

# 1. COMMUNICATION PAR MESSAGE

## → Message synchrone (rendez-vous simple)

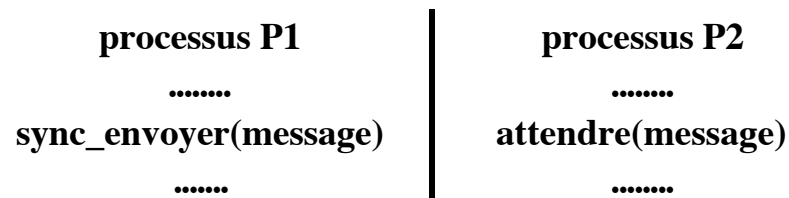
- l'émetteur attend que le récepteur ait lu le message
- le récepteur qui attend un message est bloqué jusqu'à son arrivée

### avantages

- émetteur et récepteur sont dans un état connu
- on peut implanter des styles de calcul concurrents par flot de données ou par calcul systolique

### inconvénients

- fort couplage entre les correspondants : communication 1 - 1



## → Message asynchrone (sans attente)

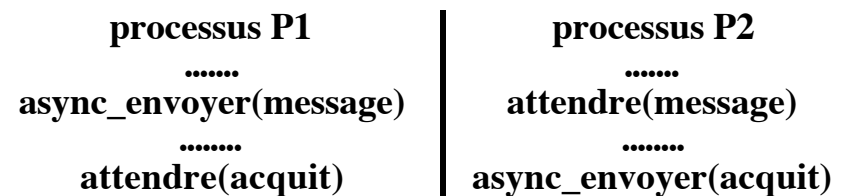
- l'émetteur n'est pas bloqué en attente de la réception
- le récepteur a un rythme autonome de réception, avec deux modes :
  - API de réception bloquante si pas de message
  - API de réception non bloquante, avec un témoin de réception
- schéma producteur consommateur

### avantage

- indépendance temporelle des correspondants : communication N - 1

### inconvénients

- pas d'acquittement implicite
- pas de relation entre les états de l'émetteur et du récepteur => difficultés en cas d'erreur



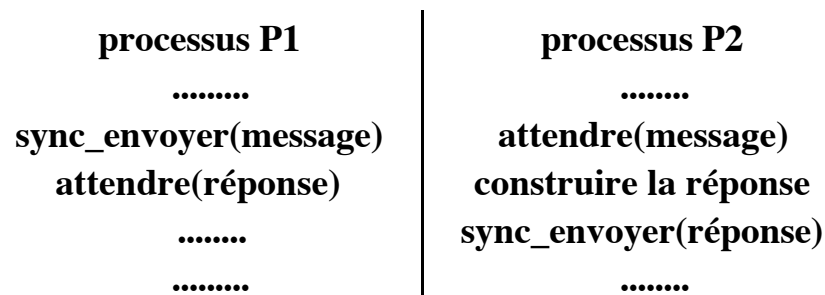
## → Invocation à distance

(rendez-vous étendu)

- demande l'exécution d'une procédure à un autre processus
- est accompagnée d'un passage de messages entre processus
- comme pour l'appel de procédure, l'appelant "attend" la réponse de l'appelé (la réponse est facultative) , mais ici appelant et appelé ne sont pas le même processus
- son écriture ressemble à un appel de procédure

avantage

- sémantique claire; correspondants dans des états connus  
communication N - 1  
paradigme clients -serveur
- à comparer à l'appel de procédure distante (APD) ou "Remote Procedure Call" (RPC)



## → Analogies

- poster une lettre <<◇>> message asynchrone •
- envoyer un fax <<◇>> message synchrone •
- téléphoner <<◇>> invocation à distance avec réponse •

## → Implantation

Messages synchrones

- centralisé par mémoire commune
  - réparti par réseau synchrone
- exemples CSP, Occam, peu répandu

Messages asynchrones :

- centralisé : utilisation de tampons et du schéma producteur consommateur
- réparti : interface des réseaux (Internet, OSI, SNA), des systèmes répartis classiques (Chorus, Mach, Amoeba) : datagrammes, sockets,...

Invocation à distance

- appel de procedure local ou distant (RPC)

**COMPARAISON**

→ **appel de procédure à distance**  
(APD, ou "remote procédure call" RPC)

- la procédure appelée est sur un autre site
- pour permettre l'exécution, on met en place une communication entre site du processus appelant et site de la procédure appelée; on installe les ressources nécessaires sur les deux sites et on assure la gestion de la tolérance aux pannes de communication ou d'exécution.
- on est au niveau physique, pas au niveau logique (voir cours réseau : souche client, souche serveur, passage des arguments et des résultats, gestion de l'exécution sur le site serveur par un processus subrogé.

Voir normes DCE, CORBA, ...).

INVOCATION A DISTANCE	APPEL DE PROCEDURE DISTANTE
construction <i>langage</i> niveau logique appel d'un processus	construction <i>réseaux</i> niveau physique réparti appel d'un site
<i>appel exécuté par un autre processus, le processus appelé, avec son environnement (dont sa pile)</i>	<i>appel exécuté par un processus subrogé ou courtier ("proxy") créé sur un autre site, le site appelé</i>
<i>Appel exécuté quand le processus appelé le permet; il peut y avoir des conditions de synchronisation; l'appel peut n'être jamais exécuté</i>	<i>Appel exécuté dès que possible (délai de mise en place des ressources) comme une procédure; en général, il n'y a pas de condition d'exécution</i>
les deux processus sont en <i>rendez-vous</i>	appel de procédure entre des espaces d'adressage différents

## 2. LE RENDEZ-VOUS ENTRE TÂCHES ADA

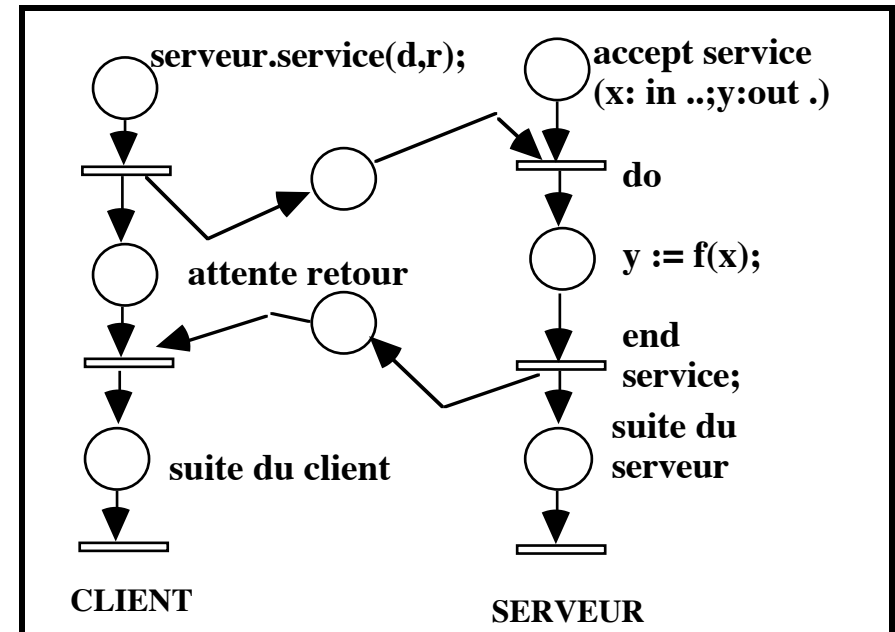
- rendez-vous asymétrique  
entre client appelant et serveur appelé
  - le serveur est connu des clients;  
mais le serveur ne connaît pas les clients.
- coté serveur appelé (tâche appelée)
- points d'appel déclarés par entry
  - acceptation d'un rendez-vous par l'appelé :  
instruction accept
  - rendez-vous étendu : appel, traitement, retour
  - passage de paramètres  
données in et résultats out
  - appel gardé par une condition chez l'appelé :  
when condition => accept  
la condition ne peut contenir ni le nom de  
l'appelant ni de paramètre de l'appel
  - fin d'un rendez-vous initié par l'appelé  
accept ... do ... end; le ";" entraîne le retour des  
résultats puis le réveil de l'appelant  
do...end permet l'exécution avec client bloqué

