Algorithme de Naimi-Tréhel et tolérance aux pannes

Raimana BOUISSOU, Tom GIMENEZ, Léo RIZZON

Ici, j'écris à la suite, sans mise en page, sans fichier séparé. Je le ferai ensuite.

1 Introduction

Dans le cadre de notre unité d'enseignement intitulée Algorithmie Distribuée nous avons réalisé un projet qui consiste en l'implémentation d'une amélioration du modèle de tolérance de l'algorithme de Naimi-Tréhel .

Nous nous sommes basés sur un article co-écrit par Julien Sopena, Luciana Arantes, et Pierre Sens qui propose un algorithme d'exclusion mutuelle, tolérant aux fautes. Basé sur l'algorithme de Naimi-Tréhel , il apporte une nette amélioration du coût de recouvrement en terme de messages.

2 Objectifs

L'objectif de ce projet était, d'une part, la compréhension poussée d'un algorithme d'exclusion mutuelle et l'étude d'un système tolérant aux pannes. Toutefois, il s'agissait assimiler les problèmes et défauts qu'il comportait pour bien appréhender l'amélioration proposée.

D'autre part, l'implémentation de ces modèles pour appliquer toute la théorie à un exemple concret bien que futile.

Notons également que l'étude s'est faite à partir d'un article en anglais, comme la majorité des articles scientifiques, ce qui constitue une expérience intéressante.

L'implémentation consiste donc en l'application de l'algorithme de Naimi-Tréhel à un certain nombre de sites, en réseaux, qui cherchent à acceder à une même ressource alors qu'il n'y qu'un accès possible à la fois.

Algorithm 1 Initialisation

Require: élection d'un site pour génèrer le jeton

1: $last$ = identifiant$_{siteElu}$
2: $next = \emptyset$

Algorithm 2 Envoi par i de <TOKEN>

1: $next = \emptyset$

3 Les algorithmes étudiés

Cette partie nous servira à apporter une explication, avec nos propres mots et exemples, des algorithmes que nous avons étudiés.

4 Algorithme de Naimi-Tréhel

L'algorithme de Naimi-Tréhel est un modèle permettant à plusieurs sites de fonctionner autour d'une ressource. A un moment donné on veut être assuré qu'un seul site sera en Section Critique. C'est un algorithme d'exclusion mutuelle fonctionnant sur le principe du jeton.

Fonctionnement de l'algorithme Le fonctionnement général repose sur la construction dynamique de deux arbres. Aucun site n'a de vue globale de ces arbres et ne connait alors que son *last* , dernier possesseur connu du jeton par ce site et son *next* à qui ce site doit passé le jeton une fois sorti de S.C.

Ainsi, le cas écheant, un site sait vers qui demandé le jeton et vers qui le transmettre une fois terminé.

A l'initialisation un site possède le jeton et tous les autres ont leur *last* qui pointe vers lui. On peut arriver à cet état par un système d'élection par exemple mais nous avons choisi de le faire à la main dans notre configuration, plus petite.

Lorsqu'un site fait une demande de jeton, il l'envoie systématiquement à son *last* . La demande va alors transiter dans l'arbre des *last* jusqu'à arriver à la racine. A chaque fois, le site relai va faire pointer son *last* sur le site demandeur (en effet, c'est celui-çi qui aura le jeton en dernier). La racine va alors faire pointer son *next* vers le site demandeur.

Notons que pour éviter une corruption de l'arbre on empêche le site possédant déja le jeton de faire une nouvelle demande. En effet on risquerait un écrasement de l'ancien *next* si une nouvelle requête venait au possesseur du jeton.

Implémentation Ecrire l'algo en pseudo langage

Algorithm 3 Reception par i de <TOKEN_REQUEST, j>

```
1: if last_i ≠ i then
2:    envoyer(<TOKEN_REQUEST,j>) à last
3:    last = j
4: else
5:    next = j
6:    last = j
7:    Attendre fin S.C.
8:    envoyer(<TOKEN>) à next
9: end if
```

Algorithm 4 Envoi par i de <TOKEN_REQUEST, i>

```
1: envoyer(<TOKEN_REQUEST, i>) à last
2: last = i
```

5 L'extension de Naimi-Tréhel permettant la tolérance aux pannes

Dans sa version la plus simple, l'algorithme de Naimi-Tréhel n'assure pas la tolérance aux pannes. Une extension existe toutefois pour répondre à ce besoin.

