Un protocole de tolérance aux pannes pour objets actifs non préemptifs

Françoise Baude — Denis Caromel — Christian Delbé
Ludovic Henrio

Equipe Oasis, INRIA - CNRS - I3S
2004, route des Lucioles
F-06902 Sophia Antipolis
{francoise.baude,denis.caromel,christian.delbe,ludovic.henrio}@inria.fr

RÉSUMÉ. *Les protocoles de points de reprises induits par messages semblent être l'approche la plus adaptée aux applications s'exécutant sur des systèmes hétérogènes avec un faible taux de panne. Mais ces protocoles supposent qu'il soit toujours possible de prendre un point de reprise de manière préemptive, avant la prise en compte d'un message.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 des processus. A la différence de nombreux protocoles existants, ce protocole assure la cohérence forte des lignes de recouvrement formées, et permet une reprise complètement asynchrone du système réparti en cas de panne.*

ABSTRACT. *Communication induced checkpointing protocols seem to be the most adapted approach for applications running on heterogenenous systems with low failure rate. However, these protocols make the assumption that it is always possible to preemptively trigger a checkpoint before delivering a message.We offer, within the framework of an active objects model, a communication induced checkpointing protocol, adapted to the non-preemptibility of processes. Unlike many protocols, this one ensures strong consistency of recovery lines, and enables a fully asynchronous recovery of the distributed system after a failure.*

MOTS-CLÉS : *tolérance aux pannes, points de reprise, journalisation de messages, objets actifs.*

KEYWORDS: *fault tolerance, checkpointing, message logging, active objects.*

RTSI - TSI. Volume 24 – n° 10/2005, pages 1199 à 1226

1. Introduction

La tolérance aux pannes dans les systèmes répartis est un domaine qui a été largement étudié depuis une trentaine d'années (Conan *et al.*, 1998; Elnozahy *et al.*, 1999). Ces études ont donné lieu à un nombre important de protocoles. Pourtant, on trouve encore à ce jour très peu d'intergiciels de communication intégrant de la tolérance aux pannes par recouvrement arrière[1]. Ce manque peut s'expliquer par la chute de performance qu'entraînent certains protocoles de tolérance aux pannes, mais aussi par la complexité de mise en œuvre dans des conditions *réelles* des protocoles proposés : ils sont le plus souvent théoriques, et requièrent pour fonctionner des hypothèses qui ne sont pas compatibles avec les systèmes concrets.

Nous nous intéressons aux applications s'exécutant sur des systèmes hétérogènes avec un faible taux de panne. Les protocoles de points de reprises induits par messages semblent être l'approche la plus adaptée à ce contexte.

Ces protocoles supposent qu'il soit toujours possible de prendre un point de reprise avant la prise en compte d'un message. Cette hypothèse implique donc la préemptivité des processus : l'activité doit pouvoir être stoppée à tout instant durant l'exécution.

Dans le cas de l'intergiciel ProActive (Caromel *et al.*, 1998), un intergiciel de communication 100% Java, les activités communiquantes ne peuvent pas être interrompues de manière préemptive car elles sont composées en partie d'un *thread* Java. Cependant, il existe des états appelés *états stables* dans lesquelles un point de reprise est possible.

Dans cet article, nous adaptons un protocole de tolérance aux pannes par points de reprise induits par messages pour la bibliothèque ProActive : les points de reprises déclenchés par la réception d'un message peuvent être repoussés jusqu'au prochain état stable. De plus, le protocole doit maintenir l'ordre FIFO point-à-point des messages qui est une caractéristique inhérente du modèle ProActive.

Tolérance aux pannes dans les systèmes répartis

On distingue principalement deux familles de protocoles utilisant les points de reprise : elles se différencient sur la façon de *synchroniser* les points entre eux afin de former des états globaux cohérents, ou *ligne de recouvrement*. La première approche synchronise de manière explicite les prises de points à l'aide de messages spécifiques. L'article fondateur (Chandy *et al.*, 1985) propose un tel algorithme, basé sur l'utilisation de messages «marqueurs». La deuxième approche utilise les messages de l'application pour coordonner les points de reprise de manière à former des états globaux cohérents. On parle de points de reprise *induits par messages* (Briatico *et al.*, 1984; Lai *et al.*, 1987).

1. On parle de tolérance aux pannes *par recouvrement arrière* pour la distinguer de la tolérance aux pannes par réplication (Cooper, 1985).

D'autre part, la tolérance aux pannes peut être obtenue par *journalisation des messages* : tous les messages qui circulent sont enregistrés pour pouvoir être rejoués en cas de panne. Il existe trois types de journalisation : *pessimiste* (Bouteiller *et al.*, 2003), *optimiste* (Strom *et al.*, 1985) et *causale* (Alvisi *et al.*, 1998). Ces approches se distinguent par le type d'information enregistré sur support stable et la fréquence de ces enregistrements.

Bien que la journalisation de message soit intrinsèquement adaptée au cas des processus non-préemptifs, les propriétés des protocoles par points reprises induits par messages les rendent plus adaptés à nos besoins. Le protocole proposé par Chandy et Lamport (Chandy *et al.*, 1985) est adapté aux processus non-préemptifs mais implique un surcoût trop important en communications. Les points de reprise forcés dans les protocoles de Lai et Yang (Lai *et al.*, 1987) ou de Briatico (Briatico *et al.*, 1984) impliquent une préemptivité sur les processus. Dans le contexte de processus non-préemptifs, pour implémenter une prise de vue cohérente du système, il suffirait que le processus recevant un message déclenchant un point de reprise forcé repousse la prise en compte de ce message au prochain état stable. Mais ce décalage introduit des problèmes de cohérence des états globaux formés et peut affecter l'ordre des messages, en particulier dans le cadre de communications FIFO.

Nous proposons donc une adaptation du protocole de Briatico aux processus non-préemptifs : un protocole induit par messages hybride utilisant un mécanisme de journalisation optimiste pour rendre cohérent les états globaux formés par des points de reprise retardés.

2. Objets actifs en Java : ProActive

Notre travail se place dans le cadre de 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 adapté.

2.1. *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 (Caromel, 1993), c'est-à-dire un objet qui possède une activité propre. Elle fournit en particulier :

– des appels de méthodes asynchrones avec futurs transparents, c'est-à-dire une communication par messages (requête et réponse),

– de la mobilité faible, c'est-à-dire non préemptive,

– des communications de groupe.

ProActive utilise un protocole à méta-objet pour réifier les objets et les appels de méthodes entre objets actifs (Caromel *et al.*, 2001). On appellera «objet réifié» l'objet Java standard rendu actif par l'ajout de méta-objets, et «objet actif» l'ensemble composé de l'objet réifié et des méta-objets.