- le rendez-vous peut être traité par plusieurs entrées  
successivement (deux ou plus) :  
instruction requeue  
sans libérer l'appelant au moment du requeue.
  - attente sélective (1 parmi P choix possibles) :  
clause select
    - unicité de l'accept : un seul rendez-vous quel que  
soit le nombre d'appels consultés par select
    - indéterminisme : rendez-vous avec l'un quelconque  
des possibles
  - attente sélective avec choix de terminaison  
clause terminate  
avec attente limitée clause delay [until]  
avec rendez-vous conditionnel clause else
  - notion de famille d'entrées
- coté client appelant
- appel procédural avec :  
nom tâche.nom entrée(paramètres effectifs)
  - appel avec rendez-vous conditionnel :  
select <appel>; else .... end select;
  - appel avec attente limitée :  
select <appel>; or delay [until].... end select;

# SEMANTIQUE DE L'INVOCATION A DISTANCE

(RENDEZ-VOUS ETENDU)

CLIENT APPELANT	SERVEUR APPELÉ
<b>d : demande;</b> <b>r : réponse;</b> .....	<b>entry service</b> (x: <u>in</u> demande; y : <u>out</u> réponse);
<b>d := préparer_demande;</b>	.....
<b>serveur.service(d,r);</b> -- invocation	<b>accept service</b> (x: <u>in</u> demande; y : <u>out</u> réponse);
-- bloqué en attente de retour de la réponse r	<b>do</b> y := f(x); -- élaboration -- de la réponse <b>end service;</b>
<b>suite_du_client</b> .....	<b>suite_du_serveur</b> ....



Réseau de Petri de l'invocation à distance

## PROCESSUS COMMUNIQUANT PAR INVOCATION À DISTANCE

Le processus principal crée deux fils;  
chaque processus fils lance une impression

with Text\_Io; use Text\_Io;

-- paquetage standard d'impression

procedure Main is-

```
task type TT is
  entry Start( Id : Integer); -- interface
end TT;                          -- de rendez-vous
```

Fils1, Fils2 : TT;-- création de deux fils serveurs

```
task body TT is -- corps des fils
begin
  accept Start( Id : Integer) do
    Put_Line("voici le fils"& Integer'Image(Id));
  end Start;
end TT; -- fin du fils
```

begin -- début du père, et activation des fils

Put\_Line("Voici le père");

Fils1.Start(1);-- appel et réveil du fils1

Fils2.Start(2);-- appel et réveil du fils2

Put\_Line("Fin du programme");

end Main;

-- Ada est normalisé ISO/IEC 8652:1995(E)

## SERVEUR COHÉRENT DE DONNÉES PARTAGÉES

Procedure Main is

```
task Compte is
  entry Ajouter(X : in Integer);
end Compte;
```

```
task body Compte is
  CompteClient : Integer := 0 ; -- simule un fichier,
  Y : Integer;
begin loop
  accept Ajouter(X : in Integer) do
    Y := CompteClient ;
    Y := Y + X;
    CompteClient := Y;
  end Ajouter;
end loop; end Compte;
-- sert les clients l'un après l'autre, un par un
```

task type Processus; task Processus is

A : Integer := 1;

begin

for I in 1..16 loop

ActionHorsSectionCritiqueEtCalculDe(A);

Compte.Ajouter(A);

end loop;

end Processus;

Application : array(1..N) of Processus;

begin null; end Main;

**-- TÂCHE SERVEUR DE CONTRÔLE D'ACCÈS EN  
EXCLUSION MUTUELLE**

**procedure Main is**

```
task Mutex is
  entry P; -- contrôle l'entrée de section critique
  entry V; -- contrôle de sortie de section critique
end Mutex ;
```

```
task body Mutex is
begin
  loop
    -- séquence infinie de rendez-vous successifs
    accept P; -- autorise une entrée
    accept V; -- puis une sortie de section critique
  end loop;
end Mutex ;
-- implémente un sémaphore d'exclusion mutuelle
```

**task type Proc; Appli : array(1..N) of Proc;**

**task body Proc is**

```
begin
  -- actions hors section critique
  Mutex.P; -- demande de rendez-vous avec Mutex
  Section_Critique_du_Processus;
  Mutex.V;
end Proc;
```

**begin null; end Main;**

**TÂCHE SERVEUR DE SIGNAL**

**with Ada.Text\_IO; use Ada.Text\_IO;**

**procedure Main is**

```
task Signal is
  entry Attendre; -- contrôle l'attente de signal
  entry Envoyer; -- contrôle l'envoi de signal
end Signal ;
```

```
task body Signal is
begin
  loop
    -- séquence infinie de rendez-vous successifs
    accept Envoyer; -- autorise une exécution
    accept Attendre; -- qui est attendue ici
  end loop;
end Signal ;
-- implémente un sémaphore d'exclusion mutuelle
```

**task Proc1; task Proc2; -- X : nb envois; Y : nb reçus**

**task body Proc1 is X : Integer := 0;**

```
begin loop
  X := X + 1; Put_Line("X =" & Integer'Image(X));
  Signal.Envoyer;
end loop; end Proc1;
```

**task body Proc2 is Y : Integer := 0;**

```
begin loop
  Signal.Attendre;
  Y := Y + 1; Put_Line("Y =" & Integer'Image(Y));
end loop; end Proc2;
```

**begin null; end Main;**

**--SERVEUR D'ACCÈS À UNE DONNÉE PARTAGÉE**

**Procedure Main is****--SERVEUR D'ACCÈS À UNE DONNÉE PARTAGÉE**

```

task Compte is
  entry Ecrire(X : in Integer);
  entry Lire(X : out Integer);
  entry Ajouter(X : in Integer);
end Compte;

```

```

task body Compte is
  CompteClient : Integer := 0; -- simule un fichier
  Y : Integer;
begin -- attend l'initialisation avant lecture ou ajout
  accept Ecrire(X : in Integer) do
    CompteClient := X; end Ecrire;
  loop
    select
      accept Ecrire(X : in Integer) do
        CompteClient := X; end Ecrire;
      or
      accept Lire(X : out Integer) do
        X := CompteClient ; end Lire;
      or
      accept Ajouter(X : in Integer) do
        Y := CompteClient ;
        Y := Y + X;
        CompteClient := Y;
      end Ajouter;
    end select;
  end loop;
end Compte;
-- sert les clients l'un après l'autre, un par un

```

```

task type Processus;

```

```

with Ada.Text_IO; use Ada.Text_IO;

```

```

task Processus is

```

```

  A : Integer := 1;

```

```

begin

```

```

  CalculDe(A);

```

```

  Compte.Ecrire(A);

