Algorithmes Distribués

Rapport de Projet 2010

Boris Albar
(M1 Mathématiques & Informatique)

29 avril 2010

1.1 Introduction

L'objectif de ce projet est d'implémenter l'agorithme décrit dans [SABS]. Nous récpitulons les points importants et mécanismes de l'algorithme et nous les illustrons par des schémas et des *logs* du programme.

Nous avons implémenté un sytème de planification d'évènements, ce qui permet de mettre en place des scénarios à des fins de test.

La programmation s'est effectué avec Simgrid. Simgrid[1] est une plateforme pour le développement et la simulation d'applications distribués sous licence libre. L'intérêt de la simulation est d'une part de pouvoir tester rapidement l'algorithme en condition proche du réel avec un grand nombre de machines simulées tout en supprimant certains aléas dus dans les tests en condition réelles. On peut aussi stocker des informations statistiques globales sur l'algorithme (par exemple de le nombres envoyés, etc.). De plus Simgrid offre aussi la possibilité, avec le même code, de lancer le programme dans un contexte "réel". En effet Simgrid génère deux exécutables, l'un s'éxécutant dans un mode se simulation et l'autre adapté à une utilisation réelle.

1.2 Mécanismes pricipaux de l'algorithme

1.2.1 Comportement en l'absence de fautes

L'algorithme est une extension de l'algorithme de Naimi-Trehel tolérante aux fautes. Ainsi en l'absence de fautes, l'algorithme se comporte de la même manière que celui de Naimi-Trehel. On maintient donc, de manière totalement distribué un arbre dynamique qui permet de remonter vers le dernier demandeur ainsi qu'une file de requête.

Par exemple si l'on considère la configuration initiale suivante : où le carré représente le possésseur

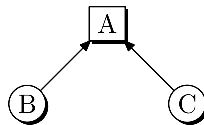


FIG. 1.1: Configuration initiale

du jeton, les flèche noires, les relations de last et les flèches grises les relation de next.

A l'état initial les sites B et C ont leur last pointant vers la racine A. Maintenant si B fait une demande d'un site alors, il envoit une message *request_ token* à son last qui le retransmet jusqu'à la racine, dans ce cas A. On a alors que B invalide son last puisqu'il est maintenant la racine et A tous les noeuds sur le chemin de B à la racine inversent leur *last*. Enfin la racine fait pointer son *next* vers B. On se retrouve alors dans la configuration suivante :

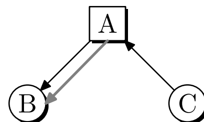


FIG. 1.2: Configuration après l'envoi du message *request_ token* par B

Ce qui au niveau de l'éxécution dans le simulateur se traduit en :

[1]http ://simgrid.gforge.inria.fr/

```
[Locris:node:(1) 0.000000] [NaimiTrehel/INFO] Asking for the token to
    Terminus
[Terminus:node:(3) 0.005588] [NaimiTrehel/INFO] We received a Token
    Request from Locris redirected by Locris.
[Terminus:node:(3) 0.005900] [NaimiTrehel/INFO] Sending the commit
    message to Locris
[Locris:node:(1) 0.011491] [NaimiTrehel/INFO] We have a commit,
    position 1 and first of the predecessor list Terminus
```

avec "Terminus" jouant le rôle de A, "Locris" jouant le rôle de B et "Haven" le rôle de C.

Maintenant si C demande à son tour le jeton, alors la demande remonte jusqu'à B en passant par A, et on obtient alors la configuration suivante :

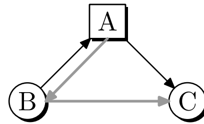


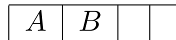
FIG. 1.3: Configuration après l'envoi du message *request_token* par C

ce qui se traduit lors de l'éxécution par :

```
[Haven:node:(2) 2.000000] [NaimiTrehel/INFO] Asking for the token to
    Terminus
[Terminus:node:(3) 2.005587] [NaimiTrehel/INFO] We received a Token
    Request from Haven redirected by Haven.
[Locris:node:(1) 2.011175] [NaimiTrehel/INFO] We received a Token
    Request from Haven redirected by Terminus.
[Locris:node:(1) 2.011487] [NaimiTrehel/INFO] Sending the commit
    message to Haven
[Haven:node:(2) 2.017081] [NaimiTrehel/INFO] We have a commit, position
     2 and first of the predecessor list Terminus
```

Les messages de type *commit* sont spécifiques à la gestion des fautes et ne sont pas présents dans l'algorithme de Naimi-Trehel. Ils contiennent principalement une liste d'au plus k prédécesseurs dans le file des *nexts*. La construction de cette file est aussi distribuée. Ainsi lors de l'envoi du premier *commit* à B, comme A n'a pas de précécésseur dans la file des *nexts* donc la file se constitue seulement de A :

puis lors de l'envoi du deuxième commit à C. B joint les $k-1$ derniers éléments (au plus) de sa propre file et s'ajoute lui-même en queue de file :



Nous analysons maintenant le comportement de l'algorithme en cas de fautes :

1.2.2 Comportement lors de fautes dans la file des nexts

Nous nous plaçons ici dans le cas où le site détectant une faute appartient à la file des *next*, c'est à dire à une liste des prédécesseurs, alors une première méthode de recouvrement de la faute est utilisé.