Explication du mécanisme Dans des conditions réelles d'utilisation il existe toujours un risque de panne sur un ou plusieurs sites. Les modèles utilisant le principe du jeton posent de plus le problème de la perte de celui-çi. Enfin, dans notre cas, la construction d'arbres virtuels dépourvue de vision globale engendre le risque de perversion complète de l'organisation des sites.

Ici, on utilisera, pour détecter une faute, un timeout. Il s'agit d'un timer que l'on considère comme étant un temps raisonnable d'attente avant de suspecter un problème. Notons que ce timeout est à adapter à la configuration : temps de transfert des messages, nombre de sites, temps de S.C..

Lors de la demande du jeton, un site attendra un certain timeout avant de lancer une procédure de vérification de défaillance. Ainsi il envoie un message spécifique <CONSULT> à tous les autres sites afin de vérifier si son prédecesseur dans la file des *next* (alors inconnu) est opérationnel. Il attend alors à nouveau un certain timeout car si c'est bien son prédecesseur qui est tombé, il ne recevra pas de réponse. Dans le cas d'une réponse, ce site n'a alors plus de questions à se poser et ne changera rien, on déduit en effet, s'il existe bien une panne, que c'est son prédecesseur fera alors le travail nécéssaire.

Dans le cas d'absence de réponse on peut avoir deux cas de figure. En effet, il est possible que ce soit le site possédant le jeton qui a eu une panne. Le jeton est alors perdu et ne peut être régèneré n'importe comment.

Le site suspectant une panne envoie un nouveau message spécfique, <FAILURE>, en broadcast, à l'intention du possesseur du jeton. Si celui-çi répond, le site questionneur refera alors simplement sa demande de jeton pour palier à une éventuelle perte de message lors de la première demande.

Par contre si celui-çi ne répond pas, un réinitialisation globale très coûteuse est effec-

Algorithm 5 Demande de jeton par i

```
1: envoyer(<TOKEN_REQUEST>) et armer le Timer
2: if fin du Timer ∧ pas reçu le TOKEN then
3:    broadcast(<CONSULT>)
4:    if pas de réponse au message <CONSULT> then
5:       broadcast(<FAILURE>)
6:       if réponse du possesseur du jeton then
7:          envoyer(<TOKEN_REQUEST>) au site qui a répondu
8:       else
9:          broadcast(<ELECTION>)
10:         if si aucune réponse ou si je n'ai pas abandonné then
11:            régèneration du jeton
12:         end if
13:      end if
14:   end if
15: end if
```

Algorithm 6 Reception par i de <CONSULT>

```
1: if next_i = i then
2:    envoyer(<T_MON_NEXT>) à emetteur
3: end if
```

tuée. Un message est alors envoyé à tous pour vérifier si il ni y'a pas d'autres sites qui envisagent de créer un nouveau jeton. Si il y a concurrence, le site au plus petit identifiant génère un jeton, unique, mais toute l'organisation est perdue est doit être refaite. Toutes les demandes doivent être effectuées de nouveau.

Implémentation Ecrire l'algo en pseudo langage

Atouts et inconvénients de cette extension Cette extension répond de manière sûre au problème de pannes. Néanmoins dans le cas de la perte du jeton on a affaire à une réinitialisation complète particulièrement peu efficace et coûteuse. Pire encore, on peut arriver à des coûts encore plus élevés dans certain cas où le jeton n'est pas perdu : on constate alors des procédures de recouvrement individuel cooncurentes.

5.1 Explication du mécanisme

5.2 Implémentation

5.3 Atouts et inconvénients de cette extension

5.4 Amélioration de l'aogorithme de tolérance aux pannes

Nous avons vu que, malgré qu'elle parvienne au résultat escompté, l'extension de l'algorithme de Naimi-Tréhel coûte cher. C'est dans une optique d'amélioration de ce système que l'algorithme équitable tolérant aux fautes à été imaginé.

