau du déroulement d'une exécution distribuée : *ce n'est plus parce qu'un évenement survient après réception d'une requête que cet évenement succède causalement cette réception* (Voir 1.1.2). Dans notre modèle, pour qu'un évenement soit la conséquence de la réception d'une requête, il faut qu'il ait lieu pendant ou après *le service* de cette requête.

On note que si l'ordre des réceptions de requêtes est bien-sûr déterminant pour l'état de l'objet actif, une étude formelle du modèle de ProActive [27] a montré que l'ordre de réception des réponses n'a absolument *aucune influence* sur un objet actif.

4.2.4 Communications avec Rendez-Vous

Enfin, les communications dans ProActive sont asynchrones *avec rendez-vous*. Ainsi, lorsqu'un objet actif envoie un message, il ne continue son exécution que lorsque le message est dans le contexte du récepteur, c'est à dire dans la queue de requête pour une requête et dans le futur correspondant pour une réponse. Cette propriéte va nous apporter deux avantages intéressants pour la conception d'un protocole de tolérance aux pannes :

– les communications entre objets actifs sont FIFO (*First In, First Out*) point à point, c'est à dire que si un objet envoie deux messages au même récepteur, alors ces messages arriveront *forcément* dans l'ordre dans lequel ils ont été émis.
– le rendez-vous se caractérise par une attente de l'émetteur que le message soit dans le contexte du récepteur. Il est alors possible de renvoyer à l'émetteur à la fin de ce rendez-vous une information sur l'état du récepteur *au moment de la réception du message*. De plus, cette information est reçue avant que l'émetteur ne continue son exécution normale.

4.2.5 Notations

Nous décrivons ici comment nous allons schématiser ces propriétes dans la représentation temporelle d'une exécution répartie avec ProActive.

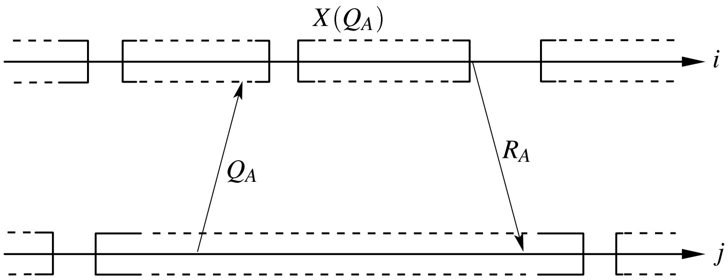

FIG. 4.1 – *Objets actifs communiquants*

On a sur la figure 4.1 deux objets actifs qui communiquent. j appelle une méthode sur l'objet i : une requête (Q_A) est émise de j vers i, puis cette requête est servie sur l'objet i ($X(Q_A)$), et enfin une réponse (R_A), résultat de la méthode exécutée, est envoyée de i vers j.

Les rectangles pointillés représentent des moments où l'objet est en service de requête, et n'est donc pas dans un état stable : les états stables sont les moments où l'exécution n'est représenté que par une simple ligne. On notera que les services de requêtes peuvent être annotés avec la requête servie (ici, $X(Q_A)$ indique que la requête Q_A en en train d'être servie).

On voit que la réception d'une requête "n'entre pas" dans le rectangle lorsque l'objet est en cours de service : on représente ici le fait que la réception de requête n'a pas d'impact sur l'état de l'objet au niveau applicatif. A l'inverse, la réception d'une réponse entre dans le rectangle pour signifier la modification potentielle de l'objet au niveau applicatif.

Nous rajoutons dans cette notation le point de reprise (figure 4.2). Sa représentation inclut son numéro (**n**), l'état de la queue de requêtes de l'objet actif ($[Q_A, Q_C, Q_D]$), ainsi qu'une autre queue de messages ($]Q_B, R_E[$). Cette autre queue est ajoutée à chaque objet actif pour les besoins du protocole. Elle contient[2] tous les messages applicatifs qui devront être réémis avant le redémarrage de l'activité de l'objet en cas de reprise depuis ce point de reprise. Si l'une de ces queue est vide, on la note $[\emptyset]$ ou $]\emptyset[$.

2. nous verrons par la suite précisement ce qu'elle contiendra

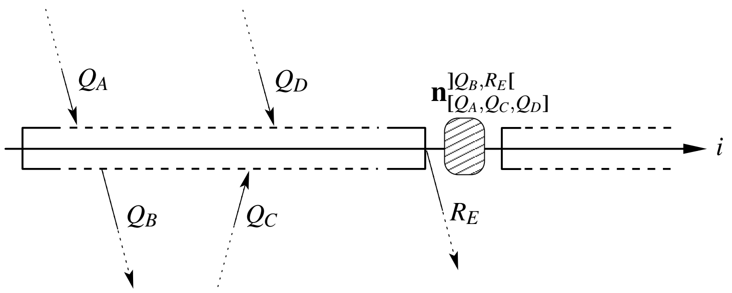

FIG. 4.2 – *Objet actif prenant un point de reprise*

deuxième partie

Solution proposée

Chapitre 1

Principes

1.1 Description

Avant de donner une vue formelle du protocole que nous avons développé, nous présentons les différents points qui justifient son comportement.

1.1.1 Modèle

Le protocole proposé pour le modèle à objet actif est un protocole de points de reprise induits par messages, avec reprise asynchrone, inspiré sur [10] et [11]. Nous avons choisi une approche basée sur les index.

Le principe est le suivant : chaque objet actif prend régulièrement des points de reprise, en utilisant un compte à rebour. Chaque point de reprise pris est numéroté (dans l'ordre séquentiel), et un objet actif annote tous les messages qu'il émet avec l'index du dernier point pris.

Cependant, il est possible qu'un objet soit forcé de prendre un point de reprise "imprévu", par exemple sur réception d'un message : si le message porte un numéro n supérieur au dernier point du récepteur, celui ci doit alors prendre un (ou plusieurs) point de reprise indexé par ce numéro *avant* la prise en compte du message, de façon à ne pas rendre incohérent l'état formé par les deux points indexés n sur les deux objets.

De cette façon, le protocole assure l'absence de dépendance causale entre les points de reprise portant le même numéro, et assure donc la cohérence d'un état global formé de points portant tous le même index. Cette propriéte permet de déterminer la ligne de recouvrement en cas de panne.

1.1.2 Problème de la non-préemptivité

On a vu dans la partie 4.2.2 que les objets actifs du modèle ProActive ne peuvent pas prendre de point de reprise de façon préemptive : dans ce cas, un objet qui reçoit un message ne peut pas réagir tout de suite si il est en train de servir une requête. Nous allons donc utiliser une stratégie de réaction *a posteriori* : comme nous ne pouvons pas placer les points de reprise, nous allons virtuellement déplacer les messages, lorsque cela est possible. Nous traiterons différemment le cas des requêtes et des réponses :

– si le message reçu est une requête, alors cette requête sera exclue[1] de la queue de requêtes de l'objet actif au moment de la prise du prochain point de reprise ; ainsi, comme on le voit sur la figure 1.1, on simule la réception du message *après* la prise du point. Bien-sûr, seule l'image de l'objet actif utilisée pour le point de reprise est modifiée : lorsque l'objet continue son exécution après la prise du point, sa queue de requêtes est toujours la même qu'avant la prise de ce point.

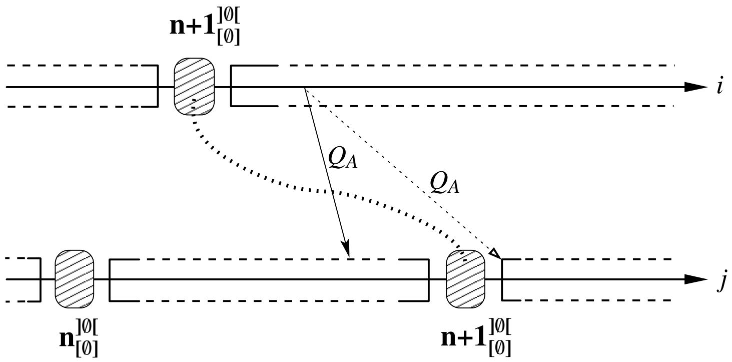

FIG. 1.1 – *La requête Q_A est exclue de la queue de requêtes lors de la prise du point $n+1$ sur j*

– si le message reçu est une réponse, on ne peut plus alors "annuler l'effet" de la réponse comme on le fait pour une requête. En particulier, si l'objet récepteur est bloqué dans un état d'attente par nécessité sur cette réponse, il ne peut pas passer dans un état stable avant d'avoir pris en compte cette réponse. Elle fera alors partie de l'état de l'objet lors de la prise du prochain point. En cas de reprise depuis ce point (dans l'exemple le point $n+1$), le récepteur recevra donc *une deuxième fois* cette réponse, puisque l'émetteur reprendra lui aussi depuis ce point.
  Le mécanisme des futurs permet à l'objet récepteur d'automatiquement ignorer cette deuxième occurence de la même réponse, puisqu'il n'existera plus de futur associé à cette réponse. Il faudra alors s'assurer que la deuxième occurence soit bien-sûr strictement la même que la première, puisque le récepteur n'en tiendra pas compte.