Dans un premier temps, cette méthode essaye de retrouver un prédécesseur encore vivant dans la file des *nexts*. Si il en trouve un de vivant, il lui envoie un message de reconnexion (cf. Exemple

1).

Si par contre il ne trouve aucun prédécesseur vivant, alors il amorce une diffusion d'une message *search_pos* pour retrouver la plus grande position dans la liste des *nexts* et s'y reconnecter (cf. Exemple 2).

Étant dans un contexte de simulation, les *pings* ne pouvaient être effectués aussi proprement que dans un contexte réel, nous avons choisi de simplifier l'algorithme et d'envoyer directement un message de reconnexion aux prédécesseurs. Cela est particulièrement efficace si le timer de reconnection est court. En effet, si le site est mort alors le message de reconnexion échouera et si il est vivant, alors on aura reconnecté la file des *next*. Il peut arriver un cas où un élément de la file n'a plus le jeton car il a déjà envoyé son jeton au suivant mais reçoit néanmoins un message de reconnection. En effet si les successeurs sont mort, alors le jeton est perdu et doit être régénéré. Dans ce cas nous avons choisi d'ignorer le message de reconnexion donc la suite s'effectue avec un diffusion du message de recherche de la position puis, à l'arrivé du timer, à la régénération du jeton.

Il aurait été possible d'éviter cette diffusion en procédant de la manière suivante :

Si un site reçoit un message de reconnection venant d'un autre site en phase de recherche des prédécesseurs, alors c'est que tous les sites dans la file des next entre lui et le site qui cherche sont morts. En particulier comme celui-ci ne possède plus le jeton et que le jeton n'est pas arrivé à celui qui fait la recherche des prédécesseurs, c'est que le jeton est perdu. Ainsi, celui qui reçoit le message *prev_ alive* regénère le jeton et accepte la reconnexion.

Exemple 1

On se place dans la configuration suivante avec $k = 2$:

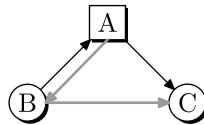


Fig. 1.4: Configuration initiale

Ainsi C possède B et A dans la file des *next*. Or B va mourrir alors que A possède encore le jeton :

```
[Locris:node:(1) 2.011487] [NaimiTrehel/INFO] Sending the commit
    message to Haven
[Haven:node:(2) 2.017081] [NaimiTrehel/INFO] We have a commit, position
     2 and first of the predecessor list Terminus
[Locris:node:(1) 7.000000] [NaimiTrehel/INFO] Locris has failed!
```

puis C détecte l'absence du jeton au boût d'un certain temps et recherche ses prédécesseurs et se reconnecte au premier site vivant :

```
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Token timer elapsed!!
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Searching a valid
    predecessor
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Sending ping to Locris
[Haven:node:(2) 18.005740] [NaimiTrehel/INFO] Not alive
[Haven:node:(2) 18.005740] [NaimiTrehel/INFO] Sending ping to Terminus
[Terminus:node:(3) 18.011636] [NaimiTrehel/INFO] Sending the
    ack_reconnect message to Haven
```

```
[Haven:node:(2) 18.017227] [NaimiTrehel/INFO] We have been reconnected
    to Terminus
[Haven:node:(2) 18.017227] [NaimiTrehel/INFO] We have ack_reconnect,
    position 1 and first of the list Terminus
```

De plus un mécanisme consitant à diffuser les noms des sites défectueux, ainsi les sites ayant leur *last* pointant vers l'un de ses site peuvent le mettre à jour en le faisant pointer vers le site originaire du message. On arrive ainsi à la configuration finale :

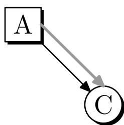


Fig. 1.5: Configuration après recherche des prédécesseurs

Enfin C reçoit le jeton de A :

```
[Terminus:node:(3) 27.005429] [NaimiTrehel/INFO] We loose the token for
     Haven :(
[Haven:node:(2) 27.005429] [NaimiTrehel/INFO] We are the master of the
    token !!!
```