Algorithm 7 Reception par i de <FAILURE>

```
1: if j'ai le jeton then
2:    envoyer(<J_AI_JETON>) à emetteur
3: end if
```

Algorithm 8 Reception par i de <ELECTION>

```
1: if je cherche déja à régènerer le jeton then
2:    if identifiant_i > identidiant_emetteur then
3:       abandon de la procédure de régèneration
4:    end if
5: end if
```

On va chercher à conserver l'ordre initial des requêtes de demande de jeton en récupérant les morceaux non corrompus de la file des *next* . Si la reconstruction n'est pas possible une nouvelle file des *next* et un nouvel arbre des *last* seront créés.

Les différents mécanismes Tout d'abord les sites possèdent désormais plus d'informations que dans l'algorithme précédent. Ils connaissent leur position dans la file des *next* (3e après le possesseur actuel du jeton par exemple) et également un certain nombre de leurs prédecesseurs. On voit une réponse directe aux problèmes posés par l'algorithme précedant. Là où un broadcast était nécessaire pour trouver son prédecesseur avant, aucun message n'est utile ici !

Paradoxalement, on va ajouter un certain nombre de messages pour en économiser si une panne survient.

En effet, un message <COMMIT> sera envoyé en réponse, par la racine de l'arbre des *last* , à chaque <TOKEN REQUEST> qui lui parvient. Il peut ainsi, avec ce <COMMIT> mettre à jour la liste des prédecesseurs du site demandeur, en s'incluant dedans, tout en confirmant la réception de sa demande et en envoyant sa position dans la file des *next* .

Après avoir reçu le message <COMMIT>, le site demandeur va régulièrement tester la validité de son plus proche prédecesseur, assurant une réactivité face à une éventuelle panne.

Lors d'une détection de panne par un site, on distingue trois grands mécanismes répondant à des cas de figure différents : Mécanisme 1 (M1) : Le site a reçu un message <COMMIT> et il y a moins de panne consecutives dans la file des *next* que de prédecesseurs connus par ce site. Mécanisme 2 (M2) : Le site a reçu un message <COMMIT> et il y a plus de panne consecutives dans la file des *next* que de prédecesseurs connus par ce site. Mécanisme 3 (M3) : Le site n'a pas reçu de message <COMMIT>

Détaillons un peu ces mécanismes. Mécanisme M1 : Le site va envoyer à ses prédecesseurs, du plus proche au plus éloigné, un message <ARE_YOU_ALIVE>. On attend une réponse qui sera celle du premier site opérationnel avant le site en question. Ce site, qui répond un message <I_AM_ALIVE> , prendra comme *next* le site questionneur. La file des *next* est ainsi reconstruite en conservant son ordre.

Mécanisme M2 : On n'a pas de réponse au message <ARE_YOU_ALIVE> ici. Tous les sites connus comme prédecesseurs sont défaillants. Le site va alors chercher à se raccorder avec un morceau de la file des *next* valide. Il envoie un message <SEARCH_PREV>, qui contient sa position dans la file, à tout le monde. Seuls les sites étant plus proche du

Algorithm 9 Reception par i de <TOKEN_REQUEST, j> de k

```
1: if last_i ≠ i then
2:    envoyer(<TOKEN_REQUEST, j>) à last
3:    last = j
4:    next = ∅
5:    position_i = ∅
6: else
7:    next = j
8:    last = j
9:    envoyer(<COMMIT, liste_pred_i, position_i+1>) à last
10: end if
```

possesseur du jeton doivent répondre et le site va alors choisir parmis ceux-çi celui qui a la position la pls grande (donc le plus éloigné du possesseur du jeton) pour qu'il devienne son plus proche prédecesseur. Un message <CONNEION> informera le site concerné de la manoeuvre de reconnexion Par contre, si le site ne recoit aucune réponse à sa diffusion de <SEARCH_PREV>, il en déduira qu'il n'a aucun prédecesseur, donc que le jeton est perdu. Il peut alors en régeneré un.