Ces deux points nous permettent de contourner le problème de la non préemptivité des objets actifs, et donc d'appliquer dans le cas de ProActive une méthode de points de reprise induits par messages et basé sur l'index.

1.1.3 Cohérence forte

Le protocole n'assure pour l'instant que la cohérence simple des lignes de recouvrement formées. Mais le rendez-vous des communications de ProActive nous permet d'appliquer aussi

1. de manière à ne pas modifier l'ordre de réception, elle sera en fait remplacée par une requête en attente ; voir 1.1.6

le dual du protocole qui assure la cohérence simple, et donc assurer la cohérence forte des lignes de recouvrement. En effet, comme nous venons de le voir, lors de la récéption d'un message, le récepteur peut se synchroniser sur l'émetteur grâce au numéro porté par ce message. Grâce au rendez-vous, le récepteur peut renvoyer le numéro de son dernier point de reprise à l'émetteur, de telle sorte que celui-ci puisse également se synchroniser. Ainsi, les deux objets peuvent se synchroniser au cours d'une seule communication.

Supposons qu'un objet envoie une requête (ou une réponse, le traitement dans ce cas est équivalent) à un autre, et qu'il reçoive de la part du récepteur un numéro $n+k$ supérieur à son dernier point de reprise. Il doit alors, pour conserver la cohérence forte de la ligne $n+k$, prendre un point de reprise *avant* l'émission de ce message. Ceci est impossible puisque l'objet émetteur est en train de servir une requête, et n'est donc pas dans un état stable. Pour résoudre ce problème, il faut enregistrer l'émission de cette requête dans le prochain point de reprise, de manière à ce qu'elle ne soit pas perdue en cas de rejeu. Comme on peut le voir sur la figure 1.2, on simule de cette manière l'envoi du message *après* la prise du point de reprise.

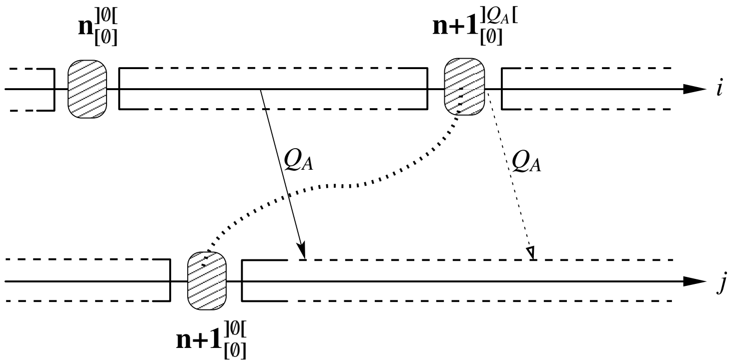

FIG. 1.2 – *La requête Q_A est enregistrée pour réémission dans le point de reprise $n+1$ de l'objet actif i*

1.1.4 Points de reprise multiples et points de reprise impossibles

Le protocole est basé sur le fait qu'un objet actif, lorsqu'il le peut, va "rattraper" les autres, en terme de numéro de points de reprise. Il peut donc arriver qu'un objet actif soit en retard de plus d'un point sur les autres. C'est le cas de j sur la figure 1.3 : lorsqu'il arrive à un état stable, il doit prendre 2 points de reprise pour rattraper l'état global en cours de construction.
C'est pour cette raison que les objets peuvent avoir à maintenir des informations sur plusieurs points de reprise en même temps : on voit ici que l'ensemble des requêtes à réémettre est différent pour les deux points $n+1$ et $n+2$ sur j.

Cependant, il sera parfois impossible à un objet actif de réaliser un point de reprise n : si il a déjà servi une requête qui indiquait que son émetteur avait *déjà* pris son point n. En effet, elle

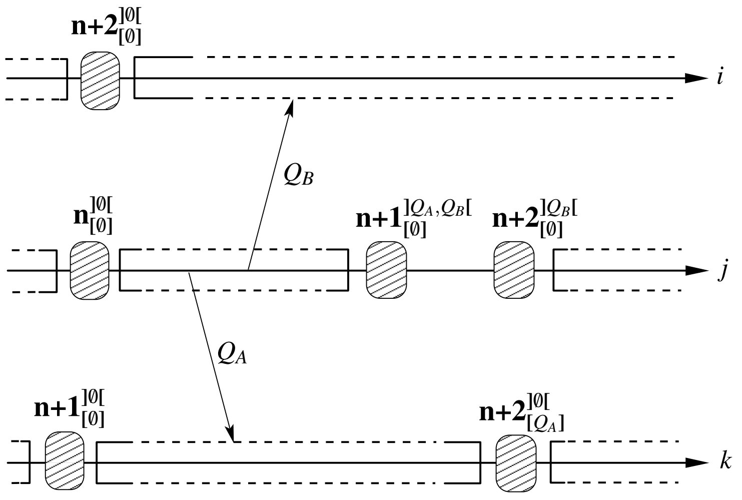

FIG. 1.3 – *L'objet actif j doit faire 2 points de reprise d'affilé*

ne peut alors plus être exclue de la queue de requêtes comme on le fait dans la figure 1.1. Dans ce cas, la ligne de recouvrement n ne pourra jamais être formée.

Si, entre chaque service de requête, l'objet actif est dans un état stable, alors il peut toujours prendre un point de reprise ; c'est le comportement par défaut dans ProActive. Cependant, l'utilisateur peut modifier ce comportement [28] et réaliser par exemple des services imbriqués ; alors, la fréquence d'occurence d'un état stable est imprévisible, et certains points de reprise peuvent alors être impossibles.

1.1.5 Equivalence des exécutions

L'enregistrement de messages dans les points de reprise et la possibilité de recevoir deux fois la même réponse impliquent que deux exécutions partant du même état doivent être équivalente, *au moins jusqu'à un certain point dans le temps*. Ceci pour deux raisons : d'abord, regardons le cas de la figure 1.4. En cas de reprise depuis la ligne $n+1$, l'objet actif k devra réémettre la requête Q_B et recevra une deuxième fois la réponse R_B, réponse qu'il ignorera. Imaginons alors que la requête Q_A arrive, pendant la deuxième exécution, *après* Q_B : j servira alors les requêtes dans un ordre différent, ce qui pourra rendre la deuxième occurence de la réponse R_B *différente* de la première ! Il faut donc que la deuxième exécution de j soit équivalente (i) *jusqu'au moment où on peut être sûr qu'une réponse envoyée par j ne sera pas reçue par le récepteur avant que celui-çi ait pris son point de reprise* $n+1$.

La deuxième raison qui oblige à avoir des exécutions équivalentes est dûe au rejeu de certaines émissions de messages lors d'une reprise : les relations de causalité entre les réceptions doivent être conservées. Considérons le cas de la figure 1.5 : lors de l'exécution initiale, j recevra forcément Q_A avant Q_C car la réception de Q_C par j est une conséquence de l'émission de Q_B par k, émission qui se passe *après* l'émission de Q_A. Cet ordre de réception est garanti grâce au rendez-vous des communications de ProActive.

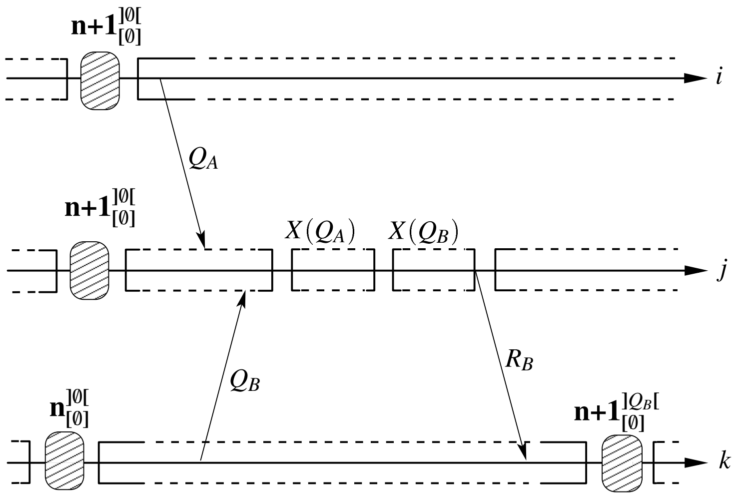

FIG. 1.4 – *Une deuxième exécution à partir de* $n+1$ *peut être différente*

En cas de reprise depuis la ligne de recouvrement $n+1$, il faut assurer que j recevra encore une fois Q_A avant Q_C, ce qui n'est pas forcément le cas : comme Q_A a été enregistrée pour être réémise dans le point de reprise $n+1$ sur k, Q_A n'a plus de relation de causalité avec Q_C. Cette fois encore, il faut s'assurer que la deuxième exécution sera équivalente (ii) *jusqu'au moment où on peut être sûr qu'il n'y aura plus par la suite de requête à l'intention de* j *qui soit enregistrée pour réémission dans un point de reprise* $n+1$ *sur un des autres objets actifs.*

Les conditions (i) et (ii) reviennent à dire qu'en cas de reprise depuis un point n, il faut que la réexécution soit strictement équivalente à la première jusqu'au moment où, dans la première exécution, l'état global n avait été terminé (soit le moment où tous les objets du système avaient pris le point de reprise n).

