Ecole Polytechnique

Cours : "Parallélisme"
Travaux dirigés

E. Goubault, S. Putot & O. Bouissou

TD 7

22 mars 2006

1 Simulation de processeurs en anneau

Implémenter en RMI un objet permettant de passer un message d'une machine à une autre, de telle façon à simuler les opérations de passage de message sur un réseau en anneau vues en cours. On appelera un serveur (classe `Node`) sur chaque noeud du réseau en anneau que l'on voudra simuler. A la ligne de commande, on donnera au serveur le nom de la machine serveur et le numéro de port sur lequel communiquer. Chaque objet créé par ce serveur pourra exécuter les méthodes suivantes:

- `void send(int x);` (on pourra se content de la faire bloquante, dans cette première partie)
- `int recv();` comme dans le cours sur les réseaux en anneau
- `void setBuffer(int x);`: stocke la valeur `x` dans l'entier local (privé) à l'objet distant. En fait, `send` sur un noeud fait essentiellement `setBuffer` sur le noeud suivant
- `public void connect(String nodeName,String port);` pour connecter le noeud courant avec celui représenté par `nodename` sur le port (de `rmiregistry`) `port`

Une façon de programmer cela est de considérer chaque objet distant s'exécutant sur un noeud comme étant un client pour le serveur du noeud suivant, et un serveur pour l'objet représentant son noeud prédécesseur.

On aura donc:

```
public class Node extends UnicastRemoteObject implements NodeInterface {
    private String id;
    private int buffer;
    private NodeInterface next;
...
```

`id` contient le nom du noeud, `buffer` le tampon de communication entrant au noeud (qui vaudra `-1` si aucun message n'est entrant, un nombre positif sinon), et `next` est le pointeur vers l'objet distribué représentant le noeud suivant dans l'anneau.

On pourra implémenter une classe `Ring` qui connectera tous les noeuds choisis par l'utilisateur (donnés à la ligne de commande) ainsi que les ports correspondants, en un anneau. Enfin on pourra tester la bonne circulation des messages en changeant un peu la classe `Ring` afin qu'elle fasse circuler un entier de noeud en noeud, en faisant afficher la valeur transitant sur chaque noeud, à partir du premier noeud.

2 Diffusion sur un anneau

Tester la simulation d'anneau de processeurs en programmant la diffusion simple pipelinée vue en cours. On pourra tester les performances du réseau et de l'algorithme de diffusion pipelinée en faisant varier le paramètre r. Pour tester la diffusion pipelinée, il faut créer de longs messages. On remplacera donc les `int` dans l'interface précédente, par des `String` (cf. sur la page web: `NodeStringInterface2`):

```
import java.rmi.*;

public interface NodeStringInterface2 extends Remote {
    public void setBuffer(String x) throws RemoteException;
    public void send(String x) throws RemoteException;
    public String recv() throws RemoteException;
    public void connect(String nodeName,String port) throws RemoteException;
}
```

Il sera facile de créer de longs messages, par exemple, par:

```
static String init;

...

init = "toto";
for (int i=0; i<20;i++)
    init = init+init;
System.out.println(init.length());
```

(ici on un une message d'environ 4 Mo).

Les noeuds de l'anneau sont implémentés par une classe `NodeString2`, une variante de la classe `Node` faite précédemment:

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class NodeString2 extends UnicastRemoteObject
             implements NodeStringInterface2, Runnable {
    private String id;
    private Vector buffer;
    private NodeStringInterface2 next;
...
```

Les difficultés à résoudre sont en effet nombreuses:

- Pour que la simulation ait un sens, il est indispensable que le `send` soit non-bloquant. C'est pourquoi on a choisi ici de faire en sorte que `NodeString2` implémente `Runnable`: chaque `send` à partir du noeud courant créera un thread, responsable de faire `setBuffer` sur le noeud suivant.
- Pour gérer les différents messages pouvant arriver à un noeud donné, il nous faut utiliser un `buffer` plus complexe. On imitera ce qui est fait dans la classe `MsgQueue` du TD 1 (crible d'Erathotène), pour savoir comment manipuler les `Vector`.
- Il faudra bien faire attention à rendre le `recv` bloquant, de façon correcte. On pourra utiliser par exemple un `wait()`, tant que `buffer.size()==0`. Un `send` doit donc faire un `notify` sur le même moniteur. On pourra par exemple utiliser `id` (du receveur!) comme moniteur.

La classe diffusionpipelinee est chargée de donner les ordres aux différents noeuds:

```
import java.rmi.*;

public class diffusionpipelinee extends Thread {
    NodeStringInterface2[] node;
    int procnum;
    int totprocnum;
    String msg;
    static String[] init;
...
```

La aussi plusieurs difficultés sont à résoudre:

- Tous les noeuds étant donnés dans l'ordre à la ligne de commande (avec les numéros de port), on pourra déterminer (une fois les noeuds bien démarrés sur chaque serveur) les NodeStringInterface2 correspondants (par Naming.lookup), ainsi que le nombre total de processeurs sur l'anneau (comme dans le cours, totprocnum).
- Le message à envoyer sera tronçonné en plusieurs sous-messages, stockés dans l'ordre dans le tableau String[] init.
- Le code (SPMD) commun à tous les noeuds est une série de transactions (appels aux méthodes distribuées) à envoyer de façon asynchrones à tous les noeuds, il est donc naturel de créer un thread par noeud; la méthode run de diffusionpipelinee contiendra donc le code vu en cours; typiquement:

```
public void run() {
  try {
  if (procnum == 0) {
    for (int i=0; i<8; i++)
        node[0].send(init[i]);
  } else {
    if (procnum == totprocnum-1) {
        msg = "";
        for (int i=0; i<8; i++)
          msg = msg+node[procnum].recv();
    }
    else {
        String[] message;
        message = new String[8];
        for (int i=0; i<8; i++) {
          message[i] = node[procnum].recv();
          node[procnum].send(message[i]);
        }
    }
  }
  } catch (Exception e) { System.out.println
                ("Exception in thread: "+e); };
}
```

  ici pour une subdivision du message en 8 sous-messages.

- String msg est le message couramment reçu par un processeur