CM - CSR

Table des matières :

Introduction

Définition

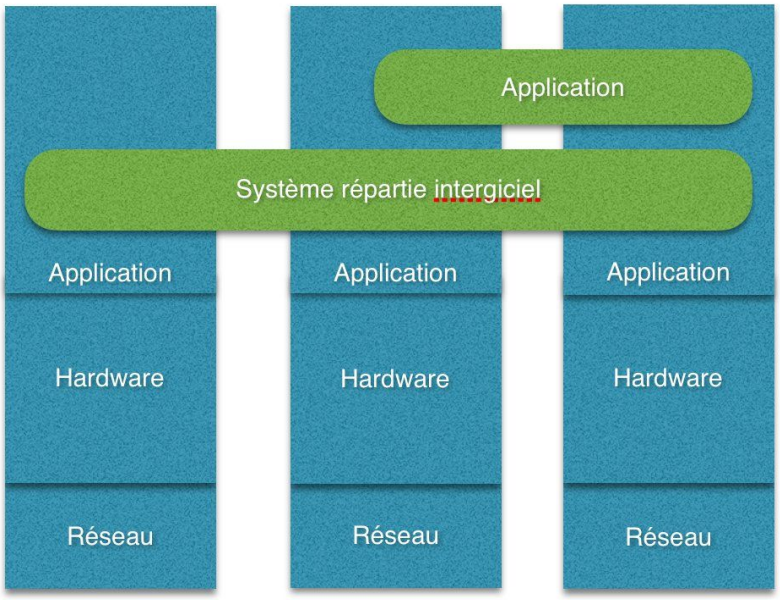
Collection d'ordinateurs indépendants qui apparaissent comme un système unique.

Caractéristiques

- indépendant / autonome (géographiquement distant)
- 1 seul système (du point de vue de l'utilisateur)
  => Mise en place de moyens de coopération

éléments

- hétérogènes : tout type de machine exemple : montre connecté, super-calculateur
- Distribués (temps de communication)
- Faillible (panne matériel)
- Pas de point de vue central (connaître l'observateur)



L'intérêt est d'utiliser un middleware (intergiciel) qui permet de repartir une application sur plusieurs serveurs (exemple : GFS Système de fichiers répartis)

1 Buts des systèmes répartis

- Rendre les ressources accessibles (sans se poser la question de Où? Quel protocole?)
  - imprimantes

    - fichiers
        - NFS
        - GFS
    - Microscopes
    - Télescopes (agrégation de plusieurs petits télescopes et reconstitution d'image car impossibilité de créer 1 seul gros télescope à cause des contraintes physiques de la Terre)

⇒ Sûreté de fonctionnement (la chose à telle était faite? et bien faite? Coeur artificiel)

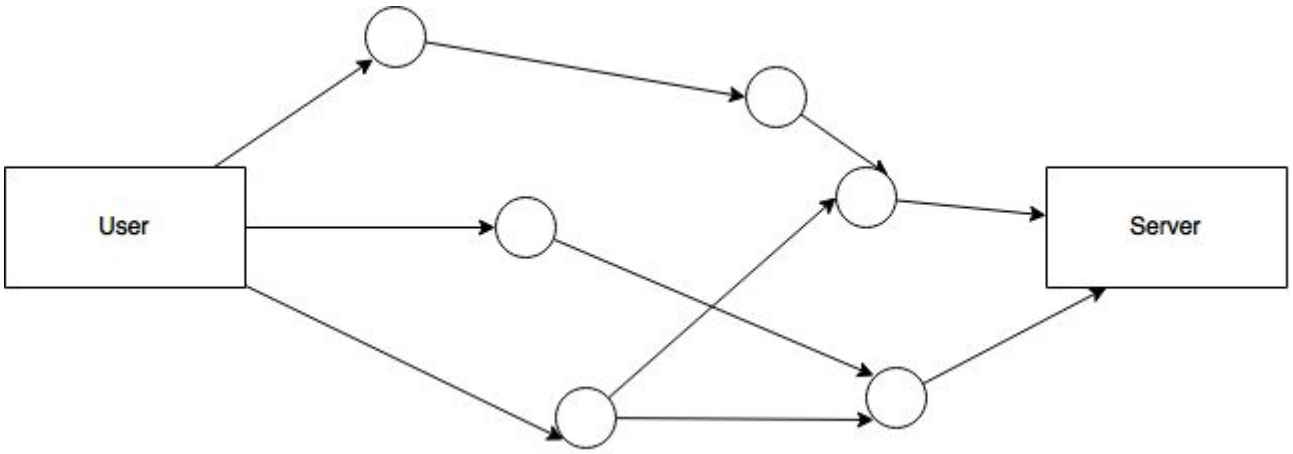
⇒ Sécurité (authentification, chiffrement, protection des données personnelles)

