Spécification des Systèmes distribués – Master Technologie de l'Internet 1ère année

Correction TD Linda

Eric Cariou

Modèle Linda

`out(t)` dépose le tuple (n-uplet en français) t dans la mémoire. `in(t')` et `read(t')` permettent de retirer ou de lire un tuple apparié avec t' (t' est un tuple à trous) selon le principe du pattern-matching : on recherche des tuples avec la même structure dans le contenu de la mémoire.

Par exemple, un processus pourra exécuter `out("bonjour", 12)` et un autre pourra exécuter `in(?s, 12)` : il retirera de l'espace des tuples le premier tuple et retournera la valeur "bonjour" pour le premier point d'exclamation (chaine `s`). On cherche dans la mémoire partagée (l'espace des tuples), un tuple qui a le même nombre de champs, de mêmes types et pour les mêmes valeurs précisées aux mêmes endroits.

Linda est à la fois un système de mémoire partagée et de coordination de processus parce que les primitives `in` et `read` sont bloquantes tant qu'un tuple apparié n'est pas présent dans la mémoire.

Linda permet par exemple de facilement implémenter un point de synchronisation de type rendez-vous entre deux processus : l'un va déposer une valeur "a" puis attendre une valeur "b" et l'autre va faire l'action symétrique. On sait que les processus auront passé le point de rendez-vous en même temps :

P1 : P2 :

```
out("a")        out("b")
in("b")         in("a")
```

Question 1

On a une classe `Personne` qui contient un attribut `nom` et un attribut `age`.

Format des tuples dans la mémoire partagée :

- Information sur une personne : (id, nom, age)
- Un tuple particulier de gestion de l'identifiant unique contient un compteur. Quand on ajoute une personne, on retire le tuple de la mémoire, on incrémente le compteur, on redépose le tuple avec la nouvelle valeur du compteur et on ajoute la personne dans la mémoire. Le fait de retirer le tuple avec le compteur empêche deux ajouts de personnes par deux processus en parallèle avec la même valeur de compteur. On initialisera le compteur avec un `out("compteur", 0)`.

Implémentation des 4 primitives :

```
int addPersonne(Personne p) {
      in("compteur", ?id)
      id++
      out(id, p.nom, p.age)
      out("compteur", id)
      return id
}

Personne getPersonne(int id) {
      read(id, ?nom, ?age)
      return new Personne(nom, age)
}

int getId(Personne p) {
      read(?id, p.nom, p.age)
      return id
}

void removePersonne(int id) {
      in(id, ?nom, ?age)
      // on ne fait rien avec les valeurs récupérées
      // on devait juste les enlever de la mémoire
}
```

Question 2

Les opérations `in` et `read` sont bloquantes tant qu'un tuple apparié n'est pas présent dans l'espace : si on cherche une personne ou on veut la supprimer alors qu'elle n'existe pas dans l'espace, les opérations `getPersonne`, `getId` ou `removePersonne` seront bloquantes.