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 réifié en une requête qui est envoyée au récepteur pour y être stockée dans sa *queue de requêtes* et qui sera *servie* ultérieurement. Lors de cette émission, si la requête nécessite le retour d'un résultat, l'objet émetteur reçoit un *futur* pour lui permettre de continuer son exécution. Ce futur correspond 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.

Si la réification des communications permet d'avoir des appels de méthodes asynchrones, 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 déposée dans la queue de requêtes du récepteur. De même, lors du retour d'un résultat, l'émetteur de ce résultat ne peut continuer son exécution que lorsque le futur attendant ce résultat a bien été mis à jour.

On notera dans la suite :

– $Q_{i,j}$ une requête de i vers j
– $R_{j,i}$ une réponse de j vers i
– $M_{i,j}$ un message de i vers j, indifféremment une requête ou une réponse
– $\mathbb{Q}_i^{rcv}$ la queue des requêtes en attente de service dans l'objet actif i.

2.2. *Propriétés*

2.2.1. *Changements d'état*

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

La réception d'une requête ou d'une réponse par un objet actif n'a pas le même impact au niveau des objets eux-mêmes : la réception d'une requête n'a d'effet sur les objets qu'au niveau méta (seule la queue de requêtes va être modifiée).
La réception d'une réponse modifie en plus les objets au niveau de l'application (mise à jour d'un futur). Cette mise à jour va très 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.

On note que si l'ordre des réceptions de requêtes est bien sûr déterminant pour l'état de l'objet actif puisqu'il peut induire un ordre de service, une étude formelle du modèle de ProActive (Caromel *et al.*, 2004) a montré que l'ordre de réception des réponses n'a absolument *aucune influence* sur l'état d'un objet actif.

2.2.2. *Existence d'états stables*

ProActive est entièrement écrit en Java : un objet actif est en fait un ensemble d'objets Java standards. Nous allons donc pouvoir utiliser la sérialisation (SunMicrosystems, 1997) proposée par Java pour avoir une représentation des objets qui puisse être stockée et récupérée en cas de reprise du système. Lorsqu'on va vouloir réaliser un enregistrement d'un objet, on va le sérialiser, mais l'objet `Thread` représentant le fil d'exécution de l'activité ne peut pas faire partie de l'enregistrement.

La conséquence est que les prises de points de reprise ne seront possibles qu'à certains moments de l'exécution, moments que l'on appellera *états stables*. Ces moments sont ceux pendant lesquels l'objet actif n'est pas en train de servir une requête, comme par exemple entre deux services consécutifs.

On notera $stableState(i)$ 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.

2.2.3. *Rendez-vous et Communications FIFO*

Les communications (requêtes et réponses) dans ProActive sont asynchrones *avec rendez-vous*. La principale conséquence est que les communications entre objets actifs sont systématiquement FIFO point à point. De plus, ce rendez-vous étant réalisé à l'aide d'un message d'acquittement, il est 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*.

2.2.4. *Promesses de requêtes*

Nous introduisons dans ProActive un nouveau mécanisme : les *promesses de requête*. Ces promesses sont en fait des requêtes «vides», qui doivent être remplies avec une requête normale provenant d'un émetteur donné ; une promesse de requête contient donc simplement l'identificateur de l'émetteur attendu. On notera $Q_{i,j}^{awaited}$ une promesse de requête de l'émetteur i qui se trouve dans la queue de requêtes de j.

Le service d'une promesse de requête $Q_{i,j}^{awaited}$ peut déclencher une attente par nécessité. 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 rencontrée 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.

L'utilisation de promesses de requête permet de forcer de façon paresseuse et asynchrone l'ordre de réception des requêtes en cas de reprise du système.

2.3. *Hypothèses*

2.3.1. *Types de défaillance*

Les pannes tolérées sont les pannes *franches* ; les objets actifs sont de types *fail-stop* (Schlichting *et al.*, 1983), c'est-à-dire que toute défaillance se traduit par un arrêt complet de l'activité de l'objet. Cet arrêt est supposé être détecté au bout d'un temps fini arbitrairement long par un détecteur de défaillance de type (Chandra *et al.*, 1996). Enfin, nous supposerons qu'il existe toujours une machine hôte disponible sur laquelle redémarrer un objet actif défaillant.

2.3.2. *Support de stockage*

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

2.3.3. *Processus de reprise*

Il doit exister un processus particulier appelé «processus de reprise» qui a accès au support de stockage stable ainsi qu'à l'ensemble des objets actifs formant le système.

2.3.4. *Serveur de localisation*

Le système utilise un serveur de localisation pour rétablir les connexions entre les objets qui sont tombés en panne (et qui ont donc été replacés dans le système) et les autres. Le comportement de ce serveur est simple. Il est informé :

– à chaque nouveau placement par le processus de reprise d'un objet tombé en panne de la nouvelle localisation de ce dernier, sous forme d'un couple (*identifiant unique de l'objet actif, nouvelle localisation*)

– à chaque reprise *après panne* d'un objet actif de la reprise de *l'activité* de cet objet.

De cette manière, le serveur de localisation pourra déterminer si un objet est inaccessible parce qu'il est en panne, ou parce qu'il est *en cours de reprise*.
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 au serveur contenant l'identifiant de l'objet recherché. Son appel est bloqué par le serveur tant que l'objet recherché est en cours de reprise. 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.

Le processus de reprise ainsi que le serveur de localisation doivent être protégés contre les fautes, par exemple en étant répliqués, puisqu'ils ont un comportement simple.

2.4. *Notations*

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

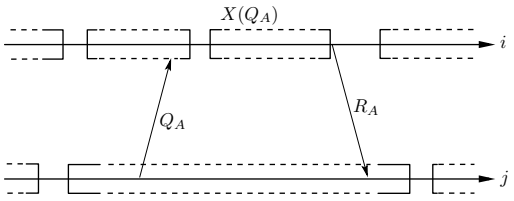

Figure 1. *Objets actifs communiquants*

On a sur la figure 1 deux objets actifs i et j qui communiquent. L'objet actif 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 ($X(Q_A)$, exécution de la requête Q_A), et enfin une réponse (R_A), résultat du service de la requête, 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 représentée que par une simple ligne. On notera que les services de requêtes peuvent être annotés ou non avec la requête servie (ici, $X(Q_A)$ sur i).

On voit que la réception d'une requête (ici, Q_A) «n'entre pas» dans le rectangle : la réception de requête n'a pas d'impact sur l'état de l'objet au niveau applicatif. À l'inverse, la réception d'une réponse (R_A) 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 2). Sa représentation inclut son numéro (n), l'état de la queue de requêtes de l'objet actif en indice ($[Q_A, Q_C, Q_D]$), ainsi qu'une autre queue de messages en exposant ($]Q_B, R_E[$). Cette autre queue sera ajoutée à chaque objet actif pour les besoins du protocole. Elle contiendra (cf. § 3.1.2) tous les messages applicatifs qui devront être réémis avant le

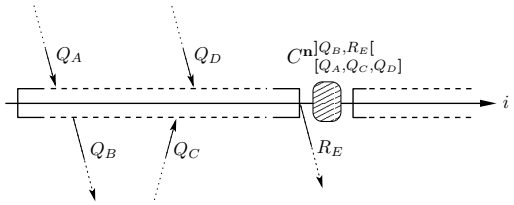

Figure 2. *Objet actif prenant un point de reprise*

redémarrage de l'activité de l'objet en cas de reprise depuis ce point de reprise n. Si l'une de ces queues est vide, on la note $[\emptyset]$ ou $]\emptyset[$.

