

Systemes répartis

Modèle standard et principes algorithmiques

Gérard Padiou

Département Informatique et Mathématiques appliquées
ENSEEIH

20 octobre 2011



plan

- 1 Le modèle standard
 - Approche événementielle
 - Causalité
 - Abstraction d'un calcul
- 2 Les protocoles de communication
 - Protocole causalement ordonné
 - Protocoles de diffusion
 - Synchronisme virtuel
 - Structure de contrôle répartie
- 3 Description des algorithmes
 - Description du comportement des processus
 - Algorithme d'élection
 - Arbre de recouvrement



Plan

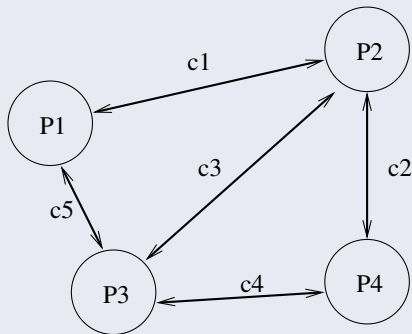
- 1 Le modèle standard
 - Approche événementielle
 - Causalité
 - Abstraction d'un calcul
- 2 Les protocoles de communication
 - Protocole causalement ordonné
 - Protocoles de diffusion
 - Synchronisme virtuel
 - Structure de contrôle répartie
- 3 Description des algorithmes
 - Description du comportement des processus
 - Algorithme d'élection
 - Arbre de recouvrement



Vision statique : Graphe de processus

Description graphique

- Sommets \equiv processus
- Arcs \equiv liaisons de communication



Propriétés

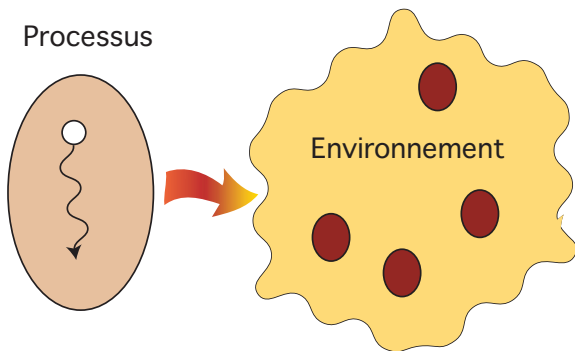
Propriétés des processus

- Un processus possède une identité unique ;
- Un processus possède un état rémanent ;
- Un processus exécute un code séquentiellement ;
- Un processus n'a qu'une connaissance partielle des autres ;
- Un processus peut communiquer avec un voisinage.
- Pas de panne (par arrêt ou comportement byzantin).

Propriétés du réseau

- Multiples paramètres : point à point ou diffusion, (a)synchrone, fiable, délais bornés, etc
- Messages : ni duplication ni erreur ;

Connaissances d'un processus

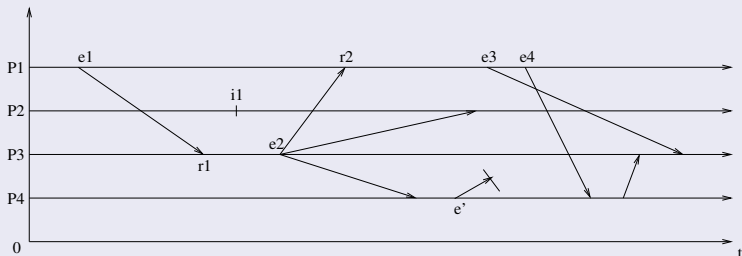


- Nombre de processus ?
- Voisinage de communication ?
- Structure du réseau : maillé, anneau, statique/dynamique, etc

Vision dynamique : Chronogramme

Représentation événementielle

- 3 types d'événements : émission, réception, interne ;
- Mise en évidence de la causalité.

