Introduction aux Systèmes Distribués (Répartis)

Chapitre 1

Plan

- Définition
- Critères de Conception d'un S.E.D
- Logiciels?
- Caractéristiques des systèmes repartis
- Algorithmes repartis
- Qualités d'un algorithme reparti
- Critères d'un système distribué

Définitions d'un SD

- Définition [Tanenbaum]: *Un ensemble d'ordinateurs indépendants qui apparaît à un utilisateur comme un système unique et Cohérent*

- Les machines sont autonomes
- Les utilisateurs ont l'impression d'utiliser un seul système.

Définitions d'un SD

- Définition [Lamport]

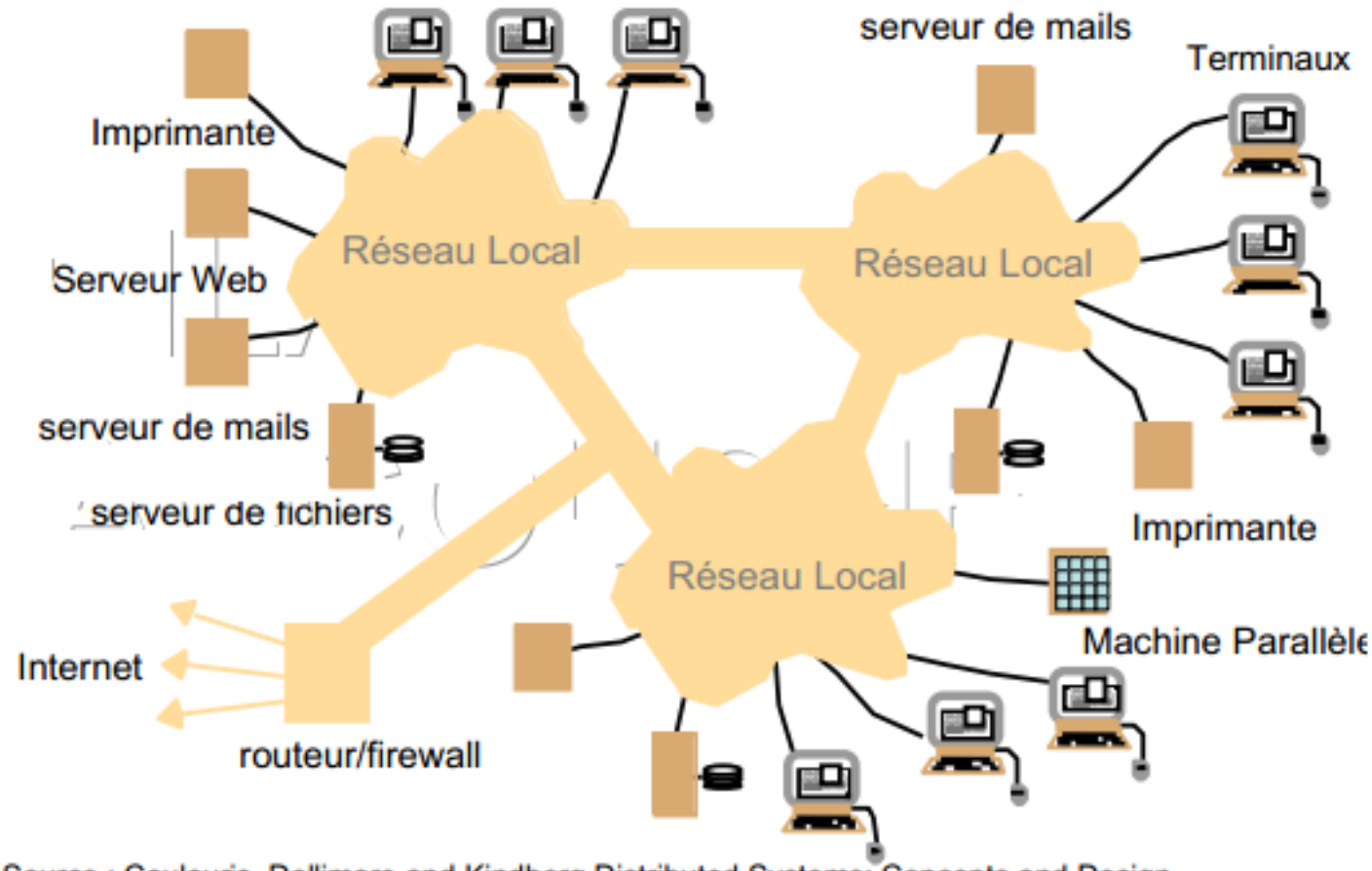
A distributed system is one that stops you from getting any work done when a machine you've never heard of crashes.

Un système réparti est un système qui vous empêche de travailler quand une machine dont vous n'avez jamais entendu parler tombe en panne.

➡ **Définition humoristique, mais qui met l'accent sur deux points essentiels :**

- **l'interdépendance entre les éléments d'un système réparti**
- **l'importance du traitement des défaillances.**

Exemple de Système Réparti : Un intranet



Source : Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edition 3, © Addison-Wesley 2001

Définition

- Un système à plusieurs processeurs n'est pas forcement un système réparti.

- Qu'est ce qu'un système réparti, distribué, parallèle?

Concepts matériels

- Taxonomie de Flynn (1972)

On différencie les systèmes sur la base du flux d'instructions et de données.

– SISD : PC monoprocesseur
– SIMD : machines vectorielles
– MISD : pipeline
– MIMD : machines multiprocesseurs faiblement et fortement couplées (systèmes parallèles, systèmes distribués, systèmes d'exploitation réseaux)

Définition

Un **système réparti** est un ensemble de sites reliés par un réseau, comportant chacun une ou plusieurs machines.

Un **système d'exploitation réparti** fournit et contrôle l'accès aux ressources du système réparti.

Un **système d'exploitation parallèle** contrôle l'exécution de programmes sur une machine parallèle (multiprocesseurs).

Un **système d'exploitation de réseaux** fournit une plate-forme de machines reliées par un réseau chacune exécutant son propre système d'exploitation.

parallèle ≠ réparti ≠ réseau ?

Systèmes parallèles	Systèmes répartis	Systèmes d'exploitation de réseaux
processeurs	sites	ressources
homogènes	hétérogènes	hétérogènes
Partage ou non de mémoire	Mémoires individuelles	Mémoires individuelles
Couplage fort	Couplage faible	Couplage faible
Topologies du réseau d'interconnexion	Réseau local LAN + WAN	Réseau local
Les users sont au courant de la multiplicité des processeurs	Les users ont l'impression d'être dans un système centralisé	Les users sont au courant de la multiplicité des machines

Objectifs

- Coût : plusieurs processeurs à bas prix
- Puissance de calcul et de stockage : aucune machine centralisée ne peut rivaliser
- Performance (accélération) : via du calcul parallèle
- Adaptation : à des classes d'applications réelles naturellement distribuées
- Fiabilité : résistance aux pannes logicielles ou matérielles
- Extensibilité : croissance progressive selon le besoin

Avantages/Inconvénients

Avantages
- partage de données
- partage de périphériques
- communication
- souplesse (politiques de placement)

Inconvénients
- Très peu de logiciels existent sur le marché.
- Le réseau peut très vite saturer.
- La sécurisation des données sensibles est compliquée.
- La mise en œuvre est difficile.

Critères de Conception d'un S.E.D. ou d'un Syst. Opératoire pour une architecture distribuée

- Transparence
- Souplesse
- Fiabilité
- Performances
- Dimensionnement

Critères de Conception d'un S.E.D Transparence

- Transparence à l'emplacement: L'utilisateur ne connaît pas où sont situées les ressources
- Transparence à la migration: Les ressources peuvent être déplacées sans modification de leur nom
- Transparence à la duplication: L'utilisateur ne connaît pas le nombre de copies existantes
- Transparence à la concurrence: Plusieurs utilisateurs peuvent partager en même temps les mêmes ressources
- Transparence au parallélisme: Des tâches peuvent s'exécuter en parallèle sans que les utilisateurs le sachent

Critères de Conception d’un S.E.D souplesse

- La facilité de modification, de configuration et d'extension

Critères de Conception d'un S.E.D Fiabilité

- **Disponibilité**

La disponibilité est la fraction de temps pendant laquelle le système est utilisable :

- limiter le nombre des composants critiques
- dupliquer les parties clés des composants logiciels et matériels (redondance)
- mais implique de savoir maintenir la cohérence des copies

Critères de Conception d'un S.E.D Fiabilité

- **Securité**

Les ressources doivent être protégées contre des utilisations abusives et malveillantes. En particulier le problème de piratage des données sur le réseau de communication

Critères de Conception d'un S.E.D Fiabilité

- **Tolérance aux pannes**

Le système doit être conçu pour masquer les pannes aux utilisateurs. La panne de certains serveurs (ou leur réintégration dans le système après la réparation) ne doit pas perturber l'utilisation du système en terme de fonctionnalité

Critères de Conception d'un S.E.D Performances

- **Critères**

- temps de réponse
- débit (nombre de travaux par heure)
- taux d'utilisation du système
- pourcentage utilisé de la bande passante du réseau

Critères de Conception d'un S.E.D Performances

- **Problèmes**

-La communication est en général assez lente dans les systèmes distribués par rapport accès fichier

- Le mécanisme de tolérance aux pannes requiert beaucoup d'opérations "inutiles" si pas de panne

- **Solutions**

Minimiser les échanges de message

- réduire le champ d'application des mécanismes de reprises sur pannes

Critères de Conception d’un S.E.D Performances: Dimensionnement

- Goulots d'étranglement potentiel si débit d'information très important

Logiciel?

Nous distinguons, historiquement, trois étapes dans la Conception des logiciels d'exploitation des réseaux d'ordinateurs.

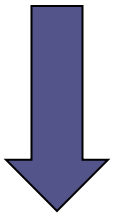
Les systèmes client-serveur

- Chaque site dispose d'un système d'exploitation propre
  - Peut être client : demandeur de services aux autres sites
  - Ou serveur : offre des services aux autres sites

Systèmes homogènes

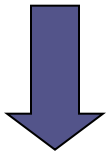
❖ Interconnexion de Systèmes homogènes avec partage de ressources

➢ Répartition plus ou moins visible ⟶ Partage de ressources Réparties sur les différents sites



Possibilité de réaliser à distance la plupart des appels système

Les deux premières approches permettent à un utilisateur, à partir d'un site donné, d'accéder aux services offerts par un autre site par ajout des fonctions liées à la répartition à un système déjà existant.



- Sites parfaitement autonomes
- Architecture matérielle et localisation des différentes fonctions du système visibles à l'utilisateur.

Systèmes d'exploitation répartis

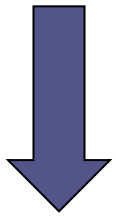
- Développement de la technologie des microprocesseurs
- Apparition des réseaux locaux à haut débit

Concevoir un système unique mis en œuvre sur un ensemble
De sites communiquant entre eux par échange de messages.

- Systèmes conçus au départ comme répartis
- Leurs fonctions de base prennent en compte la répartition.

• Ces fonctions sont assurées par un noyau réparti dont une copie est localisée sur chacun des sites.

• Les fonctions qui mettent en jeu plusieurs sites sont réalisées Par la coopération des noyaux de ces sites.



L'utilisation de plusieurs processeurs
Est donc transparente aux usagers

Caractéristiques des Systèmes répartis

- **Absence de mémoire commune**

Impossibilité de capter instantanément
L'état global d'un système réparti au
moyen d'un ensemble de variables partagées.

- **Absence d'une horloge physique commune**

+

- **Variabilité des délais de transmission des messages**

Deux éléments distincts du système peuvent avoir une perception différente de l'état d'un troisième élément ou sous-système et de l'ordre des événements qui s'y produisent.

- Délai de transmission d'un message entre deux processus s'exécutant sur deux machines distinctes TRES GRAND par rapport au temps qui sépare deux points observables consécutifs sur une même machine.

A un instant donné, un processus s'exécutant sur une machine ne peut connaître que de manière approchée l'état d'une autre Machine.

Adéquation architecture / logiciel

- Recherche d'une adéquation entre l'architecture et la structure fonctionnelle des problèmes à résoudre

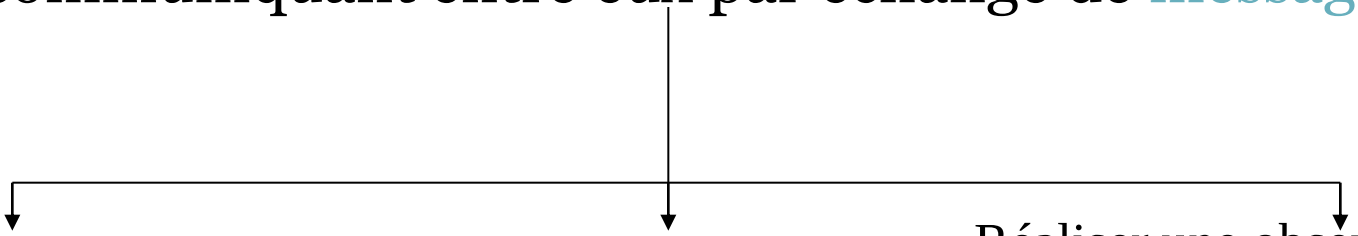
Faire coopérer plusieurs sites à la résolution d'un problème.

Construire un algorithme résolvant un problème comme un Ensemble d'unités fonctionnelles (ex: des processus) s'exécutant sur des sites distincts et communiquant par échange de messages.

Algorithmes répartis

Définition:

- Un algorithme réparti est un algorithme parallèle composé d'un ensemble fini de processus séquentiels communiquant entre eux par échange de messages.



Calcul de fonctions
- Election
- Arborescence couvrante
- Chemins minimaux,…

Réaliser un service
- Exclusion mutuelle
- Protocoles de communication :

✓Synchrone (RDV)

✓Synchronisme virtuel (total, causal, atomique,…)

Réaliser une observation
- Détection d'interblocage
- Détection de la terminaison
- Détection de propriétés stables (indépendamment d'une propriété stable particulière).

Quelques algorithmes de contrôle réparti

- Exclusion mutuelle:

Il s'agit d'attribuer un privilège équitablement à un ensemble de processus communicants qui coopèrent à la réalisation d'un but commun (le privilège ne peut être possédé indéfiniment par un même processus).

- Election:

Il y a élection d'un processus par l'ensemble des autres processus lorsque le privilège lui sera attribué une fois pour toute.

Quelques algorithmes de contrôle réparti

- Interblocage:

C'est la situation dans laquelle se trouve un ensemble de processus (au moins deux) telle que chaque processus de l'ensemble attend l'occurrence d'un événement qui ne peut être produit que par un autre processus de ce même ensemble.

➢ Evénement :

✓ **Libération d'une ressource**
✓ **Arrivée d'un message**

Deux types d'interblocage

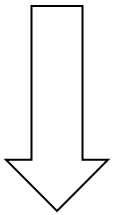
- **Interblocage de compétition (lié aux ressources)**
- **Interblocage lié à la communication : chaque processus** attend **un message d'un autre processus alors qu'aucun message n'est en transit.**

Quelques algorithmes de contrôle réparti

- Terminaison:

Comment peut-on détecter la terminaison d'un algorithme réparti ?

Facile si l'on pouvait disposer d'un état global instantané décrivant L'état des processus et des voies de communication.



Un algorithme réparti sera dit terminé si tous les processus Sont passifs et il n'y a aucun message en transit.

Qualité d'un algorithme réparti

- Quelques critères
  - ❖ Réseau de communication
    - ➢ Duplication des messages
    - ➢ Altération des messages
    - ➢ Déséquencement des messages
    - ➢ Délai de transmission fini
    - ➢ Délai de transmission borné
  - ❖ Trafic engendré
    - ➢ Nombre de messages engendrés pour produire un résultat
  - ❖ Temps de calcul

Qualité d'un algorithme réparti

❖ Symétrie

Un algorithme est dit symétrique si tous les processus exécute le même texte. Tous les processus jouent alors le même rôle.

⇩

Meilleure résistance aux pannes des processus

❖ Etat local ou global

La connaissance d'un état local uniquement pour prendre une décision

⇩

- Réduction du nombre de messages échangés
- Chaque processus est plus autonome par rapport aux autres

⇩

Meilleure résistance aux pannes

Caractéristiques des systèmes répartis

- Le système doit pouvoir fonctionner (au moins de façon dégradée) même en cas de défaillance de certains de ses éléments
- Le système doit pouvoir résister à des perturbations du système de communication (perte de messages, déconnexion temporaire, performances dégradées)
- Le système doit pouvoir résister à des attaques contre sa sécurité (tentatives de violation de la confidentialité et de l'intégrité, usage indu de ressources, déni de service)
- Le système doit pouvoir facilement s'adapter pour réagir à des changements d'environnement ou de conditions d'utilisation
- Le système doit préserver ses performances lorsque sa taille croît (nombre d'éléments, nombre d'utilisateurs, étendue géographique) → dimensionnement

Caractéristiques des systèmes répartis

Difficultés

- Propriété d'asynchronisme du système de communication (pas de borne supérieure stricte pour le temps de transmission d'un message
  - ✓ Conséquence : difficulté pour détecter les défaillances
- Dynamisme (la composition du système change en permanence)
  - ✓ Conséquences : difficulté pour définir un état global
  - ✓ Difficulté pour administrer le système
- Grande taille (nombre de composants, d'utilisateurs, dispersion géographique)
  - ✓ Conséquence : la capacité de croissance (scalability) est une propriété importante, mais difficile à réaliser

Merci pour votre attention!

Communication dans les systèmes distribués

Plan

- Couches de protocoles
- Modèle client-serveur
- Appels de procédures à distance RPC
- La communication de groupe
- Les middlewares
- Objets distribués

Couches de protocoles

Modèle en couches OSI

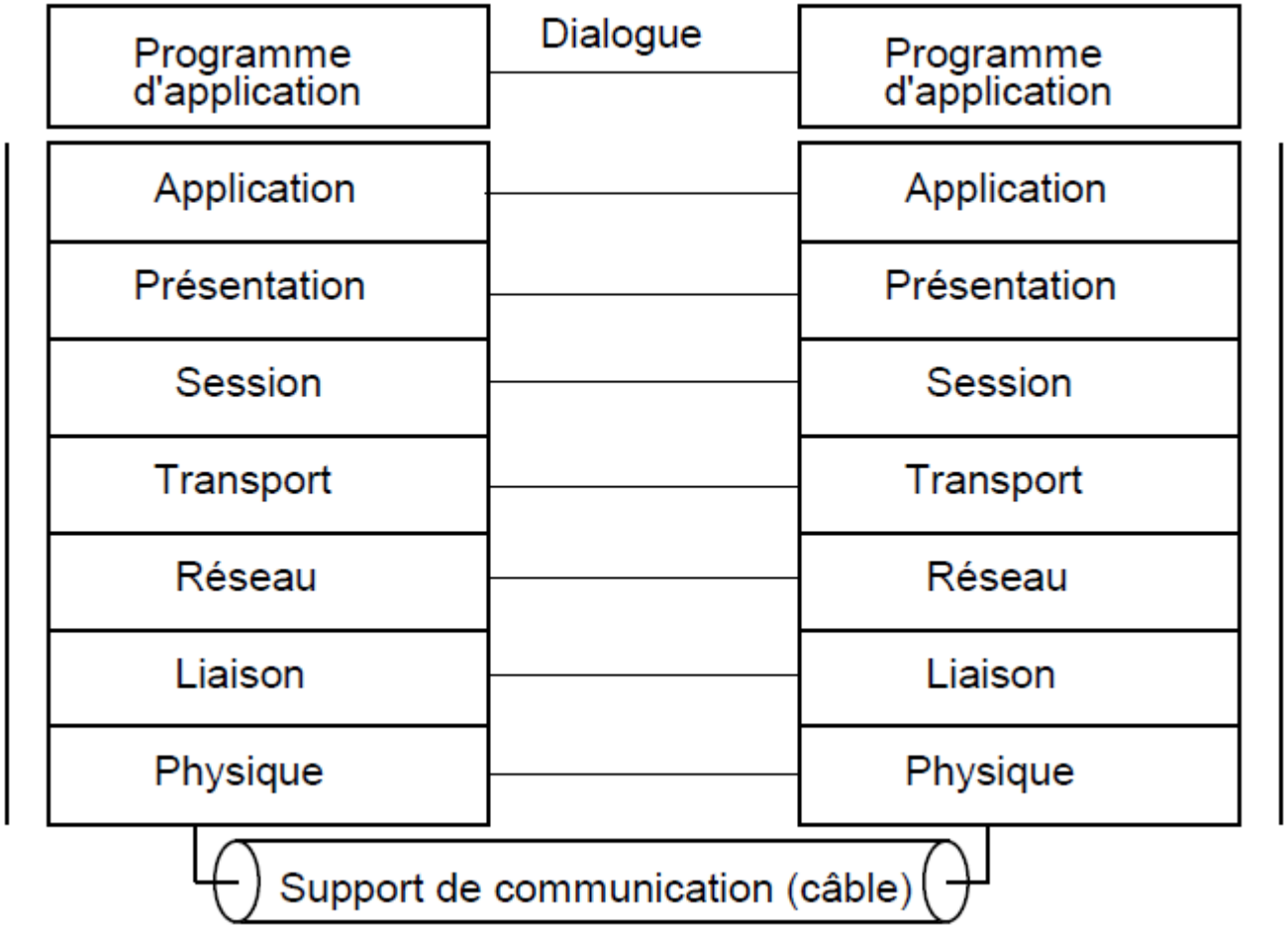
- Une couche à un rôle particulier
- Une couche d'une entité communique avec une couche de même niveau d'une autre entité en respectant un certain protocole de communication
- Pour communiquer avec une autre entité, une couche utilise les services de sa couche locale inférieure
- Données échangées entre 2 couches : trames ou paquets
  - ✓ Données structurées
  - ✓ Taille bornée
  - ✓ Deux parties:

-Données de la couche supérieure à transmettre

-Données de contrôle de la communication entre couches

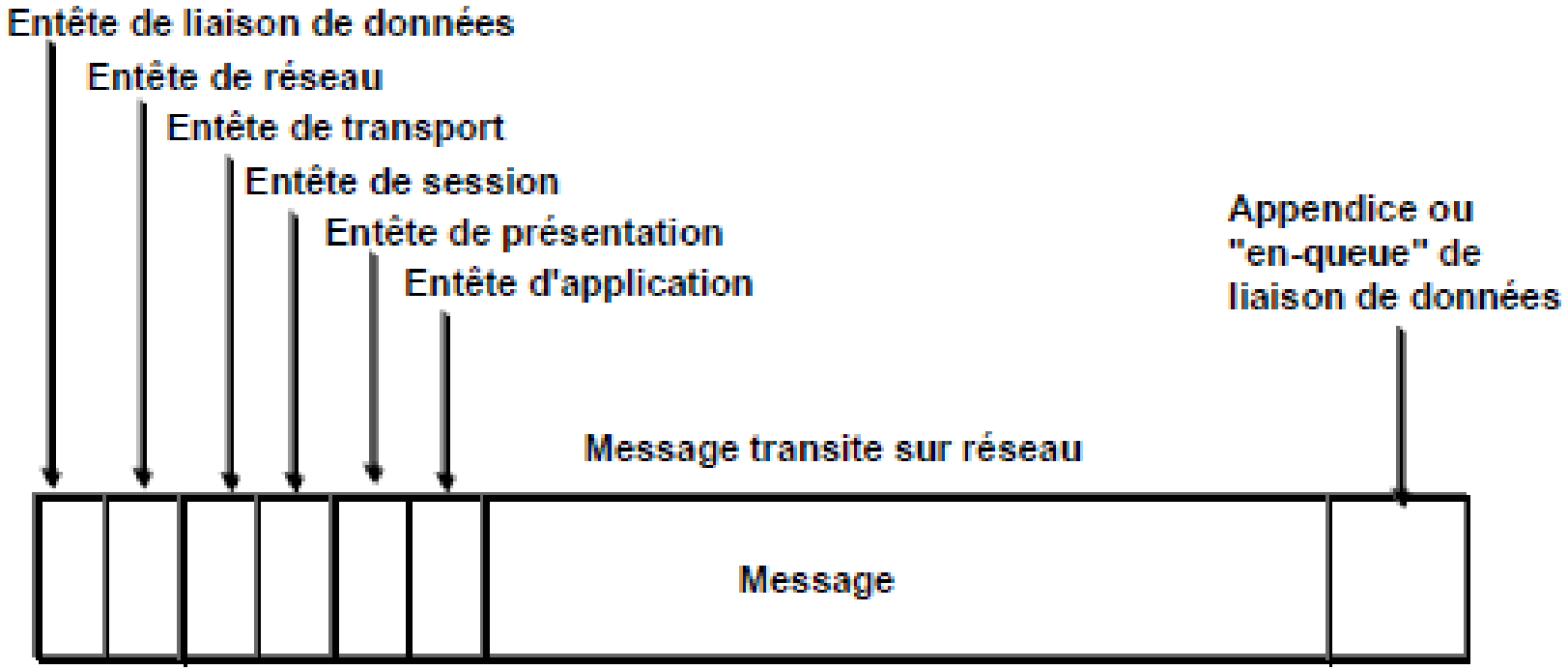
Rappels des protocoles OSI

- **Modèle en couches OSI**

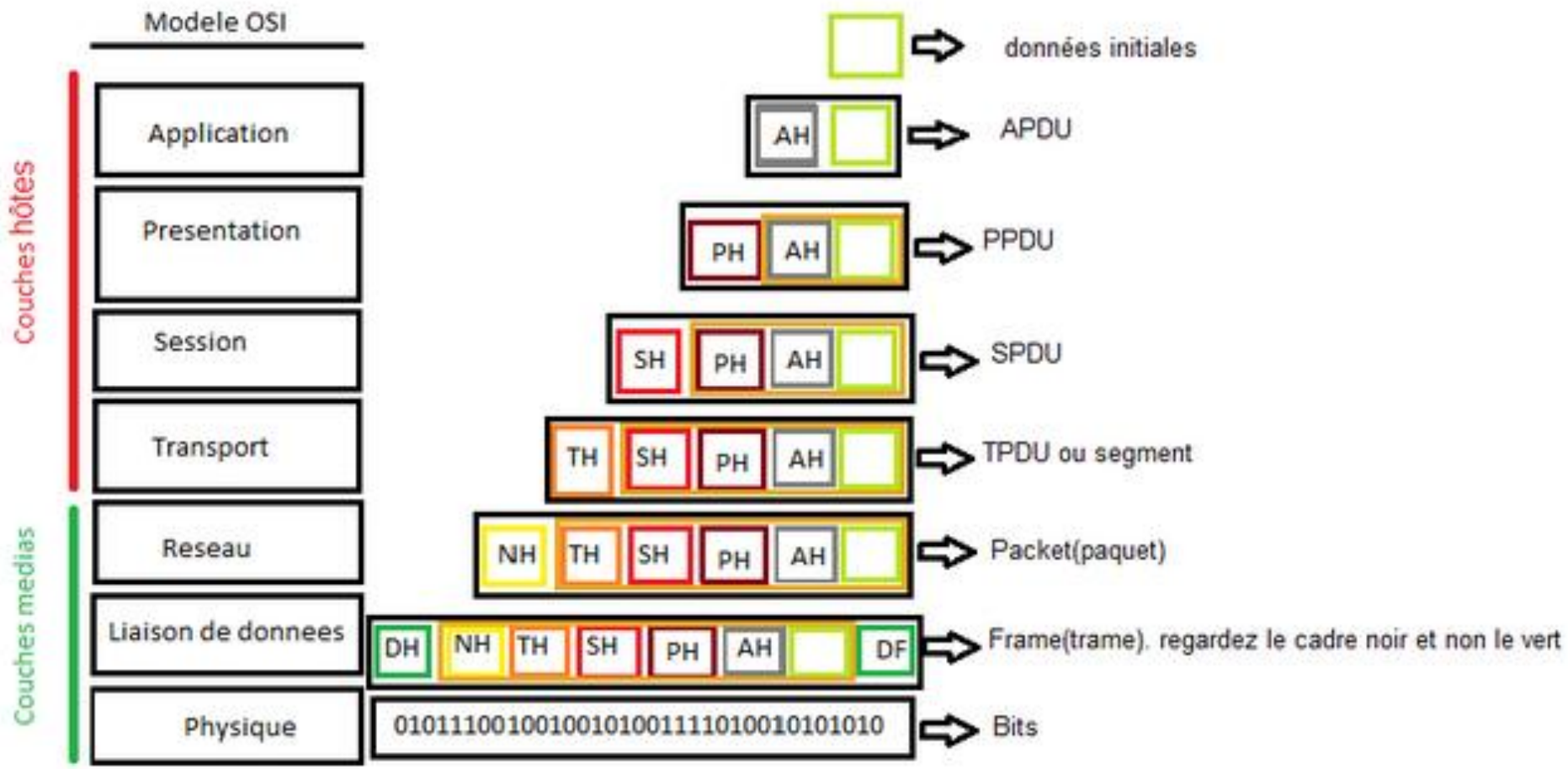


Rappels des protocoles OSI

- **Modèle en couches OSI**



Rappels des protocoles OSI



Rappels des protocoles OSI

architecture en 7 couches

- **Physique** : transmission des données binaires sur un support physique
- **Liaison** : gestion d'accès au support physique, assure que les données envoyées sur le support physique sont bien reçues par le destinataire
- **Réseau** : transmission de données sur le réseau, trouve les routes à travers un réseau pour accéder à une machine distante
- **Transport** : transmission (fiable) entre 2 applications
- **Session** : synchronisation entre applications, reprises sur pannes
- **Présentation** : indépendance des formats de représentation des données (entiers, chaînes de caractères...)
- **Application** : protocoles applicatifs (HTTP, FTP, SMTP ...)