Mécanisme M3 : Ici, donc, un ou plusieurs sites suspectent une panne après une attente trop importante du message <COMMIT>. Le ou les sites concernés n'ont donc ni file des prédecesseurs ni position dans la file des *next* .

Les sites suspectant une panne envoient en broadcast un message <SEARCH_QUEUE,nbSC> pour tenter de retrouver une file potentiellement existante. La variable *nbSC* correspond au nombre de fois où le site émetteur est entré en section critique depuis le début. Une réponse, du type <ACK_SEARCH_QUEUE,*position*,*avoirNext*> avec *position* qui indique la position de l'émetteur par rapport au possesseur du jeton et *avoirNext* qui précise si il a un *next* ou pas, n'est envoyée par un site que s'il a une position dans la file des *next* .

Si jamais plusieurs sites détectent une panne en même temps, plusieurs envois de <SEARCH_QUEUE,nombreSectionCritique> vont se croiser. Si un site, qui a déja commencé une procédure <SEARCH_QUEUE,nbSC> reçoit a son tour le même message en provenance d'un autre site (qui a donc lui aussi détecté une panne) il s'agit de décider qui va mener le recouvrement jusqu'au bout. La priorité va au site ayant le moins d'accès à la section critique depuis le début ou, si égalité, ayant le plus petit identifiant.

Une fois toutes les réponses (messages <ACK_SEARCH_QUEUE,position,avoirNext>) reçues le site va sélectionner le site qui possède la plus grande position afin de se raccrocher au bout de la file des *next* . Si ce site a un *next* alors c'est ce *next* qui est défaillant puisque si il avait répondu, il aurait eu une plus grande position et aurait été sélectionné. Le site va donc directement se connecter au site selectionné (envoi d'un message <CONNEXION>). C'est le dernier site valide du morceau sain de la file des *next* . Si ce site n'a pas de *next* le site demandeur va directement se connecter à sa suite avec une <TOKEN_REQUEST> mais sans passer par l'arbre des *last* habituel. Par contre, si le site demandeur n'a reçu aucune réponse il peut en déduire qu'il n'y aucun site valide dans la file des *next* et il peut donc se considérer comme le premier d'une nouvelle file et regénèrer le jeton.

Implémentation

Algorithm 10 Reception par i de <TOKEN> de j

1: liste_pred$_i$ = $\emptyset$
2: avoir_jeton = vrai
3: position$_i$ = 0

Algorithm 11 Reception par i de <COMMIT, liste_pred, position> de j

1: liste_pred$_i$ = liste_pred
2: position$_i$ = position

5.5 L'algorithme de Naimi-Tréhel sur des exemples

5.6 L'algorithme de Naimi-Tréhel sur des exemples

5.7 L'algorithme de Naimi-Tréhel sur des exemples

Algorithm 12 Reception par i de <ARE_YOU_ALIVE> de j

1: *next* = j
2: envoyer(<I_AM_ALIVE>) à *next*

Algorithm 13 Reception par i de <SEARCH_PREV, position> de j

1: **if** position > position$_i$ **then**
2: envoyer(<POSITION, position$_i$ >) à j
3: **end if**

6 Le developpement

Cette partie apportera quelques précisions sur la manière dont nous avons travaillé,

6.1 Choix techniques

6.1.1 Le langage de programmation

Nous avons choisi le C comme langage de programmation. Plusieurs raisons ont motivé ce choix. D'une part nous avons une solide expérience avec ce langage. Nous trouvons facilement nos marques. De plus, avec le C, nous avons les clés dont nous avions besoin : réseau, traitement d'une ressource, multi-threading.

6.1.2 L'architecture

L'implémentation soulève un certain nombre de questions qui restent en retrait dans la partie théorique. Nous avons alors dû choisir une architecture cohérente avec nos besoins : réseau, travail en Section Critique.

Par exemple, dans la théorie, les sites restent à l'écoute de messages alors qu'ils sont en S.C. ce qui nécessite des actions non bloquantes. Ainsi, le choix du multi-threading s'est revelé idéal. Ainsi, on retrouve un schéma d'éxécution comme présenté figure 1.

Tous les sites sont équivalents, ils sont tous écrit avec le même code.