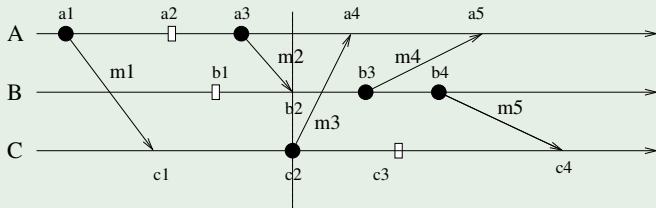


Définition de la relation de causalité

Traduit l'ordre partiel entre événements

- L'émission d'un message précède causalement sa réception ;
- Tous les événements d'un processus sont totalement ordonnés ;
- Transitivité : $\forall e, e', e'' : e \prec e' \prec e'' \Rightarrow e \prec e''$

Exemple

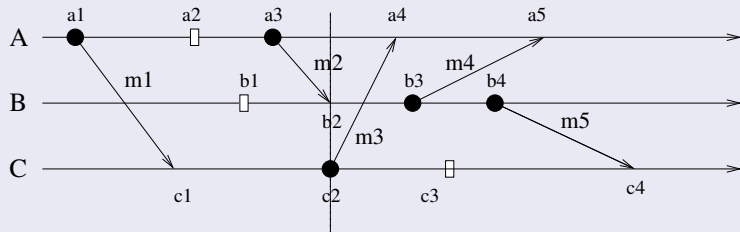


$a_1 \prec a_2 \prec \dots$ mais aussi $a_1 \prec c_1$, $c_2 \prec a_4$, $b_4, \prec c_4$

27

Abstraction d'un calcul réparti

Exécutions causalement équivalentes



- Ensemble d'événements + relation causale \rightarrow ensemble d'exécutions réelles équivalentes ;

$$a_1; b_1; c_1; a_2; \dots \equiv a_1; a_2; c_1; b_1; \dots$$

$$a_1; c_1; a_2; \dots \not\equiv c_1; a_1; a_2; \dots \text{ car } a_1 \prec c_1$$

- Le choix des événements fixe un niveau d'observation.

Plan

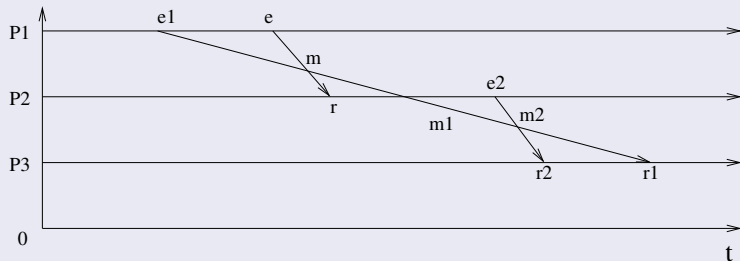
- 1 Le modèle standard
 - Approche événementielle
 - Causalité
 - Abstraction d'un calcul
- 2 Les protocoles de communication
 - Protocole causalement ordonné
 - Protocoles de diffusion
 - Synchronisme virtuel
 - Structure de contrôle répartie
- 3 Description des algorithmes
 - Description du comportement des processus
 - Algorithme d'élection
 - Arbre de recouvrement



Protocole causalement ordonné

Objectif : Mettre de l'ordre ...

Réceptions incohérentes par rapport aux émissions

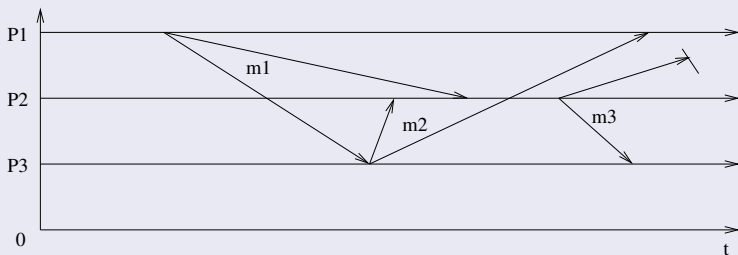


$r_2 \prec r_1$ alors que $e_1 \prec e_2$

Protocole de diffusion

Beaucoup d'imprévus ...

