RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE AMAR TELIDJI

LAGHOUAT

FACULTÉ DES SCIENCES

DÉPARTEMENT DE MATHÉMATIQUE ET INFORMATIQUE

MÉMOIRE DE MASTER

DOMAINE : MATHÉMATIQUE ET INFORMATIQUE

FILIÉRE : INFORMATIQUE

OPTION : RÉSEAUX, SYSTÉMES ET APPLICATIONS RÉPARTIS (**ReSar**)

Théme:

ETUDE ET SIMULATION D'UN ALGORITHME DE POINT DE REPRISE AVEC PLUSIEURS INITIATEURS

Présenté par:

BOUGRINE SOUMIA

Soutenu devant le jury composé de:

M^{r}	M. YAGOUBI	Université de Laghouat	(Président)
M^{r}	T. BEN DOUMA	Université de Laghouat	(Examinateur)
M^{r}	T. ALLAOUI	Université de Laghouat	(Examinateur)
M^{elle}	Z. ABDELHAFIDI	Université de Laghouat	(Rapporteur)

ANNÉE /...../2011/2012

Je dédie ce mémoire
à
mes parents
qui m'ont toujours soutenu et encouragé
au cours de la réalisation de ce mémoire de fin d'étude.
à
tous les professeurs et les enseignants universitaires
qui m'ont permis, par leurs efforts, d'atteindre un tel niveau de formation.

Remerciements

Je remercie tous ceux qui m'ont apporté leur aide de près ou de loin, m'ont poussé, encouragé, enfin m'ont permis d'arriver à terminer ce mémoire

" Elhamde wa echoukr lil'lah ", je remercie Allah miséricordieux de m'avoir fait rencontrer des êtres bienveillants comme mon encadreur Melle. ABDELHAFIDI Zohra qui m'a stimulé, m'a redonné le goût du travail bien appliqué et fignolé et m'a accompagné jusqu'à la fin .

Je remercie beaucoup tous mes enseignants au cours de tout mon cursus universitaire.

Je remercie tous les membres du jury pour avoir accepté et pris la peine de juger, d'apprécier et d'évaluer mon travail .

Résumé

Trois classes des protocoles de calcul des points de reprise global ont été proposées dans la littérature. La premiere est la classe coordonnée, la deuxième est la classe non-coordonnée et la derniere est la classe CIC (Communication induced checkpointing).

Dans ce mémoire, on s'intéresse à la classe coordonnée avec plusieurs initiateurs. A cet effet on a proposé deux solutions qui calculent le point de reprise global cohérent tout en minimisant le nombre de message utilisé, une évaluation de performances est réalisée en utilisant la simulation avec NS2. Les résultats obtenus montrent que la deuxième solution est meilleure que la premiere solution .

Mots-clés : Système réparti, NS2, point de reprise avec plusieurs initiateurs .

Abstract

Three classes of checkpointing has been proposed in the literature coodinated protocols, uncoordinated protocol and CIC (Communication-Induced Checkpointing) .

In this manscript, we are intersseted on cooordinated class with concurrent initiators. Hence, we have proposed two solutions that calculate a consistent global checkpoint white minimizing number of messages used, en evaluation of performance has been realized using NS2 simulator, the obtained results show that the second solution is better than the first solution .

Key-words : Distributed system, NS2, checkpointing with concurrent initiators.

Table des matières

Liste des figures

Introduction générale

Les progrès remarquables dans les systèmes informatiques pendant les dernières années ont permis une forte évolution des systèmes qui sont caractérisés par une tendance très forte vers la décentralisation, et des problèmes tels que la tolérance aux fautes .

Par ailleurs, l'exigence de tolérance aux fautes est incontournable avec les systèmes répartis, Car la tolérance aux fautes est un besoin induit par la multiplicité des ressources et pour garantir la sûreté de fonctionnement.

L'absence d'état global est l'une des caractéristiques essentielle (et malheureuse) des systèmes répartis. Un site ou processus n'a qu'une connaissance approximative de l'état des autres sites ou un processus puisque seul l'échange de messages permet d'obtenir des informations sur les partenaires distants (logiquement).

Parmi les techniques permettant d'assurer la tolérance aux fautes dans les systèmes répartis est le calcul d'état global (Checkpointing) , cette technique peut être classifié en trois catégories :

- Approche non coordonnée (Uncoordinated Checkpointing).
- Approche coordonnée (Coordinated Checkpointing).
- Approche induite par communication (Communication-Induced Checkpointing).

Dans la classe coordonné, un ou plusieurs processus peut (peuvent) lancer le calcul de point de reprise global cohérente, il enregistré son point de reprise et envoie des requêtes aux autre processus pour enregistrer leurs points de reprise.

Dans ce mémoire, nous avons généralisé l'algorithme NNB pour le cas de plusieurs initiateurs. pour cela nous avons proposé deux solutions . Pour évaluer la performance des solutions, nous avons utilisé le simulateur (NS-2) .

Ce mémoire est composé de trois chapitres et deux annexes :

Dans le 1er **chapitre**, nous avons présenté des généralités sur les systèmes répartis avec leurs inconvénients et leurs avantages, les techniques de checkpointing, et leurs classifications.

Dans le **chapitre** 2, nous avons présenté deux algorithmes avec plusieurs initiateurs, le premier concernant les canaux FiFo et l'autre concernant les canaux Non FiFo. Ainsi que le généralisation d'algorithme NNB pour plusieurs initiateurs.

Dans le **chapitre** 3 regroupe les étapes de simulation et l'intégration des algorithmes simulés dans NS-2. Les résultats de simulation obtenus sont représentés par des courbes .

À la fin, une conclusion résume ce mémoire.

Système réparti et points de reprise 1

Sommaire

Dans ce chapitre on va présenter les systèmes répartis, leurs avantages et leurs inconvénients, leurs problèmes tels que la tolérance aux pannes.

1.1 Système réparti

1.1.1 Qu'est-ce qu'un système réparti ?

Un système réparti est défini comme un ensemble de nœuds de calcul ou processeurs interconnectés par un système de communication et communiquant uniquement par messages [LMP93] .

Définition [**Tanenbaum**] : Un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et cohérent (Les machines sont autonomes, Les utilisateurs ont l'impression d'utiliser un seul système) [Rod07] .

Définition [**Lamport**] : Un système réparti est un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne [Rod07] .

1.1.2 Avantages d'un système réparti

Le système réparti offre un ensemble d'avantages [BL07]

Partage et Mise à disposition : La répartition est avant tout un moyen de mise à disposition et donc de partage de ressources et services. Cette idée de partage et de disponibilité constitue même la sémantique que l'on peut donner au mot répartition.

Répartition géographique : La répartition est un moyen essentiel pour mettre à disposition des usagers les moyens informatiques locaux dont ils ont besoin tout en gardant la possibilité d'accéder aux ressources et services distants de leurs collègues.

Puissance de calcul : La connexion de machines en réseau permet d'obtenir à moindre coût une puissance de calcul importante. Bien entendu, l'exploitation optimale de ces performances potentielles nécessite de paralléliser les algorithmes de calcul et de disposer d'un environnement d'exécution répartie adéquate.

Disponibilité : La disponibilité d'un service développé sur une architecture répartie peut être rendue plus grande que celle d'un service centralisé. La relative indépendance des défaillances qui peuvent survenir dans les nœuds, permet de continuer à offrir un service même dégradé. En particulier, la réplication d'un même service par l'installation de plusieurs serveurs équivalents est une solution classique mais de mise en œuvre délicate notamment si les serveurs sont à état rémanents.

Flexibilité : Une architecture répartie est par nature modulaire. Il est donc plus facile d'ajouter ou d'enlever un nœud connecté au réseau, cette flexibilité se situe aussi au niveau logique. Un nouveau service peut être installé sur un nœud sans nécessité d'une reconfiguration des autres nœuds.

1.1.3 Inconvénients d'un système réparti

Les principaux inconvénients d'un système réparti sont regroupés dans les points suivant [BL07]

Pas d'état global : Dans une architecture répartie, un nœud ne possède pas une connaissance immédiate exacte de l'état d'un autre nœud. En effet, cette connaissance passe par l'échange d'un message qui introduit obligatoirement un délai. Un nœud connaît son état local et n'a qu'une connaissance du passé (certes parfois très proche) des autres. Cette difficulté conduira à la recherche d'algorithmes qui permettent de construire des états globaux qui représenteront un état passé possible de l'application.

Pas d'horloge globale : Chaque nœud possède sa propre horloge pour dater les événements qui lui sont locaux. Par conséquent, si les horloges indépendantes de chaque nœud ne sont pas parfaitement synchronisées, l'ordre des événements répartis dans l'application n'est pas déductible à partir des datations locales. Cette difficulté conduira à définir des datations logiques qui permettent de corriger ce problème.

Ces deux propriétés augmentent fondamentalement la difficulté de compréhension et d'analyse des applications réparties.

Sécurité relative : Une architecture répartie est plus difficile à protéger contre les malveillances qu'une architecture centralisée. Ceci pour deux raisons :

– les points d'accès aux ressources du système global sont multiples et souvent " hors les murs ". Autrement dit, l'utilisateur doit être authentifié.

– le réseau de communication est un point d'accès potentiel pour toute tentative d'intrusion.

1.2 Point de reprise dans les systèmes répartis

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.

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 .

Il existe deux types de point de reprise local ou global .

1.2.1 Point de reprise local

Lorsqu'un processus enregistre son état dans la mémoire stable, on dit qu'il a pris un point de reprise.

- $C_{i,x}$: est x$^{\text{ième}}$ point de reprise local d'un processus P_i .
- *Intervalle* : la séquence des événements produits par un processus. P_i entre deux points de reprise successifs $C_{i,x-1}$ et $C_{i,x}$ est appelée intervalle noté $I_{i,x}$ (voir la figure 1.1) .

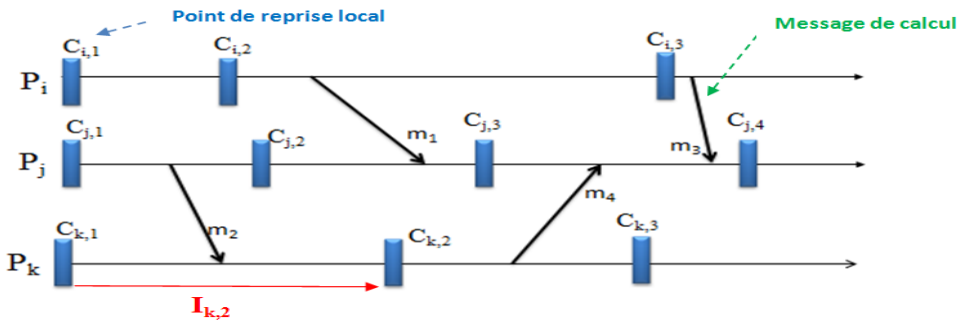

Figure 1.1 – *Point de reprise local.*

1.2.2 Point de reprise global

Un point de reprise global est un ensemble des points de reprises locaux un par processus.

Un message est dit orphelin par rapport ou couple $C_{i,x}$ et $C_{j,y}$, Si son émission est apparue après $C_{i,x}$, et sa réception avant $C_{j,y}$. Un point de reprise global est cohérent s'il n'existe pas de message orphelin par rapport à ce point [CL85] .

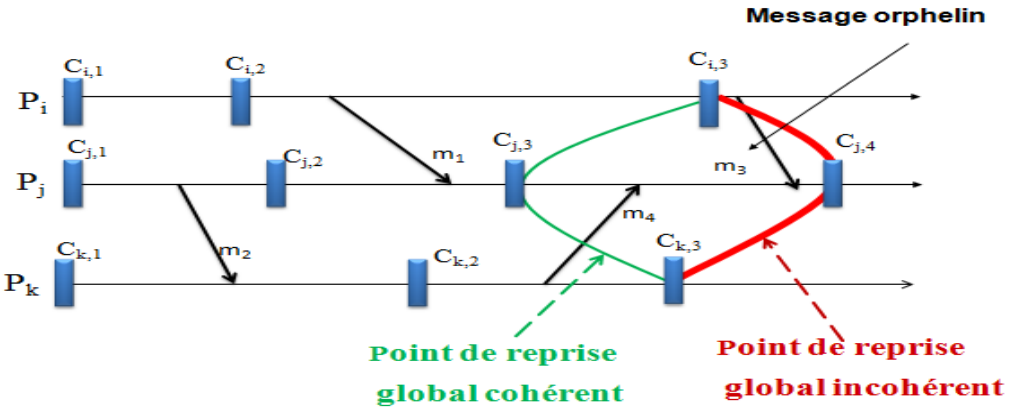

Figure 1.2 – *Point de reprise global.*

Dans la figure 1.2 le point de reprise global { $C_{i,3}$, $C_{j,4}$, $C_{k,3}$ } est incohérent à cause du message orphelin m_3 .

En d'autres termes, si un point de reprise sur le processus P_i a enregistré la réception d'un message provenant du processus P_j, alors il faut que le point de reprise sur P_j appartenant au même point de reprise global ait enregistré l'émission de ce message .

On note qu'un message peut être envoyé mais non reçu (le message m_4 sur la figure 1.2).

1.2.3 Les protocoles de points de reprise

Les protocoles des points de reprise réalisent des calculs régulièrs de l'état des processus. Pour redémarrer, ils utilisent le dernier ensemble des points de reprise formant un état global cohérent.

1.2.3.1 Protocoles non coordonnés

Dans les techniques des points de reprise non coordonnés (ou indépendants), chaque processus prend un point de reprise au moment où il le souhaite. Ceci offre une certaine flexibilité [Mak11].

Si on a une panne, Les processus vont redémarre à partir de leur état le plus récent, rien ne peut garantir que l'ensemble des états individuells de chaque processus forme un point de reprise global cohérent .

Par ailleurs, Ces protocoles souffrent de l'effet Domino .