Modèle client-serveur

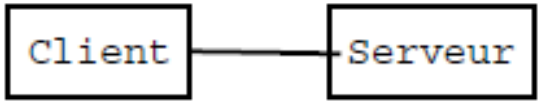
Modèle client/serveur

- Deux rôles distincts
  - ✓**Client** : demande que des requêtes ou des services lui soient rendus
  - ✓**Serveur** : répond aux requêtes des clients

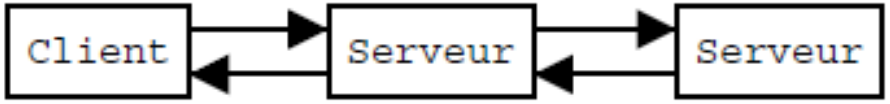
- Interaction **(Requête / Réponse)**
  - ✓Message du client vers le serveur pour faire une requête
  - ✓Exécution d'un traitement par le serveur pour répondre à la requête
  - ✓Message du serveur vers le client avec le résultat de la requête

Types d'architecture client-serveur

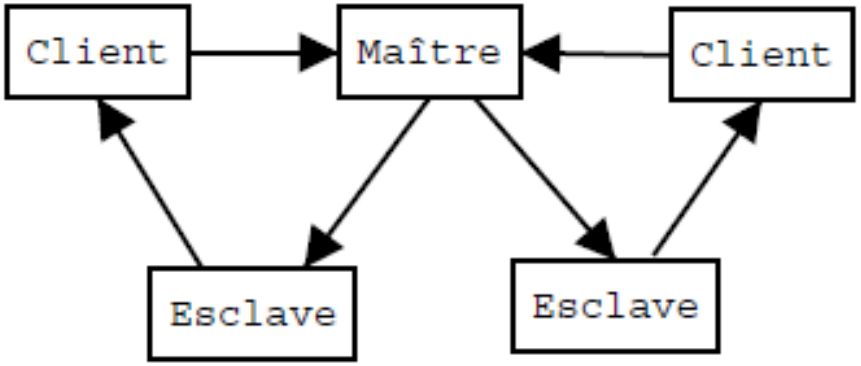
Un client, un serveur :



Un client, plusieurs serveurs :



Plusieurs clients, un serveur :



C/S orienté client ou serveur

Client lourd

- stocke les données et les applications localement. Le serveur stocke les fichiers mis à jour
- Le client effectue une bonne partie du traitement
- Le serveur est plus allégé

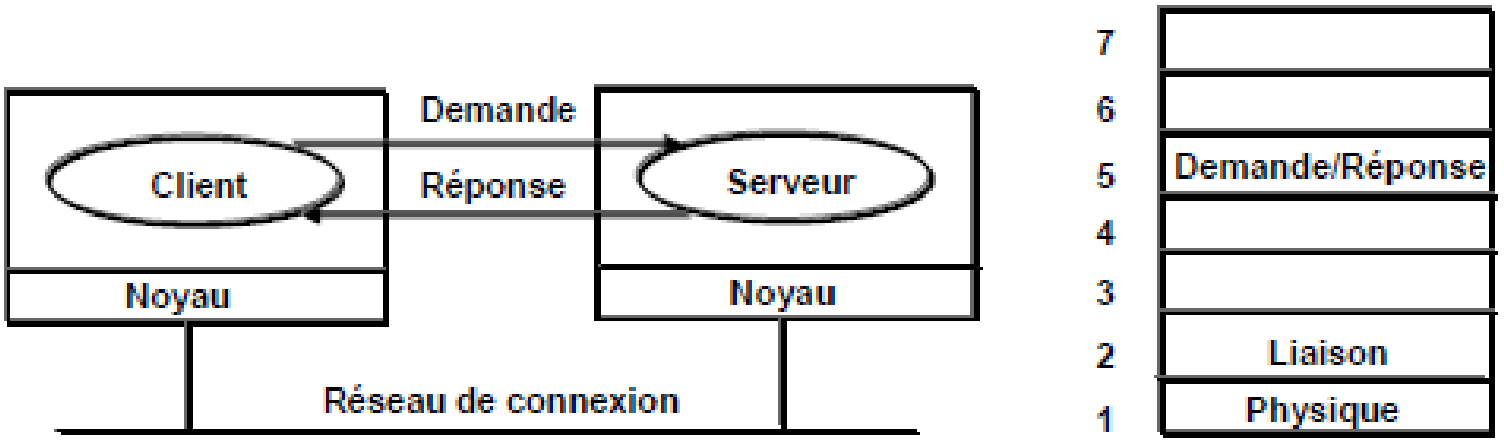
Serveur lourd

- On effectue plus de traitements sur le serveur : transactions, groupware, etc

Client léger

- Client à fonctionnalité minimale (terminaux X, périphérique réseau (Network Appliance), ordinateur réseau (network computer))
- Beaucoup de charge sur le serveur et le réseau

Description

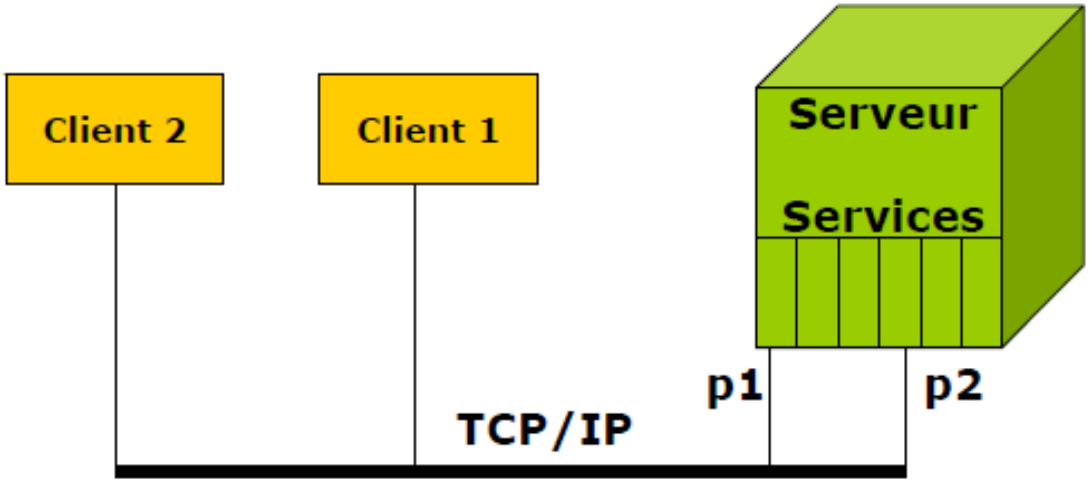


- Protocole simple sans connexion de type Question / Réponse
- Avantage : simplicité - efficacité
- Noyau : assurer l'envoi et réception des messages avec 2 primitives :

  - send (dest, &ptrm)

  - receive (adr, &prtm) (adr : l'adresse d'écoute du récepteur)

Principes

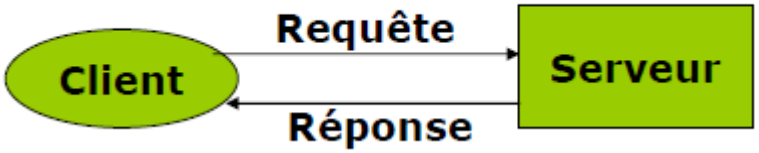


Client :
- émet des requêtes de demandes de service
- mode d'exécution synchrone
-le client est l'initiateur du
-dialogue client - serveur

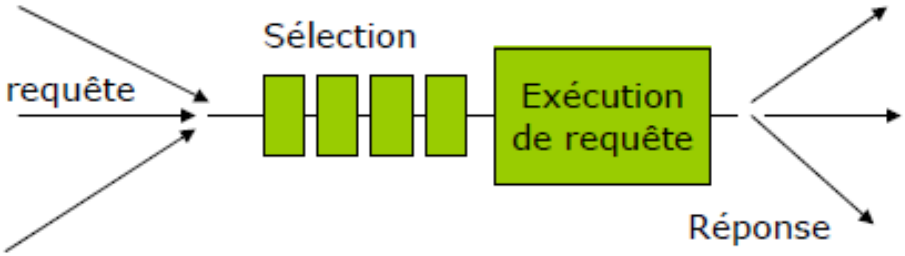
Serveur :
-ensemble de services exécutables
-exécute des requêtes émises par des clients
-exécution (séquentielle ou parallèle) des services

Principes

Client :
-Interface d'utilisateur
-Saisie de requête
-Envoie de requête
- attente du résultat
- affiche du résultat



Serveur :
- Gestion des requêtes (priorité)
- Exécution du service (séquentiel, concurrent)
- Mémorisation ou non de l'état du client



Deux modèles d'exécution :
- Client-Serveur à primitives : SEND et RECEIVE
- Client-Serveur à RPC (Remote Procedure Call)

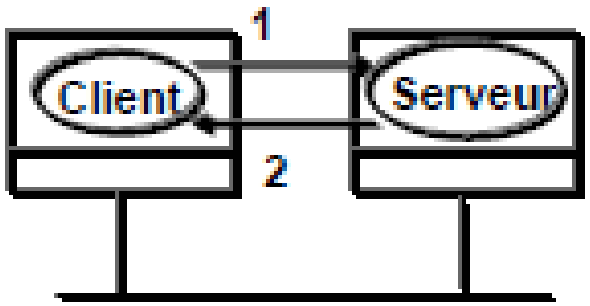
Messages client-serveur

- Trois grands types de message : REQ, REP et ACK
- Autres types possibles : AYA (Are You Alive), BUSY(ordinateur temporairement occupé), ERR (Erreur), etc.

Code	Type de paquet	source-dest.	Description
REQ	demande	client-serveur	le client désire un serveur
REP	réponse	serveur-client	la réponse du serveur au client
ACK	accusé de réception	noyau-noyau	le paquet précédent est arrivé
AYA	es-tu vivant ?	client-serveur	le serveur fonctionne-t-il ?
IAA	Je suis vivant	serveur-client	le serveur n'est pas en panne
TA	essaye à nouveau	serveur-client	le serveur est saturé
AU	adresse inconnue	serveur-client	aucun processus sur serveur utilise cette adresse

Modèle à primitifves: **Adressage**

- **Localisation connue**

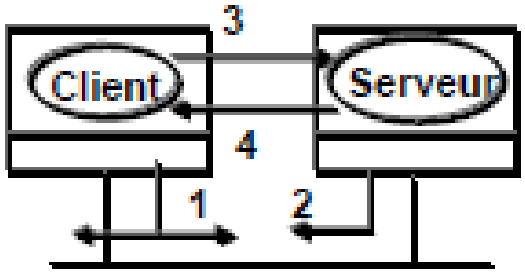


1 : demande vers S.p1
2 : réponse vers C.p2

- tout client doit connaître la localisation et le nom interne de tous les services. Il demande donc explicitement sous format: machine.processus

Modèle à primitifves:**Adressage**

- **Localisation inconnue**

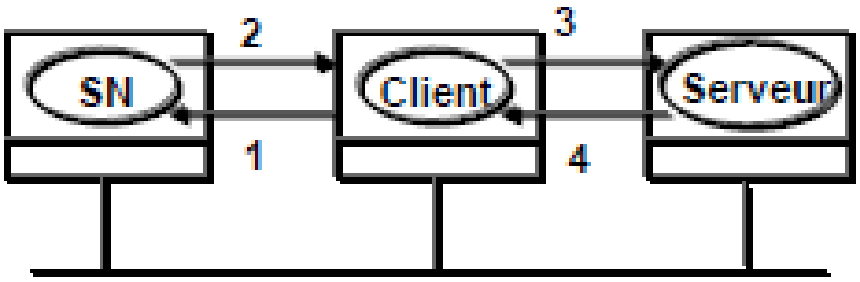


1 : recherche par diffusion
2 : je suis là
3 : demande
4 : réponse

- Chaque processus choisit un nom aléatoire telle sorte qu'il soit unique
- Client diffuse une demande de localisation d'un processus
- Le serveur contenant ce processus donne son adresse machine

Modèle à primitifves:Adressage

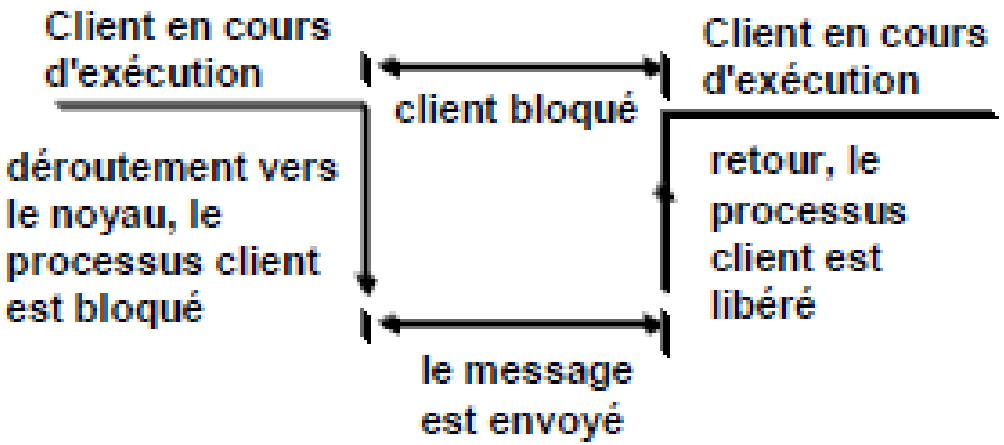
- **Localisation centralisée**



1 : demande à SN
2 : réponse de SN
3 : demande
4 : réponse

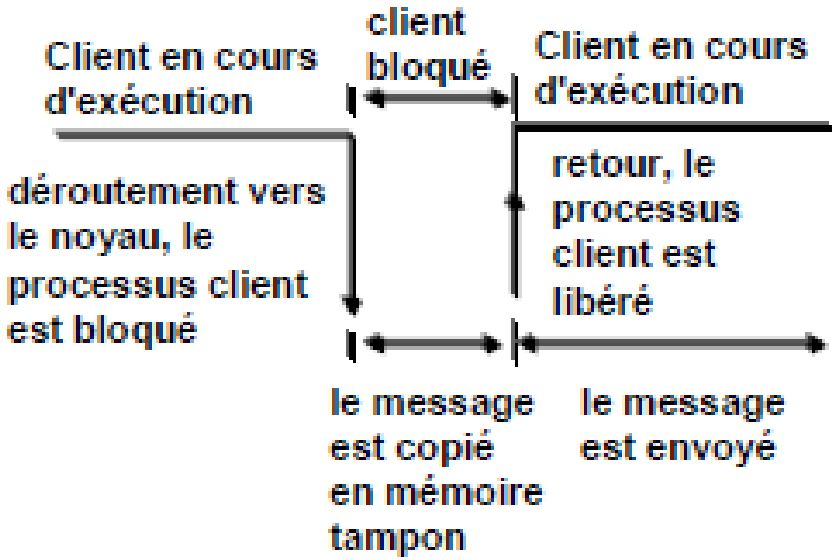
- les serveurs ont leur nom unique en code ascii
- un serveur gère la localisation et le nom interne des machines
- le client connaît seulement l'adresse du serveur de nom. Il lui demande de fournir l'adresse des serveurs à l'exécution

Primitives bloquantes et non bloquantes



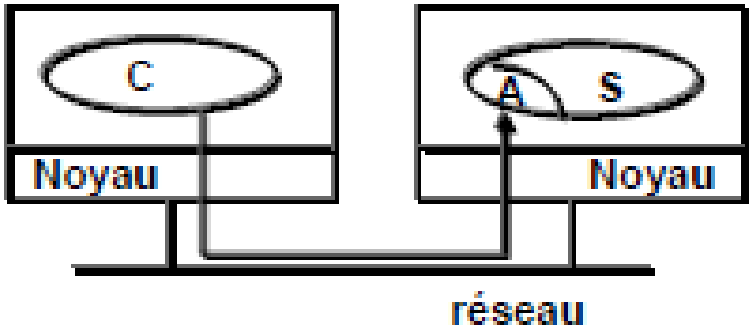
- le processus appelant send ou receive sera bloqué pendant l'émission (ou réception) du message
- l'instruction qui suit send (ou receive) sera exécutée lorsque le message a été complètement émis (ou reçu)
- avantage : simple et claire, facile à la mise en oeuvre
- inconvénient : moins performant

Primitives bloquantes et non bloquantes



- le processus appelant send sera libéré toute de suite, le message sera envoyé en // à l'exécution
- avantage : performant
- inconvénient : perte de mémoire tampon, difficile à réaliser, risque de modif. du tampon

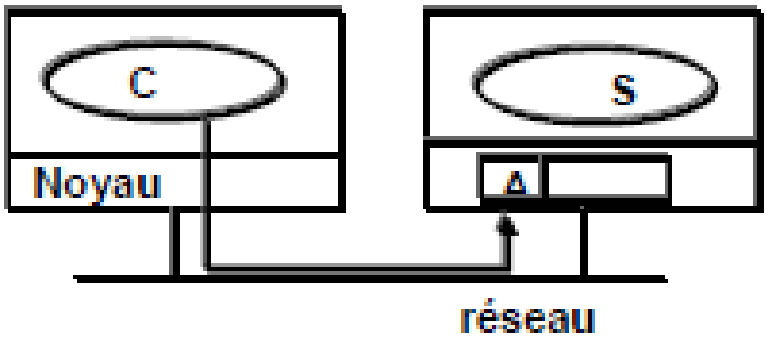
Primitives avec tampon ou sans tampon



- **sans tampon** : le message envoyé par le client sera copié par le noyau du serveur dans la zone ptrm et pour le processus A à l'adresse adr dans l'appel receive(adr,ptrm) déclenché par le processus A

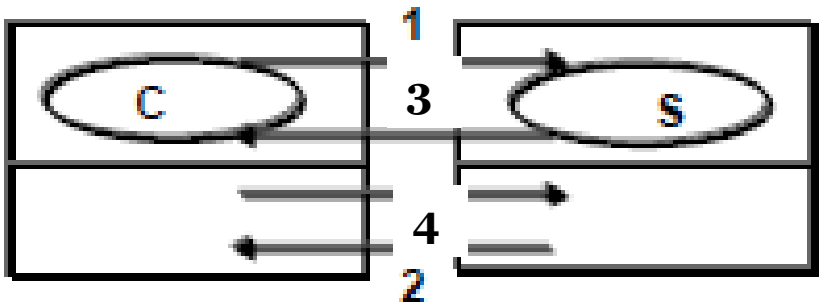
- **problème** : si le processus client a appelé send avant que le processus serveur appelle receive alors, le noyau du serveur n'a ni adresse du processus serveur ni l'adresse de l'endroit où il doit copier le message reçu !!!

Primitives avec tampon ou sans tampon



- **avec tampon** : les messages envoyés par les clients sont déposés dans une boite aux lettres. Le processus serveur A qui appelle un receive va déclencher une recherche par le noyau dans la boite aux lettres. Si le message existe, le noyau va le copier dans la zone de mémoire ptrm et libérer le processus A. Sinon l'appel receive sera bloqué pour l'attente de l'arrivée du message
- **problème** : quand la boite aux lettres est saturée, on doit refuser tous les messages qui arrivent !!!

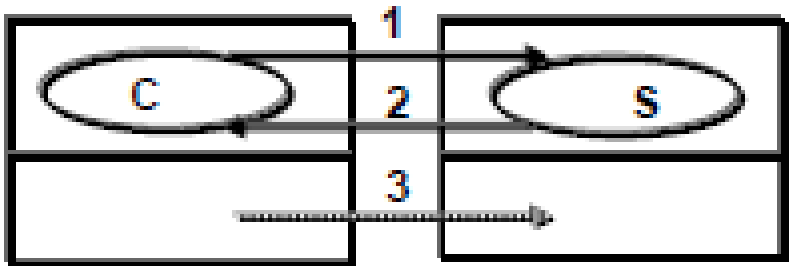
Primitives fiables et non fiables



Message avec accusés de réception individuels

1- Demande (client au serveur)
2- ACK (noyau à noyau)
3- Réponse (serveur à client)
4- ACK (noyau à noyau)

Primitives fiables et non fiables



Message avec la réponse = 1 accusé de réception

1- Demande (client au serveur)

2- Réponse (serveur à client)

3- ACK dans certain cas (noyau à noyau) (si le calcul est cher, on peut bloquer le résultat au serveur jusqu'à qu'il reçoit un ACK du client (3)

Primitives fiables et non fiables

On peut avoir un compromis entre 2 solutions précédentes :

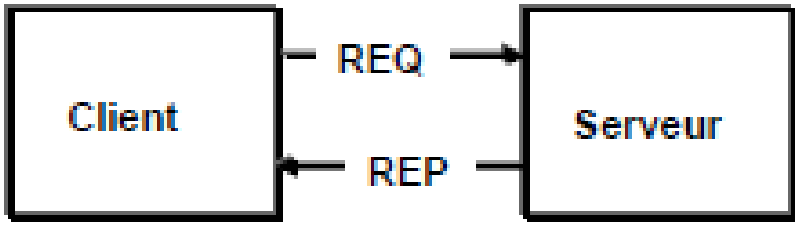
1- Demande (client au serveur)
2- ACK éventuellement si la réponse n'est pas revenue dans certain temps (noyau à noyau)
3- Réponse servant ACK si dans le lapes de temps prévu (serveur à client)
4- ACK éventuellement (noyau à noyau)

Modele Client/Serveur:Modèle à primitives

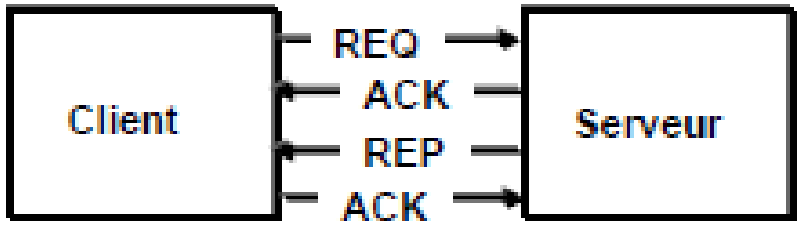
- Résumé

Objet	Option 1	Option 2	Option 3
Adressage	numéro de la machine	adresse choisie par le proc.	nom ASCII avec SN
Blocage	primitives bloquantes	primitives non-bloquantes avec copie vers le noyau	primitives non-bloquantes avec interruption
Mémorisation	pas de tampon	tampon temporaire	boite aux lettre
Fiabilité	non fiable	demande-ACK-réponse-ACK	demande--réponse-ACK

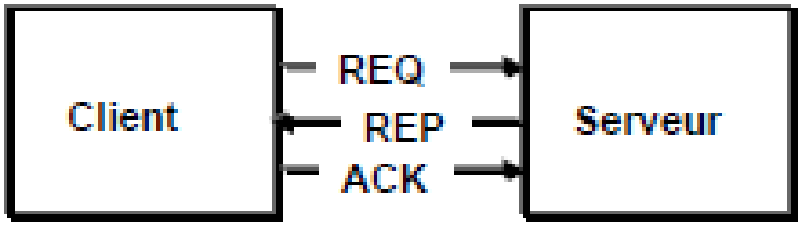
Exemples des échanges des paquets dans le protocole Client/Serveur



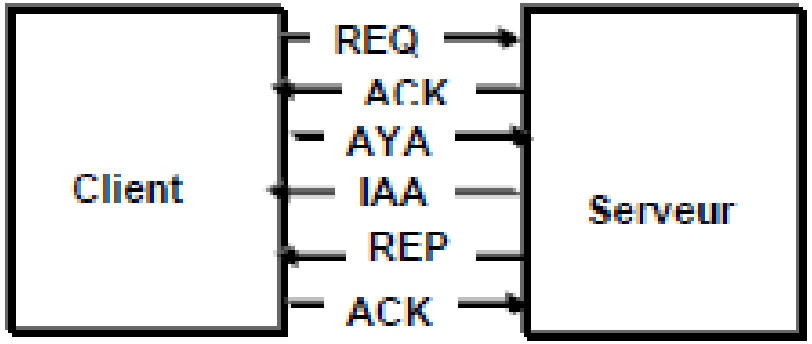
a)



b)



c)



d)

Les Sockets

Serveur en gestion multi-clients

- Par principe, les éléments distants communiquants sont actifs en parallèle
- Plusieurs processus concurrents
- Avec processus en pause lors d'attente de messages

Concurrence

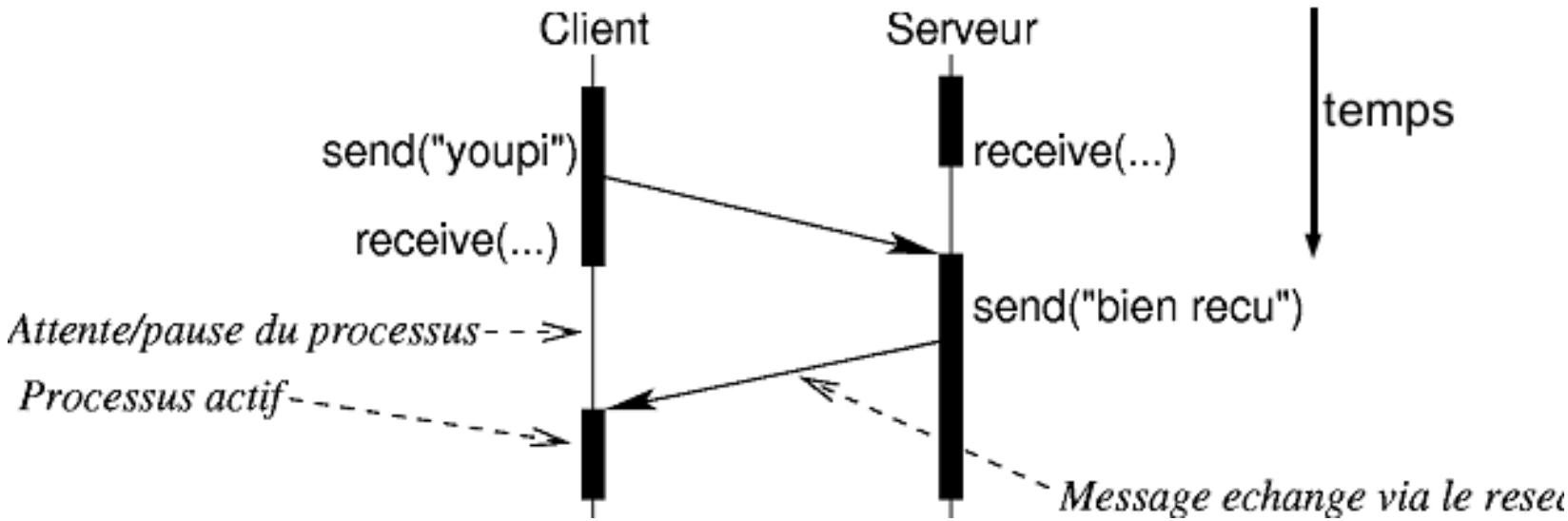
Gestion plusieurs clients

- Particularité coté serveur en TCP
- Une socket d'écoute sert à attendre les connexions des clients
- A la connexion d'un client, une socket de service est initialisée pour communiquer avec ce client
- Communication avec plusieurs clients pour le serveur
- Envoi de données à un client
  UDP : on précise l'adresse du client dans le paquet à envoyer
  TCP : utilise la socket correspondant au client
- Réception de données venant d'un client quelconque
  UDP : se met en attente d'un paquet et regarde de qui il vient
  TCP : doit se mettre en attente de données sur toutes les sockets actives

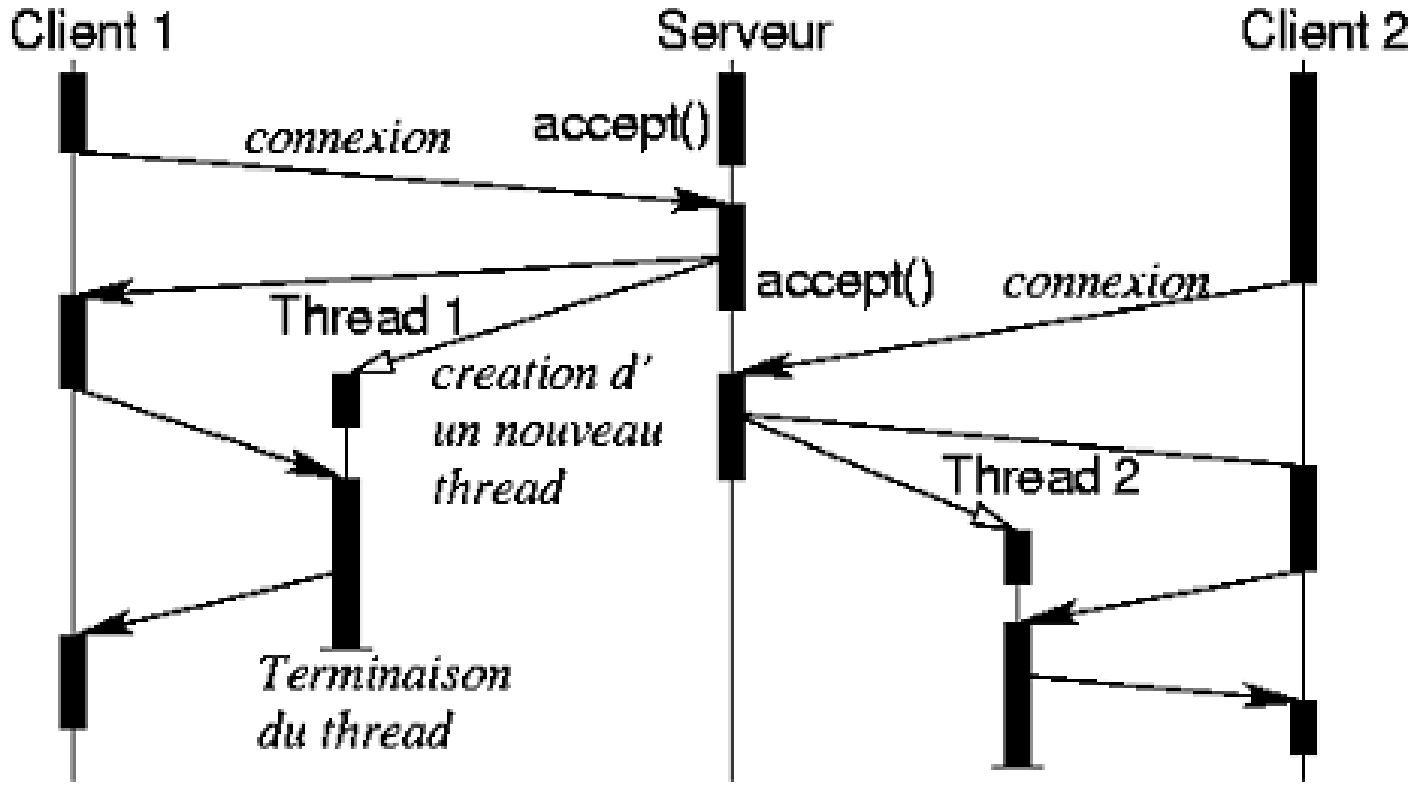
Gestion plusieurs clients

- Contrainte
- Lecture sur une socket : opération bloquante: Tant que des données ne sont pas reçues
- Attente de connexion : opération bloquante: Jusqu'à la prochaine connexion d'un client distant
- Avec un seul flot d'exécution (processus/thread)
- Si ne sait pas quel est l'ordonnancement des arrivées des données des clients ou de leur connexion au serveur: impossible à gérer
- Donc nécessité de plusieurs processus ou threads
- Un processus en attente de connexion sur le port d'écoute
- Nouvelle connexion : un nouveau processus est créé pour gérer la communication avec le nouveau client

Gestion plusieurs clients



Gestion plusieurs clients (suite)



Appels de procédures à distance (RPC)

Modèle RPC :Introduction

- **Problème du modèle Client/Serveur**

Concept basé sur l'échange explicite de données donc orienté Entrées/Sorties qui ne peut pas être le principe clé d'un système distribué

- **Appels de procédures à distance**

Concept introduit par Birrell et Nelson en 1984 : lorque un processus sur la machine A appelle une procédure sur une machine B, le processus sur A est suspendu pendant l'exécution de la procédure appelée qui se déroule sur B. Des informations peuvent être transmises de l'appelant vers l'appelé ou l'inverse par des paramètres

Qu’est ce que RPC?

