IFT1025 Été 2010 — Devoir 4

Miklós Csűrös

13 juillet 2010

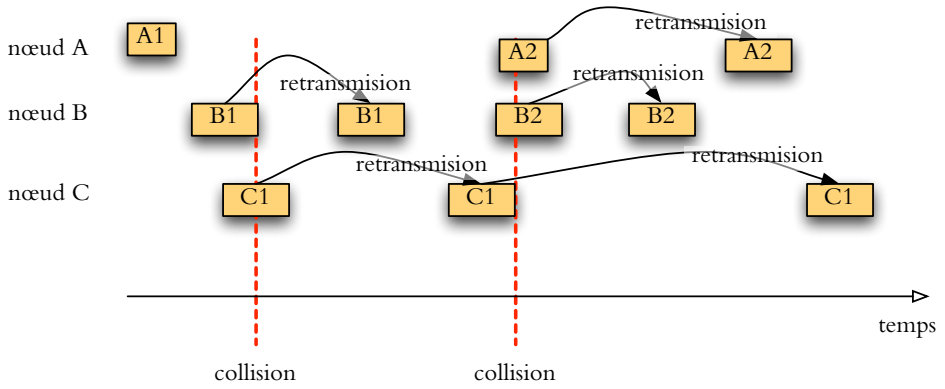
À remettre avant 9 :30 mardi le 27 juillet par courriel à csuros@iro.umontreal. . . . Ce travail est destiné à des équipes de 2 ou 3 étudiant/es.

1 Contrôle d'accès multiple

Le but de ce TP est de se familiariser avec la programmation *multithread* à l'aide de l'implantation d'un protocole d'accès à un médium partagé.

1.1 Accès multiple

Supposons que multiples nœuds ont accès à un médium de communication tel qu'un réseau Ethernet. Si plus qu'un nœud veut utiliser le médium, on a une *collision* et les messages sont perdus. Pour assurer la communication, on doit implanter un protocole d'accès qui permet de détecter et gérer les collisions. Dans ce TP, on veut simuler un protocole inspiré par le protocole d'accès sur les réseaux Ethernet, appellé CSMA/CD.



Dans notre protocole, chaque nœud essaie de transmettre des messages de taille fixe à des intervalles aléatoires. Lors d'une collision, les nœuds essaient de retransmettre le même message après un temps d'attente aléatoire.

On utilise l'algorithme suivant pour la transmission d'un message.

```
   Algorithme TRANSMIT(m)          ▷ transmission de message m
T1 initialiser θ
T2 faire
T3     si le canal n'est pas occupé alors communiquer m ; si collision alors succès ← faux
T4     sinon succès ← faux
T5     si pas de succès alors attendre une intervalle aléatoire t selon Distribution(θ)
T6     θ ← 2θ
T7 tandis que pas de succès
```

1.2 Simulation

Afin d'étudier le comportement de ce système, l'implantation utilise des *threads*. La détection de collisions et la transmission de messages se font à l'aide du mécanisme **wait**-**notify**.

Canal. Le canal sert comme un moniteur/verrou pour coordonner l'accès. Pour «communiquer» un message en Ligne T3, on se met en attente pour une intervalle de longueur fixe : 100 millisecondes (utiliser **wait**(`int`), après avoir stocké le fait que le canal est occupé. La conclusion complète de cette attente veut dire qu'on a eu succès. Une collision se produit quand un autre nœud essaie de transmettre un message pendant cette attente, et on est reveillé par **notify**. (Donc en retournant de **wait**, il faut vérifier si c'était l'expiration du temps ou une collision plutôt qui a mené au retour.)