L'effet domino : le retour arrière d'un site entraîne le retour des autres au delà du dernier point de reprise et cela parfois jusqu'au début du traitement [Kai04]

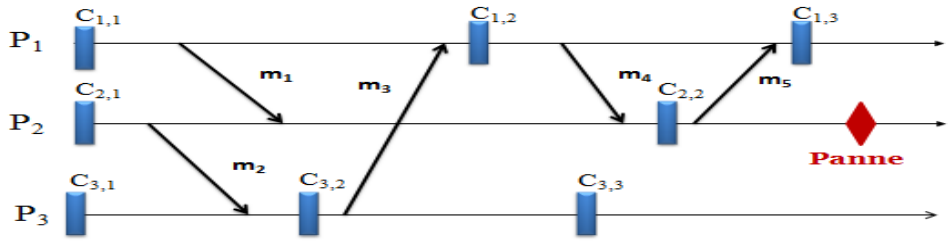

Figure 1.3 – *Effet Domino.*

Dans la figure 1.3 après la panne, P_2 remonte à $C_{2,2}$, mais comme l'émission de $m5$ est perdue, il faut faire remonter P_1 à $C_{1,2}$; puis comme l'émission de m4 est perdue, il faut faire remonter P_2 à $C_{2,1}$; etc. Aucune point de reprise global { $C_{1,a}$, $C_{2,b}$, $C_{3,c}$} n'est cohérent sauf le 1^{ere} point de reprise global { $C_{1,1}$, $C_{2,1}$, $C_{3,1}$}. Pour éviter ce problème, les processus doivent coordonner leurs travaux pour construire un état cohérent du système.

1.2.3.2 Protocoles coordonnés

Cette technique permet aux processus de prendre des points de reprise d'une manière synchrone. Quand un processus reçoit un message pour déterminer un état global cohérent, il arrête son exécution et sauvegarde point de reprise. L'avantage est que l'effet domino ne se produit pas et l'espace de stockage est réduit .

Il existe différentes manières de coordonner les processus. On distingue en particulier [Abd07] :

- Approche coordonnée bloquante.
- Approche coordonnée non bloquante.

1.2.3.2.a Les protocoles coordonnés bloquants

Une manière pour calculer le point de reprise global est de bloquer l'application pendant que les processus prennent les points de reprise la figure 1.4 .

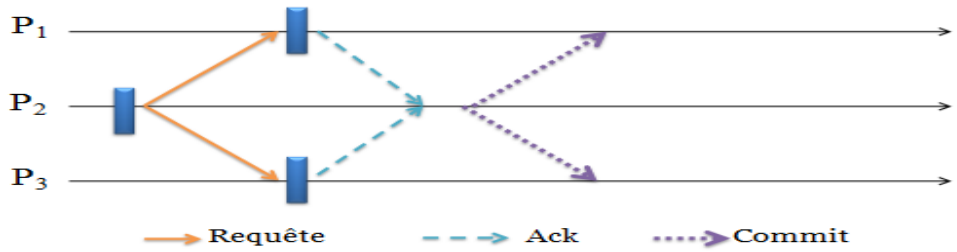

Figure 1.4 – *le protocole coordonné bloquant .*

1. Un initiateur prend un point de reprise et diffuse des requêtes (Requête) vers tous les processus.

2. Un processus qui reçoit un tel message, il se bloque, prend un point de reprise et envoie un acquittement (Ack) vers l'initiateur.

3. Après la réception de tous les acquittements, l'initiateur envoie des messages de validation (Commit) vers les processus.

4. Un processus qui reçoit un message de validation, il supprime l'ancien point de reprise et considère le nouveau point de reprise comme permanent. Ainsi il pourra poursuivre son exécution interrompue.

1.2.3.2.b Les protocoles coordonnés non bloquants

Une alternative à l'approche bloquante est la coordination non bloquante. Dans ce type de coordination les processus ne bloquent pas l'application courante pour prendre des points de reprise.

Cette approche est basée sur la superposition des informations de contrôle aux messages d'application et l'utilisation des messages de contrôle .

Dans la figure 1.5 si n'existe pas message de contrôle donc on a un point de reprise global n'est pas cohérent (car le message m est enregistré reçu dans $C_{j,y}$ et pas enregistré envoyé qui dans $C_{i,x}$).

Dans la figure 1.6 , Le message m est envoyé par le processus P_i au processus P_j après la réception de la requête , si ce message arrive à P_j avant la réception de la requête, il peut conduire à l'incohérence de point de reprise global qui contient $C_{i,x}$ et $C_{j,y}$ due à la présence de m qui vérifie la relation de dépendance entre $C_{i,x}$ et $C_{j,y}$.

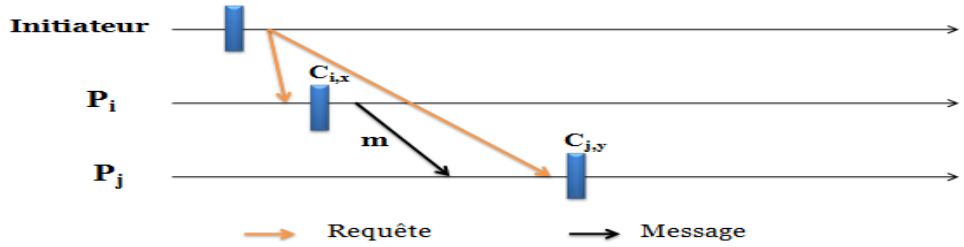

Figure 1.5 – *incohérence entre points de reprise .*

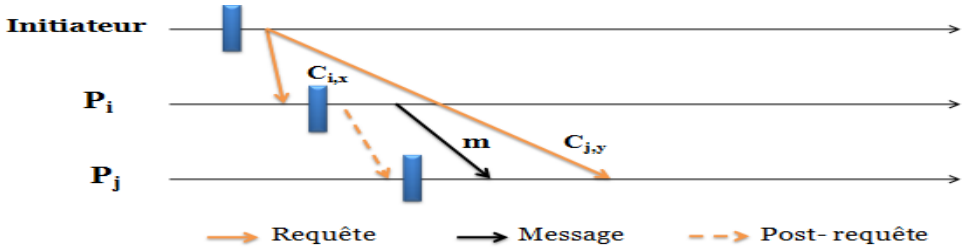

Figure 1.6 – *Cohérence entre points de reprise canaux FIFO .*

Une stratégie adoptée par Chandy et Lamport [CL85] , pour résoudre le problème précédent est faite de la manière suivante :

- Un processus P_i est forcé d'envoyer une post-requête avant d'envoyer un message m, tout processus P_j prend un point de reprise dès l'arrivée de la première requête.

- Dans ce cas un processus P_j peut consommer le message m sans risque de l'incohérence du point de reprise global.

Une adaptation de la solution de Chandy et Lamport pour les canaux non FIFO est fondée sur le principe de superposition de post-requête sur les messages d'application.

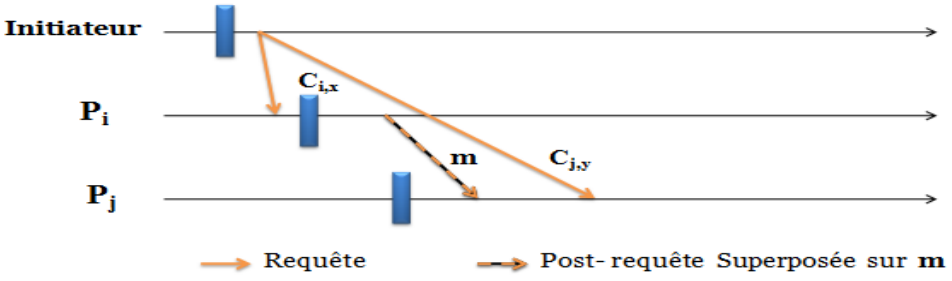

Figure 1.7 – *Cohérence entre points de reprise canaux non FIFO .*

Dans la figure 1.7, la post-requête (du processus P_i vers le processus P_j) est superposée au message m, le processus P_j prend un point de reprise avant la consommation d'un tel message. Il est possible de remplacer les post-requêtes par des nombres de séquence, un processus P_j prend un point de reprise lorsque son nombre de séquence est inférieur à celui superposé au message m.

1.2.3.2.c Protocoles basés sur la coordination minimale

Les protocoles coordonnés [Abd07] requièrent que tous les processus participent dans la construction du point de reprise global ce qui implique un surcoût élevé imposé sur l'exécution.

Il est souhaitable que le nombre de processus participants soit minime, ceci peut être réalisé quand seuls les processus qui ont communiqué avec l'initiateur (d'une manière directe ou indirecte) depuis son dernier point de reprise ont besoin de coordonner pour prendre des points de reprise.

Un schéma à deux phases présenté par Koo et Tueg [KT87] réalise la coordination minimale.

- Durant la première phase l'initiateur identifie tous les processus qui ont communiqué avec lui depuis son dernier point de reprise et leur envoie des requêtes. Dès l'arrivée de la requête, tous les processus doivent se comporter comme l'initiateur jusqu'à ce qu'il n'y ait plus de processus à identifier.

- Durant la deuxième phase, les processus identifiés prennent des points de reprise, et de ce fait le point de reprise global résultant est cohérent.

- Dans ce protocole l'application est bloquée jusqu'à la terminaison de la deuxième phase.

- Une amélioration de ce protocole présentée par Cao et Singhal [CS03] minimise potentiellement le temps de blocage et de minimiser le nombre de points de reprise au cours de points de reprise.

1.2.3.2.d Protocoles basés sur plusieurs initiateurs

Une autre manière qui permet d'améliorer le surcoût des protocoles coordonnés est de permettre à plusieurs processus d'initier le calcul de point de reprise global en concurrence [Abd07] .

Pour que ces initiateurs maintiennent les mêmes informations concernant le point de reprise global du système, il existe deux approches :

1. L'approche de l'intersection : un processus qui a déjà pris un point de reprise pour un initiateur, ignore les requêtes des autres. Une fois que cette étape est terminée, les initiateurs partagent entre eux les informations pour avoir une idée complète sur le point de reprise global (voir la figure 1.8).

2. L'approche de l'union : un processus peut prendre plusieurs points de reprise en réponse aux requêtes des différents initiateurs, dans ce cas l'ensemble des derniers points de reprise dans chaque processus forment un point de reprise global qui est

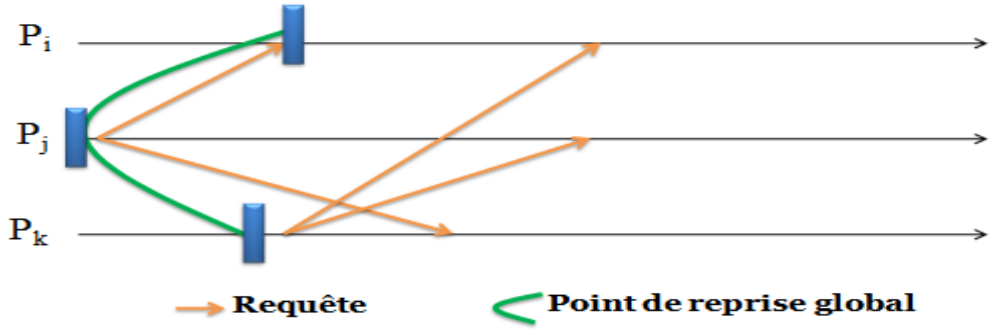

Figure 1.8 – *L'approche de l'intersection.*

plus récent par rapport à la première approche (voir la figure 1.9) .

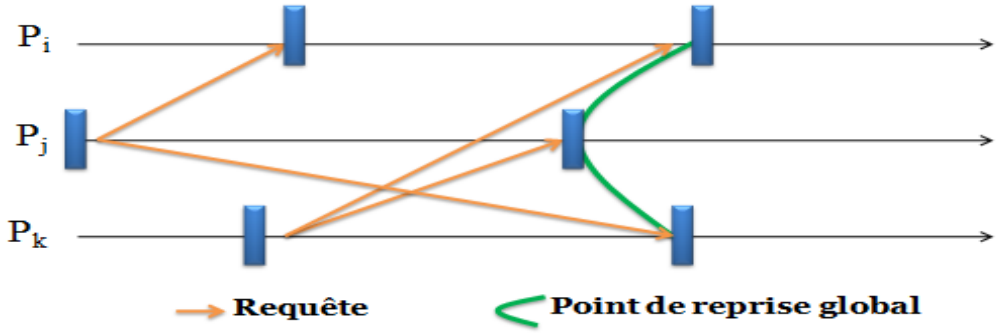

Figure 1.9 – *L'approche de l'union*

1.2.3.3 Protocoles de type CIC

Cette classe de protocoles Communication-Induced Checkpointing est un compromis entre les protocoles coordonnés et les protocoles incoordonnés. Chaque processus prend régulièrement des points de reprise de manière indépendante. La synchronisation se fait via l'utilisation des messages de l'application (superposition des informations supplémentaires) [Abd07] .

Cependant, en fonction des messages reçus, envoyés et des informations qui sont superposées sur ces messages, le processus devra peut être prendre un point de reprise additionnel. Ces protocoles assurent que le point pris fasse partie d'un point de reprise global cohérent, et donc qu'il existe une ligne de recouvrement toujours assez récente.

1.3 Conclusion

Dans ce chapitre, nous avons défini les systèmes répartis tout en donnant leur avantages et leur inconvénients . Nous avons présenté deux types de point de reprise dans les

systèmes répartis (local , global).

Trois approches sont mises en œuvre pour assurer la cohérence : non coordonnés , coordonnés et induite par communication .

2 Algorithmes de point de reprise avec plusieurs initiateurs

Sommaire

2.1 Introduction

La Classe Coordonnée est une approche intéressante dans la tolérance aux pannes pour les applications distribuées, car elle évite l'effet domino et minimise les besoins de mémoire stable.

Dans cette approche, l'état de chaque processus dans le système est sauvegardé dans la mémoire stable, ce qui est appelé un point de reprise (Checkpoint) du processus, les processus doivent synchroniser leurs activités. En d'autres termes, quand un processus prend un point de reprise, il demande (en envoyant des requêtes) à tous les processus partenaires de prendre des points de reprise.

Nous décrivons par la suite les algorithmes avec plusieurs initiateurs .

2.2 Algorithme avec plusieurs initiateurs

Il existe plusieurs algorithmes nous allons présentés deux, la différence entre eux est le type de canaux FiFo ou Non-Fifo .

2.2.1 Algorithme de R.Prakash et M.Singhal pour les canaux FiFo

Cet algorithme est proposé par R.Prakash et M.Singhal en 1994 [PS94] , cet algorithme calcule le point de reprise globale en deux phases :

1. Chaque processus dans le système prend un (ou plusieurs) point (s) de reprise local tentatif.
2. Les initiateurs changent les informations et chaque processus qui a pris un point de reprise tentatif (ou plusieurs), il rend un seul permanent.