- Virtualiser de la complexité = rendre transparent la complexité pour l'utilisateur (capteurs dans un champs agricoles, valeur moyenne de l'humidité = on virtualise la réalité)
    - peu importe la technologie utilisation (**Accès**)
    - peu importe la **localisation** des données
    - **relocalisation** / **migration** des données
    - **réplication** des données (utilisateur ne doit pas savoir que les données sont répliquées et/ou dupliquées pour assurer les sauvegardes)
    - **Concurrence** (plusieurs utilisateurs, fonctionnement identique et indépendant de la charge utilisateurs)
    - **Fautes** (tolérances aux pannes, on sait qu'il va y avoir des fautes c'est devenu le cas normal)
    - Limites (pour facturation)
        - accounting (décompte de la consommation)
        - cohérence (garantie sur les données)
        - latence du système (si une machine est indisponible, il faut que ça entraîne une latence la plus petite et insensible possible pour l'utilisateur)
- ouvert (ajout facilité de ressources au système)
- Passage à l'échelle (démultiplication des ressources pour matcher les besoins)

1.1 Technologie

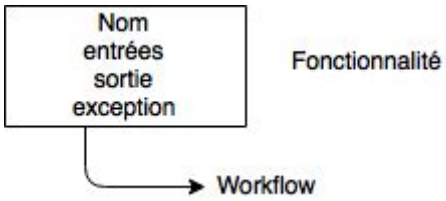
TCP / IP

C'est un système distribué avec une vision locale pour chaque routeur pourtant on arrive à avoir un comportement global qui permet de router un paquet de n'importe quelle source vers n'importe quelle destination.
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm#Applications_in_routing



1.1.1 Spécification

IDL : Interface de définition de langage
On utilise un IDL pour définir les capacités / fonctionnalités du système.



⇒ Cela nous donne le workflow.
La partie technique est différentes des politiques mises en place.

1.1.2 Techniques

Elle reste très compliquée et finalement peu vraie dans les faits. Peu de réutilisation de briques logicielles (hors bibliothèques extérieures)

- Briques
- Réutilisabilité
- Sémantique (RDF - OWL)
- Module à composants (Fractal)

1.1.3 Politique d'utilisation

- Autorisation d'accès

1.2 Passage à l'échelle

Deux échelles finalement très simples à modéliser pour un développeur et elles apportent peu de différences par rapport à un cas normal:

- Échelle géographique (attendre une réponse ayant un énorme délais, sonde Philae ⇒ Facile logiciellement d'attendre)

- Échelle du nombre d'évènement (la souris donne un flux constant d'évènement juste pour savoir si on a bougé d’un pixel ⇒ Facile)

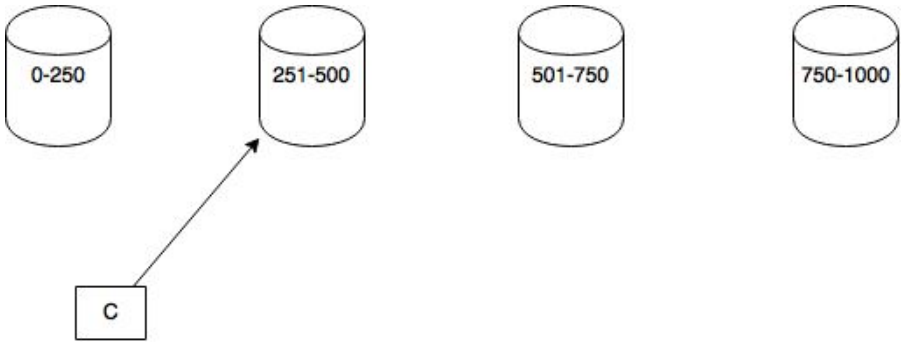
Échelle plus complexe :

- Échelle du nombre d’éléments (Connaître l’impact d’une requête supplémentaire en plus des 117 000/s arrivant sur le datacenter de Google ⇒ Pas de contrôle, Difficile)
    - Complexité structurelle
        - machines
        - services
        - administrateurs
        - données sensibles (carte vitale)

1.2.1 Point de vue performance

Les algorithmes naïfs sont linéaires en O(n), exemple on double le nombre de machines cela double le temps. Les solution sont :

- Réplication / Répartition : CDN (Content Delivery Network) Ils améliorent la latence de la livraison de données en cachant les data au plus proche de l'utilisateur. Cela évite de demander à la source en plusieurs fois. Par exemple AKAMAI.
    - Réduction de la latence
    - Réduction de la charge de la source
- Structuration : (récupération de données chez un hébergeur, indexation au lieu de parcours des serveurs)



- Cacher la latence :
    - Techniques asynchrones
        - Callback
        - Interruptions
    - Recouvrir la latence (Faire un calcul et la communication en même temps au lieu de le faire l'un à la suite de l'autre)

1.2.2 Compromis

On utilise une méthodologie en fonction de l'utilisation, exemple du CDN : ne pas utiliser la **réplication** si beaucoup de mises à jour de données, il faut préférer la **structuration**.