Attente de longueur aléatoire. En Ligne T5, la longueur d'attente avant de retransmission est un nombre aléatoire selon une distribution fixée. En Ethernet, on utilise la distribution uniforme : on prend

$$t = \texttt{RND}.\textbf{nextInt}(\theta)$$

où `RND` est une instance de `java.util.Random` et $\theta = 2, 4, 8, 16, \ldots$ dans les itérations succéssives. Dans ce TP, on utilise la distribution exponentielle : on prend

$$t = -\texttt{Math.log}(\texttt{RND}.\textbf{nextDouble}()) * \theta$$

avec $\theta = 100, 200, 400, 800, 1600, \ldots$.

Nœud. Pour modéliser chaque nœud, on utilise 2 procéssus légers. Un procéssus est en charge de transmission, et l'autre est en charge de générer des messages.

Le nœud possède un tampon qui peut stocker jusqu'à 10 messages à transmettre. Le procéssus générateur d'un nœud produit des messages séparés par des intervalles de temps selon une distribution exponentielle de paramètre $1/\lambda$. Le procéssus se met en sommeil pour une intervalle aléatoire ($-$`Math.log(RND.`**nextDouble**`())` $* \lambda$), et ajoute un message à la fin du tampon. Si le tampon est rempli, on affiche «Échec d'envoi».

Le procéssus de transmission consulte le tampon pour savoir s'il y a des messages à envoyer (employez la technique producteur-consommateur implantée avec **wait-notify** sur le tampon). Pour communiquer un message, le procéssus de transmission du nœud utilise la méthode de transmission du canal. La valeur retournée indique le succès en Ligne T5. Cette valeur peut être fausse à cause du fait que le canal est occupé, ou à cause d'une collision qui se produit pendant l'attente du procéssus du canal. Quand le message est finalement transmis, on affiche «Envoi de message m».

1.3 Statistiques.

Après l'implantation du système, étudiez la performance du protocole. Votre logiciel est lancé avec le nombre de nœuds et le temps moyen λ entre messages. Pour chaque message, enregistrez si le message était rejeté (tampon rempli), ou le temps jusqu'au succès d'envoi (stockez `System.currentTimeMillis()` lors de l'insertion, ét vérifiez la différence à `System.currentTimeMillis()` lors du succès). Le logiciel devrait reporter le pourcentage de messages rejetés, et le temps moyen d'attente pour les messages envoyés, après 1000 messages générés au total (par tous les nœuds). Montrez la dépendance de ces deux quantités en fonction du nombre n de nœuds et du temps λ pour $\lambda = 10000, 5000, 2000, 1000, 500, 200, 100$ et $n = 1, 2, 3, 4, 5, 10$. (En principe, c'est $\lambda \cdot n$ qui détermine la performance du système – est-ce que vos expériences peuvent valider cette hypothèse ?)

2 Remise de travail

Mettez vos classes dans un package nommé `mac`. Compilez un archive appellé `Mac.jar` qui contient toutes vos classes compilées avec leur code source, ainsi qu'un fichier `Lisezmoi.pdf` où vous expliquez la structure de votre logiciel, et les graphes ou tableaux avec les statistiques. Votre simulation sera lancée par

```
% java -cp Mac.jar mac.Launch lambda n
```

où `lambda` est le temps moyen entre messages chez un nœud, et `n` (> 0) est le nombre de nœuds utilisant le même canal. Le logiciel lancera donc $2n$ procéssus pour modéliser les nœuds. N'oubliez pas de les terminer après avoir généreé assez (1000) de messages (sinon votre système d'explitation sera envahi par des procéssus zombies !).

Envoyez le fichier jar en attachement dans un courriel à `csuros@iro.umontreal...`. L'évaluation considère les aspect suivants de votre travail.

- ⋆ Conception de classes
- ⋆ Clarité du code (commentaires javadoc !)
- ⋆ Fonctionnalité
- ⋆ Analyse des simulations

Boni Vous pouvez avoir jusqu'à 25% de boni pour des expériences avec d'autres politiques de choisir le temps d'attente (p.e., distribution uniforme, ou θ initialisé et augmenté différemment).

Travail estimé. Il est possible d'implanter le sytème avec rélativement peu de code, mais faites attention à la synchronisation des *thread*s.