2.2.1.1 Variable et structure de donnée

- node-color : La couleur de processus initialisé à blanc .
- init-color : La couleur de l'initiateur initialisé à blanc .
- V : Table de N éléments . $V_i[j]$ contient le temps où le processus P_j a prend un point de reprise tentatif de l'initiateur P_i.
- concurrent : Liste des initiateurs concurrent, initialisé à Null .
- weight : Réel. Utilisé pour détecter la terminaison de l'algorithme initialisé à 0 .

2.2.1.2 Les messages utilisés

Cet algorithme utilise cinq types de messages :

- **Request** : Demande de prendre un point de reprise, il contient les variables node-color et init-color de l'initiateur .
- **Response** : Réponse envoyée par le processus ayant reçu un message Request, il contient temps où il prend point de reprise tentative et init-color .
- **find-maximal** : Ce message est utilisé pour commencer le calcul de point de reprise global maximal, il contient le vecteur V .
- **Commit** : L'initiateur envoie commit à chaque processus pour rendre les points de reprise tentative en point de reprise permanants . contient le vecteur V .
- **Message** : Message d'application.

2.2.1.3 Description de l'algorithme

2.2.1.3.a L'envoi d'un message d'application

Chaque message envoyé contient les variables node-color et temps d'envoi de ce message .

2.2.1.3.b Lancement du calcul du point de reprise global

Lorsque un processus P_i lance le calcul de point de reprise global, il prend un point de reprise tentatif et la couleur de l'initiateur (init-color)et change sa couleur (color-node) à noir et envoie des requêtes (Request) à ces voisins.

Remarque : Les requêtes et tous les messages envoyés par P_i seront de couleur noire. Si un processus a une couleur (node-color) blanche et reçoit un message noir donc il change la couleur du processus à noir.

2.2.1.3.c Réception d'une Requête

Quand un processus P_j reçoit une requête depuis P_i , si P_j a reçu cette requête pour la première fois, il prend un point de reprise tentatif .

- Il change sa couleur à color-node de processus P_i , il met à jour color-init et envoie des requêtes (Request) à ces voisins.
- Il envoie à l'initiateur le message "responce" qui contient le temps où il a pris le point de reprise tentatif .

S'il a déjà reçu une "request" depuis cet initiateur alors envoyé le message "responce" à l'initiateur (temps égal Null) .

Si P_j est un initiateur alors

1. Il envoie le message "responce" contient propre init-color .

2. Il ajoute P_i à la liste des initiateurs concurrents (concurrent) s'ils sont la même couleur d'initiateur (init-color de P_i et init-color de P_j égal blanc) .

2.2.1.3.d Réception de Message " Responce "

Lorsque P_j a reçu un message responce depuis P_i :

- Il met a jour son vecteur V. Le $V_i[j] :=$ reçu-temps, si reçu-temps n'est pas $Null$.

- Il compare la couleur reçue de l'initiateur avec son init-color si c'est la même couleur, il ajoute P_i à la liste concurrent de P_j .

- Si P_j a reçu toutes les reponses alors il envoie le message "find-maximal" qui contient son vecteur V aux processus dans la liste concurrente .

2.2.1.3.e Réception de Message " Find Maximal "

L'initiateur P_i a reçu tous les vecteurs V. Il envoie des messages "commit" aux processus P_j dont $V_i[j]$ est le maximum entre $V_k[j]$, $\forall k \neq i$.

2.2.1.3.f Réception de " Commit "

Lorsque P_i a reçu le message commit depuis P_j , il rend son point de reprise tentatif permanent (Chaque processus reçoit un seul commit) .

2.2.2 L'algorithme D. Goswami et S. Majumder pour les cannaux Non FiFo

Cet algorithme est proposé par D. Goswami et S. Majumder en 2011 [GM11] , Cet algorithme est divisé en deux phases .

2.2.2.1 Variable et structure de donnée

Chaque processus P_i à les variables locaux suivants :

- $SendVector_i$: Est un vecteur qui contient les compteurs des messages envoyés par le processus P_i .
- $ReceiveCount_i$: Variable comptant le nombre de messages reçu .
- $CurrSnapId_i$: 'Currant Snapshot Identifiant ' Dernière valeur de csn [1] de P_i .
- $concurrent_i$: Liste des identifiants des processus en concurrence avec le processus P_i .
- $state_i$: Il indique l'état de point de reprise [tentatif ou permanant].
- $firstInitiator_i$: L'identifiant de la première initiateur .

Remarque : Lorsque un processus P_i envoyé un message il ajoute $CurrSnapId_i$ et $state_i$ aux message.

2.2.2.2 Les messages utilisés

- **Request** : Demande de prendre un point de reprise.
- **Response** : Réponse envoyée par le processus ayant reçu un message Request Message .
- **Dummy-Response** : Message est envoyé par un processus à un initiateur lorsqu'il reçoit un message Request depuis un autre initiateur .
- **Compile** : Un initiateur envoie ce message au leader. Ce message contient les vecteurs $SendVector_i$ et $ReceiveCount_i$.
- **In-Transit-Count** : Ce message contient le nombre de tous les messages en transit ce message est envoyé par le leader à chaque initiateur .
- **Final-Response** : Envoyé par chaque processus à tous les initiateurs à la terminison de procedure de calcul de point de reprise global .

1. the Checkpoint Sequence Number

2.2.2.3 Description de l'algorithme

2.2.2.3.a Premiere phase

- Lancement du calcul de point de reprise global

Lorsque un processus P_i a lancé le calcul de point de reprise global :

Il prend un point de reprise tentatif et incrémente $CurrSnapid_i$, $firstInitiator_i = P_i$ et ajoute P_i à la liste $concurrent_i$.

Il envoie des requêtes (**Request**) aux processus voisins. Pour chaque Request P_i envoyé $concurrent_i$ et $state_i$.

- La réception d'un " Request "

Lorsque P_i a reçu un Request depuis P_j :

Si P_i a reçu une requête pour la 1^{ere} fois alors il enregiste son point de reprise et $firstInitiator_i$ contient l'identité P_j , il envoie une reponse à l'initiateur contient $SendVector_i$ et $ReceiveCount_i$.

Si P_i a déjà pris une point de reprise tentatif, il envoie le message Dummy-Responce contient $firstInitiator_i$ et ajoute P_j à la list de processus concurrent de P_i .

- La réception d'une Réponse

Lorsque P_i a reçu un Réponse (Response ou Dummy-Responce) depuis P_j ,il incrément $ResponseCount_i$ et l'initiateur P_i copie $SendVector_j$ reçu dans j^{eme} ligne de matrice $SendMatrix_i$ et copie $ReceveCount_j$ dans j^{eme} casse de vector $ReceiveVector_i$.

A la réception DummyReponce , P_i ajoute $firstInitiator_j$ à la liste de processus $concurrent_i$.

- La réception de Message d'application

Lors de la réception de message d'application par le processus P_i depuis P_j on a trois cas :

$1^{ere} cas$: si $CurrSnapId_i = CurrSnapId_j$ alors P_i consomme le message .

$2^{eme} cas$: Si $CurrSnapId_j > CurrSnapId_i$,P_i est encours de procedure de calcul de point de reprise et le considère comme étant une message en transit .

$3^{eme} cas$: Si $CurrSnapId_j < CurrSnapId_i$, P_i enregistre ce message dans la file d'attente jusqu'a $CurrSnapId_i$ égal $CurrSnapId_j$.

2.2.2.3.b Deuxième phase

Si un initiateur a reçu $(n-1)$ réponse [n : nombre de processus dans le système]. Il choisit d'abord un leader [qu'est l'identite minimal] depuis la liste de processus concurrent [tous les initiateurs à la même liste concurrent donc le leader est unique dans le système], l'initiateur P_i envoie message compile à leader contient $SendMatrix_i$ et $ReceveVector_i$.

- La réception d'un message Compile

Si le leader a reçu m messages Compile [m :nombre d'initiateur] alors

- Il fusionne les données dans son propre $SendMatrix$ et $ReceiveVector$.
- Il Calcule le nombre de message en transit pour chaque processus et ajoute cette valeur dans le vecteur $TransitVector$.
- Il diffuse le message In-Transit-Count contenant $TransitVector$ à tous les processus .

- La réception d'un message In-Transit-Count

Lorsque un processus P_i a reçu un message In-Transit-Count depuis le leader P_j

- Si le processus a reçu tous messages en transit enregistres, il rend son point de reprise tentatif permanant et diffuse Final-Response à tous les initiateurs.
- Les initiateurs detectent la fin de l'algorithme s'ont reçu $(n-1)$ Final-Response .

2.3 Généralisation de l'algorithme NNB avec plusieurs initiateurs

Le Nouveau algorithme Non Bloquant proposé par Z.Abdelhafidi et M.Djoudi et M.B.Yagoubi en 2012 [ADY12], C'est un protocole en deux phases et enregistre trois types de points de reprise sur le stockage stable (tentatif , Mutable , Permanent) .

2.3.1 Algorithme NNB avec un seul initiateur

2.3.1.1 Les messages utilisés

Cet algorithme, utilise quatre types de messages :

- **Request** : Demande de prendre un point de reprise.
- **Reply** : Réponse envoyée par le processus ayant reçu un message Request.

– **Commit** : Envoyé par l'initiateur à chaque processus pour rendre les points de reprise tentatif en point de reprise permanant.

– **Message** : Message d'application.

2.3.1.2 Les deux phases de l'algorithme

1^{ere} **phase** : L'initiateur prend un point de reprise tentatif et force tous les processus ayant envoyé des messages d'application (avant de prendre ce point de reprise) de prendre un point de reprise tentatif et propage cette demande. Chaque processus informe l'initiateur qu'il a reçu la demande en envoyant un message Replay .

2^{eme} **phase** : Si l'initiateur a reçu les messages Replay de tous les processus qu'a envoyé dans 1^{ere} phase, donc il envoie un message Commit pour rendre les points de reprise tentatif en point de reprise permanant .

2.3.1.3 Les variables locales

- DV_i : Table de N élément. Le Vecteur de Dépendance $DV_i[j] = 1$ si P_i a une relation de dependance avec P_j .

- csn_i : Table de N élément. Checkpoint Sequence Number, $csn_i[j]$ contient l'information concernant le numéro de séquence de point de reprise de P_j .

- trigger : Enregistrement (pid , inum) { pid : l'identificateur de l'initiateur est un entier ; inum :le csn de l'initiateur est un entier } .

- $sent_i$: Booléen est égal à faux. Si le processus n'a envoyé aucun message. Il est égal à vrai si P_i a envoyé un message d'application .

- $state_i$: Booléen. égal à vrai, si P_i en cours de la phase de calcul de point de reprise.

- $old - csn_i$: Entier. Dernier numéro de séquence.

- nb-req : Entier. Compteur de message Request.

- nb-replay : Entier. Compteur de message Replay.

– CP_i : Enregistrement { Mutable , DV[N] , Trigger , sent } utilisé pour sauvegarder les informations concernant le point de reprise mutable .

  – Mutable : Point de reprise .

  – DV[N] : Le Vecteur de Dépendance de P_i avant de prendre Checkpoint Mutable.

  – Trigger : Les informations de l'initiateur .

– sent : La valeur de variable senti de P_i avant de prendre Checkpoint Mutable .

– S : Table de N élément, S_i[j] = 1 si P_i a envoyé un Request à P_j .

2.3.1.4 Description de protocole NNB

2.3.1.4.a L'envoi d 'un message d'application

Quand un processus P_i a envoyé un message d'application, il y a deux cas (figure 2.1) :

– S'il y a un calcul de point de reprise en cours ($state_i = 1$) alors P_i superpose sur le message.les informations sur l'initiateur (trigger), son csn (csn_i) et son vecteur de dépendance DV_i sur le message .

– Sinon P_i envoie ce message avec trigger est NULL, son csn csn_i et son vecteur de dépendance DV_i .

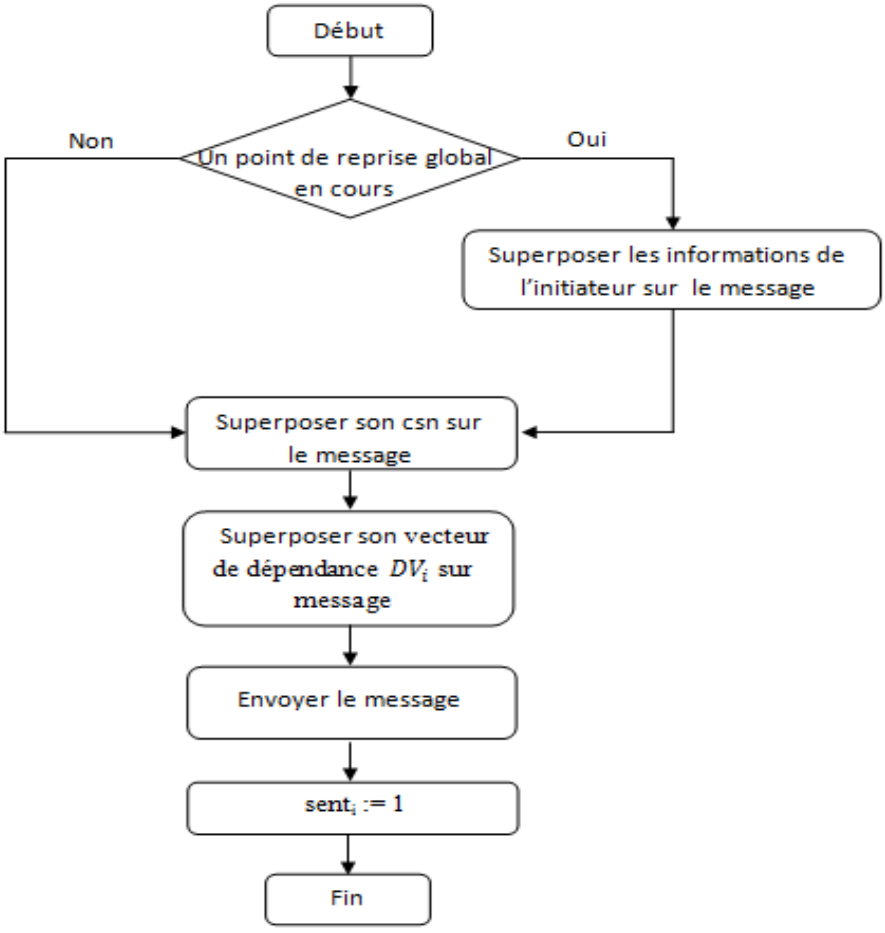

Figure 2.1 – *le diagramme des étapes d'envoi des messages d'application.*

2.3.1.4.b Lancement du calcul du point de reprise global

Quand un processus P_i souhaite de lancer un calcul de point de reprise global, il prend un point de reprise tentatif et envoie des requêtes (Request) aux processus dont il dépend (figure 2.2) .

