Une introduction a quelques techniques du controle distribué à travers un exemple

Noël Plouzeau, Michel Raynal, Jean-Pierre Verjus

IRIA

CENTRE DE RENNES

IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France

Tél. (1) 39 63 55 11

Rapports de Recherche

N° 522

UNE INTRODUCTION À QUELQUES TECHNIQUES DU CONTRÔLE DISTRIBUÉ À TRAVERS UN EXEMPLE

Noël PLOUZEAU
Michel RAYNAL
Jean-Pierre VERJUS

Avril 1986

Une introduction à quelques techniques du contrôle distribué à travers un exemple

Noël Plouzeau, Michel Raynal, Jean-Pierre Verjus

Résumé

Après avoir donné une spécification du problème des producteurs/consommateurs, qui constitue un des paradigmes des problèmes de contrôle rencontrés dans les systèmes et les protocoles, des solutions en sont données dans un contexte distribué. Au travers de celles-ci des méthodes générales de distribution du contrôle et des variables sont dégagées dans une présentation qui se veut essentiellement analytique et pédagogique.

Abstract

A specification of the producer/consumer problem is first given ; such a problem is a paradigm of the control problems faced in systems and protocols, and solutions are given for a distributed context. Meanwhile general methods for performing control and data distribution are exposed in an analytic and pedagogic way.

1. Introduction

Les noyaux de systèmes distribués se distinguent fondamentalement de leurs analogues centralisés par un certain nombre de caractéristiques. Parmi celles-ci l'incapacité d'un des processus du système à capter instantanément un état global de ce système semble être la plus marquante [CHL 85]. Une conséquence de cette particularité est l'exigence de techniques spécialement adaptées au contexte des systèmes distribués pour résoudre un certain nombre des problèmes de contrôle que présente leur mise en œuvre. Maîtriser la conception des systèmes distribués nécessite donc la connaissance et la maîtrise de concepts, techniques, mécanismes et outils adéquats [RAY 85, VER 83] ; le but de cet article est d'en présenter certains. Pour cela une démarche volontairement pédagogique est suivie. Le problème des *Producteurs-Consommateurs*, qui constitue un des paradigmes des problèmes rencontrés dans la conception et la mise en œuvre des systèmes distribués et des protocoles, sert de terrain sur lequel l'étude est effectuée. Dans la partie 2 on pose le problème à un certain niveau d'abstraction dans lequel il y a une seule entité de production et une seule entité de consommation ; la partie 3 en donne une solution. La partie 4 expose les problèmes qui apparaissent lorsque l'entité de production est mise en œuvre par plusieurs processus producteurs. Les parties 5, 6 et 7 présentent trois solutions qui sont alors possibles. Chacune est fondée sur des techniques et des outils particuliers. Au travers de ces présentations, on met en évidence des concepts et des mécanismes fondamentaux du contrôle des systèmes distribués ainsi que des principes de conception structurée.

2. Le problème abstrait

2.1. Enoncé du problème

On considère un système composé de deux processus, respectivement appelés *producteur* et *consommateur*. Le producteur envoie des messages au consommateur, qui doit les absorber. Les messages en attente de consommation sont stockés dans un tampon d'une capacité de n messages. Le processus producteur ne doit pas produire si le tampon est plein ; le consommateur ne peut consommer si le tampon est vide.

2.2. Conditions de production et de consommation

Les conditions de production et de consommation peuvent facilement s'exprimer à l'aide de compteurs ; comme dans [ROV 77] nous considérons les quatre compteurs suivants :

debprod : nombre de productions commencées
finprod : nombre de productions terminées
debcons : nombre de consommations commencées
fincons : nombre de consommations terminées

Le producteur est autorisé à produire lorsque le prédicat Cp est vrai :

(Cp) $debprod - fincons < n$

Le consommateur est autorisé à produire lorsque le prédicat Cc est vrai :

(Cc) $debcons - finprod < 0$

Les prédicats Cp et Cc constituent une solution abstraite au problème. Nous allons les concrétiser dans un système distribué.

3. Une première mise en œuvre

3.1. Distribution du système

Considérons un site de production et un site de consommation sur lesquels sont respectivement placés le

processus *producteur* et le processus *consommateur*, connectés par un réseau (nous confondrons par la suite les termes *site* et *processus* lorqu'il n'y a pas d'ambiguité). Les variables de contrôles (les compteurs) doivent être réparties sur chaque site. En effet, le prédicat autorisant un site à agir (c'est-à-dire à produire ou bien à consommer) comporte deux variables. Considérons le prédicat Cp, fonction de *debprod* et *fincons*. La variable *debprod* n'est lue et modifiée que par le producteur. Nous dirons qu'elle est *locale* au producteur. La variable *fincons* est modifiée par le consommateur et est lue par le producteur. Nous dirons qu'elle est *globale*. Nous appelerons *site origine* d'une variable globale le site qui est le seul à avoir l'accès en lecture et en écriture à cette variable. Nous appelerons *site consultant* un site n'ayant que l'accès en lecture à une variable globale. Une variable globale ne possède pas de site origine si tous les sites y ont accès en lecture et en écriture. Si la variable est un compteur croissant, il est possible que le site consultant ait accès à une variable globale au moyen d'une copie ; cette copie, à cause des délais de transmission, aura une mise à jour retardée.

3.2. Régularité des conditions

Lorsque le site de consommation a modifié *fincons*, il transmet sa valeur au site de production. Celui-ci travaille donc avec une copie de *fincons*, appelée *ifincons*. La valeur de *ifincons* peut être différente de celle de *fincons* à un instant donné. Le compteur *fincons* étant monotone croissant (par construction), nous avons à tout instant :

(I) $ifincons \leq fincons$ (du fait de la mise à jour retardée)

Soit le prédicat Cp' :

(Cp') $debprod - ifincons < n$

Nous voyons que I et Cp' impliquent Cp. Ceci signifie que le producteur peut employer Cp' comme condition de production sans que cela ne lui autorise des comportements interdits. Si nous reécrivons Cp' sous la forme Cp", nous obtenons un prédicat qualifié de régulier [BOC 79].

(Cp") $ifincons - debprod > -n$

Rappelons la définition de la régularité. On considère le prédicat E :

(E) $\Sigma\alpha_i C_i \geq k$

où les C_i représentent des compteurs monotones croissants. Le prédicat est dit régulier par rapport à un compteur C_i si son coefficient α_i est positif. L'évaluation d'un prédicat *P*, qui contient des copies de compteurs, est valide si *P* est régulier par rapport à ces copies.

