Communication par messages

— notations, sémantique, applications —

Introduction

Motivation

- En réseau, les processeurs ne partagent pas de la mémoire.
- En multi-processeurs, le partage de la mémoire pose des problèmes ; peut-on simplifier la communication ?

Plusieurs solutions

- canaux
- rendez-vous
- appel de procedure à distance (*Remote Procedure Call*)
- espace commun de travail (*spaces* de Linda)

Modèles de la communication par messages

Caractéristiques générales

- les processus envoient (**émetteur**) et reçoivent (**récepteur**) des messages
- la réception d'un message ne peut pas précéder son envoi
- toute autre variable a un seul propriétaire

Différentes modèles selon :

1. la synchronisation entre réception et envoi : *synchrone vs. asynchrone*
2. l'adressage des participants : *symétrique, asymétrique ou sans adressage*
3. le sens de la communication : *uni ou bi-direction*

Synchrone versus asynchrone en fonction du dégrée de coopération entre processus participants :

- communication synchrone : l'échange de message est atomique et demande la présence simultanée des deux processus.
- communication asynchrone : sans dépendance temporelle entre l'émission et la réception (l'émetteur peut envoyer des messages sans être bloque, le récepteur lit les messages quand il veut).

Exemples :

- communication synchrone = téléphone
- communication asynchrone = email ou poste

Remarques :

- Même en asynchrone, il y a une synchronisation sur des conditions tampon vide, tampon plein.
- La communication synchrone est usuellement implémentée directement en matériel, la communication asynchrone demande de la mémoire pour le tampon, donc usuellement implémentée en logiciel.

Adressage

- **symetrique** : quand les deux participants connaissent
  - leurs adresses (email, par example)
  - l'adresse d'un **canal** commun
- **asymetrique** : souvent utilisee en systemes **client-serveur**, où le client doit connaitre l'adresse du serveur à contacter (le téléphone, par exemple).
- **sans adresses** : on diffuse un message sans connaître les récepteurs.

Flot des données :

- **uni-direction** : circulation de l'émetteur vers le récepteur
- **bi-direction** : circulation dans les deux sens par communication (message + acquittement).

Communication par canaux

Un canal connecte un émetteur à un récepteur pour la communication de message de *type* selon un modèle :

- synchrone ou asynchrone
- d'adressage symétrique
- uni-direction

Notations

- déclaration d'un canal :

```
chan ch = [ N ] of { typ1 , . . . , typM }
```

- opérations sur un canal :

```
ch ! e1 , . . . , eM    /* envoi de message */
ch ? v1 , . . . , vM    /* reception de message */
ch ? [v1 , . . . , vM]  /* test du vide */
```

Historique

- Hansen 1970 propose un noyau basé sur échange de messages
- Hoare 1978 propose *Communicating Sequential Processes* (CSP)

Sémantique

Si $N > 0$ = communication asynchrone :

- envoi non bloquant si place disponible, copie des valeurs dans une place (message) du canal ;
- réception non-bloquante si message disponible, copie d'un message dans les variables locales.

```
ch ! e1,...,eM |-+        |
               | |        |
               | +------->| ch ? v1,...,vM
```

Si $N = 0$ = communication synchrone :

- envoi bloquant jusqu'à la réception
- réception bloquante jusqu'à l'envoi

affectation des valeurs lors de la rencontre.

```
ch ! e1,...,eM |<------->| ch ? v1,...,vM
```


Exemple 1 : producteur-consommateur synchronisés

```
chan ch : [0] of int
active proctype Producer () {
  int x;
  do :: true -> x = /* produce */ ;
                ch ! x
  od
}
active proctype Consumer () {
  int y;
  do :: true -> ch ? y;
                /* consume y */
  od
}
```

Exemple 2 : processus d'addition de deux entrées

```
chan in1, in2, out : [N] of { int };

proctype Add() {
  int x, y;
  do :: true -> in1 ? x; in2 ? y; out ! (x+y)
  od
}
```

Exemple : sémaphore binaire

Problème : implémenter l'exclusion mutuelle avec la communication synchrone de messages.

```
chan bsem : [0] of { char }
proctype Semaphore () {
  int count = 1;
  do :: count == 1 && (bsem ? P) -> count = 0 % TODO
     :: bsem ? V -> count = 1
  od
}
proctype C () {
  ...
  bsem ! P;
  /* section critique */
  bsem ! V;
  ...
}
```


Exemple : réseau de tri asynchrone

Problème : Trier une liste de n nombres en ordre croissante.