3. Principes

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

Le principe est le suivant : le protocole fait régulièrement prendre à chaque objet actif des points de reprise, à l'expiration d'un compte à rebours temporel, réinitialisé ensuite. Chaque point de reprise pris est numéroté (dans un ordre strictement croissant), 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 propriété permet de déterminer la ligne de recouvrement en cas de panne.

On notera par la suite :

– C_i^n le n-ième 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.

– TTC_i le compte à rebours de point de reprise de l'objet actif i (Time To Checkpoint).

– $N_{\overrightarrow{M_{i,j}}}$ le numéro de point de reprise courant de l'objet actif i porté par un message de i vers j.

– $N_{\overline{M_{i,j}}}$ le numéro de point de reprise courant de j porté par l'acquittement de j vers i.

A la différence de (Manivannan *et al.*, 1996) et (Briatico *et al.*, 1984), lorsqu'un objet actif prend un point de reprise forcé, son compte à rebours déclenchant la prochaine prise de point est réinitialisé. Cette modification permet de réduire considérablement le nombre total de points de reprise pris durant une exécution, et donc de réduire le surcoût du protocole.

3.1. *Messages orphelins et en transit*

3.1.1. *Cohérence faible*

On a vu dans la section 2.2.2 que les objets actifs dans 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 s'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 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 3, on simule la réception du message *après* la prise du point. Bien sûr, cette modification de la queue de requête a lieu uniquement sur l'image de l'objet actif utilisée pour le point de reprise : 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.
Nous verrons dans la section 3.2.1 que, dans certains cas, l'ordre de réception des requêtes doit être conservé. C'est pour cette raison qu'au lieu d'être définitivement exclue de la queue de requête, cette requête sera en réalité remplacée par une promesse de requête, de manière à pouvoir être réinsérée à la même position après une reprise (cf. § 3.2.2).

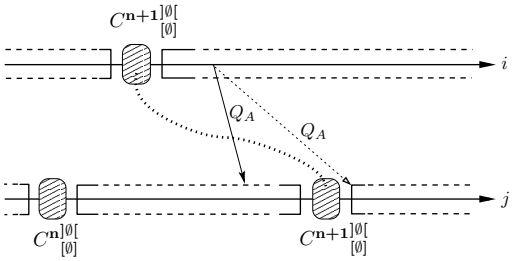

Figure 3. *Q_A est exclue de la queue de requêtes lors de la prise du point $n+1$ sur j*

– si le message reçu est une réponse, on ne peut pas «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, le récepteur recevra donc *une nouvelle fois* cette réponse lors de la deuxième exécution puisque l'émetteur reprendra lui aussi depuis ce point.
Le mécanisme des futurs permet à l'objet récepteur d'automatiquement ignorer cette deuxième occurrence de la même réponse, puisqu'il n'existera plus de futur associé à cette réponse. Il faudra alors assurer que la deuxième occurrence soit strictement la même que la première, puisque le récepteur n'en tiendra pas compte. En effet, si les services de requête qui ont précédés l'envoi de cette réponse ne sont pas réexécutés dans le même ordre, la valeur de la deuxième occurrence peut alors être différente de la première.

Ces deux traitements spécifiques des messages nous permettent de contourner le problème de la non préemptivité des objets actifs afin d'obtenir l'absence de message orphelin. Nous appliquons dans le cas de processus non-préemptifs, et en particulier dans le cas de ProActive, une méthode de points de reprise induits par messages et basé sur les index.

3.1.2. *Cohérence forte*

Nous allons voir que le rendez-vous des communications de ProActive va nous permettre d'assurer la cohérence forte des lignes de recouvrement. Selon Hélary, Netzer et Raynal (Hélary *et al.*, 1999), si on applique un protocole assurant l'absence de message orphelin en inversant les récepteurs et les émetteurs des messages, ce protocole assure alors l'absence de message en transit. En d'autres termes, si on applique à la fois un protocole assurant la cohérence des lignes de recouvrement et *son dual*, alors on assure la cohérence forte des lignes de recouvrement.

Or, on peut voir le rendez-vous comme un message d'acquittement, c'est à dire un message qui est envoyé du récepteur à l'émetteur d'un message applicatif. Nous pouvons donc, dans le cadre de ProActive, appliquer le protocole assurant l'absence de message orphelin en utilisant les messages applicatifs, et le dual de ce protocole en utilisant les messages d'acquittement : lors de la réception d'un message applicatif, le récepteur se synchronise sur l'émetteur grâce au numéro porté par ce message. Grâce au message d'acquittement, le récepteur renvoie 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 se synchronisent au cours d'une seule communication, et assurent une cohérence forte entre eux.

Supposons que i envoie une requête $Q_{i,j}$ (ou une réponse, le traitement dans ce cas est équivalent), et qu'il reçoive en retour $N_{\overline{M_{i,j}}} = n + 1$ supérieur à son dernier point de reprise numéroté n. L'objet i devrait alors, pour conserver la cohérence forte

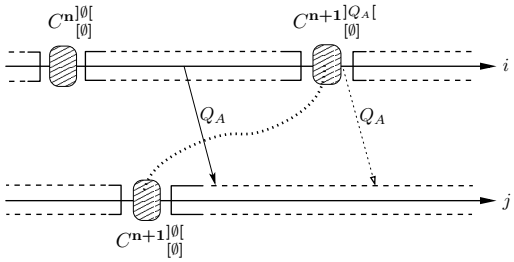

Figure 4. *Q_A est enregistrée pour réémission dans le point de reprise $n+1$ de i*

de la ligne $n+1$, prendre un point de reprise[2] *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 4, on simule avec l'enregistrement pour réémission de Q_A dans le point de reprise $n+1$ ($]Q_A[$) l'envoi de Q_A *après* la prise du point de reprise $n+1$.

On notera $\mathbb{M}_i^{resend}(n)$ la liste 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 successivement, cette liste leur sera aussi ajoutée (cf. § 3.1.3).

3.1.3. *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 5 : lorsqu'il arrive à un état stable, il doit prendre deux 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. On notera que, dans la pratique, lorsqu'un objet actif doit prendre plusieurs points de reprise successifs, il n'en prend en fait qu'un seul. En effet, seules les informations associées aux points changent lors de prises consécutives : l'état de l'objet actif n'est enregistré qu'une seule fois.