Le choix du TCP vient principalement de notre expérience en réseau : nous sommes plus à l'aise avec ce protocole. On peut noter toutefois que TCP fournit un certain nombre de services qui ne doivent pas être utilisés dans notre projet. En effet, la détection de pannes est alors aisée avec le système en mode connecté qui surveille la perte de lien entre deux sites. Dans l'interêt de notre projet nous n'avons bien sûr pas utilisé cette fonction.

6.2 L'implémentation

6.2.1 L'implémentation des messages

Tous nos messages sont envoyés comme chaînes de caractère. Par exemple le message <ACK_SEARCH_QUEUE,position,avoirNext> est codé "ACK_SEARCH_QUEUE2,1" et nous effectuons, à la reception, un découpage pour isoler les valeurs. Ici position = 2 et avoirNext = vrai.

Algorithm 14 Reception par i de <CONNEXION> de j

1: $next$ = j
2: envoyer(<COMMIT, liste_pred$_i$, position$_i$+1>) à $next$

Algorithm 15 Reception par i de <SEARCH_QUEUE, nbSC> de j

1: **if** a déja fait un broadcast de <SEARCH_QUEUE, nbSC> **then**
2: procédure envoi TOKEN_REQUEST
3: **else**
4: **if** position$_i \neq \emptyset$ **then**
5: envoyer(<ACK_SEARCH_QUEUE, position$_i$, avoirNext>) à j
6: **end if**
7: **end if**

6.2.2 Des variables globales pour plus de simplicité

L'interêt du projet n'étant pas forcément la création d'une architecture très poussée nous avons voulu faire au plus simple. Réseau local et utilisions de threads suffisait à simuler un système distribué. Comme des procédures devaient ne pas se bloquer (procédure de traitement de message, de timer etc.) nous avons lancé un grand nombre de threads qui ont également l'avantage de se partager des variables globales. Bien qu'une implémentation qu'on pourrait qualifiée de peu élégante, elle nous a énormément nous simplifié la vie.

Algorithm 16 Procédure d'envoi TOKEN_REQUEST par i de j

```
1: envoyer(<TOKEN_REQUEST, i>) à j
2: if j == last then
3:    last = i
4: end if
5: attendre <COMMIT>
6: if i n'a pas reçu de <COMMIT> then
7:    Mécanisme 3
8: else
9:    envoyer(<ARE_YOU_ALIVE>) au dernier prédecesseur de i
10:   if si pas de réponse then
11:      on recommence pour les k-1 prédecesseurs de i
12:   end if
13:   if si une réponse then
14:      //Mécanisme 1
15:      envoyer(<CONNEXION>) à celui qui a répondu
16:      recommencer à l'envoi de <ARE_YOU_ALIVE> précedent
17:   else
18:      //Mécanisme 2
19:      broadcast(<SEARCH_PREV, position_i >)
20:      attendre(<POSITION, position>)
21:      if i a reçu au moins une position then
22:         envoyer(<CONNEXION>) au site qui a la plus grande position
23:      else
24:         avoir_jeton = vrai
25:         liste_pred_i = ∅
26:         position_i = ∅
27:      end if
28:   end if
29: end if
```

7 Conclusion

7.1 Bilan technique

Bilan par rapport aux objectifs. Trucs qui marchent pas, à améliorer...

7.2 Bilan personnel

Ce que ça nous a apporté, interêt, expérience...

Algorithm 17 Mécanisme 3

```
1: broadcast(<SEARCH_QUEUE, nbSC>)
2: attendre les <ACK_SEARCH_QUEUE, position, avoirNext>
3: if i n'a pas eu à abandonner then
4:    if  au moins un site a répondu then
5:       if le site qui a la plus grande position a un next  then
6:          procédure envoi TOKEN_REQUEST au site qui a la plus grande position
7:       else
8:          envoyer(<CONNEXION>) au site qui a la plus grande position
9:       end if
10:   else
11:      avoirJeton = vrai
12:      position_i = 0
13:   end if
14: else
15:   test de la vivacité du plus proche prédecesseur
16:   appel les mécanismes 1 et 2 au besoin (voir Procédure d'envoi TOKEN_REQUEST)
17: end if
```