```

```

  for I in 1..16 loop

```

```

    NouveauCalculDe(A);

```

```

    Compte.Ajouter(A);

```

```

  end loop;

```

```

  Compte.Lire(A);

```

```

  Put_Line("A =" & Integer'Image(A));

```

```

end Processus;

```

```

  Application : array(1..N) of Processus;

```

```

begin null; end Main;

```

**-- TÂCHE SERVEUR POUR PARADIGME  
PRODUCTEUR-CONSOMMATEUR**

**procedure Prod\_Cons is**

**N : constant integer := 4 ;**

```

task type T_Stock is    -- tâche serveur
  entry Deposer(M : in Message);
  entry Retirer(L : out Message);
end T_Stock ;

```

**Tampon : T\_Stock ;**

**task type Producteur;**

**task type Consommateur;**

**task body Producteur is**

**Envoi : Message;**

**begin**

**Envoi := Préparation\_du\_message;**

**Tampon.Deposer(Envoi) ;**

**end Producteur;**

**task body Consommateur is**

**Retrait : Message;**

**begin**

**Tampon.Retirer(Retrait);**

**exploitation\_du\_message(Retrait);**

**end Consommateur;**

**PROD : array (1 .. 15) of Producteur;**

**CONS : array (1 .. 10) of Consommateur;**

**task body T\_Stock is** -- tâche serveur

**Casier : array(0 .. N-1) of Message;**

**I, J : Integer range 0 .. N-1 := 0;**

**Nombre : Integer range 0 .. N := 0;**

**begin**

**loop**

**select**

**when Nombre < N =>**

**accept Deposer(M : in Message) do**

**Casier(I) := M; end Deposer;**

**I := (I + 1) mod N;**

**Nombre := Nombre + 1;**

**or**

**when Nombre > 0 =>**

**accept Retirer(L : out Message) do**

**L := Casier(J); end Retirer;**

**J := (J + 1) mod N;**

**Nombre := Nombre - 1;**

**or**

**terminate;**

**end select;**

**end loop;**

**end T\_Stock ;**

**begin null;**

**end Prod\_Cons;**

**-- TÂCHE SERVEUR DE  
RENDEZ-VOUS SYMÉTRIQUE ANONYME**

**procedure Main is**

```
task Canal_Sym is -- declaration d'interface
  entry Emettre(Emis : in Message);
  entry Recevoir(Recu : out Message);
end Canal_Sym ;
```

```
task body Canal_Sym is
begin
  loop
    select
      accept Emettre(Emis : in Message) do
        accept Recevoir(Recu : out Message) do
          Recu := Emis; -- communication directe
        end Recevoir;
      end Emettre;
    or
      terminate;
    end select;
  end loop
end Canal_Sym ;
```

```
task type Emetteur; task type Recepteur;
M : array (1..15) of Emetteur;
R : array (1..10) of Recepteur;
```

```
task body Emetteur is
  Envoi : Message;
begin
  Envoi := Préparation_du_message;
  Canal_Sym.Emettre(Envoi) ;
end Emetteur;
```

```
task body Recepteur is
  Retrait : Message;
begin
  Canal_Sym.Recevoir(Retrait);
  exploitation_du_message(Retrait);
end Recepteur ;
```

```
beginnull; end Main ;
-- chaque Emetteur communique avec n'importe
-- quel Recepteur
-- la tâche Canal_Sym les met en communication
-- anonyme directe
-- remarque : ce programme ne se termine pas.
-- pourquoi?
-- réponse : 15 appels à Emettre, 10 à Recevoir
```

**REPAS DES PHILOSOPHES ASSIS**

**procedure Application is**

```
task Chaise is
  entry Prendre;
  entry Rendre;
end Chaise;
```

```
task body Chaise is
  Libre : Integer := 4;
begin
  loop
    select
      when Libre>0 => accept Prendre;
      Libre := Libre-1;
    or
      accept Rendre; Libre := Libre + 1;
    end select
  end loop;
end Chaise;
```

```
task type RessourceCritique is
  entry Prendre;
  entry Rendre;
end Ressource;
```

```
task body RessourceCritique is
begin
  loop
    accept Prendre; accept Rendre;
  end loop;
end Ressource ; -- contrôle l'exclusion mutuelle
```

```
type Philo_Id is mod 5;
Baguette : array(Philo_Id) of RessourceCritique ;
next_id : Philo_Id := Philo_Id'first; -- ici vaut 0
function unique_id return Philo_Id is
begin
  next_id:= next_id + 1 ; --addition modulo N
  return next_id ;
end unique_id;
task type philo(X : Philo_Id := unique_id) ;
philosophe : array(Philo_Id ) of philo ;
task body philo is
begin
  loop
    Penser;
    Chaise.Prendre;
    Baguette(X).Prendre; Baguette(X+1).Prendre;
    Manger ;
    Chaise.Rendre;
    Baguette(X).Rendre; Baguette(X+1).Rendre;
  end loop ;
end philo ;
begin null; end Application ;
-- rappel + signifie : + modulo 5
```

**REPAS DES PHILOSOPHES**  
avec allocation globale

**procedure Application is**

**type Philo\_Id is mod 5;**

**task Repas is**  
  **entry Demander(x : in Philo\_Id) ;**  
  **entry Conclure(x : in Philo\_Id);**  
**private entry Re\_Demander(x : in Philo\_Id) ;**  
**end Repas;**

**task body Repas is separate;**

**next\_id : Philo\_Id := Philo\_Id'first; -- ici vaut 0**

**function unique\_id return Philo\_Id is**

**begin**  
  **next\_id:= next\_id + 1 ; --addition modulo N**  
  **return next\_id ;**  
**end unique\_id;**

**task type philo(x : Philo\_Id := unique\_id) ;**  
**philosophe : array(Philo\_Id ) of philo ;**

**task body philo is**  
**begin**  
  **loop**  
    **Penser;**  
    **Repas.Demander(x) ;**  
    **Manger ;**  
    **Repas.Conclure(x) ;**  
  **end loop ;**  
**end philo ;**

**begin null; end Application ;**

**-- rappel + signifie : + modulo 5**

```
separate(Application)
```

```
task body Repas is
```

```
  En_Attente : Integer := 0;
```

```
  type Bool_Array is array(Philo_Id) of Boolean;
```

```
  Baguette : Bool_Array := (others => true);
```

```
begin
```

```
  loop
```

```
    select
```

```
      accept Demander(x : in Philo_Id) do
```

```
        if Baguette(x) and Baguette(x + 1) then
```

```
          Baguette(x) := false;
```

```
          Baguette(x + 1) := false;
```

```
        else
```

```
          requeue Re_Demander;
```

```
        end if;
```

```
      end Demander;
```

```
    or
```

```
      when En_Attente > 0 =>
```

```
        accept Re_Demander (x : in Philo_Id) do
```

```
          En_Attente := En_Attente - 1;
```

```
          if Baguette(x) and Baguette(x + 1) then
```

```
            Baguette(x) := false;
```

```
            Baguette(x + 1) := false;
```

```
          else
```

```
            requeue Re_Demander ;
```

```
          end if;
```

```
        end Re_Demander ;
```

```
or
```

```
  accept Conclure(x : in Philo_Id) do
```

```
    Baguette(x) := true; Baguette(x + 1) := true;
```

```
    En_Attente := Re_Demander'count;
```

```
    -- Re_Demander'count donne le nombre de tâches
```

```
  end Conclure;
```

```
or terminate;
```

```
  end select;
```

```
  end loop;
```

```
end Repas;
```

```
-- la tâche serveur Repas
```

```
  -- alloue les deux baguettes ou aucune.
```

```
-- un client non servi est aiguillé
```

```
  -- vers l'entrée Re_Demander où il attend.
```

```
-- quand un client rend ses deux baguettes
```

```
-- par Conclure, les requêtes des clients non servis sont
```

```
-- toutes réexaminées; quelques clients sont servis ;
```

```
-- les clients non servis sont remis en attente
```

```
-- sur l'entrée Re_Demander
```

```
-- on voit l'usage du requeue
```

```
-- rappel + signifie : + modulo 5
```

