POLYTECH'RICM2
Examen d'algorithmique distribuée
Durée 2h

Tous documents autorisés – Le barême est indicatif

22 avril 2004

Vague (3 pts) Soit un réseau incomplètement maillé par des liaisons bidirectionnelles FIFO fiables. Chaque site possède un ensemble L de liaisons. Les sites sont identifiés par un numéro de 0 à N-1. Tout site peut émettre des vagues sans restriction.

1) Rappeler l'algorithme général[1].

La propriété FIFO des liaisons est préservée au FLUX de vagues issues d'un même initiateur mais perdue au REFLUX du fait que ces vagues peuvent remonter à l'initiateur par des chemin différents.

```
/* FLUX */
décision_vague ==>
      LastH[ICI()]=HL++;
      VAGUES.insert(<ICI(),LastH[ICI()]>, < L,{}>);
      pourtout l dans L l!<TOK, ICI(),LastH[ICI()] >

exist l dans L l?<TOK,I,h> && card(L)==1  ==>
       l!<ACK, I,h>

exist l dans L l?<TOK,I,h> && card(L)>1  ==>
      if( lastH[I]<h) // nouvelle vague
          LastH[I]=h;
          VAGUES.insert(<I,h>, < L-{l},{l}>);
          pourtout ll dans L-{l} ll!<TOK, I,h >
      else // boucle
          wait,up=VAGUE.extract(<I,h>);
          wait-={l};
          if(wait=={} && I!=ICI()) up!>ACK,I,h>
          elseif(wait=={} && I==ICI())  TERMINER
          else VAGUES.insert(<I,h>, <wait,up>);

/* REFLUX */
exist l dans L l?<ACK,I,h,.....> && I==ICI()  ==>
      wait,up=VAGUE.extract(<I,h>);
      wait-={l};
      if(wait=={})  TERMINER
      else VAGUES.insert(<I,h>, <wait,up>);

exist l dans L l?<ACK,I,h,.....> && I!=ICI()  ==>
```

[1] On ne doit garder que les description des vagues en cours et non terminées.

```
        wait,up=VAGUES.extract(<I,h>);
        wait-={l};
        if(wait=={}) up!>ACK,I,h>
        else VAGUE.insert(<I,h>, <wait,up>);
```

2) Préciser le rôle des variables d'état utilisées

HL horloge logique de numérotation des vagues d'un noeud
LastH[] tableau de taille égale au nombre de noeuds qui maintient les derniers numéros des vagues issues des noeuds
VAGUE[] table associative qui permet à un noeud d'associer à toute vague désignée par la paire <initiateur, date logique> le chemin de retoure (up) et l'ensemble des liaisons sur lesquelles le noeud attend un ACK.

3) Caractériser les propriétés requises et problème d'implantation des types de données utilisés.

Le seul problème est la gestion de la table associative VAGUE car sa taille varie dynamiquement du fait que le nombre de vagues en cours issues des noeuds dépend de la capacité d'émission des noeuds (débit) et du délai de transit dans le réseau qui dépend du diamètre et de l'engorgement. On doit donc gérer explicitement cette table en insérant ou retirant des associations via les opérateurs `insert()` et `retract()` . C'est une implantation de type table de hachage, arbre équkibrée etc..

Première réponse au plus

(6 pts) Une vague générale accumule les réponses de tous les noeuds. Ce n'est pas toujours nécessaire. Dans le cas de vagues de recherche d'information, on peut se satisfaire d'une seule réponse d'un noeud quelconque. On va donc s'intéresser à définir une vague qui rend **une seule** réponse(la première) et **oublie** les autres réponses.

Il faut pouvoir modéliser si une réponse satisfaisante est possible. On supposera donc disposer d'une procédure de décision `SUCCESS()`qui permet de savoir si un noeud dispose de la bonne réponse directement à la réception du TOK ou d'un ACK

4) Comment assurer la terminaison d'une vague lors du reflux?