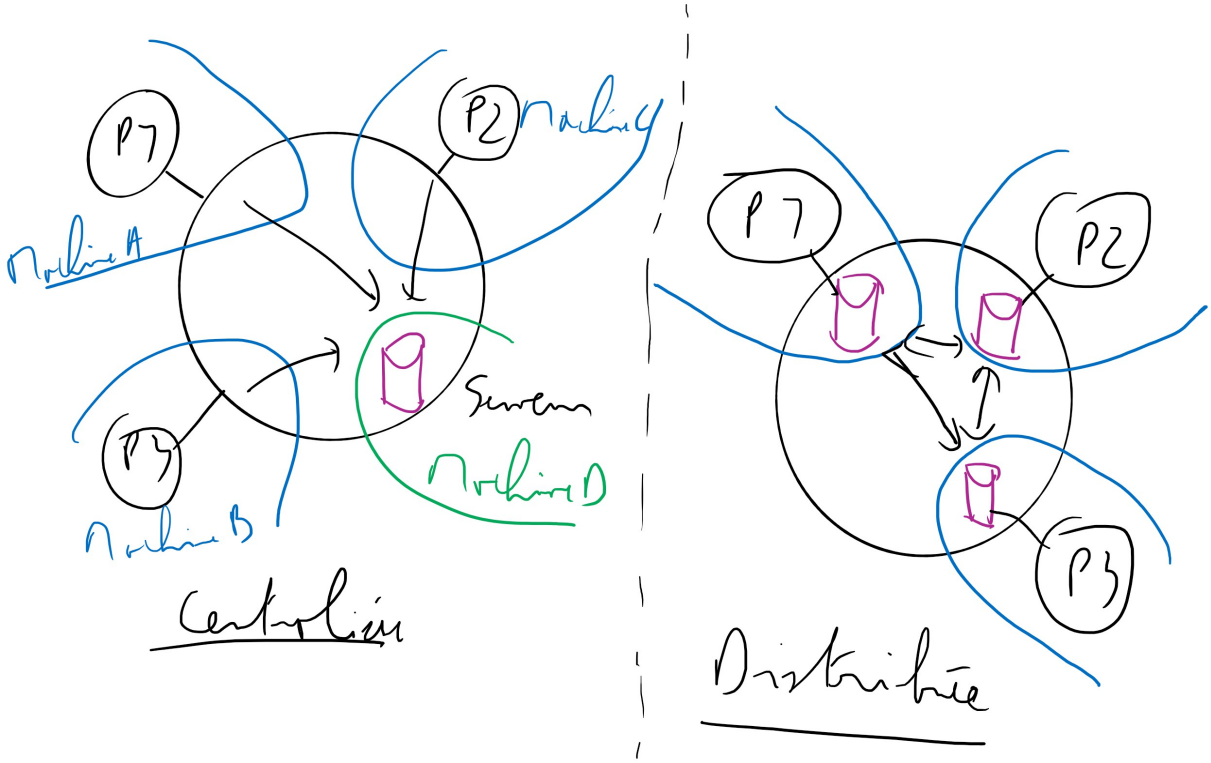
Il faudrait utiliser des variantes avec un temps maximum à attendre et récupérer une valeur particulière (une exception) si on dépasse le délai. Il y a des variantes de Linda qui permettent de faire cela.

Question 3

Au niveau de l'implémentation et de l'accès à la mémoire, il y a 3 modèles applicables (1 centralisé et 2 distribués) :

- Toute la mémoire est localisée sur un serveur. Les processus envoient leurs requêtes pour faire des `out`, `in` ou `read` au serveur. C'est la version la plus simple à implémenter mais dans un contexte de faute (crash de processus), si le serveur plante, plus rien ne fonctionne, tout le contenu de la mémoire est perdu.
- Pour gérer le crash de processus, on peut faire une duplication complète de la mémoire sur chaque processus. Si un processus plante, on ne perd rien puisque chaque processus possède toute la mémoire. Par contre, il faut gérer la cohérence des données : chaque espace mémoire local doit être strictement identique à tous les autres espaces mémoire à tout instant. Dès qu'on fait une modification (un `out` ou un `in`), on doit modifier simultanément tous les espaces locaux : on peut pour cela utiliser un algorithme de transaction distribuée.

- Il y a une variante de la version distribuée : chaque processus embarque une partie de la mémoire sans duplication des tuples. Au niveau de l'implémentation, cela facilite les choses puisqu'il n'y a pas la cohérence des données à assurer vu que les espaces mémoire n'ont pas le même contenu. Au niveau fiabilité, quand un processus plante, on ne perd qu'une partie de la mémoire. Ca n'est pas idéal mais un peu mieux que de tout perdre avec la version centralisée.