1.3 Fausses Hypothèses

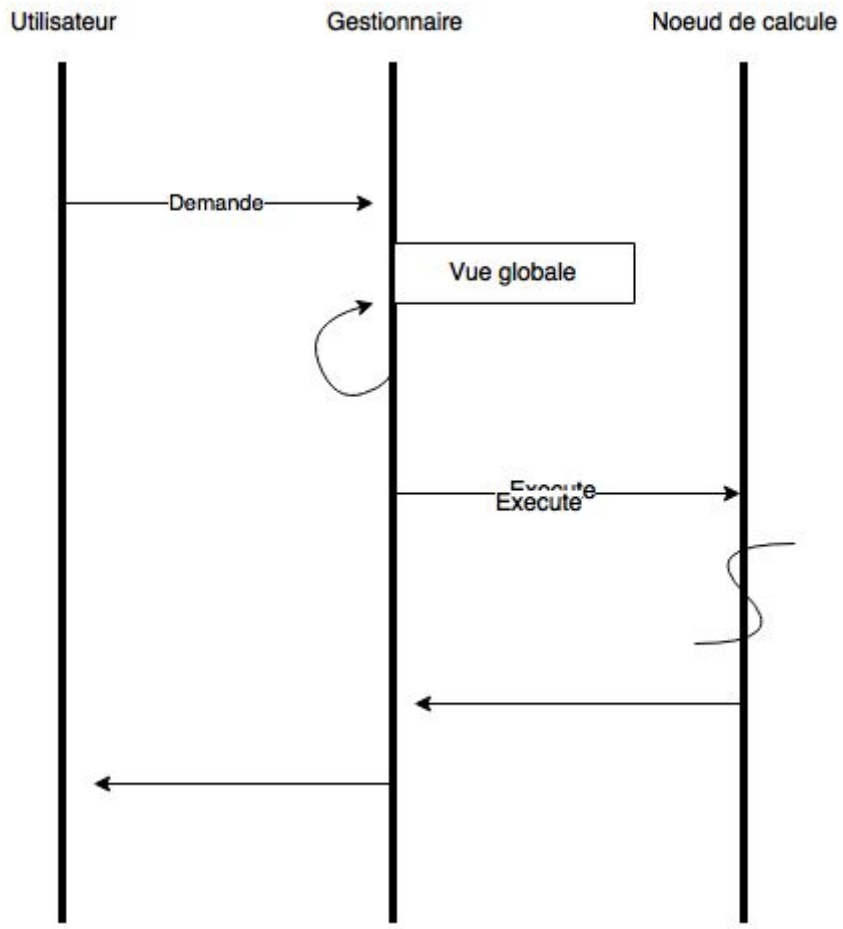
- Le réseau est fiable et performant = FAUX
    - TCP on peut avoir des timeouts
    - Latence non bornée
    - Bande passante limitée
- Utilisateur coopératifs = FAUX
    - Hackers
- Matériel ne tombe jamais en panne = FAUX
    - pas fiable
    - pas stable
- Point de vue global = FAUX
    - Cohérence des opérations
- Coûts de transport = FAUX
    - Coût électrique
    - Coût système
    - Amazon EC2 = 10cents / Gb
- Administration centralisée = FAUX
    - impossible de changer un protocole de routage
        - BGP
        - IPv6

2 Types de systèmes distribuées

2.1 Calcul (Cluster)

Propriétés d'un cluster

- homogène
- localisé
    - physiquement
    - administration
    - organisation
        - first-fit (mettre dans le premier créneau futur disponible)
        - best-fit (mettre dans le créneau disponible le plus court)
        - back-filling (décalage temporel)



2.1.1 Grilles de calcul

Mise en réseau de plusieurs clusters (10 minimum). Exemple : Egee, cluster du CERN réparti sur 30 sites.
Propriétés de la grille :

- Hétérogénéité du matériel
- Administration
    - plus complexe
    - répartition des usages avec des **VO : Virtual Organisation** pour organiser les demandes avec des règles
    - Middleware : gLite (Europe, Open Grid Forum, CERN) et Globus (US)

2.2 Système d’information

2.2.1 Les problématiques d’un système d’information

- cohérence
- intéropérabilité
    - accès
    - formats des données
- intégration
    - serveur
    - distribuée sémantiquement

2.2.2 Échanges transactionnels

Propriétés ACID comme dans les bases de données.

- Atomique
- Cohérence
- Isolation
- Durabilité

2.2.3 EAI Entreprise Application Intégration

C’est une vision haut niveau des sockets

- RPC (xml-rpc)
- RMI (Tavor)
- publish/subscribe
- système orienté message
- asynchrone

2.3 Systèmes pervasifs IoT

Des petits systèmes présent partout avec des contraintes de ressources

- calcul
- stockage
- batterie

Exemples :

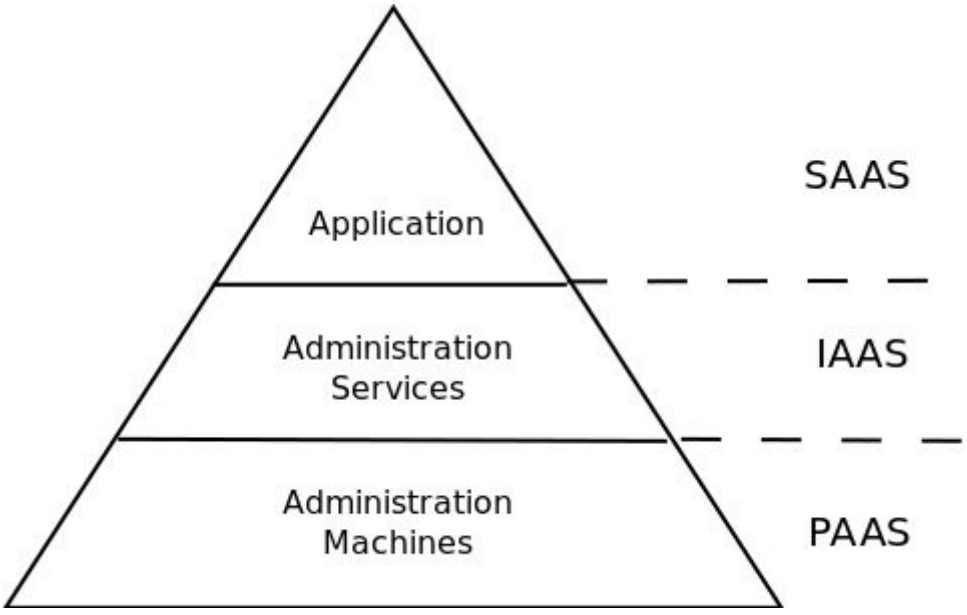
- Système de mesure
    - champs
    - maison
- Système de santé
- Réseaux de senseurs
    - champs
    - feu de bois