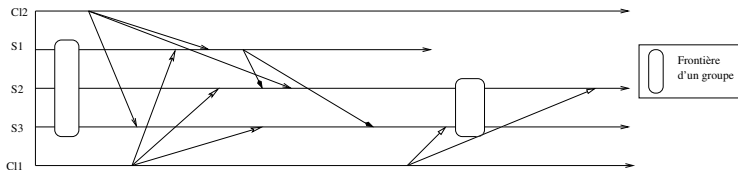
Diffusion vers un groupe de destinataires



Diffusion anarchique

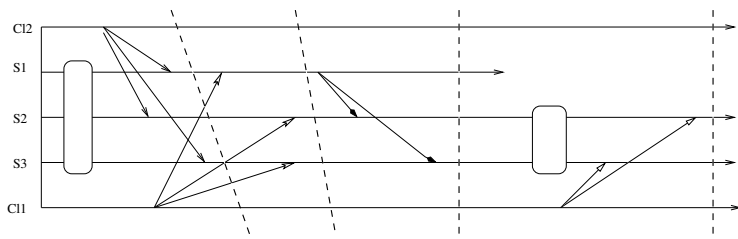
Diffusion vers le groupe $G = \{S_1, S_2, S_3\}$

- Groupe de processus destinataires d'une requête $\{S_1, S_2, S_3\}$
- Réplication des requêtes des clients C_1 et C_2



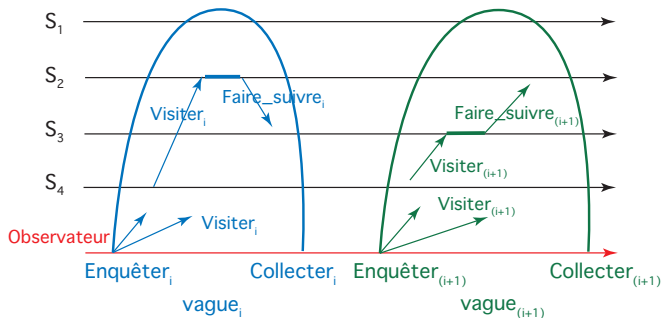
Diffusion virtuellement synchrone

- Même ordre de réception des messages ;
- Atomicité des diffusions ;
- Reconfiguration cohérente.



Notion de vague : itération répartie

Michel Raynal



Notion de vague : itération répartie

Opérations sur une vague

```
class Vague {  
    Enqueter (Object Quoi );  
    /* lance une nouvelle vague */  
    Visiter (Object Quoi, Collecte);  
    /* passage d'une vague sur un site */  
    Faire_Suivre(Object Quoi, Collecte);  
    /* propagation de la vague par un site */  
    Collecte Collecter();  
    /* collecte des résultats et fin de la vague */  
}
```



Plan

- 1 Le modèle standard
 - Approche événementielle
 - Causalité
 - Abstraction d'un calcul
- 2 Les protocoles de communication
 - Protocole causalement ordonné
 - Protocoles de diffusion
 - Synchronisme virtuel
 - Structure de contrôle répartie
- 3 Description des algorithmes
 - Description du comportement des processus
 - Algorithme d'élection
 - Arbre de recouvrement



Principes algorithmiques

- Algorithmes symétriques
 - code répliqué,
 - données initiales propres : identité, voisinage de communication.
- Structurer les échanges de messages :
 - Utilisation de réseaux en anneau ;
 - Utilisation de la structure d'arbre.
- Etudier des problèmes génériques :
 - Les services : datation, exclusion mutuelle, consensus, élection, etc ;
 - Les observations de propriétés stables : terminaison, interblocage ;
 - La tolérance aux fautes : réplication, atomicité.



