Huitième partie

Processus communicants – CSP/Ada

Plan

Processus communicants

Synchronisation obtenue via des primitives de communication bloquantes : envoi (bloquant) de messages / réception bloquante de messages

- Communicating Sequential Processes (CSP) / Calculus of Communicating Systems (CCS) / π-calcul
- Ada

Les principes détaillés des échanges et leur utilisation pour développer des applications sont vus dans le module « intergiciels ». On ne s'intéresse ici qu'à la synchronisation.

Quelle synchronisation ?

Réception

Réception bloquante : attendre un message

Émission

- Émission non bloquante ou asynchrone
- Émission bloquante ou synchrone : bloque jusqu'à la réception du message = rendez-vous élémentaire entre l'activité émettrice et l'activité destinataire
- Rendez-vous étendu : bloquant jusqu'à réception + réaction + réponse $\approx$ appel de procédure

- Émission asynchrone $\Rightarrow$ buffers (messages émis non reçus)
- Synchrone $\Rightarrow$ 1 case maximum

Désignation du destinataire et de l'émetteur

Nommage

- Direct : désignation de l'activité émettrice/destinataire

```
SEND message TO processName
WAIT message FROM processName
```

- Indirect : désignation d'une boîte à lettres ou d'un canal de communication

```
SEND message TO channel
WAIT message FROM channel
```

Alternative

Alternative en émission ou en réception = choix parmi un ensemble de communications possibles :

```
WAIT msg FROM channel1 OR channel2
(SEND msg1 TO pid1) OR (SEND msg2 TO pid2)
(WAIT msg1 FROM channel1) OR (SEND msg2 TO channel2)
```

- Si aucun choix n'est faisable $\Rightarrow$ attendre
- Si un seul des choix est faisable $\Rightarrow$ le faire
- Si plusieurs choix sont faisables $\Rightarrow$ sélection non-déterministe (arbitraire)

Multiplicité

$1-1$

Désignation de l'activité : 1 émetteur / 1 récepteur désignés

$n-1$

Canal réservé en lecture (consommation) : envoi par n'importe quelle activité ; réception par une seule, propriétaire du canal

$n-m$

Canal avec envoi par n'importe qui, réception par n'importe qui :

- pas de duplication : un seul destinataire consomme le message
- duplication à tous les destinataires : $\approx m$ fois $n-1$

Complexe et coûteux à mettre en œuvre, nécessite une synchronisation globale entre tous les récepteurs

Divers

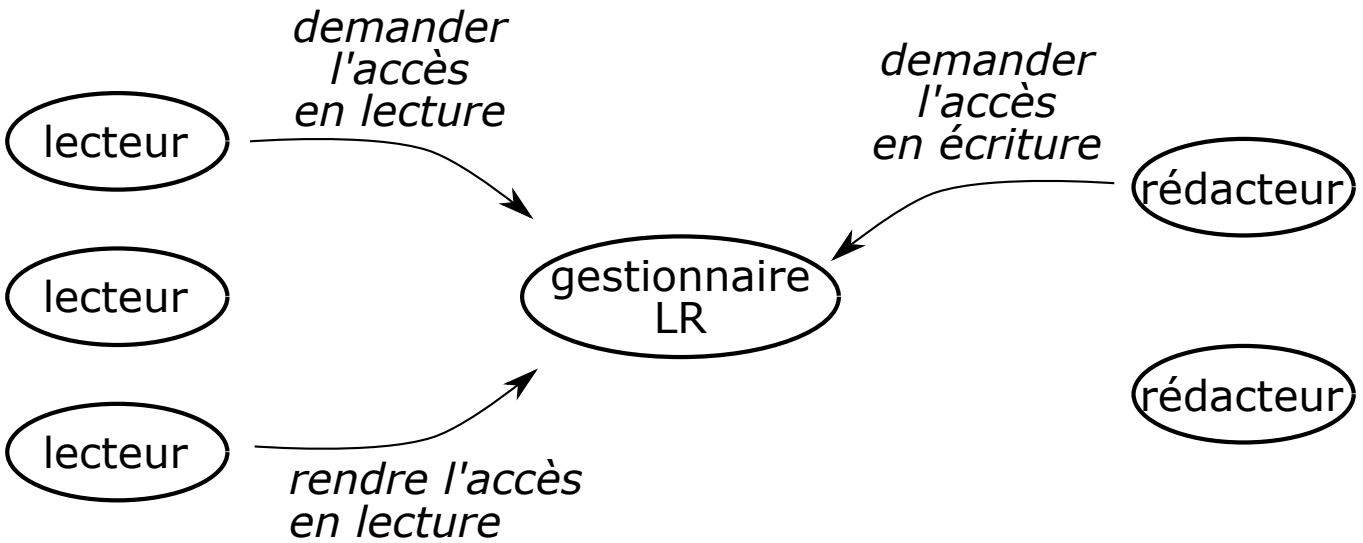
- Asynchrone $\Rightarrow$ perte de messages ? (buffer plein par exemple)
- Construction d'une émission bloquante (rendez-vous) en cas d'émission non bloquante :

```
   (SENT m TO ch; WAIT _ FROM ack)
|| (WAIT m FROM ch; SENT _ TO ack)
```

- Construction d'une émission non bloquante à partir d'émission bloquante : introduire une boîte intermédiaire qui accepte immédiatement tout message et le stocke dans une file.

Architecture

La résolution des problèmes de synchronisation classiques (producteurs/consommateurs...) ne se fait plus en synchronisant directement les processus via des données partagées, mais indirectement via une activité de synchronisation.