Problématiques :

- Sécurité (accès aux données)
- Structuration
    - algorithmes de routage vers points
        - découverte de l'arbre courant
        - routage
    - routage Ad_hoc
        - VAnet
    - Réseaux Industriels
        - Sygfox

2.4 Cloud

- SAAS : Software As A Service
- IAAS : Infrastructure As A Service
- PAAS : Plate-forme As A Service



3 Architectures

Introduction :

- logicielle
- système
    - client/serveur (Web + BD)
- de gestion
    - observation
    - reconfiguration

3.1 Architecture Logicielle

Définition :

- Type de composants
    - interaction
    - capacités

Pour le remplacement et la ré-utilisabilité.

Exemples :

A. Modèle en couche
    a. TCP / IP -> OSI
B. Modèle à base d'objet
    a. RMI
C. Modèle à base de données
    a. SGBD
    b. NFS / GFS
D. Modèle orienté évènements
    a. publish / subscribe

La puissance de l'architecture logicielle vient du **middleware**.

3.2 Architecture Système

3.2.1 Client / Serveur

- Problématiques à gérer
    - cohérence
    - priorité
    - état

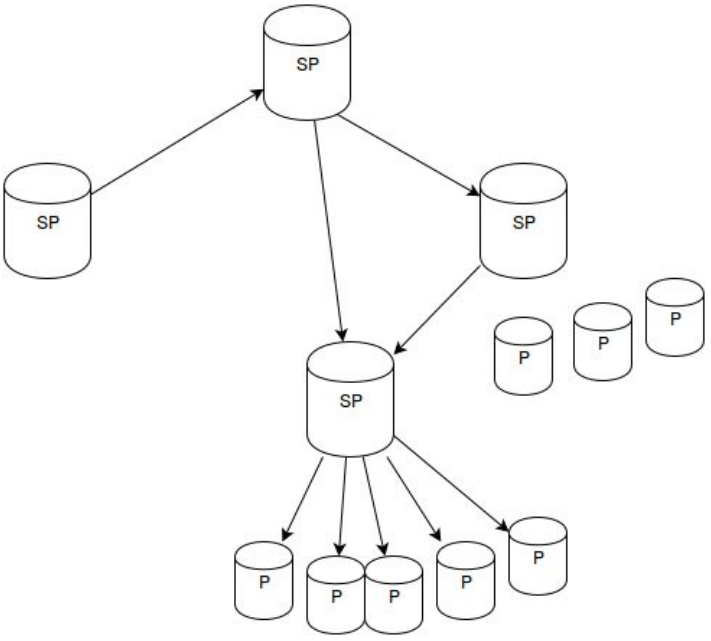
3.2.2 Décentralisée

- Distribution Verticale (1 machine par fonction)
- Distribution Horizontale (toutes les machines ont toutes les fonctions)
    - savoir qui détient quoi?
    - définir les interactions entre les machines
    - Exemple : système peer-to-peer

3.2.3 Système décentralisés non structurés

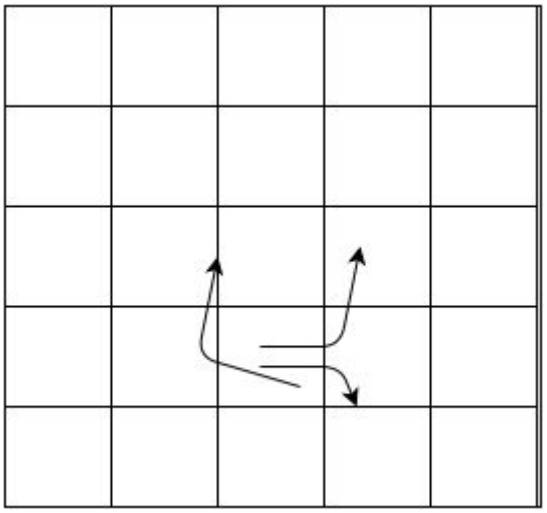
- Systèmes à inondation (avec table de voisinage et TTL)
    - Le TTL induit une couverture partielle de l'architecture (Time To Live).