**ALLOCATION DE RESSOURCE UNE À UNE**

```

task type Ressource is
  entry Prendre;
  entry Rendre;
end Ressource ;

```

```

task body Ressource is
  Libre : Integer := 50;
begin
  loop
    select
      when Libre > 0 => accept Prendre;
      Libre := Libre - 1;
    or
      accept Rendre; Libre := Libre + 1;
    or terminate;
    end select;
  end loop;
end Ressource ;

```

**PLUSIEURS RESSOURCES ALLOUÉES FIFO**

```

task type Ressource is
  entry Prendre(X : in Integer) ;
  entry Rendre(X : in Integer) ;
  private entry Finir(X : in Integer);
end Ressource ;

```

**PLUSIEURS RESSOURCES ALLOUÉES FIFO**

**service selon l'ordre d'arrivée des requêtes**  
**-- une tâche au plus attend sur l'entrée Finir**  
**-- et elle bloque alors les requêtes sur l'entrée Prendre**

```

task body Ressource is
  Libre : Integer := 50;
  --si négatif, indique les ressources manquantes
begin
  loop
    select
      when Finir'count = 0 =>
      accept Prendre(X : in Integer) do
      Libre := Libre - X ;
      -- enregistre la requête, alloue si possible
      -- sinon met en attente sur Finir
      if Libre < 0 then requeue Finir; end if;
      end Prendre;
    or
      when Libre >= 0 =>
      accept Finir(X : in Integer);
      -- X maintenant disponibles donc accordées
    or
      accept Rendre(X : in Integer) do
      Libre := Libre + X ; end Rendre;
    or terminate;
    end select;
  end loop;
end Ressource ;
  -- Finir'count donne le nombre de tâches
  -- en attente sur l'entry Finir

```

**LECTEURS RÉDACTEURS****procedure MAIN is****task Synchro is****entry AvantLire;****entry ApresLire;****entry AvantEcrire;****entry ApresEcrire;****end Synchro;****task body Synchro is****NL : Integer := 0;****begin -- le premier accès doit être pour écrire****accept AvantEcrire; accept ApresEcrire;****loop****select****when NL = 0 =>****accept AvantEcrire; accept ApresEcrire;****or****accept AvantLire; NL := NL + 1;****or****when NL > 0 => accept ApresLire; NL := NL - 1;****or****terminate;****end select;****end loop;****end Synchro;****task type Proc; Appli : array(1..N) of Proc;****task body Proc is****begin****-- le processus agit comme rédacteur, il vient :****Synchro.AvantEcrire;****EcritureDansLeFichier;****Synchro.ApresEcrire;****-- le processus agit comme lecteur, il vient :****Synchro.AvantLire;****LectureDansLeFichier;****Synchro.ApresLire;****end Proc;****begin****null;****end Main;**

**ATTENTE SÉLECTIVE COTÉ SERVEUR****APPEL D'ENTRÉE CONDITIONNEL  
COTÉ APPELANTE****-- ATTENTE LIMITÉE DANS LE TEMPS**

```

select
  accept Signal_de_Veille;
  Signal_Recu := True;
or
  delay 30.0; -- durée d'attente du rendez-vous
  Arrêter_le_Train; -- arrêt d'urgence
  -- seulement si le rendez-vous n'a pas eu lieu
  Signal_Recu := False;
end select;
if Signal_Recu then ... else ... end if; -- actions

```

**-- RENDEZ-VOUS CONDITIONNEL  
(OU IMMÉDIAT)**

```

select
  accept Rendre
  do .... end Rendre;
else
  PUT("pas de retour de ressource);
  -- il n'y avait pas de client au rendez-vous
end select;

```

**-- DEMANDE DE RENDEZ-VOUS CONDITIONNEL  
(OU IMMÉDIAT)**

```

select
  R1.Prendre;
  -- le serveur est au rendez-vous
  PUT("R1 acquise");
else
  PUT("R1 non acquise");
  -- le serveur n'est pas au rendez-vous
end select;

```

**-- APPEL D'ENTRÉE  
À ATTENTE LIMITÉE DANS LE TEMPS**

```

select
  R2.Prendre;
  PUT("R2 acquise");
or
  delay 20.0; -- durée d'attente du rendez-vous
  PUT("R2 non acquise");
  -- seulement si le rendez-vous n'a pas eu lieu
end select;

```

# la famille **POSIX, UNIX, LINUX**

## **Mécanismes de communication inter-processus**

### IPC locaux :

- tubes (pipe), tubes nommés,
- fichiers avec mécanisme de verrouillage (lock)
- interruptions logicielles (signaux)
- sockets,
- files de messages
- sémaphores,
- mémoire partagée,

### IPC distants :

- sockets (Unix BSD) ou streams (Unix System V)

# la famille **POSIX, UNIX, LINUX**

## Tube ou "pipe"

### **pipe()** ou - "|" au niveau shell

Un tube est un canal unidirectionnel qui fonctionne en mode flot d'octets. Adapté à l'envoi de caractères.

Les écritures et lectures dans un tube sont atomiques normalement.

Deux processus qui écrivent en même temps ne peuvent mélanger leurs données

(à condition de ne pas dépasser un certain volume max - 4096 octets souvent).

La suite d'octets postée dans un tube n'est pas obligatoirement retirée en une seule fois par l'entité à l'autre bout du tube. Lors de la lecture, il peut n'y avoir qu'une partie des octets retirés, le reste est laissé pour une lecture ultérieure. On délimite les messages envoyés en les entrecoupant de "\n".

L'utilisation d'un tube se fait entre processus de même "famille"/"descendance".

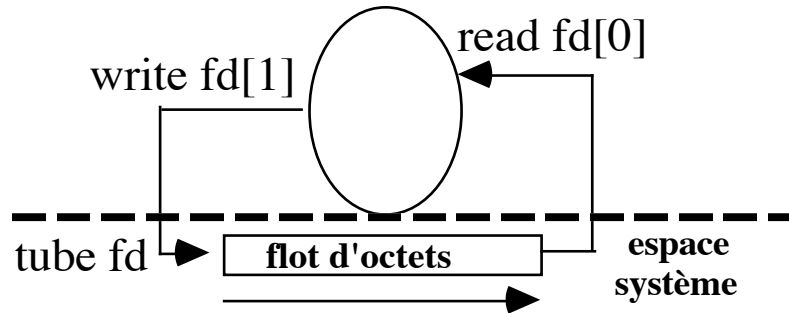
L'usage courant est :

```
int pipefd[2];                /* déclaration d'un nom de canal*/
pipe(pipefd);                /* création du canal avec 2 entrées pipefd[0] et pipefd[1]*/
write (pipefd[1], ...);      /* envoi , "send" */
read (pipefd[0], ...);       /* réception , "receive" */
```