Avec la politique de service par défaut de ProActive, un objet actif est dans un état stable avant chaque service de requête. Il peut donc toujours prendre un ou plusieurs points de reprise avant un service de requête. Cependant, l'utilisateur peut modifier cette politique (OASIS, 2003) et réaliser, par exemple, des services imbriqués : il est alors possible qu'un objet actif doive servir une requête, *sans pouvoir prendre de point*

2. voire plusieurs (cf §3.1.3).

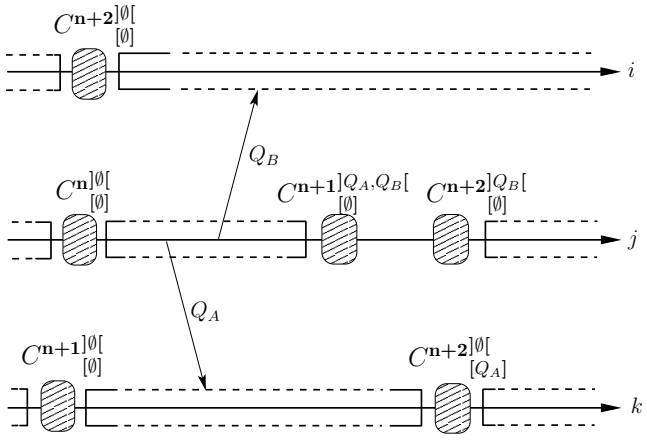

Figure 5. *L'objet actif j doit faire deux points de reprise d'affilé*

de reprise avant ce service : ceci a pour effet de rendre impossible la prise d'un point avant certains services de requêtes. Nous supposons ici que cette situation est assez rare pour éviter un effet domino. Cette hypothèse est réaliste car une modification de la politique de service, par défaut FIFO, peut être généralement évitée. De plus, la modification de la politique n'implique pas forcément l'utilisation de requêtes imbriquées.

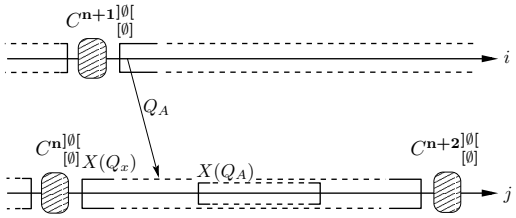

Figure 6. *L'objet actif j ne peut pas prendre le point $n+1$*

Prenons le cas de la figure 6 : la requête Q_A indique à j qu'il doit prendre dès que possible le point de reprise $n+1$. Mais, *avant de parvenir à un état stable*, j doit servir Q_A : il ne peut donc pas prendre de point avant ce service. Une fois dans un état stable (après le service de Q_x), j ne peut plus prendre le point $n+1$, sinon Q_A serait orphelin par rapport à l'état global $n+1$, qui ne serait donc plus cohérent. En effet, on ne peux plus annuler la réception de Q_A (cf. § 3.1.1), puisque Q_A est déjà servie et n'est donc plus dans la queue de requêtes de j. Dans ce cas, j prend directement le point $n+2$, et la ligne $n+1$ ne pourra donc pas être complétée ; c'est le *seul cas* dans

lequel des points de reprise (ici, les points $n+1$ sur les autres processus) deviennent inutilisables, c'est-à-dire qu'ils ne peuvent pas être utilisés pour la reprise du système.

Pour pouvoir prendre en compte les points de reprises multiples et impossibles, nous introduisons les valeurs suivantes :

– 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 i.

$[N_i^{min}, N_i^{max}]$ est donc l'ensemble des points de reprise à réaliser dès que possible, c'est-à-dire lorsque $stableState(i)$ est vrai.

3.2. *Maintien de la causalité et équivalence*

Par la suite, nous entendrons par message *réémis* un message qui a été enregistré dans un point de reprise pour réémission et qui est envoyé lors de la reprise du système, et message *rejoué* un message qui est envoyé par un processus lors de sa rééxecution après une reprise, mais qui avait déjà été joué lors de l'exécution précédente.

Le protocole se base donc sur la réémission de certains messages en cas de reprise du système. Nous allons voir dans cette section que ces réémissions peuvent poser des problèmes :

– au niveau des dépendances causales entre les messages réémis et rejoués ;

– au niveau des réponses rejouées, qui pourraient être différentes.

3.2.1. *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 équivalentes, *au moins jusqu'à un certain point dans le temps*. D'abord, regardons le cas de la figure 7. 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 servirait alors les requêtes dans un ordre différent, ce qui pourrait rendre la deuxième occurrence de la réponse R_B *différente* de la première. Il faut donc que la deuxième exécution de j soit équivalente à la première (Condition **i)** *jusqu'au moment où, dans la première exécution, on pouvait être sûr qu'une réponse envoyée par j après le point $n+1$ ne serait pas reçue avant le point $n+1$ du récepteur*.

La deuxième raison qui oblige à avoir des exécutions équivalentes est due 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 8 : lors de l'exécution initiale, j recevra forcément Q_A avant Q_C car la réception de Q_C par j est probablement 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.

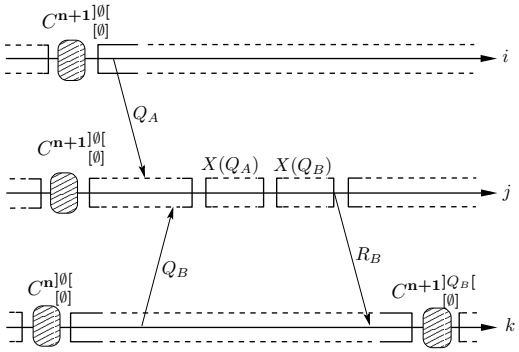

Figure 7. *Un changement de l'ordre relatif de Q_A et Q_B pourrait changer R_B*

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 depuis le point $n+1$ soit équivalente à la première (Condition **ii**) *jusqu'au moment où, dans la première exécution, on pouvait être sûr qu'il n'y aurait plus par la suite de requête à l'intention de j qui soit enregistrée pour réémission dans un point de reprise $n+1$ sur d'autres objets actifs.*

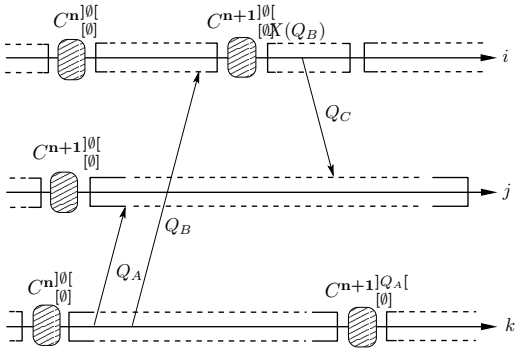

Figure 8. *La réception sur j de Q_C doit précéder celle de Q_A*

La conjonction des conditions **(i)** et **(ii)** données ci-dessus revient 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 : 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 (Strom *et al.*, 1985), c'est-à-dire qu'à partir d'un état précis et d'une histoire de messages, toute exécution d'un processus est équivalente. En effet, les exécutions devant être dans certaines conditions équivalentes, il faut qu'un objet recevant deux fois les mêmes messages dans le même ordre ait strictement le même comportement à chaque fois.