Cette nécessité d'équivalence implique deux choses :

- Les processus constituant l'application doivent être déterministes par morceaux, comme défini dans la partie 2.1. En effet, les exécutions devant être parfois équivalentes, il faut assurer qu'un objet recevant deux fois les mêmes messages dans le même ordre aura strictement le même comportement à chaque fois.
- Il faut parfois assurer que pendant deux exécutions à partir du même état, les mêmes messages arrivent dans le même ordre pour tous les objets actifs : nous allons utiliser un *historique de réception* pour les objets actifs.

1.1.6 Historique

Tout point de reprise est associé à un historique de réception, c'est à dire une liste chronologique des réceptions de message. On parlera de l'historique n, associé au point de reprise n.

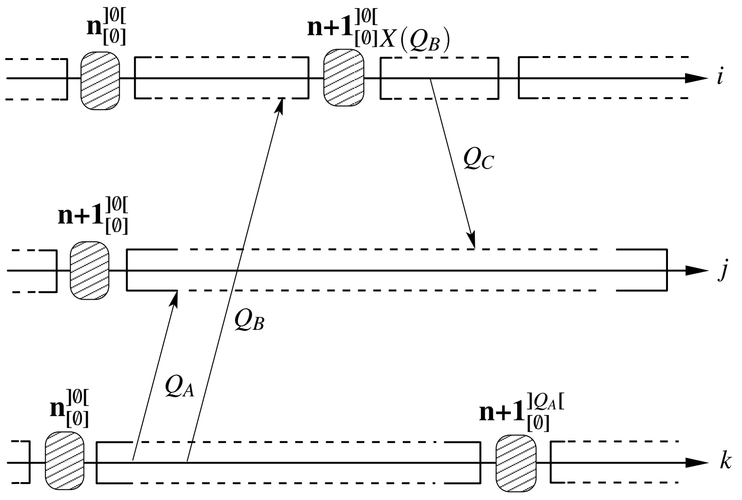

FIG. 1.5 – *La réception sur j de Q_C doit précéder celle de Q_A*

Les caractéristiques du modèle à objet actif considéré permettent à cet historique d'avoir les propriétes suivantes :

- Il sest constitué uniquement des réceptions de *requêtes*. En effet, Caromel, Henrio et Serpette ont montré formellement dans [27] que l'ordre de réception des réponses n'a pas d'impact sur le comportement d'un objet actif. Cette propriété est due à l'utilisation de futurs et au mécanisme d'attente par nécessité.
- Chaque réception de requête est identifiée dans l'historique simplement par l'*identificateur unique de son émetteur*. Grâce aux propriétes de communication FIFO point-à-point et de déterminisme par morceaux des objets actifs, cette information est suffisante pour assurer l'ordre des réceptions [27].
- En pratique, cet historique est réalisé en utilisant un mécanisme équivalent à celui des futurs et de l'attente par nécessité : il sera constitué de requêtes dîtes *en attente*. Ces requêtes sont en fait des requêtes “vides”, qui doivent être remplies avec une requête normale provenant d'un émetteur en particulier. Une requête en attente contient donc simplement l'identificateur de cet émetteur attendu.
  Le comportement de ce système est une attente par nécessité : lorsque l'objet tente de servir une requête de ce type, il est bloqué. L'arrivée d'une requête provenant de l'émetteur attendu le débloquera, et il pourra servir cette requête normalement. Une requête arrivant d'un autre émetteur, si il n'existe pas de requête en attente sur cet émetteur, sera placée normalement, à la fin de la queue de requêtes, sans débloquer l'objet.
  Cette requête en attente sera aussi utilisée pour remplacer une requête normale lorsque celle-ci doit être exclue de la queue de requêtes lors de la prise d'un point de reprise (Voir 1.1.2). On assure ainsi qu'elle sera reçue virtuellement “au même moment” en cas de reprise depuis ce point.

Prenons l'exemple de la figure 1.6 : l´historique n au moment noté $\diamondsuit$ sur l'objet actif i est de la forme $\{j,k\}$, où j et k sont des identifiants uniques d'objets actifs. j indique la réception de la requête Q_B émise par j. k représente la réception de la requête Q_D émise par k. On note que cette historique est ordonné, et donc qu'ici, en cas de reprise depuis le point n, i bloquera en attente d'une requête de la part de j même si une requête de la part de k arrive. Il n'y a pas dans l'historique de trace de la réception de la réponse R_C.

Comme on l'a vu dans la partie précédente, cet historique est fini dans le temps. Le protocole doit donc déterminer le moment à partir duquel l'historique n (associé à un point de reprise n sur l'objet actif i) est suffisant, c'est à dire que la réexécution de i après une reprise depuis le point n n'a plus besoin d'être équivalente à l'exécution initiale. A ce moment, l'historique peut être clôturer, donc ne plus tenir compte des requêtes entrantes.

Le moment à partir duquel l'historique n est suffisant est le moment où *tous* les objets actifs ont pris le point indexé n, c'est à dire quand l'état global n est formé. En effet, il ne peut plus dans ce cas y avoir de nouveaux messages à réémettre enregistrés dans un point de reprise n : la suite de l'exécution n'a plus besoin d'être strictement équivalente en cas de reprise depuis le point n.

A partir de ce moment, i pourra clôturer l'historique n, c'est à dire ne plus y ajouter les requêtes entrantes, et l'associer au point de reprise n, et ce point deviendra alors valide pour être utilisé dans une ligne de recouvrement.

Cependant, tous les objets actifs ne vont pas clôturer de manière synchronisée leur historique : il peut donc survenir des problèmes de cohérence. Prenons le cas de la figure 1.6. Les objets ont clôturé leur historique n, mais ces historiques ne sont pas cohérents : la requête Q_E a été reçue dans l'historique de j, mais n'a pas été émise dans l'historique de k. En cas de reprise depuis le point n, il est possible que la réexécution de k soit différente (après la fin de l'historique), et que ce message ne soit jamais envoyé en cas de reprise depuis depuis la ligne n : j restera bloqué en attente indéfiniment. Il aurait fallu que l'historique sur j soit clôturé avant la réception de Q_E, comme indiqué en pointillé sur la figure 1.6. On notera cependant que la requête Q_F ne pose pas de problème de cohérence entre les clôtures sur i et j.

On voit bien ici que le message Q_E est un message orphelin par rapport aux points de clôture d'historique : ces points doivent donc être *cohérents* entre eux, au sens de [1] ! On va donc utiliser un protocole induit par messages assurant la cohérence simple pour les coordonner, celui décrit dans 1.1.1. Ce modèle peut dans ce cas être appliqué directement puisque la clôture d'historique peut être préemptive (il suffit de fermer l'historique avant de prendre en compte le message déclenchant cette fermeture, comme la clôture en pointillé est faîte avant la prise en compte de Q_E).

