RENPAR/CFSE/SYMPA
Paris, cité des sciences et de l'industrie, 24 – 27 avril 2001

Protocole de membership hautement extensible : conception et expérimentations*

Bertil Folliot, Gaël Thomas

Université Paris 6, LIP6/CNRS
4, place Jussieu – F-75252 Paris
prenom.nom@lip6.fr

Résumé
La gestion du membership (connaissance des machines actives et détection des machines fautives) est un composant essentiel pour la plupart des serveurs basés sur des grappes de machines. La plupart des protocoles de membership existants sont faiblement extensibles avec le nombre de noeuds, limitant ainsi l'extensibilité du service. Cet article présente la conception d'un protocole de membership basé sur une structure en anneau à deux niveaux de multicast, extensible à plus de 1000 noeuds. Des expérimentations sur 70 machines simulant des grappes jusqu'à 1024 noeuds montrent pour de nombreux cas (démarrage à froid, noeuds fautifs, partitionnement du réseau) l'efficacité de ce protocole.

Mots-clés : Membership, extensibilité, serveur sur grappe de noeuds

1. Introduction

Les serveurs réseaux basés sur des grappes de machines (cluster-based servers) sont de plus en plus populaires. En particulier, un réseau de machines PC ordinaires connectées par un réseau à haut-débit combinent les avantages d'extensibilité incrémentale, de haute disponibilité et de faible coût [11]. Ces grappes sont utilisées notamment pour les calculs parallèles et répartis [3], les systèmes de fichiers [2], et tout particulièrement les services Internet [11], comme le WWW [13] ou le courrier électronique [16]. Pour accomplir sa tâche, chaque noeud de la grappe doit pouvoir déterminer l'ensemble des noeuds atteignables, même en cas de perte de messages, de fautes de machines ou de partition du réseau. C'est le rôle du gestionnaire de membership de conserver la liste des noeuds actifs, de détecter les noeuds fautifs ou inatteignables et de réintégrer les nouveaux noeuds actifs.
Le membership est connu pour être un problème difficile, impossible à résoudre dans un système réparti asynchrone [10, 5]. Un noeud peut apparaître fautif parce que le processus, ou le réseau, est trop lent. De plus, le réseau peut être partitionné en multiples sous-réseaux inatteignables. Le problème du membership a été largement étudié, mais la plupart des travaux considèrent la présence d'un coordinateur global, en charge de constituer et diffuser le membership [4, 1, 7]. Lorsque le nombre de noeuds augmente le réseau devient saturé, pouvant empêcher certains noeuds de confirmer leur vivacité au coordinateur dans des délais raisonnables. Le résultat est qu'au delà de quelques douzaines de noeuds, soit un nombre arbitraire de noeuds actifs est déclaré fautif, soit le temps de constitution du membership devient très élevé (pouvant empêcher le serveur de fonctionner pendant ce temps et augmentant la probabilité que des fautes/partitions apparaissent, rendant le membership caduc). Cette approche est donc peu extensible, limitant ainsi l'extensibilité des serveurs sur grappe de noeuds [18, 9].
Nous présentons et évaluons dans ce papier un nouveau protocole de membership dont le but est d'être extensible à plus d'un millier de noeuds dans une grappe (où les temps de communication sont à peu près homogènes). Ce protocole est basé sur une structure en anneau à deux niveaux. Chaque noeud appartient à un des nombreux anneaux de deuxième niveau (appelés anneaux locaux). L'unique anneau du premier niveau (appelé anneau global) est construit de manière dynamique par un ensemble de noeuds, chaque

*La plus grande partie de la conception du protocole présenté a été faite en collaboration avec l'Université de Seattle à Washington, dans le cadre du serveur de mail hautement extensible Porcupine [15].

noeud étant le représentant unique de son anneau local. Cette architecture est hautement extensible, car elle limite la congestion du réseau et ne présente pas un seul noeud centralisateur de l'information. La faible profondeur de notre architecture procure l'efficacité dans le temps de convergence du membership (le temps à partir duquel, en l'absence de faute, tous les noeuds ont la même vision du membership). Les congestions du réseau sont en plus limitées en utilisant des communications multicasts, en réinitialisant le membership lorsque trop de fautes sont détectées sur un court intervalle de temps et en introduisant des délais aléatoires dans les communications lors du démarrage. L'objectif est de montrer que le temps de convergence reste raisonnable et borné dans le pire des cas. Les mesures effectuées sur 70 machines simulant une grappe jusqu'à 1024 noeuds montrent que le temps de convergence du membership est d'environ 10 secondes lors du démarrage à froid, entre 4 et 20 secondes pour la détection de noeuds fautifs, et de 21 à 29 secondes en cas de partitionnement du réseau.
Nous présentons les hypothèses de fonctionnement de notre protocole dans la section 2, puis une vue d'ensemble dans la 3. La section 4 détaille les différents états du protocole. Les résultats obtenus sont présentés dans la section 5. La section 6 compare notre approche avec celles existantes. La section 7 conclut et présente des perspectives à ces travaux.