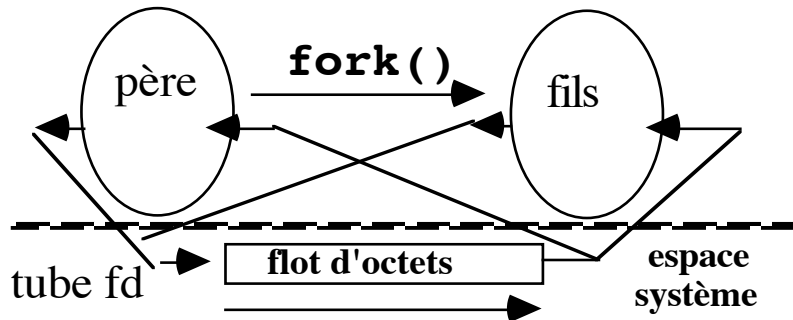
# Etapes d'un échange père/fils par un tube

## 1. création du tube fd :

```
int pipefd[2];
pipe(pipefd);
```



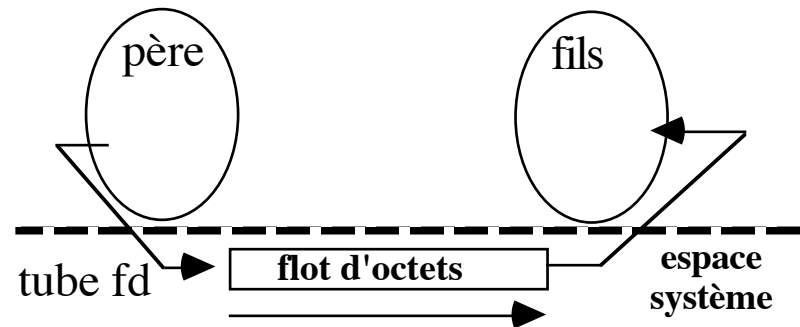
## 2. fork() du père :



## 3. fermeture des extrémités non utilisées :

Le père, qui ne lit pas, ferme, par `close(fd[0])`, l'entrée 0 (in) de fd. (entrée de lecture)

Le fils, qui n'écrit pas, ferme, par `close(fd[1])` l'entrée 1 (out) de fd. (entrée d'écriture)



situation équivalente à celle de "ls -l | more"

## Étapes d'un échange client/serveur avec tube

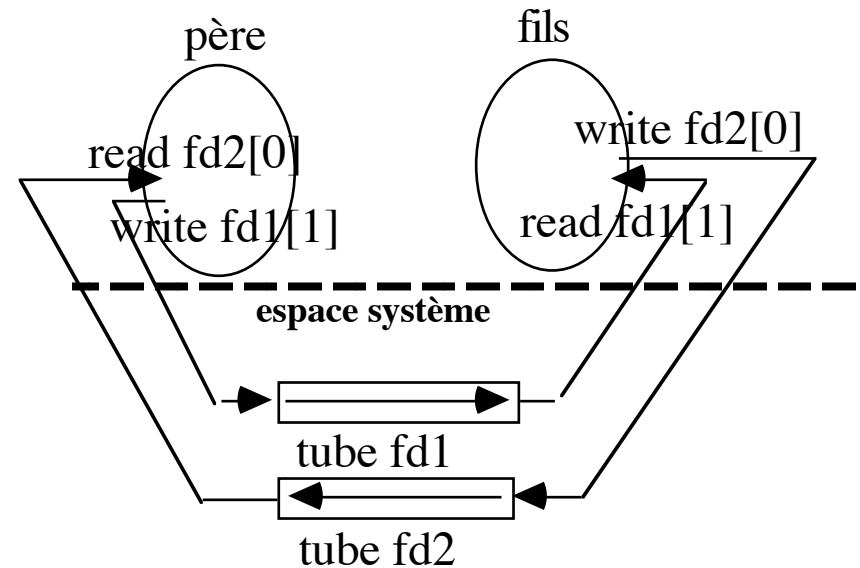
1. création des tubes fd1 et fd2 :

2. `fork()` du père :

3. fermeture des extrémités non utilisées :

Le père, qui écrit dans fd1 et lit dans fd2, ferme par `close(fd1[0])` l'entrée 0 (in) de fd1 et par `close(fd2[1])` l'entrée 1 (out) de fd2.

Le fils, qui écrit dans fd2 et lit dans fd1, ferme par `close(fd1[1])` l'entrée 1 (out) de fd1 et par `close(fd2[0])` l'entrée 0 (in) de fd2



# la famille **POSIX, UNIX, LINUX**

## **Tubes Nommés ou FIFOs**

**Les tubes nommés sont publics au contraire des tubes classiques qui sont privés et connus de leur seule descendance par le mécanisme d'héritage associé à la création de processus par "fork".**

**Le tube nommé est créé par :**

```
mknod("chemindaccès", mode d'accès)
```

**ou par**

```
/etc/mknod chemindaccès p
```

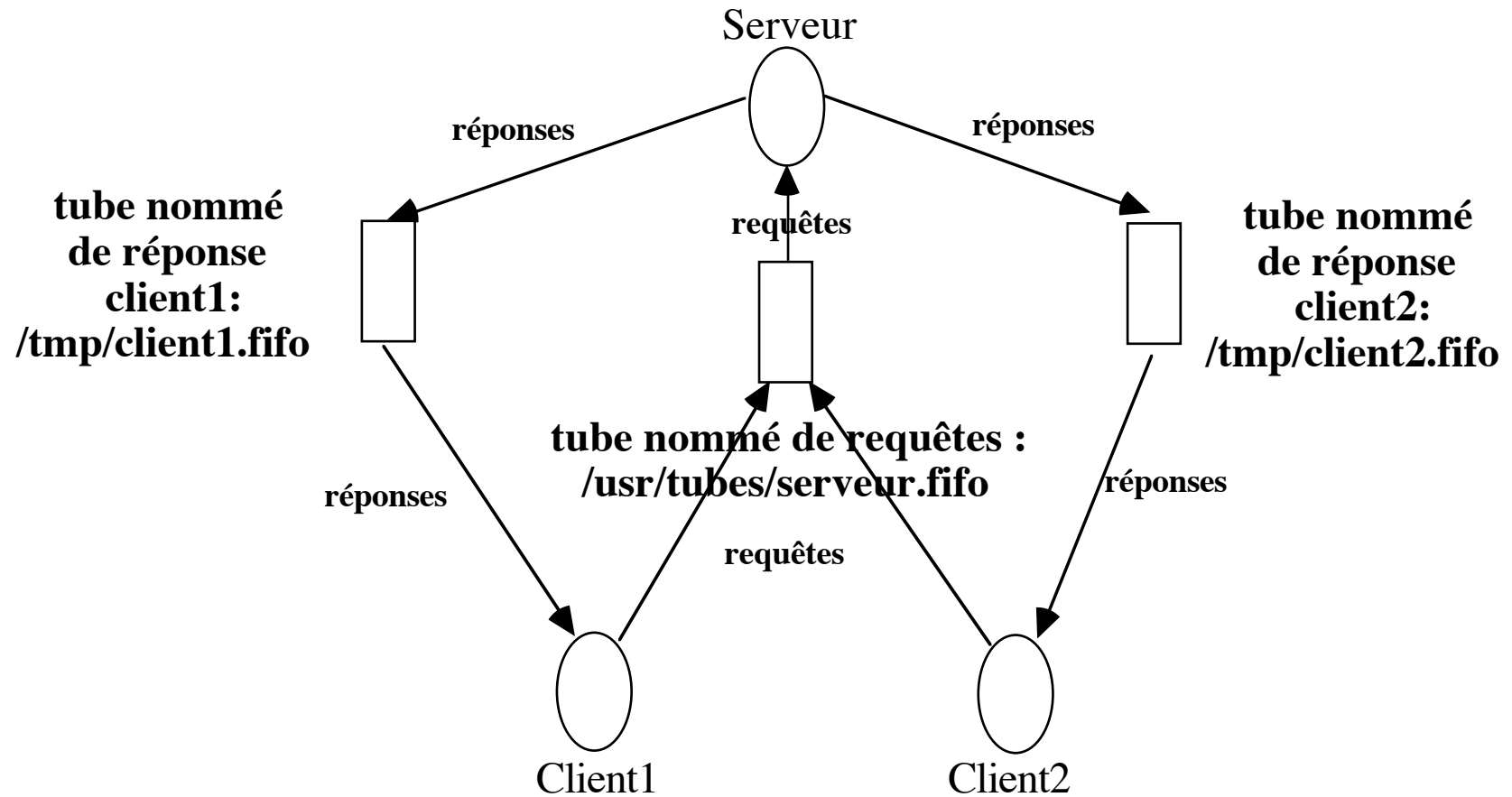
**C'est presque l'équivalent d'un fichier pour le système.**