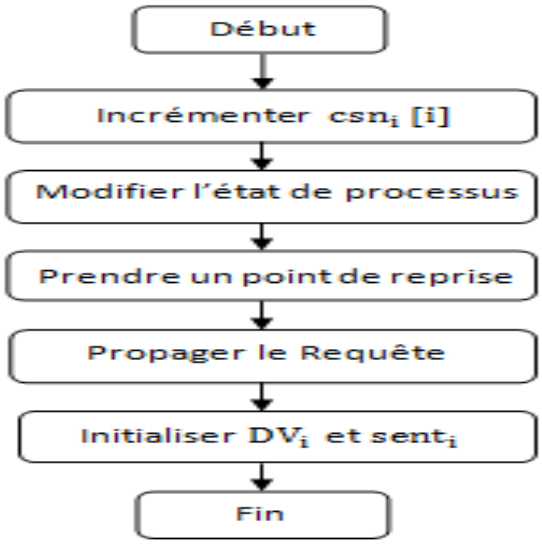

Figure 2.2 – *le diagramme des étapes de lancement le calcul de point de reprise global .*

2.3.1.4.c Réception d'une requête

Quand un processus P_i a reçu une requête depuis P_j, il compare le csn reçu dans la requête avec son ancien csn . Pour voir s'il a besoin de propager la requête (figure 2.3) .

– Si le cas (csn reçu dans la requête $<$ son ancien csn), il envoie un message Reply à l'initiateur.

– Sinon, on a deux cas :

1. csn reçu dans la requête $=$ son ancien csn

   - P_i a reçu un message d'application qui porte des informations sur le calcul de point de reprise, alors P_i a déjà pris un point de reprise mutable. il le rend tentatif, propage la requête et envoie une réponse (message Replay qui contient le $CP_i.DV$ et le vecteur $CP_i.csn$) à l'initiateur .

   - P_i a déjà reçu une requête , il envoie une réponse (message Replay qui contient son vecteur de dépendance DV_i et le vecteur csn_i) à l'initiateur.

2. csn reçu dans la requête $>$ son ancien csn , P_i n'a reçu aucune requête , alors il prend un point de reprise tentatif et propage la requête ensuite il envoie une

réponse (message Replay qui contient le DV_i et le vecteur csn_i) à l'initiateur .

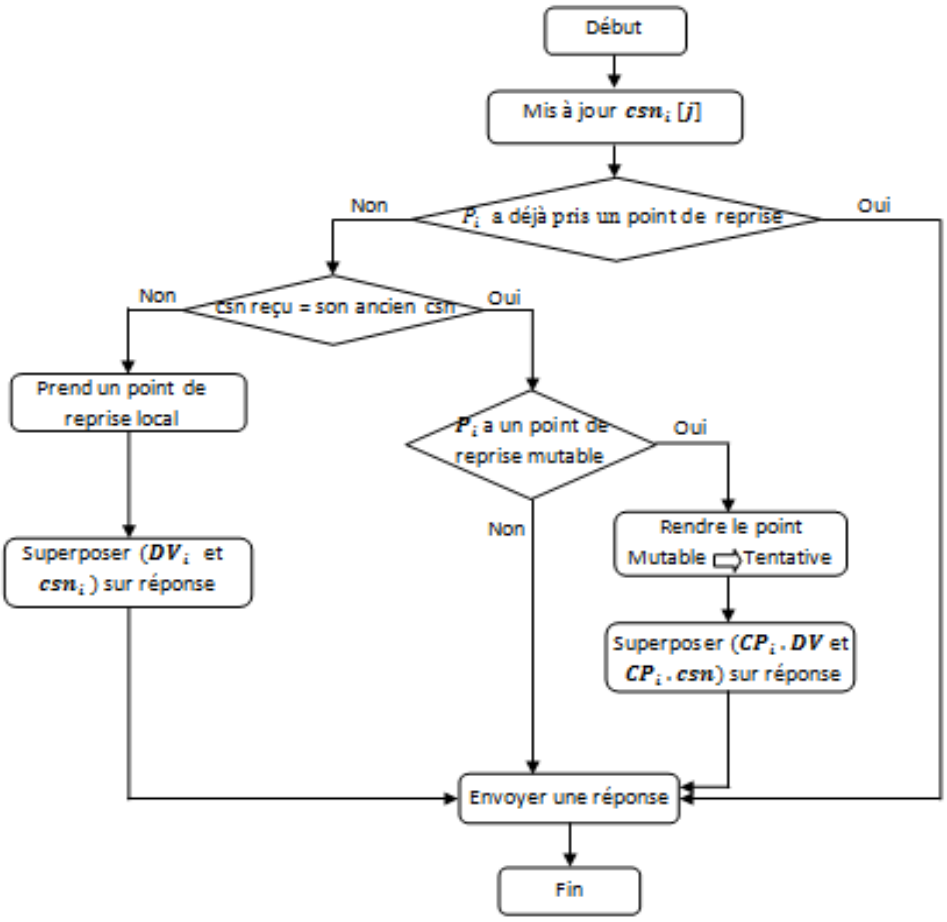

Figure 2.3 – *Le diagramme réception d'une requête.*

2.3.1.4.d La réception des messages d'application

Lorsque P_i a reçu un message d'application depuis P_j, P_i compare le csn du message reçu avec son csn_i[j] (la figure 2.4) :

1^{ere} **cas** : Si ReçuCsn $< csn_i$[j] alors il consomme le message .

2^{ime} **cas** : Sinon, cela implique que P_j a pris un point de reprise avant d'envoyer m. Dans ce cas, P_i teste l'information portée par le message concernant l'initiateur .

– Sinon, il met à jour csn_i[j] à ReçuCsn et test les conditions suivant :

- **Condition** 1 **:** P_j a envoyé le message m après le lancement de processus de calcul d'état global (msg-trigger $\neq$ Null) .
- **Condition** 2 **:** P_i a envoyé un message depuis le dernier point de reprise $sent_i = 1$.
- **Condition** 3 **:** P_i n'a pas pris un point de reprise pour cet initiateur (msg-triggre $\neq ownntrigger$).

Si toutes ces conditions sont satisfaites, P_i prend un point de reprise mutable .

Si seule la 1^{ere} conditions est satisfaite, P_i incrémente $csn_i[i]$, change son état et consomme le message.

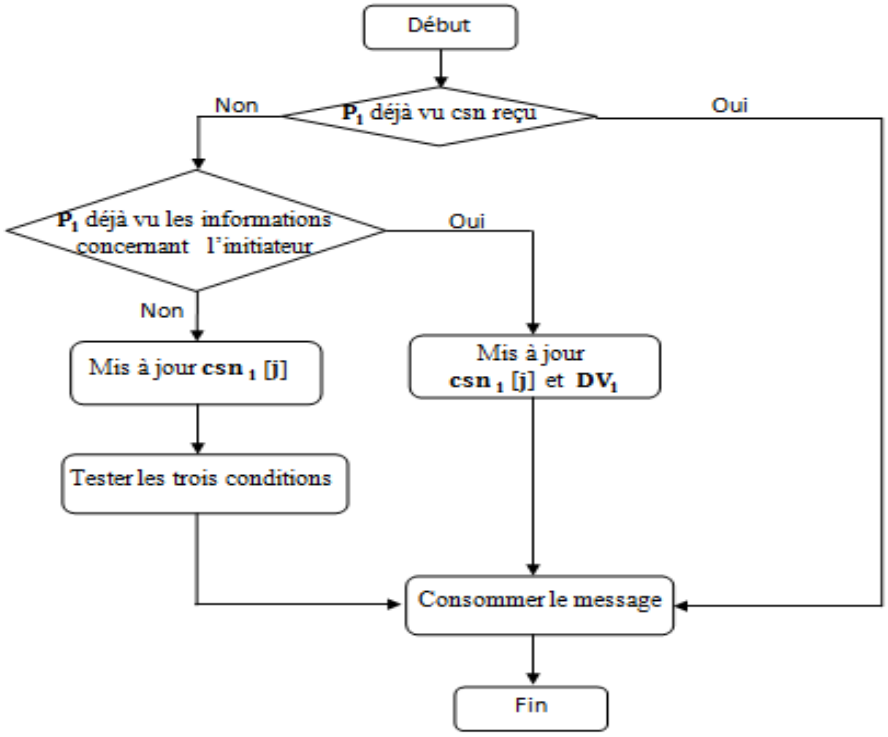

Figure 2.4 – *Le diagramme de réception d'un message.*

2.3.1.4.e La réception de messages Reply

Il existe deux cas (la figure 2.5) :

- 1^{ere} **cas** : Si l'initiateur a reçu des messages "Reply" avec des nouvelles informations. Il envoie des messages "request" à ces nouveaux processus .

- 2^{eme} **cas** : sinon, il declanche la fin de cette phase et envoie des messages Commit .

2.3.1.4.f La réception d'un Commit

Lorsque P_i a reçu un message commit depuis P_j , si processus P_i a pris un point de reprise tentatif le rend permanant, et les point de reprise mutable seront éliminés (la figure 2.5) .

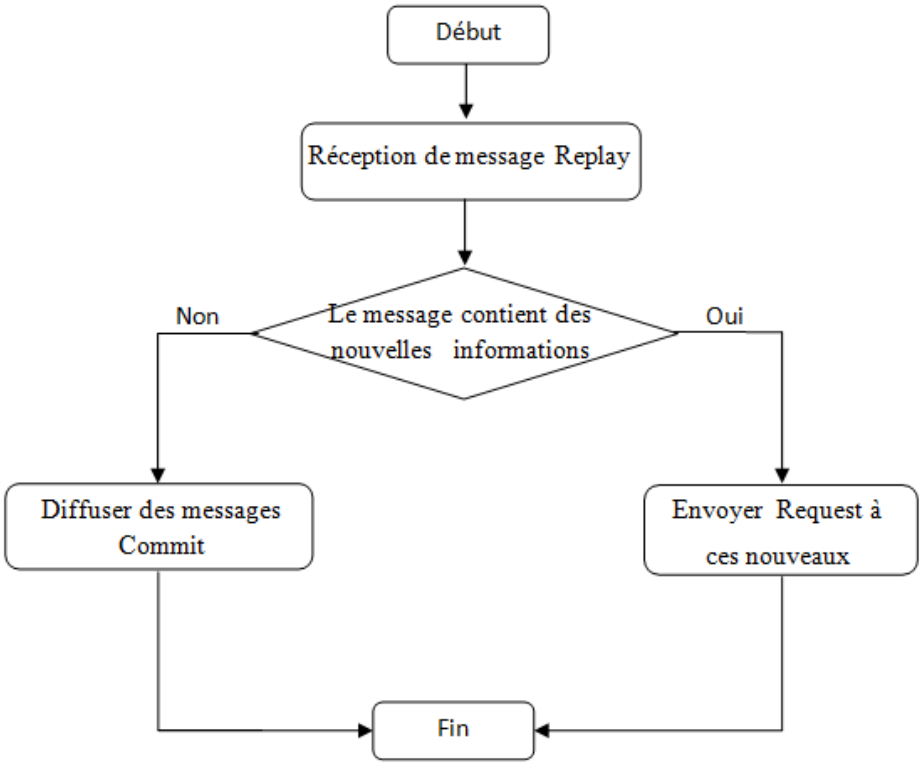

Figure 2.5 – *Le diagramme de réception d'un message Replay .*

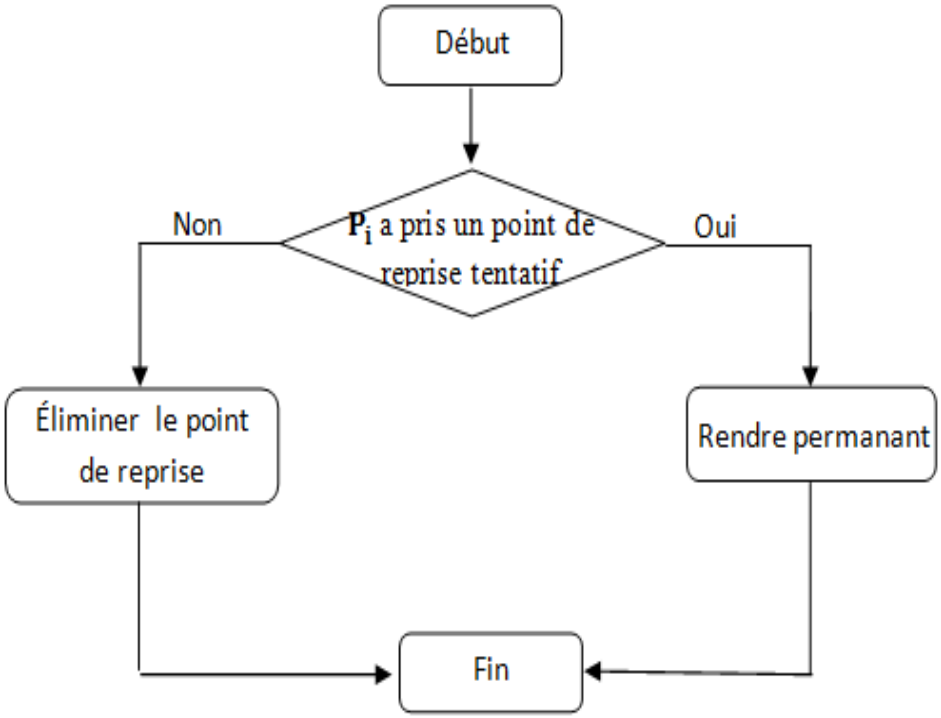

Figure 2.6 – *Le diagramme de réception d'un message Commit .*

2.3.1.5 Exemple d'algorithme NNB

P_2 lance la procédure de checkpointing alors il envoie des requêtes à P_0 , P_1 , P_3 et P_4 . à la réception de Reply depuis le processus P_0 , il détecte qu'il y a un nouveau processus P_5 alors il lui envoie un requête. Par la suite l'initiateur n'a reçu aucune nouvelle information alors il détecte la fin de 1^{ere} phase il passe à la 2^{eme} phase alors il envoie des messages commit aux processus (figure 2.7) .