2. Hypothèses de fonctionnement

Nous présentons les hypothèses de fonctionnement de notre protocole dans cette section : modèles de système réparti, de réseau, de fautes, de partition, de membership. Nous avons essayé de ne retenir que des hypothèses qui correspondent à la pratique d'utilisation d'un serveur sur grappe de PCs.
Modèle de système réparti. Nous supposons un système réparti faiblement couplé composé d'un nombre fini de noeuds, composant une grappe (cluster). De nouveaux noeuds peuvent être ajoutés au cours du fonctionnement. Ces noeuds coopèrent pour rendre un ou plusieurs services (serveur de fichier, de mails, de pages de la Toile, etc.), coopération qui nécessite d'avoir la vision la plus correcte possible de la liste des noeuds actifs. Les noeuds sont considérés comme proches physiquement, et on considère que le réseau est homogène. Cela signifie que les temps de communication et de latence sont approximativement les mêmes partout. Les autres caractéristiques de chaque noeud sont : temps de traitement non pris en compte (i.e. vitesse suffisante pour que le processeur ne devienne pas un goulot d'étranglement), capacité mémoire non prise en compte, communication par unicast (point-à-point) et multicast (diffusion sélective), dérive d'horloge insignifiante devant les temps de timeout (délais de garde), identificateur unique dans la grappe (actuellement adresse IP), espace de stockage qui survit après une faute (disque dur).
Modèle de réseau. Notre protocole utilise les communications multicast fournies par les routeurs et les systèmes d'exploitation actuels (il pourrait sans doute fonctionner, avec moins d'extensibilité, uniquement avec des communications unicast). Nous supposons que les communications sont symétriques (si un noeud I communique avec un noeud J, J peut aussi communiquer avec I) et que l'atteignabilité des communications est la même en unicast et en multicast. Bien que les messages puissent être perdus en unicast ou en multicast, nous considérons que la plupart des messages ne le sont pas, excepté pendant des périodes d'instabilité (pouvant empêcher le protocole de membership de converger) ou en cas de partitionnement.
Modèle de fautes. Les processeurs peuvent être fautifs où devenir inatteignables d'une façon arbitraire. Nous considérons le modèle "fail-silent" (arrêt silencieux sur défaillance) où un processeur fautif arrête de communiquer avec les autres [14], et où il n'existe pas de faute maligne (auto-génération ou corruption des messages). Les fautes d'omission ou de délai sont également prises en compte. Un processeur qui ne répond pas après l'envoi d'un nombre donné de messages est considéré fautif. Les processeurs fautifs (à tort ou à raison) peuvent être redémarrés et réinsérés dans la grappe. En dehors de période d'instabilité de la grappe, le temps moyen entre deux fautes consécutives est considéré comme raisonnable (quelques minutes).
Modèle de partition. La grappe peut devenir partitionnée de manière à ce qu'un noeud à l'intérieur d'une sous-partition soit incapable de communiquer avec un noeud d'une autre sous-partition. Les partitions peuvent disparaître, rétablissant les communications entre les noeuds. Nous n'avons pas considéré le modèle de partition primaire, où seule une sous-partition continue à délivrer le service du membership. En effet, ce modèle peut empêcher le service d'une sous-partition (serveur de mail en mode dégradé par exemple). Ce modèle pourrait être implémenté comme un service au-dessus de notre protocole de

membership.

Modèle de membership. Le protocole de membership est réparti. Chaque noeud possède une copie locale de "sa" vision de la liste des sites actifs et participe à la détection des fautes. Cette liste est délivrée localement au service par l'intermédiaire d'un appel de fonction de bibliothèque. Potentiellement, une liste incorrecte peut être délivrée au service (incluant des noeuds fautifs ou excluant des noeuds actifs) et nous supposons que le service est conscient de ce fait (par exemple, délai de garde additionnel pour la reconfiguration du service). Le service sur un site doit également prendre en compte son exclusion erronée (réinitialisation à chaud).

3. Structure générale du protocole

Les machines participant au service de membership forment une structure à deux niveaux d'anneaux : le second niveau regroupe un sous-ensemble des machines (appelé groupe local), le premier niveau regroupe tous les groupes locaux (appelé groupe global).

La communication au sein d'un groupe se fait au moyen d'une adresse multicast, appelée adresse multicast locale. Chaque noeud possède une et une seule adresse multicast locale. Cette adresse identifie "son" groupe. La communication entre les groupes se fait aussi au moyen d'une adresse multicast (adresse globale). Toutes les machines reçoivent les messages émis sur cette adresse. Ces deux adresses (adresses multicast locale et globale) sont les seules données qui doivent être récupérées après une faute. Elles sont stockées sur le support stable.

Au sein d'un groupe, les machines élisent un coordinateur. Lors de la période d'élection, les messages restent cantonnés au groupe local. Le coordinateur change au cours du temps en fonction des machines présentes dans le groupe. Chaque noeud possède une image de son groupe. C'est le membership local. Les coordinateurs s'occupent d'émettre régulièrement le membership de leur groupe local sur l'adresse globale. De cette manière, tous les membres de tous les groupes connaissent le membership global (qui n'est autre que la réunion des memberships locaux).

3.1. Anneau de second niveau

Les machines sont réparties en plusieurs groupes locaux. Le nombre de groupes et la répartition des machines entre les groupes est un point délicat de l'algorithme (cf. section 5). Une machine peut appartenir à n'importe quel groupe mais ne doit pas en changer après son démarrage. Nous considérons (pour des raisons de bonne gestion des liens de communication) que la taille et la forme des groupes suivent à peu près la structuration du réseau (par exemple par sous-masque Ethernet [18]).

Au sein d'un groupe local, l'élection du coordinateur se fait à partir de l'identifiant des noeuds (adresse IP) : le membre de plus petit identifiant devient le coordinateur.

La surveillance des stations se fait de manière cyclique : les noeuds sont classés par ordre décroissant d'identifiant. Un noeud n surveille le noeud qui le précède, c'est à dire le noeud qui a le plus grand identifiant inférieur à n. Le coordinateur surveille le noeud de plus grand identifiant. La détection de la faute d'un membre est active : chaque membre envoie un message unicast vérifiant la validité de son prédecesseur et si il ne répond pas au bout d'un certain temps, il est déclaré fautif auprès du coordinateur. C'est le coordinateur qui s'occupe de diffuser l'information sur les adresses de multicast locale et globale. À noter que seul le coordinateur peut envoyer un message sur l'adresse multicast globale.

3.2. Anneau de premier niveau

Chaque groupe est identifiable grâce à son adresse multicast locale. Le nombre de groupes peut varier dynamiquement au cours du temps. L'ajout d'un groupe se fait en ajoutant des noeuds qui ont une nouvelle adresse de multicast locale. Les groupes sont classés par adresses de multicasts croissantes.