Intérêt

+ découplage entre les activités clientes : l'interface partagée est celle de l'activité de synchronisation
+ réalisation centralisée et répartie
+ transfert explicite d'information : traçage
+ pas de données partagées $\Rightarrow$ pas de protection nécessaire
+ contrôle fin des interactions
+ schéma naturel côté client : question/réponse = appel de fonction
– multiples recopies (mais optimisations possibles)
– parallélisation du service : au cas par cas

Plan

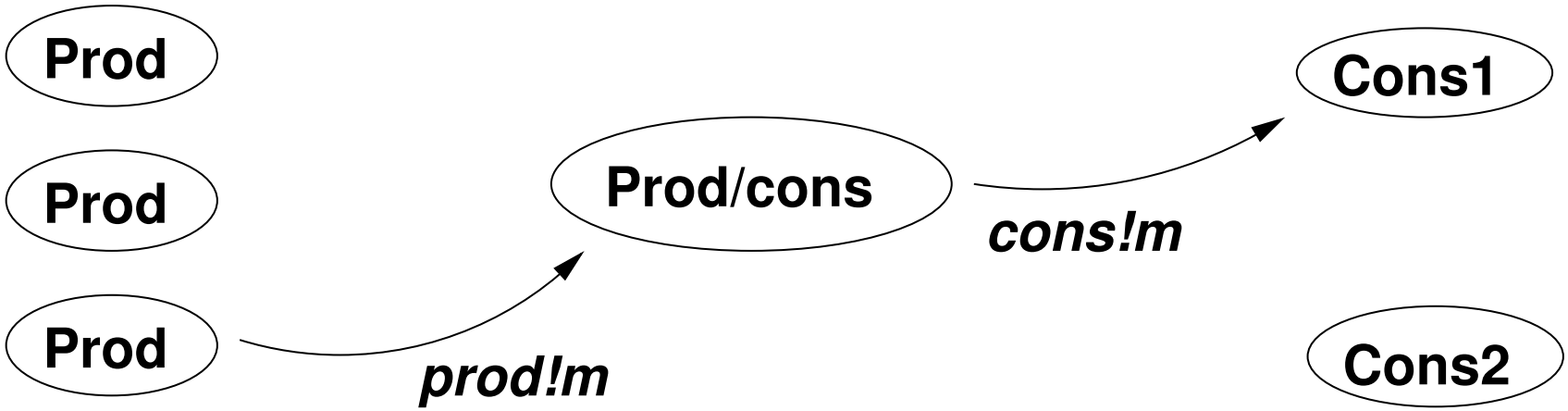
CSP – Communicating Sequential Processes

- Émission bloquante
- Désignation de canaux de communications
- Existence statique des canaux (pas de création dynamique)
- Alternative en réception seulement ou mixte (variantes)

Syntaxe

- envoi d'un message sur le canal c : $c!msg$ (msg est une valeur)
- réception d'un message depuis le canal c : $c?msg$ (msg est une variable)
- alternative en réception : $c_1?m_1 \Box c_2?m_2$
- alternative en émission : $c_1!m_1 \Box c_2!m_2$
- alternative mixte : $c_1!m_1 \Box c_2?m_2$
- réception gardée : $guard \rightarrow c?m$

Architecture simpliste



- un canal pour les demandes de dépôt (sens producteur $\rightarrow$ tampon)
- un canal pour les transmissions des valeurs déposées (sens tampon $\rightarrow$ consommateur)

Producteurs/consommateurs

Producteur

```
Processus producteur
  boucle
    ...
    msg := ...
    prod!msg
    ...
  finboucle
```

Consommateur

```
Processus consommateur
  boucle
    ...
    cons?msg
    utiliser msg
    ...
  finboucle
```