Exemple 2

On se place dans la même configuration initale que pour l'exemple précédent mais cette fois on suppose que $k = 1$. Ainsi C ne possède que B dans la liste des prédécesseurs et celui-ci va mourrir :

```
[Haven:node:(2) 2.017077] [NaimiTrehel/INFO] We have a commit, position
     2 and first of the predecessor list Locris
[Locris:node:(1) 4.000000] [NaimiTrehel/INFO] Locris has failed!
```

Comme précédemment, C va chercher les prédécesseurs vivant mais cette fois ne va en trouver aucun. Il va donc poursuivre le recouvrement en diffusant un message de type *search_pos* à tous les noeuds :

```
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Token timer elapsed!!
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Searching a valid
    predecessor
[Haven:node:(2) 13.000000] [NaimiTrehel/INFO] Sending ping to Locris
[Haven:node:(2) 18.005740] [NaimiTrehel/INFO] Not alive
[Haven:node:(2) 18.005896] [NaimiTrehel/INFO] Sending search pos to
    Locris
[Haven:node:(2) 18.011642] [NaimiTrehel/INFO] Sending search pos to
    Terminus
```

Dans son message *search_pos*, le noeud donne sa place dans la liste des *nexts* qu'il a reçu lors du message *commit*. Tous les noeuds ayant une place inférieure répondent avec leur place dans la liste et le site choisi celui dont la place dans la file est la plus grande. Dans se cas, c'est A et sa place est 0 :

```
[Haven:node:(2) 24.017233] [NaimiTrehel/INFO] Peer with maximum
    position who respond to search pos : Terminus
[Haven:node:(2) 24.017233] [NaimiTrehel/INFO] Reconnection message
    sended to Terminus
[Terminus:node:(3) 24.023129] [NaimiTrehel/INFO] Sending the
    ack_reconnect message to Haven
```

```
[Haven:node:(2) 24.028720] [NaimiTrehel/INFO] We have ack_reconnect,
    position 1 and first of the list Terminus
```

Et la file des nexts est alors reconstruite.

Dans le cas où le site ne reçoit aucun message en réponse à son message *search_pos*, il regénère simplement le jeton et passe en section critique.

1.2.3 Comportement lors de fautes en dehors de la file des nexts

On considère maintenant les erreurs détectés par des sites n'étant pas dans la liste des *nexts*.

Cela peut par exemple arriver lorsque le message de commit n'est pas reçu car le site qui devait l'envoyer est mort avant.

Ce cas est plus compliqué à traiter car une partie de l'arbre doit être reconstruite. Néanmoins, pratiquement, cette reconstruction est totalement distribué est ne comporte aucune difficulté. Elle consiste en :

- pour les sites attendant le jeton sans position dans la file, à placer leur *last* à la même valeur que leur *next*.
- pour les autres à faire pointer leur *last* vers le site qui à envoyer le message *search_pos*.

Exemple

On se place dans la même configuration initale que pour dans l'Exemple 2 précédent mais cette fois B va mourrir avant d'avoir pu envoyer le message *commit* à C :

```
[Locris:node:(1) 0.011491] [NaimiTrehel/INFO] We have a commit,
    position 1 and first of the predecessor list Terminus
[Haven:node:(2) 2.000000] [NaimiTrehel/INFO] Asking for the token to
    Terminus
[Locris:node:(1) 2.000000] [NaimiTrehel/INFO] Locris has failed!
[Terminus:node:(3) 2.005587] [NaimiTrehel/INFO] We received a Token
    Request from Haven redirected by Haven.
```

Or comme B est mort, il n'enverra jamais le *commit* à C, et C va diffuser un message *search_queue* pour se reconnecter à un noeud valide :

```
[Haven:node:(2) 8.000000] [NaimiTrehel/INFO] Commit timer elapsed!!
[Haven:node:(2) 8.000156] [NaimiTrehel/INFO] Sending search queue to
    Locris
[Haven:node:(2) 8.005897] [NaimiTrehel/INFO] Sending search queue to
    Terminus
[Haven:node:(2) 8.017068] [NaimiTrehel/INFO] Reconnection message
    sended to Terminus
[Terminus:node:(3) 8.022965] [NaimiTrehel/INFO] Sending the
    ack_reconnect message to Haven
[Haven:node:(2) 8.028555] [NaimiTrehel/INFO] We have ack_reconnect,
    position 1 and first of the list Terminus
```

De la même manière que précédemment, si celui qui à entammer la recherche ne reçoit aucune réponse, il regénère le jeton.

Les messages de type *search_queue* sont estampillés avec la date de début du recouvrement. Ainsi cela permet d'élire un unique site privilégié qui procèdera au recouvrement.

Bibliographie

[SABS] , J. Sopena, L. Arantes, M. Bertier, P. Sens, A Fault-Tolerant Token-Based Mutual Exclusion Algorithm Using a Dynamic Tree