**Le tube est connu par son chemin d'accès et accessible comme un fichier par :**

```
open(), read(), write(), close().
```

**Les tubes nommés fonctionnent comme les tubes.**

# Client/Serveur avec des tubes nommés



**Ne pas oublier de détruire les tubes nommés après utilisation, car ils sont publics !!!**

# Les sémaphores UNIX

## Notion Unix de groupes de sémaphores

et, sur un groupe, opérations réalisées en bloc :  
soit toutes, soit aucunes (but : éviter l'interblocage)

- `int semget(key_t key, int nsems, int semflg) ;`  
/\* création d'un groupe de sémaphores  
key : clé externe identifiant le groupe  
nsems : nombre de sémaphores du groupe, (0.. nsems-1)  
semflg : droits d'accès \*/
- `struct sembuf {`  
short sem\_num; /\* numéro du sémaphore \*/  
short sem\_op; /\* opérations (+n, -n, ou attente de 0) \*/  
short sem\_flg; /\* option (bloquant ou non) \*/  
};
- `int semop(int semid, struct sembuf* sops, int nsops) ;`  
/\* réalise de manière indivisible nsops opérations sur le  
groupe de sémaphores semid ; ces opérations sont  
décrites par le tableau sosp de type struct sembuf \*/
- `int semct(int semid, int semnum,`  
int cmd, union semum arg) ;  
initialise (cmd=SETVAL), consulte, modifie, le  
sémaphore semnum du groupe semid  
supprime le groupe (cmd=IPC\_RMID)

## Définir P, V, E0 avec les sémaphores UNIX

```
int Num; /* variable globale, nom du groupe */
/* créer un groupe de nom Num avec N sémaphores*/
void CreerSem(key_t cle, int N) {
    Num = semget(cle, N, 0600 | IPC_CREAT); }
/* détruire le groupe de sémaphores Num */
void DetruireSem() { semct(Num, 0, IPC_RMID, 0) ; }
/* initialiser à I le sémaphore S du groupe Num */
void E0(int S, int I) { semct(Num, S, SETVAL, I); }
/* opération P sur le sémaphore S du groupe Num
void P(int S) { struct sembuf TabOp; /*une opération*/
    TabOp.sem_num = S;
    TabOp.sem_op = -1;
    TabOp.sem_flg = 0 ; /* bloquant */
    semop(Num, &TabOp, 1); }
/* opération V sur le sémaphore S du groupe Num
void V(int S) { struct sembuf TabOp; /*une opération*/
    TabOp.sem_num = S;
    TabOp.sem_op = +1;
    TabOp.sem_flg = 0 ;
    semop(Num, &TabOp, 1); }
```

## **4. LES SYSTÈMES RÉPARTIS ASYNCHRONES**

**Ensemble fini de sites interconnectés par un réseau de communication**

### **CHAQUE SITE**

- **processeur, mémoire locale, mémoire stable (permanence des données en dépit de défaillances)**

### **RÉSEAU DE COMMUNICATION**

- **connexe : tout processus peut communiquer avec tous les autres**
- **communication et synchronisation entre processus par messages via le réseau de communication**

### **PROPRIÉTÉS CARACTÉRISTIQUES**

- **Pas de mémoire commune**
- **Pas d'horloge physique partagée par 2 processus ou plus**
- **Absence de majorant connu sur le temps de transfert des messages**
- **Absence de minorant connu sur les vitesses des processeurs**

**Nota : appelé aussi modèle à délais non bornés**

**Exemple : cas de Internet et des réseaux longue distance (WAN)**

**DIFFICULTÉS CARACTÉRISTIQUES DES SYSTÈMES RÉPARTIS ASYNCHRONES****POUR LES OBSERVATIONS :**

deux observations faites sur deux sites distincts peuvent différer

- par l'ordre de perception des événements
- par leur date

**POUR DES DÉCISIONS COHÉRENTES ENTRE PLUSIEURS SITES = CONSENSUS**

- visions différentes de l'état des ressources du système
- pas d'état global de référence en temps réel au moment où il faut prendre des décisions
- risque accru de défaillance (plus de composants)
  - => la défaillance d'un élément n'est pas un événement rare
  - => importance des hypothèses sur les défaillances possibles
  - => nécessité de détecteurs de défaillances (contrainte : utiliser des détecteurs non fiables)

**Problème.** Un processus  $P_i$  ne peut pas savoir si un autre processus  $P_j$  est défaillant ou si la réponse qu'il attend de  $P_j$  est en préparation par  $P_j$  (processus très lent) ou encore en route (message très lent). En pratique, il est essentiel d'introduire une notion de temps (temps réel et non pas temps du processus  $P_i$ ) : combien de temps  $P_i$  doit-il attendre avant de suspecter  $P_j$  de défaillance

**suspecter une défaillance  $\neq$  détecter une défaillance**

**attention, à ne pas confondre avec les systèmes parallèles centralisés**

**encore appelés des multiprocesseurs à mémoire commune**

- existence d'une mémoire commune (physique, réflexive, virtuelle)
- existence d'une horloge ou d'un rythme commun