Le prédicat Cp" auquel est assujettie l'opération de production, évalué sur le site de production, est régulier par rapport à *ifincons* : le coefficient de la copie à mise à jour retardée *ifincons* est en effet positif. Il est donc valide de l'employer comme condition de production. Remarquons qu'il n'est pas nécessaire, si les messages ne se perdent pas, de transmettre la valeur de *fincons* au producteur mais qu'un simple signal "*fin de consommation*" suffit. Si par contre les messages peuvent se perdre il est nécessaire que le consommateur réémette de temps à autre les valeurs de son compteur *fincons*.

3.3. Une mise en œuvre

Le site de production comporte un producteur *Prod* produisant des messages et un contrôleur de production *P* chargé de les transmettre au site de consommation. Celui-ci se compose d'un tampon, d'un consommateur *Cons* et d'un contrôleur de consommation *C* chargé de recevoir les messages envoyés par *P*. Sans que cela limite la généralité de la solution on suppose que sur un site donné une entité (*Prod* ou *Cons*) et son contrôleur communiquent par rendez-vous [HOA 78]. Les communications entre contrôleurs sont asynchrones. Après un rappel de la syntaxe utilisée,nous présentons des algorithmes pour les contrôleurs. Ceux-ci sont immédiatement dérivables des prédicats donnés ci-dessus, et de la signification des variables

utilisées. Par exemple *debprod* est incrémenté à chaque début de production et le prédicat Cp' définit la garde associée à chaque production. Le passage d'une solution abstraite à un programme est explicité dans [AHV 83].

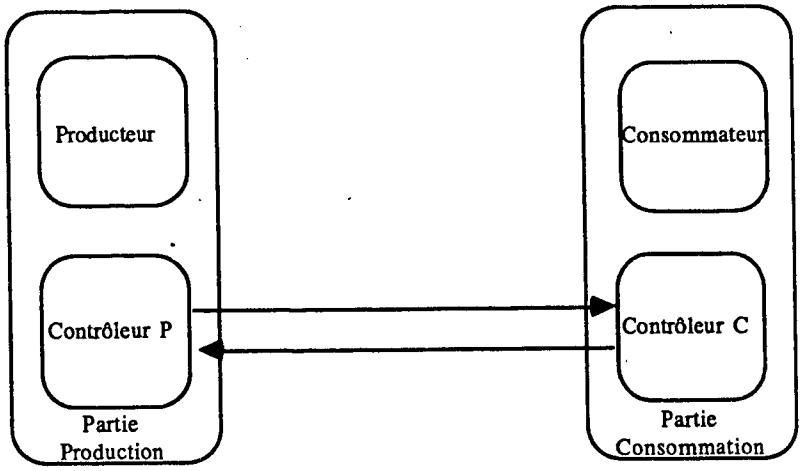

Figure 1.

3.3.1. Conventions d'écriture

Le langage utilisé est proche de *CSP* [HOA 78] : il s'agit d'une formulation selon le shéma *condition* → *action* ; une condition est composée d'une conjonction d'expressions booléennes et éventuellement d'une primitive de communication. L'opérateur de conjonction est représenté par un point-virgule. Une condition est vraie lorsque toutes les expressions booléennes de la conjonction sont vraies et que la communication spécifiée par la primitive de communication est possible. Une alternative est une construction regroupant plusieurs branches *condition→action* ; l'exécution d'une alternative comprend tout d'abord l'évaluation des conditions de chaque branche. On choisit arbitrairement une des conditions évaluées à vrai et on exécute l'action qui lui est associée. Une alternative répétitive est une alternative exécutée tant qu'une des branches au moins a une condition vraie.

Syntaxe de l'alternative :

```
[
   condition → action
[]
   condition → action
...
]
```

La syntaxe de l'alternative répétitive est obtenue à partir de celle de l'alternative simple, en faisant précéder le crochet ouvrant d'une étoile : *[... [] ... [] ...].

Les primitives de communication utilisées sont :

P!x	Envoi d'un message *x* au processus *P* avec rendez-vous.
P?x	Réception d'un message *x* venant de *P* avec rendez-vous.
P!!x	Envoi d'un message *x* au processus *P* sans rendez-vous (émission asynchrone)
P??x	Réception d'un message venant de *P* sans-rendez-vous (réception asynchrone).
!?P	A la manière de [MAR 85a] on utilisera des sondes booléennes relatives à la possibilité de communication. La sonde *!?P* a la valeur vraie si le processus *P* est bloqué en attente de

communication.

Les commentaires sont précédés du double tiret (--) et se terminent à la fin de la ligne. Les phrases entre les symboles < et > sont des macro-instructions.

3.3.2. Algorithme du contrôleur de production

```
-- Définition du processus P
P ::
-- Initialisations
   debprod := 0; -- compteur du nombre d'opérations de production ayant débuté
   ifincons := 0; -- copie locale au processus P du compteur du nombre d'opérations de
                  consommation ayant terminé (le compteur source de la copie est
                  détenu par le contrôleur de consommation C)
   req_en_cours := faux

   *[
      -- Accepter un message du producteur si ce n'est pas déjà fait :
      non req_en_cours ; Prod?message →
                  req_en_cours := vrai

   []
      -- Si une production est prête et si la condition de production est vraie alors
      -- émettre la production :
      req_en_cours ; debprod - ifincons < n →
                  debprod := debprod + 1
                  C!!message
                  req_en_cours := faux


   []
      -- Attendre le signal de fin de consommation venant du contrôleur du consommateur
      -- et mettre à jour la copie ifincons du compteur fincons :
      C??signal_fincons → ifincons := ifincons + 1
   ]
```

3.3.3. Algorithme du contrôleur de consommation

```
C ::
      debcons := 0
      ifinprod := 0
   *[
      -- Transmettre une production au processus consommateur
      -- si la condition de consommation est vraie :
      debcons - ifinprod < 0  ; !?Cons →
                              debcons := debcons + 1
                              <Extraire un message du tampon>
                              Cons!message
                              -- Signaler la fin de l'opération de consommation
                              -- afin que le contrôleur de production puisse mettre à jour
                              -- sa variable ifincons :
                              P!!signal_fincons

   []
      -- Attendre l'arrivée d'un message de production :
```

```
    P??message → ifinprod := ifinprod + 1
    <Mémorisation du message dans le tampon>
]
```