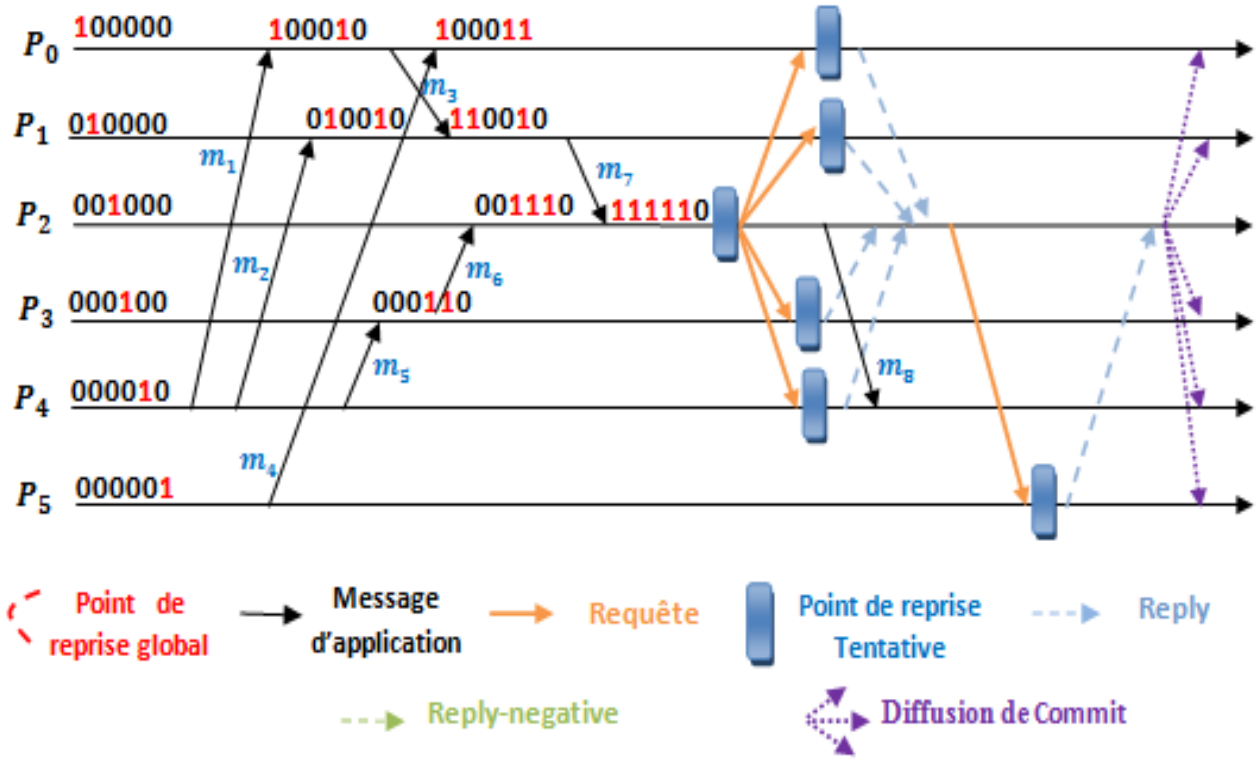

Figure 2.7 – *Exemple d'algorithme NNB .*

2.3.2 Algorithme NNB avec plusieurs initiateurs

Pour gérer le cas de plusieurs initiateurs il y a deux approches

1. Un processus ayant reçu une requête il prend un point de reprise si cette requête est la première sinon il envoie une réponse négative .

2. A chaque réception de requête un processus prend un point de reprise et envoie une réponse à l'initiateur

Puisque la 2^{eme} approche est couteux on terme de point de reprise, alors on adopte la 1^{ere} approche.

2.3.3 Problématique

S'il y a plusieurs initiateurs concurrents on a plusieurs cas :

2.3.3.1 Premiere Cas

Si les ensembles de dépendance disjointe alors le point de reprise local reste cohérent (la figure 2.8).

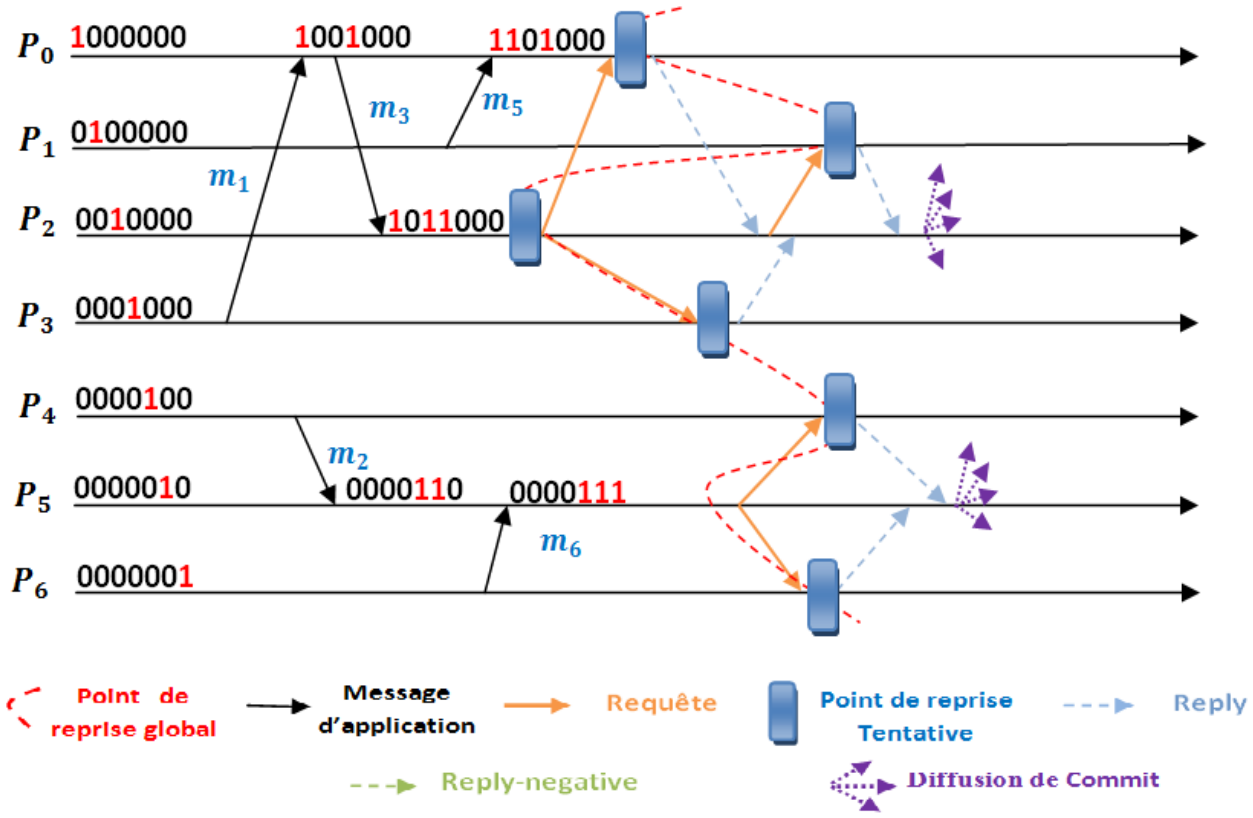

Figure 2.8 – *Exemple de premier cas.*

2.3.3.2 Deuxième Cas

Si un processus a reçu deux requêtes différents, et il n'y a pas des événements de communication entre ces deux requêtes, il envoi une réponse négative pour la deuxième requête.

le processus P_3 a reçu deux requête depuis P_2 et P_4 alors il répond négativement à la requête de P_4 car il n'y a pas un évènement de communication entre les deux requêtes (figure 2.9) .

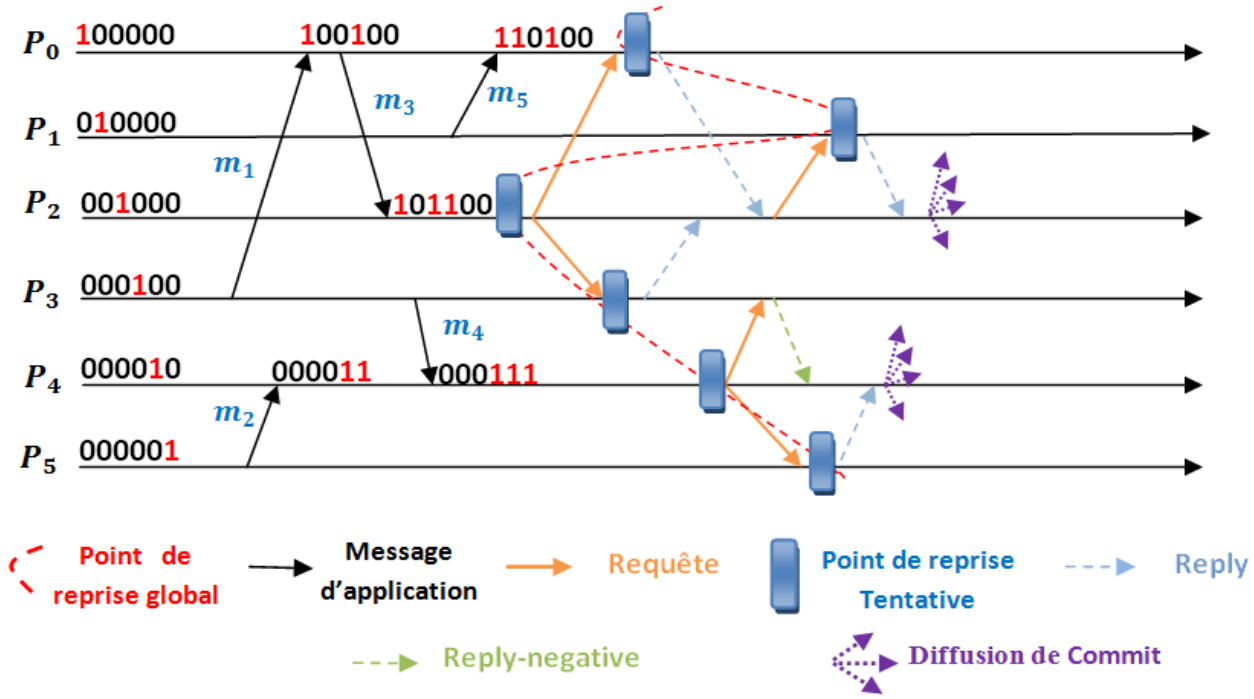

Figure 2.9 – *Exemple de Deuxième cas.*

2.3.3.3 Troisiéme Cas

S'il y a des événements de communication entre deux requête différents, pour bien expliquée le problème nous avons les deux figures 2.10 et 2.11 :

Si P_3 a envoyé un message au processus P_6 qui n'appartient pas à l'ensemble de dépendance de P_4 alors il envoyé tout simplement un reply-négative la figure 2.10 .

Par contre dans la figure 2.11 , P_3 a envoyé un message à P_4 le message m_4 devient orphelin pour résoudre le problème nous avons proposé deux solutions .

