Détection de la terminaison

1 Terminologie

- Un processus est dit *actif* si il est en train d'exécuter des instructions de son programme. Un processus est dit *inactif* (ou *passif*) dans le cas contraire.
- Un processus inactif est soit *terminé*, s'il a complété toutes les tâches qu'il doit effectuer, soit *en attente* de messages provenant d'autres processus.

Slide 1

- Un message est en *transit* si il a été envoyé mais n'a pas encore été reçu.
- Si tous les processus sont inactifs *et* qu'aucun message n'est en transit, alors le programme distribué est *terminé*.

Dans un système distribué où les communications entre processus se font par échange de messages, le problème de détecter la terminaison de l'ensemble des processus n'est pas un problème trivial : à un instant donné, il est possible qu'un processus soit (localement) inactif, donc prêt à terminer, mais que des messages dirigés vers ce processus soient en transit dans les canaux, messages pouvant alors conduire à la réactivation de ce processus.

Slide 2

2 Envois asynchrones : difficulté de la détection de la terminaison

Avec des envois *asynchrones*, l'algorithme suivant *ne détecte pas* correctement la terminaison :

```
chan messages[1:n](Message);
chan terminaison();

void traiter_message( Message m ) {
  ...
  ... envoyer des messages sur un ou plusieurs canaux messages ...
  ...
}

process Maitre {
  ... envoyer des messages sur un ou plusieurs canaux messages ...

  # Recevoir un signal de terminaison de chacun des processus.
  for[i = 1 to n]
    receive terminaison();
  exit();
}
```

Slide 3

```
process Proc[i = 1 to n] {
  while( !empty(message[i]) ) {
    Message m;

    receive message[i](m);
    traiter_message(m);
  }
  # Aucun message en attente: on termine.
  send terminaison();
}
```

3 Algorithme avec passage de jetons

(Andrews, section 9.6)

Slide 4

Une classe d'algorithme distribuée rencontrée fréquemment en programmation concurrente et distribuée est celle des algorithmes avec *passage de jetons*.

De tels algorithmes sont souvent utilisés dans le contexte d'un *anneau virtuel* reliant les processus entre eux (*token-passing algorithm on a virtual ring*) :

- L'algorithme est *distribué* en ce sens qu'il n'y a pas de controleur centralisé. Toutefois, on retrouve parfois un processus qui *initie* la tâche de contrôle et de manipulation du jeton.
- L'algorithme utilise un *jeton*, c'est-à-dire une sorte spéciale de message pouvant être utilisé pour transmettre des permissions ou pour accumuler de l'information sur l'état global du système.
- Une topologie d'anneau est *surimposée* sur l'ensemble des processus. Cet anneau est *virtuel*, non pas physique, et ne sert qu'à transmettre le jeton entre les processus voisins.

Slide 5

Exemple : Algorithme distribué pour l'accès exclusif à une ressource
(Andrews, section 9.6.1)

On définit une topologie en anneau entre les processus pour la transmission du jeton, topologie indépendante des liens déjà existants entre les processus pour accomplir leur tâche normale

- La topologie en anneau des processus `Helper` : Figure 9.14 (p. 458)
- Les processus `User` et `Helper` : Figure 9.15 (p. 459)

Un processus `User` ne pourra recevoir la permission d'entrer dans sa section critique (via le canal `go`) *que si* le processus `Helper` associé est celui qui possède actuellement le jeton.

Slide 6

4 Détection de la terminaison dans un anneau : Algorithme de Dijkstra

Première tentative

Supposons n processus $P_0, \ldots, P_{n-1}$ organisés en anneau. Lorsque P_0 devient inactif, il génère un jeton qu'il transmet à P_1. Lorsqu'un processus P_i possède le jeton et qu'il devient inactif, il transmet alors le jeton à $P_{(i+1)\%n}$. Lorsque P_0 reçoit le jeton, alors il signale la terminaison.

Question : Pourquoi est-ce que cet algorithme ne fonctionne pas toujours? Quelles conditions faudrait-il avoir pour qu'il fonctionne correctement?

Slide 7

Deuxième tentative

Les n processus sont encore organisés en anneau. Un processus peut être dans deux états possibles, les processus étant tous initialement `blanc`s :

- `blanc` : le processus, lorsqu'il deviendra inactif, sera prêt à terminer ;
- `noir` : le processus, après avoir reçu le jeton de terminaison, a envoyé un message à un autre processus qui avait déjà reçu le jeton ;

Le jeton peut lui-même être de deux types :

- `blanc` : indique que, jusqu'à présent, tout se déroule bien pour la terminaison ;
- `noir` : indique qu'un des processus qui précèdent a (possiblement) été réactivé.

Les règles de transmissions du jeton et de changement d'état du jeton et des processus sont alors les suivantes :

1. Lorsque le processus P_0 devient inactif, il devient `blanc` et envoie un jeton `blanc` à P_1 ;
2. Si un processus P_i envoie un message normal (sans lien avec le jeton de terminaison) au processus P_j, avec $j < i$, alors P_i devient `noir` (pcq. P_j était peut-être inactif) ;
3. Si un processus P_i a le jeton et que P_i est inactif, alors il passe le jeton à $P_{(i+1)\%n}$. Si P_i était `noir`, alors il transmet un jeton `noir`. Si P_i était `blanc`, alors il transmet le jeton sans toucher à sa couleur.

   Après avoir transmis le jeton, P_i devient `blanc`.

L'algorithme se termine lorsque P_0 reçoit un jeton `blanc`.

Slide 8

D'après la littérature — bien que ces conditions ne soient pas toujours explicites — il semble que l'une ou l'autre des conditions suivantes doivent s'appliquer pour que cet algorithme fonctionne correctement :

- Les communications se font de façon synchrone (style rendez-vous) ;
- Les canaux respectent toujours l'ordre *FIFO* d'envoi des messages ;

Malheureusement, aucune de ces conditions ne s'appliquent en Threaded-C.

Autre approche possible : solution présentée par Garavel, Mateescu et Smarandache dans "*Parallel State Space Construction for Model-Checking*" (p. 7) : algorithme de distribution des transitions d'un automate sur les noeuds d'une machine NOW (style EARTH) et où les communications se font par l'intermédiaire de *sockets* (comme dans le RTS EARTH).

Idée générale : le jeton (transmis en plusieurs vagues) indique le nombre total de messages reçus (première vague) puis transmis (deuxième vague). Si ces deux valeurs sont identiques, alors on peut terminer (troisième vague).

Slide 9