3.3.4. Perte de messages

L'algorithme ci-dessus suppose que les messages *signal_fincons* ne peuvent pas se perdre. La perte d'un tel message a pour conséquence la perte définitive d'une entrée libre du tampon. Si nous ôtons l'hypothèse de non-perte des messages *signal_fincons*, il est nécessaire de transmettre la valeur *fincons* qui vaut alors *debcons*, au lieu d'un simple signal. L'algorithme doit alors être modifié de la façon suivante. Dans le texte du contrôleur de production la dernière branche de l'alternative répétitive devient :

```
C??x →fincons := x
```

Le contrôleur de production reçoit alors du contrôleur de consommation la valeur de *fincons*. Dans le texte du contrôleur de consommation il faut ajouter une variable *fincons*, incrémentée après chaque consommation, remplacer l' émission P!!signal_fincons par P!!fincons qui transmet la valeur de *fincons* au contrôleur *P*. Il faut également ajouter une branche à l'alternative répétitive de *C* :

```
-- Compenser les pertes de messages en réémettant la valeur de fincons
-- de temps en temps.
   vrai → P!!fincons
```

Dans le cas où tous les messages (qu'ils véhiculent du contrôle ou des données) peuvent se perdre il est nécessaire d'envisager des règles de comportement particulières : le protocole de Stenning fournit une solution à ce problème [STE 76].

Remarque sur les compteurs croissants.

Afin d'éviter d'utiliser des compteurs croissants (potentiellement à l'infini), on peut dans l'algorithme ci-dessus les remplacer par les variables :

$$Nb\_tampons\_pleins = debprod - ifincons$$
$$Nb\_tampons\_vides = debcons - ifinprod + N$$

qui évoluent dans le domaine $[0:n]$. Mais on peut aussi mettre en œuvre chacun de ces compteurs modulo $n+1$ [RIA 83, VET 86]. Dans la suite de l'article, nous conserverons les compteurs croissants, pédagogiquement plus parlant.

4. Un système avec plusieurs producteurs

Considérons maintenant un système où le producteur unique du paragraphe précédent a été remplacé par un groupe de m producteurs. Nous pouvons séparer le protocole de coopération *Consommateur/Producteurs* en deux parties.

- Le consommateur communique avec un groupe de producteurs considéré comme formant un tout du point de vue du service rendu. Les problèmes vus plus haut ainsi que leurs solutions sont inchangés.

- Les producteurs sont en compétition pour l'accès à une ressource partagée, à savoir les places libres du tampon du consommateur. Le compteur *debprod* est partagé en lecture et en écriture, le compteur *finprod* est partagé en écriture.

Ces deux parties du protocole *Consommateur/Producteurs* sont bien distinctes ; leur mise en œuvre et les réseaux qu'elles utilisent peuvent être différents.

Distribution d'une variable globale

Distribuer la variable *debprod* partagée en lecture et en écriture nécessite l'emploi de l'exclusion mutuelle entre les producteurs. Deux propriétés sont exigées d'un algorithme d'exclusion mutuelle (en plus de la propriété d'exclusion) :

- l'équité d'attribution du privilège de l'exclusion (un site demandant l'exclusion doit l'obtenir au bout d'un temps fini),
- l'absence d'interblocage entre les sites.

Nous allons examiner différentes techniques pour résoudre ce problème.

5. La technique du jeton circulant

5.1. Principe

Le privilège de l'exclusion mutuelle peut être représenté par la possession d'un jeton, qui est un message figurant en un seul exemplaire dans le système. Ce jeton transporte les variables à distribuer en assurant l'exclusion mutuelle. Dans notre exemple, on y déposera *debprod* puis on fera circuler le jeton dans l'ensemble des processus. La circulation du jeton est subordonnée à la topologie du réseau interconnectant les sites de production. Puisque *privilège* est synonyme de *possession du jeton* , le trajet de ce dernier doit assurer l'équité de l'exclusion mutuelle. L'algorithme de routage équitable (*a priori*) le plus simple est de faire circuler le jeton sur un anneau virtuel. Nous allons examiner les conséquences du choix de cette topologie pour le réseau reliant ces producteurs.

5.2. Structure du système

Les *m* producteurs sont répartis sur un anneau unidirectionnel. Les producteurs obtiendront l'exclusion mutuelle selon l'ordre de passage du jeton, qui est un ordre statique. Il apporte l'équité puisqu'à chaque tour du jeton sur l'anneau chaque site obtient le privilège. L'algorithme n'apporte pas d'interblocage.

Chaque producteur doit recevoir régulièrement la nouvelle valeur de *fincons*. Le système de transmission de copie pourrait utiliser un réseau distinct de l'anneau reliant les producteurs. Afin de ne pas utiliser un réseau trop complexe nous emploierons le service de transport offert par le jeton. Le site de consommation sera donc inséré dans l'anneau.

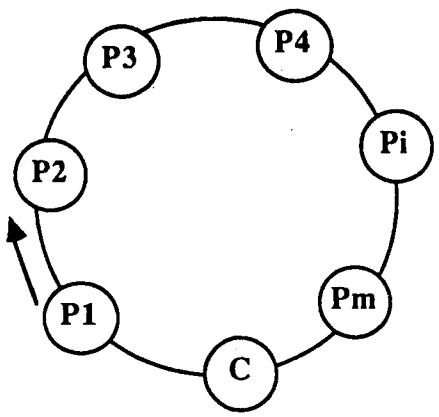

Figure 2.

Les sites **P1** à **Pm** sont les sites de production, **C** est le site de consommation.

5.3. Une première solution.

Le jeton va transporter les compteurs :

- *debprod* pour assurer l'exclusion mutuelle,
- *ifincons* pour fournir une copie retardée de *fincons* à chaque producteur.

L'algorithme employé par chaque producteur est simple. Lorsqu'un producteur **Pi** reçoit le jeton et désire produire, il évalue son prédicat d'autorisation de production, à l'aide des variables transportées par le compteur. Si la condition est satisfaite, **Pi** produit et modifie *debprod* . **Pi** transmet ensuite le jeton à son successeur sur l'anneau. Il lui transmet ce faisant le privilège et *ifincons*, la copie retardée de *fincons*.