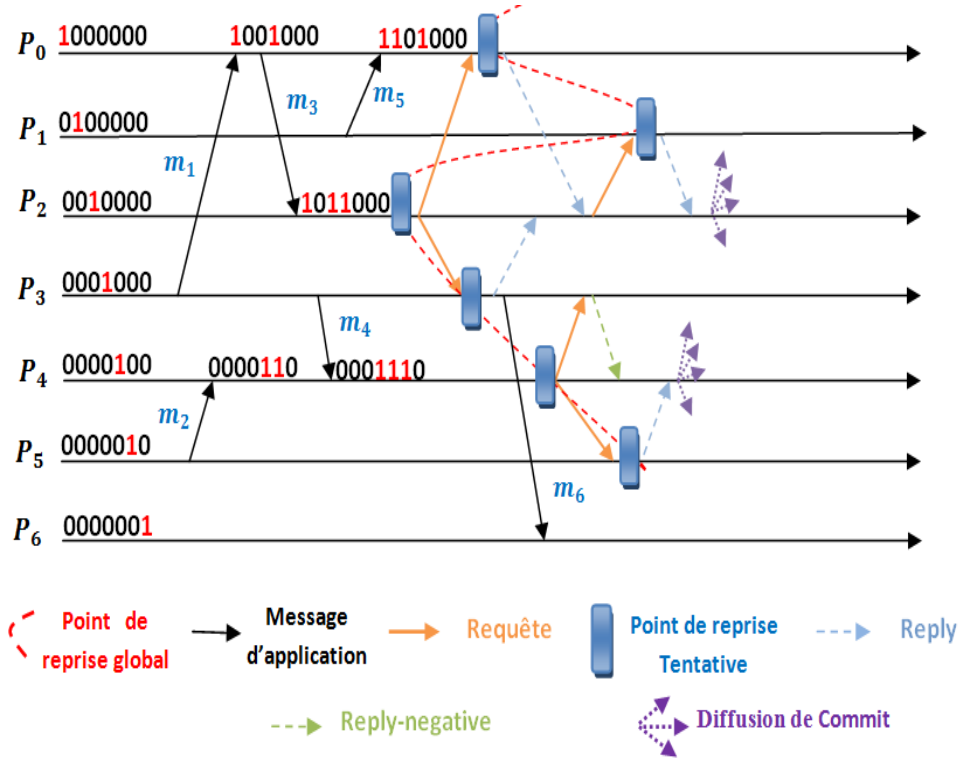

Figure 2.10 – *Message d'application envoyé à un autre .*

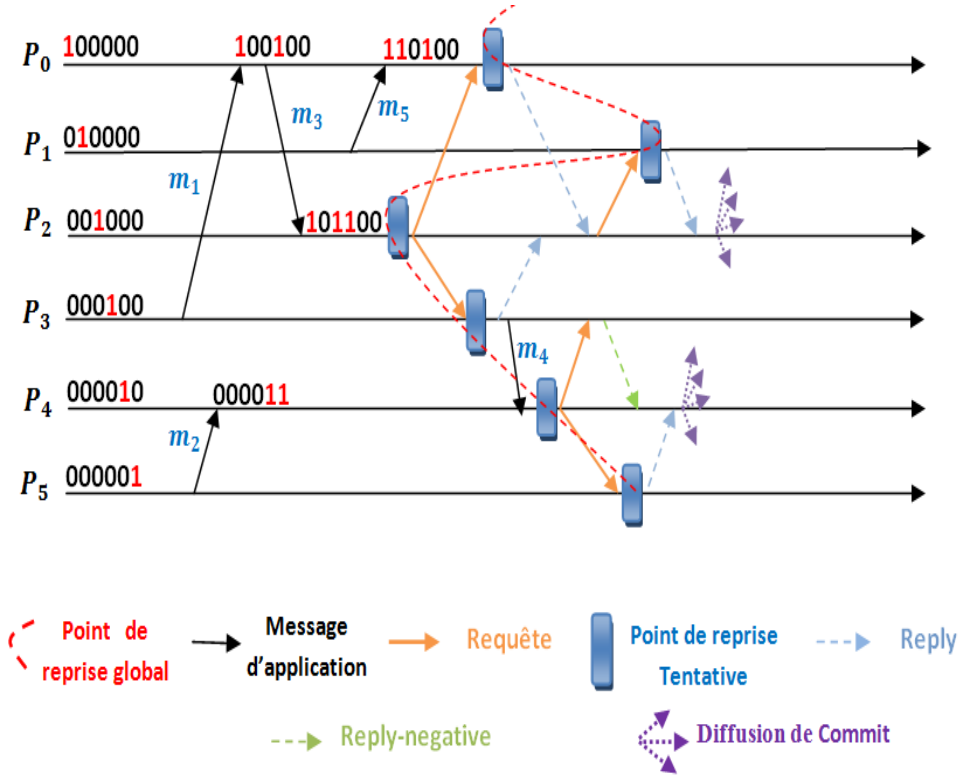

Figure 2.11 – *Message d'application orphelin .*

2.3.3.3.a Première proposition : gestion de l'ensemble de conflit

Pour bien expliquée ce problème, nous allons montré la solution dans la figure 2.12 :

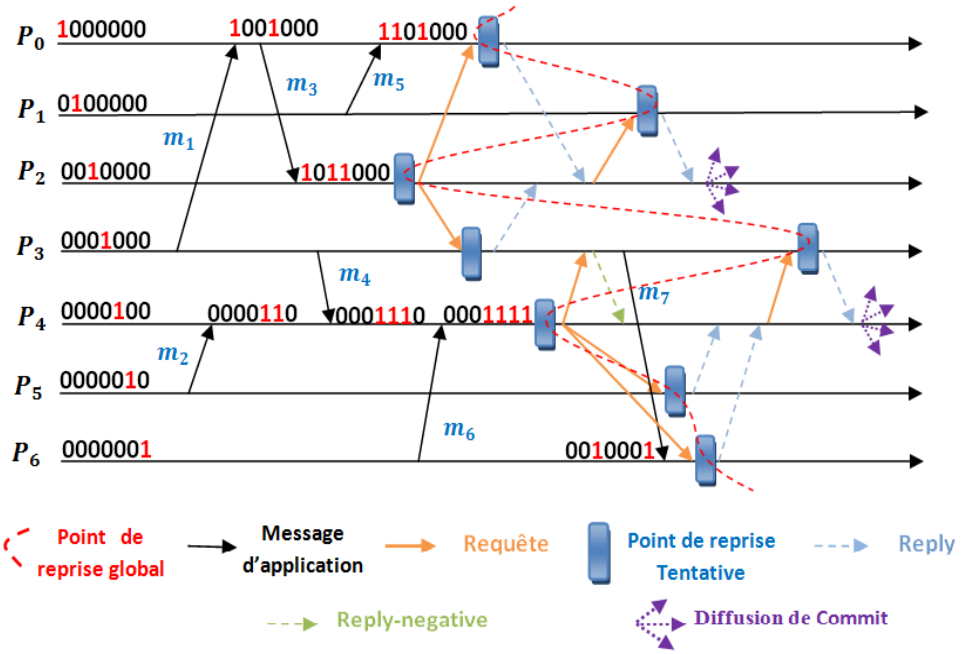

Figure 2.12 – *Exemple de* 1^{ere} *proposition .*

P_4 détecte que le processus P_3 appartient à l'ensemble de conflit (entre P_2 , P_4) après la réception de message reply de P_6 et P_3 a envoyé un message m_7 à P_6 et $P_6 \in$ à l'ensemble de dépendance de P_4 dans ce cas P_3 doit prendre un point de reprise et supprimer l'ancien point de reprise pour garder la cohérence de point de reprise globale.

2.3.3.3.b deuxième proposition : retarder la réception de message

Dans la figure 2.13 ,P_6 test le csn porté par le message m_7 , si ce csn est supérieur ou égale à son csn alors il enregistre ce message et retarde la consommation jusqu'il prend un point avec un csn égal a celle porté par le message .

En plus pour améliorer cette solution nous avons utilisé un leader .

L'idée derrière l'utilisation de leader : si nous laissons les initiateurs diffusent les messages commit alors on a $(n * m)$ message de commit [n : nombre de processus, m : nombre de initiateur] , cette diffusion peut surcharger le réseau pour cela nous avons utilisé un leader.

Quand l'initiateur lance le calcul de point de reprise, il envoie un message information au leader.

Si l'initiateur détecte la fin de la première phase, il envoie un message end-phase.

Si le leader a reçu tous les messages end-phase de tous les initiateurs, il diffuse un message commit (on a seulement n message commit) .

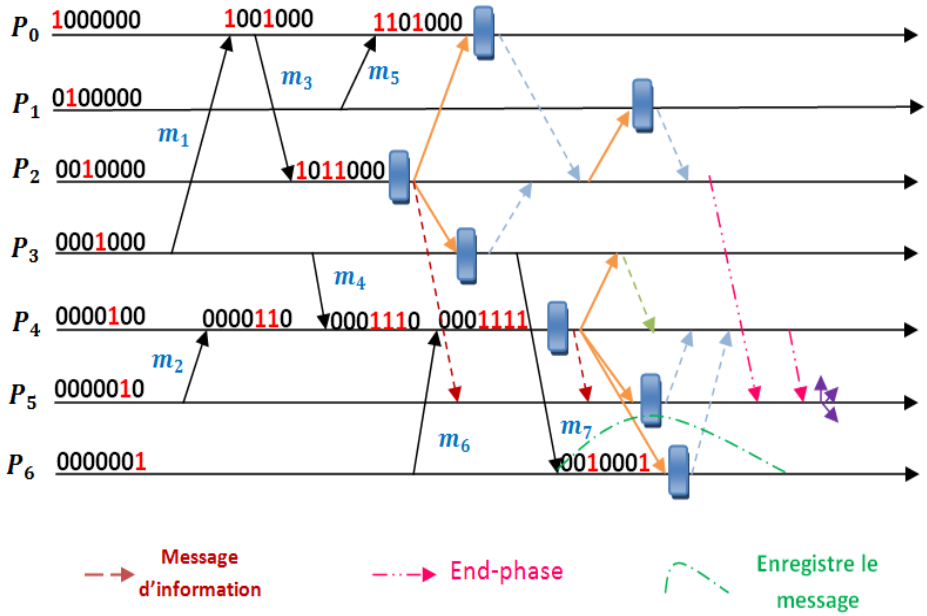

Figure 2.13 – *Exemple de* 2^{eme} *proposition .*

2.4 Conclusion

Dans ce chapitre nous avons présenté les algorithmes [Algorithme de R.Prakash et M.Singhal pour les canaux FiFo et L'algorithme D. Goswami et S. Majumder pour les canaux Non FiFo] ainsi que la généralisation de l'algorithme NNB avec plusieurs initiateurs .

Nous avons propose deux solution la 1^{ere} gère l'ensemble de conflit et le deuxième en registrée le message plus l'utilisation d'un processus leader pour minimiser le nombre de message commit.

Evaluation des performances 3

Sommaire

3.1 Introduction

La simulation permet de tester à moindre coût les nouveaux protocoles et d'anticiper les problèmes qui pourront se poser dans le futur afin d'implémenter la technologie la mieux adaptée aux besoins.

Dans ce présent chapitre nous allons essayer de réaliser et analyser la simulation pour évaluer les performances des protocoles présentés dans le chapitre 2.

Lors de ces études on va interpréter également les courbes obtenues à partir des scénarios proposés pour chaque protocole.

3.2 L'outil de simulation

Nous avons utilisé NS2 (Network Simulator 2) comme outil de simulation, cet outil est un logiciel de simulation de réseaux informatiques, il est principalement bâti avec les idées de la conception par objets, de réutilisabilité du code et de modularité, il est devenu aujourd'hui un standard de référence en ce domaine.

Ce logiciel est open source. Son utilisation est gratuite et exécutable tant sous Unix et Windows.

Nous avons réalisé la simulation sous Linux Mandriva 2010, en utilisant la version ns-allinone-2.34 , et on a utilisé un ordinateur dont la configuration suivante :

- Processeur : Pentium(R) Dual-Core CPU $T4400$ @ $2.20GHz$.
- Mémoire : 3 Go.
- Disque dur : 250 GB /Go.

3.3 Réalisation de simulation

Afin de réaliser notre simulation des algorithmes $(AM1, AM2)$, nous avons intégré ces protocoles dans le simulateur NS-2.

Pour intégrer ces algorithmes, nous avons besoin de la création de deux fichiers sources écrits en langage c++ (*.h, *.cc) :

- le fichier *.h est utilisé pour la déclaration des variables et contient la structure des messages utilisés dans chaque protocole.

- le fichier *.cc est utilisé pour définir l'algorithme qui contient ces procédures.

Les étapes d'intégration sont les suivantes :

1. Modifier le fichier makefile, pour cela nous avons ajouté à la variable OBJ_CC :lechemin/AM1.o .

2. Modifier le fichier packet.h pour connaître notre agent (dans le répertoire common, ajouter : (static const packet_t PT_AM1 = 72 ;) à (typedef unsigned int packet_t ;) et (name_[PT_AM1]="AM1" ;) à (class p_info).

3. Modifier le fichier ns-default.tcl (dans le répertoire tcl/lib, ajouter : Agent/AM1 set packetSize_ 64 et à la suite, initialiser les variables utilisées dans le programme).

4. À la fin de ces modifications, il faut recompiler NS-2 par la commande make. Un fichier de type objet (*.o) sera généré après la recompilation de NS-2.

On peut résumer ces étapes dans la figure suivante :

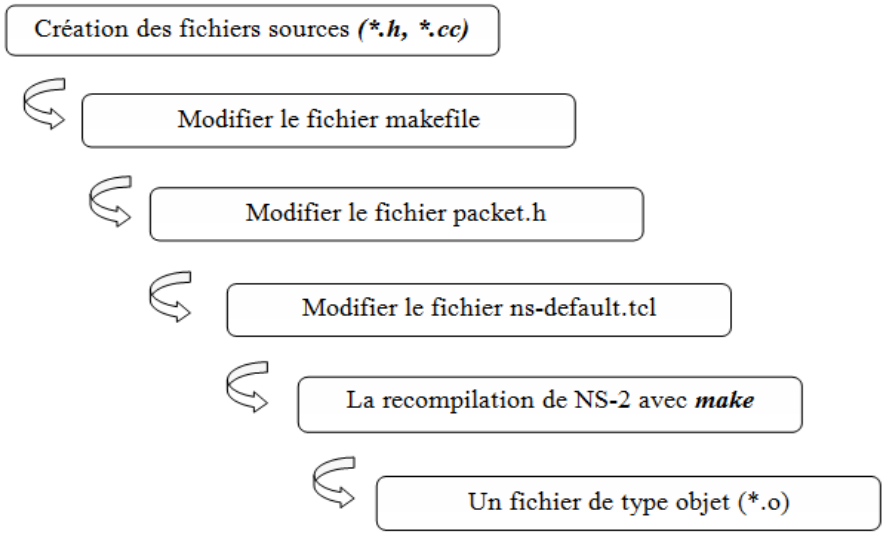

Figure 3.1 – *Les étapes de la création d'un protocole.*

3.4 Les étapes de simulation

Après l'ouverture du terminal et celle du répertoire où se trouvent les fichiers contenant le programme à exécuter :

– Le fichier (scenario.tcl) permet à l'utilisateur de saisir les informations nécessaires pour la simulation (le nombre de processus, le nombre de messages, le nombre d'initiateur) ces informations sont enregistrées dans un fichier appelé scenario.txt.

– Le fichier messages.txt est devenu aussi comme résultat de premier fichier, contient les informations suivantes (le processus source, le processus destination, et le moment d'envoie).

Ces fichiers (*.txt) sont nécessaires pour l'exécution des autres fichiers (AM1.tcl, AM2.tcl), ces derniers fichiers permettent de :

- Lancer le Nam où on peut animer la simulation. Le lancement de simulation permet de voir les nœuds, les liens entre les nœuds, l'envoi et la réception des messages,etc.

- Obtenir les résultats de la simulation pour tracer des courbes afin d'analyser la performance de l'algorithme simulé.