- Systèmes Super-Peers
    - 2 types de participants : Peers et Super-Peers (sorte de système par grappe)
    - toujours le défaut de couverture partielle
    - gain en efficacité pour la recherche de messages (sous réseau)

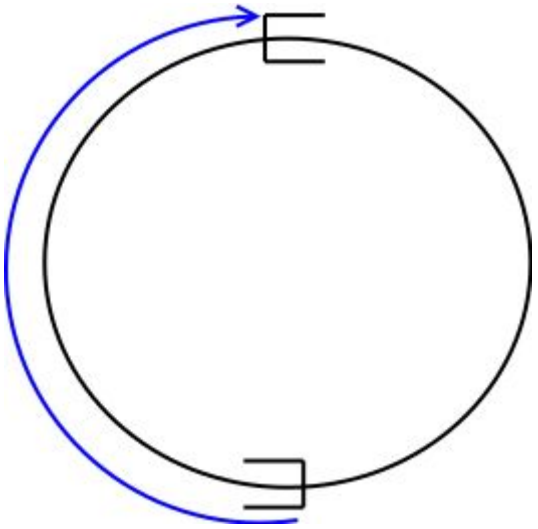


3.2.4 Systèmes Structuré

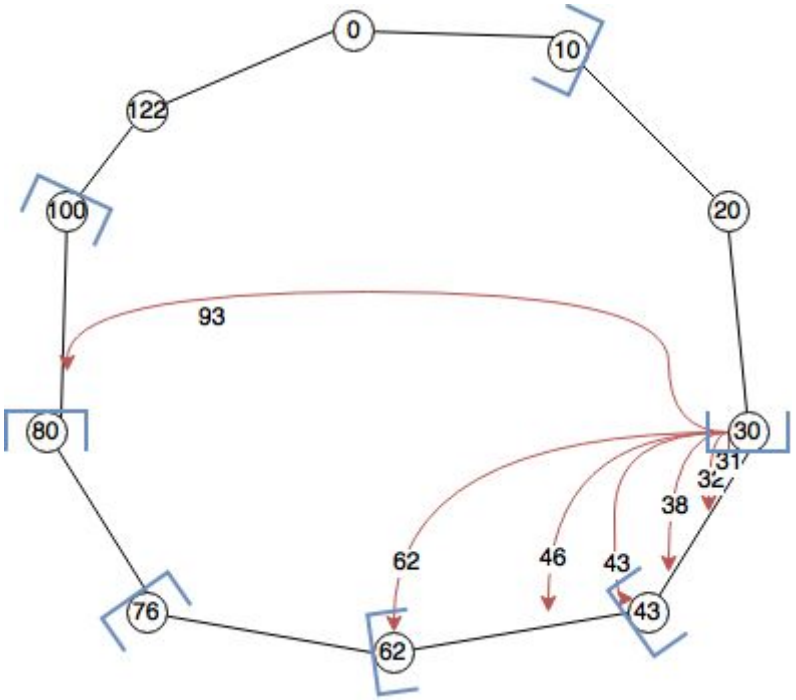
- CAN (routage par contenu en O(√(n)))
    - Bonne performance en recherche
    - Couverture totale (facile de trouver des chemins alternatifs)
    - coût de gestion en communication de routage



- Chord



  - Chaque noeud est responsable des valeurs qui le précèdent jusqu'au prochain noeud. Il faut lire tout le cercle si on cherche une valeur éloignée.
  - Amélioration de l'algorithme avec une table de voisinage avec 2 suivants et 1 précédent, on peut sauter sur le noeud de notre table de voisinage qui est le plus proche de la valeur. On gagne en efficacité. (communication comme le réseau TOR, Oignon routing)



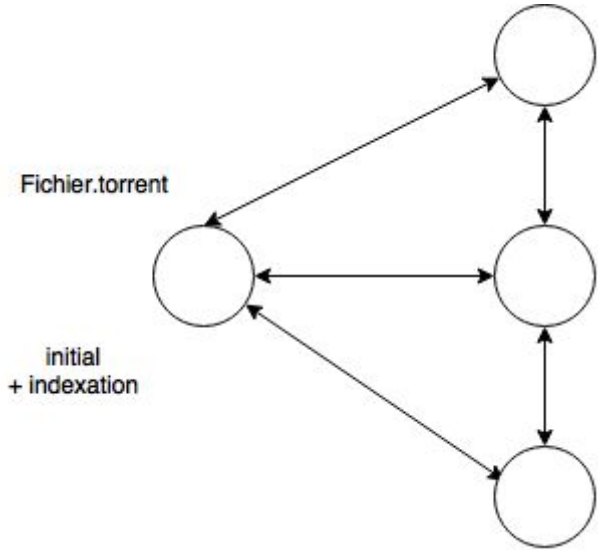
- Insertion d’un noeud
  - Ajout du noeud 100, prévenir noeud 100 et 5
  - 110 + 2mod(i) → ajout respectifs dans la table de voisinage
  - 5 et 110 → mise à jour de leur table de voisinage
  - récupérer les donnés de 101 à 110 → coordonner la récupération avec 5
- Enlever un neud
  - 1/ajouter un noeud
- Réplication
  - k-réplication
    - si 62 plante les données sont répliquées sur 70 car de toutes façons si 62 est pas là c’est 70 qui gère.
  - il y a un surcoût en gestion de cohérence
  - gain en résilience

3.2.5 Architectures Hybrides

en partie décentralisée (fonctionnement du système), en partie client/serveur (pour le démarrage du système)

Exemples :

- Super Peers
- BitTorent



4 Gestion Automatique

Système self*

- self-prédiction
- self-managing
- self-healing

Exemple : si le système décide que pour continuer sa gestion, il faut que son processeur soit au-dessous de 20% de charge, il va alors se diviser en 2 noeuds, ou refuser des clients suivant sa politique.