On va voir ici comment implémenter la gestion de la mémoire pour le dernier cas :

- Chaque processus possède un espace local de tuples
- Un tuple est présent (en un seul exemplaire) dans un des espaces locaux

Le fonctionnement général est le suivant :

- Quand le processus Pi cherche un tuple avec un `in` ou un `read`, on le cherche d'abord localement dans l'espace de Pi. S'il n'est pas trouvé localement, on diffuse à tous les processus une requête pour savoir s'ils possèdent ce tuple dans leur espace local et on bloque le processus Pi en attendant une réponse.
- Quand le processus Pi dépose un tuple (avec un `out`), s'il satisfait à une requête d'un autre processus, on envoie le tuple à ce processus sinon on le dépose dans l'espace local de Pi.
- Quand on reçoit une requête d'un processus Pj sur Pi, si on possède localement un tuple qui satisfait à la requête, on retire le tuple de l'espace local de Pi et on l'envoie au processus Pj. Sinon, on enregistre la requête du processus Pj pour y répondre éventuellement ultérieurement.
- Quand on reçoit sur Pi un tuple en réponse à une requête : si c'était la requête en cours, on débloque le processus Pi qui était bloqué sur le `in` ou le `read` pour lui retourner le tuple. Si c'était une requête déjà satisfaite pour Pi ou si c'était la terminaison d'une requête de type `read`, on se retrouve avec un tuple dont on n'a plus besoin mais qui pourrait satisfaire une requête d'un autre processus. Pour vérifier cela, on fait localement un `out` avec ce tuple.

Localement, sur Pi, on définit un vecteur de requête de taille N (avec N le nombre de processus) que l'on nommera VR. VR[j] est le patron de tuple que recherche Pj ou `null` si aucune demande en cours. On aura un compteur `nb` qui identifiera la requête qu'on enverra aux autres processus et qui permettra de savoir si on reçoit une réponse à une requête déjà satisfaite.

Initialisation sur Pi :

- Pour tous les cases : VR[j] = null
- nb = 0

out(t) :

1. Parcourt VR pour trouver une case j telle que t répond au patron de tuple cherché : on envoie t à Pj avec le nb de sa requête.
2. Si on n'a pas trouvé de requête à satisfaire, on dépose t dans l'espace local.

in(t) / read(t) :

1. On cherche localement un tuple apparié avec t. Si on le trouve, on le retourne et on le retire de l'espace local en cas de in.
2. Si on ne le trouve pas, on diffuse une requête "t, nb" aux autres processus et on se bloque en attendant une réponse.

Quand on reçoit une requête "t, nb" de Pj :

1. Si on trouve localement un tuple t' qui est apparié avec t, on le retire localement et on l'envoie à Pj avec son nb : "t', nb"
2. Si on ne le trouve pas, on enregistre la requête : VR[j] = "t, nb" (on écrase au passage la requête précédente car si on reçoit une nouvelle de Pj, sa précédente était forcément satisfaite)

Quand on reçoit de Pj une réponse "t, nb" :

1. Si le nb de la réponse est égal au nb local, on vient de recevoir la première réponse à notre requête : on incrémente nb de 1 pour noter que la requête est satisfaite. On débloque le processus et on lui retourne le tuple t. Si on était bloqué sur un read, on exécute out(t)
2. Si le nb de la réponse était inférieur au nb local, on reçoit une réponse à une requête déjà satisfaite : on exécute out(t)

L'unicité d'un tuple est gérée de manière simple : soit un tuple est dans un espace local, soit il est en transit entre deux processus.

Question 4

Deux optimisations sont possibles :

- Pour éviter qu'on réponde de la part de Pi à une ancienne requête de Pj, quand Pj a sa requête satisfaite, il peut diffuser l'information et chaque processus mettra la case VR[j] à `null` pour éviter d'envoyer ultérieurement à Pj des tuples dont il n'a plus besoin.
- Le partage de données n'est pas optimal entre les processus (mais est correct) : on cherche localement un tuple donc on va en priorité récupérer les tuples qu'on a soi-même déposé. On peut systèmatiquement faire une diffusion de la requête pour récupérer si possible un tuple qui se trouve l'espace local d'un autre processus.