Description des algorithmes

Principe des commandes gardées (E.W. Dijkstra)

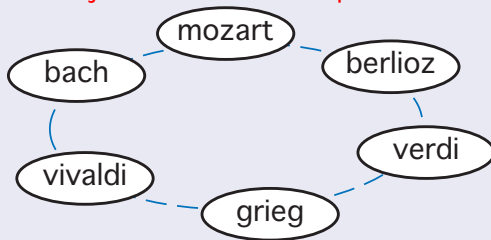
```
process P(i :0..N-1) {
  type Etat = { <Définition des états possibles> };
  Etat EtatCourant= Init;
  <Déclaration de variables locales>
  <Actions d'initialisation>
  while (<Condition>) {
    select {
      [when <Condition-1> =>] [receive M-1(<params-1>);] <Action-1>
      ||
      ...
      ||
      [when <Condition-p> =>] [receive M-p(<params-p>);] <Action-M>
    } // select
  } // while
}
```

Pour un envoi : `send M(<args>) to <destinataire(s)>`

Election

Le problème de l'élection

Objectif : Elire un seul processus



- Un processus a une identité unique qu'il connaît ;
- Un processus ne connaît pas le nombre global de processus ;
- Un processus ne connaît pas l'identité des autres ;
- Communication sur un anneau.

Solution correcte ou fausse ?

☞ On suppose que les processus sont totalement ordonnés (ici par leur indice, en pratique, par leur adresse IP par exemple)

```
process P(i :0..N-1) {  
  //  $\ominus$  et  $\oplus$  : opérateurs modulo  $N$   
  type Etat = {candidat, élu};  
  Etat EtatCourant=candidat;  
  int qui;  
  while (EtatCourant == candidat) {  
    receive Candidat(int qui) from P[i $\ominus$ 1];  
    if(qui < i) send Candidat(qui) to P[i $\oplus$ 1];  
    else if(qui == i) EtatCourant = élu;  
  }  
}
```

Solution qui conduit à l'élection (du + petit)

```
process P(i :0..N-1) {  
  type Etat = {candidat, élu};  
  Etat EtatCourant=candidat;  
  int qui;  
  P[i⊕1].send Candidat(i); /* chacun candidate */  
  while (EtatCourant == candidat) {  
    receive Candidat(int qui) from P[i⊖1];  
    if(qui < i) send Candidat(qui) to P[i⊕1];  
    else if(qui == i) EtatCourant = élu;  
  }  
}
```

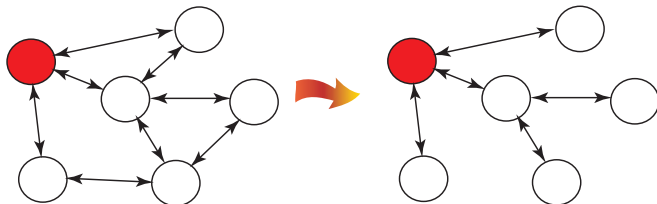
Pas parfait : un seul processus se termine ...

Solution plus complète (tous les processus s'arrêtent)

```
process P(i :0..N-1) {
  type Etat = {candidat,élu,perdant};
  Etat EtatCourant=candidat;
  int qui; int gagnant;
  send Candidat(i) to P[i⊕1]; /* chacun candidate */
  while (EtatCourant == candidat) {
  select {
    receive Candidat(int qui) from P[i⊖1];
    if(qui < i) send Candidat(qui) to P[i⊕1];
    else if(qui == i) EtatCourant = élu;
  }
  receive Elu(int gagnant) from P[i⊖1];
  EtatCourant=perdant; send Elu(gagnant) to P[i⊕1];
} // select
} // textitwhile
if (EtatCourant==élu) {
  send Elu(i) to P[i⊕1]; receive Elu(gagnant) from P[i⊖1];
}
}
```

Construction d'un arbre de recouvrement

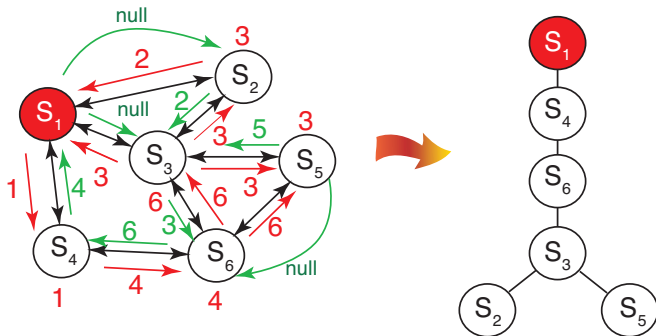
Objectif : structurer la communication grâce à la notion d'arbre