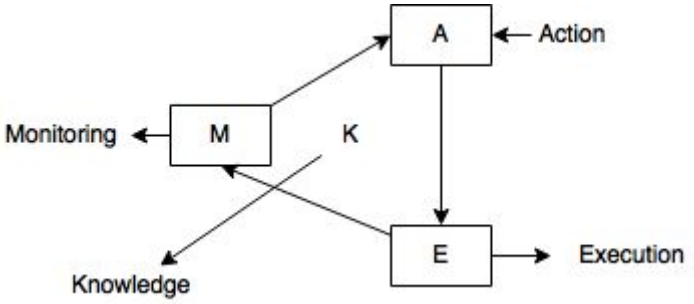


Schéma : **MAPE-K**

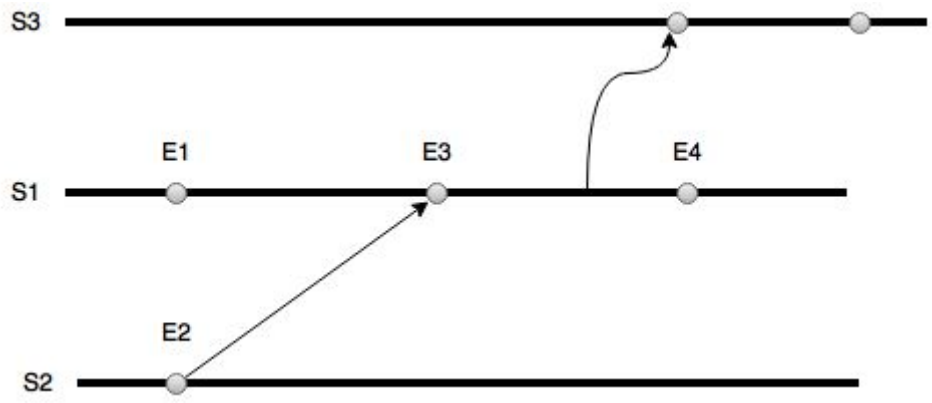
Les systèmes utilisant **MAPE-K** sont des systèmes à agents ou à comportement émergeant.

5 Synchronisation

5.1 Ordre causal

Point de vue local

E2 implique E3 : E2->E3 on sait que E2 est avant E3
mais on ne sais pas si E1 est avant E2
la communication E2 vers E3, il existe E3 tel que E2 -> E3 && E3 -> E4

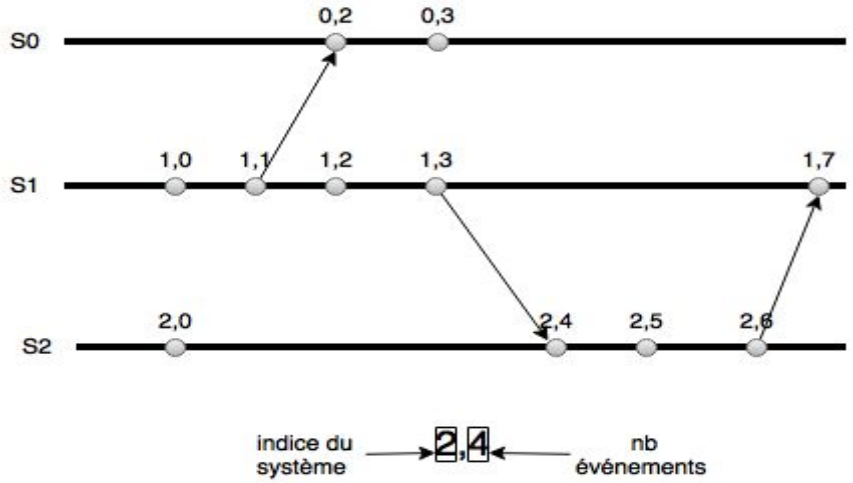


E1 // E2 ⇔ /(E1 -> E2) & /(E2 -> E1)
(la barre devant une parenthèse veut dire : NON)

5.2 Horloge de LAMPORT et Horloge Vectorielle

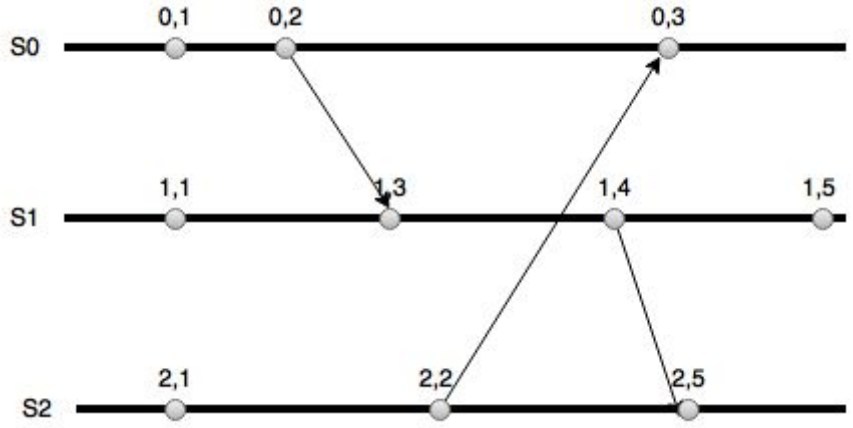
5.2.1 Horloge de LAMPORT

Les horloges sont propres aux systèmes, chaque évènement ou communication fait évoluer l'horloge interne du système local mais en prenant en compte la plus grande horloge des systèmes.
Les évènements qui précèdent ont une horloge plus petite.
Les horloges d'évènement ne sont révélées uniquement lors d'une communication.