Implémentation 1 : Processus filtre qui reçoit en entrée la liste de nombres et produit en sortie la liste triée.

```
chan input, output : [N] of { int }
proctype Sort () {
  /* recoit tous les nombres du canal input */
  /* trie les nombres */
  /* envoie les nombres tries sur le canal output */
}
```

Propriétés :

- Les nombres envoyés sur output sont en ordre croissante.
- Les nombres envoyés sur output sont une permutation de ceux reçus sur input.

Implémentation 2 : Réseau de processus filtre **Merge** : chaque processus reçoit deux listes triées sur in1 et in2 et envoie leur union triée sur out. La fin de chaque liste est signalée par EOS.

```
chan in1, in2, out : [N] of { int }
proctype Merge () {
  int v1, v2;
  in1 ? v1 ; in2 ? v2;
  do :: (v1 != EOS) && (v2 != EOS) ->
        if :: v1 <= v2 -> out ! v1; in1 ? v1
           :: else -> out ! v2; in2 ? v2
        fi
     :: (v1 == EOS) && (v2 != EOS) ->
        out ! v2; in2 ? v2
     :: (v1 != EOS) && (v2 == EOS) ->
        out ! v1; in1 ? v1
     :: else -> out ! EOS; break
  od }
```

Propriétés :

- in1 et in2 sont vides et $sent[out][n+1] == EOS$.
- Les nombres envoyés sur out sont en ordre croissante et consituent une permutation de ceux reçus sur in1 et in2.

Communication par rendez-vous

Primitive de communication initialement introduite en Ada, le rendez-vous est :

- synchrone : les deux participants doivent être présents
- adressage asymétrique (!) : la location (**entrée**) du rendez-vous appartient à un processus (**accepteur**) dont l'adresse et l'entrée doivent être connues par le processus **appelant**.
- à flot de données bidirectionnel : l'appelant transmet des paramètres et l'accepteur renvoie des résultats.

```
              Appelant   Accepteur
appel entrée | t1        |
             |           |
             |------->t2| parametres acceptes
             .           |
             .           |
             |<-------t3| revoi des resultats
             |           |
```

Exemple : client-serveur

```
proctype Client() {
  int param, result;
  do :: true ->
     param = /* compute parameters */;
     Server.service(param, result);
     /* use result */
  od
}
proctype Server() {
  int p, r;
  do :: true ->
     /* entry point service */
     accept service(p, r) {
       /* compute r from p */
     }
  od
}
```


Appel de procédure à distance (RPC)

Généralisation de l'appel de procédure classique, il permet à un client d'appeler un service d'un serveur qui peut être distant ; un processus est créé pour exécuter la procédure.

Communication :

- synchrone mais un processus est crée pour traiter l'appel
- adressage asymétrique client-serveur
- à flot de données bidirectionnel

Langages : Ada, Java (RMI), CORBA.

Étapes de la mise en place de la procédure distante :

- Écrire la procédure distante.
- Écrire l'interface de cette procédure.
- Compiler le client et le serveur avec l'interface de la procédure pour générer un “talon” (*stub*) par machine.
- Exporter la procédure pour que son nom soit connu par d'autres clients.
- Exécuter un appel de procédure.

Exécution d'un appel RPC

- Le client appelle la procédure.
- De façon transparente pour le client, si on constante que la procédure est distante, le talon local encapsule en messages (*marshaling*) les paramètres.
- Le talon client transmet les paramètres au serveur qui loge cette procédure.
- Les paramètres sont transformés dans le format attendu par la procédure (*unmarshaling*) dans le talon serveur.
- Un processus exécutant la procédure est lancé.
- Les résultats sont encapsulés par le talon du serveur et transmis au client.
- Le talon client transforme le résultat dans le format attendu par le client.
- Le client, bloqué pendant les étapes ci-dessus, est débloqué et continue son exécution.