1.1.7 Reprise après panne

L'historique de réception va être utilisé au moment de la reprise. Le système ne peut repartir depuis une ligne de recouvrement n que si tous les points de reprise indexés n ont clôturé leur historique. Dans ce cas, chaque point de reprise est associé à son historique. Lorsque l'objet actif i va reprendre depuis un oint de reprise n, il va d'abord concaténer l'historique n à sa queue de

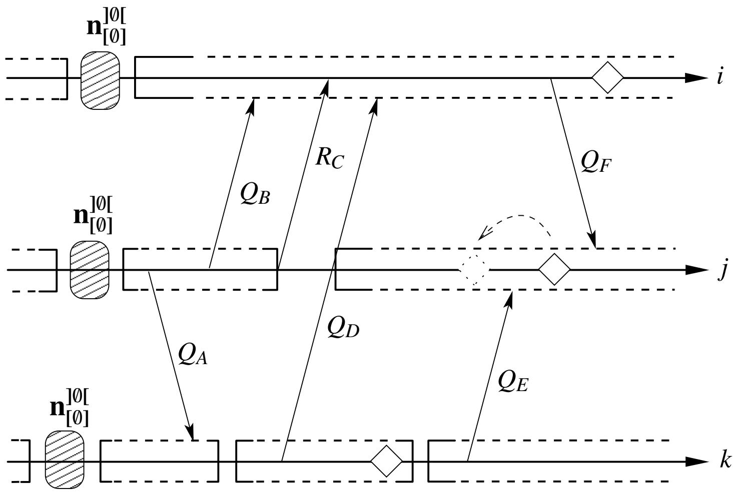

FIG. 1.6 – *Les clôtures d'historique ($\diamondsuit$) sur j et k ne sont pas cohérentes entre elles*

requête ; la nouvelle queue de requêtes sera donc formée de la queue de requête enregistrée au moment de la prise du point (pouvant contenir des requêtes en attente), suivie d'une queue de requêtes en attente. Ainsi, on force de manière paresseuse l'ordre de réception des requêtes, de façon à ce qu'il soit identique à celui de l'exécution précédente.

Avant de reprendre son activité, il devra aussi réémettre toutes les requêtes ou réponses qui auraient été enregistrées dans le point de reprise (comme par exemple dans la figure 1.2). La réémission des requêtes devra se faire *dans l'ordre dans lequel celles-ci ont été émises lors de la première exécution*. En effet, l'ordre point à point doit être conservé : si deux de ces requêtes sont attendues par des requêtes en attente[2] chez un unique recépteur, ce recépteur ne pourrait pas les réordonner à leur arrivée, puisqu'une requête en attente ne contient que l'identifiant de l'émetteur. Ainsi, la reprise du système peut-être totalement asynchrone, puisque l'attente des objets actifs se fait de façon indépendante grâce au mécanisme de requête en attente.

La procédure de reprise étant asynchrone, il va être possible à un instant donné que deux objets actifs ne soient pas dans la même occurence d'exécution : si l'un a déjà effectué sa reprise, alors que l'autre ne "sait pas encore" qu'il faut reprendre. Si ces deux objets s'envoient un message, il faut alors être capable de détecter cette incohérence et d'y remédier. C'est pour cela que l'on va utiliser un *numéro d'incarnation* [29, 10], qui va représenter le nombre de reprises effectuées par un objet depuis le lancement du système.

Chaque message porte le numéro d'incarnation de son émetteur, ainsi que le numéro du point de reprise depuis lequel s'est effectuée la dernière reprise. De cette façon, le récepteur pourra savoir si l'objet qui lui envoie un message est dans la même occurence d'exécution que lui. Soit

2. Ces requêtes réémises sont toujours attendues par des requêtes en attente chez leurs destinataires : ce sont des requêtes qui ont été émise alors que l'état global en formation n'était pas terminé (par exemple la figure 1.5) et qui ont donc été enregistrées dans un historique

le numéro d'incarnation porté est :

- inférieur au numéro local, et ce message peut être ignoré,
- supérieur au numéro local, alors c'est que le récepteur doit reprendre. L'objet émetteur est bloqué sur l'émission du message jusqu'à ce que le recépteur ait terminé sa reprise,
- égal au numéro local, et le message sera traité normalement.

1.2 Synthèse

Le fonctionnement du protocole est donc basé principalement sur l'échange d'informations au moment des communications applicatives entre objets actifs. Nous proposons ici un tableau synthétique des actions à effectuer en fonction des évènements qui vont survenir dans la vie d'un objet actif.

TAB. 1.1 – *Synthèse*

ÉVÈNEMENT	ACTIONS
Emission d'un message	• Si l'index n du dernier point pris par le récepteur est plus grand que m, celui du dernier pris localement → Programmer la prise des points $m+1$ à n dès que possible et enregistrer le message pour réémission dans ces points.
Réception d'un message	• Si le numéro d'incarnation de l' émetteur est plus petit que le numéro local → Ignorer le message. • Si le numéro d'incarnation de l'émetteur est plus grand que le numéro local → Reprendre depuis le même point. • Si le numéro n du dernier historique clos sur l'émetteur est plus grand que le numéro local m → Clôturer les historiques de m à n. • Si l'index n du dernier point pris par l'émetteur est plus grand que m, celui du dernier pris localement → Programmer la prise des points $m+1$ à n dès que possible. • Si le message est une requête et s'il existe des historiques ouverts → ajouter l'identifiant de l'émetteur à ces historiques.
Etat global n formé	• Clôturer les historiques ouverts jusqu'au $n$ième.
Etat stable	• Prendre tous les points de reprise programmés.
Prise d'un point de reprise n	• Prendre une image de l'état de l'objet. • Remplacer dans la queue de requêtes de cette image les requêtes qui ont été émises aprés un point dont l'index est supérieur ou égal à n par des requêtes en attente. • Ouvrir l'historique n.
Reprise depuis un point n	• Récupèrer l'image d'état n et l'historique n. • Concatèner l'historique à la queue de requêtes de l'image récupérée. • Emettre dans l'ordre tous les messages enregistrés sur le point n. • Redémarrer l'activité.

Chapitre 2

Spécification algorithmique

2.1 Définitions et notations

Nous donnons ici le formalisme utilisé pour la descritption du protocole. Nous allons distinguer deux parties : l'existant dans le modèle à objet actif considéré (requête, réponse,...), et ce qui sera ajouté pour permettre au protocole de fonctionner.

2.1.0.1 L'existant dans ProActive

On notera :

– $Q_{i,j}$ une requête de i vers j
– $R_{j,i}$ une réponse de j vers i
– $M_{i,j}$ un message de i vers j, indifférement une requête ou une réponse
– $\mathbb{Q}_i^{rcv}$ la liste des requêtes en attente de service dans l'objet actif i à un instant donné. C'est la queue de requêtes des objets actifs de ProActive.
– $stableState(i)$ un prédicat, vrai si l'objet actif i est dans un état stable, c'est à dire dans un état dans lequel un point de reprise peut être réalisé.

2.1.0.2 L'ajout du protocole

Nous noterons tout d'abord C_i^n le $n$ième point de reprise pris par l'objet actif i. Le protocole va nécessiter l'ajout de variables dans chaque objet actif, représentant son état par rapport à l'état global en construction, et l'ajout de variables aux messages pour communiquer cet état aux autres objets actifs.

On ajoutera donc aux objets actifs les valeurs et ensembles suivants :

– TTC_i le compte à rebour de point de reprise de l'objet actif i.
– $N_i^{current}$ le numéro du point de reprise courant de l'objet actif i, c'est à dire le numéro du dernier point pris par i.
– N_i^{max} le numéro maximum du prochain point de reprise de l'objet actif i.

- N_i^{min} le numéro minimum du prochain point de reprise de l'objet actif.
  $[N_i^{min}, N_i^{max}]$ est l'ensemble des points de reprise à réaliser lorsque l'occasion se présente, c'est à dire lorsque $stableState(i)$ est vrai. Il peut en effet y en avoir plusieurs à faire d'affilé si un objet est resté longtemps sans passer par un état stable (voir 1.3).
- I_i le compteur d'incarnation de l'objet actif i, c'est à dire le nombre de reprises qu'a fait cet objet depuis le début de l'exécution.
- $\mathbb{M}_i^{resend}(n)$ l'ensemble des messages à enregistrer dans le point de reprise n pour qu'ils soient réémis. Cependant, si les points suivants $(n+1, n+2, ...)$ sont réalisés d'affilé, cet ensemble leur sera aussi ajouté.
- $Q_{i,j}^{awaited}$ une requête en attente de l'émetteur i qui se trouve dans la queue de requêtes de j. Si j tente de servir cette requête, alors il est immédiatement bloqué. Lorsqu'une requête $Q_{i,j}$ arrive de i,
  - soit j est en attente sur une $Q_{i,j}^{awaited}$: j est alors débloqué et sert la requête $Q_{i,j}$
  - soit j n'est pas en attente mais il existe dans sa queue de requête des $Q_{i,j}^{awaited}$: la première rencontée dans l'ordre de la queue de requêtes est remplacée par $Q_{i,j}$.
  - soit j n'est pas en attente et il n'existe pas de $Q_{i,j}^{awaited}$ dans la queue de requêtes de j : $Q_{i,j}$ est placée normalement en fin de queue de requêtes.
- $\mathbb{H}_i(n)$ une liste de requêtes en attente (1.1.6). C'est l'historique des réceptions de requêtes du point de reprise n de l'objet actif i.
- $\mathcal{M}_n^{globalState}$ un message indiquant que l'état global n est formé, c'est à dire que tous les objets actifs du système ont réalisé le point de reprise n. Ce message indique au récepteur que les historiques des points de reprise indexés k tel que $k \leq n$ peuvent être clos.
- $\mathcal{M}_n^{rec}$ un message indiquant au récepteur de reprendre depuis le point de reprise n.