– Il faut assurer que durant deux exécutions différentes à partir du même état, les mêmes messages arrivent dans le même ordre pour tous les objets actifs pendant une période donnée : cet ordre sera garanti par un *historique de réception*.

Notons que ces implications se retrouvent habituellement dans les protocoles de tolérance aux pannes par journalisation des messages : on voit que le protocole proposé comporte, du fait de la contrainte de non préemptivité, un caractère *hybride* entre les points de reprise induits par messages et la journalisation des messages sur le récepteur. Nous reviendrons sur cet aspect lors de la conclusion.

3.2.2. *Historique*

Nous allons associer à tout point de reprise un historique de réception, c'est-à-dire une liste chronologique des messages reçus. On notera $\mathbb{H}_i(n)$ l'historique associé au point de reprise n de l'objet actif i : un historique sera ouvert à chaque prise de point de reprise. Il est important de noter qu'il peut donc y avoir *plusieurs* historiques ouverts au même moment sur un objet actif. Les caractéristiques du modèle à objet actif considéré permettent à cet historique d'avoir les propriétés suivantes :

– Il est constitué uniquement des réceptions de *requêtes*. En effet, Caromel, Henrio et Serpette ont montré formellement dans (Caromel *et al.*, 2004) que l'ordre de réception des réponses n'a pas d'impact sur le comportement d'un objet actif. Cette propriété est due principalement 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* (sans aucun identifiant de requête, ni aucun paramètre effectif). Grâce aux propriétés de communication FIFO point-à-point et de déterminisme par morceaux des objets actifs, cette information est suffisante pour fixer complètement l'ordre des réceptions (Caromel *et al.*, 2004).
Prenons l'exemple de la figure 9 : l'historique n de l'objet actif i au moment noté $\diamondsuit$ est de la forme $\{j, j, k\}$, où j et k sont des identifiants uniques d'objets actifs. Le premier j indique la réception de la requête Q_B émise par j. Le deuxième j indique la réception de la requête Q_D émise par j. L'entier k représente la réception de la requête Q_E émise par k. En cas de reprise depuis le point n, i bloquera le service de la prochaine requête tant qu'il ne s'agit pas d'une requête de la part de j, même si une requête de la part de k est déjà arrivée. Si i reçoit deux requêtes consécutives de j, la propriété FIFO point-à-point nous assure que la première arrivée correspond à la première occurrence de l'identifiant j dans l'historique de i. Il n'y a pas dans l'historique de trace de la réception de la réponse R_C.

En pratique, cet historique est une liste de promesse de requête. Le mécanisme d'attente par nécessité lié au service de ce type de requête permet d'assurer le même ordre de service de requête lors de la réexécution.

3.2.3. *Clôture des historiques*

Chaque historique est fini dans le temps ; le protocole doit donc déterminer le moment à partir duquel l'historique $\mathbb{H}_i(n)$ est suffisant. À ce moment, *tous* les objets actifs doivent avoir pris le point indexé n, c'est-à-dire que l'état global n doit être 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 (en particulier, les ordres relatifs de réception ne sont plus forcément identiques). À partir de cet instant, l'historique peut être *clos*, ce qui signifie qu'il n'est plus nécessaire de mémoriser les émetteurs des requêtes entrantes. Une fois clos, cet historique est enregistré sur un support stable et le point de reprise qui lui est associé (i.e. celui portant le même numéro) devient alors valide pour être utilisé dans une ligne de recouvrement.

En pratique, la clôture des historiques peut-être décenchée de deux manières différentes. D'abord, le processus de reprise est capable d'identifier 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 clore les historiques au plus tôt, et donc de minimiser la taille des historiques. On notera $\mathcal{M}_n^{globalState}$ le message indiquant que l'état global n est formé. Ce message indique au récepteur que les historiques des points de reprise indexés k tel que $k \leq n$ peuvent être clos.

Cependant, cette première méthode ajoute dans l'exécution autant de communications multipoints additionnelles 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. Les historiques sur un objet actif 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.

Cependant, tous les objets actifs ne vont pas clore de manière synchronisée leur historique ; il peut donc survenir des problèmes de cohérence. Prenons le cas de la

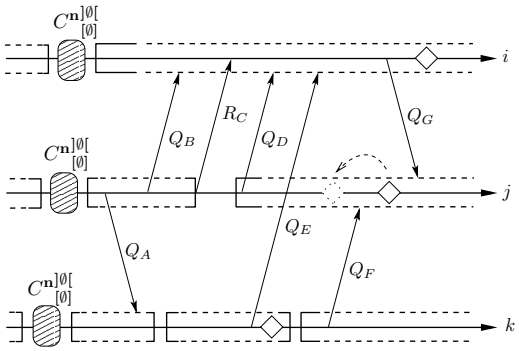

Figure 9. *Les clôtures d'historique (♢) sur j et k ne sont pas cohérentes entre elles*

figure 9. Les objets ont clos leur historique n, mais ces historiques ne sont pas cohérents : la requête Q_F a été reçue avant la clôture de l'historique n sur j, mais a été émise après la fermeture de l'historique n sur k. En cas de reprise depuis le point n, il est possible que la réexécution de k soit différente (si k reçoit des messages dans un ordre différent de la première exécution après la fin de l'historique). Alors Q_F pourrait ne jamais être envoyée en cas de reprise depuis la ligne n : j restera bloqué en attente indéfiniment. Il aurait fallu que l'historique sur j soit clos avant la réception de Q_F, comme indiqué en pointillé sur la figure 9. On notera cependant que la requête Q_G ne pose pas de problème de cohérence entre les clôtures sur i et j, puisqu'elle est en transit par rapport aux clôtures d'historique.

On voit bien ici que le message Q_F est un message orphelin par rapport aux points de clôture d'historique : ces points doivent donc être *cohérents* entre eux, au sens de (Chandy *et al.*, 1985). On va donc utiliser un protocole induit par messages assurant la cohérence faible similaire à (Briatico *et al.*, 1984) pour les coordonner : chaque message porte le numéro du dernier historique clos sur l'émetteur, noté $H_{\overrightarrow{M_{i,j}}}$. à la réception, si le message porte un numéro n supérieur au numéro du dernier historique clos sur le récepteur, celui-ci doit alors fermer l'(es) historique(s) de numéro inférieur ou égal à n *avant* la prise en compte du message. Ce protocole peut être appliqué directement, puisque la clôture d'historique peut être faîte à tout instant. Dans la figure 9, on doit clore l'historique n sur j avant que j ne prenne en compte Q_F (en pointillé).