## DÉSORDRE NATUREL DES COMMUNICATIONS

### Objet réparti et répliqué sur 4 sites S1, S2, S3, S4

**S1** : traitement t1 avec la copie locale; envoi M1; délai ; traitement t2 avec la copie locale; envoi M2;

**S2** : traitement t4 avec la copie locale; envoi M4;

**S3** : traitement t3 avec la copie locale; envoi M3;

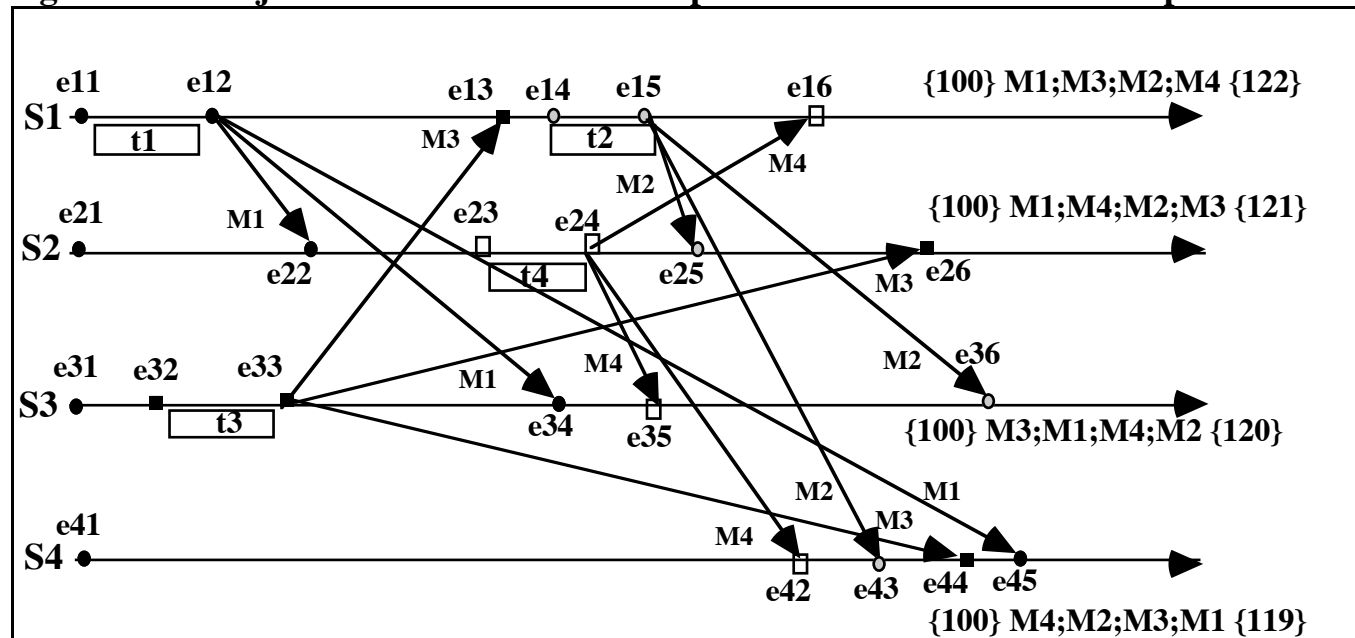
t1 : ajouter 20

t2 : retrancher 10

t3 : multiplier la valeur par 110%

t4 : lire et sauvegarder (ou imprimer ou visualiser) la valeur locale

M1 à M4 : messages de mise à jour diffusés à toutes les copies. Mi : exécuter ti sur la copie locale



Les sites visualisent 122, 120, 130, 100. Belle coopération!  
 Les répliques de l'objet valent 122, 121, 120, 119. Belle cohérence!

**Par la nature physique de la communication, l'émission d'un message sur un site précède nécessairement la réception du message sur le site destinataire. Toute réception d'un message est causée par une émission antérieure. (il ne peut y avoir de réception spontanée de message)**

**Cette relation causale permet d'établir, dans un système réparti, une relation d'ordre partiel entre l'événement d'émission d'un message sur un site et l'événement de réception du message sur un autre site destinataire. Cette relation se note (on lit précède) :**

**$\forall m, \text{EMISSION}(m) \rightarrow \text{RECEPTION}(m)$**   
**(notée "happened before" par Lamport)**

**Plus exactement la relation est observée lorsque le message a été reçu**

**$\forall m, \text{RECEPTION}(m) \Rightarrow (\text{EMISSION}(m) \rightarrow \text{RECEPTION}(m))$**

**RELATION DE PRÉCÉDENCE ENTRE DES ÉVÉNEMENTS RÉPARTIS**

**événement : instruction exécutée par un processeur**  
**(précédence : "happened before", L.Lamport, CACM 21,7, 1978)**

**L1) Ordre d'exécution local. A "précède" A' si A et A' sont des événements qui ont été générés dans cet ordre sur le même site S :  $A \rightarrow A'$**

**L2) Ordre causal pour chaque message. A "précède" A' si A est l'événement d'émission d'un message M par le site P et que A' est l'événement de réception du message M sur Q :  $A \rightarrow A'$**

**L3) La relation de précédence dans un système réparti est la fermeture transitive des deux relations précédentes.**

**si  $A \rightarrow B$  et  $B \rightarrow C$  alors  $A \rightarrow C$**

**La précédence est un ordre partiel entre les événements du système réparti**

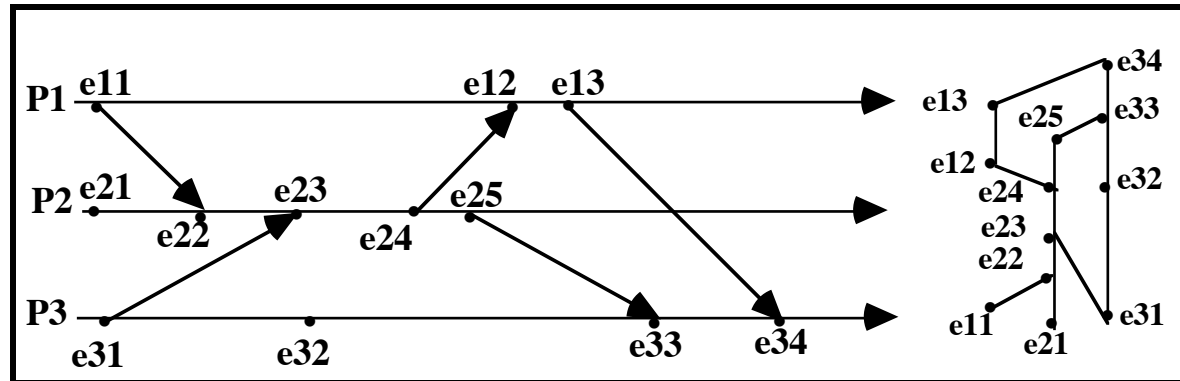
**La causalité entre événements implique la précédence entre eux :**

**$A \text{ cause } B \Rightarrow A \rightarrow B$**

**Une précédence entre événements exprime une causalité potentielle :**

**(si  $A \rightarrow B$ , A peut avoir influencé B).**

### RELATION D'ORDRE PARTIEL



L1 : e11 -> e12 -> e13      L1 : e21 -> e22 -> e23 -> e24 -> e25      L1 : e31 -> e32 -> e33 -> e34

L2 : e11 -> e22      L2 : e31 -> e23      L2 : e24 -> e12      L2 : e25 -> e33      L2 : e13 -> e34

L3 : e11 -> e22 -> e23 -> e24 -> e25 -> e33 -> e34

L3 : e31 -> e23 -> e24 -> e12 -> e13 -> e34

INCOMPARABLES (e11, e21) (e11, e31) (e21, e31) (e31, e22) (e23, e32) (e13, e25) (e12, e33) etc...