La détection de faute d'un groupe (plus aucun membre n'est atteignable) est plus complexe que pour l'anneau de deuxième niveau. En effet, on ne peut vérifier de manière active l'existence d'un groupe. En revanche, on sait que chaque coordinateur de groupe doit émettre un membership régulièrement. La disparition d'un groupe est donc détectée de manière passive. Si un coordinateur d'un groupe ne reçoit pas de membership d'un autre groupe pendant une période trop longue, c'est que le groupe est peut-être en faute. Une phase de détection de faute de groupe est alors démarrée (cf. section 4.4).

4. États et messages du protocole

Nous décrivons dans cette section notre protocole sous la forme d'une machine à états. Chaque noeud peut se trouver à un instant donné dans un des quatre états suivants :
- ADD : il essaie de s'insérer dans un groupe.
- LEAF : il est inséré dans un groupe (et n'est pas coordinateur).
- LOCAL : il souhaite devenir coordinateur.
- GLOBAL : il est coordinateur du groupe.

Les messages utilisés par notre protocole sont :
- Add(NodeId) : émis sur l'adresse multicast locale pour s'insérer dans un groupe.
- Rm(NodeId) : émis par un membre au coordinateur lorsqu'il détecte la perte de son prédécesseur.
- Ping(Ping-Id) : émis par un membre à son prédécesseur pour vérifier son existence.
- Ack(Ping-Id) : réponse à un ping.
- Ms(Membership) : envoi du membership, soit en état *local* sur l'adresse locale (*ms* local), soit en état *global* sur l'adresse globale (*ms* global).
- Knownmbs(GroupId[]) : envoi par le coordinateur des identificateurs de groupes connus sur l'adresse multicast locale.
- Reinit() : émis par un membre sur l'adresse locale pour remettre à zéro les tables de membership de tous les membres du groupe.

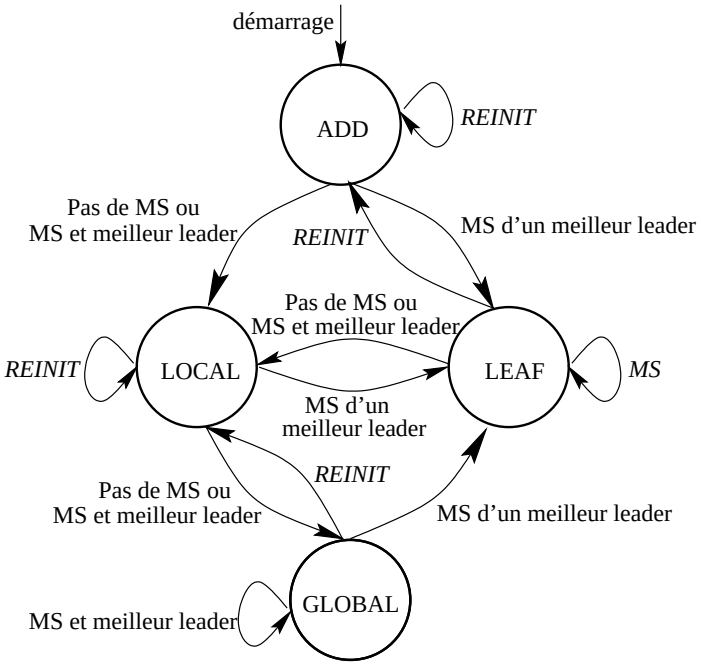

FIG. 1 – États et changements d'état

La figure 1 présente les états, et les changements d'états survenant à la réception d'un message de son groupe local (seuls les messages *ms* et *reinit* font changer d'état). Une machine qui souhaite s'insérer dans le membership commence dans l'état *add*. Ensuite, elle passe en *local* ou en *leaf*, selon qu'elle est ou non potentiellement le prochain coordinateur. Dans l'état *local*, après la période d'élection, une machine passe dans l'état *global* (devient le nouveau coordinateur) ou dans l'état *leaf* si un meilleur coordinateur existe. À noter les changements d'états dus au message *reinit*, qui force la réinitialisation du groupe locale, lorsque le temps de détection des membres fautifs dans le groupe devient trop long face au temps d'initialisation (voir la section 4.3).

Chaque noeud maintient localement (sur support non stable) les connaissances suivantes : la table de membership local, la liste de toutes les adresses de multicast des groupes actifs, la table de membership associée à chaque groupe, ainsi que les temporisateurs indiquant quand envoyer un message *add* ou *ping*, ou passer dans l'état *local* ou *global*. Le noeud dans l'état *local* a trois temporisateurs supplémentaires indiquant quand envoyer le membership, détecter la faute de son groupe successeur, détecter la faute de tous les autres groupes. Chaque temporisateur a une variable associée indiquant le nombre de réémissions en cours.
Les différents états sont décrits dans les sections suivantes.

4.1. État *add*

Lorsqu'une station veut se joindre à un groupe, elle entre dans l'état *add*. Elle ne peut être dans cet état qu'au démarrage ou lorsqu'elle reçoit un membership local où elle ne figure pas. À l'entrée dans cet état, la station attend un temps aléatoire appelé *cold-timeout*. Cette attente répartit dans le temps l'envoi des messages *add* lors du démarrage à froid. Ensuite, le noeud envoie un message *add* sur l'adresse locale avec son identifiant, puis il se met en attente pendant un temps *init-timeout*. Pendant cette période, si il reçoit un message *ms* local dans lequel il figure, il change d'état. S'il est meilleur coordinateur que le coordinateur actuel (il a la plus petite adresse IP du membership local), il passe dans l'état *local*. Sinon il passe dans l'état *leaf*.
Si il ne reçoit pas de message *ms* local, il recommence *init-retry* fois l'envoi du message *add* et l'attente. Après expiration de ces délais sans réception de message *ms* local, il passe en état *local*.