Evt1 -> Evt2 L(evt1) < L(evt2)

Exercice corrigé :

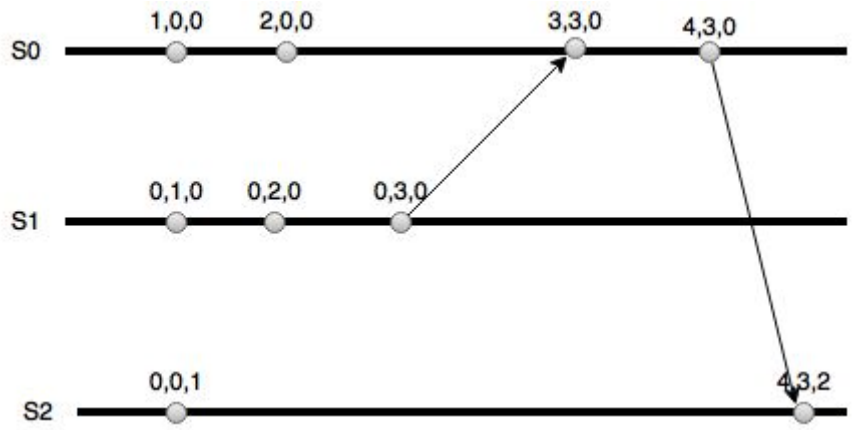


Dans le système de **LAMPORT**, pour savoir qui est arrivé avant :
t1 < t2 méthode 1 : l'un des timming est plus petit que l'autre,
t1 == t2 && s1 < s2 méthode 2 : si les timmings sont identiques, le numéro du système s1 est plus petit que le système s2

Problématique : LAMPORT gère l'ordre total mais ne préserve pas la causalité.

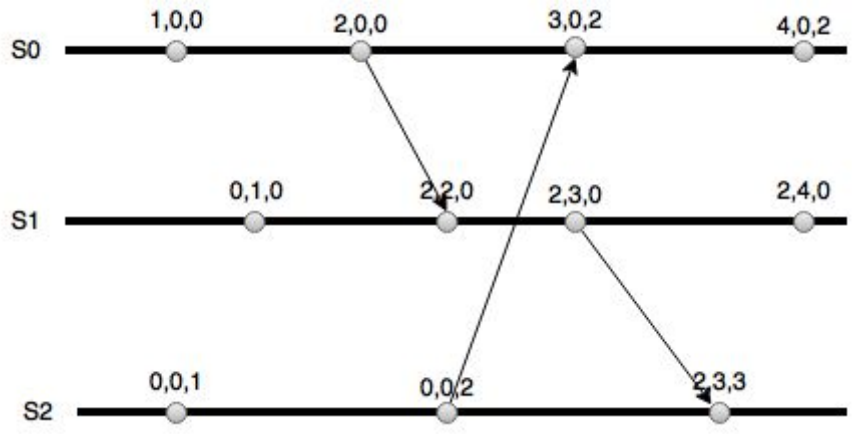
5.2.2 Horloge de MATTERN (horloge vectorielle)

On a les horloges de tous les systèmes dans chacun des systèmes. On fait évoluer les horloges lors de la communication avec la valeur max du tableau des horloges. On peut dire que l'evt1 est avant/après/en parallèle de l'evt2.



M(e1) < M(e2) ⇔ e1 -> e2
M(e1) =< M(e2) ⇔ (a,b,c) =< (a',b',c') ⇔ a=<a', b=<b', c=<c'

Exercice corrigé:



Cas d'utilisation :

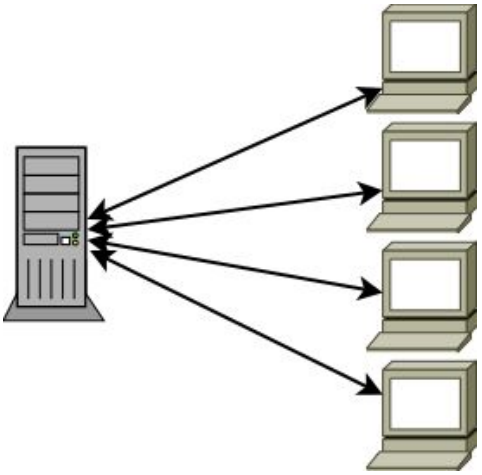
- File d'attente distribuées avec exclusion mutuelle sur la file d'attente
- mémoire virtuelle répartie
- synchronisation d'horloge CPU de machine