- Les RPC (Remote Procedure Call : Appels de Procédures Distantes) ont été introduits par SUN. Ils sont basés sur les mécanismes de client-serveur. Ils perme ttent l'appel et l'exécution de procédures sur des machines distantes.

- Les RPCs sont construits au-dessus des sockets dont les mécanismes sont masqués au programmeur, présentant ainsi une interface de plus haut niveau.

Qu’est ce que RPC?

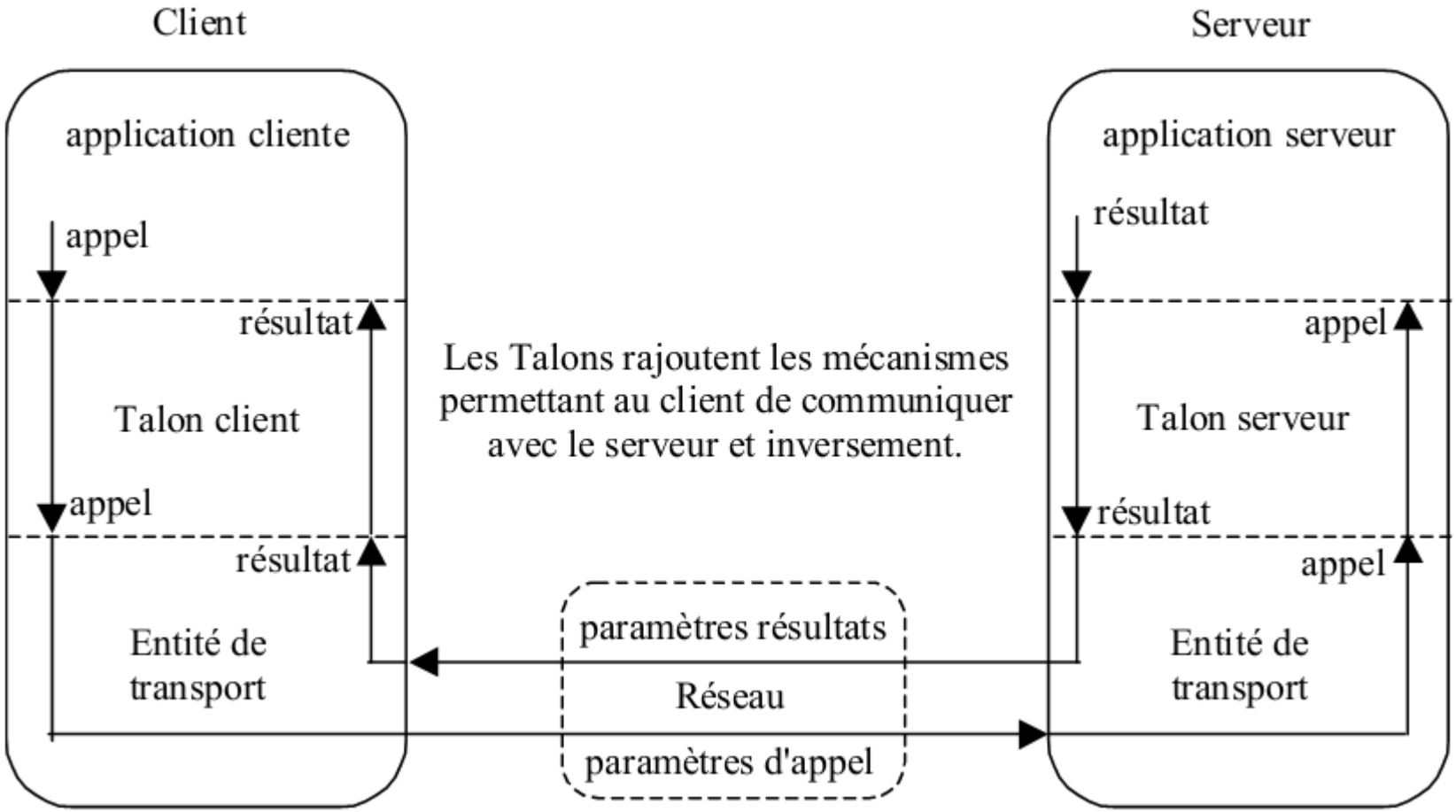
- Vous pouvez utiliser les RPC pour toutes sortes d'applications distribuées, par exemple l'utilisation d'un ordinateur très puissant pour des calculs intensifs (décryptage de données, calculs numériques ...). Cet ordinateur sera donc le serveur. Un autre ordinateur sera le client et appellera la procédure distante pour commander des calculs au serveur et récupérer le résultat

Principe de fonctionnement

- Il existe dans le programme client une fonction locale qui a le même nom que la fonction distante et qui, en réalité, appelle d'autres fonctions de la bibliothèque RPC qui prennent en charge les connexions réseaux, le passage des paramètres et le retour des résultats.

- Côté serveur, il suffira (à quelques exceptions près) d'écrire une fonction comme on en écrit tous les jours, un processus se chargeant d'attendre les connexions clientes et d'appeler votre fonction avec les bons paramètres. Il se chargera ensuite de renvoyer les résultats.

Fonctionnement détaillé d'un RPC



- Le programme client appelle un sous-programme à travers le réseau. Pour le client, tout se passe comme si l'appel était local. En effet le modèle RPC est similaire au modèle d'appel de procédure locale. Dans le cas local, les arguments d'appel sont stockés dans la pile (ou dans des registres) puis le client donne la main à la procédure avec éventuellement un contrôle sur le transfert des données. Ensuite, le résultat de la procédure est extrait de la pile (ou d'un registre) et le client continue son exécution.

- Les RPCs sont similaires : Le processus appelant envoie une requête et attend la réponse. Le message de requête contient entre autres les paramètres d'appel de procédure et la réponse, ceux de retour s'il y a lieu. Côté serveur, le processus est en attente d'une requête. Lors de l'arrivée d'une requête, il extrait les paramètres d'appel, exécute la fonction et récupère le résultat qu'il renvoie. Enfin, il se remet en attente de requête.

Fonctionnement d'une RPC

Les RPCs se décomposent en trois parties principales :

- La partie cliente, dite aussi "l'appelant", est l'application qui utilise une ou plusieurs procédures distantes.
- L'interface réseau, découpée en deux parties souvent appelées des Talons (« stubs » en anglais), est la partie qui gère tous les aspects distribués.
- La partie serveur, dite aussi "l'appelé", contient les procédures distantes.

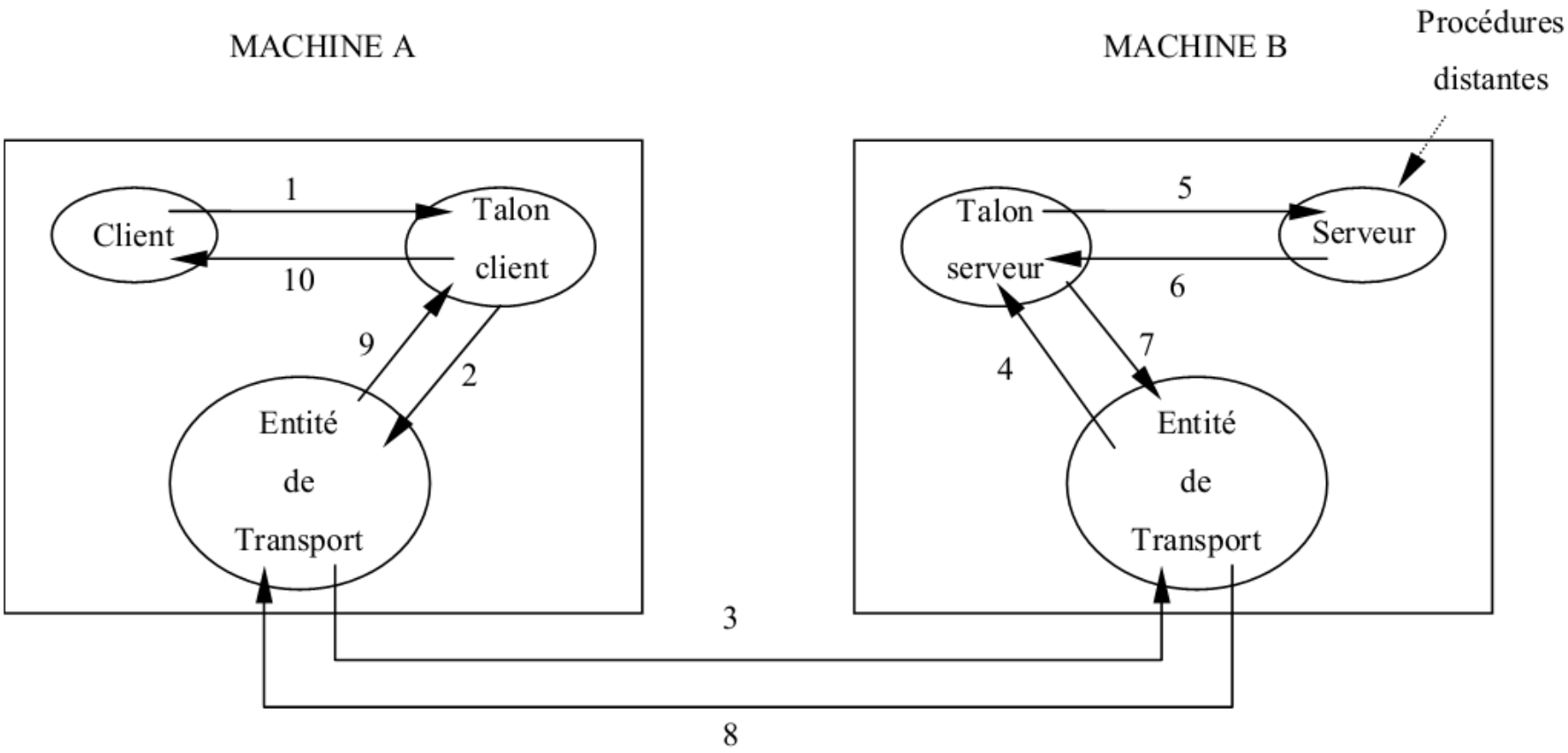
Principe de fonctionnement

- Les fonctions qui prennent en charge les connexions réseaux sont des « **stub** ». Il faut donc écrire un **stub client** et un **stub serveur** en plus du programme client et de la fonction distante.

- Le travail nécessaire à la construction des stubs client et serveur sera automatisé grâce au programme **rpcgen** qui produira du code C qu'il suffira alors de compiler.

- Il ne restera plus qu'à écrire le programme client, qui appelle la fonction et en utilise le résultat (par exemple en l'affichant), et la fonction elle-même.

Fonctionnement détaillé d'un RPC



Fonctionnement détaillé d'un RPC

1. Appel de la procédure Talon_client avec les paramètres (Il peut exister plusieurs procédures Talon, ce qui permet de différencier les procédures distantes à appeler).
2. La procédure talon construit un message réseau da ns lequel elle place les paramètres utilisateur :

phase de codage.

3. Transmission sur le réseau.
4. La procédure Talon_serveur décode le message.
5. La procédure serveur appelle la fonction demandée avec les para mètres qui viennent d'être extraits

du message.

Fonctionnement détaillé d'un RPC

6. La procédure Talon_serveur récupère le (ou les) résultat(s) de l'exécution de la fonction demandée.
7. La procédure Talon_serveur construit un message dans lequel elle place les éventuels résultats.
8. Transmission sur le réseau.
9. La procédure Talon_client décode le message.
10. La procédure Talon_client retourne les résultats au client qui reprend son exécution.

Le passage de paramètres

- Le passage de paramètres dans le cadre des RPCS n'est pas forcément quelque chose de simple.
- Dans les systèmes répartis, les machines peuvent être différentes. Des problèmes de conversion peuvent apparaître.

- La problématique va donc être de connaître le format du message et la plateforme cliente :

• Le premier octet (Byte) du message détermine le format utilisé par le client.

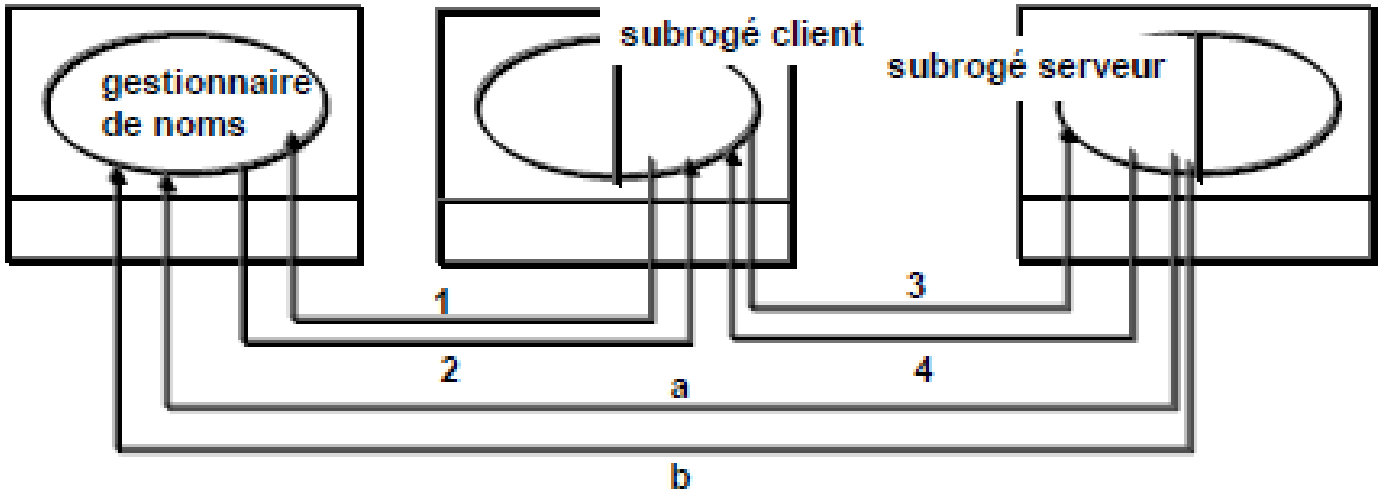
• Le serveur compare cet octet avec le sien : Si c'est le même aucune transformation n'est nécessaire, sinon il effectue la transformation.

Localisation du serveur

- La localisation d'un serveur peut se faire de différentes manières :
- Statique : Codée en dur l'adresse du serveur dans le client. En cas de changement de l'adresse, le client doit être recompilé (les programmes qui accèdent au serveur).
- Dynamique : lien dynamique (dynamic binding). Ce lien permet de faire dynamiquement connaître les clients et les serveurs.

Nommage dynamique (dynamic binding)

- Problème de la localisation du serveur par le client : écrire dans le code client l'adresse du serveur ou utiliser le mécanisme de nommage dynamique



a : enregistrer un nom (exportation de l'interface)
b : supprimer un nom
1- recherche un nom (importation)
2- retourne l'identificateur et descripteur
3 et 4- RPC

Le lien dynamique

La construction du lien dynamique s'effectue en plusieurs étapes.

Définition de la spécification du serveur qui servira à la génération des souches.

À l'initialisation, le serveur exporte son interface. Il l'envoie à un binder (relieur) pour signaler son existence.

C'est la phase d'enregistrement. Le serveur envoie ses informations :

- nom ;
- version identifiant (unique sur 32 bits en principe) ;
- handle (adresse IP, protocoles, etc.).

Lorsqu'un Client appelle une procédure pour la première fois :

- Il constate qu'il n'est pas relié à un serveur.
- Il envoie un message au relieur pour importer la version de la procédure qu'il désire invoquer.
- Le relieur lui renvoie l'identifiant unique (ID) et le handle.

Le lien dynamique

Avantage :

- Cette méthode d'importation et exportation des interfaces est très flexible.
- Les serveurs peuvent s'enregistrer ou se désenregistrer.
- Le client fournit le nom de la procédure et la version et il reçoit un ID unique et handle.

L’outil jrpcgen

C’est le compilateur qui permet de créer automatiquement les stubs (client et serveur) à partir d’un fichier en langage RPCL

RPCGEN

- Pour éviter d'avoir à programmer la partie réseau de l'application, en particulier celle concernant le serveur, pour faciliter la production des filtres XDR, il existe un utilitaire d'aide au développement des programmes utilisant les RPC.
- Cet outil s'appelle **rpcgen** (cf. man rpcgen).Il s'agit d'un précompilateur qui engendre plusieurs fichiers à partir d'une description synthétique des services, description donnée dans un fichier dont le type est x. Le langage de description ressemble au langage C.

RPCGEN

Exemple:

si le fichier de description s'appelle fichier.x, les fichiers produits par l'exécution de la commande : rpcgen fichier.x seront les suivants :

- un fichier à inclure dans les programmes du serveur et des clients (fichier.h),
- un squelette du code du serveur (fichier_svc.c),
- les procédures réalisant les appels distants pour les clients (fichier_clnt.c),
- les filtres xdr (fichier_xdr.c).
- Il reste donc à écrire deux fichiers : un fichier contenant le code des procédures fournies par le serveur qui complétera fichier_svc.c, lui-même contenant déja la fonction main,
- le code du client qui se servira de fichier_clnt.c, ici on doit écrire la fonction main.

Les problèmes liés aux RPCs dans les systèmes répartis.

1. Le client est incapable de localiser le serveur
2. La demande du client au serveur est perdue
3. La réponse du serveur au client est perdue
4. Le serveur tombe en panne après avoir reçu une demande
5. Le client tombe en panne après avoir envoyé une demande

Les problèmes liés aux RPCs dans les systèmes répartis.

Le serveur est en panne :

- Il faut dissocier la panne avant l'exécution de la requête ou après l'exécution.

Trois écoles pour résoudre ce problème :

1. Attendre que le serveur redémarre et relancer le traitement. « Au moins un traitement est réussi. » (peut-être plus).
2. Abandon du client et rapport de l'erreur. « Au plus un traitement est réussi. » (peut-être aucun).
3. Ne rien dire au client (aucune garantie) Seuls avantages : Facile à gérer et à implémenter. En règle générale, **crash du serveur = crash du client.**

Les problèmes liés aux RPCs dans les systèmes répartis.

Le client est en panne :

Lorsqu'un client tombe en panne alors qu'il a demandé un traitement à un serveur, on dit que l'échange devient orphelin. Cela a pour effet de gaspiller du temps CPU, de bloquer des ressources, lorsqu'il redémarre, il peut recevoir des messages antérieurs à la panne (problème de confusion).

Les problèmes liés aux RPCs dans les systèmes répartis.

Quatre solutions peuvent être envisagées :

1. **L'extermination** : Le client enregistre les appels dans une log. Au redémarrage l'orphelin est détruit. Beaucoup d'écritures disque à chaque RPC.
2. **La réincarnation** : Le client tient compte du temps. Lorsqu'il redémarre, il annule les échanges ente deux époques et reprend les échanges orphelins.
3. **La réincarnation à l'amiable** : Comme précédemment, avec en plus la vérification avec le serveur concerné des échanges orphelins.
4. **Expiration du délai** : Un compteur de temps détermine quand un échange devient orphelin. Les orphelins sont détruits. Un temps T déterminé identique pour tous est donné à 1 RPC. Difficile de choisir un temps T (les RPCs sont sur des réseaux différents).

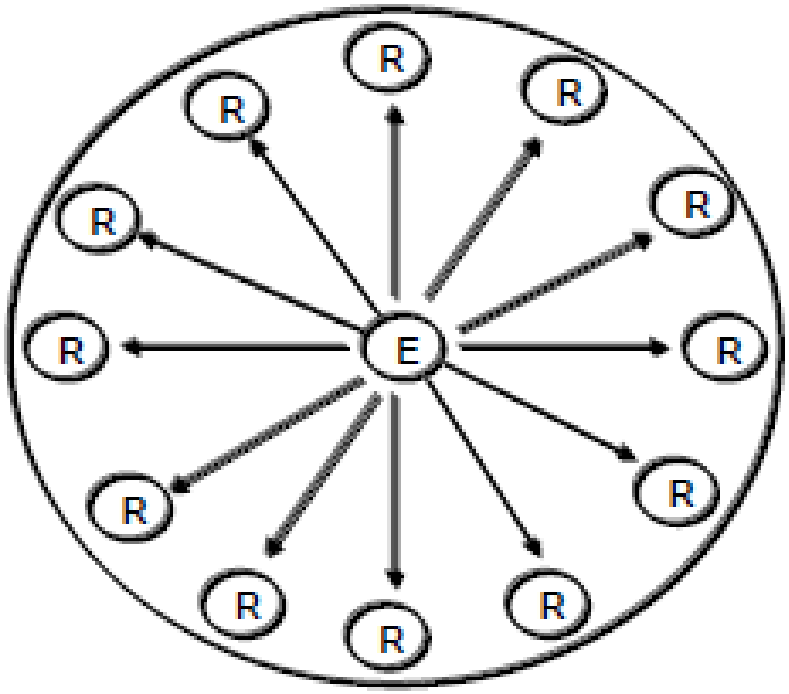
La communication de groupe

Introduction

Point-à-Point

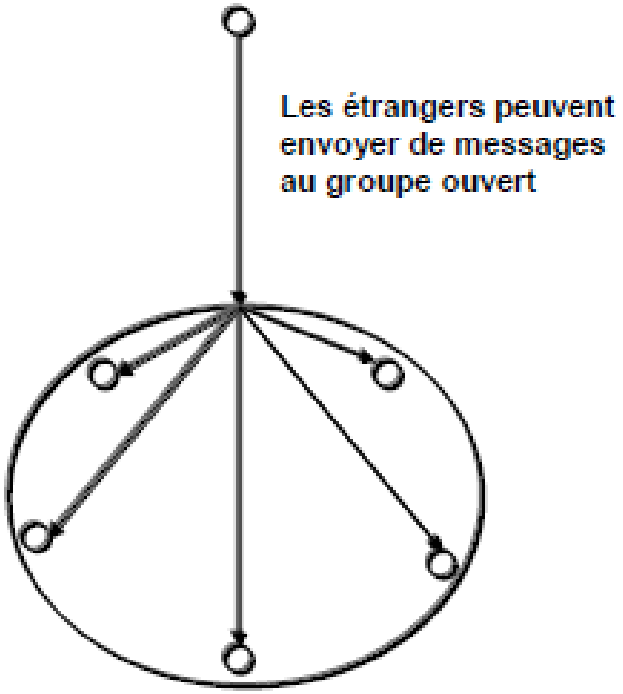
Généralisation du RPC

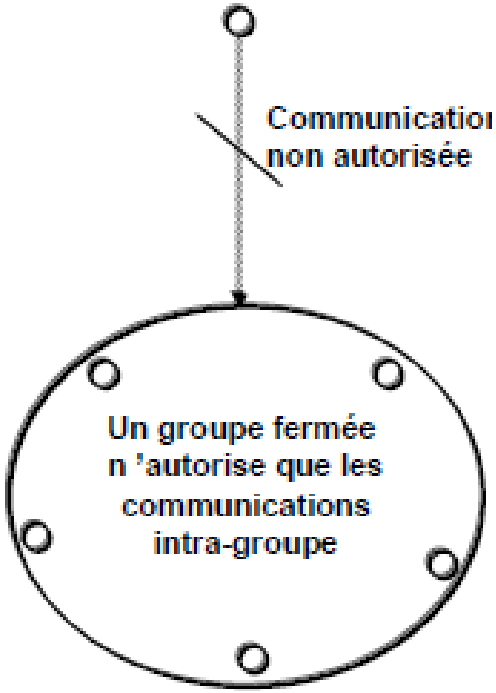
Un émetteur doit communiquer en parallèle à n récepteurs regoupés dans un groupe sémantique



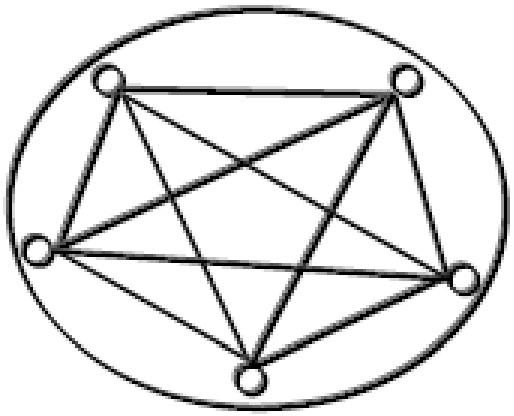
Un-à-Plusieurs

Classement
a) Groupes fermés et groupes ouverts

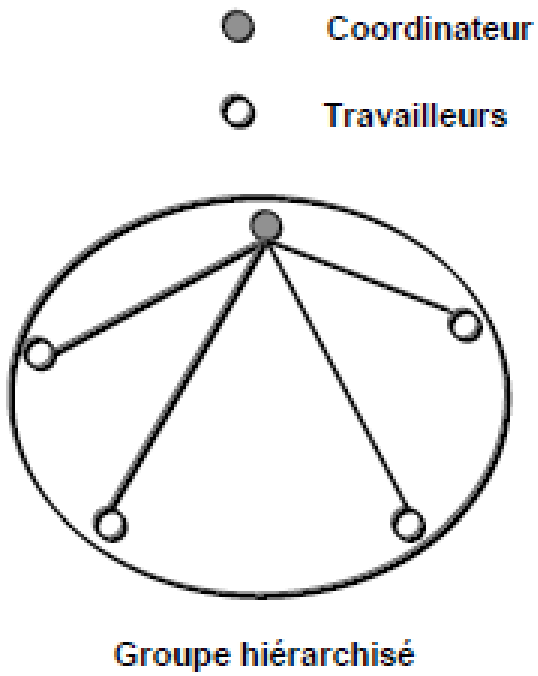




Classement
b) Groupes de paires ou groupes hiérarchisés



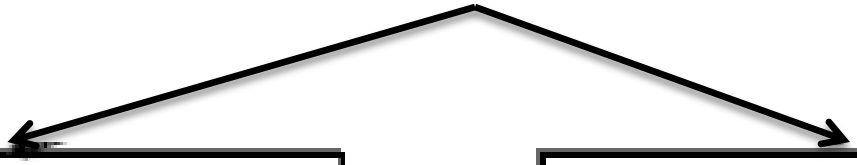
Groupe de paires



Groupe hiérarchisé

Gestion des groupes

-Création, dissolution d 'un groupe
-Adhésion à un groupe ou retrait d 'un groupe



Gestion centralisée

- Un serveur de groupes qui gère une base de données des groupes et une base de données des membres

- Avantages : efficace, claire et facile à implanter

- Inconvénients : fragile devant les pannes

Gestion distribuée

- Un groupe se crée s 'il y a au moins un adhérant
- pour adhérer à un groupe, le processus demandeur doit envoyer sa demande à tous les membres du groupe

- Avantage : Robustesse
- Inconvénient : complexe et coûteuse

Adressage des groupes

Diffusion restreinte
(multicasting)

Diffusion globale
(broadcasting)

monodifusion
(point-to-point)

Atomicité

- Propriété de diffusion atomique (atomic broadcast)

Un message envoyé à un groupe doit être reçu par tous ses membres ou par aucun

Algorithme de « relais intra-groupe » (Joseph et Birman 1989)

1- L 'expéditeur envoie un message à tous les membres d 'un groupe (des temporisateurs sont armés et des retransmissions faites si nécessaire)

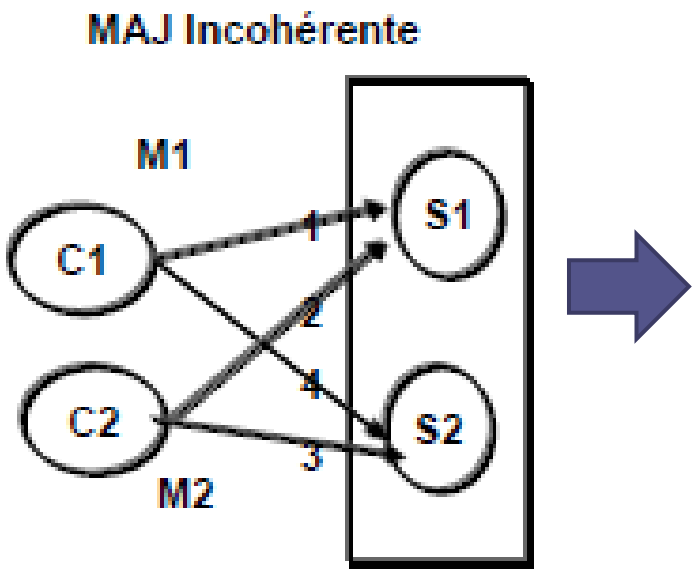
2- Tout membre de ce groupe, qui a reçu ce message pour la première fois, doit l 'envoyer à tous les membres du groupe (des temporisateurs sont armés et des retransmissions faites si nécessaire)

3- Tout membre de ce groupe, qui a reçu ce message plus qu 'une fois, doit simplement le détruire

Séquencement

Propriété de séquencement

Soient A et B 2 messages à envoyer à un groupe de processus G. Si A est envoyé avant B sur le réseau, ils doivent être reçus par tous les membres du groupe G dans le même ordre : A puis B

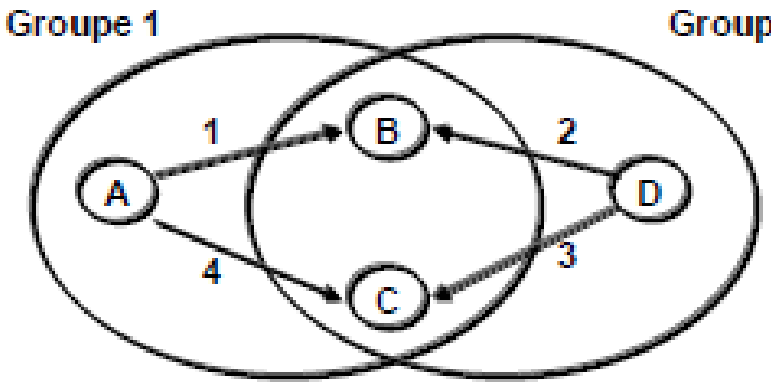


Solution

-Séquencement global (global time ordering) : conserver, en réception, l 'ordre des messages à émission.
Si C1 envoie M1 avant que C2 envoie M2, alors le système doit assurer que tous les membres de G reçoivent M1 avant M2

- Séquencement cohérent (Consistent time ordering) :
Le système établi l 'ordre d 'envoie des messages et assure que cet ordre soit respecté à la réception des messages.