4.2. État *local*

Un noeud passe dans l'état *local*, soit après le démarrage (cf. ci-dessus), soit lorsque sa table de membership lui indique qu'il est le meilleur coordinateur. Dès qu'il entre dans cet état, il émet *changeleader-number* fois son membership sur l'adresse multicast locale. Les autres membres peuvent donc savoir s'ils sont meilleurs coordinateurs et, dans l'affirmative, passer aussi en état *local*.
Après cette émission, le noeud attend *changeleader-timeout* secondes avant de passer en état *global*. Si pendant ce temps il reçoit un message *ms* local d'un noeud qui est meilleur coordinateur, il repasse en état *leaf*. Le but de cette attente est d'élire un coordinateur localement, sans émettre de message sur l'adresse multicast globale.
Pour éviter des basculements successifs entre les états *local* et *leaf*, nous avons introduit un temporisateur de faible valeur (*leaf2local-delay*). En effet, si le noeud est passé en état *local* et qu'il repasse en état *leaf*, c'est qu'il existe un meilleur coordinateur.

4.3. État *leaf*

Dans l'état *leaf* le noeud est membre du groupe et il existe un meilleur coordinateur. Le noeud s'occupe de vérifier la présence du noeud précédent à l'aide de messages *ping*. Cette vérification est standard : on envoie un message *ping* au noeud prédécesseur et on attend sa réponse *ack* avant expiration du timeout *ping-timeout*. On considère qu'une station est fautive si elle ne répond pas *ping-retry* fois consécutivement aux messages *ping*. Lorsqu'une station est détectée fautive, le détecteur envoie un message *rm* au coordinateur avec l'identifiant de la station perdue, supprime la station perdue de son membership et commence à envoyer des *ping* à la station précédente. Si il vérifiait la présence du coordinateur et qu'il s'aperçoit de sa disparition, il passe directement en état *local* (c'est d'ailleurs pour cette raison qu'il vérifie le noeud précédent et non le noeud suivant).
Quand une station répond à un message *ping*, l'émetteur attend *ping-delay* avant de recommencer à lui envoyer un message *ping* de manière à ne pas saturer le réseau avec de tels messages.
Cette manière de détecter les fautes pose un problème classique en cas de partition ou lorsque beaucoup de noeuds sont fautifs simultanément. En effet, le temps de convergence devient très long (dans le pire des cas : nombre de noeuds fautifs * *ping-timeout* * *ping-retry*). Par exemple, dans un groupe de 100 noeuds, si 99 noeuds deviennent fautifs le noeud restant (avec *ping-timeout* = 1 s, *ping-retry* = 3) aura un membership correct après 5 minutes. Nous avons cherché à borner le délai maximum de convergence du protocole dans le pire des cas. Considérons les deux cas de partitions : le coordinateur est fautif ou non.
Le premier cas est détecté avec le temporisateur. Si au bout d'un temps *leaf-timeout*, un noeud dans

l'état *leaf* n'a pas reçu de message *ms*, il passe en état *local* après avoir supprimé la partie basse de son membership (tous les noeuds de moins bon identifiant que lui). De cette façon, en l'absence de nouvelles fautes, il faudra de l'ordre de *leaf-timeout* + *changeleader-timeout* secondes avant d'obtenir à nouveau un membership correct (soit, en prenant *leaf-timeout* = 8,75 s, *changeleader-timeout* = 3,75 s, de l'ordre de 13 secondes).
Le second cas est détecté au moyen d'une limite sur le nombre de noeuds perdus : *max-losed*. Si un noeud s'aperçoit qu'il a détecté *max-losed* fautes successives, il envoie *reinit-number* messages *reinit* sur l'adresse locale. Un noeud qui reçoit ce message va :
– ne rien faire si il est en état *add* ou *local*,
– vider sa table de membership sinon, puis passer en *add* si il était en *leaf*, ou en *local* si il était en *global*.
De cette façon, en l'absence de nouvelles fautes, le temps de convergence du membership est de l'ordre de *changeleader-timeout* + *ping-timeout*ping-retry*max-losed* (soit, avec les mêmes valeurs que précedemments et *max-losed* = 3, de l'ordre de 13 secondes).

4.4. État *global*

Un noeud dans l'état *global* est le coordinateur du groupe. Il s'occupe de gérer :
– la surveillance du noeud de plus grand identifiant (il ferme l'anneau),
– l'envoi régulier du membership sur l'adresse globale et la centralisation des changements de membership,
– la surveillance du groupe suivant,
– la surveillance de tous les autres groupes ;
Lorsqu'un noeud entre dans l'état *global*, il envoie *knownmbs-number* messages *knownmbs* au groupe local (puis régulièrement avec la période *max-multicast-timer*). Ce message contient tous les identifiants des groupes que le coordinateur suppose actifs. Cela met ainsi d'accord les membres du groupe local sur l'existence des autres groupes (toutes les machines appartenant à des groupes absents de la liste envoyée par le coordinateur sont supprimées).
Lorsqu'un noeud entre dans l'état *global*, ou lorsque sa liste de membership change (réception d'un message *add*, *rm*, perte du noeud de plus grand identifiant), il place le temporisateur *multicast-timer* à *min-multicast-timer*. En l'absence de changement, dès que *multicast-timer* est écoulé, le noeud émet à nouveau son membership en global et multiplie *multicast-timer* par *multicast-factor*.
Comme indiqué précédemment, la détection de faute de groupe est passive, c'est pourquoi il faut que le membership soit émis avec une période maximum, même en période de stabilité. La valeur de *multicast-timer* est donc bornée à *max-multicast-timer*. Lorsque le membership change, le coordinateur émet le nouveau membership après épuisement du minimum entre *min-multicast-timer* et l'ancien temps restant après la prochaine émission de membership. Ce délai évite d'émettre n membership lorsque n changements de membership sont notifiés au coordinateur dans un faible intervalle de temps. Cette variation de la période d'envoi du membership permet de combiner une réaction rapide dans les périodes d'instabilité avec une faible surcharge dans les périodes où le membership est stable.
La dernière fonction du coordinateur est de vérifier périodiquement la présence de son groupe sucesseur, ainsi que la présence de tous les autres groupes. Il compare localement si un membership a bien été envoyé, tous les *firstlevel-period* pour son groupe successeur, et tous les *firstlevel-timeout* pour les autres groupes. L'idée est que la détection de la faute d'un groupe soit faite de manière privilégiée par le coordinateur du groupe prédecesseur, mais pour éviter un délai trop long (instabilité dans l'élection du coordinateur chargé de détecter la faute) ou la persistance de groupes fantômes, tous les coordinateurs surveillent passivement tous les autres groupes.
Lorsque un (ou plusieurs) groupe est suspecté fautif par un coordinateur, il émet un message *ms* vide sur l'adresse de multicast **locale** du groupe suspecté. Ce message a pour effet de faire passer tous les noeuds de ce groupe dans l'état *local*, accélèrant ainsi l'élection du coordinateur du groupe suspecté fautif. Le coordinateur attend ensuite *groupfault-timeout* avant de recommencer une vérification. Il recommence *groupfault-retry* fois. Si après ce temps il n'a toujours reçu aucun message *ms* du groupe fautif, il envoie un message *ms* vide à la place du (ou des) groupe(s) fautif(s) sur l'adresse multicast **globale** (il prend donc la place du coordinateur du groupe fautif). C'est le seul cas où un coordinateur envoie un membership qui n'est pas celui de son groupe local. À la réception de ce message, chaque noeud supprime toutes les machines du groupe de sa liste de membership.