Chaque site a une vision équitable du jeton. Malheureusement, celui-ci n'est pas utilisé équitablement. En effet, le producteur situé immédiatement après le consommateur sur l'anneau peut se servir le premier, puis transmet l'information au second, qui peut se servir, etc... Dans le meilleur des cas (lorsque le tampon est entièrement vide) seuls les n premiers producteurs désirant produire pourront le faire. Si le nombre total m de producteurs du système est supérieur à n , les $m - n$ derniers peuvent être en situation de famine si les n premiers veulent tous produire quand passe le jeton.

5.4. Une solution équitable.

Pour rendre le service équitable, nous pouvons :

- soit ordonner les requêtes de productions et les servir dans cet ordre,
- soit assurer une coopération directe entre le consommateur et un producteur.

En effet, l'inégalité entre les producteurs est due à une technique de transmission de *ifincons* favorisant les producteurs placés sur l'anneau après le consommateur. Si tous les producteurs étaient informés équitablement de la valeur de *fincons*, leur connaissance de l'état du système serait identique (au retard de transmission près). Cette solution met en cause la topologie du réseau de communication *Consommateur/Producteurs* .

Une solution n'exigeant pas de modification de la topologie du réseau est de compléter l'ordre statique dans lequel le jeton visite les processus par un ordre dynamique sur les requêtes, assurant l'équité en servant en priorité les requêtes les plus anciennes. Chaque requête de production se voit attribuer un numéro, qui est son rang dans la séquence des requêtes émises. Le compteur *comptereq* servant à les estampiller est partagé en lecture/écriture par les producteurs. Puisque nous disposons déjà d'un jeton nous assurerons la diffusion et la protection de *comptereq* en le plaçant sur le jeton. Chaque producteur ne peut avoir au plus qu'une requête en attente. Son numéro d'ordre sera mémorisé dans la variable locale $rang_i$. Le comportement d'un producteur pour contribuer à l'ordonnancement dynamique des requêtes est alors le suivant.

Le jeton transporte maintenant trois compteurs : *debprod, ifincons, comptereq*. Ces variables ne peuvent être manipulées que par le producteur possesseur du jeton. Nous noterons dans la suite les variables locales à un producteur en les indiçant.

Lorsque le producteur **Pi** reçoit le jeton et désire produire, il prend son rang dans la file des requêtes (qui n'a pas d'existence physique) après avoir incrémenté le compteur *comptereq* . Le code exécuté est donc :

$comptereq := comptereq + 1$
$rang_i := comptereq$

Intuitivement, le producteur aura un comportement équitable s'il ne se sert que lorsque le nombre de places libres *nblibres* est supérieur au nombre des requêtes en attente plus anciennes que celle de **Pi**. (L'expression

nblibres représente le nombre de places libres tel que le perçoit le propriétaire du jeton). Le producteur possédant le jeton a connaissance du nombre $nreqprio_i$ des requêtes prioritaires (i.e. situées avant la sienne). En effet :

$$nreqprio_i = rang_i - debprod - 1$$
$$nblibres = n - (debprod - ifincons)$$

La condition de production assurant l'équité est locale au producteur et s'écrit :

(Ce) $\quad nblibres > nreqprio_i$

(Cette expression suppose que le possesseur du jeton prend son rang avant de tester `Ce`). La valeur $nreqprio_i$ représente le nombre de requêtes plus prioritaires que la sienne. La condition `Ce` implique `Cp'` ; en effet :

(Ce) $\quad n - (debprod - ifincons) > rang_i - debprod - 1 \Leftrightarrow rang_i - ifincons \leq n$ (Ce2)

Avant que le producteur ne produise, $rang_i > debprod$ d'où :

(Ce2) $rang_i - ifincons \leq n \Rightarrow debprod - ifincons < n$ c'est à dire :

`Ce2` $\Rightarrow$ `Cp'`

Ce qui démontre que nous pouvons adopter `Ce2` comme condition de production. Lorsqu'un producteur Pi a pris son rang, son expression $rang_i - ifincons$ est monotone décroissante. En effet chaque tour du jeton sur l'anneau le fait passer sur le site de consommation qui fait croître la valeur de *ifincons* alors que la variable $rang_i$ reste constante. Un producteur verra donc sa requête de production autorisée au bout d'un temps fini. Le service est donc équitable ; l'équité réalisée est de type *linéaire*. En effet le temps qui s'écoule entre le dépôt d'une requête et sa satisfaction est proportionnel au nombre de producteurs.

En effet de $rang_i - debprod \leq m$ (il y a moins de m producteurs avec des requêtes prioritaires)

et $debprod - ifincons \leq n$ (invariant du système)

on déduit :

$rang_i - ifincons \leq n + m$ (la valeur extrême est atteinte lorsque toutes les places du tampon sont occupées et que tous les producteurs désirent produire).

On suppose que le consommateur ne retransmet pas le jeton si rien n'a été consommé ; de plus, un producteur ayant une requête en attente ne peut pas produire une seconde fois tant que la production précédente n'est pas terminée. Dans le pire des cas, *fincons* n'augmente que d'une unité à chaque tour du jeton sur l'anneau. Un producteur attendra donc au maximum m tours du jeton sur l'anneau.

On remarque que lors d'un tour du jeton les producteurs dont la condition de production est vraie produisent selon l'ordre de passage du jeton ; l'ordre de service n'est donc pas nécessairement l'ordre de prise de rang.

Toutes les techniques de contrôle distribué qui emploient le concept du jeton circulant doivent faire face, si tel est le cas, au problème de la perte de ce jeton. Lorque le jeton est perdu, l'algorithme que nous venons de présenter ne fonctionne plus. Pour qu'il puisse fonctionner de nouveau il faut recréér un jeton à l'aide d'une technique de regénération de jeton [LEL 77, MIS 83, RAR 85].