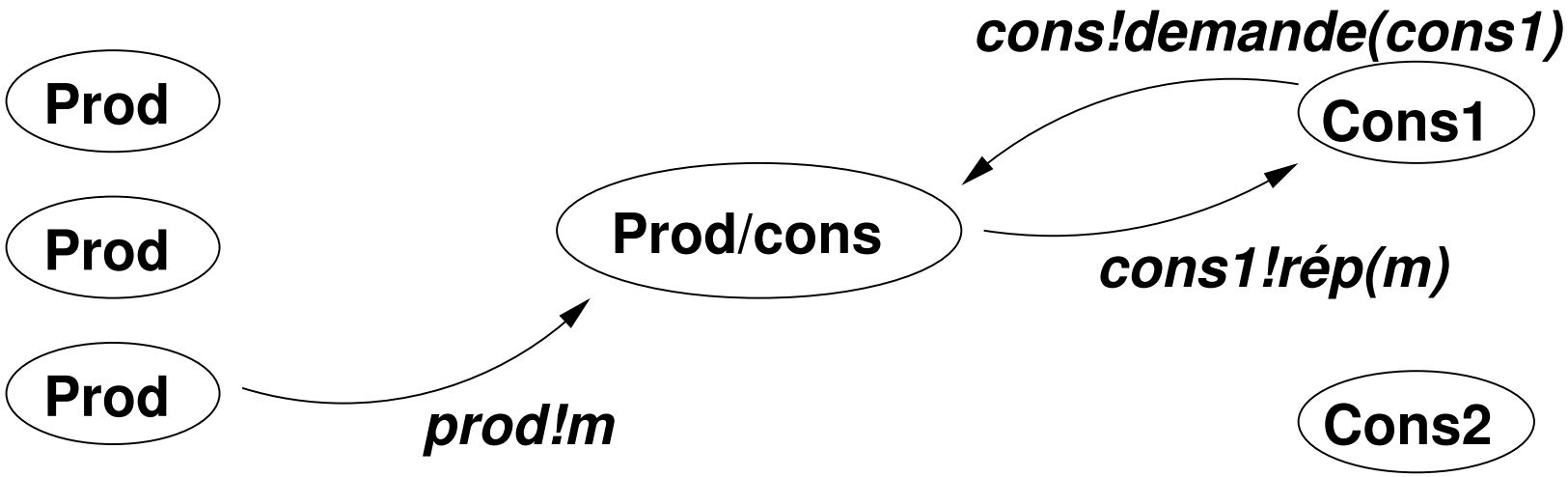
Producteurs/consommateurs

Activité de synchronisation

```
Processus prodcons
  var
    int nbocc = 0;
    message m;
    file<messages> tampon;
  boucle
     nbocc < N → prod?m; nbocc++; tampon.ranger(m);
  □
     nbocc > 0 → m := tampon.extraire; cons!m; nbocc--;
  finboucle
```

- Nécessite l'alternative mixte (plus complexe que l'alternative en réception)
- Nécessite la réception multiple (plusieurs activités attendent sur le canal cons)

Architecture effective



- Un canal pour les demandes de dépôt
- Un canal pour les demandes de retrait
- Un canal par activité demandant le retrait (pour la réponse à celle-ci)

Producteurs/consommateurs

Producteur

```
Processus producteur
  boucle
    ...
    msg := ...
    prod!msg
    ...
  finboucle
```

Consommateur

```
Processus consommateur1
Canal moi
  boucle
    ...
    cons!demande(moi);
    moi?rép(msg);
    utiliser msg
  finboucle
```

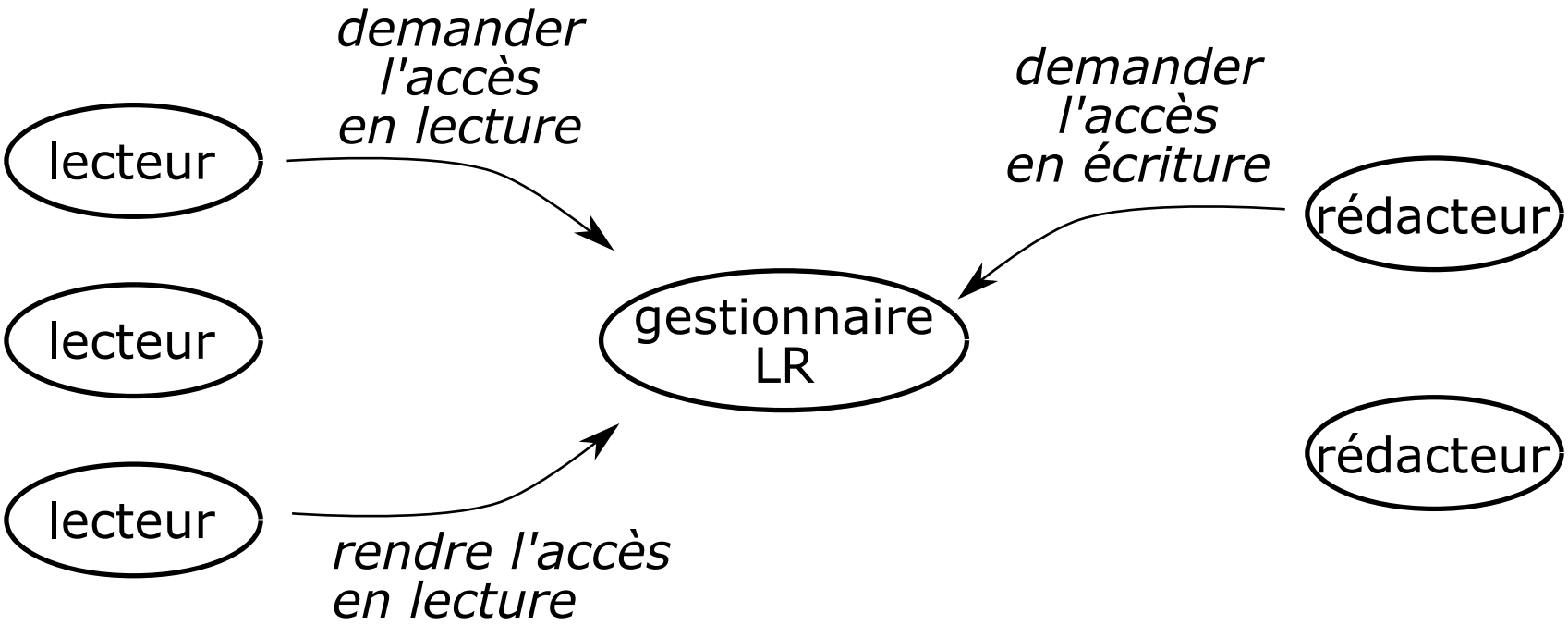
Producteurs/consommateurs

Processus de synchronisation

```
Processus pc
  var
    int nbocc = 0;
    message m;
    canal c;
  boucle
     nbocc < N → prod?dépôt(m); nbocc++; ranger m
  □
     nbocc > 0 → cons?demande(c); m := ...;
                 c!rép(m); nbocc--;
  finboucle
```

Nécessite des variables « canal », passables en paramètre.