5. Évaluation du protocole

L'évaluation d'un protocole censé fonctionner sur plus de 1000 machines est un problème délicat. La simulation du protocole nécessiterait trop de mémoire, de puissance de calcul et de temps. Le seul moyen à notre disposition est de réaliser effectivement le protocole sur une grappe de 1000 machines. Nous n'avons pas accès à une grappe de cette taille mais à une grappe de 70 machines (Institut de Programmation de l'Université Paris 6), utilisée principalement par les étudiants de deuxième cycle de l'Université Paris 6. Sur ces 70 machines, nous avons réalisé un gestionnaire de threads répartis qui permet de simuler plusieurs machines sur une seule. Faute de place, ce gestionnaire n'est pas décrit ici. Signalons cependant que le gestionnaire de threads réparti se sert lui-même du protocole de membership (sur les 70 machines) pour s'initialiser et surveiller le bon déroulement de l'évaluation des performances du protocole de membership (jusqu'à 1024 machines).

Nous avons effectué trois types de tests :

– les tests de démarrage à froid : démarrage simultané de toutes les machines ;
– les tests de perte de membres, avec ou sans perte du coordinateur ;
– les tests de perte de groupes.

Tous les tests présentés ont été réalisés au moins 5 fois. Les valeurs des différents paramètres du protocole figurent en annexe. À noter que le réseau des 70 machines n'est physiquement composé que d'un seul lien Ethernet (10 Mb), ce qui est une condition défavorable pour l'évaluation de notre protocole (possibilité de congestion du réseau en cas d'élections en parallèle dans plusieurs groupes). La plupart des pertes de messages ont lieu dans les piles IP des stations (la bande passante utilisée par le protocole est assez faible). La taille de ces piles a été laissé à 64 Ko (ce qui représente environ 500 tables de membership locaux de 128 membres).

Dans la suite, un groupe témoin signifie un groupe ayant un seul membre.

5.1. Démarrage à froid

Nous avons réalisé deux séries de tests de démarrage à froid (en absence de fautes) avec 256 membres et avec 1024 membres. Dans chaque série de tests, nous avons fait varier le nombre groupes. Le nombre de membres par groupe est réparti équitablement.

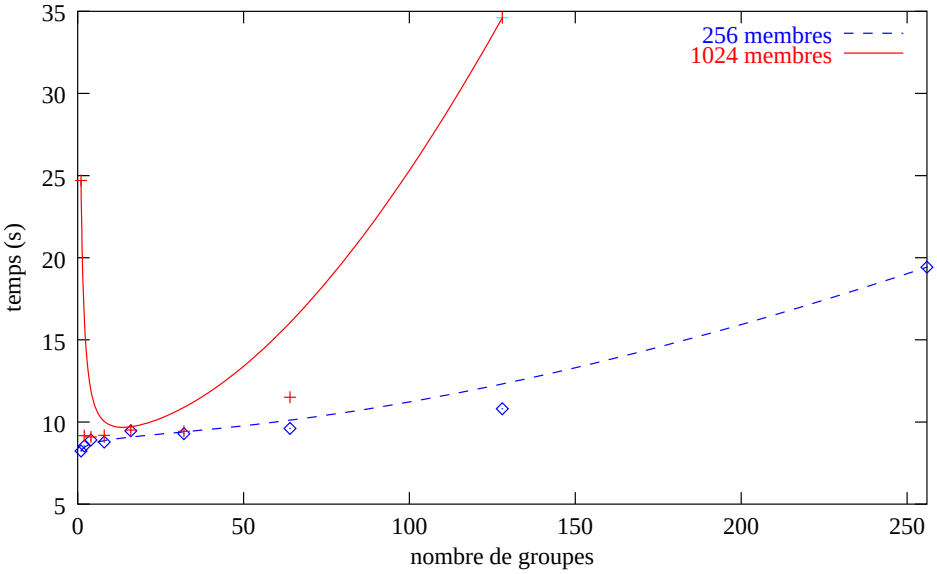

Fig. 2 – Démarrage à froid

L'interpolation présentée figure 2 a été réalisée au moyen de la méthode de Bézier. On constate une

grande différence entre les deux courbes lorsqu'il n'y a qu'un seul groupe (dans ce cas on est dans le cas classique avec un seul coordinateur global). Avec 256 membres, tous les *add* sont pris en compte en un temps record (2 secondes), mais avec 1024 membres il faut 24 secondes. Cette différence s'explique par la saturation des piles IP. Comme présupposé, le calibrage du nombre de groupes en fonction du nombre de machines est un facteur important. On constate que pour 256 machines les meilleurs résultats sont obtenus avec 1 ou 2 groupes et avec 2 à 16 groupes pour 1024 machines (soit entre 64 et 512 machines par groupe). Ces résultats laissent supposer que le démarrage à froid pourrait être efficace jusqu'à 16 groupes de 256 machines (soit plus de 4000 machines), mais ce test n'a pu être réalisé faute de ressources.