Les communications applicatives vont faire transiter les valeurs suivantes :
- $N_{\overrightarrow{M_{i,j}}}$ le numéro de point de reprise courant de l'objet actif i, émetteur du message $M_{i,j}$. Cette valeur est portée par le message.
- $I_{\overrightarrow{M_{i,j}}}$ le numéro d'incarnation courant de l'objet actif i, émetteur du message $M_{i,j}$, associé au numéro du point de reprise utilisé pour la dernière reprise (noté $I_{\overrightarrow{M_{i,j}}}.n$). Ce couple de valeurs est porté par le message.
- $H_{\overrightarrow{M_{i,j}}}$ est le numéro du dernier historique clos de l'émetteur du message $M_{i,j}$. Cette valeur est portée par le message.
- $N_{\overleftarrow{M_{i,j}}}$ le numéro de point de reprise courant du récepteur message $Q_{i,j}$. Cette valeur est renvoyée par le récepteur à la fin du rendez-vous.

On notera enfin les actions suivantes sur les messages et les ensembles :
- $M_{i,j} \in \mathbb{M}_i^{resend}(n)$ si l'émission du message $M_{i,j}$ doit être ajoutée dans la collection $\mathbb{M}_i^{resend}(n)$
- $\mathbb{M}_i^{resend}(m) \in C_i^n$ si la collection des messages à réémettre $\mathbb{M}_i^{resend}(m)$ est ajouté au point de reprise C_i^n

- $Q_{i,j}^{awaited} \in \mathbb{H}_i(n)$ si une requête en attente de l'émetteur i doit être ajoutée dans l'historique $\mathbb{H}_i(n)$.
- $Q_{i,j} \rightarrow Q_{i,j}^{awaited}$ si on remplace la requête $Q_{i,j}$ par $Q_{i,j}^{awaited}$ dans la queue de requêtes de j.

2.2 Protocoles

2.2.1 Protocole de points de reprise

Nous décrivons de façon formelle dans la figure 2.1 le déroulement du protocole durant une exécution sans panne. Pour cela, nous distinguons les différents événements survenant dans la vie d'un objet actif (émission, réception, service,...), et nous donnons les actions associées.

On notera que :
- La procédure **Initialisation de l'objet** i est appelée à la création de l'objet actif i.
- La procédure **Tentative de checkpoint** est appelée automatiquement quand $stableState$ est vrai.
- Le message $\mathcal{M}_n^{globalState}$, qui va indiquer que les historiques $\mathbb{H}_i(k)$ tel que $k \leq n$ peuvent être clos, pourra être reçu par un objet actif dans différents cas que nous ne détaillons pas dans ce protocole. Nous verrons dans la section 2.2.3 qui peut envoyer ce message, et à quel moment.

2.2.2 Protocole de reprise

Nous décrivons dans la figure 2.2 le déroulement du protocole de reprise, suite à la panne d'un noeud du système.

La reprise du système va être asynchrone, grâce à l'utilisation des requêtes en attente. Un message de reprise est diffusé par un processus (voir 2.2.2.3) aux objets actifs, puisque tout le système doit reprendre. Cependant, les objets peuvent reprendre avant d'avoir reçu ce message, puisque qu'une communication avec un objet qui a déjà repris entraîne la reprise de l'autre (Voir **Réception de** $M_{i,j}$).

2.2.2.1 Détection des fautes

On considère ici qu'un objet actif est en panne lorsqu'il n'est plus accessible, c'est à dire qu'une tentative de communication avec lui échoue. En particulier, le protocole ne prend pas en compte les fautes malignes (ou Byzantines), qui n'entrainent pas l'isolation de l'objet en faute.

La panne d'un objet actif est détectée dès lors qu'un autre objet essaie de communiquer avec lui : celui-ci déclenche alors la procédure de reprise du système. Le problème de la détection de panne n'est pas étudié plus loin dans le cadre de ce stage. On pourra donc supposer qu'il existe un protocole sous-jacent [30] qui détecte les pannes, et qui appelle la procédure de reprise.

FIG. 2.1 – *Protocole de points de reprise*

1. **Initialisation de l'objet** i
   $N_i^{current} = 0$
   $N_i^{min} = 0$
   $N_i^{max} = 1$
   $TTC_i = TTC\_INIT$
   $I_i = 0$
   Tentative de checkpoint

2. **Emission de** $M_{i,j}$
   $\underline{\text{si}}$ $N_{\overleftarrow{M_{i,j}}} > N_i^{current}$ $\underline{\text{alors}}$ $N_i^{max} = max(N_i^{max}, N_{\overleftarrow{M_{i,j}}})$ $\underline{\text{et}}$ $M_{i,j} \in \mathbb{M}_i^{resend}(N_i^{max})$

3. **Réception de** $M_{i,j}$
   $\underline{\text{si}}$ $I_{\overrightarrow{M_{i,j}}} == I_j$ $\underline{\text{alors}}$
   $\forall k$ $\underline{\text{si}}$ $\mathbb{H}_i(k)$ non-clos $\underline{\text{et}}$ $k < H_{\overrightarrow{M_{i,j}}}$ $\underline{\text{alors}}$ clôture $\mathbb{H}_i(k)$
   $\underline{\text{si}}$ $M_{i,j}$ est une $Q_{i,j}$ $\underline{\text{alors}}$ $\forall \mathbb{H}_i(k), Q_{i,j}^{awaited} \in \mathbb{H}_i(k)$
   $\underline{\text{si}}$ $N_{\overrightarrow{M_{i,j}}} > N_j^{current}$ $\underline{\text{alors}}$ $N_j^{max} = max(N_j^{max}, N_{\overrightarrow{M_{i,j}}})$
   $\underline{\text{retourne}}$ $N_j^{current}$
   $\underline{\text{sinon si}}$ $I_{\overrightarrow{M_{i,j}}} > I_j$ $\underline{\text{alors}}$
   bloquer i sur l'émission de $M_{i,j}$
   Reprise de i **sur** $I_{\overrightarrow{M_{i,j}}}.n$
   $\underline{\text{sinon}}$ ignorer le message

4. **Réception de** $\mathcal{M}_n^{globalState}$
   $\forall k$ $\underline{\text{si}}$ $\mathbb{H}_i(k)$ non-clos $\underline{\text{et}}$ $k < n$ $\underline{\text{alors}}$ clôture $\mathbb{H}_i(k)$

5. **Tentative de checkpoint**
   $N_i^{current} = max(N_i^{min} - 1, N_i^{current})$
   $\underline{\text{si}}$ $N_i^{max} > N_i^{current}$ $\underline{\text{alors}}$
   $\underline{\text{tant que}}$ $N_i^{max} > N_i^{current}$ $\underline{\text{faire}}$ **Réalisation du checkpoint** $C_i^{N_i^{current}+1}$
   $\underline{\text{sinon}}$
   $\underline{\text{si}}$ $TTC_i == 0$ $\underline{\text{faire}}$ **Réalisation du checkpoint** $C_i^{N_i^{current}+1}$

6. **Réalisation du checkpoint** C_i^n
   $\forall Q_{j,i} \in \mathbb{Q}_i^{rcv}$, $\underline{\text{si}}$ $N_{\overleftarrow{Q_{j,i}}} \geq n$ $\underline{\text{alors}}$ $Q_{i,j} \rightarrow Q_{i,j}^{awaited}$
   $\forall \mathbb{M}_i^{resend}(m)$ t.q. $m \geq n$, $\mathbb{M}_i^{resend}(m) \in C_i^n$
   effacer $\mathbb{M}_i^{resend}(n)$
   $N_i^{current} = n$
   ouvrir $\mathbb{H}_i(n)$
   $TTC_i = TTC\_INIT$

7. **Service d'une requête** $Q_{i,j}$
   $\underline{\text{si}}$ $N_{\overrightarrow{Q_{i,j}}} > N_i^{current}$ $\underline{\text{alors}}$
   $\underline{\text{si}}$ $stableState$ $\underline{\text{alors}}$
   $N_i^{max} = N_{\overrightarrow{Q_{i,j}}}$ $\underline{\text{et}}$ **Tentative de checkpoint**
   $\underline{\text{sinon}}$ $N_i^{min} = N_{\overrightarrow{Q_{i,j}}} + 1$

FIG. 2.2 – *Protocole de reprise*

1. **Détection d'une panne de l'objet actif** i
   placement de i
   Reprise de i **sur** n
   diffuse $\mathcal{M}_n^{rec}$

2. **Réception de** $\mathcal{M}_n^{rec}$ **par l'objet actif** i
   Reprise de i **sur** n

3. **Reprise de** i **sur** n
   arrêt de l'activité
   recupère C_i^n et $\mathbb{H}_i(n)$
   restaure l'état à patir de C_i^n
   concatène $\mathbb{H}_i(n)$ à $\mathbb{Q}_i^{rcv}$
   envoie tous les messages de $\mathbb{M}_i^{resend}(n)$
   reprise de l'activité

2.2.2.2 Placement des objets

Le problème du placement des objets tombés en panne n'est pas étudié ici [31]. On considera que le processus qui relance le système est capable de trouver un emplacement pour relancer l'objet fautif.