Lecteurs/rédacteurs



- Un canal pour chaque type de requête
- Émission bloquante $\Rightarrow$ n'accepter un message (une requête) uniquement si l'état l'autorise

Lecteurs/rédacteurs

Utilisateur

```
Processus utilisateur
  boucle
    DL!_;  // demander lecture
    ...
    TL!_;  // terminer lecture
    ...
    DE!_;  // demander écriture
    ...
    TE!_;  // terminer écriture
  finboucle
```

Lecteurs/rédacteurs

Processus de synchronisation

```
Processus SynchroLR
  var
    int nblec = 0;
    boolean ecr = false;
  boucle
     nblec = 0 ∧ ¬ecr → DE?_; ecr := true;
  □
     ¬ecr → DL?_; nblec++;
  □
     TE?_; ecr := false;
  □
     TL?_; nblec--;
  finboucle
```

Plan

Modèle Ada

Intérêt

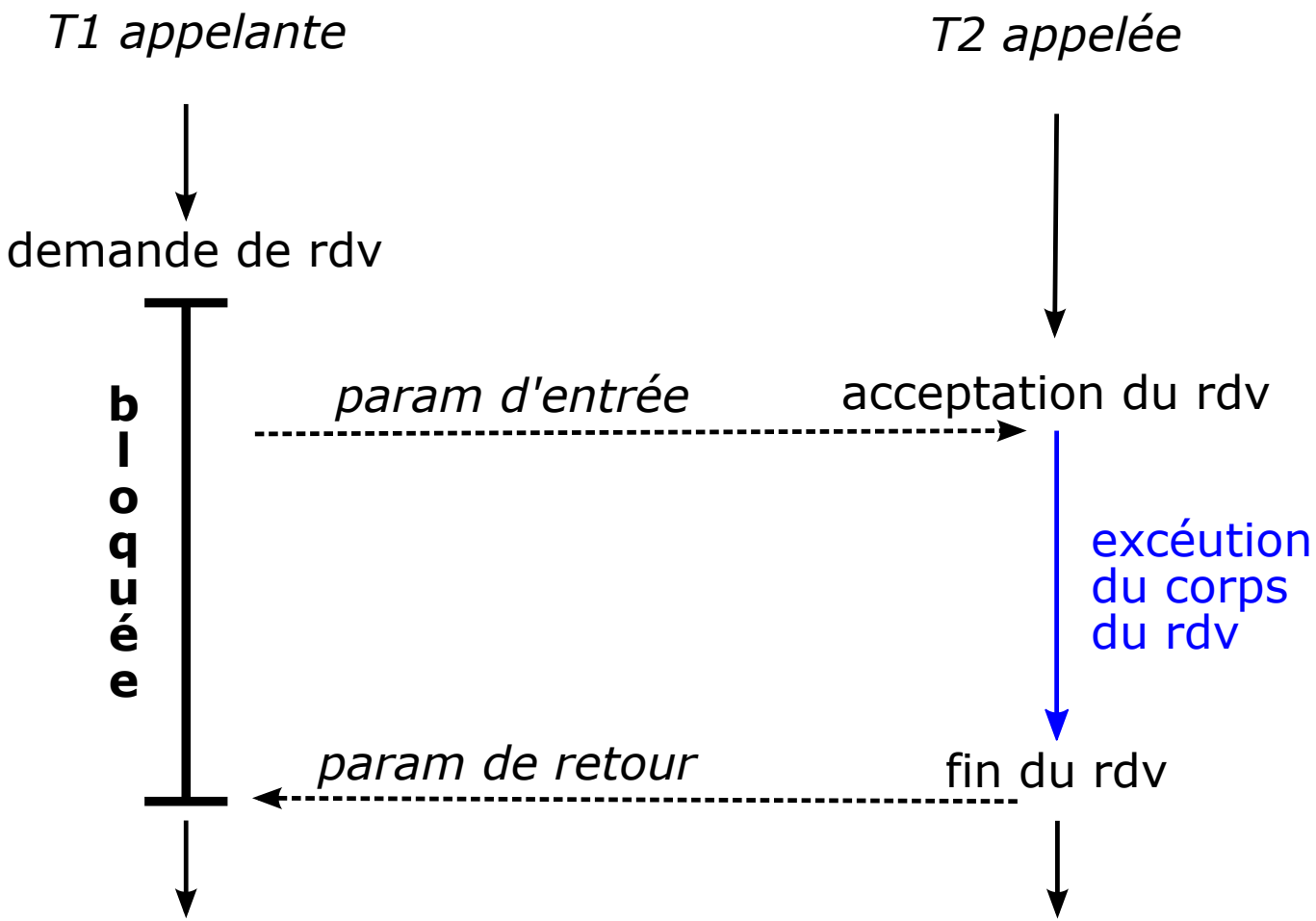
- Modèle adapté à la répartition, contrairement aux sémaphores ou aux moniteurs, intrinsèquement centralisés.
- Similaire au modèle client-serveur.
- Contrôle plus fin du moment où les interactions ont lieu.