Groupes non disjoints



L 'ordre de recouvrement des groupes non respecté :

- B reçoit le message de A puis D
- C reçoit le message de D puis A

Si B et C est une base de données dupliquée on peut avoir une incohérence de MAJ

Solution:

Introduction des mécanisme de séquencement qui tient compte du recouvrement des groupes

Probléme:

La mise en oeuvre est compliquée et coûteuse

Middleware

Middleware ou intergiciel

- **Motivations**

Dans un systeme reparti, meme l'interface fournie par les systemes d'exploitation et de

communication est encore trop complexe pour etre utilisee directement par les applications.

- Heterogeneite
- Complexite des mecanismes (bas niveau)
- Necessite de gerer (et de masquer, au moins partiellement) la repartition

Solution

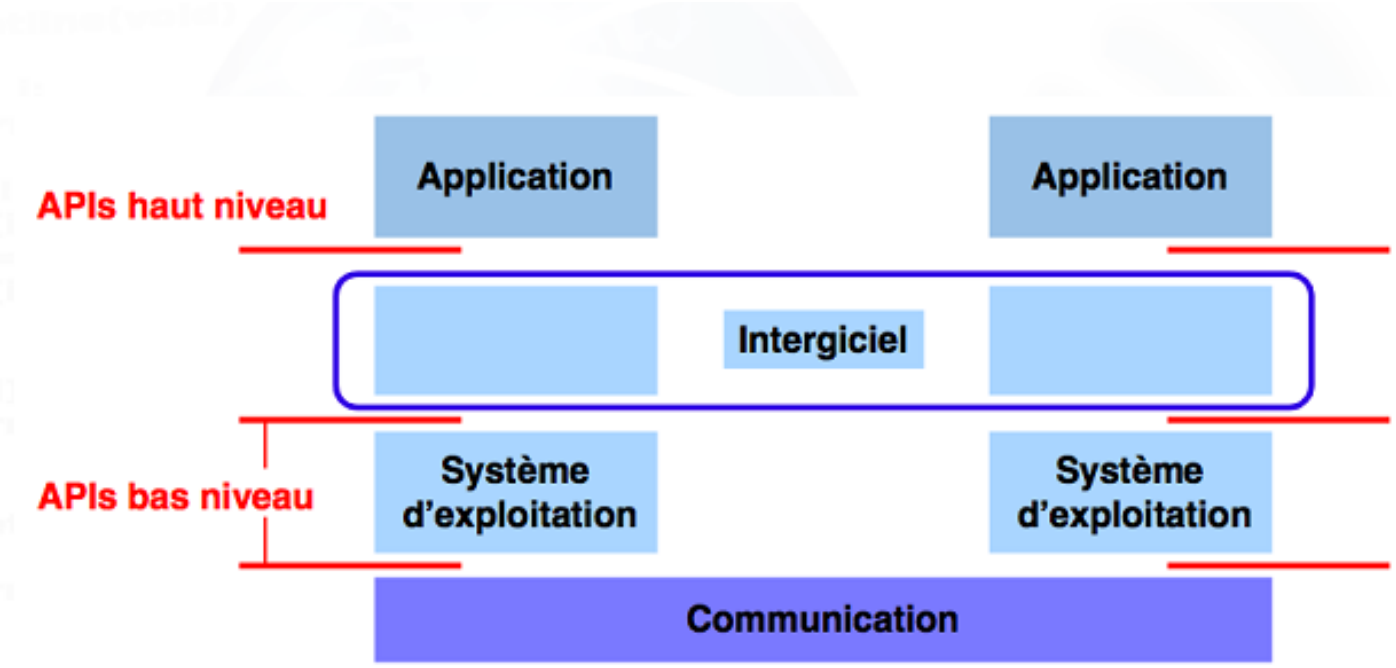
Introduire une couche de logiciel intermediaire (repartie) entre les niveaux bas (systemes et communication) et le niveau haut (applications) : c'est l'intergiciel

L'intergiciel joue un role analogue a celui d'un "super-systeme d'exploitation" pour un systeme reparti

Middleware ou intergiciel

- Couche logiciel
- S'intercale entre le système d'exploitation/réseau et les éléments de l'application distribuée
- Assure les connexions entre les serveurs de données et les outils de développement sur les postes clients
- Ensemble de services logiciels construits au dessus d'un protocole de transport afin de permettre l'échange de requêtes et des réponses associées entre client et serveur *de manière transparente*.

Middleware ou intergiciel



Types de middleware

Général

- Protocoles de communication, répertoires répartis, services d'authentification, service de temps, RPC, etc
- Services répartis de type NOS (Networked OS) : services de fichiers, services d'impression.

Spécifique

- de BD : ODBC, IDAPI, EDA/SQL, etc
- de groupware : Lotus Notes
- d'objets : CORBA, COM/DCOM, .NET

Composantes du middleware

Les canaux

- Services de communications entre composants et applications : RPC (synchrone), ORB (synchrone), MOM (Message Oriented Middleware) (asynchrone) Services de support de communication : SSL, annuaires (LDAP)

Les plate-formes

- Serveurs d'applications qui s'exécutent du côté serveur
- Offrent les canaux de communication
- Assurent la répartition, l'équilibrage de charge, l'intégrité des transactions, etc

Fonctions d'un middleware

L'intergiciel a quatre fonctions principales

- Fournir une interface ou API (Applications Programming Interface) de haut niveau aux applications
- Masquer l'heterogeneite des systemes materiels et logiciels sous-jacents
- Rendre la repartition aussi invisible ("transparente") que possible
- Fournir des services repartis d'usage courant

L'intergiciel vise a faciliter la programmation repartie

- Developpement, evolution, reutilisation des applications
- Portabilite des applications entre plates-formes
- Interoperabilite d'applications heterogenes

Merci pour votre attention!

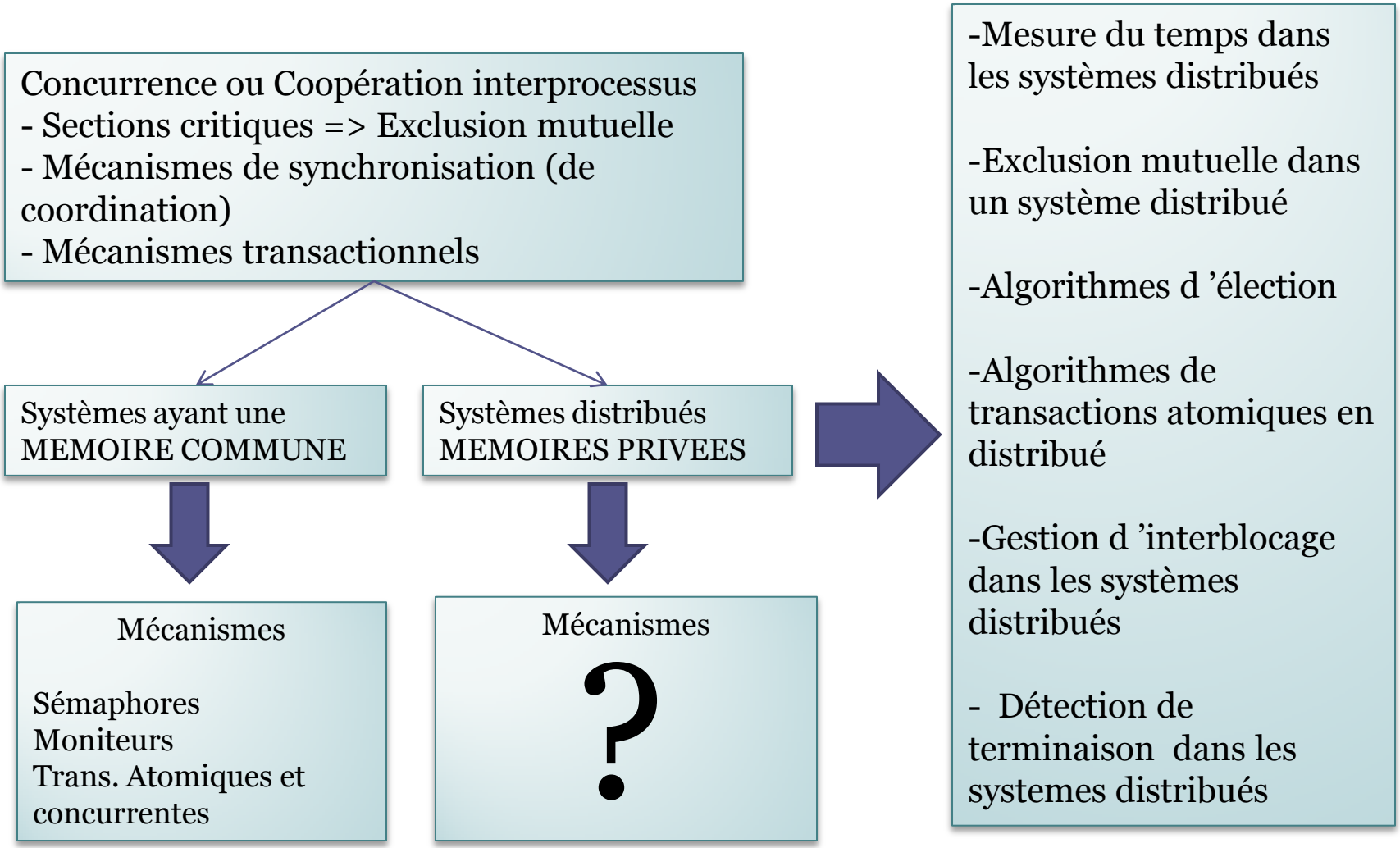
Synchronisation dans les systèmes distribués

Chapitre 3

Plan

- Synchronisation d'horloge
- Exclusion-mutuelle
- Algorithmes d'élection
- Transactions atomiques
- Détection de terminaison

Introduction



Synchronisation d'horloge

Introduction

Propriétés d'algorithmes distribués

1- les informations importantes sont disséminées sur plusieurs Machines

2- les processus prennent des décisions fondées uniquement sur l'information disponible Localement

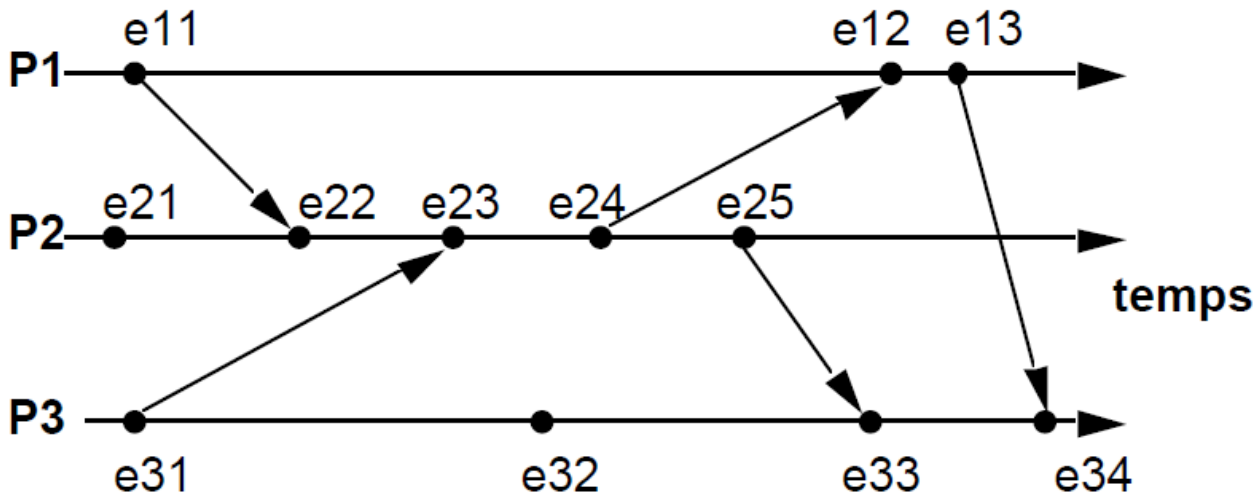
3- la garantie de la fiabilité doit être renforcée

4- il n 'existe pas d 'horloge commune ni de source de temps universel

Synchronisation de l'Horloge :But

- Synchronisation de l'Horloge dans un système distribué a pour but :
  - Ordonnancement temporel d'événement qui se produisent dans des processus concurrent
  - Synchronisation entre l'émetteur et le récepteur d'un message
  - Synchronisation entre plusieurs processus pour le déclanchement d'une action commune
  - Sérialisation des accès concurrent sur une ressource partagée

Un modèle pour l'étude de la synchronisation dans un système réparti



Système *= ensemble de processus (1 par site) + canaux de communication*
Processus *= séquence d'événements locaux + mémoire locale + horloge locale*
Evénement *= changement d'état du processus (événement interne), ou émission d'un message, ou réception d'un message*

Calculs et observations distribués

- *Observation d'un calcul réparti E = "linéarisation" de E (définition d'un ordre total, sur les événements de E)*

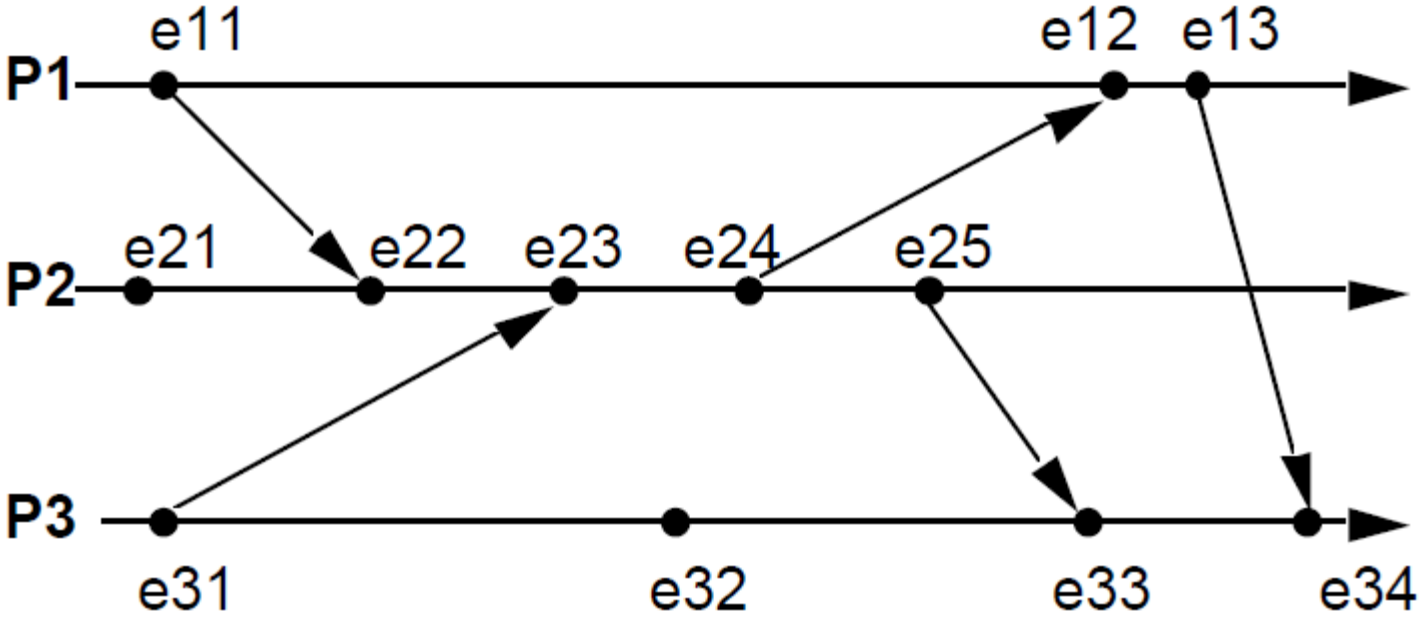
- *Exemple : vue de la séquence des événements de E par un observateur interne ou externe au système*

- *Observation valide = compatible avec la relation de précédence causale (pourrait être observée par un observateur externe réel)*

Calculs et observations distribués

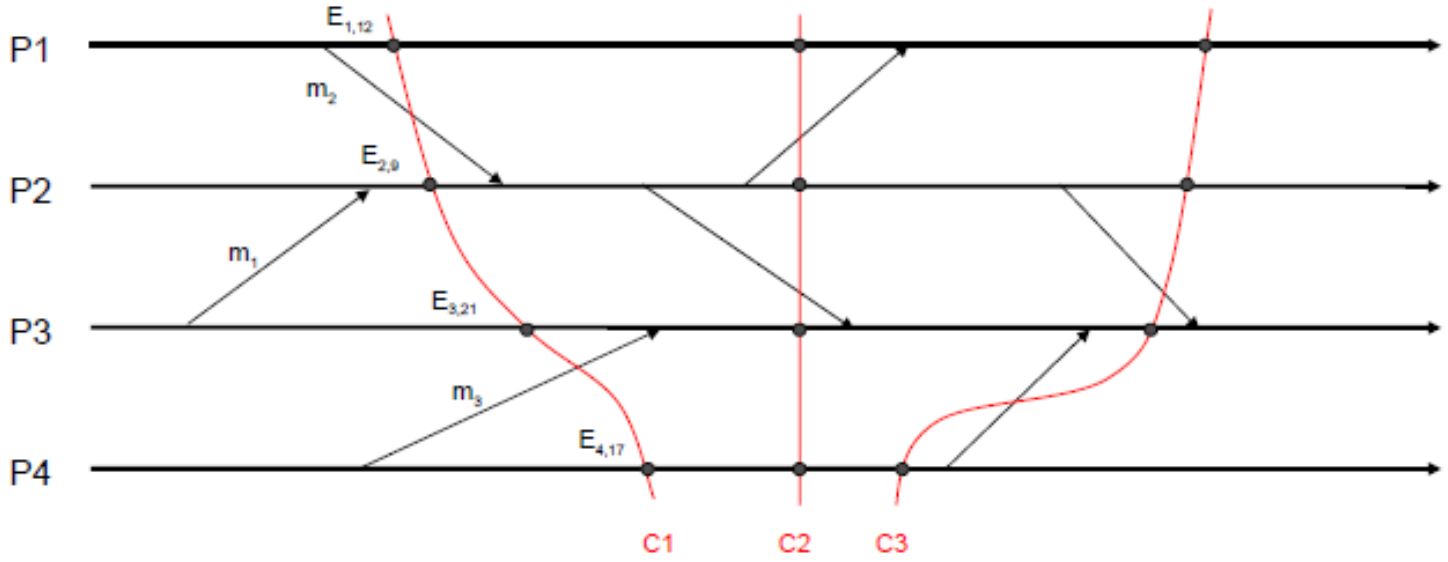
- ***Exemple :***

- *e11 e21 e31 e22 e23 e32 e24 e25 e12 e33 e13 e34 valide*
- *e11 e21 e31 e22 e32 e23 e24 e25 e12 e33 e13 e34 Valide*
- *e11 e21 e31 e22 e23 e32 e24 e25 e12 e33 e34 e13 invalide*



Etat global

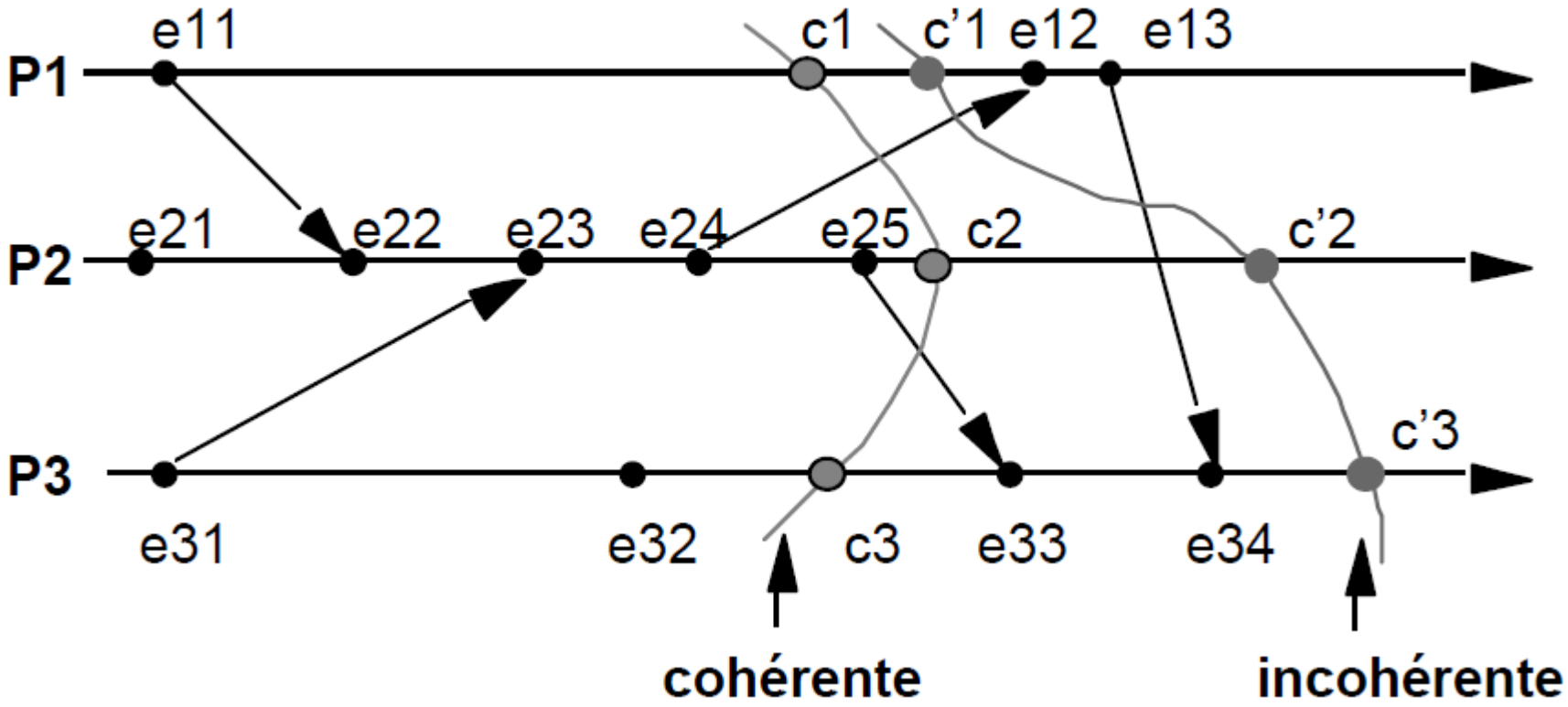
Un état global peut être représenté par une courbe, appelée aussi **coupure**, sur le diagramme temporel du système réparti. La coupure divise le diagramme en deux zones (sur chaque processus) : passé et futur.



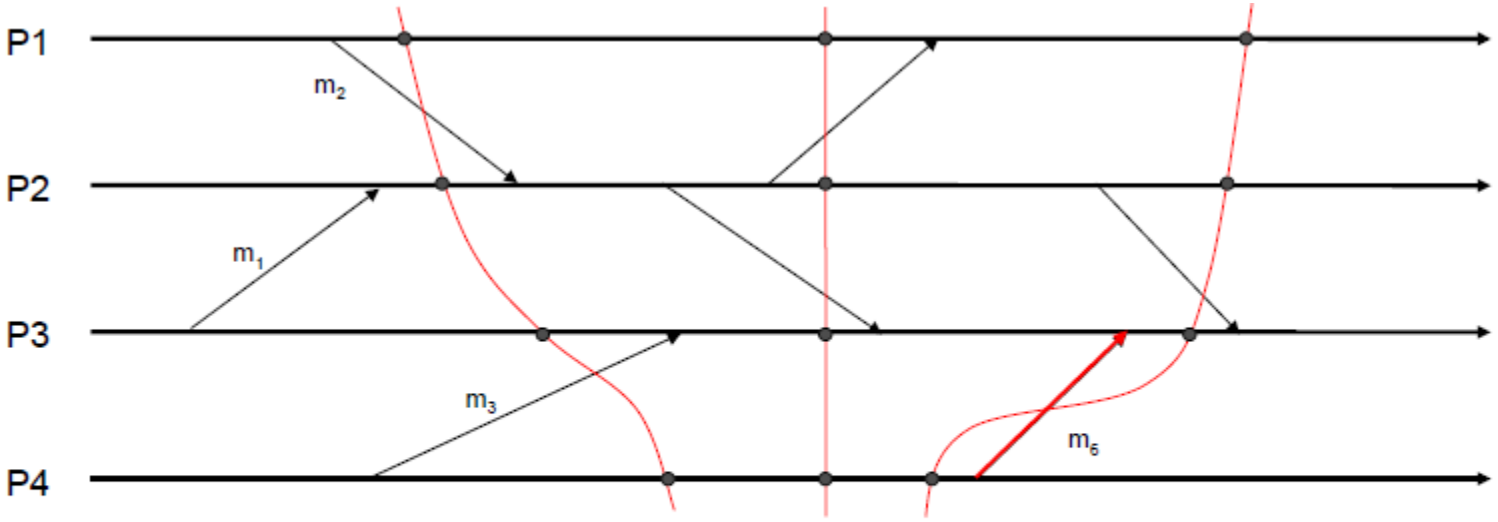
$E_{C1}(SR) = E(P_1) = E_{1,12} \cup E(P_2) = E_{2,9} \cup E(P_3) = E_{3,21} \cup E(P_4) = E_{4,17} \cup E(C_{12}) = \{m_2\} \cup E(C_{43}) = \{m_3\}$

Coupures cohérentes

- ***Cohérence = respect de la causalité (un message ne peut pas venir du futur)***



Coupures cohérentes



C1 et C2 sont cohérentes

C3 **n'est pas** cohérente :
réception(m_6) $\in$ Passé(C3) et
émission(m_6) $\notin$ Passé(C3)

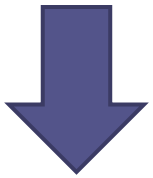
Temps Physique vs Temps Logique

- Temps logique : Ordre des évènements
- Temps physique : Quant s'est déroulé un évènement

Temps global logique

Horloge logique

Temporisateur associé à chaque processeur permettant de générer un compteur de temps local (des impulsions d ’horloge)



Problème de dérive d ’horloge

Dans un système distribué au fil du temps, la synchronisation des horloges se dégrade

Introduction des erreurs

Horloge Logique

- Objectif : ordonner des événements dans un système distribué
- Chaque processeur maintient une horloge locale
- Pas de point central pour fournir le temps
- Ne donne pas le temps d'un événement, elle s'intéresse à préciser les événements qui se sont déroulés avant ou après

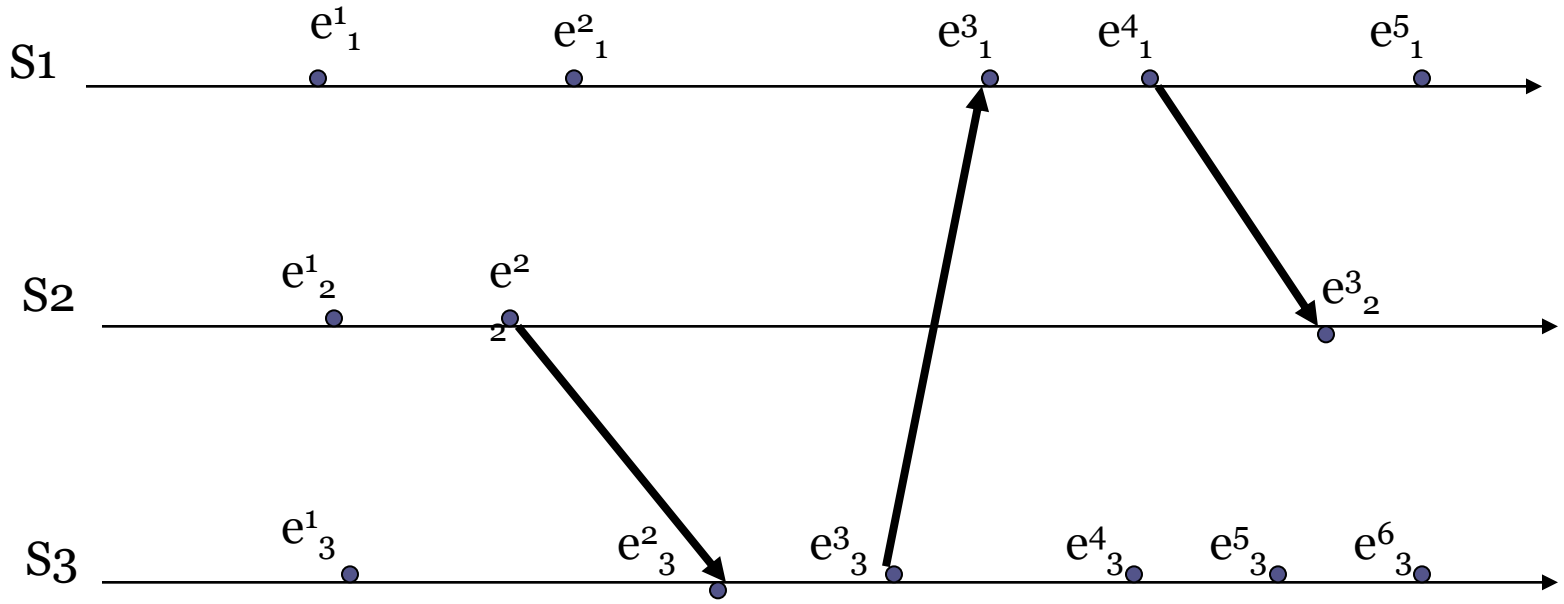
Dans un système réparti composé de n sites connectés par des canaux :

- **Les événements sur un même site sont totalement ordonnés (émission, réception, interne)**
- **Pour chaque message, l'événement émission précède toujours sa réception.**

La fermeture transitive de ces relations de précédence définit une relation de dépendance causale notée $\rightarrow$ sur l'ensemble des événements produits par une exécution répartie.

Cette relation est un ordre partiel.

Exemple : diagramme d'une exécution répartie.



$e^1_1 \rightarrow e^2_1$

$e^1_2 \rightarrow e^3_1$

Deux événements x et y tels que ni $x \rightarrow y$ ni $y \rightarrow x$ sont dits Indépendants ou concurrents. On notera $x \mid\mid y$.

Problème :

Trouver des mécanismes qui permettent d'associer des dates aux événements concernés tels que si :

a $\rightarrow$ ***b*** **alors la date associée à *b* doit être relativement à un temps logique global, après la date associée à *a*.**
(propriété de monotonie)

Mécanismes d'estampillage

- **Deux mécanismes d'estampillage essentiels :**
  - Temps linéaire (Lamport-78)
    - Temps logique représenté par un entier
  - Temps vectoriel (Mattern-89, Fidge-89)
    - Temps représenté par un vecteur de dimension n

- **Modèle**

Ces deux mécanismes obéissent au même modèle :

- **Structure de données pour représenter le temps**

A chaque site sont associées des variables qui lui permettent :

- **Mesurer sa propre progression, ce qui est assuré par son horloge logique locale (mise à jour par la règle1)**

- **Avoir une bonne représentation du temps logique global; c'est une vue locale du temps global (mise à jour par la règle2).**

➢ Protocole

Assure que l'horloge logique locale et la vue locale du temps Global de chaque site sont gérées de manière cohérente relativement à la relation $\rightarrow$.