3.3. *Reprise*

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 clos leur historique. Dans ce cas, chaque point de reprise est associé à son historique. Lorsque l'objet actif i reprend depuis un point de reprise n, il concatène l'historique n à sa queue de requête. La nouvelle queue de requêtes est donc formée de

la queue de requête enregistrée au moment de la prise du point (pouvant contenir des promesses de requête qui représentent les requêtes exclues - cf. § 3.1.1), suivie d'une queue de promesse de requête (l'historique lui même). 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. 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.

Avant la reprise depuis un point n, l'objet actif doit réémettre toutes les requêtes ou réponses qui auraient été enregistrées dans $\mathbb{M}_i^{resend}(n)$. 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 promesses de requête[3] chez un unique récepteur, ce récepteur ne pourrait pas les réordonner à leur arrivée, puisqu'une promesse de requête ne contient que l'identifiant de l'émetteur. Ainsi, l'attente des objets actifs se fait de façon indépendante lors d'une reprise : la reprise du système peut donc être totalement asynchrone.

La procédure de reprise va être déclenché par le processus de reprise après détection d'une panne, en émettant vers tous les objets du système un message de reprise. On notera $\mathcal{M}_{n,k}^{rec}$ le message indiquant au récepteur la k ième reprise du système, à faire à partir du point de reprise n. La reprise du système étant asynchrone, il va être possible à un instant donné que deux objets actifs ne soient pas dans la même occurrence d'exécution : l'un pourrait avoir effectuer sa reprise, tandis que l'autre ne saurait pas 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* noté I_i (Strom *et al.*, 1985; Manivannan *et al.*, 1996), qui va représenter le nombre de reprises effectuées par l'objet actif i depuis le lancement du système. Chaque message de i vers j porte le numéro d'incarnation de son émetteur i, noté $I_{\overrightarrow{M_{i,j}}}$, ainsi que le numéro du point de reprise de la dernière reprise, noté $R_{\overrightarrow{M_{i,j}}}$. De cette façon, le récepteur pourra savoir si l'objet qui lui envoie un message est dans la même occurrence d'exécution que lui. Le numéro d'incarnation $I_{\overrightarrow{M_{i,j}}}$ peut être :

– inférieur au numéro local I_j, et ce message peut être ignoré. Le récepteur j informe l'émetteur i de la dernière reprise.

– supérieur au numéro local I_j, et dans ce cas le récepteur j doit reprendre. L'objet émetteur i est bloqué sur l'émission du message jusqu'à ce que j ait terminé sa reprise.

– égal au numéro local I_j, et le message sera traité normalement.

Le processus de reprise connaît à tout instant le nombre de reprises effectuées par le système depuis la première exécution noté I^{global}.

3. Ces requêtes réémises sont toujours attendues par des promesses de requête chez leurs destinataires : ce sont des requêtes qui ont été émises alors que l'état global en formation n'était pas terminé (par exemple Q_B dans la fi gure 7) et qui ont donc été enregistrées dans les historiques des récepteurs.

3.4. *Récapitulatif*

Le fonctionnement du protocole est donc basé principalement sur l'échange d'informations au moment des communications applicatives entre objets actifs. Nous proposons un tableau synthétique (Tab. 1) des actions à effectuer en fonction des événements qui vont survenir dans la vie d'un objet actif.

4. Spécification algorithmique

Cette section présente de façon formelle le protocole de construction de points de reprise et le protocole de reprise. On notera enfin les actions suivantes sur les messages et les ensembles :

– $M_{i,j} \oplus \mathbb{M}_i^{resend}(n)$ si l'émission du message $M_{i,j}$ doit être ajoutée à la liste $\mathbb{M}_i^{resend}(n)$

– $\mathbb{M}_i^{resend}(k) \oplus C_i^n$ si la liste des messages à réémettre $\mathbb{M}_i^{resend}(k)$ est ajouté au point de reprise C_i^n

– $Q_{i,j}^{awaited} \oplus \mathbb{H}_j(n)$ si une promesse de requête 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.

4.1. *Protocoles*

4.1.1. *Construction des points de reprise*

Nous décrivons de façon formelle dans la figure 10 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** est appelée à la création d'un objet actif.

– La procédure **Tentative de checkpoint** est appelée automatiquement quand $stableState$ est vrai.

4.1.2. *Reprise du système*

Nous décrivons dans la figure 11 le déroulement du protocole de reprise, suite à la panne d'un nœud du système.

ÉVÈNEMENT	ACTIONS
Emission d'un message	• Si l'index n du dernier point pris par le récepteur est plus grand que l'index 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 et informer l'émetteur de la reprise. • 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 du dernier historique clos m → Clore 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	• Clore 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 après un point dont l'index est supérieur ou égal à n par des promesses de requête. • Ouvrir l'historique n.
Reprise depuis un point n	• Récupérer l'image d'état n et l'historique n. • Concaténer l'historique à la queue de requêtes. • Emettre dans l'ordre tous les messages à réémettre enregistrés dans le point n. • Redémarrer l'activité.

Tableau 1. *Synthèse*

1) **Initialisation**
$I_i = 0$, $R_i = 0$, $N_i^{current} = 0$, $N_i^{min} = 0$, $N_i^{max} = 1$
$TTC_i = TTC\_INIT$
Tentative de checkpoint

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

3) **Réception de $M_{i,j}$ (sur j)**
si $I_{\overrightarrow{M_{i,j}}} == I_j$ alors
  $\forall k$ si $\mathbb{H}_i(k)$ non-clos et $k < H_{\overrightarrow{M_{i,j}}}$ alors clôture $\mathbb{H}_i(k)$
  si $M_{i,j}$ est une $Q_{i,j}$ alors $\forall \mathbb{H}_i(k)$ non-clos,$Q_{i,j}^{awaited} \oplus \mathbb{H}_i(k)$
  si $N_{\overrightarrow{M_{i,j}}} > N_j^{current}$ alors $N_j^{max} = max(N_j^{max}, N_{\overrightarrow{M_{i,j}}})$
  retourne $N_j^{current}$
sinon si $I_{\overrightarrow{M_{i,j}}} > I_j$ alors
  bloquer i sur l'émission de $M_{i,j}$
  $I_j = I_{\overrightarrow{M_{i,j}}}$
  Reprise de j sur $R_{\overrightarrow{M_{i,j}}}$ (fi gure 11)
sinon
  envoyer $\mathcal{M}_{R_j,I_j}^{rec}$ à i
  ignorer $M_{i,j}$

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

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

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

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

Figure 10. *Protocole de points de reprise*

1) **Panne de l'objet actif i par le processus de reprise**
placement de i
$I^{global} = I^{global} + 1$
Reprise de i sur n
diffuse $\mathcal{M}^{rec}_{n,Iglobal}$

2) **Réception de $\mathcal{M}^{rec}_{n,k}$ par l'objet actif i**
si $I_i < k$ alors
  $I_i = k$
  Reprise de i sur n
sinon
  ignorer $\mathcal{M}^{rec}_{n,k}$

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

Figure 11. *Protocole de reprise*

5. Expérimentations et validation

Nous avons suivi un plan de validation du protocole en trois étapes. Dans un premier temps, une application de test a été réalisée de manière à évaluer le surcoût moyen induit par le protocole et à vérifier le comportement dans un cas simple. Cette évaluation est présentée dans cette section.