Vocabulaire : tâche = activité

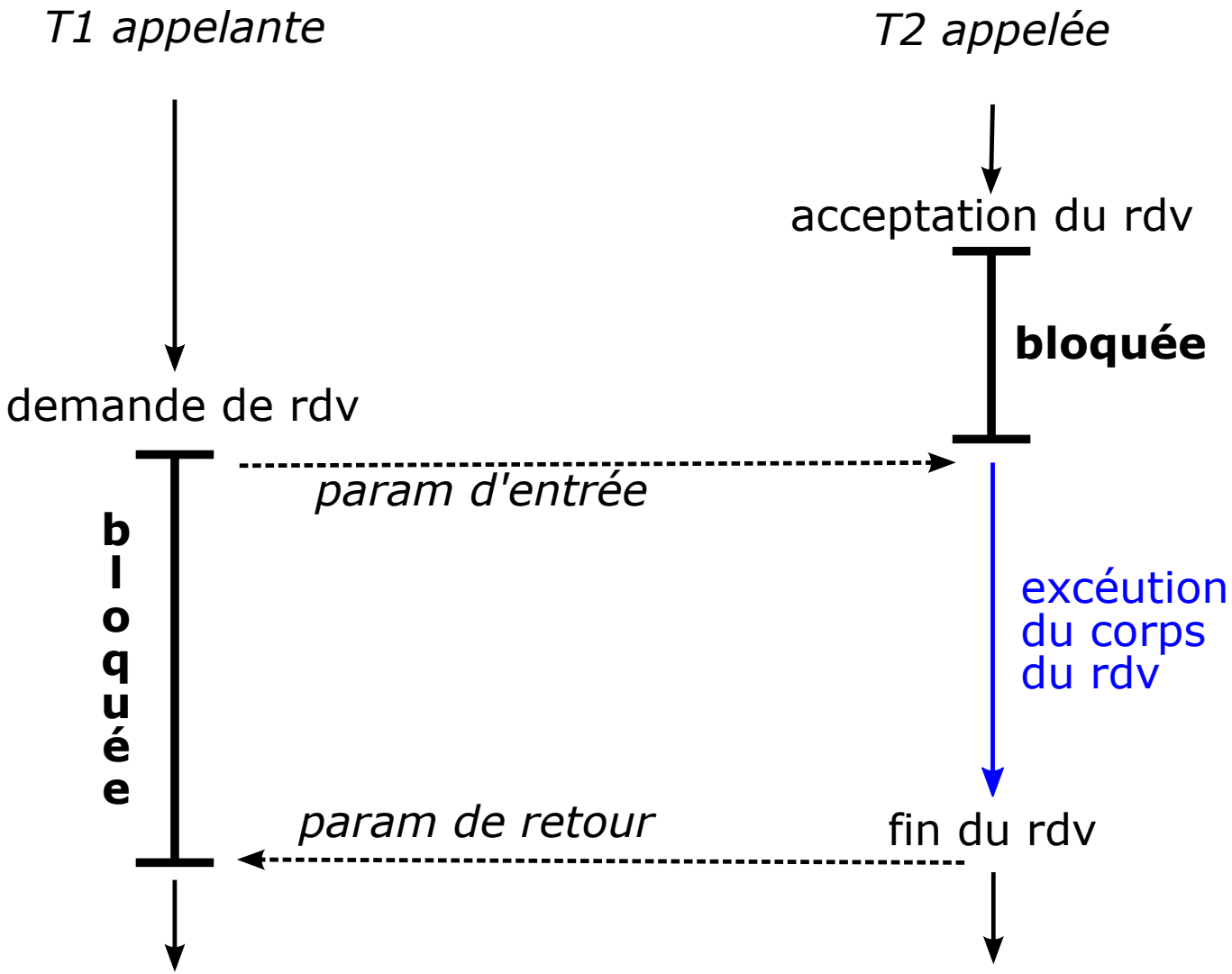
Principe du rendez-vous

- Une tâche possède des points d'entrée de rendez-vous.
- Une tâche peut :
  - demander un rendez-vous avec une autre tâche désignée explicitement ;
  - attendre un rendez-vous sur un (ou plusieurs) point(s) d'entrée.
- Un rendez-vous est dissymétrique : tâche appelante ou cliente vs tâche appelée ou serveur.
- Échanges de données :
  - lors du début du rdv, de l'appelant vers l'appelé ;
  - lors de la fin du rdv, de l'appelé vers l'appelant.

Rendez-vous – client en premier



Rendez-vous – serveur en premier



Principe du rendez-vous

- Si un client demande un rendez-vous alors que le serveur n'est pas prêt à l'accepter, le client se bloque en attente de l'acceptation.
- Si un serveur indique qu'il est prêt à accepter un rendez-vous et qu'il n'y a pas de demandeur, il se bloque.
- En outre, l'appelant est bloqué pendant l'exécution du corps du rendez-vous.

Important : il est impossible d'accepter/refuser un rendez-vous selon la valeur des paramètres.

Déclaration d'une tâche

Déclaration

```
task <nom> is
   { entry <point d'entrée> (<param formels>); }+
end
```

Exemple

```
task X is
    entry A;
    entry B (msg : in T);
    entry C (x : out T);
    entry D (a : in T1; b : out T2);
end X
```

Appel de rendez-vous

Appel de rdv

```
<nom tâche>.<point d'entrée> (<param effectifs>);
```

Similaire à un appel de procédure.