2.2.2.3 Processus de reprise

On suppose ici qu'il existe un processus au comportement simple qui gère la reprise du système. Ce processus connait à tout instant la ligne de recouvrement n du système (on rappelle que la ligne de recouvrement est le dernier état global cohérent dont tous les points de reprise ont leur historique clos). Il faut donc qu'il ait accès à tous les supports de stockage de point de reprise, de manière à déterminer la ligne de recouvrement courante.
Il est prévenu d'une panne par un objet actif qui ne réussit pas à communiquer avec un autre : il reçoit l'identifiant de l'objet fautif. Il lance alors la procédure de reprise, dans laquelle il doit diffuser un message à tous les objets actifs ; il doit donc référencer tous les objets du système.

2.2.2.4 Protocole

On notera que lorsqu'un objet reprend depuis un point de reprise, il n'a plus d'historique en cours d'enregistrement. Un historique sera réouvert au prochain point de reprise pris par l'objet.

2.2.3 Clôture des historiques

Les messages de type $\mathcal{M}_n^{globalState}$ qui déclenchent la clôture des historiques sont supposés reçus à un moment par l'objet actif. Nous allons voir ici comment gérer l'envoi de ces messages.

L'objectif principal est de clôturer le plus rapidement possible les historiques, de manière à ce que :

– l'état global le plus récent devienne rapidement une ligne de recouvrement qui pourra être utilisée en cas de panne. En effet, si les historiques sont trop longs, la ligne de recouvrement peut être assez ancienne bien qu'il y existe des états globaux récents, et une panne entrainerait un retour arrière important.
– la taille des données à stocker et/ou à envoyer sur le réseau soit minimal. Des historiques trop longs entraineraient des pertes de performances au moment de leur clôture.

Nous allons utiliser le processus de reprise : en effet, il identifie à tout moment la ligne de recouvrement, et peut donc, à chaque formation d'un état global, diffuser un message indiquant la fin de la construction de cet état. Cette méthode nous permet de clôturer les historiques au plus tôt.

Cepedant, elle ajoute dans l'exécution autant de communications multipoints additionnels qu'il y aura d'états globaux formés. Le surcoût engendré peut-être prohibitif. On peut alors utiliser une stratégie de "pulling" : l'information du numéro du dernier état global formé peut-être obtenue au moment où l'objet actif réalise un point de reprise, et engage donc une communication pour stocker ce point sur un support stable (voir 3.1.1). Les historiques seraient alors clos au moment des prises de points ; ils auraient donc une taille plus importante, mais les diffusions de messages seraient évitées.

Enfin, on peut imaginer utiliser une méthode hybride des deux précédentes : la diffusion pourrait alors être non-fiable, donc moins coûteuse, ou alors ne concerner qu'un sous-groupe du système. En effet, la clôture d'historique se propageant par envoi de messages applicatifs, si les objets actifs *centraux* (en terme de maillage de l'application) déclenchent une clôture d'historique, cette clôture peut rapidement se propager, et donc minimiser la taille d'un grand nombre d'historiques. Le mécanisme de "pulling" devra cependant être conservé pour assurer la clôture des objets communiquant très peu.

Chapitre 3

Mise en oeuvre et expérimentations

Nous donnons ici les éléments nécessaires à la mise en en oeuvre de ce protocole, puis nous présentons les premières évaluations de performance du protocole, à travers notre implémentation prototype.

3.1 Eléments de mise en oeuvre

3.1.1 Support de stockage stable

Il doit exister un support de stockage stable accessible par tous les objets actifs du système. Ce support peut cependant être distribué : pour éviter une trop grande contention, il peut exister plusieurs supports stables.

Ce support pourra être accédé par un objet actif pour trois raisons :

- lorsque celui-ci prend un point de reprise, ce point est alors envoyé au support stable pour pouvoir le récuperer en cas de panne,
- lorsqu'il clot un historique : cet historique est envoyé avec le numéro du point de reprise auquel il correspond, et est stocké avec ce point de reprise,
- lors d'une reprise du système : puisque tous les objets doivent reprendre, il faut donc que chaque objet récupère son point de reprise avec l'historique associé.

On voit que le dernier point en particulier peut poser un problème de contention au niveau de l'accès au support stable. Une optimisation simple est alors possible : lorsqu'un objet actif va prendre un point de reprise, il va, en plus de l'envoyer sur le support stable[1], conserver une copie en local. Ainsi, en cas de reprise depuis ce point, l'objet n'aura pas à communiquer avec le support stable.

Ces copies conservées en local pourront être régulièrement effacées, en fonction des réceptions des messages $\mathcal{M}_n^{globalState}$. On notera que la réception de ce message n'indique pas que la *ligne de recouvrement n* est formée, mais simplement l'état global (les historiques n ne sont pas tous clos). Bien-sûr, en cas de reprise depuis un point n, un objet actif pourra effacer toutes les copies de points dont les index sont différents de n.

1. Cet objet pourrait lui-même tomber en panne, et donc perdre sa copie locale.

3.1.2 Processus de reprise

Le processus de reprise doit être à tout instant accessible : le système ne supporte pas la panne de ce processus. Il doit donc être protégé contre les fautes, par exemple en étant répliqué, puisqu'il a un comportement simple. La difficulté est que ce processus doit connaitre tous les objets du système : il faut donc qu'un objet arrivant dans le système s'inscrive auprès de lui. Cette inscription peut-être réalisé lors de la première prise de point de reprise du nouvel arrivant.

Le ramassage des points de reprises devenus inutiles peut être réalisé par le processus de reprise, en utilisant la propriété de cohérence forte des ensembles de points portant le même numéro. Ce processus peut facilement déterminer régulièrement le dernier état global formé et, si tous les historiques de cet état sont clos, effacer tous les points de reprise dont l'index est inférieur à l'index de cet état.

On note qu'en cas de reprise du système depuis une ligne de recouvrement n, tous les points de reprise stockés dont l'index est différent de n peuvent être effacés. En effet, tous ceux dont l'index est inférieur sont devenus inutiles, et tous ceux dont l'index est supérieur seront regénérés par le suite.

3.1.3 Serveur de localisation

Le système va devoir utiliser un serveur de localisation pour rétablir les connexions entre les objets qui sont tombés en panne (et qui ont été replacés dans le système) et les autres. Le comportement de ce serveur est simple : il est informé à chaque reprise, par le processus de reprise, des objets tombés en panne, et de leur localisation, sous forme d'un couple (*identifiant unique, nouvelle localisation*). Lorsqu'un objet tente de communiquer avec un autre et que cette communication échoue, il ne va pas tout de suite déclencher une procédure de reprise. Il envoie d'abord une requête de relocalisation contenant l'identifiant de l'objet recherché. Si le serveur renvoie une nouvelle localisation, l'objet tente à nouveau de communiquer. Si la localisation renvoyée est la même que celle connue de l'objet, ou si aucune localisation n'est renvoyée, alors une procédure de reprise est lancée.

3.2 Implémentation et évaluation des performances

Nous avons réalisé une implémentation prototype dans la bibliothèque ProActive, qui nous a permis de vérifier le comportement du protocole et d'obtenir les premières évaluations de performance.

3.2.1 Choix d'implémentation

Nous avons d'abord choisi d'utiliser un serveur de point de reprise unique pour cette première implémentation : tous les objets connaissent l'adresse de ce serveur, et y stockent les enregistrements pris au cours de leur exécution. Nous avons aussi opté pour l'approche *pulling* en ce qui concerne la clôture des historiques, c'est à dire que le serveur de point de reprise renvoie à chaque communication le numéro du dernier état global formé (voir 2.2.3).

Nous n'avons pas implémenté de protocole de détection de panne, ni de protocole de relocalisation des objets après panne. Les tests du protocole de reprise ont donc été réalisé en relancant les applications *volontairement*, à partir du serveur de points de reprise.