5.2. Perte de membre

Deux séries de tests ont été réalisées. La première avec 128 machines réparties sur un seul groupe avec un groupe témoin. La seconde avec 1024 machines réparties sur 4 groupes. Dans ces séries de tests, on s'intéresse au temps que met le protocole à converger au sein d'un groupe. Pour les tests avec 1024 machines, les fautes ont lieu dans le même groupe (indiqué 256 membres dans la suite). Les autres groupes sont spectateurs.

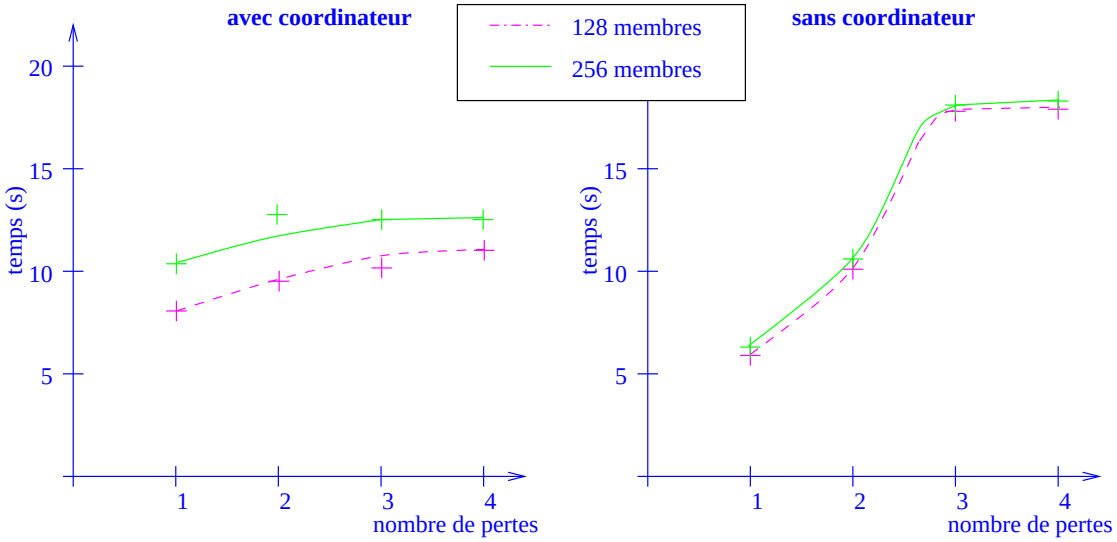

FIG. 3 – Petites pertes avec et sans coordinateur

Nous évaluons en premier les résultats lorsque de une à 4 fautes de noeuds successifs dans l'anneau surviennent simultanément. Lorsque le coordinateur est fautif (figure 3, avec coordinateur), le temps de convergence est à peu près le même, entre 7,5 et 10,5 secondes pour 128 machines, et environ 2 secondes supplémentaires pour 256 machines. Ce temps correspond au temps de la détection de la faute du coordinateur plus le temps de l'élection du nouveau coordinateur. Lorsque le coordinateur n'est pas fautif (figure 3, sans coordinateur), les résultats ne sont pas stables. Un seuil apparaît à partir de la faute de 3 noeuds, reflétant en fait la valeur de la constante *max-loosed*. En effet, lorsque cette valeur est atteinte, le site détecteur force la réinitialisation du groupe en émettant le message *reinit*.
Nous évaluons ensuite les résultats lorsque la moitié du groupe disparaît simultanément. La figure 4 présente ces résultats pour les cas où le coordinateur est fautif ou non. Nous observons que la taille du groupe ne change pratiquement pas le résultat. Pour le cas sans coordinateur, le temps comprend le délai *leaf-timeout* plus l'envoi de 64 ou de 128 messages *add* (la moitié des groupes). Pour le cas avec coordinateur, on dépasse toujours *max-losed* pertes de noeuds. En revanche, les temps de convergence sont plus courts que ceux présentés dans la figure 3, puisqu'il y a moins de machines actives et donc moins de passages dans l'état *local*.

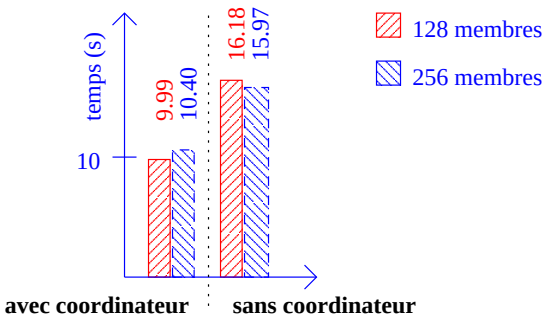

FIG. 4 – Perte d'un demi groupe

5.3. Perte de groupes

Deux tests ont été réalisés : le premier avec 16 groupes de 1 membre et le second avec 64 groupes de 16 membres. Les tests effectués sont soit des pertes de groupes successifs, soit des cribles. Dans un test de crible n, n machines successives sont fautives sur $n+1$. On choisit de supprimer n membres successifs, de garder le membre suivant et de recommencer.