Exemple

```
X.A;
X.D(x,y);
```

Acceptation d'un rendez-vous

Acceptation

```
accept <point d'entrée> (<param formels>)
     [ do
         { <instructions> }+
       end <point d'entrée> ]
```

Exemple

```
accept D (a : in Natural; b : out Natural) do
     if a > 6 then
         b := a / 4;
     else
         b := a + 2;
     end if;
end D;
```

Acceptation parmi un ensemble

Alternative gardée

```
select
     when C1 =>
          accept E1 do
               ...
          end E1;
   or
     when C2 =>
          accept E2 do
               ...
          end E2;
   or
     ...
end select;
```

Producteurs/consommateurs

Déclaration du serveur

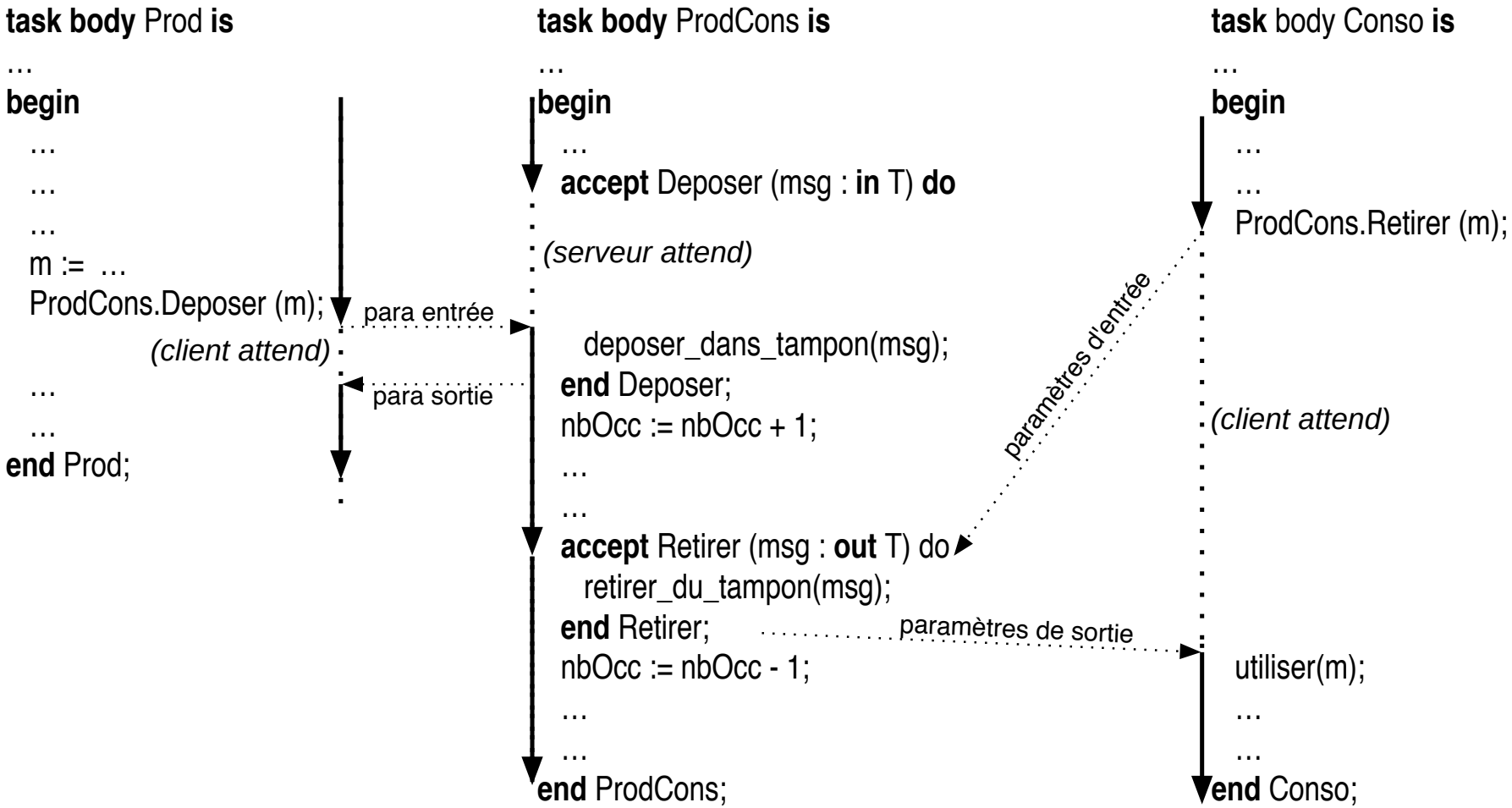
```
task ProdCons is
   entry Deposer (msg: in T);
   entry Retirer (msg: out T);
end ProdCons;
```

Client : utilisation

```
begin
   -- engendrer le message m1
   ProdCons.Deposer (m1);
   -- ...
   ProdCons.Retirer (m2);
   -- utiliser m2
end
```

```
task body ProdCons is
   Libre : integer := N;
begin
   loop
      select
         when Libre > 0 =>
            accept Deposer (msg : in T) do
               deposer_dans_tampon(msg);
            end Deposer;
            Libre := Libre - 1;
      or
         when Libre < N =>
            accept Retirer (msg : out T) do
               msg := retirer_du_tampon();
            end Retirer;
            Libre := Libre + 1;
      end select;
   end loop;
end ProdCons;
```

Producteurs/consommateurs – un exemple d'exécution



Remarques

- Les `accept` ne peuvent figurer que dans le corps des tâches.
- `accept` sans corps $\rightarrow$ synchronisation pure.
- Une file d'attente (FIFO) est associée à chaque entrée.
- `rdv'count` (attribut des entrées) donne le nombre de clients en attente sur une entrée donnée.
- La gestion et la prise en compte des appels diffèrent par rapport aux moniteurs :
  - la prise en compte d'un appel au service est déterminée par le serveur ;
  - plusieurs appels à un même service peuvent déclencher des traitements différents ;
  - le serveur peut être bloqué, tandis que des clients attendent.