5.5. Algorithmes de contrôle

Des conditions de production et de consommation présentées plus haut nous dérivons un algorithme où chaque contrôleur de production *Pi* possède deux variables booléennes *req_en_cours* et *rang_pris*. La variable *req_en_cours* de *Pi* vaut **vrai** lorsque le contrôleur de production sait que le producteur qui lui est associé désire produire. La variable *rang_pris* de*Pi* vaut **vrai** lorque le contrôleur a pris son rang dans la file des contrôleurs désirant produire. Ceci ne peut se faire que si *req_en_cours* vaut **vrai.**

5.5.1. Algorithme d'un contrôleur de production.

```
Pi ::
   req_en_cours := faux

   *[
            -- Accepter un message du producteur si ce n'est déjà fait :
            non req_en_cours ; Prod?message →
                              req_en_cours := vrai
                              rang_pris := faux
      []
            -- Accepter le jeton venant du prédécesseur sur l'anneau :
            Pi-1?jeton(debprod,ifincons,comptereq) →
                  [
                  -- Si le producteur désire produire et si la prise de rang n'a pas encore
                  --  été faite alors prendre son rang :
                  req_en_cours ; non rang_pris →
                              comptereq := comptereq + 1
                              rang := comptereq
                              rang_pris := vrai
                  ]
                  [
                  -- Si le rang a été pris et si la condition de production est vraie
                  -- alors produire :
                  rang_pris ; rang – ifincons ≤ n →
                              debprod := debprod + 1
                              C!!message
                              req_en_cours := faux
                              rang_pris:=faux
                  ]
                  -- Transmettre le jeton au successeur sur l'anneau :
                  Pi+1!jeton(debprod,ifincons,comptereq)
      ]
```

5.5.2. Algorithme du contrôleur de consommation

```
C ::

   ifinprod := 0
   fincons := 0
   debcons := 0
   debprod := 0
   comptereq := 0
```

```
*[
   -- Accepter le jeton venant du dernier contrôleur de production sur l'anneau et
   -- fournir une copie à jour du compteur fincons au premier contrôleur sur l'anneau
   -- en lui transmettant le jeton :
   Pm?jeton(debprod,ifincons,comptereq) →
            P1!jeton(debprod,fincons,comptereq)
[]
   -- Accepter un message venant d'un contrôleur de production :
   Pi??message →
            ifinprod := ifinprod + 1
            <Inserer le message dans le tampon>
[]
   -- Si la condition de consommation est vraie et si le consommateur désire
   -- consommer alors lui expédier un message de production :
   debcons - ifinprod < 0 ; !?Cons→
            debcons := debcons + 1
            <Extraire un message du tampon>
            C!message
            fincons := fincons + 1
            -- La mise à jour de la copie ifincons se fera lors de la transmission du jeton par
            -- le contrôleur de consommation.

]
```

Remarques

La mise à jour de la copie *ifincons* n'est pas ici aussi explicite que pour les algorithmes examinés au paragraphe 1. Ici nous n'avons qu'une seule copie du compteur *fincons* et elle est transportée par le jeton. Cette copie n'est remise à jour qu'au moment du passage du jeton dans le contrôleur de consommation.

6. Distribution par duplication

6.1. Principes

Au lieu de transporter le compteur *debprod* au moyen d'un jeton, nous pouvons le dupliquer sur chaque site de production. Un problème majeur surgit alors : il faut assurer la cohérence des copies multiples. Cette cohérence comporte deux aspects :

a) Cohérence des copies en lecture ; en l'absence de message en transit dans le réseau les copies doivent être identiques.

b) Cohérence lors des accès conflictuels à une variable globale partagée en écriture.

Une méthode classique pour obtenir a) est de diffuser à tous les sites la valeur d'une copie locale lorsque son propriétaire vient de la modifier. Ceci suppose l'existence d'un réseau permettant la diffusion (à l'aide d'un bus ou d'un maillage complet) entre tous les sites. La technique couramment utilisée pour réaliser b) est l'estampillage des messages par une horloge logique [LAM 78] et la gestion d'une file d'attente des demandes de modification. Les hypothèses de transmission sur le réseau sont le délai fini (non-perte) et le non-déséquencement de messages.

6.2. Structure du système

Le système pris comme exemple comporte une partie *consommation* constituée d'un contrôleur *C* et

d'un consommateur *Cons*. La partie *production* comprend m couples constitués d'un contrôleur de production P_i et d'un producteur $Prod_i$. Les P_i communiquent au moyen d'un réseau complètement maillé, sans perte ni déséquencement de messages. Un autre réseau relie chaque P_i au contrôleur C, avec la propriété de non-perte des messages.

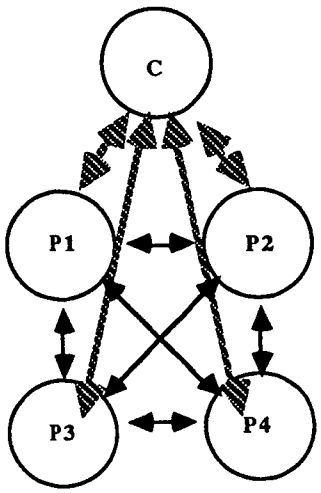

Figure 3.

L'algorithme exécuté par un contrôleur P_i doit assurer quatre fonctions :

- cohérence de la mise à jour des copies : on emploie l'estampillage et la diffusion des mises à jour (la diffusion assure que tout site reçoit les modifications et l'estampillage qu'il est possible de les considérer dans le même ordre sur tous les sites).

- gestion de l'exclusion mutuelle sur la variable *debprod* ; il s'agit de la fonction *coopération entre les producteurs* (protocole *PPP*) ;

- coopération avec le contrôleur de consommation (protocole *PCP*) ;

- coopération avec le producteur $Prod_i$. associé à P_i .

L'algorithme exécuté par le contrôleur C doit assurer deux fonctions :

- coopération avec les P_i (protocole *PCP*)

- coopération avec le consommateur *Cons*.

6.3. Algorithmes de contrôle

Les algorithmes de contrôle associés aux sites de production sont fondés sur l'algorithme d'exclusion mutuelle de Lamport [LAM 78] pour réaliser le protocole *PPP* ; les principes en sont les suivants [RAY 84]. Lorsqu'un site désire l'exclusion mutuelle, il envoie à tous les autres sites un message de requête *req* portant une estampille, qui est la valeur de son horloge logique locale au moment de l'émission du message. Chaque site gère une copie de la file des requêtes en attente. Cette file locale est mise à jour lors de la réception d'un message *req* ; elle est maintenue triée selon les valeurs d'estampille croissantes. Afin que les différentes copies de la file des requêtes soient identiques un site recevant un message *req* répond par un message d'acquittement *acq*. L'exclusion mutuelle est attribuée au site en tête de la file (celui dont la requête possède l'estampille la plus petite). Lorsqu'un site relâche l'exclusion mutuelle, il envoie un

message *rel* à tous les autres sites. Tous les messages portent une estampille et lors de la réception d'un message l'horloge logique locale est mise à jour de manière à permettre d'ordonner totalement les messages du système à partir de leur estampille.
En ce qui concerne le protocole *PCP* les comportements des sites sont analogues au cas précédent.