**Il existe des applications qui peuvent être traitées avec seulement un ordre partiel : toutes les applications du genre client serveur, sans consensus coopératif entre serveurs ou clients**

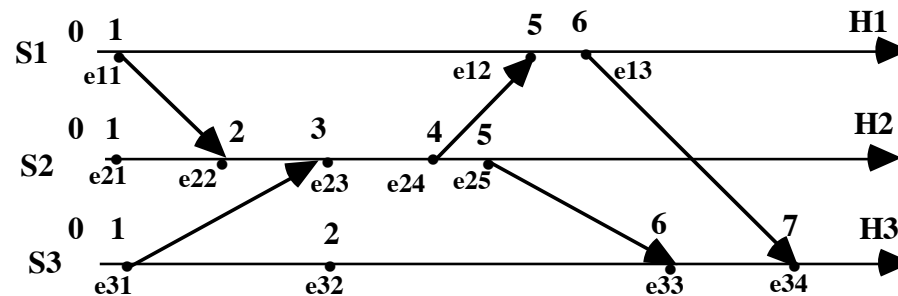
**Il existe des applications qui peuvent pas et qui demandent un ordre total parce qu'elles utilisent : copies multiples, exclusion mutuelle, diffusion globale ordonnée, base de données réparties,...**

# DATATION TOTALE PAR DES HORLOGES LOGIQUES

- Objectif (Lamport 1978) : réaliser une datation totale des événements
  - respectant la dépendance causale et déterminable par consultation locale

Un compteur entier  $H_i$ , initialisé à 0, est maintenu sur chaque site  $S_i$ .

- A chaque événement  $e$ , localisé sur  $S_i$  :
  - 1)  $H_i := H_i + 1$
  - 2)  $e$  est daté par  $H_i$  :  $H(e) := H_i$
- Si  $e$  est l'émission d'un message  $m$ , celui-ci est estampillé par la date de son émission sur  $S_i$ , obtenue en lisant  $H_i$  :
  - estampille  $E(m) = H(\text{émission}(m)) := H_i$
- Si  $e$  est la réception d'un message  $m$  venant de  $S_j$ , l'estampille  $E(m)$  est utilisée par le compteur  $H_i$  (l'horloge logique locale sur  $S_i$ ) pour rattraper le retard sur  $H_j$ , s'il en a, avant d'incrémenter  $H_i$  pour dater  $e$ .
  - $H_i := \max(H_i, E(m)) + 1$

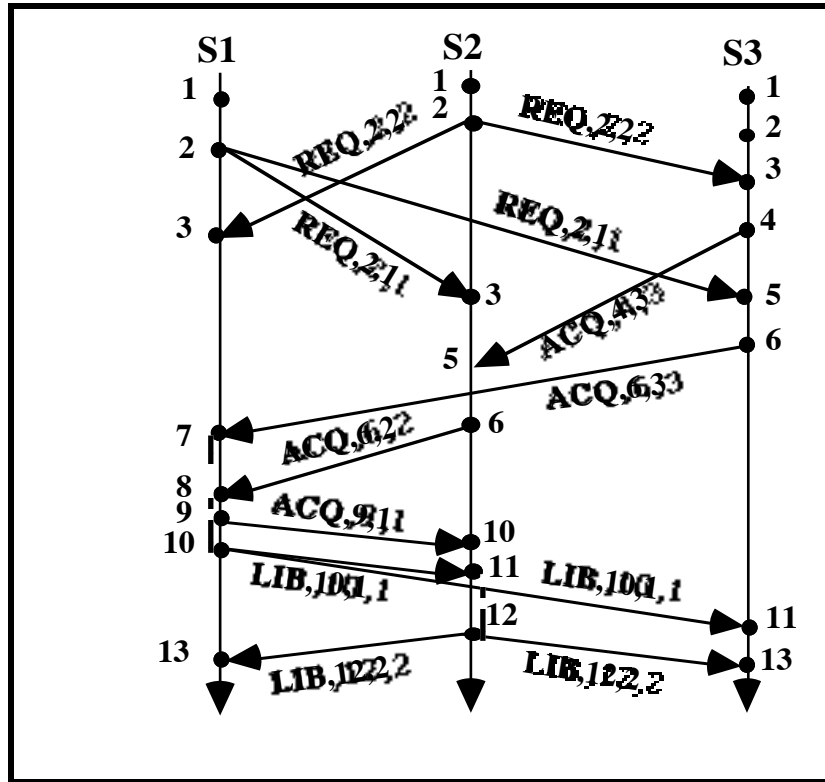


- Réalisation d'un ordre total ( $\ll$ ) :
  - soit  $a$  sur  $S_i$ ,  $b$  sur  $S_j$  donc  $H(a) = H_i(a)$  et  $H(b) = H_j(b)$
  - $a \ll b \Leftrightarrow (H(a) < H(b))$  ou  $(H(a) = H(b) \text{ et } i < j)$
  - l'ordre total sur les sites est arbitraire (mais il doit être le même partout)

**EXCLUSION MUTUELLE REPARTIE (Lamport 1978)**

- **Hypothèses :**
- le nombre **N** des sites est connu
  - les canaux sont **FIFO**
  - ordre total réalisé par horloge logique
- **Principe : connaissance mutuelle répartie acquise par chaque site"**
- **Demande d'entrée d'un site  $S_i$  en section critique :**  
diffusion de  $(req, H(req), i)$  à tous les sites,  $S_i$  compris
- **Entrée en section critique quand  $S_i$  sait que:**
- a) tous les sites ont reçu sa demande ou qu'ils en ont émis une aussi
  - b) sa demande est la plus ancienne de toutes
- **Réception par  $S_i$  d'une demande de  $S_j$  : réponse systématique par  $(acq, H(acq), S_i)$**
- **Libération de la section critique par  $S_i$  : diffusion de  $(lib, H(lib), S_i)$  à tous les sites,  $S_i$  compris**
- **Structure de données sur chaque site : tableau de  $N$  messages, 1 par site**
- initialement sur  $S_i$   $M_{ij} := (lib, 0, S_j)$  pour tout  $j$
  - réception de  $M = (req, H(req), j)$  ou  $(lib, H(lib), S_j) \Rightarrow M_{ij} := M$
  - réception de  $M = (acq, H(acq), S_j) \Rightarrow$  si  $M_{ij} \neq (req, H(req), j)$  alors  $M_{ij} := M$   
si l'acquittement vient après une requête de  $S_j$ , on n'oublie pas celle-ci
- **Règle de décision pour chaque site  $S_i$  :**
- **$S_i$  entre en section critique quand sa demande  $M_{ii} \ll M_{ij}$  pour tout  $j$  (rappel :  $\ll$  :ordre total)**  
**En effet, comme les canaux sont FIFO, il n'y a plus de requête plus ancienne en chemin dans un canal**

**EXEMPLE**



	H1	H2	H3
M11	LIB,0,1 0	M21 LIB,0,1 0	M31 LIB,0,1 0
M12	LIB,0,2 0	M22 LIB,0,2 0	M32 LIB,0,2 0
M13	LIB,0,3 0	M23 LIB,0,3 0	M33 LIB,0,3 0
M11	REQ,2,1 2	M21 REQ,2,1 3	M31 REQ,2,1 5
M12	REQ,2,2 3	M22 REQ,2,2 2	M32 REQ,2,2 3
M13	ACQ,6,3 7	M23 ACQ,4,3 5	M33 LIB,0,3 0
		M21 LIB,10,1 11	M31 LIB,10,1 11
		M22 REQ,2,2 2	M32 LIB,12,2 13
		M23 ACQ,4,3 5	M33 LIB,0,3 0

S1 entre en s.c.  
à H1 = 7

S2 entre en SC à H2 = 11

→ Propriétés :

- Si est seul en section critique.
- Il n'y a pas de coalition car les entrées suivent l'ordre total
- Mais il faut 3(N - 1) messages