Allocateur de ressources

Un système comporte des ressources critiques c'est-à-dire non partageables et non préemptibles, comme les pages mémoire. L'allocateur de ressources est un service qui permet à un processus d'acquérir par une seule action plusieurs ressources. On ne s'intéresse qu'à la synchronisation et on ne s'occupe pas de la gestion effective des identifiants de ressources.

Déclaration du serveur

```
task Allocateur is
   entry demander (nbDemandé: in natural;
                   id : out array of RessourceId);
   entry rendre (nbRendu: in natural;
                 id : in array of RessourceId);
end Allocateur;
```

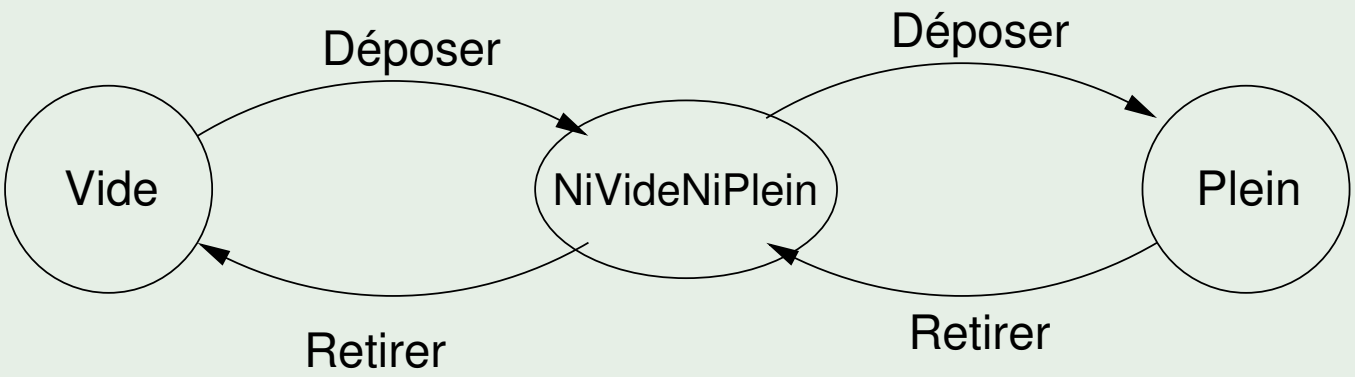
```
task body Allocateur is
   nbDispo : integer := N;
begin
   loop
      select
          accept Demander (nbDemandé : in natural) do
              while nbDemandé > nbDispo loop
                  accept Rendre (nbRendu : in natural) do
                      nbDispo := nbDispo + nbRendu;
                  end Rendre;
              end loop;
              nbDispo := nbDispo - nbDemandé;
          end Demander;
      or
          accept Rendre (nbRendu : in natural) do
              nbDispo := nbDispo + nbRendu;
          end Rendre;
      end select;
   end loop;
```

Méthodologie par machine à états

Construire un automate fini à états :

- identifier les états du système
- un état est caractérisé par les rendez-vous acceptables
- un rendez-vous accepté change (éventuellement) l'état

Producteurs/consommateurs à 2 cases



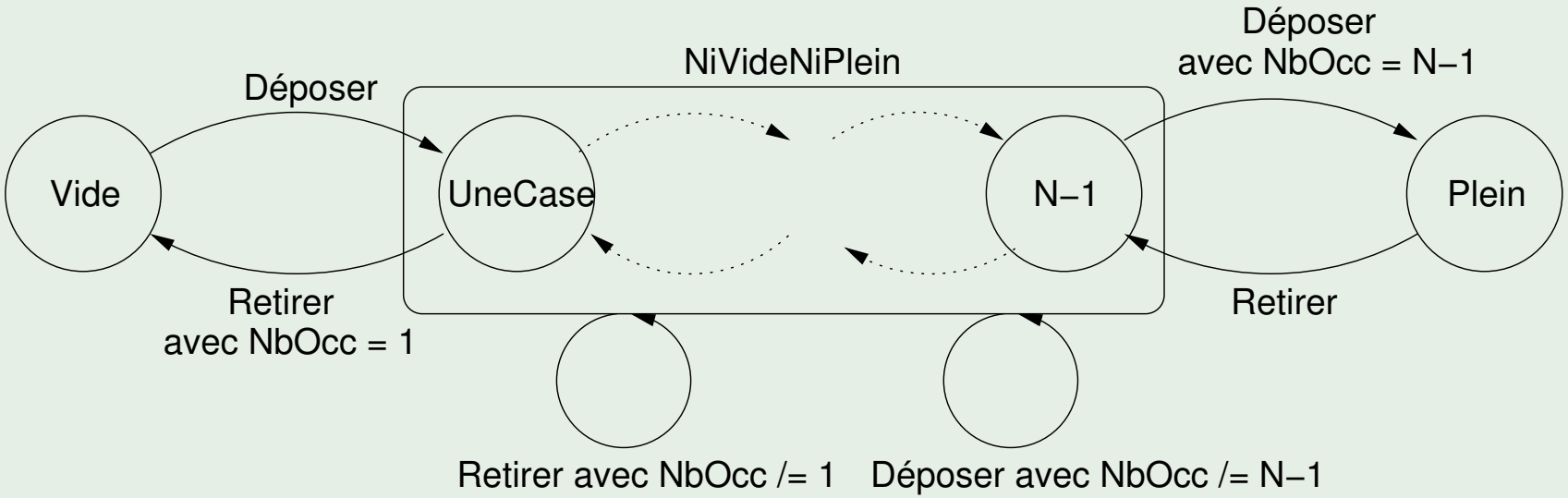
```
task body ProdCons is
   type Etat is (Vide, NiVideNiPlein, Plein);
   etat : Etat := Vide;
begin
   loop
      if etat = Vide then
            select
               accept Deposer (msg : in T) do
                  deposer_dans_tampon(msg);
               end Deposer;
               etat := NiVideNiPlein;
            end select;
      elsif etat = NiVideNiPlein then
            select
               accept Deposer (msg : in T) do
                  deposer_dans_tampon(msg);
               end Deposer;
               etat := Plein;
            or
               accept Retirer (msg : out T) do
                  msg := retirer_du_tampon();
               end Retirer;
               etat := Vide;
            end select;
```

```
      elsif etat = Plein then
            select
               accept Retirer (msg : out T) do
                  msg := retirer_du_tampon();
               end Retirer;
               etat := NiVideNiPlein;
            end select;
      end if;
   end loop;
end ProdCons;
```