On peut résumer ces étapes dans la figure suivant :

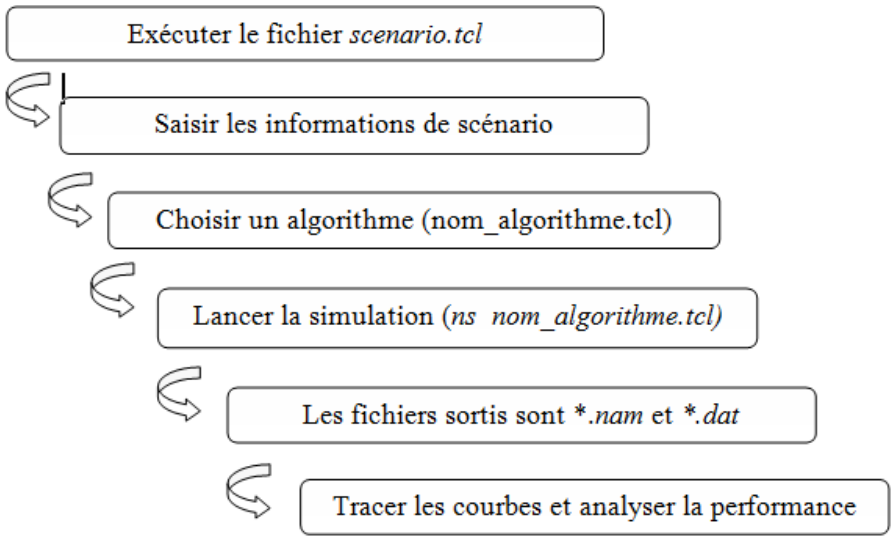

Figure 3.2 – *Les étapes de simulation.*

3.5 La simulation

Dans notre simulation , nous avons varier le nombre de processus (Nbr-proc) de 15 à 40 . Nous avons fixé le nombre de message (Nbr-msg) à 5000 et le nombre d'initiateur concurrent (Nbr-int-conc) à 8 pour une durée de simulation 100 s .

- La durée moyenne

- Dans la figure 3.3 on remarque que la durée moyenne augmente de manière progressive du a l'augmentation de nombre de processus. L'amélioration 1 prend plus de durée que l'amélioration 2, car que le deuxième enregistrée le message d'application.

- On prend un exemple : Nb-proc est 30, la durée moyenne de amélioration 1 est 1.1 et de amélioration 2 est 0.25 .

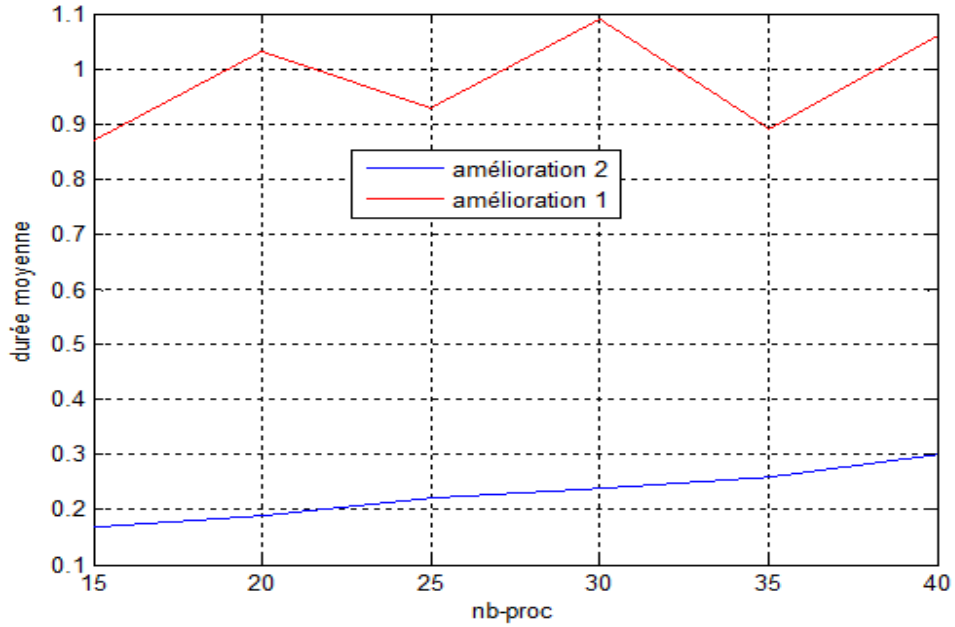

Figure 3.3 – *La durée moyenne .*

- Le nombre de requêtes

On a obtenu la résultat presenté dans la figure 3.4 :

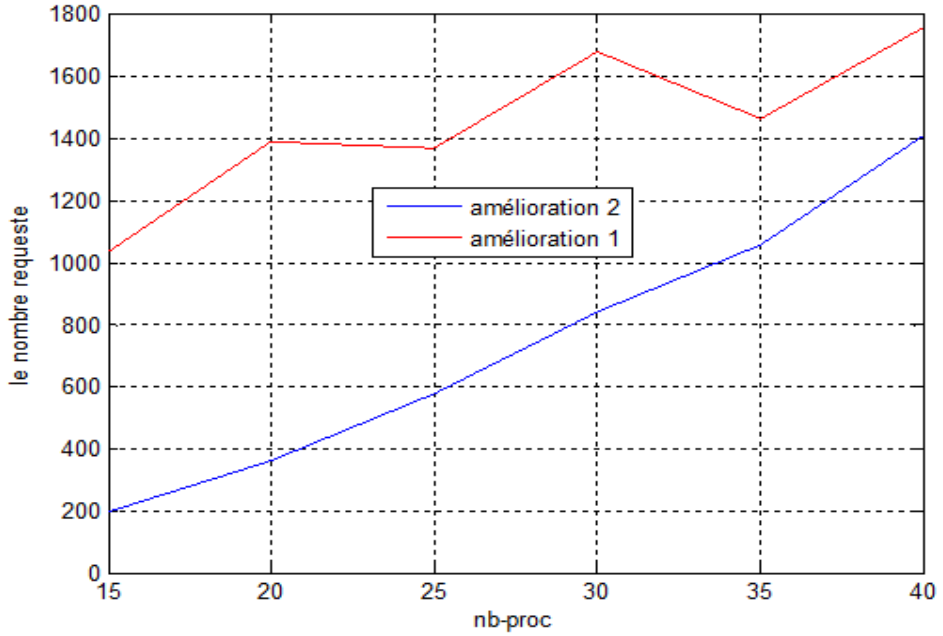

Figure 3.4 – *Le nombre de requête .*

– Dans la figure 3.4 en remarque que le nombre de requêtes augmente de manière

progressive du a l'augmentation de nombre de processus.

– L'amélioration 1 prend plus de nombre de requête que la deuxième, la premier solution gère un ensemble de conflit pour la quelle il doit envoyer des requêtes.

- Le nombre de point de reprise tentatif

on a obtenu la résultat suivant :

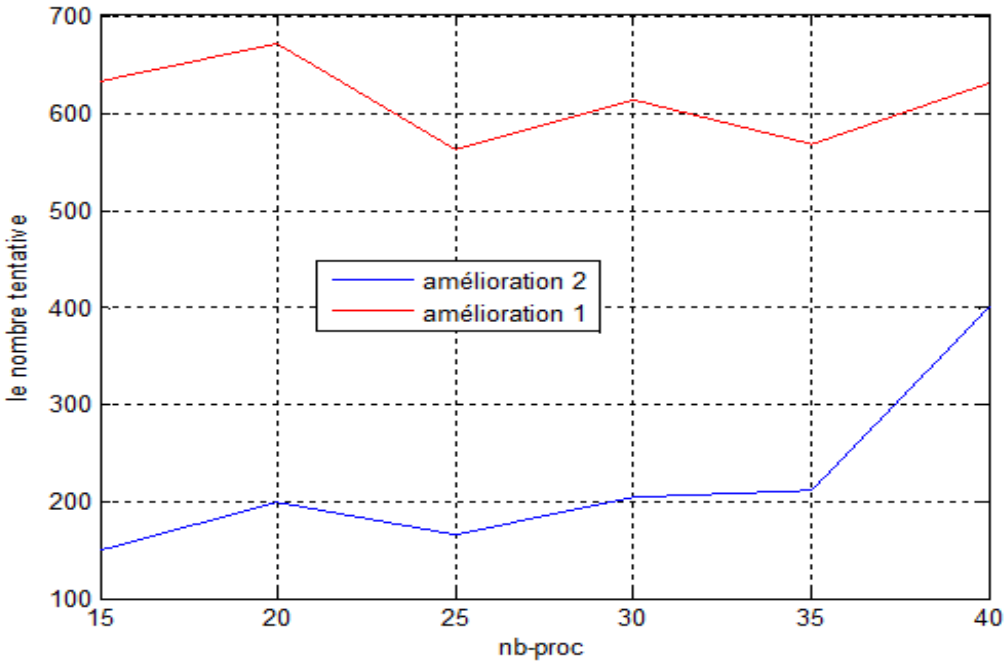

Figure 3.5 – *Le nombre de point de reprise tentatif .*

– la figure 3.5 montre que le nombre de point de reprise tentatif augmente de manière progressive du a l'augmentation de nombre de processus.

– L'amélioration 1 prend plus de nombre de point de reprise tentatif que la deuxième puisque la deuxième solution utilise la technique de enregistrement de message, et il n'oblige pas les processus de prendre le point de reprise tentatif pour assurer la cohérence de point de reprise globale.

3.6 Conclusion

Dans cette simulation, nous avons essaye d'évaluer la performance des solutions proposé, nous avons remarqué que la deuxième solution et plus performante que la première solution.

Conclusion Générale

Il existe trois approches pour le calcul du point de reprise global cohérent de la reprise des systèmes :

- L'approche coordonnée (Coordinated Checkpointing).
- L'approche non coordonnée (Uncoordinated Checkpointing).
- L'approche induit par communication (Communication-Induced Checkpointing).

À cet effet, Le but de notre travail est de simuler des algorithmes dans la classe coordonnée. on a proposé deux approches pour généraliser l'algorithme NNB proposé par Z.Abdelhafidi et M.Djoudi et M.B.Yagoubi en 2012 permet de calculer l'état global dans les systèmes répartis avec plusieurs initiateurs .

L'évaluation de ces algorithmes a été réalisée sous NS-2, pour cela nous avons varie le nombre de processus pour étudier son influence sur les performances.

À partir de cette simulation, nous avons remarqué que la deuxième proposition est toujours meilleur que la proposition 1 .

Travaux futurs : Améliorer la performance de la deuxième solution et réduire le nombre de requêtes utilisé.

Bibliographie

[Abd07] Z. Abdelhafidi. Points de reprise dans les systèmes répartis etude basée sur la simulation des protocoles cic assurant la propriété rdt. *Mémoire de Magister de l'université Amar Telidji- Laghouat Spécialité informatique*, 2007. (Cité pages 7, 10 et 11.)

[ADY12] Z Abdelhafidi, M. Djoudi, and M.B. Yagoubi. An improved schema of coordinated checkpointing protocol for distributed systems based on popular process. 2012. (Cité page 19.)

[BL07] S. Benkouider and S. Labgaa. Etude des performances des algorithmes répartis d'exclusion mutuelle avec le simulateur ns-2. *Thèse d'ingénieur de l'université Amar Telidji-Laghouat Spécialité informatique*, 2007. (Cité page 4.)

[CL85] K. Chandy and L Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 1985. (Cité pages 6 et 9.)

[CS03] G. Cao and M Singhal. Checkpointing with mutable checkpoints. *Theoretical ComputerScience* 290, 2003. (Cité page 10.)

[GM11] D. Goswami and S. Majumder. A global snapshot collection algorithm with concurrent initiators with non-fifo channel. *ICA3PP*, 2011. (Cité page 17.)

[Kai04] C. Kaiser. Systèmes et applications répartis : Bases pour l'algorithmique répartie. *Cours B4 code* 193002 *de Conservatoire National des Arts et Métiers CNAM*, 2004. (Cité page 7.)

[KT87] R. Koo and Toueg.S. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 1987. (Cité page 10.)

[LMP93] G. Lelann, P. Minet, and D. Powell. tolérance aux pannes et systèmes répartis : Conception et mécanismes. *Rapport de recherche : Institut National de Recherche en Informatique et Automatique*, 1993. (Cité page 4.)

[Mak11] C. Makassikis. Conception d'un modèle et de frameworks de distribution d'applications sur grappes de pcs avec tolérance aux pannes faible coût. *Doctorat de l'université Henri Poincaré - Nancy spécialité informatique*, 2011. (Cité page 7.)

[PS94] R. Prakash and M. Singhal. Maximal global snapshot with concurrent initiators. *Sixth IEEE Symp. Parallel and Distributed Processing*, 1994. (Cité page 14.)

[Rod07] J. Rodriguez. Systèmes et applications répartis. *Cour NFP*111 *de Conservatoire National des Arts et Métiers CNAM*, 2007. (Cité page 4.)

4 Annexe : Les procédures de l'algorithme AM 1

les variables ajoute :

$conf_i$[j] :Tableau booléen de N éléments . égale à 1 ,s'il existe un conflit entre le processus P_i et P_j.

$confcsn_i$[j] :Tableau d 'entier de n élément . contient le csn de processus P_j en conflit avec P_i .

$SendVection_i$: Tabeau de n élément . SendVection[j] $>$ 0 si P_i a envoyé un message

d'application à P_j .