6.3.1. Contexte d'un contrôleur de production

Variables locales à P_i

debprod : copie locale du compteur, distribuée selon le protocole *PPP*
ifincons : copie locale du compteur, distribuée selon le protocole *PCP*
req_en_cours : booléen, variable d'état du processus
tab_req : message[1..*m*], file des requêtes, distribuée selon la technique de Lamport
h : horloge locale, utilisée dans le protocole *PPP* pour la cohérence des copies.

Fonctions

premier(t,i) : prédicat valant **vrai** si la plus vieille requête figurant dans la file *t* appartient à P_i
req_reçue(t,i) : prédicat valant **vrai** si *t[i]* contient une requête

Procédures

mise_à_jour(h_locale,h_distante) : met à jour l'horloge locale *h_locale* lors de la réception d'un message : *h_locale := max(h_locale,h_distante) + 1*

diffuser(x,i) : envoie le message *x* à tous les P_j , $j \neq i$, et met à jour *tab_req[i]* ; les messages sont envoyés au moyen d'une primitive de communication asynchrone.

6.3.2. Algorithme d'un contrôleur de production

```
Pi ::
   h := 0
   tab_req:=nil;
   deb_prod:=0;
   ifincons:=0;
   req_en_cours:=faux;

*[
   -- Accepter une requête venant de Pj et la mémoriser :
   Pj??req_prod(hj)  → tab_req[j]:=<req_prod,hj>
                       mise_à_jour(hi,hj)
                       -- Envoyer à Pj un accusé de réception :
                       Pj!!acq(hi)
[]
   -- Accepter un accusé de réception venant de Pj  et le mémoriser si Pj n'a pas de requête
   -- en cours :
   Pj??acq(hj) →
                  [ non req_reçue[tab_req,j] →
                       tab_req[j]:=<acq,hj>
```

mise_à_jour(h_i,hj)

]

[]

-- Accepter un message de fin de requête venant de P_j

P_j??rel(h_j) →

tab_req[j]:=<rel,h_j>

mise_à_jour(h_i,hj)

[]

-- Partie concernant le dialogue entre le contrôleur de production et le producteur :

-- Accepter un message de production venant du producteur si ce n'est pas déjà fait :

non req_en_cours;$Prod_i$?message →

req_en_cours:=vrai;

-- Envoyer un message de requête à tous les autres contrôleurs :

diffuser(<req_prod,h_i>);

[]

-- Si la requête de P_i est la première de la file des requêtes et si la condition

-- de production est vraie alors envoyer un message de production au contrôleur

-- de consommation et envoyer aux autres contrôleurs de production un message

-- signalant que la requête de P_i a été satisfaite :

premier(tab_req,i);(deb_prod - ifincons) < n →

debprod:=debprod+1

-- Emission asynchrone :

C!!message

diffuser(<rel_prod,h_i>,i)

req_en_cours:=faux

[]

-- Réception asynchrone :

C??req_cons(ifincons) →

]

6.3.3. Algorithme du contrôleur de communication

Variables

debcons : compteur

ifinprod : compteur mis à jour par la réception d'un message consommable

C ::

debcons:=0

ifinprod:=0

*[

-- Protocole avec le client :

```
-- Si la condition de consommation est vraie alors envoyer un message de production
-- au consommateur :
ifinprod>debcons  ; !? Cons→
                    debcons:=debcons+1
                    <Extraire un message du tampon>
                    Cons!message
                    -- Gestion explicite du compteur
                    fincons:=fincons+1
                    -- Diffuser de manière asynchrone la nouvelle valeur du compteur
                    -- fincons afin que chaque contrôleur de production puisse
                    -- mettre à jour sa variable locale ifincons :
                    k∈[1..m];Pk!!req_cons(fincons)
[]
-- Protocole avec les producteurs :

-- Réception asynchrone d'un message de production venant de l'un des
-- contrôleur de production :
j∈[1..m];Pj??message →
                    ifinprod:=ifinprod+1
                    <Mémorisation du message dans le tampon>
]
```

Remarques

L'ordre de service est l'ordre d'obtention de l'exclusion mutuelle ; ceci donne au service les propriétés d'équité et d'absence d'interblocage.

7. Distribution par éclatement

7.1. Une solution statique

La méthode précédente de distribution par duplication utilisait des variables locales reliées par des contraintes globales. Il est possible de diminuer fortement ces contraintes en divisant les places du tampon entre chaque site de production : on parle alors de distribution par éclatement de variable. Chaque contrôleur de production reçoit un certain quota de places libres n_i qui lui sont ainsi allouées d'une manière statique. La compétition entre les producteurs est nulle car ils ne se partagent aucune ressource.
Nous divisons donc n et le compteur *fincons* en m parties :

$$n = \Sigma\, n_i \quad ; \quad fincons = \Sigma\, fincons_i$$

Le compteur $fincons_i$ contient le nombre d'opérations de consommation ayant consommé une production venant de P_i et ayant terminé. Chaque contrôleur de production P_i utilise la condition de production suivante :

(Cpi) $\quad debprod_i - ifincons_i < n_i$

où $ifincons_i$ est une copie locale à P_i du compteur $fincons_i$. Montrons que l'emploi de **Cpi** comme condition de production est valide, c'est-à-dire que :

$$(\exists i \mid debprod_i - ifincons_i < n_i) \Rightarrow debprod - fincons < n$$

Comme chaque contrôleur de production ne produit que si sa condition de production est vraie on a nécessairement $debprod_i$- $ifincons_i \leq n_i$ pour tout i. Comme nous avons $ifincons_i \leq fincons_i$, s'il existe k tel que **Cpk** soit vraie alors nous avons aussi :

$$\forall i \neq k,\ debprod_i - fincons_i \leq n_i \ \wedge\ debprod_k - fincons_k < n_k$$
$$\Rightarrow (\Sigma_{i \neq k}\, debprod_i + debprod_k) - (\Sigma_{i \neq k} fincons_i + fincons_k) < (\Sigma_{i \neq k} n_i + n_k)$$

Etant donné que $debprod = \Sigma\, debprod_i$, $fincons = \Sigma fincons_i$, $n = \Sigma\, n_i$, cette dernière condition s'écrit

$$debprod - fincons < n$$

Nous venons de montrer que si un contrôleur de production trouve sa condition **Cpi** vraie alors la condition globale de production est vraie. La réciproque n'est pas vraie : le tampon peut contenir des entrées libres alors qu'aucun contrôleur de production n'a sa condition de production vraie. Ce phénomène peut survenir de deux façons. La première cause possible est le décalage entre $fincons_i$ et $ifincons_i$, du fait du retard dans la transmission. Ce problème a déjà été rencontré. La seconde cause possible est le caractère statique de l'allocation des places dans le tampon. En effet, un producteur peut épuiser son quota de places libres et donc voir sa condition de production devenir fausse, alors qu'il existe encore des places libres dans le tampon mais elles sont allouées à d'autres producteurs. L'allocation statique offre par contre l'avantage d'augmenter le degré de parallélisme du système en raison de l'absence de partage d'objet entre les producteurs. Ceux-ci n'ont pas à communiquer entre eux. Le graphe des communications entre les processus du système est un graphe en étoile, le contrôleur de consommation en occupant le centre.