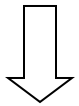
- Règle 1:

Avant de produire un événement (émission, réception, interne), un site doit incrémenter son horloge locale (parce qu'il progresse).

- Règle 2:

Pour que la date (estampille) correspondant à la réception d'un message soit après la date correspondant à l'événement émission du même message, $\Rightarrow$ chaque message m

transporte la valeur du temps logique global vu par l'émetteur Au moment de l'émission.



Ceci permet au récepteur de mettre à jour sa vue du temps global. Puis il exécute la règle 1.

Temps linéaire

❑ **Protocole**

A chaque site est associée une variable entière h_i ayant des valeurs croissantes.

Règle 1: **Avant de produire un événement (émission, réception, interne) Faire :**

$$h_i := h_i + d \ (d > 0)$$

% Chaque fois que cette règle est exécutée, d peut avoir une valeur
% différente.

Règle 2: **Lorsqu'un site S_i reçoit un message *(m,h)* Faire :**

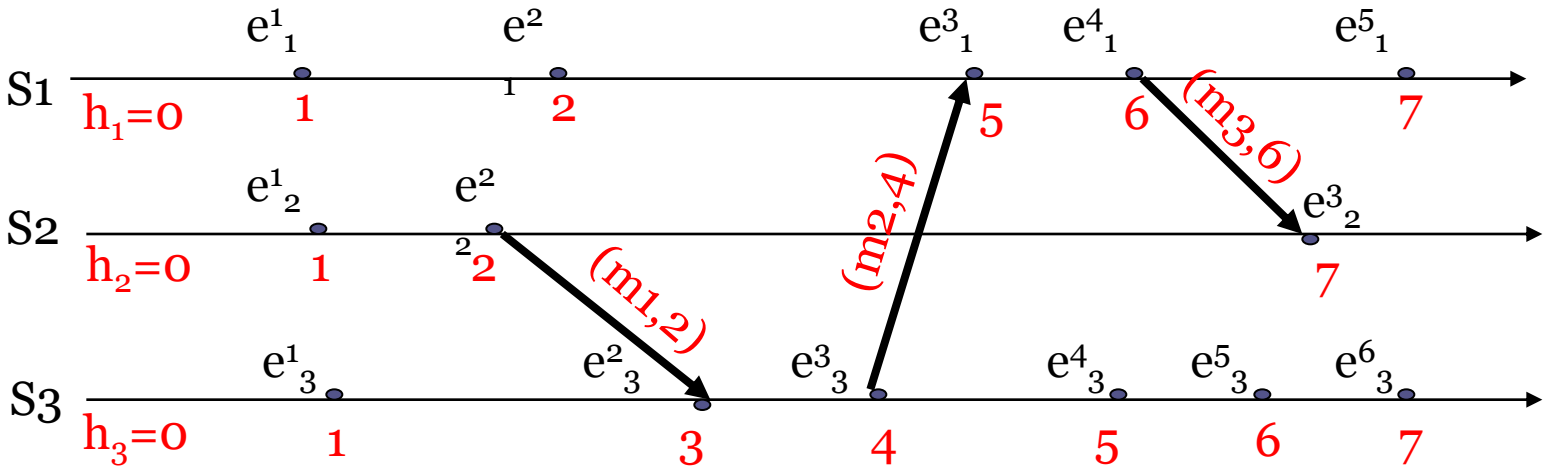
$h_i := max(h_i, h)$
Puis exécuter la règle 1.

❖ Propriétés

Il est possible d'utiliser ce mécanisme d'estampillage pour construire un ordre total noté $\rightarrow^{T}$.

- **Estampille (e) ≡ date d'occurrence + identité du site qui a produit e.**
- **Si on a deux événements x et y estampillés respectivement par *(h,i)* et *(k,j),* on définit :**

$$x \rightarrow^{T} y \Leftrightarrow (h < k \text{ ou } (h = k \text{ et } i < j))$$



- Les estampilles ne sont pas “denses”

  si H(e) < H(e'), on ne peut pas savoir s’il existe e"

  tel que e précède e" et/ ou e"‘ précède e ’.

  On ne peut pas prendre immédiatement une décision

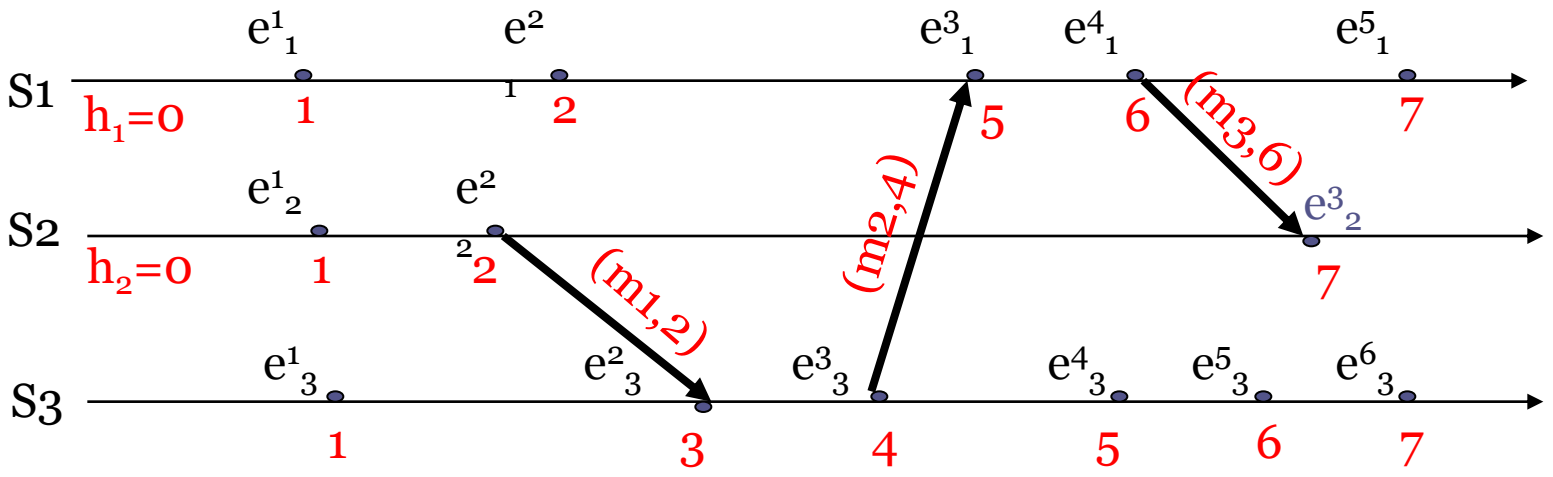
- Pour certaines applications (mise au point, mesure du parallélisme) on a besoin de caractériser l’indépendance causale.

  Propriété recherchée pour une horloge V:

  e $\rightarrow$ e' est équivalent à V(e) < V(e')

Horloges vectorielles

- **Une méthode de datation causale : les historiques**
  - Rappel : passé d'un événement e
  - hist(e) = l' ensemble des e' tels que e' → e



Hist(e32) = {e11,e21,e31,e41,e12,e22,e32,e13,e23,e33}

- Idée : utiliser le passé d'un événement e pour la datation
- Le passé permet de déterminer la dépendance causale :
  - hist(e) : ensemble des événements e' tels que e' $\rightarrow$ e

    e $\rightarrow$ e' alors e $\in$ hist(e')

    e || e' alors (e $\notin$ hist(e')) et (e' $\notin$ hist(e))
- Inconvénient : taille de hist(e)
- Remède : pour définir hist(e), un événement par site suffit.

  C'est l'idée des horloges vectorielles.

- Le temps est ici représenté par un vecteur de dimension n.

❑ Protocole

Chaque site S_i est doté d'un vecteur $vt_i[1..n]$ où :

- $vt_i[i]$; **décrit l'évolution du temps logique du site S_i : l'horloge logique locale de S_i. Elle prend des valeurs croissantes générées localement.**
- $vt_i[j]$: **Représente la connaissance qu'a le site S_i sur l'évolution du temps sur le site S_j. C'est une image locale de la valeur $Vt_j[j]$.**
- **Le vecteur vt_i constitue une vue locale du temps logique global utilisé pour estampiller les événements.**

Règle 1:

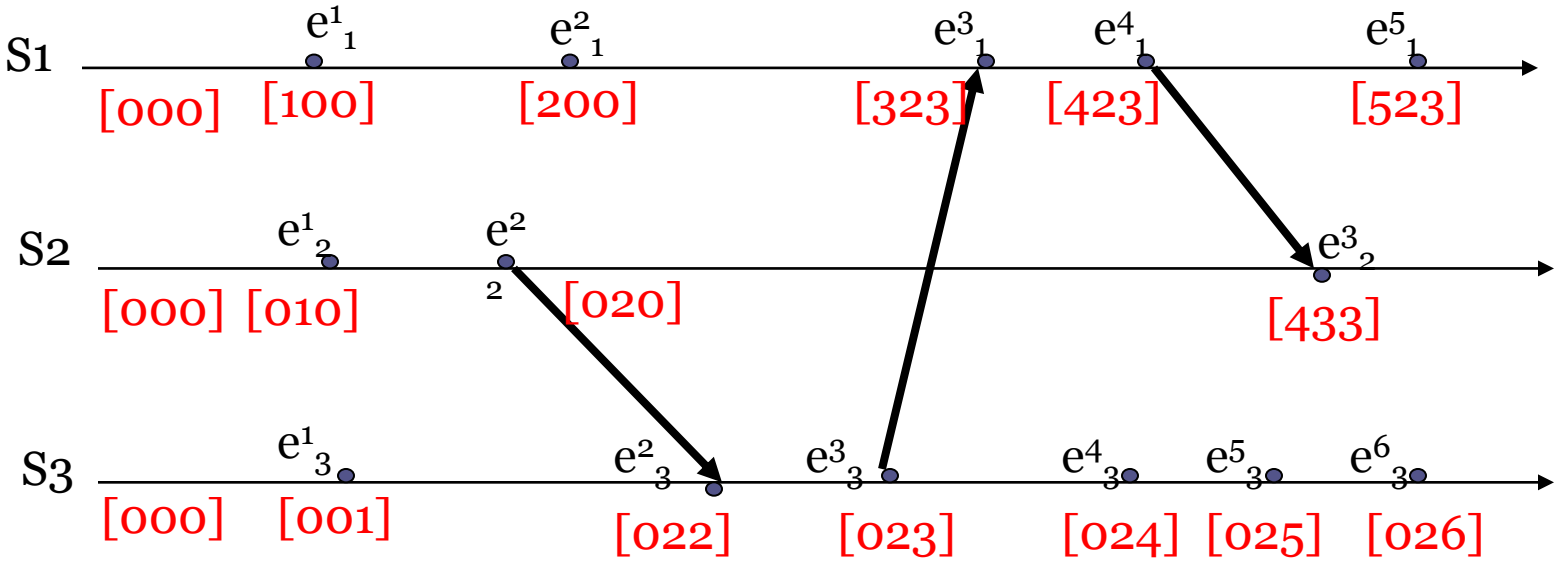
Avant de produire un événement
Faire :

$$Vt_i[i] := vt_i[i] + d \quad (d > 0)$$

Règle 2:

A la réception de *(m,vh)*
Faire :

Pour k de 1 à n faire vti[k] := max(vti[k],vh[k])
Puis exécuter la règle 1.



- Propriétés

  - $vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$
  - $vh < vk \Leftrightarrow vh \leq vk$ et $\exists x : vh[x] < vk[x]$
  - $vh \,||\, vk \Leftrightarrow \neg(vh < vk)$ et $\neg(vk < vh)$

  - Si on a deux événements x et y estampillés respectivement vh et vk alors :

    $x \rightarrow y \Leftrightarrow vh < vk$
    $x \,||\, y \Leftrightarrow vh \,||\, vk$

Horloges physiques

- Algorithme de Lamport propose une solution pour ordonner les évènements de façon non ambiguë
- Mais horloges logiques associées aux évènements ne correspondent pas forcément à la date réelle à laquelle ils se produisent
- Dans certains systèmes (trafic aérien, commerce, transactions) , le temps réel est important : un certain nombre d'algorithmes permettant de gérer ce problème existent

Propriétés des horloges physiques

Déviation, ou écart (skew) au temps t : différence entre deux horloges (par ex. sur deux machines d'un réseau)

Précision : déviation par rapport au temps réel (universel)

Dérive (drift) : divergence (déviation croissante) entre horloges due à des fréquences différentes (notamment divergence entre une horloge et l ' horloge“ idéale ” donnant l ' heure exacte)

Taux de dérive (drift rate) : mesure de la dérive entre deux horloges (en déviation par seconde).

Qu' est-ce que le “temps réel” ?

Une convention internationale définit le temps réel

- UTC : Coordinated Universal Time
- Utilise une horloge atomique (taux de dérive 10^{-13} s/s) ; ajustement périodique par rapport au temps astronomique
- L'heure universelle est diffusée par radio et satellites (GPS)
- Un ordinateur muni d'un récepteur peut donc synchroniser périodiquement son horloge interne, et fonctionner comme serveur de temps sur un réseau local

Synchronisation d'horloges physiques

Synchronisation externe

À partir d'une source de temps S (serveur), un ensemble d'horloges Hi est synchronisé de telle sorte que dans tout intervalle Δt de temps réel :

$|S(t) - H_i(t)| < D$ (écart borné) pour tout $t \in \Delta t$

Synchronisation interne

Les horloges Hi sont mutuellement synchronisées, de sorte que dans tout intervalle Δt de temps réel :

$| H_i(t)| - H_j(t)| < D$ pour tout couple i, j et pour tout $t \in \Delta t$

Synchronisation interne n' implique pas synchronisation externe : l'ensemble des horloges peut collectivement dériver par rapport à un serveur

Synchronisation externe : algorithme de Cristian (1)

Problème : synchroniser un ensemble d'horloges à partir d'un serveur de temps (lui-même alimenté par le temps universel UTC)

Difficulté : tenir compte du temps de transfert, a priori inconnu

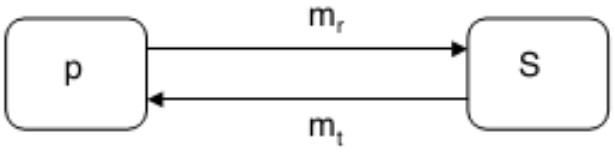
Solution : mesurer le temps d'aller-retour

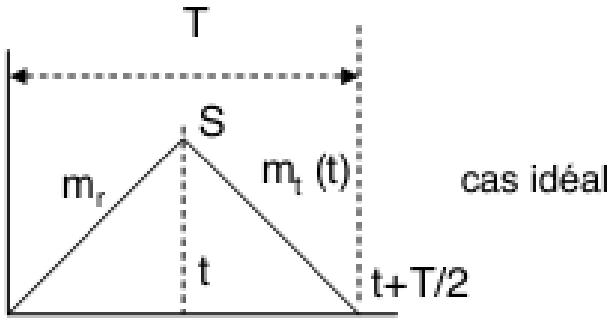
Synchronisation externe : algorithme de Cristian (1)

p demande l'heure à S (message m_r) et reçoit t (message m_t)

Par ailleurs p a mesuré le temps T d' aller-retour de p à S

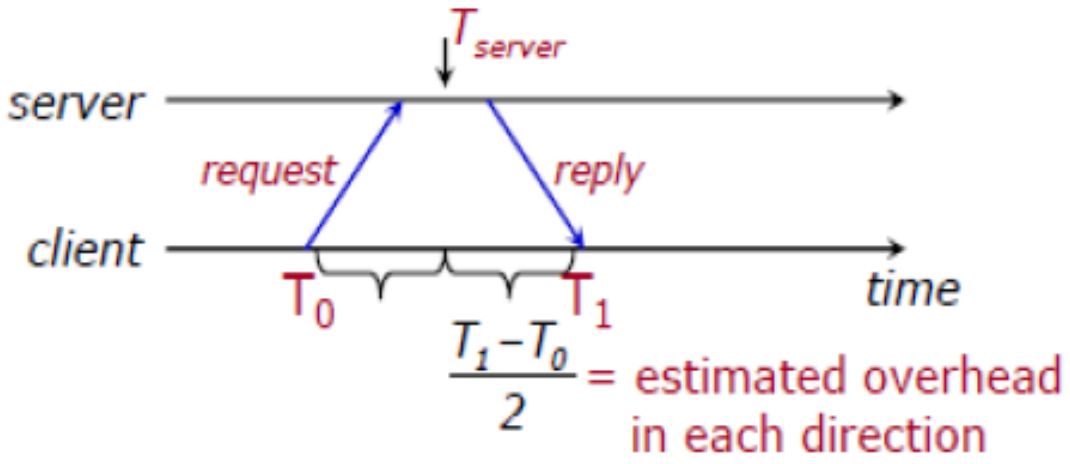
p règle son horloge au temps t + T/2





Synchronisation externe : algorithme de Cristian (2)

Chaque machine interroge Périodiquement le serveur de temps pour avoir l ’heure courante afin de pouvoir corriger d ’une manière adéquate son horloge



Client sets time to: $T_{new} = T_{server} + \frac{T_1 - T_0}{2}$

Synchronisation interne : algorithme de Berkeley

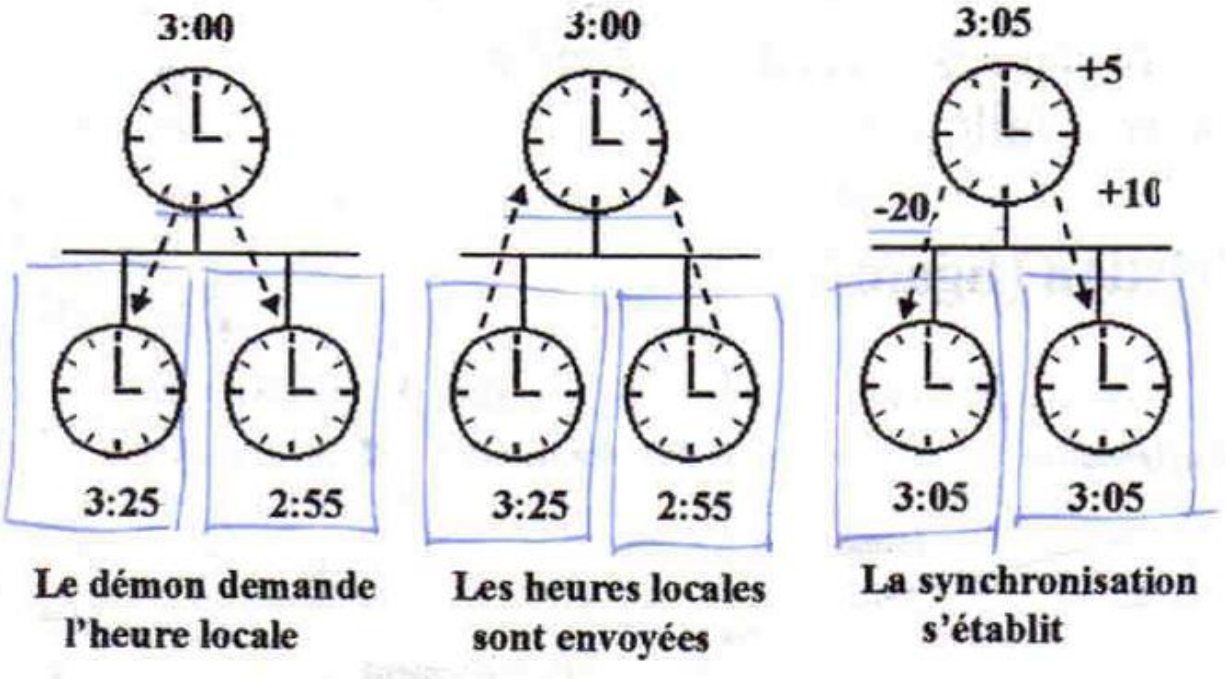
Problème : synchronisation interne d' un groupe de machines

- Un coordinateur demande aux participants (par diffusion) la valeur courante de leur horloge
- Le coordinateur estime l'heure locale de chaque participant au moyen du temps d'aller-retour (cf Cristian), et calcule la moyenne t_m de ces heures
- Le coordinateur demande aux participants de régler leur horloge sur cette heure moyenne (un participant en avance ne retarde pas son horloge, mais la ralentit pour atteindre progressivement la valeur fixée)

La précision du protocole dépend de l'estimation du temps d'aller-retour.
L'idéal est qu'il soit le même pour tous ; en pratique, on fixe un délai maximal et on ne tient pas compte des sites qui dépassent ce délai

Panne du coordinateur : on élit un autre coordinateur (cf plus loin : élection)
Le coordinateur élimine les valeurs trop déviantes

Algorithme de Berkeley

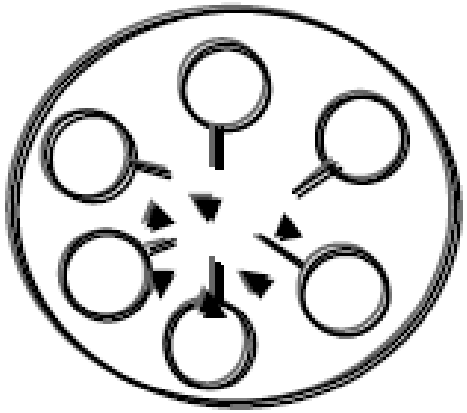


Le serveur de référence appelle les autres machines (inverse de Christian), il utilise un daemon.

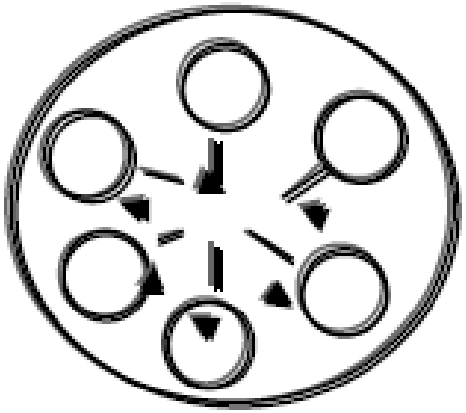
- Le daemon reçoit les réponses.
- Le daemon calcule un temps moyen, serveur compris, il ajuste le serveur si nécessaire et demande aux autres machines de s'ajuster en donnant les directives (+t1, t2, etc.).

Algorithmes basés sur la moyenne

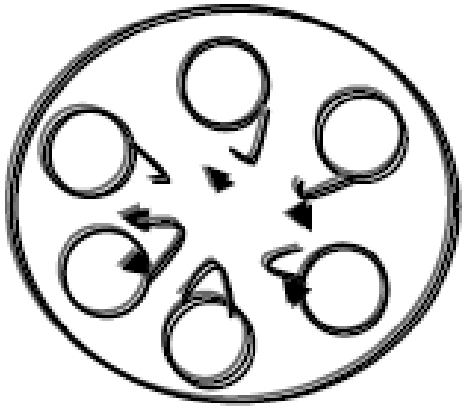
Christian et Berkeley sont des algorithmes centralisés. Il existe néanmoins des algorithmes répartis basés sur le fait que toutes les machines « broadcast » leurs temps régulièrement et chaque machine reçoit le temps de toutes les autres (ou une partie des autres), fait sa moyenne et calcule son temps.



Chaque machine diffuse son heure au début d ’une période de temps



Chaque machine démarre un temporisateur et collecte tous les messages arrivés durant un intervalle



Chaque machine exécute un même algorithme à partir des n données reçues pour l ’obtention d ’une heure moyenne

Exclusion-mutuelle

Rappel

- Une ressource partagée ou une section critique n'est accédée que par un processus à la fois
- Un processus est dans 3 états possibles, par rapport à l'accès à la ressource
  - ✓**Demandeur**
  - ✓**Dedans**
  - ✓**Dehors**
- Changement d'état par un processus
  - ✓De *dehors à demandeur*
  - ✓De *dedans à*
- Le passage de l'état *demandeur à l'état dedans est géré par le* système et/ou l'algorithme de gestion d'accès à la ressource (Couche middleware)

Rappel exclusion mutuelle

L'accès en exclusion mutuelle doit respecter deux propriétés

Sûreté (safety) : au plus un processus est à la fois dans la section critique (dans l'état *dedans)*

Vivacité (liveness) : tout processus demandant à entrer dans la section critique (à passer dans l'état *dedans) y entre en un* temps fini

Exclusion mutuelle distribuée

Plusieurs grandes familles de méthodes

Contrôle par un serveur
qui centralise les demandes d'accès à la ressource partagée

Contrôle par jeton

- un jeton unique circule sur l'ensemble des sites et donne le droit à son possesseur d'entrer en section critique. L'unicité du jeton assure la sûreté.

Contrôle par permission
le site demandeur doit recevoir l'accord d'un ensemble d'autres sites pour accéder à la section critique.

Contrôle par serveur

Principe général

- Un serveur centralise et gère l'accès à la ressource

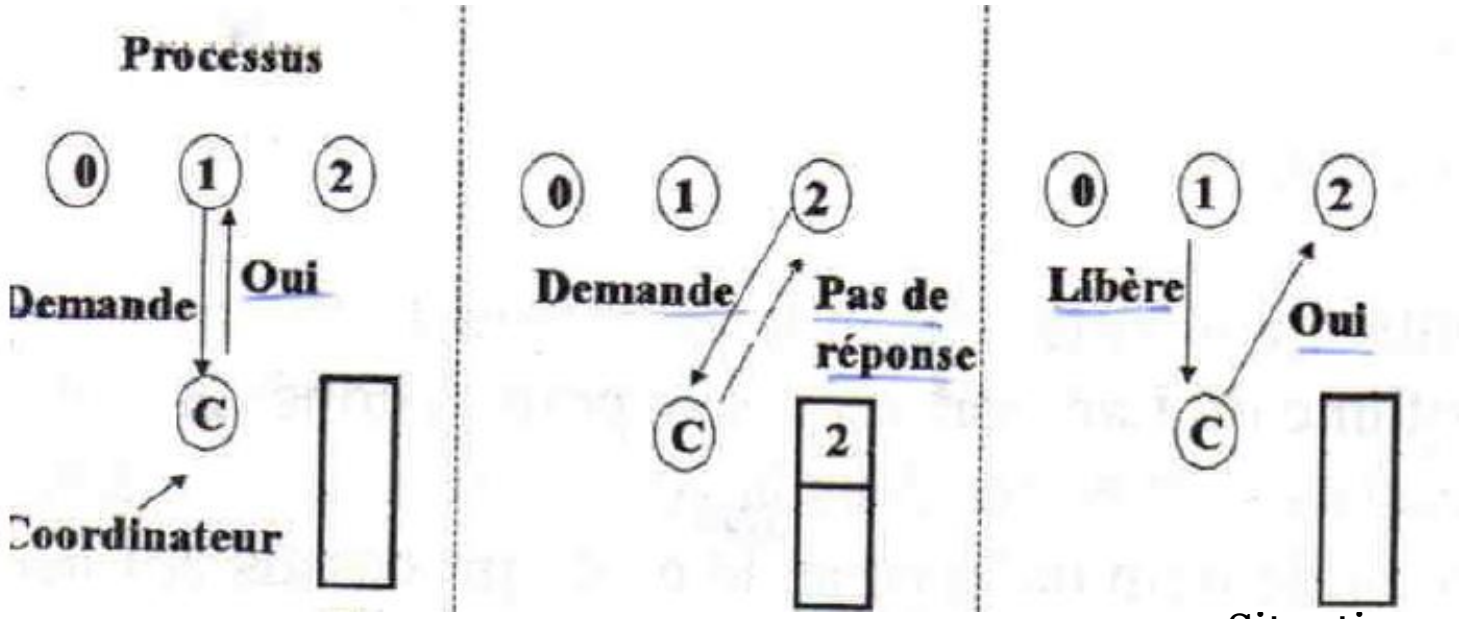
Algorithme

- Un processus voulant accéder à la ressource (quand il passe dans l'état *demandeur) envoie une requête au serveur*
- Quand le serveur lui envoie l'autorisation, il accède à la ressource (passe dans l'état *dedans)*

  ✓Il informe le serveur quand il relâche la ressource (passe dans l'état *dehors)*
- Le serveur reçoit les demandes d'accès et envoie les autorisations d'accès aux processus demandeurs

  ✓Avec par exemple une gestion FIFO : premier processus demandeur, premier autorisé à accéder à la ressource

Un algorithme centralisé implanté en réparti



Situation 1 Situation 2 Situation 3

- **Peu équitable (dépend de la vitesse de communication avec le coordinateur)**
- **Peu robuste**
- **Mais mise en oeuvre simple !**

Contrôle par permission

Méthodes par permission

- Un processus doit avoir l'autorisation des autres processus pour accéder à la ressource

Deux modes

- Permission individuelle : un processus peut donner sa permission à plusieurs autres à la fois
- Permission par arbitre : un processus ne donne sa permission qu'à un seul processus à la fois

  ✓Les sous-ensembles sont conçus alors tel qu'au moins un processus soit commun à 2 sous-ensembles : il joue le rôle d'arbitre

Permission individuelle

- Algorithme de [Ricart & Agrawala, 81]
- Chaque processus demande l'autorisation à tous les autres (sauf lui par principe)
- Se base sur une horloge logique (Lamport) pour garantir le bon fonctionnement de l'algorithme

Un algorithme distribué (Ricart & Agrawala-1981): Principe

· Quand un processus veut entrer dans la zone critique, il construit un message contenant le nom de la zone, son n° du processus et l'heure courante.
· Il envoie le message à tous les processus, lui inclus.
· Le message sera accusé (ACK) pour maintenir la fiabilité du mécanisme. La communication de groupe peut être utilisée.
· Les processus récepteurs répondront en fonction de leur état, lié à la zone en question, il y a 3 cas possibles :
1. Le récepteur n'est pas dans la zone et ne veut pas y entrer, il répond OK au demandeur.
2. Le récepteur est déjà dans la zone, il ne répond pas. Il charge la demande dans sa file d'attente.
3. Le récepteur veut aussi entrer dans la zone. Il a déjà envoyé les messages.

Un algorithme distribué (Ricart & Agrawala-1981): Principe

Il compare l'heure du message qu'il vient de recevoir avec l'heure stockée dans le message qu'il a déjà envoyé. Si l'heure du message reçu est inférieure à la sienne, il répond OK, sinon (si son heure est inférieure) il place la demande dans sa file et ne répond pas.