Procédure 1 : **Lorsque** P_i **envoie message d'application à** P_j

```
1  Début
2      Si (stat_i = 1) Alors
3          SentVector[j] := 1 ;
4          sent_i := 1 ;
5          Envoyer (P_i,message,csn_i[i],DV ,own-trigger);
6      Sinon
7          sent_i := 1 ;
8          Envoyer(P_i,message,csn_i[i],DV ,Null);
9      Finsi
10 Fin
```

Procédure 2 : **Lorsque** P_i **lance le calcul de point de reprise global**

```
1  Début
2      csn_j[j] ++ ;
3      own-trigger := (P_j , csn_j[j]);
4      state_j := 1 ;
5      pour ( k := 0 jusqu'a N ) faire
6          MR[k].csn := 0;
7          MR[k].DV := 0;
8          SentVector[k] := 0 ; conf[k]:= 0 ; confcsn[k] := 0;
9      fin
10     MR[j].csn := csn_j[j];
11     MR[j].DV := 1;
12     pour ( Chaque P_k ,tel que (DV_i[k]= 1 ) ∧ ( max(MR[k].csn, csn_i[i]) !=
       MR[k].csn)) faire
13         nb-req++ ;
14         Envoyer (P_i, request, csn_i[i], own-trigger, csn_i[k], DV_i) ;
15     fin
16     pour (k := 0 jusqu'a N ) faire
17         S_i[k] := max(DV_i[k] , S_i[k]);
18         conf_i.[k]:= 0 ;
19         confcsn_i[k]:= 0;
20     fin
21     Prend un point de reprise local ;
22     old-csn_j:= csn_j[j] ;
23     Initialiser sent_j ;
24     Initialiser DV_j ;
25 Fin
```

Procédure 3 : **Lorsque** P_i **reçoit un message request depuis** P_j

```
1  Début
2      Réception(P_j , request, recv-csn, msg-trigger, req-csn , recv-DV ) ;
3      csn_i[ j] := recv-csn ;
4      Si (old-csn_i > req-csn ) Alors
5          Envoyer(P_i, reply, NULL) à l'initiateur ; return ;
6      Finsi
7      Si ( state_i = 0) Alors
8          state_i := 1 ;
9          Si ( msg-trigger = own-trigger ) Alors
10             Si ( CP_i.trigger = msg-trigger ) Alors
11                 sauve CP_i mutable sur stable storage ; old-csn_i := csn_i[i] ;
12                 pour (k := 0 jusqu'a N) faire
13                     MR[k].csn:= CP_i.csn[k] ; MR[k].DV:= CP_i.DV[k];
14                 fin
15                 CP_i := NULL ;
16                 Envoyer (P_i, reply, MR) to the initiator ;
17             Finsi
18         Sinon
19             csn_i[i]++ ; own-trigger := msg-trigger ;
20             Prend un point de reprise local ; old-csn_i := csn_i[i];
21             pour (k := 0 jusqu'a N)faire MR[k].csn := csn_i[k] ; MR[k].DV :=
               DV[k];
22             Envoyer(P_i, reply, MR) à l'initiateur;
23             sent_i := 0 ; Initialiser DV ;
24         Finsi
25     Sinon
26         pour (k := 0 jusqu'a N )faire Si( SendVector[k] = recv-DV[k]) Alors
           count ++ ;
27         Si (count = 0 ) Alors      /* l'émission déjà sauvegarde */
28             Envoyer (P_j ,Replay-Negative, csn_i[i] );
29         Sinon
30             Si (CP_i .mutable = true ) Alors
31                 sauve CP_i mutable sur stockage stable ; old-csn_i:= csn_i[i];
32                 pour k := 0 jusqu'a N faire
33                     MR[k].csn := CP_i.csn[k] ; MR[k].DV CP_i. DV[k] ;
34                 fin
35                 CP_i :=NULL ; trigger_i := msg-trigger ;
36                 Envoyer (P_i, reply, MR) à l'initiateur ;
37             Sinon
38                 csn_i[i] ++ ; own-trigger := msg-trigger ;old-csn_i := csn_i[i] ;
39                 Prend point de reprise local;
40                 pour k := 0 jusqu'a N faire
41                     MR[k].csn := csn_i[k] ; MR[k].DV :=DV[k] ;
42                 fin
43                 Envoyer(Pi, reply, MR) à l'initiateur ; Initialiser sent_i ;
                   Initialiser DV ;
44             Finsi
45         Finsi
46     Finsi
47 Fin
```

Procédure 4 : **Lorsque** P_i **reçoit un message d'application depuis** P_j

```
1  Début
2      reçoit (P_j , m, recv-csn, MDV, msg-trigger);
3      Si ( recv-csn < csn_i[j] ) Alors
4          Traiter le message ;
5      Sinon
6          Si ( csn_i[msg-trigger.pid] ≥ msg-trigger.inum ) Alors
7              csn_i[ j] := recv-csn ;
8              pour k := 0 jusqu'a N faire
9                  DV_i := max( MDV[k], DV_i[k]) ;
10             fin
11             Traiter le message ;
12         Sinon
13             csn_i[j] := recv-csn ; ;
14             Si ((msg-trigger ≠ NULL) ∧(sent_i = 1) ∧ (msg-trigger ≠
               own-trigger)) Alors
15                 Si ( CP_i.mutable = false) Alors        /* n'existe pas
                   mutable  */
16                     sauve point de reprise mutable sur le stockage stable ;
17                     CP_i.mutable :=true ;
18                     CP_i.trigger := msg-trigger;
19                     CP_i.DV := DV_i ;
20                     CP_i.sent := sent_i ;
21                     Initialiser sent_i ; Initialiser DV_i ;
22                 Finsi
23             Finsi
24             Si ((msg-trigger ≠ NULL) ∧ (state_i = 0 )) Alors
25                 state_i := 1 ; csn_i[i] ++ ;
26                 own-trigger := msg-trigger;
27                 pour ( k := 0 jusqu'a N ) faire
28                     DV_i := max( MDV , DV_i[k]);
29                 fin
30                 Traiter le message ;
31             Finsi
32         Finsi
33     Finsi
34 Fin
```

Procédure 5 : **Lorsque** P_i **reçu un message Reply-Negative depuis** P_j

```
1  Début
2      Réception (P_j , Replay-Negative , recv-csn) ;
3      nb-reply ++ ;
4      conf_i[j] := 1 ;
5      confcsn_i[j] := recv-csn ;
6  Fin
```

Procédure 6 : **Lorsque** P_i **reçu un message Reply depuis** P_j

```
1  Début
2      Réception(P_j , reply , MR ) ;
3      nb-reply ++ ;
4      pour ( Chaque P_k , such that (MR[k].DV= 1)∧ (max(MR[k]
       csn,csn_i[k])≠ MR[k].csn)) faire
5          Si (S_i[k] = MR [k].DV) ∧ (conf_i[k] = 1 ) Alors
6              Si ( MR [k].csn ≥ confcsn_i[k] ) Alors
7                  nb-req ++ ;
8                  Envoyer(P_i,request ,csn_i[i], own-trigger,csn_i[k] ,DV);
9              Finsi
10         Finsi
11         Si (S_i [k]≠ MR [k].DV) Alors
12             nb-req ++ ;
13             Envoyer(P_i, request, csn_i[i], own-trigger, csn_i[k] ,DV) ;
14         Finsi
15     fin
16     pour (k := 0 jusqu'a N ) faire
17         S_i[k] := max( MR[k].DV , S_i[k] ) ;
18     fin
19     Si ( nb-reply =nb-req ) Alors
20         state_i := 0 ;
21         diffuser ( Commit ,own-trigger ) ;
22         nb-req:= 0 ; nb-reply:= 0 ;
23     Finsi
24 Fin
```

Procédure 7 : **Lorsque le processus** P_j **reçu message commit**

```
1  Début
2      Réception(Commit, msg-trigger) ;
3      csn_j[msg-trigger.pid]:= msg-trigger. inum ;
4      state_j := 0 ;
5      Si (CP_j.trigger = msg-trigger)∧( CP_j ≠ NULL) Alors
6          sent_j := sent_j ⋃ CPj.sent ;
7          DV_j := DV_j ⋃ CP_j.DV;
8          CP_j := NULL;
9      Finsi
10     Si Existe point de reprise tentative consernant msg-trigger Alors
11         rend permanent ;
12     Finsi
13 Fin
```

5 Annexe : Les procédures de l'algorithme AM 2

les variables ajoute :

TCT : (Tableau Commit Totale)Tableau de trigger de N éléments , cet variable utilise par le leader.

CP_i : tableau of enregistrement déclare dans l' algorithme NNB , utilise pour enregistre

les points de reprise mutables .

Procédure 1 : **Lorsque** P_i **envoie message d'application à** P_j

```
1 Début
2   Si (stat_i = 1) Alors
3     sent_i := 1 ;
4     Envoyer (P_i,message,csn_i[i],DV ,own-trigger);
5   Sinon
6     sent_i := 1 ;
7     Envoyer(P_i,message,csn_i[i],DV ,Null);
8   Finsi
9 Fin
```

Procédure 2 : **Lorsque** P_i **lance le calcul de point de reprise global**

```
1  Début
2    csn_j[j] ++ ;
3    own-trigger := (P_j , csn_j[j]);
4    state_j := 1 ;
5    pour ( k := 0 jusqu'a N ) faire
6      MR[k].csn := 0;
7      MR[k].DV := 0;
8    fin
9    MR[j].csn := csn_j[j];
10   MR[j].DV := 1;
11   Envoyer ( Information ,own-trigger ) ;
12   pour ( Chaque P_k ,tel que (DV_i[k]= 1 ) ∧ ( max(MR[k].csn, csn_i[i]) !=
     MR[k].csn)) faire
13     nb-req++ ;
14     Envoyer (P_i, request, csn_i[i], own-trigger, csn_i[k], DV_i) ;
15   fin
16   pour (k := 0 jusqu'a N ) faire
17     S_i[k] := max(DV_i[k] , S_i[k]);
18   fin
19   Prend un point de reprise local ;
20   old-csn_j:= csn_j[j] ;
21   Initialiser sent_j ;
22   Initialiser DV_j ;
23 Fin
```

Procédure 3 : **Lorsque** P_i **reçoit un message request depuis** P_j

```
1  Début
2    Réception(P_j , request, recv-csn, msg-trigger, req-csn) ;
3    csn_i[ j] := recv-csn ;
4    Si (old-csn_i > req-csn ) Alors
5      Envoyer(P_i, reply, NULL) à l'initiateur ; return ;
6    Finsi
7    Si ( state_i = 0) Alors
8      state_i := 1;
9      existe := false;
10     pour l := 0 jusqu'a N faire
11       Si ( CP_i[l].trigger = msg-trigger ) Alors
12         sauve CP_i[l] mutable sur stable storage ;
13         old-csn_i := csn_i[i] ;
14         existe := true ;
15         pour (k := 0 jusqu'a N) faire
16           MR[k].csn:= CP_i[l].csn[k] ;
17           MR[k].DV:= CP_i[l].DV[k];
18         fin
19         pour (k := 0 jusqu'a l) faire
20           CP_i[k] := NULL ;
21         fin
22         Envoyer (P_i, reply, MR) to the initiator ;
23       Finsi
24     fin
25     Si ( existe = false) Alors
26       csn_i[i]++ ; own-trigger := msg-trigger ;
27       Prend un point de reprise local ; old-csn_i := csn_i[i];
28       pour (k := 0 jusqu'a N)faire MR[k].csn := csn_i[k] ;
         MR[k].DV := DV[k];
29       Envoyer(P_i, reply, MR) à l'initiateur;
30       sent_i := 0 ; Initialiser DV ;
31     Finsi
32   Sinon
33     Envoyer(P_i, reply-négative) à l'initiateur ;
34   Finsi
35 Fin
```

Procédure 4 : **Lorsque leader reçoit message Information depuis** P_j

```
1  Début
2    count_init ++ ;
3    TCT[j] := recu-trigger ;
4  Fin
```

Procédure 5 : **Lorsque** P_i **reçoit un message d'application depuis** P_j

```
1  Début
2      reçoit (P_j , m, recv-csn, MDV, msg-trigger);
3      Si ( recv-csn < csn_i[j] ) Alors
4          Traiter le message ;
5      Sinon
6          Si ( csn_i[msg-trigger.pid] ≥ msg-trigger.inum ) Alors
7              csn_i[ j] := recv-csn ;
8              pour k := 0 jusqu'a N faire
9                  DV_i := max( MDV[k], DV_i[k]) ;
10             fin
11             Traiter le message ;
12         Sinon
13             csn_i[j] := recv-csn ; state_i := 1 ;
14             Si ((msg-trigger ≠ NULL) ∧(sent_i = 1) ∧ (msg-trigger ≠
               own-trigger)) Alors
15                 sauve point de reprise mutable sur le stockage
                   stable ;
16                 CP_i[j].mutable :=true ;
17                 CP_i[j].trigger := msg-trigger;
18                 CP_i[j].DV := DV_i ;
19                 CP_i[j].sent := sent_i ;
20                 Initialiser sent_i ; Initialiser DV_i ;
21             Finsi
22             Si (recv-csn > old − csn) Alors
23                 Si ((msg-trigger ≠ NULL) ∧ (state_i = 0 )) Alors
24                     state_i := 1 ; csn_i[i] ++ ;
25                     own-trigger := msg-trigger;
26                     pour ( k := 0 jusqu'a N ) faire
27                         DV_i := max( MDV , DV_i[k]);
28                     fin
29                     Traiter le message ;
30                 Sinon
31                     Enregistre message dans la file d'attente ;
32                 Finsi
33             Finsi
34         Finsi
35     Finsi
36 Fin
```

Procédure 6 : **Lorsque** P_i **reçu un message Reply depuis** P_j

```
1  Début
2  |  Réception(P_j , reply , MR ) ;
3  |  nb-reply ++ ;
4  |  pour ( Chaque P_k , such that (MR[k].DV= 1)∧ (max(MR[k]
   |  csn,csn_i[k])≠ MR[k].csn)) faire
5  |  |  Si (S_i [k]≠ MR [k].DV) Alors
6  |  |  |  nb-req ++ ;
7  |  |  |  Envoyer(P_i, request, csn_i[i], own-trigger, csn_i[k] ,DV) ;
8  |  |  Finsi
9  |  fin
10 |  pour (k := 0 jusqu'a N ) faire
11 |  |  S_i[k] := max( MR[k].DV , S_i[k] ) ;
12 |  fin
13 |  Si ( nb-reply =nb-req ) Alors
14 |  |  Envoyer ( End-phase , time ) à leader ;
15 |  Finsi
16 Fin
```

Procédure 7 : **Lorsque** P_i **reçoit message de reply-négative**

```
1  Début
2  |  nb-reply ++ ;
3  Fin
```

Procédure 8 : **Lorsque leader reçoit message End-phase depuis** P_j

```
1  Début
2  |  count_init −− ;
3  |  TCtp := time;
4  |  Si count_init = 0 Alors
5  |  |  diffuse (Commit ,TCT );
6  |  Finsi
7  Fin
```

Procédure 9 : **Lorsque le processus** P_j **reçu message commit**

```
1  Début
2      Réception(Commit, TCT) ;
3      k := 0jusqu'aN csn_j[TCP[k].pid]:= TCP[k].inum ;
4      Si (CP_j.trigger = TCP[k])∧( CP_j ≠ NULL) Alors
5          sent_j := sent_j ⋃ CPj.sent ;
6          DV_j := DV_j ⋃ CP_j.DV;
7          CP_j := NULL;
8      Finsi
9      state_j := 0 ;
10     Si Existe point de reprise tentative consernant msg-trigger Alors
11         rend permanent ;
12     Finsi
13 Fin
```