Nous ne détaillons pas ici la preuve de la correction du protocole. Celle-ci est détaillée dans (Baude *et al.*, 2004). Cette preuve est basée sur la notion de *consistance suffisante*, présentée dans (Delbé, 2004). Nous avons montré que la coupe cohérente formée par la clôture des historiques n est forcément une coupe de la réexecution depuis la ligne de recouvrement n.

Les prochaines expérimentations seront réalisées avec des applications plus réalistes et à plus large échelle, en particulier à partir de l'application Jem3D, développée avec ProActive (Baduel *et al.*, 2004).

5.1. *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.

5.1.1. *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 renvoie à chaque réception de point de reprise le numéro du dernier état global formé (cf. § 3.2.3).

Nous n'avons pas implémenté de protocole de détection de panne, ni de protocole de relocalisation des objets après panne, le but étant ici de vérifier le bon comportement du protocole. Les tests du protocole de reprise ont donc été réalisé en relançant les applications *volontairement*, à partir du serveur de points de reprise.

5.1.2. *Application testée*

Les évaluations du surcoût pendant l'exécution ont été réalisées sur une application de recherche du $n$ième nombre premier par le crible d'Eratosthène avec une architecture de type «maître-esclaves».

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

Bien que simple, cette application permet de tester les performances dans le pire des cas. En effet, il a été montré dans (Alvisi *et al.*, 1999) que les protocoles de points de reprise induits par messages déclenchent un grand nombre de points de reprises forcés lorsque les communications sont nombreuses et régulières. Le surcoût mesuré ici est donc une valeur pessimiste.

Les tests ont été réalisés sur un cluster de seize machines bi-Pentium III 1Ghz avec 512MB (SDRAM) connectées par Ethernet 100 Mb/s, avec le protocole de gestion de ressources LSF. Les machines virtuelles utilisées sont des JVM 1.4.0. Pour toutes les expérimentations, un objet actif ProActive est déployé sur chaque machine, que ce soit le maître et les esclaves.

5.2. *Résultats*

Nous présentons d'abord deux graphiques (figure 13) qui donnent la différence entre une exécution de l'application avec ProActive standard, et une autre avec l'ajout du protocole de tolérance aux pannes. Ces tests ont été réalisés avec 4 machines es-

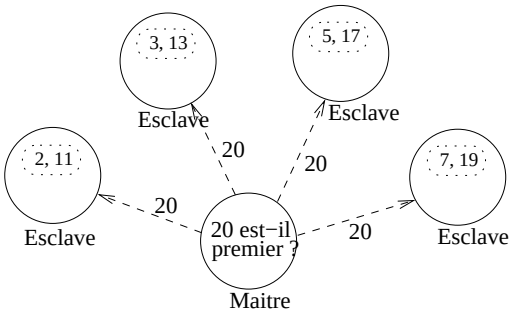

Figure 12. *Application de test : le crible d'Eratosthène*

claves, et un compte à rebours de prise de point de reprise (TTC_INIT) de 100 secondes.

Le premier graphique 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, 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. Il est important de noter que le nombre de points de reprise pris par chaque objet augmente *linéairement* avec le temps total d'exécution. En particulier, ce nombre de points de reprise est égal au temps total d'exécution divisé par le TTC du système.

Cependant, la pente de l'augmentation du surcoût 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 importants 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 nœuds de l'application. On peut voir cette évolution dans la figure 14 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.

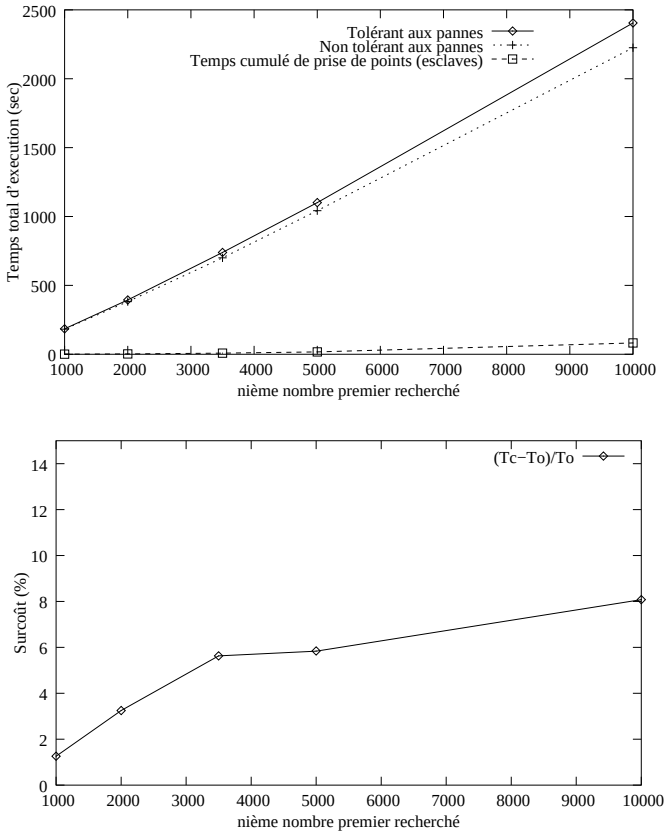

Figure 13. *Surcoût à l'exécution (4 esclaves,* $TTC\_INIT = 100$ *sec)*

6. Conclusion et perspectives

Nous avons proposé un protocole de tolérance aux pannes adapté à un contexte de processus non-préemptifs, avec reprise complètement 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 faite au moment opportun.

Pour contrôler le bon comportement de ce protocole, nous avons, dans un premier temps, réalisé un émulateur d'objets actifs. Les résultats obtenus nous ont permis de constater la formation régulière et rapide de lignes de recouvrement dans divers

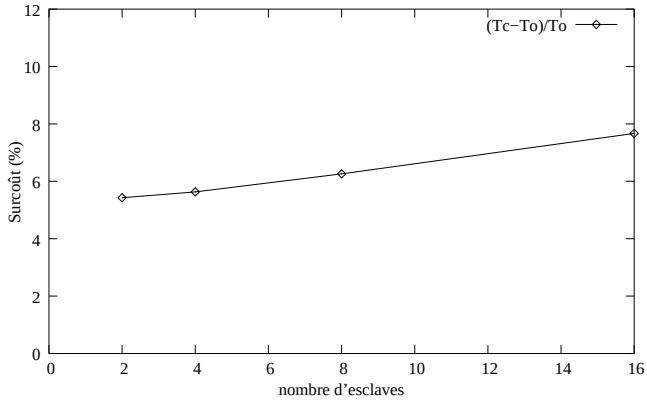

Figure 14. *Surcoût à l'exécution (recherche du 3500ème nombre premier,* $TTC\_INIT = 100$ *sec)*

contextes. Nous avons aussi vérifié que le nombre total de points de reprise pris lors d'une exécution augmente linéairement avec la durée de cette exécution.