· Quand sa demande est partie, le processus attend son acceptation par tous les autres processus, ensuite il entre dans la zone critique. Quand il en sort, il répond OK aux autres processus qui sont dans sa file d'attente et les enlève de sa file d'attente.

Un algorithme distribué (Ricart & Agrawala-1981)

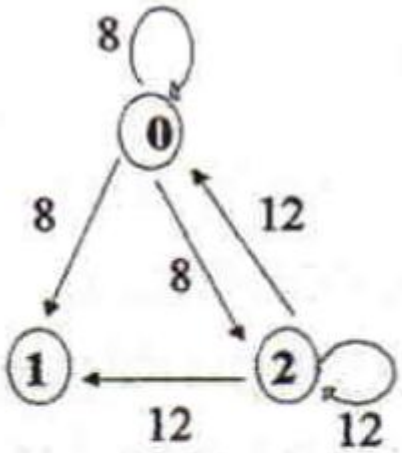
•Cet algorithme nécessite un ordre total de tous les évènements dans le système.

•Dans une série d'évènements il est essentiel que l'ordre dans lequel ils ont été demandés soit non ambigu.

•L'algorithme de Lamport, vu précédemment, sera utilisé pour préserver cet ordre et fournir les *timestamp nécessaires à l'exclusion mutuelle distribuée*.

Un algorithme distribué (Ricart & Agrawala-1981)

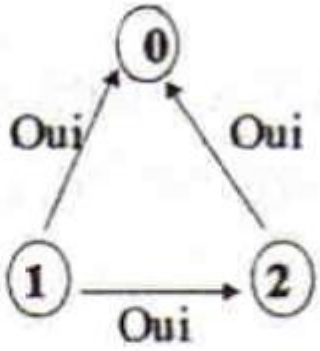
Phase 1



Les processus 0 et 2 veulent entrer dans la même zone critique. Les messages sont envoyés.

a)

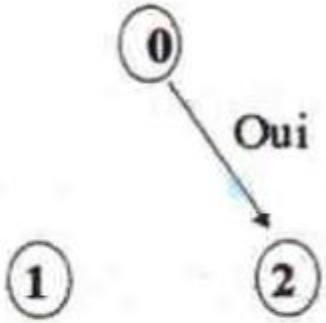
Phase 2



Le processus 0 reçoit les approbations il entre dans la zone critique

b)

Phase 3



Le processus 0 a terminé, il envoie l'approbation au processus 2 qui entre dans la zone critique

c)

Inconvénients de cet algorithme :

•Couteux en nombre de messages par SC: 2(N-1)
•Robuste (mais, savoir exactement combien de processus sont vivants (N doit constamment être exacte)

•Il n'y a plus un seul point de panne, mais n. En particulier à cause des non réponses qui peuvent être interprétées comme un refus alors qu'un processeur peut-être en panne et non apte à répondre. Les autres processus vont attendre pour rien.

•Amélioration possible : avoir une majorité (à déterminer) de OK pour rentrer en zone critique. Bien sûr un processus ne pourra pas donner la même permission à un deuxième processus tant que le premier n'a pas libérer la ressource.

Algorithme de **Ricart & Agrawala-1981**

.Respect des propriétés

- Sûreté : vérifiée
- Vivacité : assurée grâce aux datations et aux priorités associées

Permission individuelle

- amélioration de l'algorithme de [Ricart & Agrawala, 81]

- **[Carvalho & Roucairol, 83]**
  ✓Si *Pi veut accéder plusieurs fois de rang à la ressource partagée* et si *Pj entre 2 accès (ou demandes d'accès) de Pi n'a pas* demandé à accéder à la ressource
    ❖Pas la peine de demander l'autorisation à *Pj car on sait alors qu'il donnera* par principe son autorisation à *Pi*
    ❖Limite alors le nombre de messages échangés

Carvalho & Roucairol, 83

Principe de l'algorithme

Soit le cas où P_i demande l'accès à la Section Critique plusieurs fois de suite (état ***demandeur*** plusieurs fois de suite) alors que P_j n'est pas intéressé par celle-ci (état ***dehors***)

- Avec l'algorithme de Ricart et Agrawala, P_i demande la permission de P_j à chaque nouvelle demande d'accès à la Section Critique
- Avec l'algorithme de Carvalho et Roucairol, puisque P_j a donné sa permission à P_i, ce dernier la considère comme acquise jusqu'à ce que P_j demande sa permission à P_i : P_i ne demande qu'une fois la permission à P_j

Permission individuelle

amélioration de l'algorithme de [Ricart & Agrawala, 81]

- **[Chandy & Misra, 84],** améliorations tel que
  - ✓Les processus ne voulant pas accéder à la ressource et qui ont déjà donné leur permission ne reçoivent pas de demande de permission
  - ✓Horloges avec des datations bornées (modulo *m)*
  - ✓Pas d'identification des processus

Permission par arbitre

- Un processus ne donne qu'une permission à la fois

Il redonnera sa permission à un autre processus quand le processus a qui il avait donné précédemment la permission lui a indiqué qu'il a fini d'accéder à la ressource

- La sûreté est assurée car

✓ Les sous-ensemble de processus à qui un processus demande la permission sont construits tel qu'ils y ait toujours au moins un processus commun à 2 sous-ensemble

✓ Un processus commun à 2 sous-ensembles est alors arbitre

❖ Comme il ne peut donner sa permission qu'à un seul processus, les processus de 2 sous-ensembles ne peuvent pas tous donner simultanément la permission à 2 processus différents

❖ C'est donc ce processus commun qui détermine à qui donner la ressource

Permission par arbitre

- **Algorithme de [Maekawa, 85]**
  - Chaque processus *Pi* possède un sous-ensemble *Ri* d'identificateurs de processus à qui *Pi* demandera l'autorisation d'accéder à la ressource
    - $\forall i,j \in [\ 1..N\ ] : Ri \cap Rj \neq \varnothing$
      - Deux sous-ensembles de 2 processus différents ont obligatoirement au moins un élément en commun (le ou les arbitres)
      - Cela rend donc inutile le besoin de demander la permission à tous les processus, d'où les sous-ensembles Ri ne contenant pas tous les processus
    - $\forall i : |\ Ri\ | = K$
      - Pour une raison d'équité, les sous-ensembles ont la même taille pour tous les processus
    - $\forall i$: i est contenu dans D sous-ensembles
      - Chaque processus joue autant de fois le rôle d'arbitre qu'un autre processus

Permission par arbitre

Algorithme de [Maekawa, 85] (suite)

Solution optimale en nombre de permissions à demander et de messages échangés

Fonctionnement de l'algorithme

Chaque processus possède localement

- Une variable *vote permettant de savoir si le processus a déjà* voté (a déjà donné sa permission à un processus)
- Une file *file d'identificateurs de processus qui ont demandé la* permission mais à qui on ne peut la donner de suit
- Un compteur *réponses du nombre de permissions reçues*

Initialisation

- État non demandeur, vote = faux et file = ∅, réponses = 0

Permission par arbitre

- **Algorithme de [Maekawa, 85] (suite)**
  - Quand processus *Pi* veut accéder à la ressource
    - réponses = 0
    - Envoie une demande de permission à tous les processus de *Ri*
    - Quand *réponses* = | *Ri* |, *Pi* a reçu une permission de tous, il accède alors à la ressource
    - Après l'accès à la ressource, envoie un message à tous les processus de *Ri* pour les informer que la ressource est libre
  - Quand processus *Pi* reçoit une demande de permission de la part du processus *Pj*
    - Si *Pi* a déjà voté (vote = vrai) ou accède actuellement à la ressource : place l'identificateur de *Pj* en queue de *file*
    - Sinon : envoie sa permission à *Pj* et mémorise qu'il a voté
      - vote = vrai

Permission par arbitre

- **Algorithme de [Maekawa, 85] (suite)**

  - Quand *Pi* reçoit de la part du processus *Pj* un message lui indiquant que *Pj* a libéré la ressource
    - Si *file* est vide, alors vote = faux
      - *Pi* a déjà autorisé tous les processus en attente d'une permission de sa part
    - Si *file* est non vide
      - Retire le premier identificateur (disons *k*) de la file et envoie à *Pk* une permission d'accès à la ressource
      - *vote* reste à vrai

Permission par arbitre

- **Algorithme de [Maekawa, 85] probleme!**

- La vivacité n'est pas assurée par cet algorithme car des cas d'interblocage sont possibles
- Pour éviter ces interblocages, améliorations de l'algorithme en définissant des priorités entre les processus

✓ En datant les demandes d'accès avec une horloge logique

✓ En définissant un graphe de priorités des processus

Méthode par jeton

Principe général

- Un jeton unique circule entre tous les processus
- Le processus qui a le jeton est le seul qui peut accéder à la section critique

Respect des propriétés

- Sûreté : grâce au jeton unique
- Vivacité : l'algorithme doit assurer que le jeton circule bien entre tous les processus voulant accéder à la ressource

Plusieurs versions

- Anneau sur lequel circule le jeton en permanence
- Jeton affecté à la demande des processus

Méthode par jeton

Algorithme de [Le Lann, 77]

- Un jeton unique circule en permanence entre les processus via une topologie en anneau
- Quand un processus reçoit le jeton

✓ S'il est dans l'état *demandeur : il passe dans l'état dedans et* accède à la ressource

✓ S'il est dans l'état *dehors, il passe le jeton à son voisin*

- Quand le processus quitte l'état *dedans, il passe le jeton à* son voisin

Respect des propriétés

- **Sûreté** : via le jeton unique qui autorise l'accès à la ressource
- **Vivacité** : si un processus lâche le jeton (la ressource) en un temps fini et que tous les processus appartiennent à l'anneau

Méthode par jeton

Algorithme de [Le Lann, 77]

Inconvénients

- Nécessite des échanges de messages (pour faire circuler le jeton) même si aucun site ne veut accéder à la ressource
- Temps d'accès à la ressource peut être potentiellement relativement long
- Si le processus *i+1 a le jeton et que le processus i veut accéder à la* ressource et est le seul à vouloir y accéder, il faut quand même attendre que le jeton fasse tout le tour de l'anneau
- Perte du jeton
- Problème de la panne d'un pro cessu

Avantages

- Très simple à mettre en oeuvre
- Intéressant si nombreux processus demandeurs de la ressource
- Jeton arrivera rapidement à un processus demandeur
- Équitable en terme de nombre d'accès et de temps d'attente
- Aucun processus n'est privilégié

Méthode par jeton

Variante de la méthode du jeton

- Au lieu d'attendre le jeton, un processus diffuse à tous le fait qu'il veut obtenir le jeton
- Le processus qui a le jeton sait alors à qui il peut l'envoyer
- Évite les attentes et les circulations inutiles du jeton

Algorithme de [Ricart & Agrawala, 83]

- Soit *N processus avec un canal bi-directionnel entre* chaque processus
  - ✓ Canaux fiables mais pas forcément FIFO
- Localement, un processus *Pi possède un tableau nbreq*, de taille *N*
- Pour *Pi, nbreq [j] est le nombre de requêtes d'accès que le processus Pj a fait et que Pi connaît (par principe il les connaît toutes)*

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Le jeton est un tableau de taille *N*
    - *jeton [i]* est le nombre de fois où le processus *Pi* a accédé à la ressource
    - La case *i* de *jeton* n'est modifiée que par *Pi* quand celui-ci accède à la ressource
  - Initialisation
    - Pour tous les sites *Pi* : $\forall j \in [1 .. N] : nbreq[j] = 0$
    - Jeton : $\forall j \in [1 .. N] : jeton[j] = 0$
    - Un site donné possède le jeton au départ
  - Quand un site veut accéder à la ressource et n'a pas le jeton
    - Envoie un message de requête à tous les processus

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83] (suite)
  - Quand processus *Pj* reçoit un message de requête venant d
    - *Pj* modifie son *nbreq* localement : nbreq [i] = nbreq [i] + 1
      - *Pj* mémorise que *Pi* a demandé à avoir la ressource
    - Si *Pj* possède le jeton et est dans l'état *dehors*
      - *Pj* envoie le jeton à *Pi*
  - Quand processus récupère le jeton
    - Il accède à la ressource (passe dans l'état *dedans*)
  - Quand Pi libère la ressource (passe dans l'état *dehors*)
    - Met à jour le jeton : jeton [i] = jeton [i] + 1
    - Parcourt *nbreq* pour trouver un *j* tel que : nbreq [j] > jeton [j]
      - Une demande d'accès à la ressource de *Pj* n'a pas encore été satisfaite : *Pi* envoie le jeton à *Pj*
      - Si aucun processus n'attend le jeton : *Pi* le garde

Méthode par jeton

- Algorithme de [Ricart & Agrawala, 83], respect des propriétés
  - Sûreté : seul le processus ayant le jeton accède à la ressource
  - Vivacité : assurée si les processus distribuent équitablement le jeton aux autres processus
    - Méthode de choix du processus qui va récupérer le jeton lorsque l'on sort de l'état dedans
      - *Pi* parcourt *nbreq* à partir de l'indice i+1 jusqu'à *N* puis continue de 1 à i-1
      - Chaque processus teste les demandes d'accès des autres processus en commençant à un processus spécifique et différent de la liste
      - Évite que par exemple tous les processus avec un petit identificateur soient servis systématiquement en premier

Algorithmes d'élection

Election

Principe :

Choisir un et un seul leader parmi un ensemble de processus et le faire connaître de tous.

Hypothèses :

L'élection peut être déclenchée par un processus arbitraire ou éventuellement par plusieurs processus.

Leader : le processus qui a le plus grand numéro.

Propriétés :

Sûreté (safety) : un seul processus doit être élu
Vicacité(liveness) :un processus doit etre élu en un temps fini

Définition et utilisations

- *En pratique :*
  - le nombre des processus est connu a priori
  - les processus sont identifiés par un nombre (n° de processus, de site, etc)

Usage

- *Choix d'un processus maître (notamment après défaillance)*
  - pour la coordination d'un ensemble de processus
  - la régénération d'une information perdue (ex : jeton)
  - le contrôle de concurrence (séquencement)

Algorithme Brute de Force

Hypothèses :

Il y a aucune perte de messages
Il y a une borne connue sur le temps de communications.

Principe :

Les demandes d'élection sont diffusées par inondation.
Un processus répond a ceux de numéro inferieur au sien.
Un processus qui ne reçoit aucune réponse constate qu'il est élu.

Complexité :

$O(n^2)$ messages au pire des cas.

Algorithme Brute de Force

Déclanchement :

Quand un processus P s'aperçoit que le coordinateur ne répond plus à ses requêtes (time-out sur *TEMPO*), il lance l'algorithme d'élection

Lancement d'une élection par P :

Envoi d'un message `ELECTION` à tous les autres processus dont le numéro est plus grand que le sien

Réception d'un message `ELECTION` depuis P par un processus Q :

- Le processus Q envoie un message `ACK` à P lui signifiant qu'il est actif
- A son tour Q, lance une élection si ce n'est pas déjà fait

Algorithme Brute de Force

Sur le processus P :

- Si aucun processus ne lui répond avant *TEMPO*, P gagne l'élection et devient le coordinateur
- Si un processus de numéro plus élevé répond, c'est lui qui prend le pouvoir. Le rôle de P est terminé.

Annonce de l'élu

Le nouveau coordinateur envoie un message à tous les participants pour les informer de son rôle. L'application peut alors continuer à s'exécuter

Algorithme Brute de Force

Réveil d'un processus inactif

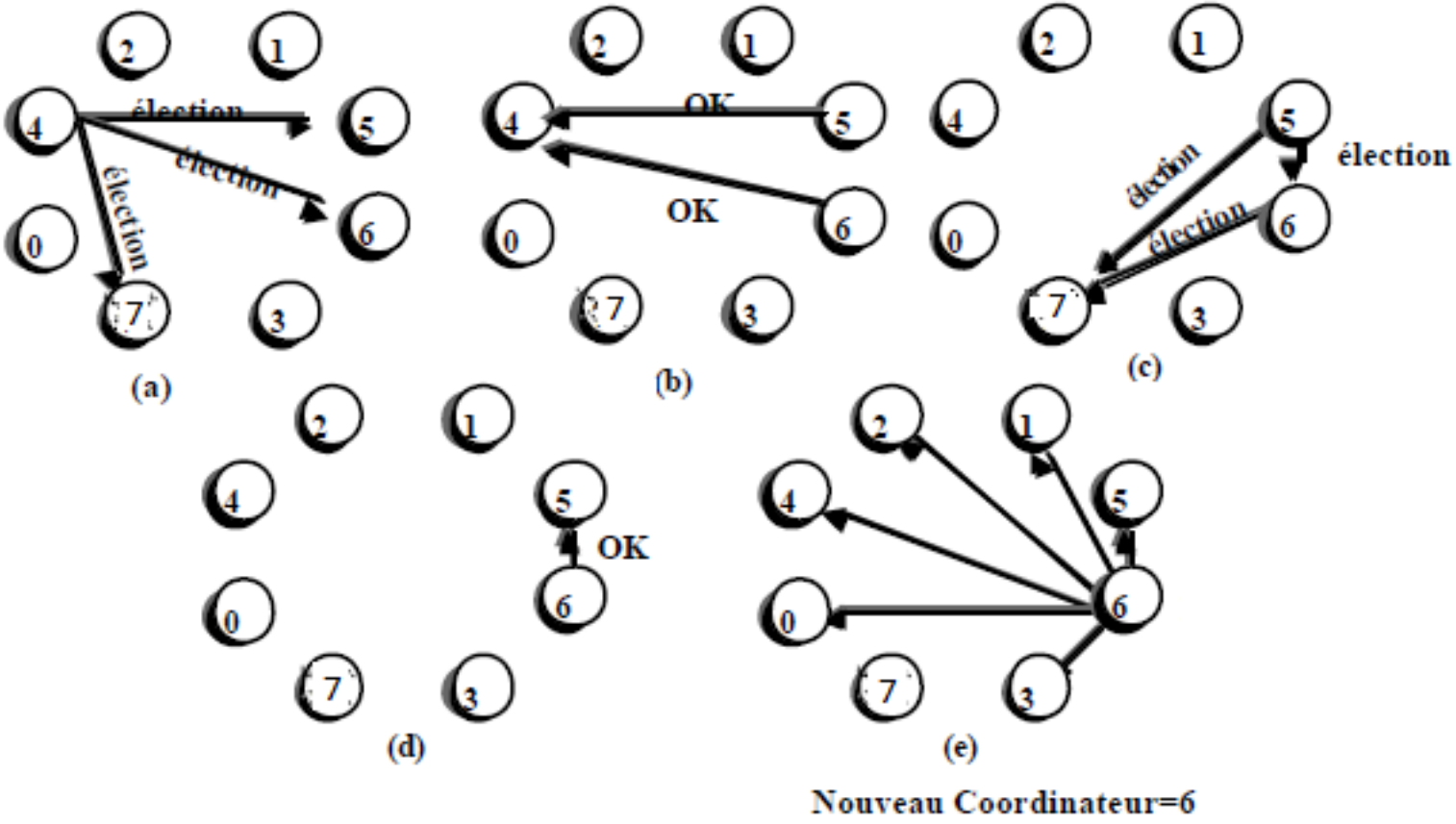
- Déclenche une élection
- S'il détient le plus grand numéro de processus en cours de fonctionnement, il gagne l'élection et devient le nouveau coordinateur

Algorithme Brute de Force

Déroulement de l'algorithme

- Application composée de 8 processus numérotés de 0 à 7
- Processus de numéro 7 tombe en panne, Processus de numéro 4 est le premier à détecter cette panne

Algorithme Brute de Force**(Garcia-Molina 1982)**



Algorithme sur un anneau

Principe :

- plusieurs sites peuvent démarrer le processus d‘élection
- Chaque site qui commence une élection envoie un jeton en précisant sa valeur
- Lors qu'un site reçoit un jeton de valeur inferieure, il le supprime

Algorithme utilisant un anneau [Chang-Roberts 79]

Principe

- *Extinction progressive des messages par filtrage*

Réalisation

- *Anneau virtuel unidirectionnel*
- *Programme du site i :*
- (chaque site i dispose d'un pointeur vers son successeur succ[i])
- plusieurs candidats simultanés possibles

Idée :

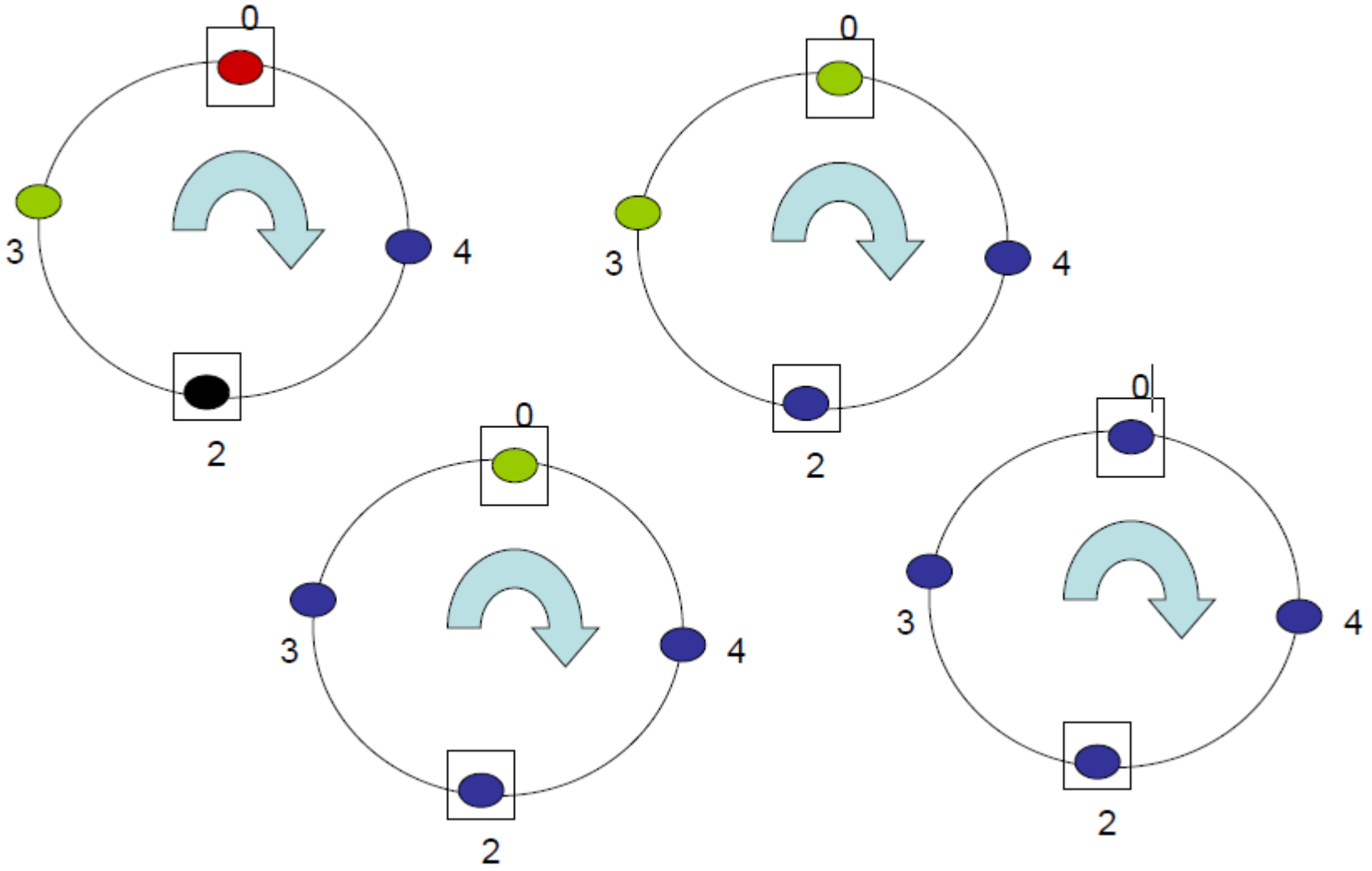
chaque candidat diffuse autour de l'anneau sa candidature; le processus ayant l'identifiant max gagne

Algorithme utilisant un anneau [Chang-Roberts 79]

Hyp othè et principe

- Supprime les jetons des processus qui ne sont pas élus : ceux qui ont un numéro de processus plus petit
- Un initiateur perd quand il reçoit un jeton qui porte un numéro supérieur
- Un processus devient leader lorsqu'il reçoit son jeton

Algorithme utilisant un anneau [Chang-Roberts 79]



Algorithme utilisant un anneau [Chang-Roberts 79]

Candidature (site pi):

candidat$_i$ = true

send (succ[i], ELECT, i)

R1: receive(pi, ELECT, j)

switch

j > i : send (succ[i], ELECT, j) *// j prioritaire*

j < i : if not candidat$_i$ *// j non prioritaire*

candidat$_i$ = true

send (succ[i], ELECT, i)

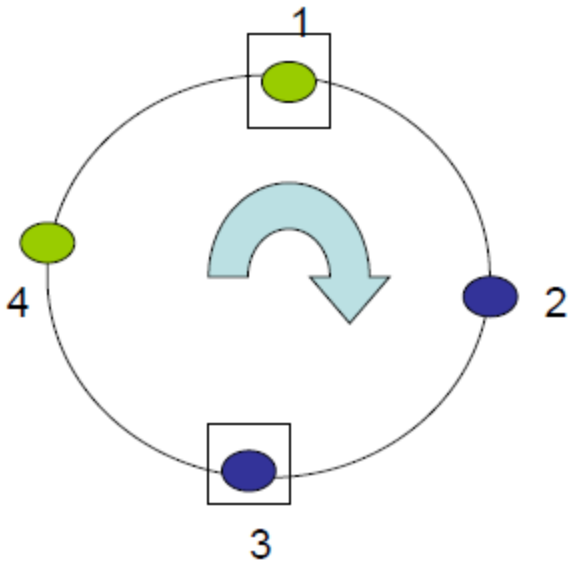
j = i : send (all, ELECTED, i) *// site i élu*

O(N log N) messages en moyenne, O(N^2) au pire

Compléxité *Chang-Roberts 79]*

Le meilleur cas : O(n)

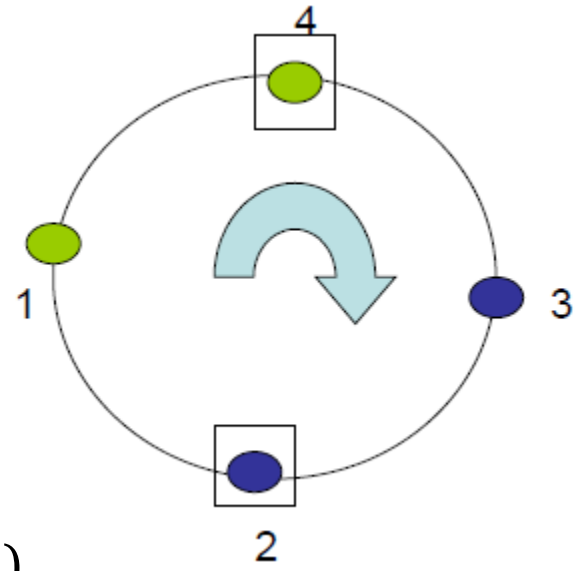
Les identifiants sont ordonnes dans l'ordre croissant autour de l'anneau



Compléxité *Chang-Roberts 79]*

Le pire cas : $O(n^2)$

· Les identifiants sont ordonnes dans l'ordre décroissant autour de l'anneau
· L'identifiant « i » visite i nœuds avant de décider son status



En moyenne : O(n log n)

Transactions atomiques

Introduction

Transaction: signifie la suite des instructions dont on veut exécuter l 'intégralité
=> transaction atomique
=> délimiter par des pseudo-instructions

Transaction répartie: si par exemple C1 et C2 sont sur des sites différents (et donc les actions sur ces objets aussi). La transaction doit se réaliser tout en étant atomique
=> coopération des sites

Transaction concurrente: si par exemple une autre transaction manipule C1 et/ou C2, en particulier en modification.
=> s 'assurer que leur résultat est comparable avec ce qu'il se serait passé si les 2 transactions s'exécutaient en séquence.