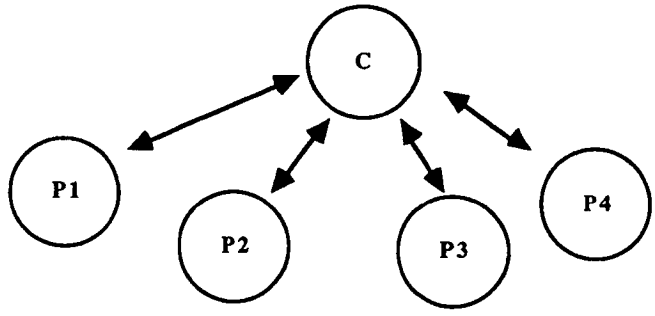

Figure 4.

7.1.1. Algorithme d'un contrôleur de production

```
Pi ::
   -- Variables locales à Pi :
   quota : entier positif ou nul -- Variable contenant ni
   debprod,ifincons : entier positif ou nul

   -- Initialisations :
   C?allocation(quota)  -- Réception du quota initial
   debprod:=0
   ifincons:=0

*[
   -- Dialogue avec le producteur client ; si le service "transmettre une
   -- production au consommateur" est possible, il est effectué immédiatement :
```

```
   debprod-ifincons<quota ; Prod_i?message →
                        C!!message
                        debprod:=debprod+1
[]
   C?fincons →  ifincons:=ifincons+1
]
```

L'algorithme du consommateur utilise une fonction *fournisseur* qui prend en argument un message et rend comme résultat l'identité du processus ayant émis ce message.

7.1.2. Algorithme du contrôleur de consommation

```
C ::

   tampon : ...
   debcons,ifinprod : entier positif ou nul
   crédit : entier positif ou nul
   nbprivilégiés : 0..m-1


crédit := n div m
nbprivilégiés := n mod m
debcons := 0
ifinprod := 0

-- Allocation d'un quota de places libres à chaque contrôleur de production :
-- Nota bene : [1..0] signifiera ici que cette instruction n'a aucun effet.

i ∈ [1..nbprivilégiés] ; P_i ! allocation( crédit + 1 )
i ∈ [privilégiés+1..m] ; P_i ! allocation( crédit )

*[
   -- Si la condition de consommation est vraie alors consommer un message :
   ifinprod - debcons>0 ; !? Cons→
                        debcons:=debcons+1
                        <Extraire un message du tampon>
                        Cons!message
                        -- Signaler la consommation du message au producteur qui l'a émis, afin
                        -- qu'il puisse mettre à  jour sa variable ifincons :
                        j:=fournisseur(message)
                        P_j!!fin_cons
[]
   -- Accepter tout message de production venant d'un contrôleur de production :
   i ∈ [1..m] ; P_i?message →
                        ifinprod:=ifinprod+1
                        <Ranger le message reçu dans le tampon>
]
```

Remarques

L'absence de compétition entre les producteurs, le caractère statique de l'allocation du tampon impliquent que l'équité de l'algorithme dépend uniquement de l'équité de l'algorithme d'allocation statique. Nous avons choisi dans notre exemple un algorithme d'équirépartition.

7.2. Solution mixte

Pour atténuer le caractère trop statique de l'algorithme que nous venons de voir nous devons permettre aux places libres inutilisées de circuler entre les contrôleurs de production en leur permettant de coopérer. Nous pouvons considérer que les algorithmes de répartition par éclatement et les algorithmes de répartition par partage tels que ceux des parties **5** et **6** constituent deux politiques extrêmes : le tampon est soit entièrement local, soit entièrement global. Si nous utilisons simultanément les deux techniques nous pouvons à la fois améliorer le degré de parallélisme et rendre l'allocation moins statique. Nous pouvons poser $n = ns + nd$ où ns est le nombre de places du tampon gérées au moyen de la technique d'éclatement de ressources (allocation statique) et nd est le nombre de places gérées selon la technique vue à la partie **5**. On définit :

$$ns = \sum ns_i \ ; \ fincons = \sum fincons_i + finconsd \ ; \ debprod = \sum debprods_i + debprodd$$

L'algorithme d'allocation dynamique s'applique à un ensemble de producteurs communiquant à l'aide d'un anneau unidirectionnel. La condition de production du processus P_i s'écrit :

(Cpi) (condition de production dans le tampon local alloué statique) **(Cps)**

$\vee$

(condition de production dans le tampon global dynamique) **(Cpd)**

c'est à dire

(Cpi) $(debprods_i - ifincons_i < ns_i)$

$\vee$ $(rang_i - ifinconsd \leq nd)$

Afin d'augmenter le parallélisme, on utilisera de préférence le tampon local lorsque **Cps** et **Cpd** seront simultanément vraies. Il n'y a pas d'interaction entre les deux techniques (pas de variables partagées) donc la mise en parallèle des deux algorithmes est valide. Ceux-ci sont tous deux équitables, l'algorithme résultat de la mise en parallèle l'est donc aussi.

8. Conclusion

Le problème *Producteurs/Consommateurs* nous a permis de présenter un certain nombre de techniques de base utilisées dans la mise en œuvre de systèmes distribués et de protocoles. Ces techniques permettent en fait de réaliser les mécanismes de synchronisation nécessités par le partage d'objets et l'assujétissement de processus. Ces deux problèmes (compétition entre processus *a priori* indépendants et coopération entre processus mutuellement dépendants) ne peuvent être résolus dans un contexte distribué que par l'échange d'informations qui rendent les processus dépendants **[LAM 84]** ; cet échange permet d'assurer les propriétés recherchées (exclusion, équité, etc...).