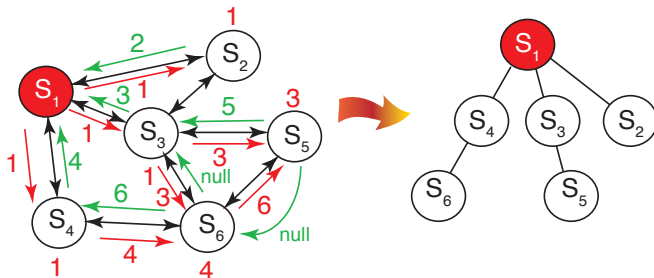
- Un processus connaît ses voisins ;
- Un processus décide de construire un arbre dont il est racine ;
- Les autres processus participent à la construction
≡ « se placent dans l'arbre »



Parcours en profondeur (séquentiel)



Parcours en largeur (en parallèle)



Exercice : Construction d'un arbre de recouvrement

Propagation en parallèle par diffusion vers les voisins

- N processus : `process P(i :0..N-1) {...}`
- Etat d'un processus : `Etat = {hors, estFils, placé}`
- Etat initial : `Etat EC = hors`
- Transitions : `hors → estFils → placé`
- Connaissance de ses voisins `Set<Integer> voisins = ...`
- Définir pour chaque processus : le père et l'ensemble des fils :
`int père; Set<Integer> fils = new Set()`
- Types de messages :
 - Message enquête : `Père(int qui)`
 - Message réponse : `Réponse(int unVoisin, bool estFils)`

Solution

```

process P(i :0..N-1) {
  type Etat = {hors, estFils, placé};
  Etat EC=hors; Set<Integer> voisins=...;
  int père; Set<Integer> fils = new Set<Integer>();
  if (i==0) { /* P0 pris comme racine : diffusion initiale et détermination des fils */
    EC=placé; père=0; send Père(0) to P[j] : j ∈ voisins;
    while(! voisins.isEmpty()) {
      receive Réponse(int unVoisin, bool estPère); voisins.Extract(new Integer(unVoisin));
      if (estPère) fils.Insert(new Integer(unVoisin));
    }
  } /* fin du code exécuté par P0 */
  /* début du code exécuté par tous les autres processus  $\forall i \neq 0$  */
  while (EC!= placé) { /* transitions : hors → estFils → placé */
    select {
      receive Père(int qui);
      if (EC==hors) { /* transition de hors à estFils */
        EC = estFils; père = qui; voisins.Extract(new Integer(père));
        if (voisin.isEmpty()) { /* le noeud est une feuille */
          EC=placé; send Réponse(i,true) to P[qui];
        } else send Père(i) to P[j] : j ∈ voisins; /* propagation du calcul */
      } else /* déjà fils */ send Réponse(i,false) to P[qui];
    }
    receive Réponse(int unVoisin, bool estPère);
    voisins.Extract(new Integer(unVoisin));
    if (estPère) fils.Insert(new Integer(unVoisin));
    if (voisins.isEmpty()) { EC=placé; send Réponse(i,true) to P[père];}
  } // select
} // while
}

```

Une autre solution

```
public class Noeud {
    private Noeud pere = null;
    public List <Noeud> voisins;
    private List<Noeud> fils = new ArrayList<Noeud>();
    // Place le noeud this et propage le calcul
    public Noeud parcourir ( Noeud appelant ) {
        if (pere != null) return null; // Déjà placé
        pere = appelant; // Placement
        for (Noeud v : voisins) { // Recherche des fils
            if (v.parcourir(this) == v) fils.add(v);
        }
        return this;
    }
}
```