Une implémentation prototype a ensuite été réalisée dans la bibliothèque ProActive. Elle nous a permis de constater, à travers différentes évaluations de performance, que l'utilisation additionnelle d'une technique de journalisation de messages (avec l'historique de réception) n'a pas eu d'effet important sur le surcoût induit par le protocole : ce surcoût d'environ 6% reste du même ordre que les meilleurs protocoles de points de reprise, qui ne peuvent pas s'appliquer dans un contexte non préemptif.

Par la suite, 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 complètement 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 nœuds peut atteindre plusieurs milliers. Les techniques de journalisation de messages sont plus adaptées à ce type de systèmes, 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 cet article, puisqu'il utilise déjà une association entre points de reprise et journalisation de messages. Cette union est une base intéressante pour développer un unique protocole paramétré, autorisant à la fois points de reprise, journalisation optimiste et journalisation pessimiste, et cela en

fonction des caractéristiques du système et de l'application au déploiement, voire en cours d'exécution, afin d'obtenir un protocole *adaptatif*.

7. Bibliographie

Alvisi L., Elnozahy E. N., Rao S., Husain S. A., Mel A. D., « An Analysis of Communication Induced Checkpointing », *Symposium on Fault-Tolerant Computing*, p. 242-249, 1999.

Alvisi L., Marzullo K., « Message Logging : Pessimistic, Optimistic, Causal, and Optimal », *Software Engineering*, vol. 24, n° 2, p. 149-159, 1998.

Baduel L., Baude F., Caromel D., Delbé C., Kasmi S. E., Gama N., Lanteri S., « A Parallel Object-Oriented Application for 3D Electromagnetism », *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE Computer Society, Santa Fe, New Mexico, USA, April, 2004.

Baude F., Caromel D., Delbé C., Henrio L., A Fault Tolerance protocol for ASP calculus : Design and Proof, Technical report, INRIA, 2004.

Bouteiller A., Cappello F., Herault T., Krawezik G., Lemarinier P., Magniette F., « A Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging », *ACM/IEEE International Conference on Supercomputing SC 2003*, 2003.

Briatico D., Ciuffoletti A., Simoncini L., « A distributed domino-effect free recovery algorithm », *IEEE International Symposium on Reliability, Distributed Software, and Databases*, 1984.

Caromel D., « Towards a Method of Object-Oriented Concurrent Programming », *CACM, Communications of the ACM, Volume 36, Number 9*, p. 90-102, 1993.

Caromel D., Henrio L., Serpette B., « Asynchronous and Deterministic Objects », *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, ACM Press, 2004.

Caromel D., Huet F., Vayssière J., « A Simple Security-Aware MOP for Java », *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, vol. 2192 of *LNCS*, Kyoto, Japan, p. 118-125, September, 2001.

Caromel D., Klauser W., Vayssière J., « Towards Seamless Computing and Metacomputing in Java », *Concurrency Practice and Experience*, vol. 10, n° 11–13, p. 1043-1061, 1998.

Chandra T. D., Toueg S., « Unreliable failure detectors for reliable distributed systems », *Journal of the ACM*, vol. 43, n° 2, p. 225-267, 1996.

Chandy K. M., Lamport L., « Distributed snapshots : Determining global states of distributed systems », *ACM Transactions on Computer Systems*, p. 63-75, 1985.

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

Cooper E. C., « Replicated distributed programs », *10th ACM Symposium on Operating System Principles*, 1985.

Delbé C., « Causal Ordering of Asynchronous Request Services », *Dependable Systems and Networks - Student Forum*, 2004.

Elnozahy M., Alvisi L., Wang Y., Johnson D., A survey of rollback-recovery protocols in message passing systems, Technical Report n° CMU-CS-99-148, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October, 1999.

Hélary J., Netzer R., Raynal M., « Consistency Issues in Distributed Checkpoints », *IEEE Transactions on Software Engineering, Vol 25*, 1999.

Lai T. H., Yang T. H., « On Distributed Snapshots », *Information Processing Letters*, vol. 25, p. 153-158, 1987.

Manivannan D., Singhal M., « A Low-Overhead Recovery Technique Using Quasi-Synchronous Checkpointing », *Proceedings of the 16th ICDCS*, 1996.

OASIS, *ProActive PDC, installation and User Guide.* 2003, www-sop.inria.fr/oasis/ProActive/.

Schlichting R. D., Schneider F. B., « Fail-stop processors : an approach to designing fault-tolerant computing systems », *ACM Transactions on Computer Systems*, vol. 1, p. 222-238, 1983.

Strom R., Yemini S., « Optimistic Recovery in Distributed Systems », *ACM Transactions on Computer Systems, Vol 3, No 3*, p. 204-226, 1985.

SunMicrosystems, « Java Object Serialization Specification », 1997. java.sun.com.

Article reçu le 18 novembre 2003
Version révisée le 12 décembre 2004

Françoise Baude *est maître de conférences à l'Université de Nice - Sophia Antipolis depuis 1993. Après l'obtention d'un Doctorat de l'Université de Paris-Sud (Décembre 1991), elle a effectué un séjour post-doctoral à l'Université de Warwick, Grande-Bretagne, puis à Queen's University, Kingston, Canada. Ses activités de recherche concernent les langages et environnements de programmation parallèle et répartie.*

Denis Caromel *est titulaire d'un Doctorat de l'Université de Nancy I (Février 1991), et d'une Habilitation à Diriger des Recherches de l'Université de Nice – Sophia Antipolis (Novembre 1996). Il est professeur dans cette même université depuis 1998, et membre de l'Institut Universitaire de France. Il a séjourné deux ans en Californie (1988 à 1990). Il concentre ses recherches sur les modèles formels et les outils pour une programmation répartie simple, sûre et efficace.*

Christian Delbé *est étudiant en thèse à l'Université de Nice - Sophia Antipolis depuis octobre 2003. Ses thèmes de recherche incluent la tolérance aux pannes dans les systèmes distribués, en particulier dans le cadre des grilles de calcul.*

Ludovic Henrio *est titulaire d'un Doctorat de l'Université de Nice Sophia Antipolis (Novembre 2003). Ses principaux sujets de recherche sont la sémantique des langages concurrents, parallèles et distribués, l'analyse statique et les langages et calculs orientés objets.*