FIG. 5 – Perte de groupes

16 groupes							
test	résultats					moyenne	écart
1	22.63	24.77	21.46	23.42	22.81	23.02	2.42
2	28.51	24.66	21.92	24.68	21.21	24.20	5.76
3	25.22	24.98	28.71	21.33	27.21	25.49	5.56
4	23.56	26.37	28.59	29.81	24.16	26.50	5.43
5	28.55	26.13	28.25	27.71	26.73	27.47	2.04
crible 1	21.89	23.01	21.60	23.40	24.80	22.94	2.56
crible 2	27.65	28.28	27.22	25.62	27.26	27.21	1.90
64 groupes							
test	résultats					moyenne	écart
1	21.66	21.50	21.13	23.05	23.79	22.23	2.27
2	23.05	21.06	22.95	24.23	24.83	23.22	2.89
4	21.83	26.90	28.52	28.57	24.91	26.15	5.68
16	28.25	26.74	28.72	28.74	27.72	28.03	1.67

Les résultats (figure 5) sont environ les mêmes avec 16 ou 64 groupes, entre 21 et 29 secondes. Ces temps correspondent bien aux valeurs des différents paramètres qui composent la détection de faute de groupe (cf. section 4.4).

6. Comparaisons avec les travaux du domaine

Selon la classification de Cristian et Schmuck [7], notre protocole appartient à la catégorie des protocoles de membership avec partitions dans un système réparti asynchrone temporisé. Le protocole à trois reprises décrit dans leur article a des similarités avec le notre. Dans ce protocole, un coordinateur global diffuse une proposition de membership à tous les noeuds. Tous les noeuds recevant la proposition doivent répondre au coordinateur. Enfin, le coordinateur diffuse le nouveau membership à tous les membres. Ce protocole est particulièrement efficace lorsque le nombre de noeuds reste faible. Dans le cas contraire, le

réseau de communication devient un goulot d'étranglement, et le nombre de messages perdus augmente, pouvant conduire à un membership souvent incorrect. En distribuant le coordinateur en un ensemble de coordinateurs locaux (anneau de premier niveau), nous évitons une telle contention du réseau.
Un détecteur de faute extensible basé sur une diffusion endémique (gossip) a été proposé par van Rennesse et al. [18]. L'extensibilité repose sur la construction des adresses IP, structurées en nom de machine, sous-réseau et domaine. Les messages de gossip sont principalement envoyés à l'intérieur de sous-réseaux, avec peu de gossips entre les sous-réseaux, et encore moins entre les domaines. Cette structure évite elle aussi la contention réseau. Les deux premiers niveaux de cet arbre sont assez similaires à notre arbre à deux niveaux (on peut considérer par exemple que les groupes locaux sont structurés en fonction des sous-réseaux physiques). En pratique, une structure à deux niveaux est suffisante pour gérer le membership de plus de 1000 noeuds. La principale différence concerne la manière de disséminer les gossips pour transmettre les informations de membership. Le protocole de membership lorsqu'il y a une seule faute dans une grappe met un nombre de secondes pour converger proportionel au nombre de noeuds dans la grappe (soit respectivement de l'ordre de 4 et 17 minutes pour 256 et 1024 noeuds). De plus, l'échange réduit d'information entre les sous-réseaux et les domaines rend ce protocole particulièrement lent lorsqu'il y a un nombre important de fautes ou dans le cas du démarrage à froid. Les auteurs rajoutent un mécanisme de diffusion pour prendre en compte ces cas, mais la perte des messages de diffusion n'est pas mentionnée. À noter que ce protocole semble converger avec des hypothèses moins restrictives que les nôtres (qui sont déjà assez faibles), comme la symétrie dans les communications ou l'atteignabilité équivalente en unicast et en multicast. Nguilla et Haddad proposent un protocole de consensus hiérarchique [9]. En dehors du fait que les machines fautives sont exclues définitivement, il est difficile de se comparer avec ce protocole, étant basé sur le modèle des détecteurs de fautes non fiables [6] (ce qui sortirait du cadre de ce papier).
Certains travaux considèrent que le membership est statique, signifiant qu'il n'y a pas de faute, partition ou ajout de machine sans arrêter le système, comme dans [13]. Le membership dans le système Star utilise un unique anneau pour la détection des fautes et un protocole similaire au protocole en 3 reprises décrit précédemment pour propager les memberships [17]. Un autre exemple de protocole de membership pour grappe est l'échange périodique de messages de présence entre tous les noeuds du système, sans aucune structure particulière [12]. Notre protocole combine ces différentes approches pour vérifier l'existence des groupes et la vivacité des membres d'un groupe (approche active dans un groupe et approche passive pour détecter la faute des groupes ainsi que la faute du coordinateur).

7. Conclusion

Dans cet article nous avons présenté et évalué un protocole de membership (liste des machines actives et détection des machines fautives ou inatteignables suite à une partition du réseau) hautement extensible pour une grappe de noeuds. Il est basé sur une structuration en deux niveaux d'anneaux de multicast. Les anneaux de deuxième niveau regroupent un sous-ensemble des machines, et l'anneau de premier niveau est formé dynamiquement par l'ensemble des coordinateurs des anneaux de deuxième niveau. Le but principal était de proposer un algorithme utilisable pour un serveur basé sur une grappe de noeuds, en supprimant le goulot d'étranglement constitué par un unique coordinateur. Un important travail de réalisation et de mesures de performances a donc été réalisé pour évaluer les conditions d'utilisation de notre protocole. Les mesures effectuées sur 70 machines simulant une grappe jusqu'à 1024 noeuds semblent prometteuses. Le temps de convergence du protocole est inférieur respectivement à 30 secondes dans le cas de partition du réseau, 20 secondes pour la détection de noeuds fautifs et une dizaine de secondes pour le démarrage à froid. Ces résultats suggèrent que notre protocole pourrait gérer efficacement une grappe de plusieurs milliers de machines.
Les perspectives de ces travaux sont multiples. En premier, la validité de ce protocole doit être prouvée (une démonstration formelle est en cours de réalisation). Ensuite, les résultats obtenus sont à notre avis assez pessimistes à cause du réseau constitué d'un seul brin Ethernet (10 MB), ce qui limite l'impact du multicast, notamment dans le cas du démarrage à froid (ainsi que dans le cas de la perte de groupes). Avec 256 noeuds le nombre de groupe optimal est de un ou deux. Avec un seul groupe on revient dans le cas classique où il y a un unique coordinateur, ce qui enlève tout intérêt à notre protocole. Il faut atteindre les 1024 noeuds pour que les meilleurs résultats soient obtenus avec plus d'un seul groupe (2 à 32 groupes). La séparation du réseau en sous-réseaux distincts, où chaque groupe est dans son sous-