La détection de terminaison est nécessaire pour nettoyer la table VAGUES. C'est normalement fait quand toutes les réponses issues d'un sous arbre ont été reçues. Il faut donc continuer à gérer cette synchronisation.

5) Quelle incidence sur la gestion des variables d'état utilisées?

Soit on continue à gérer la synchronisation du REFLUX en mémorisant dans VAGUES l'information supplémentaire que le noeud a oui ou non déja remonté une réponse valide. Soit on termine à la réception de la première réponse valide. Il faut toutefois éviter de remonter d'autres réponses valides ou de repropager la vague dans le cas d'un rebouclage du TOK.Comme la détection de boucle est assuré par LastH, on poura nettoyer la table VAGUES dès réception d'une bonne réponse sans perdre la détection de boucle. On notera que ceci n'est possible que grace à la propriété FIFO du FLUX.

6) Donner l'algorithme de la vague.

```
/* FLUX */
décision_vague ==>
      if(SUCCESS() TERMINER
      else
          LastH[ICI()]=HL++;
          VAGUES.insert(<ICI(),LastH[ICI()]>, < L,{}>);
          pourtout l dans L l!<TOK, ICI(),LastH[ICI()] >

exist l dans L l?<TOK,I,h> && card(L)==1  ==>
       l!<ACK, I,h>
```

```
exist l dans L l?<TOK,I,h> && card(L)>1  ==>
      if( lastH[I]<h) // nouvelle vague
          LastH[I]=h;
if(SUCCESS) l!<ACK,I,h>
          else
             VAGUES.insert(<I,h>, < L-{l},{l}>);
             pourtout ll dans L-{l} ll!<TOK, I,h >
      elsif  VAGUE.lookup(<I,h>) // boucle
          wait,up=VAGUE.extract(<I,h>);
          wait-={l};
          if(wait=={} && I!=ICI()) up!>ACK,I,h>
          elseif(wait=={} && I==ICI())  TERMINER
          else VAGUES.insert(<I,h>, <wait,up>);
      else // on a déja répondu
          // rien à faire

/* REFLUX */
exist l dans L l?<ACK,I,h,.....> && I==ICI()  ==>
      if  VAGUE.lookup(<I,h>) // pas de réponse positive envoyée
         wait,up=VAGUE.extract(<I,h>);
         wait-={l};
         if(wait=={})  TERMINER
         elsif SUCCESS() TERMINER
         else VAGUES.insert(<I,h>, <wait,up>);
      else // on a déja terminé
           // rien à faire

exist l dans L l?<ACK,I,h,.....> && I!=ICI()  ==>
      if  VAGUE.lookup(<I,h>) // pas de réponse positive envoyée
          wait,up=VAGUES.extract(<I,h>);
          wait-={l};
          if(wait=={}) up!<ACK,I,h>
          elsif SUCCESS()up!>ACK,I,h>
          else VAGUE.insert(<I,h>, <wait,up>);
      else // on a déja terminé
           // rien à faire
```

Réseau non fiable (6 pts) Le manque de fiabilité d'un système distribué provient des pannes de liaison ou de noeud. Ces pannes se traduisent par l'absence de tout ou partie des réponses de sous-arbres. Une façon de traiter ce problème est de faire *pour le mieux* et remonter les réponses une à une sans attendre d'avoir collecter toutes les réponses issues d'un même sous-arbre.

7) Donner l'algorithme général d'une vague "best effort".

```
/* FLUX */
décision_vague ==>
      LastH[ICI()]=HL++;
      UP.insert(<ICI(),LastH[ICI()]>, {});
      pourtout l dans L l!<TOK, ICI(),LastH[ICI()] >

exist l dans L l?<TOK,I,h> && card(L)==1  ==>
       l!<ACK, I,h>
```

```
exist l dans L l?<TOK,I,h> && card(L)>1  ==>
      if( lastH[I]<h) // nouvelle vague
          LastH[I]=h;
          UP.insert(<I,h>, l);
          pourtout ll dans L-{l} ll!<TOK, I,h >
      else // boucle
          // rien faire

/* REFLUX */
exist l dans L l?<ACK,I,h,.....> && I==ICI()  ==>
      REPONSE_DE_PLUS

exist l dans L l?<ACK,I,h.....> && I!=ICI()  ==>
      up=UP.extract(<I,h>);
      up!>ACK,I,h>
      UP.insert(<I,h>, up);
```