Outre cet aspect fondamental, les algorithmes de contrôle présentés en mettent deux autres en évidence. Il y a tout d'abord les relations étroites entre une topologie donnée et un contrôle distribué ; citons par exemple le maillage en anneau qui va de pair avec le jeton circulant ou encore le maillage complet et l'estampillage **[RAY 85]**. L'utilisation systématique d'une topologie "classique" induit donc des propriétés implicites sur la mise en œuvre distribuée du contrôle. Dès que le maillage est quelconque, un contrôle incluant de manière explicite d'autres informations (entre autres le routage des messages) doit être utilisé (le lecteur intéressé pourra comparer, en ce qui concerne l'exclusion mutuelle, les trois algorithmes suivants **[MAR 85b, HPR 86, RIA 83]** qui résolvent le problème dans un réseau dont le maillage est respectivement en anneau, quelconque et complet).

Le dernier aspect mis en évidence concerne la considération systématique des propriétés des objets pour les distribuer : le caractère monotone non-décroissant des compteurs est fondamental dans les solutions présentées. Outre leur mise en œuvre, ces propriétés en facilitent généralement la preuve **[RAY 86]**.

Bibliographie

[AHV 83] André F., Herman D., Verjus J.P., "*Synchronisation de Programmes Parallèles*", Dunod, 1983, 138 p.

[BOC 79] Bochmann G., "*Distributed Synchronisation and Regularity*", Computer Networks, n° 3, 1979, pp. 36-43.

[CHL 85] Chandy K.M., Lamport L., "*Distributed Snapshots : Determining Global States of Distributed Systems*", ACM TOCS, Vol. 3,n° 1 , Février 1985, pp. 63-75.

[HOA 78] Hoare C.A.R., "*Communicating Sequential Processes*", Comm. ACM, Vol. 21,n° 8, Août 1978, pp. 666-677.

[HPR 86] Helary J.M., Plouzeau N., Raynal M., "*A Distributed Algorithm for Mutual Exclusion in an Arbitrary Network*", Publication interne IRISA n° 281, Janvier 1986, 15p.

[LAM 78] Lamport L., "*Time, Clocks, and the Ordering of Events in a Distributed System*", Comm. ACM, Vol. 21, n° 7, Juillet 1978, pp. 558-565.

[LAM 84] Lamport L., "*Solved Problems, Unsolved Problems and Non-Problems in Concurrency*", Proc. of 3rd ACM Conf. on PODC, (1984), Reprinted in ACM Op. Systems Review, Vol. 19, n° 4, Octobre 1985, pp. 34-44.

[LEL 77] Lelann G., "*Distributed systems : Towards a formal Approach*", IFIP Congress, Toronto, Août 1977, pp. 155-160.

[MAR 85a] Martin A.J., "*The Probe : an Addition to Communication Primitives*", Inf. Proc. Letters, Vol. 20, 1985, pp. 125-130.

[MAR 85b] Martin A.J., "*Distributed Mutual Exclusion on a Ring of Processes*", Science of Computer Programming, Vol. 5, 1985, pp. 265-276.

[MIS 83] Misra J., "*Detecting Termination of Distributed Computation Using Markers*", Proc. of the 2d annual ACM Symposium on Principles of DC, Août 1983, pp. 290-294.

[RAR 85] Raynal M., Rubino G., "*Détecter la Perte de Jetons et les Régénerer sur une Structure en Anneau*", Rapport de Recherche INRIA #428, Juillet 1985, 24 p.

[RIA 83] Ricart G., Agrawala A.K., réponse à "*On Mutual Exclusion in Computer Networks*" de Carvalho et Roucairol, Comm. ACM, Vol 26,n° 2, Février 1983, pp. 147-148.

[ROV 77] Robert P., Verjus J. P., "*Towards Autonomous Description of Synchronisation Modules*", Proc. IFIP Congress, North Holland, 1977, pp. 981-986.

[RAY 84] Raynal M., "*Algorithmique du Parallélisme : le problème de l'Exclusion Mutuelle*", Dunod, 1984, 164 p.

[RAY 85] Raynal M., "*Algorithmes Distribués et Protocoles*", Eyrolles, 1985, 142p.

[RAY 86] Raynal M., "*A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks*", Inf. Processing Letters, 1986, à paraître.

[STE 76] Stenning W., "*A Data Transfer Protocol*", Computer Networks, Vol. 1 (1976), pp. 99-110.

[VER 83] Verjus J.P., "*Synchronisation in Distributed Systems : an informal introduction*", In Distributed Computing Systems, Parker Y. & Verjus J.P. Eds, Academic Press, 1983.

[VET 86] Verjus J.P., Thoraval R., "*Dérivation d'Algorithmes Distrubués d'Arbitrage*", TSI, Vol. 5, n° 1, Février 1986.

PI 284 **An overview of the GOTHIC Distributed Operating System**
Jean – Pierre Banatre, Michel Banatre, Florimond Ployette – 24 pages ; Janvier 86.

PI 285 **Optimal sensor location for detecting changes in dynamical behavior**
Michèle Basseville, Albert Benveniste, Georges Moustakides, Anne Rougée – 32 pages ; Février 86.

PI 286 **La tolérance aux fautes dans un système temps – réel à contraintes strictes**
Maryline Silly – 32 pages ; Février 86.

PI 287 **A new statistical approach for the automatic segmentation of continuous speech signals**
Régine André – Obrecht – 38 pages ; Mars 86.

PI 288 **Synthèse sur les réseaux locaux temps – réel**
Philippe Belmans – 40 pages ; Mars 86.

PI 289 **Calcul distribué d'un extrémum et du routage associé dans un réseau quelconque**
Jean – Michel Hélary, Aomar Maddi, Michel Raynal – 36 pages ; Mars 86.

PI 290 **A new matrix multiplication systolic array**
Patrice Quinton, Brigitte Joinnault, Pierrick Gachet – 12 pages ; Avril 86.

PI 291 **Une introduction à quelques techniques du contrôle distribué à travers un exemple**
Noël Plouzeau, Michel Raynal, Jean-Pierre Verjus – 22 pages ; Avril 86.

Noël PLOUZEAU

Michel RAYNAL

Jean – Pierre VERJUS

UNE INTRODUCTION A QUELQUES TECHNIQUES DU CONTROLE DISTRIBUE A TRAVERS UN EXEMPLE

Publication interne
n° 291

Avril 1986