Automate paramétré

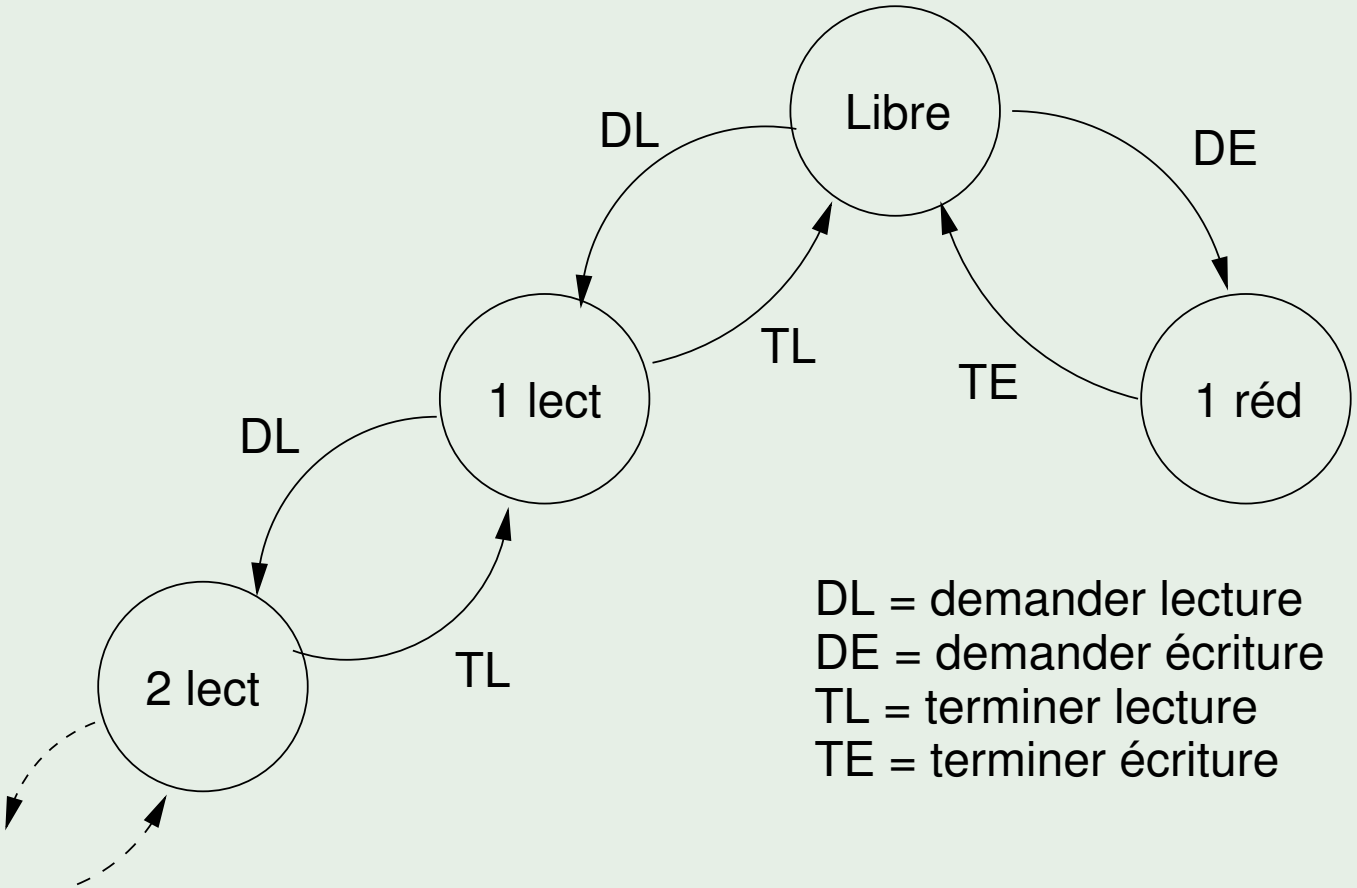
Représenter un *ensemble d'états* comme un unique état *paramétré*.
Les valeurs du paramètre différenciant les états de l'ensemble peuvent être utilisées pour étiqueter les transitions.

Producteurs/consommateurs à N cases



Lecteurs/rédacteurs

Lecteurs/rédacteurs



Lecteurs/rédacteurs prio rédacteurs

Lecteurs/rédacteurs prio rédacteurs