8) Peut on déterminer la terminaison d'une vague? Quelle incidence sur la gestion des variables d'état utilisées?
La perte possible des TOK et ACK ne permet plus de se synchroniser sur la propriété que tous les ACK issues d'un sous-arbre ont été reçus. On n'a donc plus que le chemin de retour à mémoriser.

9) Proposer une solution.**On peut nettoyer la table UP sur timeout. Une autre solution est de borner arbitrairement (valeur V) le nombre de vagues issues par un initiateur. Toute nouvelle vague remplace la plus ancienne (gestion de buffer modulo V). Il faut donc prévoir dans l'algorithme précédent que l'extraction du UP peut échouer.**

```
exist l dans L l?<ACK,I,h.....> && I!=ICI()  ==>
   if(UP.lookup( <I,h>))
      up=UP.extract(<I,h>);
      up!>ACK,I,h>
      UP.insert(<I,h>, up);
    else // rien
```

10) Peut on garantir d'avoir une réponse au moins?
NON Tout peut être perdu.
Proposer une solution.
La solution est que l'initiateur reinitie une nouvelle vague sur timeout.

Ordre de parcours (5 pts)

Il est parfois nécessaire de faire circuler l'information entre les noeuds dans un ordre séquentiel toujours le même **pour un initiateur donné**. Alors qu'une vague classique correspond à un parcours parallèle non déterministe en largeur d'un arbre couvrant du réseau, une solution serait une vague effectuant un parcours séquentiel déterministe en profondeur d'abord d'un tel arbre.

11) Donner l'algorithme général d'un tel parcours[2] dans un environnement FIFO fiable.
La seule difficulté porte sur la détection de boucle que l'on ne veut parcourir qu'une fois. A la détection de boucle, on traite le TOK comme un ACK mais à la différence de la vague classique, on génére un ACK car il n'y a pas encore eu de TOK généré sur cette liaison dans l'autre sens qui croiserait le token reçu. Ceci garantira qu'une arête fermant un cycle n'est explorée qu'une fois.

[2] L'opérateur `firstOf()` permet d'obtenir le premier émément d'un ensemmble de liaisons.

```
/* FLUX */
décision_vague ==>
      LastH[ICI()]=HL++;
      VAGUES.insert(<ICI(),LastH[ICI()]>, < L,{}>);
      firstOf(L) l!<TOK, ICI(),LastH[ICI()] >

exist l dans L l?<TOK,I,h> && card(L)==1  ==>
       l!<ACK, I,h>

exist l dans L l?<TOK,I,h> && card(L)>1  ==>
      if( lastH[I]<h) // nouvelle vague
          LastH[I]=h;
          VAGUES.insert(<I,h>, < L-{l},{l}>);
          firstOf( L-{l})!<TOK, I,h >
      else // boucle
          wait,up=VAGUE.extract(<I,h>);
          wait-={l};
          l!>ACK,I,h>
          VAGUES.insert(<I,h>, <wait,up>);

/* REFLUX */
exist l dans L l?<ACK,I,h,.....> && I==ICI()  ==>
      wait,up=VAGUE.extract(<I,h>);
      wait-={l};
      if(wait=={})  TERMINER
      else VAGUES.insert(<I,h>, <wait,up>);

exist l dans L l?<ACK,I,h,.....> && I!=ICI()  ==>
      wait,up=VAGUES.extract(<I,h>);
      wait-={l};
      if(wait=={}) up!>ACK,I,h>
      else VAGUE.insert(<I,h>, <wait,up>);
```

12) Quel est l'ordre de grandeur de la longueur du parcours effectué ?
2*Nombre d'arêtes du graphe car chaque liaison n'est parcouru qu'une fois dans les 2 sens