3.2.2 Evaluation du surcoût pendant une exécution sans panne

Les évaluations du surcoût pendant l'exécution ont été réalisé sur une application de recherche du *n*ième nombre premier par le crible d'Erathostène avec une architecure de type "maître-esclaves".

3.2.2.1 Fonctionnement de l'application test

Le maître reçoit une requête de recherche du *n*ième nombre premier. Il va alors tester tout les nombres jusqu'à *n*, en envoyant chaque nombre à tous ses esclaves. Chaque esclave possède son propre ensemble de nombres premiers, et teste le nombre reçu par rapport à cet ensemble. Si tous les esclaves répondent que le nombre est premier, alors ce nombre est envoyé par le maître à l'un d'eux pour être ajouté à son ensemble de nombres premiers.

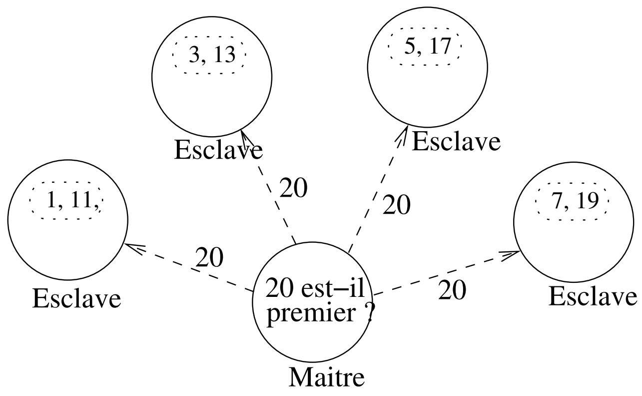

FIG. 3.1 – *Application de test : le crible d'Erathostène*

Cette procédure est réitérée jusqu'à ce que le *n*ième nombre premier soit trouvé. On note que après avoir envoyé le nombre à tester à tous ses esclaves, le maître attend *toutes* les réponses.

3.2.2.2 Contexte matériel

Les tests ont été réalisé sur un cluster de seize machines BiPentium III 1Ghz, avec le protocole de gestion de ressources LSF. Les machines virtuelles utilisées sont des JVM 1.4.1.

3.2.2.3 Premiers résultats

Nous présentons d'abord deux graphiques (figure 3.2) qui donnent la différence entre une exécution de l'application avec ProActive standard, et une autre avec l'ajoût du protocole de tolérance aux pannes. Ces tests ont été réalisés avec 4 machines esclaves, et un compte à rebours de prise de point de reprise (TTC_INIT) de 100 secondes.

Le premier présente les temps totaux d'exécution avec et sans tolérance aux pannes, en fonction du nombre premier recherché. Il présente aussi les temps cumulés de points de reprise, c'est à dire le temps moyen passé *par un esclave* à créer et à envoyer au serveur ses points de reprise. Le second graphique présente ce même résultat sous la forme du rapport des temps obtenus avec et sans tolérance aux pannes. Ce rapport est calculé de la façon suivante : $Rapport = (T_{avec} - T_{sans})/T_{sans}$.

On note que dans les deux cas, les résultats ont été dument vérifiés, et que le comportement de l'application est resté correct. Dans le cas de la version avec tolérance aux pannes, des tests de reprise depuis diverses lignes de recouvrement ont été réalisés avec succès.

On constate que le surcoût à l'exécution est assez faible, puisqu'il varie de 1% à 8% pour l'exécution la plus longue. Cette augmentation du surcoût s'explique par l'augmentation du temps total passé à prendre des points de reprise.

Cependant, la pente de cette augmentation diminue à mesure que le temps de calcul total augmente. Ceci s'explique par le fait que les temps de calcul des esclaves deviennent de plus en plus important et recouvrent donc les temps de communication : le surcoût de la communication s'efface alors, et seul reste le surcoût des prises de points.

Il nous a semblé intéressant aussi de voir l'évolution de ce surcoût en fonction du nombre de noeuds de l'application. On peut voir cette évolution dans la figure 3.3 qui montre le rapport entre les temps totaux d'exécution avec et sans tolérance aux pannes, pour la recherche du 3500ème nombre premier, en fonction du nombre d'esclaves utilisés. On constate ici que ce rapport augmente légèrement avec le nombre d'esclaves : ceci s'explique par le fait que le nombre de communications augmente et donc que le surcoût de la communication n'est plus recouvert par le temps de calcul.