Mais par souci de performance, trouver des techniques pour qu'elles s 'exécutent le plus possible en parallèle : **CONTROLE de CONCURRENCE**

Modèle de transaction

TRANSACTION

On appelle transaction l'unité de traitement séquentiel, exécutée pour le compte d'un processus, appliquée à une mémoire stable cohérente, restituant une mémoire stable cohérente

PROPRIÉTÉS

-**Atomicité** : Une transaction est une unité logique indivisible de traitement qui est soit complètement exécutée soit complètement abandonnée

-**Sérialisabilité** : Les transactions concurrentes n ’interfèrent pas entre elles (en particulier on ne veut pas voir les états intermédiaires non encore validés par d’autres)

- **Permanence** : Si une transaction est validée, tous les changements par elle sur la mémoire stable sont devenus définitifs

Propriétés ACID: Atomicité / Cohérence / Isolation / Durabilité

Modèle de transaction

Propriétés ACID

Atomicité
tout ou rien

Consistance
cohérence sémantique

Isolation
pas de propagation de résultats non validés

Durabilité
persistance des effets validés

Modèle de transaction

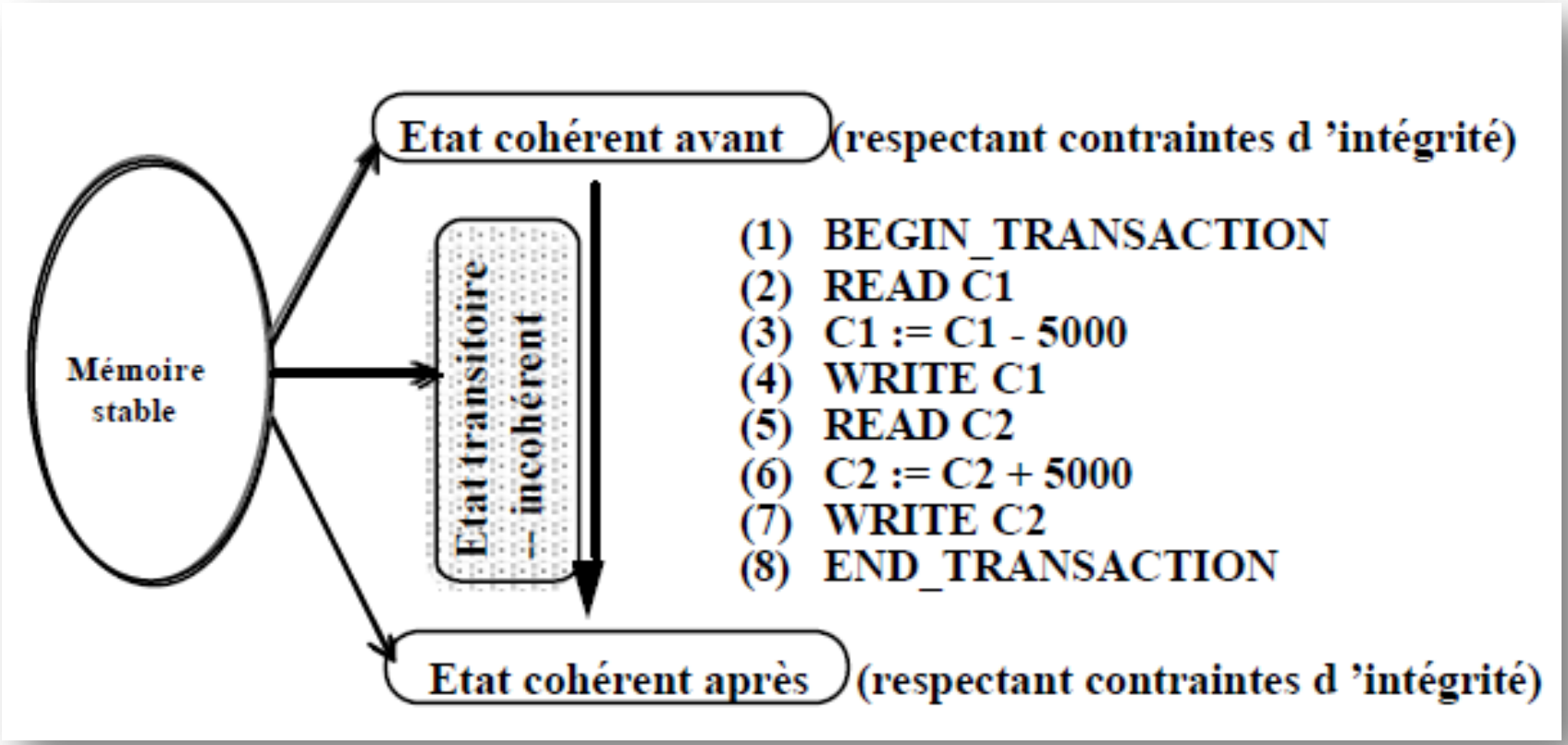


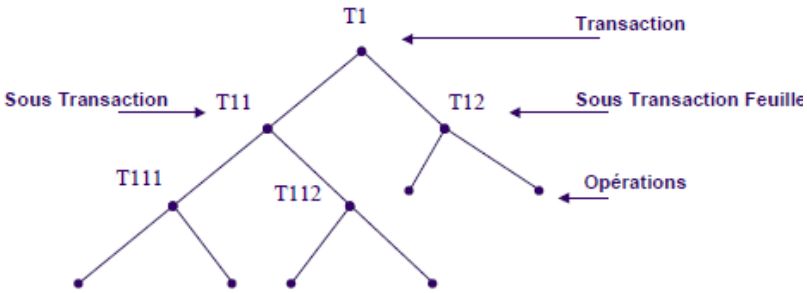
Illustration de la notion de « mise-à-jour sûre » par une transaction

Modèle de transaction: Transactions imbriquées

Une transaction peut contenir des sous-transactions appelées transactions imbriquées (peu courant; plutôt modèle plat -- flat)

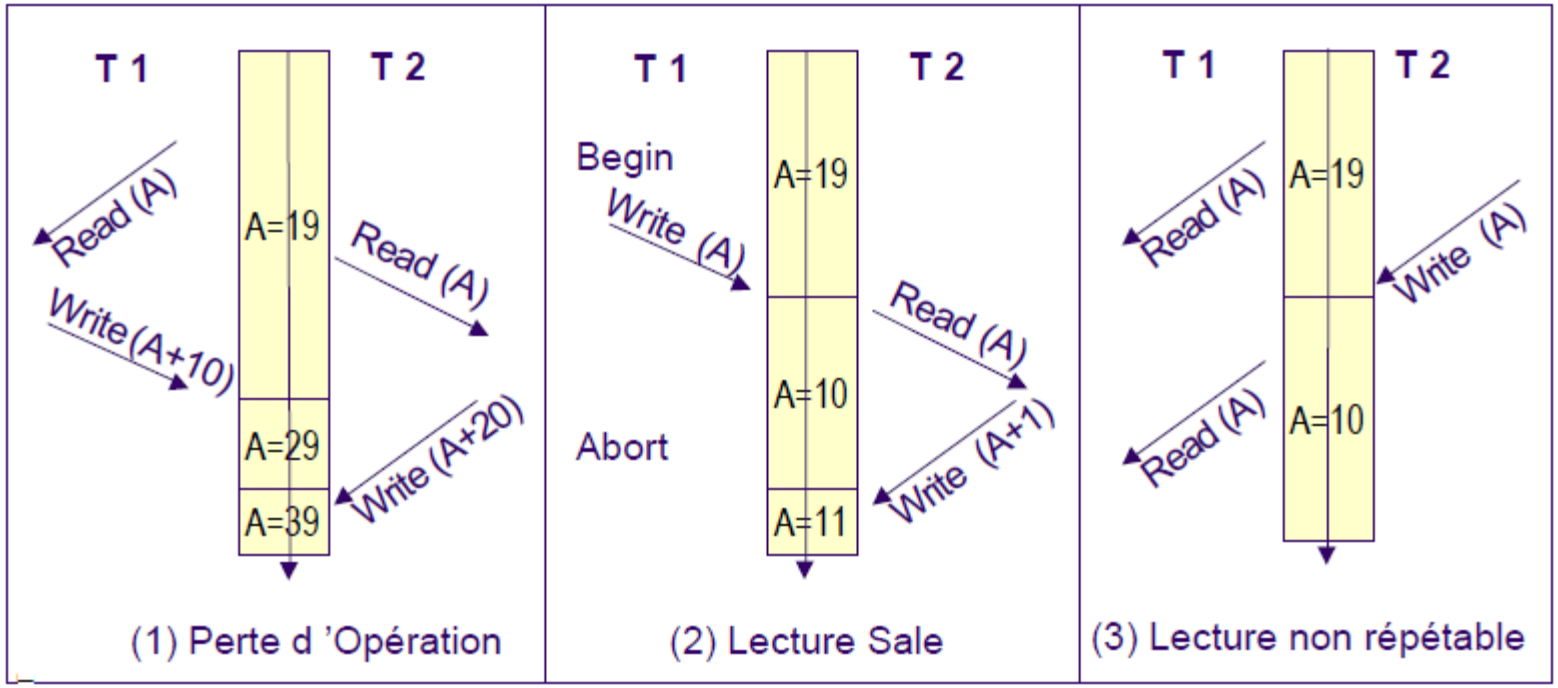
Règles de fonctionnement

- Une sous transaction démarre après la transaction mère et se termine avant elle
- L 'abandon d'une sous transaction entraîne :
  - l 'abandon de ses sous transactions descendantes,
  - pas nécessairement l 'abandon de ses ancêtres
- La validation d'une sous transaction est conditionnée à la validation de sa transaction mère
- Une sous transaction est atomique et isolée de toutes les sous transactions qui n 'appartiennent pas à sa descendance



Notion de cohérence

- Une exécution concurrente non contrôlée de plusieurs transactions crée des incohérences



Techniques de contrôle de concurrence

- **Exécution sérielle**

T1 puis T2 puis T3 puis T4
T2 puis T1 puis T3 puis T4
…

- **Sérialisabilité**

entrelacement des actions des transactions tel que le résultat est équivalent à celui d ’une des exécutions Sérielles

- **Méthodes**

Pessimiste : conflits probables
Optimiste : conflits peu probables

a) Caractérisation d ’Ordonnancements sérialisables

utilisation d’un graphe de dépendance entre transactions concurrentes

Relation de dépendance

Soit S un ordonnancement sans opération simultanée. On dit que Ti précède Tj (noté Ti<Tj) dans S si et seulement s ’il existe deux opérations non permutables Oi et Oj (parce que conflictuelles) telles que Oi est exécutée par Ti avant Oj par Tj

Théorème (Papadimitriou79)

Une condition suffisante pour qu’un ordonnancement soit sérialisable est que le graphe de précédence associé soit sans circuit

b) Techniques de Verrouillage (contrôle Pessimiste)

Opérations protocolaires

- **LOCK(g,M)** permet à une transaction de demander de poser un verrou de type M sur l ’objet g
- **UNLOCK(g)** permet à une transaction de demander de lever le verrou qu’elle a posé sur l ’objet g

Gestion de conflits:

Le système maintient une table d ’allocation de verrous sur les objets. Cette information est utilisée pour détecter les conflits.
La transaction à l ’origine du conflit attend la libération de l ’objet par la (les) transaction(s) concurrente(s).
Si le gestionnaire de verrous est distant on lui envoie un message transportant le LOCK ou le UNLOCK.
Le graphe des dépendances représente donc aussi un graphe des attentes.
Un circuit dans ce graphe signifiera donc l ’existence d ’un interblocage des transactions du circuit.

b) Techniques de Verrouillage: Condition de sérialisation

Les règles de construction suivantes représentent une condition suffisante pour garantir la propriété de sérialisabilité (Eswaran 76)

•**la transaction est bien formée**: tout accès à un objet est précédé d ’une opération de verrouillage compatible avec le mode d ’accès

•**le verouillage est à 2 phases**: aucune demande de verrouillage n ’est acceptée après la première demande de libération

c) Autres techniques de Contrôle de Concurrence

Contrôle de Concurrence optimiste (Kung et Robinson, 1981):

Idée de base

1) Allez de l ’avant et faites tout ce que vous voulez, sans faire attention à ce que les autres font

2) S ’il arrive un problème, on s ’en occupera plus tard

c) Autres techniques de Contrôle de Concurrence

Principe de fonctionnement

- considérer que chaque transaction se compose de deux étapes :
    - une étape de lecture et calcul dans un espace privé
    - une étape d'écriture réelle dans la base (commitment) (qui pourra donc échouer si concurrence)
- ordonner les transactions selon le moment où elles terminent la première étape de lecture et de calcul
- durant l ’ écriture réelle, contrôler que les accès conflictuels aux objets s'effectuent bien dans l'ordre ainsi obtenu

Evaluation

•Fonctionne bien si peu de conflits car conflits peuvent engendrer des abandons, et donc, des reprises ultérieures de transactions (beaucoup d ’abandons => grosse surcharge)

•Au contraire, les techniques pessimistes fondées par exemple sur le verouillage, et donc des attentes, est préférable pour des transactions longues.

c) Autres techniques de Contrôle de Concurrence

Technique pessimiste avec utilisation d'Estampillage (Reed 1983)

Estampille de transaction
valeur numérique unique associée à une transaction au moment de son commencement (BEGIN_TR.)
Mieux: au moment de l 'apparition d 'un premier conflit.
En réparti utiliser Horloges de Lamport

Estampille d 'objet par type d 'opération
valeur numérique associée à un objet mémorisant l 'estampille de la dernière transaction ayant opéré sur cet objet via cette opération

Principe

1- toute transaction a une estampille unique. Les estampilles engendrent un ordre total des transactions

2- une transaction ayant accédé à un objet doit stocker son estampille sur cet objet

3- une transaction voulant accéder à un objet doit comparer son estampille avec celle de l ’objet: si elle est plus jeune que celle ayant accédé en dernier à cet objet, elle obtient l ’accès sur l ’objet (et subit une attente si cet objet est en cours d ’utilisation)

4- dans le cas contraire, elle devra se suicider (l ’ordre d ’exécution indique que la transaction courante est « en retard »). La reprise se fera avec une nouvelle estampille.
=> Cette technique n ’engendre pas d ’interblocage car il y a immédiatement abandon de la transaction en retard.

d) Gestion des interblocages

Quatre stratégies de gestion

1. La politique de l ’autruche (ignorer le problème)

2. La détection (permettre aux interblocages de se produire, les détecter, puis tenter de les éliminer. Cas transactionnel: abandonner une transaction est possible alors que c ’est plus difficile pour un processus non transactionnel)

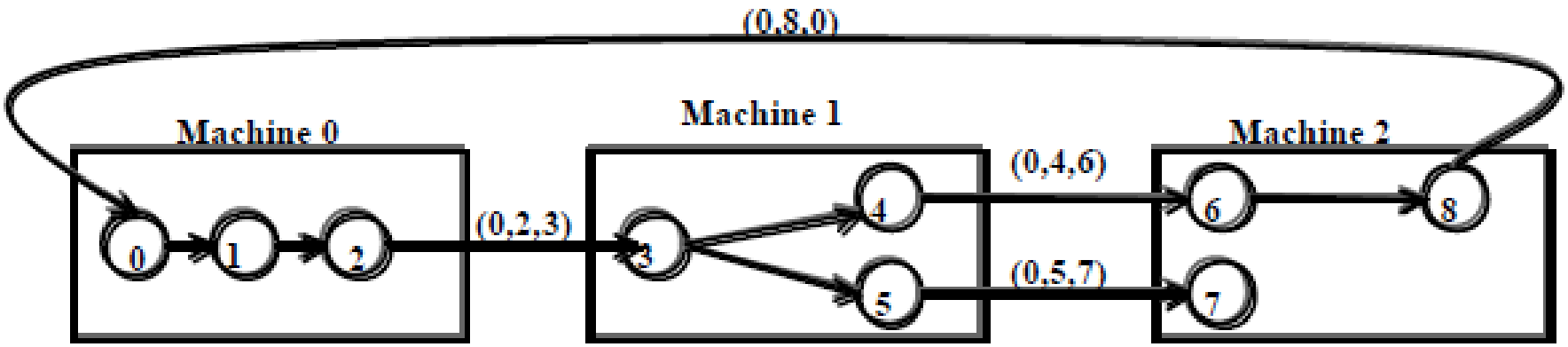
3. La prévention (rendre de façon statique les interblocages structurels impossibles)

4. L ’évitement (éviter les interblocages en distribuant les ressources avec précaution)

Détection des interblocages : algorithmes distribués

Ex: P0 se bloque sur P1 (P0 demande une ressource que P1 a verrouillé)
⇒une sonde (0, 0, 1) est émise , elle sera transmise de site en site.

Champ1: num du processus qui se bloque;
Champ2: émetteur de la sonde;
Champ3: récepteur de la sonde



Détection des interblocages : algorithmes distribués

Algorithme de Chandy-Misra-Haas (1983):

Principe d ’une sonde qui va parcourir le graphe des attentes. Si la sonde revient, alors, la nouvelle attente vient d ’engendrer un interblocage.
Guérison ? Abandonner la transaction « fautive », celle ayant le moins de verrous, la plus jeune ?

Détection/Guérison plus simple par Temporisation:

- fixer durée max d ’exécution pour une transaction
- si au bout du délai une transaction n ’a pas fini, suspicion qu’elle est interbloquée et donc abondon de la transaction

Prévention des interblocages : généralité

Principe

La prévention des interblocages consiste à construire prudemment le système de façon de rendre les interblocages structurellement impossibles. Autrement dit, elle consiste à supprimer l 'une des conditions qui rend possible l'interblocage

Contraintes d 'allocation

- un processus ne détient qu'une ressource à la fois
- un processus doit déclarer toutes les ressources dont il a besoin
- un processus doit relâcher les ressources qu'il détient pour pouvoir en obtenir une nouvelle

=> mal adapté pour verrouillage en 2 phases !

Ordonnancement des ressources

- préordonner toutes les ressources du système

- un processus doit acquérir les ressources dont il a besoin dans l 'ordre strictement croissant

Ordonnancement des transactions

- préordonner toutes les transactions par une estampille

- déterminer une politique d 'allocation des ressources adéquate permettant d 'éviter les interblocages (ex, Reed83)

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

• L'idée de l'algorithme de Rosenkrantz, Stearns et Lewis est de prévenir l'interblocage en basant uniquement sur l'estampillage.

• Principe : lorsqu'une ressource r, utilisée par une transaction T1, est demandée par une transaction T2, les conflits sont résolus par comparaison de leurs estampilles.

• Il n'y a ici aucune annonce préalable : l'algorithme proposé peut donc s'appliquer lorsque les *accès aux ressources sont définis dynamiquement*.

• Il s'agit toujours d'un algorithme de prévention : la comparaison des estampilles a pour effet d'empêcher la formation de circuits dans le graphe des conflits.

• Deux techniques sont proposées : avec ou sans réquisition.

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Technique sans réquisition : « Wait-Die System»

• On considère deux transactions t1 et t2 et une ressource r utilisée par t1 et demandée par t2.

• A la création de chaque transaction une estampille lui est associée; soient e(t1) et e(t2) les estampilles des deux transactions

• L'algorithme auquel obéit l'allocateur situé sur le site de la ressource r est le suivant :

si e(t2) < e(t1) alors bloquer t2 /* wait */
sinon annuler t2 /* die */
fsi

⇒ si la transaction t2 est la plus ancienne elle reste bloquée jusqu'à ce que t1 libère la ressource.

⇒ sinon t2 est annulée et sera redémarrée ultérieurement avec le même numéro d'estampille

⇒ cet algorithme évite la famine des transactions annulées

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Technique avec réquisition : « Wound-Wait System»

• Une solution consiste à satisfaire systématiquement les demandes de transactions plus anciennes et donc d'annuler les transactions plus jeunes qui utilisent les ressources : il y a donc réquisition.

• L'algorithme de l'allocateur situé sur le site de la ressource r est le suivant (r est utilisée par t1 et demandée par t2) :

si e(t2) < e(t1) alors annuler t1 /* Wound */
Sinon bloquer t2 /* Wait */
Fsi

• Une transaction n'attend jamais une ressource utilisée par une transaction plus jeune.

Algorithmes de prévention de Rosenkrantz, Stearns et Lewis:

Conclusion :

• Ces algorithmes ont l'avantage d'être simples

• Peuvent entraîner l'annulation de transactions qui n'auraient pas conduit à un interblocage

• Intéressant si peu de conflits sur une même ressource

Contrôle de l'atomicité

• **But**:
pouvoir « défaire » ou « refaire » ce qu 'une transaction a fait, en cas de défaillance cad, garantir la propriété d 'atomicité de la transaction.

• Défaillance parce que:

  • abandon engendré par le controle de concurrence
  • abandon programmé explicitement, ou erreur de programmation
  • panne du site ou des stockages de données

• **Principe**:
  • Garder trace de ce qui est exécuté par les transactions via la redondance : espace privé (copie de l 'espace global) sur lequel travaille la transaction, ou bien, utilisation d 'un journal de toutes les actions qui ont lieu sur l 'espace global (journal Avant ou journal Après)
  • Procéder à l 'exécution d 'une transaction en 2 phases: calcul puis validation initiée en fin de transaction. Si transaction est répartie, il faut coordonner la validation de tous les sites impliqués : validation en plusieurs phases

Contrôle de l’atomicité : « défaire » ou « refaire » en cas de défaillances

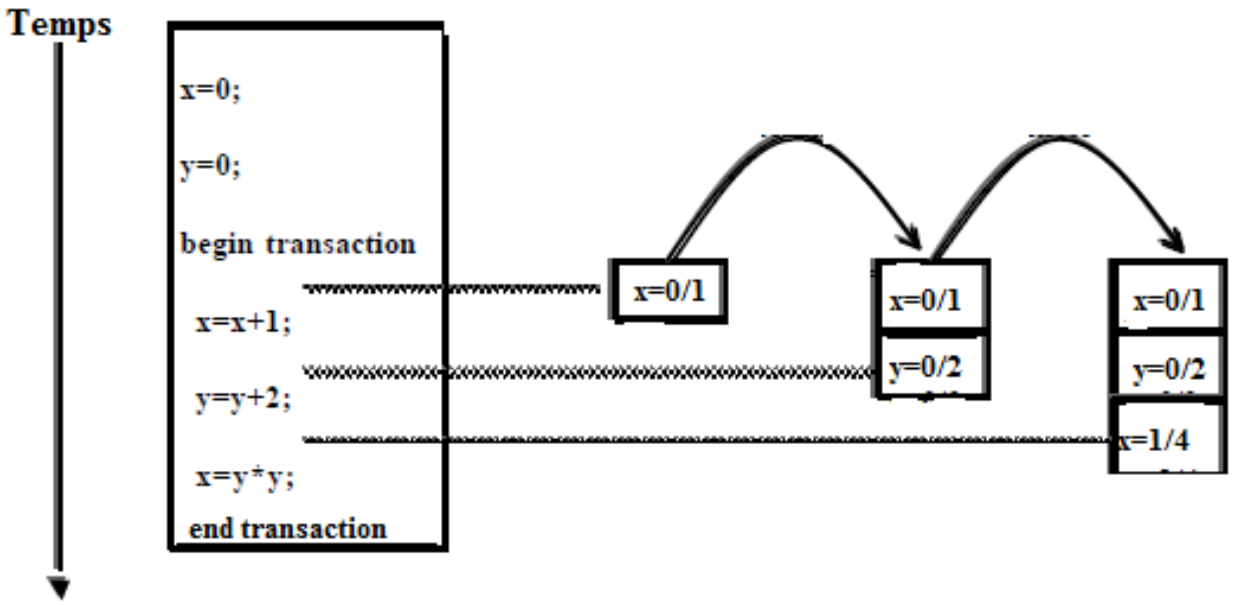
a) Mémoire d ’ombre (espace de travail privé)

Ne copier que les index de blocs et non les blocs des fichiers !

Contrôle de l'atomicité : « défaire » ou « refaire » en cas de défaillances

b)Utilisation de Journaux des inscriptions

Valeurs du journal « avant » (et stratégie de MAJ en continu) : avant de modifier une valeur, on l 'inscrit dans le log.



Contrôle de l'atomicité : « défaire » ou « refaire » en cas de défaillances

b)Utilisation de Journaux des inscriptions

Si la transaction valide, les modifications sur les fichiers ont eu lieu.
Si la transaction abandonne/plante, on peut se servir du journal avant pour défaire ce qui avait été fait avant le plantage (plantage pendant le calcul): ROLLBACK -- restauration

Avec un journal « après » et un espace de travail privé:
- les modifications n 'ont pas lieu sur les fichiers mais dans l 'espace de travail
- si plantage pendant la validation, le journal permet de finir ce que la transaction a promis de valider, cad, de finir de copier l 'espace de travail sur les vrais fichiers.

Validation atomique

- Action réalisée dans le cadre d'une transaction distribuée
- ✓ Accord entre tous les processus pour effectuer la requête de la transaction ou pas

- ✓ L'accord se passe de manière atomique pour éviter toute interférence avec d'autres transactions

- Avec les propriétés suivantes

- ✓ **Validité:** La décision prise est soit valider, soit annuler
- ✓ **Intégrité:** Un processus décide au plus une fois
- ✓ **Accord:** Tous les processus qui décident prendront au final la même décision
- ✓ **Terminaison:** Tout processus correct décide en un temps fini

Validation atomique: Principe général

- Un processus coordinateur envoie une demande de réalisation d'action à tous les processus
- Localement, chaque processus décide s'il sera capable ou non d'effectuer cette action
- Quand le coordinateur a reçu tous les décisions des processus, il diffuse la décision finale qui sera respectée par tous les processus

✓ Si tous les processus avaient répondu « oui », alors la décision est de valider et chaque processus exécutera l'action

✓ Si un processus au moins avait répondu « non », alors la décision est d'annuler et aucun processus n'exécutera l'action

➢ L'action doit être en effet exécutée par tous ou aucun

Validation atomique

- Validation atomique est simple a priori mais le devient bien moins dans un contexte de fautes …

- Contexte de fautes
  - ✓ Pannes franches de processus
  - ✓ Pertes de messages

- Pour que la validation fonctionne (plus ou moins bien) dans ce contexte, deux grands types d'algorithmes

  - ✓ Validation à 2 phases : 2 phases commit (2PC)
  - ✓ Validation à 3 phases : 3 phases commit (3PC)

Validation atomique : 2PC

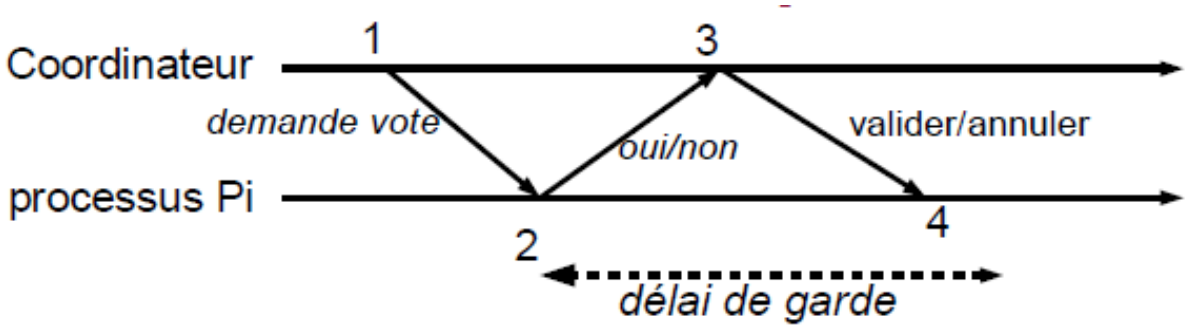
- Pour gérer perte de messages ou pannes franches
- ✓ Utilise un délai de garde : généralement un peu supérieur à 2 fois le temps de propagation d'un message

- ✓ En effet, bien souvent la communication prend la forme « envoi message » puis « réception acquittement »
  - ➢ Si pas reçu de réponse à l'envoi de message en 2 fois le temps de propagation : un message (ou l'acquittement) s'est perdu ou que le processus avec qui on communique est mort

- Validation à 2 phases (2PC)
- ✓ Première phase : demande de vote et réception des résultats
- ✓ Deuxième phase : diffusion de la décision à tous

Validation atomique : 2PC

1. Le processus coordinateur (un des processus parmi tous qui joue ce rôle) envoie une demande de vote à tous les processus
2. Chaque processus étudie la demande selon son contexte local et répond oui ou non (selon qu'il peut ou pas exécuter la requête demandée)
3. Quand le coordinateur a reçu tous les votes, il envoie la décision finale : valider ou annuler la requête
4. Chaque processus exécute la requête s'il reçoit valider ou ne fait rien s'il reçoit annuler



*Validation atomique 2PC:*Gestion des problèmes

- Si le coordinateur ne reçoit pas de réponse avant le délai de garde dela part de Pi
- ✓ Soit Pi est planté, soit la réponse de Pi est perdue ou la demande n'est pas arrivée à Pi
- ✓ Dans ce cas, on abandonne la transaction, la décision est annuler
  - ➢ On aurait pu relancer la demande à Pi, mais cela aurait retardé l'envoi de la décision finale aux autres processus et aurait déclencher d'autres problèmes

- Si la décision finale venant du coordinateur n'est pas reçue par Pi
- ✓ Le message à destination de Pi a été perdu
- ✓ Le coordinateur s'est planté et n'a pas pris ni envoyé de décision globale
- ✓ Pi peut interroger les autres processus pour savoir s'ils ont reçus une décision globale et laquelle
  - ➢ Si aucun processus ne peut lui répondre (soit ils ont rien reçu, soit ceux qui ont reçu se sont plantés)

Si Pi avait voté non : pas de problème, c'est forcément annuler

Si Pi avait voté oui : Pi n'a aucune idée de la décision finale qui a (peut-être) été prise

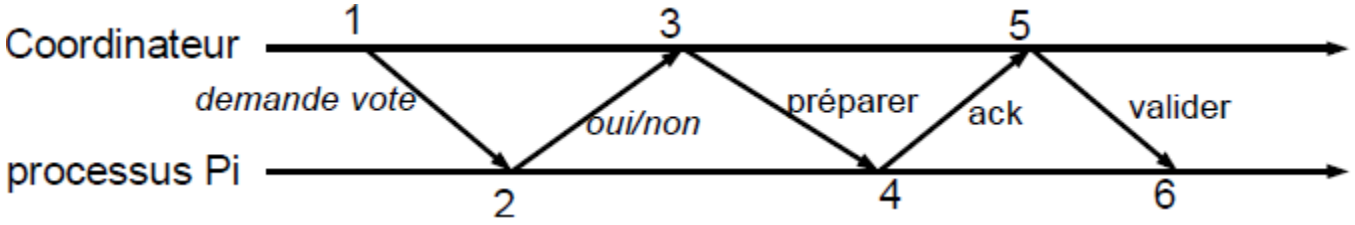
Validation atomique : vers la 3PC

- En conclusion sur la 2PC
- ✓ On peut se retrouver dans des situations où un processus ne sait pas quoi faire
- ✓ Le problème est qu'on attend pas de savoir que tout le monde est bien au courant de la décision finale avant d'exécuter ou pas la requête
  - ➢ La validation atomique à 3 phases rajoute une étape pour cela

- Validation atomique à trois phases (3PC)
- ✓ Première phase : demande de vote et attentes des votes (idemque pour 2PC)
- ✓ Deuxième phase : envoie à tous de la décision globale et attentede l'acquittement de la réception
- ✓ Troisième phase : diffusion de la demande d'exécution de la requête (sauf si décision annuler)

Validation atomique : 3PC

1. Le coordinateur envoie une demande de vote à tous les processus
2. Chaque processus étudie la demande selon son contexte local et répond oui ou non
3. Quand le coordinateur a reçu tous les votes, il envoie
- ✓ Soit annuler si un processus avait voté non
  - ➢ La validation est alors terminée, pas d'autres échanges de messages
- ✓ Soit une demande au processus de se préparer à exécuter la requête
4. Chaque processus envoie un acquittement au coordinateur de la réception de cette demande
5. Une fois tous les acquittements reçus, le coordinateur diffuse la validation finale
6. Chaque processus exécute alors la requête



Validation atomique 3PC: Gestion des problèmes

- Si un processus reçoit un « préparer » de la part du coordinateur, il sait alors que quoiqu'il arrive tous les processus ont décidé de valider la requête

- Donc même si l'acquittement qu'il a envoyé au coordinateur est perdu et/ou même si le processus ne reçoit pas le « valider » du coordinateur

✓ Le processus exécutera la requête car il sait que tout les processus avait décidé qu'ils pouvaient l'exécuter

Détection de terminaison d'une application répartie

Introduction

Ce n'est pas parce qu'un processus est momentanément passif (plus de messages à traiter en attente), que l'ensemble des processus de l'application est terminé
En effet , si un processus est encore actif, il peut tout à fait envoyer un message à un processus qui était passif

But

déterminer de manière complètement répartie un état global cohérent de l'application, ici, l' état de terminaison
Impossible de questionner les processus TOUS en MEME TEMPS !
Donc, difficile d 'obtenir une vision globale correcte en se basant uniquement sur des visions locales pas faites au même instant...

Détection de la terminaison

Principe

Méthodes permettant de passer d'un état message terminaison à un état process terminaison. Basées sur le schéma suivant :
1 détection de la configuration terminale
2 diffusion de l'information à tous les processus
3 passage dans un état terminal.