réseau, devrait avoir un effet multiplicateur sur les performances (réduction du temps de convergence et, en parallèle, augmentation du nombre de noeuds). D'autres mesures sont donc à réaliser dans ce genre d'environnement, ainsi que sur des vrais grappes de plusieurs centaines de noeuds. Ces mesures permettront également d'optimiser la valeur des différents paramètres constituant le protocole. Enfin, le simulateur de plus de 1000 machines réparti sur 70 machines réelles pourrait lui-aussi être amélioré, par exemple en simulant des réseaux non homogènes pour évaluer d'autres protocoles ou serveurs hautement extensibles sur des multi-réseaux.

Remerciements

Les auteurs tiennent à remercier : l'équipe de Porcupine à l'Université de Washington à Seattle au moment où ce protocole a été conçu : Brian Bershad, Henry Levy, Eric Hoffman, Yasushi Saito ; Les ingénieurs systèmes de l'Institut de Programmation de l'Université Paris 6 ; André Schipper de l'EPFL ; Le projet vérification de systèmes répartis du thème Systèmes Répartis et Coopératifs au LIP6.

Annexe

Constantes et temporisateurs utilisés au cours des tests. Le nombre de réémissions de message a été fixé à 3 dans tous les cas.

Initialisation		
max-cold-timeout	maximum de *cold-timeout*	$2.5s$
cold-timeout	temps aléatoire au démarrage	de 0 à $2.5s$
init-timeout	temps entre deux messages *add*	$0.42s$
init-retry	nombre de messages *add*	3
Gestion des ping		
ping-timeout	temps maximum de réponse à un *ping*	$1.25s$
ping-retry	nombre de tentatives de *ping*	3
ping-delay	délai entre deux *ping*	$2.5s$
max-losed	nombre de pertes avant *reinit*	3
Passage en *global*		
changeleader-timeout	temps pour passer de *local* à *global*	$3.75s$
changeleader-number	nombre de messages *ms* en *local*	3
Réinitialisation et *groupes connus*		
reinit-number	nombre d'émissions de *reinit*	3
knownmbs-number	nombre d'émissions de *knownmbs*	3
Surveillance de groupes		
firstlevel-period	temps avant de vérifier le groupe suivant	$17.5s$
firstlevel-timeout	temps avant de considérer un groupe inatteignable	$28.75s$
groupfault-timeout	temps avant de réémettre un message *ms* vide	$2.5s$
groupfault-retry	nombre de tentatives de *ms* vide	3
firstlevel-number	nombre d'émission de *ms* vide	3
Passage de *leaf* à *local*		
leaf-timeout	temps pour passer de *leaf* à *local*	$8.75s$
leaf2local-delay	temps min pour repasser de *leaf* à *local*	$0.2s$
Émission du *ms* en *global*		
min-multicast-timer	temps min entre deux émissions	$0.31s$
max-multicast-timer	temps max entre deux émissions	$5s$
multicast-timer	temps actuel entre deux émissions	de 0.31 à $5s$
multicast-factor	facteur pour augmenter le *multicast-timer*	4

Bibliographie

1. Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions on Computer Systems*, 13(4), Nov. 1995.

2. T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1) :41–79, Feb. 1996.
3. *High Performance Cluster Computing*. R. Buya ed. Prentice-Hall, 1999.
4. K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, California, 1994.
5. T. Chandra, V. Hadzilacos, S. Toueg and B. Charron-Bost. On the Impossibility of Group Membership in Asynchronous Distributed Systems. In *Proc. 15th ACM Syposium on Principles of Distributed Computing*, Philadelphia, USA, pp. 322–330, May 1996.
6. T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) :225–267, 1996.
7. F. Cristian and F. Schmuck. Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems. *UCSD Technical Report*, CSE95-428, University of California, San Diego, 1995.
8. F. Cristian. Group, Majority, and Strict agreement in Timed Asynchronous Distributed Systems. In *Proc. 16th Fault Tolerant Computing*, Sendai, Japan, June 1996.
9. F. Nguilla and S. Haddad. Reaching Agreement in Hierarchical Groups. In *12th Intl. Conf. on Parallel and Distributed Computing Systems*, Fort Lauderdale, USA, Aug. 1999.
10. M. Fischer, N. Lynch and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2) :374–382, Apr. 1985.
11. A. Fox, S. Gribble, Y. Chawathe, E. Brewer and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.
12. E. Lee and C. Thekkath. Petal : Distributed Virtual Disks. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 84–92, Oct. 1996.
13. V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Oct. 1998.
14. D. Powell. Failure mode assumptions and assumption coverage. In *Proc. of 22th Intl. Symp. on Fault-Tolerant Computing*, pp. 386–395, Boston, USA, Jul. 92.
15. Y. Saito, B. Bershad, H. Levy, E. Hoffman and B. Folliot. The Porcupine Scalable Mail Server. In *Proc. of 8th ACM SIGOPS European Workshop*, Sintra, Portugal, pp. 48–57, Sept. 1998.
16. Y. Saito, B. Bershad and H. Levy. Manageability, availability and performance in Porcupine : a highly scalable, cluster-based mail service. In *Proc. of 17th ACM Symposium on Operating Systems Principles*, Charleston, USA, pp. 1–15, Dec. 1999.
17. P. Sens and B. Folliot. Performance Evaluation of Fault Tolerance for Parallel Applications in Networked Environments. In *Proc. of 26th Intl. Conf. on Parallel Processing*, Bloomingdale, Illinois, Aug. 1997.
18. R. van Renesse, Y. Minsky and M. Hayden. A Gossip-Style Failure Detection Service. In *Proc. of Middleware'98*, The Lake District, England, Sept. 1998.