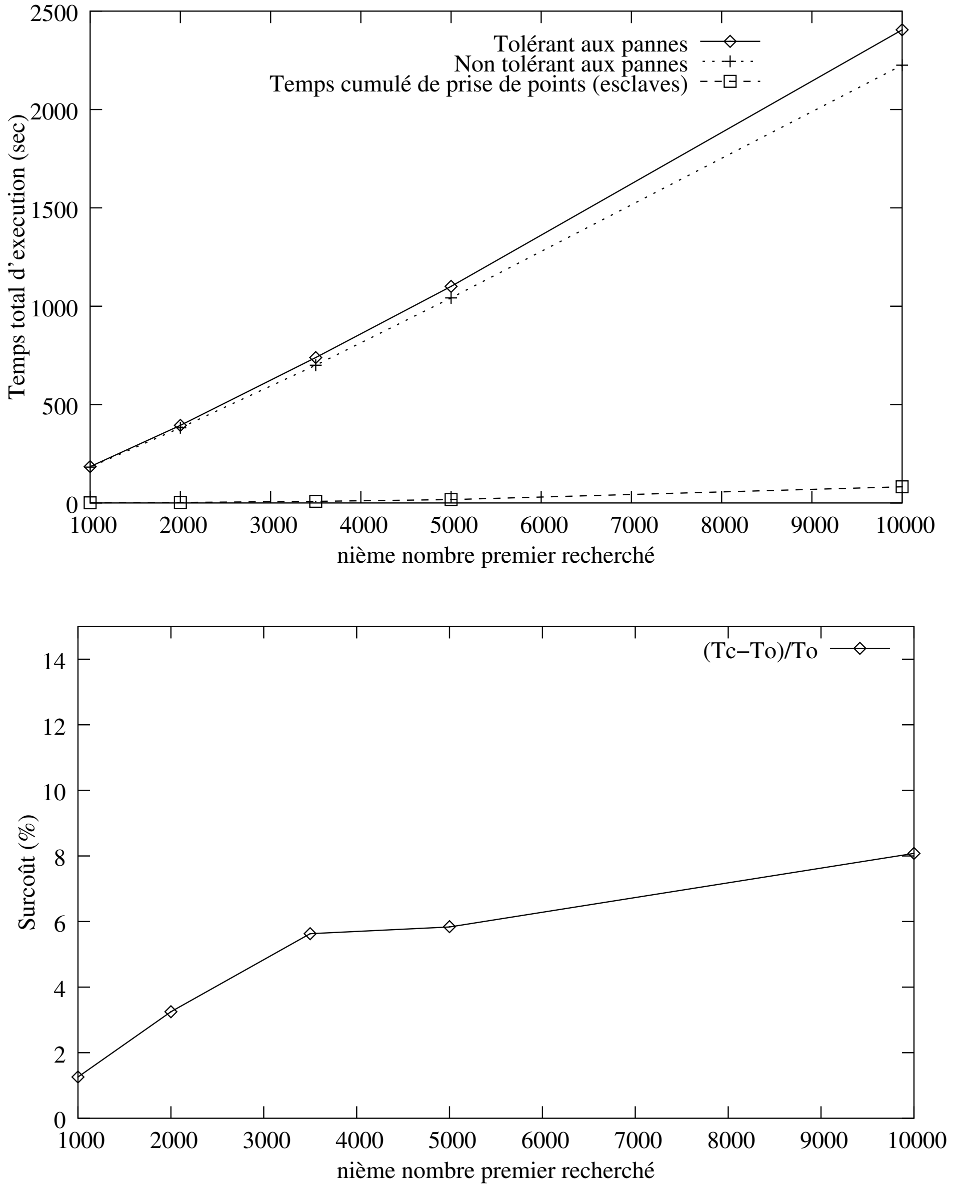

FIG. 3.2 – *Surcoût à l'exécution (4 esclaves, $TTC\_INIT = 100$ sec)*

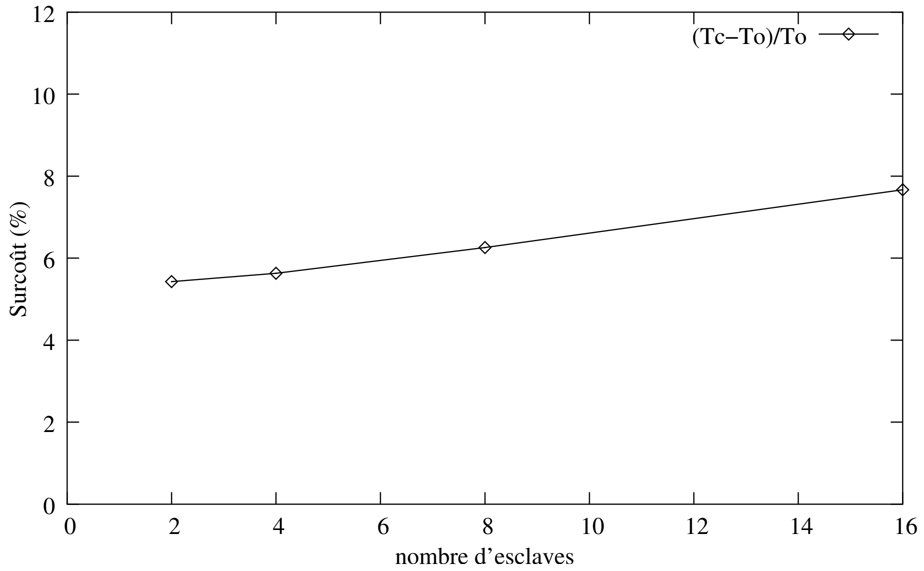

FIG. 3.3 – *Surcoût à l'exécution (Recherche du 3500ème nombre premier,* $TTC\_INIT = 100$ *sec)*

Chapitre 4

Conclusion et perspectives

Nous avons proposé un protocole de tolérance aux pannes adapté à un contexte de processus non-préemptifs, avec reprise complétement asynchrone et cohérence forte. Bien que conçu dans le cadre du modèle à objets actifs de la bibliothèque ProActive, notre approche peut s'adapter à un grand nombre d'intergiciels, en particulier à ceux écrits sur la plate-forme Java ou pour tout environnement sans préemption des processus. Elle permet aussi d'éviter d'avoir recours à des librairies spécialisées pour l'enregistrement de l'état d'une exécution, puisque la prise des points peut être reportée dans le temps, et être faîte au moment opportun. Enfin, de part l'utilisation d'un historique de réception, ce protocole présente une caractère hybride qui permettra de passer à l'échelle.

Pour vérifier le bon comportement de ce protocole, nous avons, dans un premier temps, réaliser un simulateur d'objets actifs. Les simulations obtenues nous ont permis de constater la formation régulière et rapide de lignes de recouvrements dans divers contextes (taux de communication, nombre de noeuds, maillage du système, ...).

Une implémentation prototype a ensuite été réalisée dans la bibliothèque ProActive. Elle nous a permis de constater à travers différentes évaluations de performances que l'utilisation additionnelle d'une technique de journalisation de messages n'a pas eu d'effets importants sur le surcoût induit par le protocole : ce surcoût d'environ 6% est, d'après les premières expérimentations, comparable aux meilleurs protocoles de points de reprise simples, qui assurent pourtant rarement une cohérence forte.

A court terme, nous terminerons la formalisation équationnelle du protocole débutée avec Ludovic Henrio de l'équipe OASIS durant ces quatre mois de stage. Nous nous attacherons aussi à prendre en compte et à tirer parti des autres caractéristiques de la bibliothèque ProActive, comme la mobilité des objets actifs, les communications de groupes ou bien encore les communications sécurisées.

A moyen terme, nous nous appliquerons à rendre le protocole adapté aux systèmes à large échelle. En effet, nous avons porté une attention particulière aux capacités de passage à l'échelle dans l'élaboration du protocole, avec, par exemple, une reprise du système complétement asynchrone en cas de panne. Cependant, de part ses caractéristiques de protocole de points de reprise, le protocole proposé peut s'appliquer aux systèmes de petite et moyenne taille, mais pas directement aux systèmes à large échelle dans lesquels le nombre de noeuds peut atteindre plusieurs milliers. Les techniques de journalisation de messages sont plus adaptées à ce type de système, mais restent en général très préjudiciables pour les systèmes répartis de petite et moyenne taille à cause de leur surcoût important à l'exécution. Il n'existe donc pas *un* protocole adapté à *tous* les systèmes distribués. C'est donc au protocole de s'adapter au système sur lequel il s'exécute.

C'est dans cette voie que nous désirons pousser plus loin notre travail. La capacité d'adaptation est inhérente au protocole proposé dans ce rapport, puisqu'il utilise déjà une association entre points de reprise et journalisation de message. Cette union est une base intéressante pour développer un unique protocole paramétré, autorisant points de reprise, journalisation optimiste et journalisation pessimiste, et cela en fonction des caractéristiques du système et de l'application au déploiment, voir en cours d'exécution afin d'obtenir un protocole *adaptatif*.

Bibliographie

[1] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 63–75, 1985.

[2] B. Randell. System structures for software fault tolerance. In *IEEE Transactions on Software Engineering, Vol SE1*, 1975.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM, vol. 21*, pages 558–565, 1978.

[4] R.H.B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. In *IEEE Transactions on Parallel and Distributed Systems, Vol 6*, pages 165–169, 1995.

[5] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. In *IEEE Transactions on Parallel and Distributed Systems, Vol 10*, 1999.

[6] R.H.B. Netzer J.M. Hélary, A. Mostefaoui and M. Raynal. Communication-based prevention of useless checkpoints in distributed computations. In *Proc. 16th IEEE Symposium on Reliable Distributed Systems*, pages 183–190, 1997.

[7] R.H.B. Netzer J.M. Hélary and M. Raynal. Consistency issues in distributed checkpoints. In *IEEE Transactions on Software Engineering, Vol 25*, 1999.

[8] A Mostefaoui J.M. Hélary and M. Raynal. Communication-induced determination of consistent snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):865–877, 1999.

[9] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[10] D.Manivannan and M.Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Proceedings of the 16th ICDCS*, 1996.

[11] Ciuffoletti Briatico and Simoncini. A distributed domino-effect free recovery algorithm. In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, 1984.

[12] D. Conan et G. Bernard. La reprise sur erreur par recouvrement arrière automatique dans les systèmes répartis. In *Parallélisme et répartitions (coll. Parallélisme, réseaux et répartition)*, 1998.

[13] T. Dillon J. Cao, G. H. Chan and W. Jia. Checkpointing and rollback of wide-area distributed applications using mobile agents. In *IPDPS '02*, 2002.

[14] Franco Zambonelli. On the effectiveness of distributed checkpoint algorithms for domino-free recovery. In *HPDC*, pages 124–131, 1998.

[15] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *FTCS-24: 24th International Symposium on Fault Tolerant Computing*, pages 298–309, Austin, Texas, 1994. IEEE Computer Society Press.

[16] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The 7th annual international symposium on fault-tolerant computing*. IEEE Computer Society, 1987.

[17] D. F. Bacon R. E. Strom and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proc IEEE Fault-tolerant Computing Symp.*, pages 44–49, 1988.

[18] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[19] G. Bosilca A. Bouteiller F. Cappello S. Djailali G. Fedak C. Germain T. Herault P. Lemarinier O. Lodygensky F. Magniette V. Neri and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *ACM/IEEE International Conference on Supercomputing SC 2002*, 2002.

[20] L. Alvisi S. Rao and H.M.Vin. The cost of recovery in message logging protocols. In *IEEE Transactions on Knowledge and Data Engineering, Vol 12, No 2*, 2000.

[21] D. Ely W.J. Bolosky, J.R. Douceur and M. Theimer. Faisability of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM international conference on Measurement and modeling of computer systems, SIGMETRICS*, pages 34–43, 2000.

[22] Linux Networx. http://www.linuxnetworx.com/.

[23] D. Caromel. Towards a method of object-oriented concurrent programming. In *CACM, Communications of the ACM, Volume 36, Number 9*, pages 90–102, 1993.

[24] J. Vayssière. Programmation parallèle et distribuée en Java. Conception et implémentation de Java//. Master's thesis, Supélec, 1997.

[25] Sun Microsystems. Java object serialization specification. http://java.sun.com/.

[26] S. Bouchenak and D. Hagimont. Pickling threads state in the java system. In *Third European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal.*, 1999.

[27] D. Caromel, L. Henrio, and B. Serpette. Asynchronous sequential processes. Technical report, INRIA Sophia Antipolis, 2003. RR-4753.

[28] Equipe OASIS. *ProActive PDC, installation and User Guide*. http://www-sop.inria.fr/oasis/ProActive/.

[29] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems, Vol 3, No 3*, pages 204–226, 1985.

[30] T. Deepak Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[31] P. Sens. Contribution à l'intégration de la tolérance aux fautes dans les environnements répartis, 2002. Thèse d'Habilitation de l'Université Pierre et Marie Curie.

[32] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.