Phases

Algorithme de base : gestion de l'état des processus,
Algorithmes de contrôle : ajoutés pour gérer la terminaison,

Algorithme de base

Etat des processus

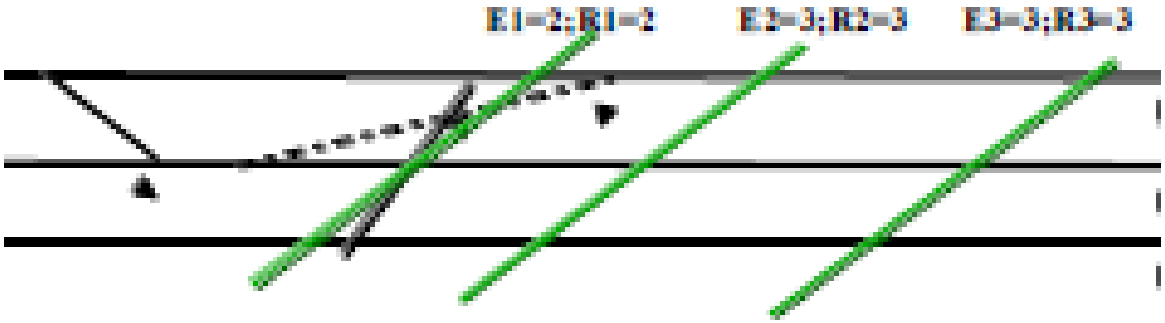
- Deux états possibles : etatp = (actif ; passif)
- Un processus est actif (resp. passif) si un événement interne ou une réception de message est (resp. n'est pas) accessible à son état.
- Un processus p passif n'admet que des réceptions et peut à tout instant devenir actif à la reception d'un message.

Règles de changement d'état des processus

Un processus actif devient passif uniquement suite à un événement interne.
Un processus devient actif à la suite de la réception d'un message.
Les événements suite auxquels p devient passif sont des événements internes à p.

Algorithme des 4 compteurs (Mattern)

•Comptabiliser, sur chaque processus, le nombre de message envoyés et reçus
•Sommer ces nombres
•Si E=R, alors, ce n'est pas forcé que ce soit fini
•Donc, recommencer une seconde collecte des compteurs locaux
•Si E1=R1=E2=R2 (4 valeurs), Alors , l'application est bien terminé



•La vague de collecte peut doubler des messages de l'application (si par ex, voies entre 2 processus non FIFO)
•Quand c'est fini, il faut 2 vagues pour le constater

Algorithme de visite des sites (Actif/Passif), sur un anneau (Misra 1983)

Hypothèses et principe

• aucune hypothèse sur la topologie des voies de communication ni sur le délai de transfert des messages.

• Les seules hypothèses sont :
- pas de perte des messages
-pas de déséquencement (canaux FIFO)

• La terminaison est réalisée lorsque tous les processus sont passifs et qu'il n'y a plus de messages en transit.

• L'algorithme de Misra consiste aussi à faire visiter les processus par un jeton. la constatation par le jeton que tous les processus sont passifs lorsqu'ils ont été visités ne permet pas de conclure à la terminaison : des processus ont pu être réactivés et des messages peuvent être en transit.

Algorithme de visite des sites (Actif/Passif), sur un anneau (Misra 1983) (suite)

• Dans le cas particulier où la topologie de communication entre processus est un anneau, le jeton peut affirmer que le calcul est terminé si, après avoir effectué un tour sur l'anneau, il constate que chaque processus est resté passif depuis sa dernière visite à ce processus.

• En effet, les messages ne pouvant pas se doubler, entre deux visites du jeton un processus a nécessairement reçu les messages qui étaient en transit lors de la première visite du jeton.

⇒ si le jeton a effectué deux tours sur l'anneau et qu'il n'a observé que des processus restés en permanence passifs, on peut conclure à la passivité des processus et à l'absence de messages en transit, i.e à la terminaison.

Hypothèses et principes

Comportement du jeton

• lorsqu'un processus devient actif il devient noir, et c'est le jeton qui le peint en blanc lorsqu'il quitte le processus (passif).

• ainsi si le jeton retrouve blanc un processus, c'est que ce dernier est resté passif en permanence depuis sa dernière visite.

• lorsque le jeton a visité les n processus et qu'il les a trouvés tous blancs, il peut en conclure la terminaison (initialement les processus sont noirs).

• tous les processus ont des comportements identiques

Hypothèses et principes (suite)

Pour admettre une topologie quelconque pour les communications et ne pas se limiter à un anneau de contrôle, le jeton doit :
- visiter tous les processus,
-s'assurer qu'il n'y a plus de messages en transit.

⇒ le jeton doit parcourir chaque arc du réseau

⇒ si le réseau est fortement connexe alors il existe un circuit C qui comporte chaque arc du réseau (au moins une fois).

⇒ il suffit alors de remplacer l'anneau par un tel circuit

L'algorithme

Le jeton véhicule une valeur j : les processus visités lors des j dernières traversées des voies de communication sont restés en permanence passifs.
On suppose que sur le circuit C (pré-calculé) les fonctions suivantes sont définies :
- fonction *taille (C : circuit) résultat entier : donne la taille du circuit*
- fonction *successeur (C : circuit, i : 1 .. n) résultat 1 .. n : donne pour le* processus Pi qui l'exécute, le numéro du successeur de Pi sur le circuit.
Le jeton détectera la terminaison lorsque l'on aura : *j = taille (C)*
Chaque processus Pi est doté des déclarations suivantes :
var couleur : (blanc, noir) initialisé à noir ;
état : (actif, passif) initialisé à actif ;
jeton_présent : booléen initialisé à faux ;
nb : entier initialisé à 0 ;
- la variable « *couleur » est associée au processus et « nb » sert à mémoriser la* valeur associée au jeton entre sa réception et sa réémission.
- les messages sont de deux types : *messages, jeton*.

L’algorithme

Début
⇒ lors de réception de (message, m) faire
début
état → actif ;
couleur → noir ;
fin
⇒ lors de attente (message, m) faire
début
état → passif;
fin

L’algorithme

⇒ lors de réception de (jeton, j) faire
début
nb →j ;
jeton_présent →vrai;
si nb = taille (C) et couleur = blanc alors
terminaison détectée
fsi ;
fin

L’algorithme

⇒ lors de émission de (jeton, j) faire
début
si jeton_présent et état = passif alors
début
si couleur = noir alors nb → 0
sinon nb → nb + 1 ;
fsi ;
envoyer (jeton, nb) à successeur (C, i) ;
couleur → blanc ;
jeton_présent → faux ;
fin
fsi
fin
Fin

Conclusion

• l'algo de Misra peut être généralisé au cas où le réseau de communication n'est pas fortement connexe : le réseau est décomposé en ses composantes maximales fortement connexes, et le jeton visite alors successivement ces composantes selon un ordre prédéfini.

• il existe d'autres algorithmes qui utilisent l'estampillage (algo de Rana) : le principe de cet algo est voisin du principe d'extinction sélective des messages de l'algo d'élection de Chang et Roberts :
⇒ il s'agit d'élire le dernier processus terminé qui diffusera ensuite l'information *terminé*.

Merci pour votre attention

Sockets et leur mise en œuvre en C

Les modes de connexion

Le mode *non connecté :* ne garantit pas :

- l'intégrité des données
- l'ordonnancement des données
- la non-duplication des données

❑Ces propriétés doivent être garanties par l'application.

Le mode *connecté :*

- garantit les propriétés ci-dessus
- implique une diminution des performances brutes par rapport au mode non-connecté.
- Permet une implémentation asynchrone des échanges

Les clients

- Une application cliente est moins complexe qu'une application serveur.
- La plupart des applications clientes ne gèrent pas d'interactions avec plusieurs serveurs.
- La plupart des applications clientes sont traitées comme un processus conventionnel ; un serveur peut nécessiter des accès privilégiés.

Les serveurs

Le processus serveur :

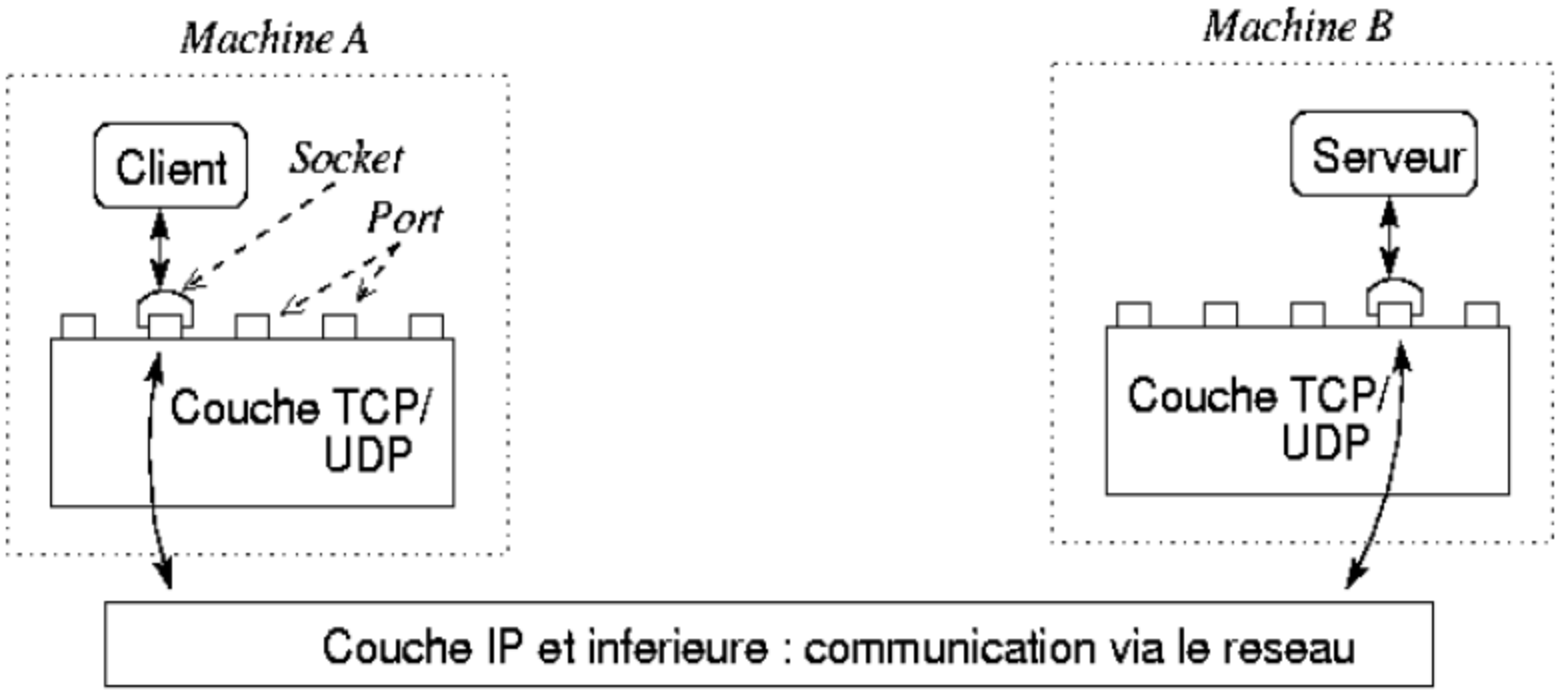
- offre un point d'entrée sur le réseau
- entre dans une boucle infinie d'attente de requêtes
- à la réception d'une requête, déclenche les processus associés à la requête, puis émet la réponse vers le client

- Deux types de serveurs :

✓ itératifs : ne gèrent qu'un seul client à la fois

✓ parallèles : fonctionnent en mode concurrent

Les sockets



Une socket est
✓Un point d'accès aux couches réseau TCP/UDP
✓Liée localement à un port
✓Adressage de l'application sur le réseau : son couple @IP:port
✓Elle permet la communication avec un port distant sur une machine distante : c'est-à-dire avec une application distante

Définition des sockets

Socket : prise

- Associée, liée localement à un port
- C'est un point d'accès aux couches réseaux
- Services d'émission et de réception de données sur la socket via le port
- En mode connecté (TCP)

✓ Connexion = tuyau entre 2 applications distantes

✓ Une socket est un des deux bouts du tuyau

✓ Chaque application a une socket locale pour gérer la communication à distance

- Une socket peut-être liée

✓ Sur un port précis à la demande du programme

✓ Sur un port quelconque libre déterminé par le système

✓ Par défaut, on ne peut lier qu'une socket par port

Principes de bases

- Les sockets permettent l'échange de messages entre 2 processus, situés sur des machines différentes

1- Chaque machine crée une socket,

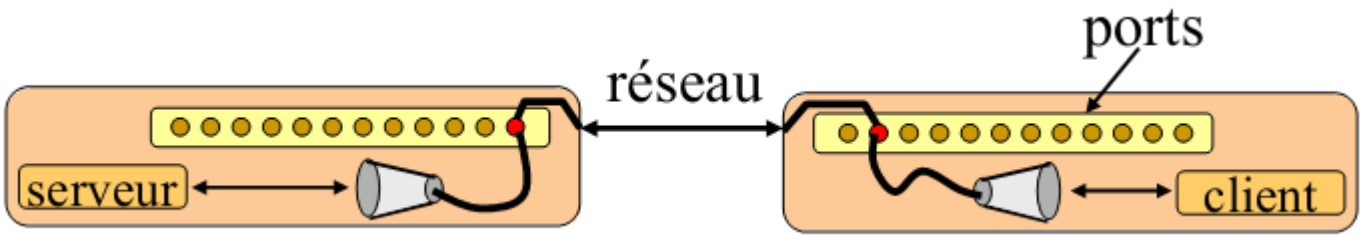
2- Chaque socket sera associée à un port de sa machine hôte,

3- Les deux sockets seront explicit ement connectées si on utilise un protocole en mode connecté …,

4- Chaque machine lit et/ou écrit dans sa socket,

5- Les données vont d'une socket à une autre à travers le réseau,

6- Une fois terminé chaque machine ferme sa socket.



Caractéristiques des sockets

Une socket possède trois caractéristiques :

- **Type:** Décrit la manière dont les données sont transmises : SOCK_DGRAM, SOCK_STREAM, SOCK_RAW.
- **Domaine:** Définit le nommage des sockets et les formats d'adressage utilisés : AF_UNIX, AF_INET, AF_INET6.
- **Protocole:** Décrit le protocole de communication à utiliser pour émettre et recevoir les données. Généralement déterminé par le domaine : PF_UNIX, PF_INET, PF_INET6.

Les domaines des sockets

- Le domaine d'une socket permet de spécifier le mode d'adressage et l'ensemble des protocoles utilisables par la socket.

- Les domaines d'adressage les plus courants sont Unix (AF_UNIX) et IP (AF_INET), voire IPv6 (AF_INET6).

Les domaines des sockets

• Exemples de familles de sockets :
- processus sur la même station Unix :
→ sockets locales (AF_UNIX)
- processus sur des stations diffé rentes à travers Internet :
→ sockets Internet (AF_INET)
• Exemples de modes de communication :
- Datagrames – ou mode non connecté (SOCK_DGRAM)
- Flux de données – ou mode connecté (SOCK_STREAM)
• Exemples de protocoles de sockets :IP, UDP, TCP,

Adressage

- Lorsqu'un processus serveur veut fournir des services, il a besoin d'un port d'attache pour la localisation de ses services.
- Un certain nombre de numéros est réservé aux services standards.
- Pour le serveur, les ports inférieurs à 1024 ne peuvent être utilisés que par le super-utilisateur root.
- Les numéros de port standard sont définis dans le fichier /etc/services, et accessibles via la commande getservbyname().

Structures de socket

- Les structures sockaddr et sockaddr_in sont définies comme suit :

```
struct sockaddr {
        sa_family_t     sa_family;      /* domaine
        char            sa_data[14];    /* adresse
}

struct sockaddr_in {
  sa_family_t           sin_family;     /* domaine
  unsigned short int    sin_port;       /* numro de
  struct in_addr        sin_addr;       /* adresse
  unsigned char         __pad[__SOCK_SIZE__ - ...]
}
```

Création d'une socket

- On crée une socket avec la primitive socket() :

s = socket (AF_INET, SOCK_STREAM, 0);

- s est l'identificateur de la socket. On peut l'utiliser pour les opérations de lecture/écriture.

Gestion des connexions

- bind() permet d'associer une adresse à une socket :

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)

- connect() est utilisée par le client pour établir la connexion :

int connect(int sockfd, const struct sockaddr,*serv_addr, socklen_t addrlen);

- listen() est utilisée par le serveur pour établir la connexion :

int listen(int s, int backlog);

Gestion des connexions (suite)

- accept() est utilisée par le serveur pour indiquer qu'il est prêt à accepter des connexions. Appel généralement bloquant :

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

- shutdown() et close() permettent de fermer une connexion :

int shutdown(int s, int how);

Transfert en mode connecté

- Les commandes send() et recv() permettent d'échanger des données :

int send(int s, const void *msg, size_t len, int flags);
int recv(int s, void *buf, size_t len, int flags);

- On peut préciser des options dans les champs flags. Par exemple, MSG_OOB pour les données urgentes.

- Il est également possible d'utiliser les appels read() et write() qui offre moins de possibilités de contrôle.

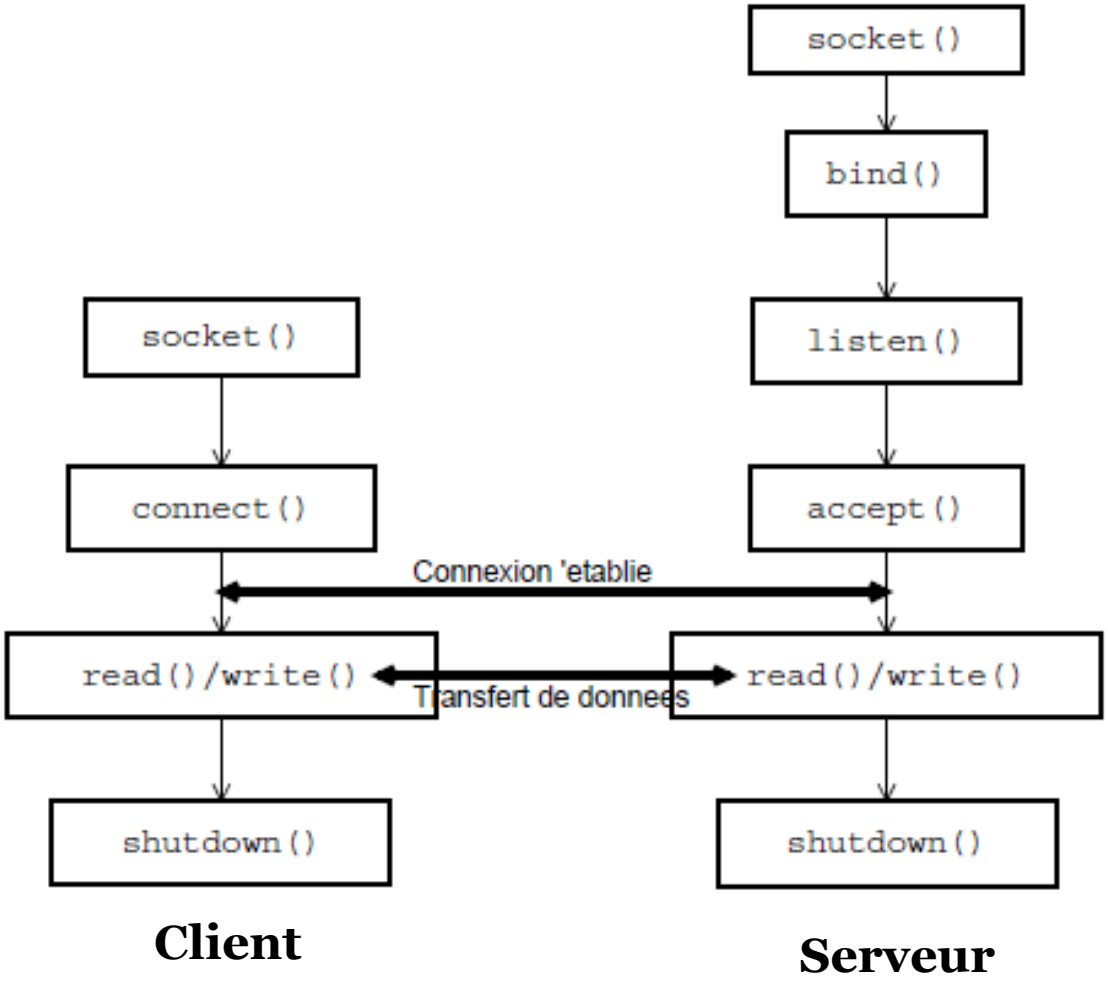
Client-serveur en mode connecté

Principe de communication

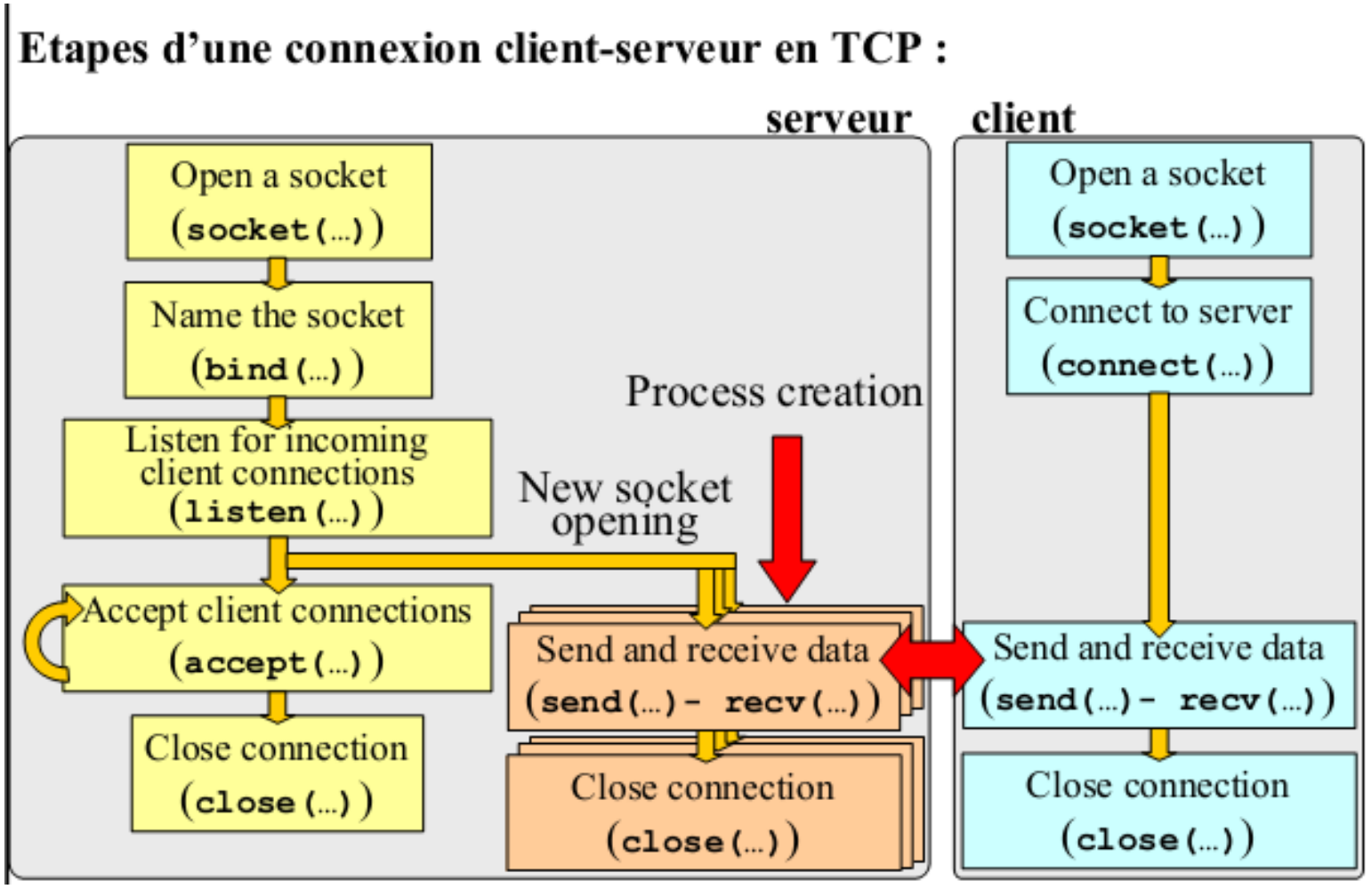
- Le serveur lie une socket d'écoute sur un certain port bien précis et appelle un service d'attente de connexion de la part d'un client
- Le client appelle un service pour ouvrir une connexion avec le serveur
- Il récupère une socket (associée à un port quelconque par le système)
- Du coté du serveur, le service d'attente de connexion retourne une socket de service (associée à un port quelconque)
- C'est la socket qui permet de dialoguer avec ce client
- le client et le serveur communiquent en envoyant et recevant des données via leur socket

Client-serveur en mode connecté

- Dans ce mode, on établit d'abord une connexion. On peut ensuite transférer des données à travers le tuyau créé.



Client-serveur en mode connecté



Client-serveur en mode connecté

Etapes d'une connexion client-serveur en TCP :

- le serveur et le client **ouvrent** chacun une « socket »
- le serveur la **nomme** (il l'attache à un de ses ports (un port précis))
- le client n'est pas obligé de la nommer (elle sera attachée automatiquement à un port lors de la connexion)
- le serveur **écoute** sa socket nommée
- le serveur **attend** des demandes de connexion
- le client **connecte** sa socket au serveur et à un de ses ports (précis)
- le serveur **détecte** la demande de connexion
- une nouvelle socket est ouverte automatiquement
- le serveur crée un processus pour **dialoguer** avec le client
- le nouveau processus continue le dialogue sur la nouvelle socket
- le serveur attendre en parallèle de nouvelles demandes de connexions
- finalement toutes les sockets doivent être **refermées**

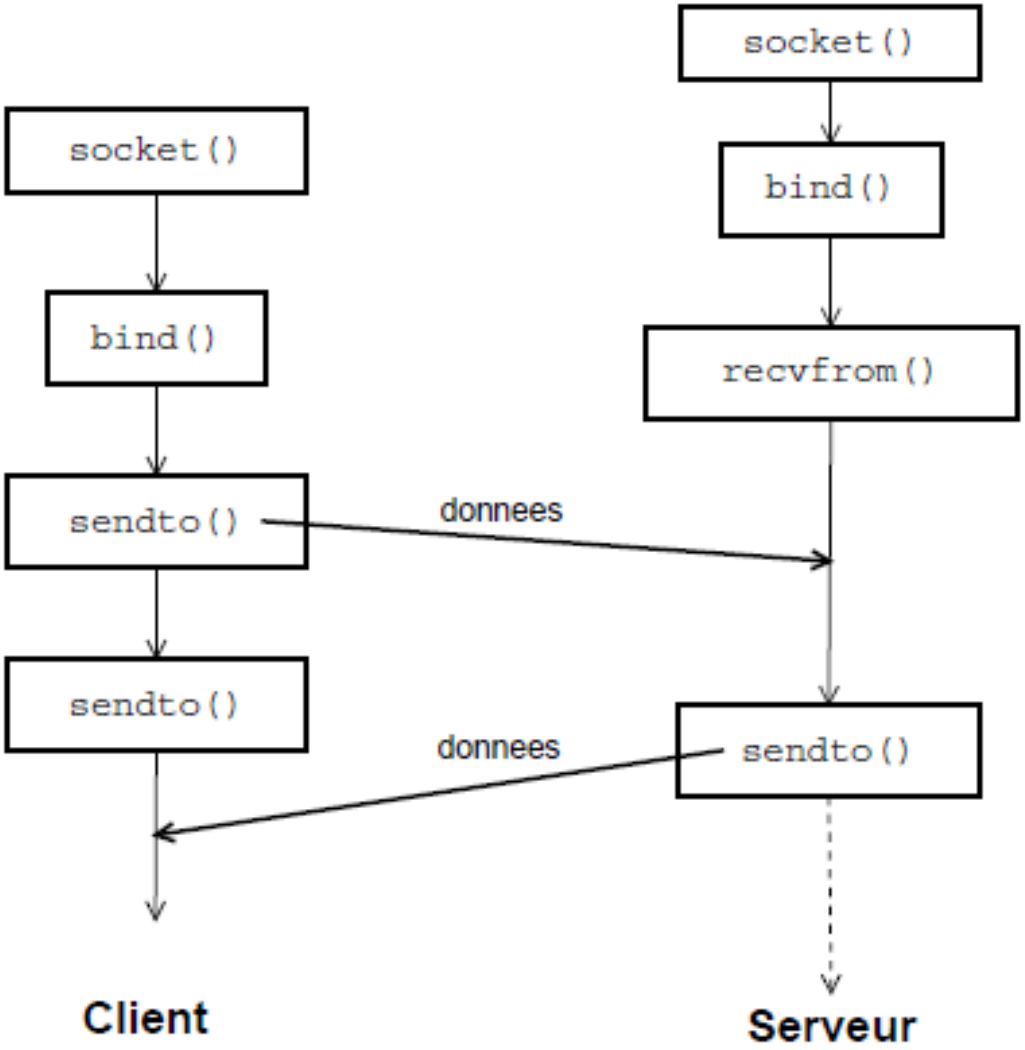
Utilisation des sockets en mode connecté SOCK_STREAM

- Côté Client (demandeur de la connexion) : le socket est dit actif
- crée un socket socket()
- se connecte à une <adresse,port> connect()
- lit et écrit dans le socket read(),recv();write(),send()
- ferme le socket close()

- Côté Serveur (en attente de connexion) : le socket est dit passif
- crée un socket socket()
- associe une adresse au socket bind()
- se met à l ’écoute des connexions entranteslisten()
- accepte une connexion entrante accept()
- lit et écrit sur le socket read(),recv();write(),send()
- ferme le socket close()

Client-serveur en mode non-connecté

- Dans ce mode, on n'établit pas de connexion au préalable.



Utilisation des sockets en mode non connecté SOCK_DGRAM

Côté Emetteur
- crée un socket socket()
- associe une adresse au socket bind()
- envoi d ’un message dans le socket sendto()/sendmsg()
- libère le socket close()

Côté Récepteur
- crée un socket socket()
- associe une adresse au socket bind()
- écoute et réception d ’un message recvfrom()/recvmsg()
- libère le socket close()

Transfert en mode non-connecté

- Aussi appelé *mode datagramme*.
- On utilise les commandes sendto() et recvfrom().

```
int
sendto(int s, const void *msg, size_t len, int flags
       const struct sockaddr *to, socklen_t tolen);
int
recvfrom(int  s,  void  *buf,  size_t len, int flags
         struct sockaddr *from, socklen_t *fromlen);
```

Obtention d'informations

- getpeername : retourne le nom de la machine connectée à une socket

Merci pour votre attention