Avec MATTERN on a un ordre total et la préservation de la causalité

5.3 Exclusion mutuelle

5.3.1 Méthode centralisée (approche classique)

1 serveur gère l'accès aux ressources

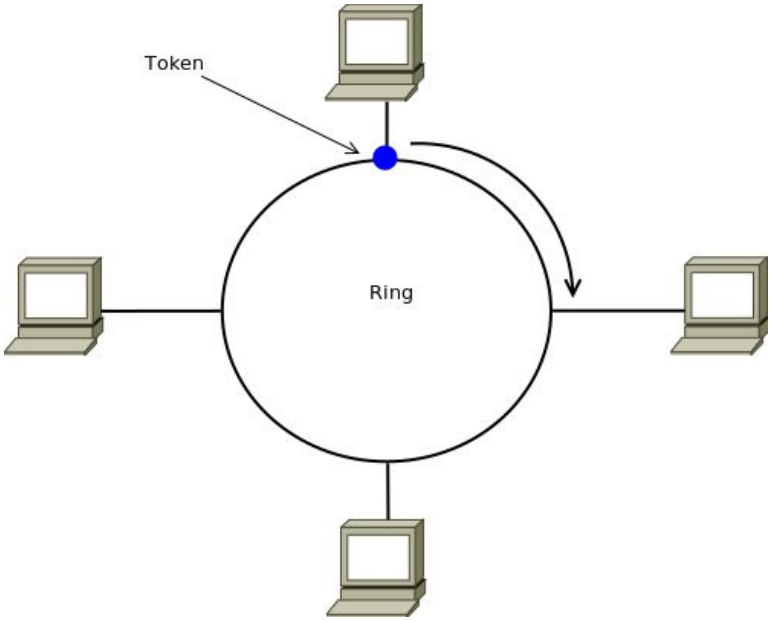


Limitations :

- Charge du serveur
- Single point of faillure
- Latence et problèmes de communication

5.3.2 Méthode à anneau (Méthode à jeton, token-ring)

Un jeton est passé de serveur en serveur et on garde le jeton tant qu'on à quelque chose à faire.

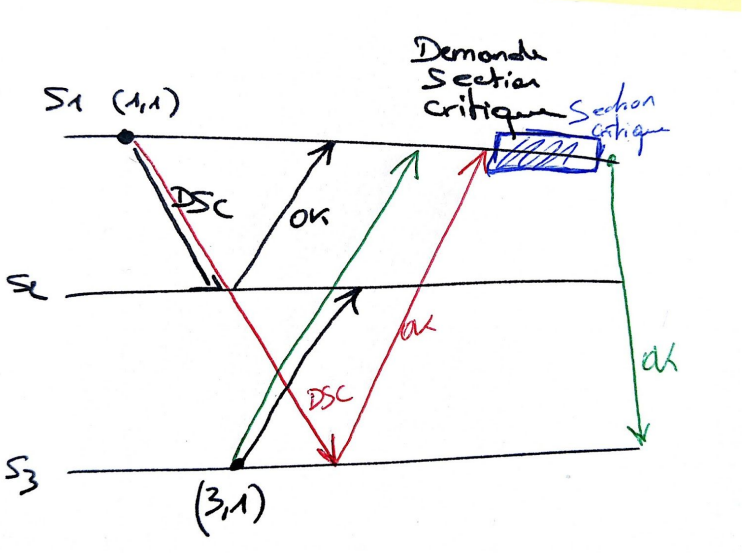


- Avantages :
    - Charge du système répartie
    - Si la charge augmente la performance augmente
    - Garantie sans famine
- Limitations :
    - Communication pour passer le jeton à des serveurs qui n'en n'ont pas besoin
    - perte du jeton (mais le ring est grande il y a moins de chance de crasher le système, mécanisme de gestion des voisins)

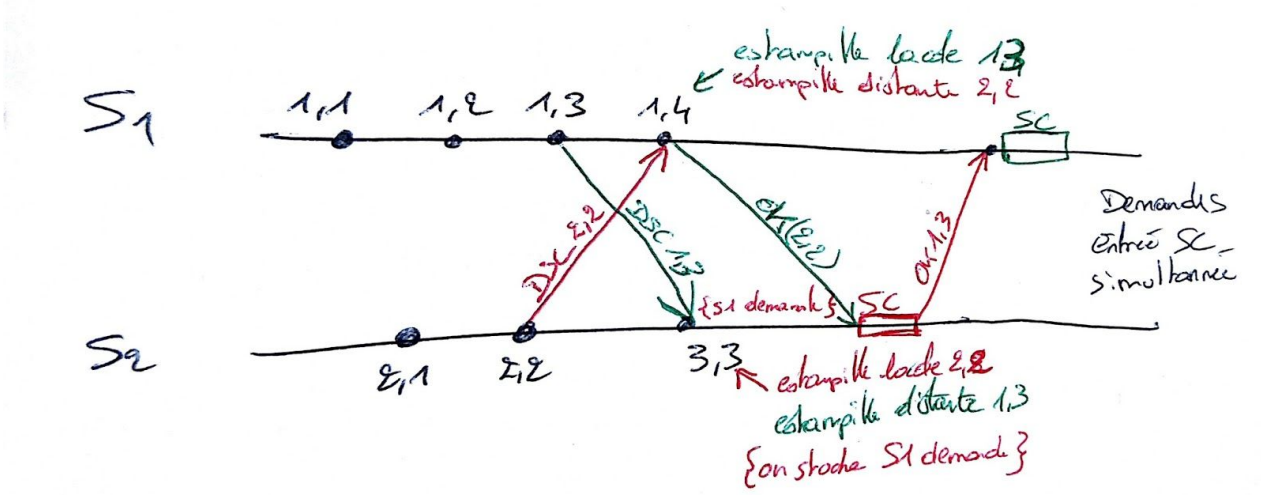
5.3.3 Méthode par estampille (de LAMPORT)

Utilisation d'horloge.

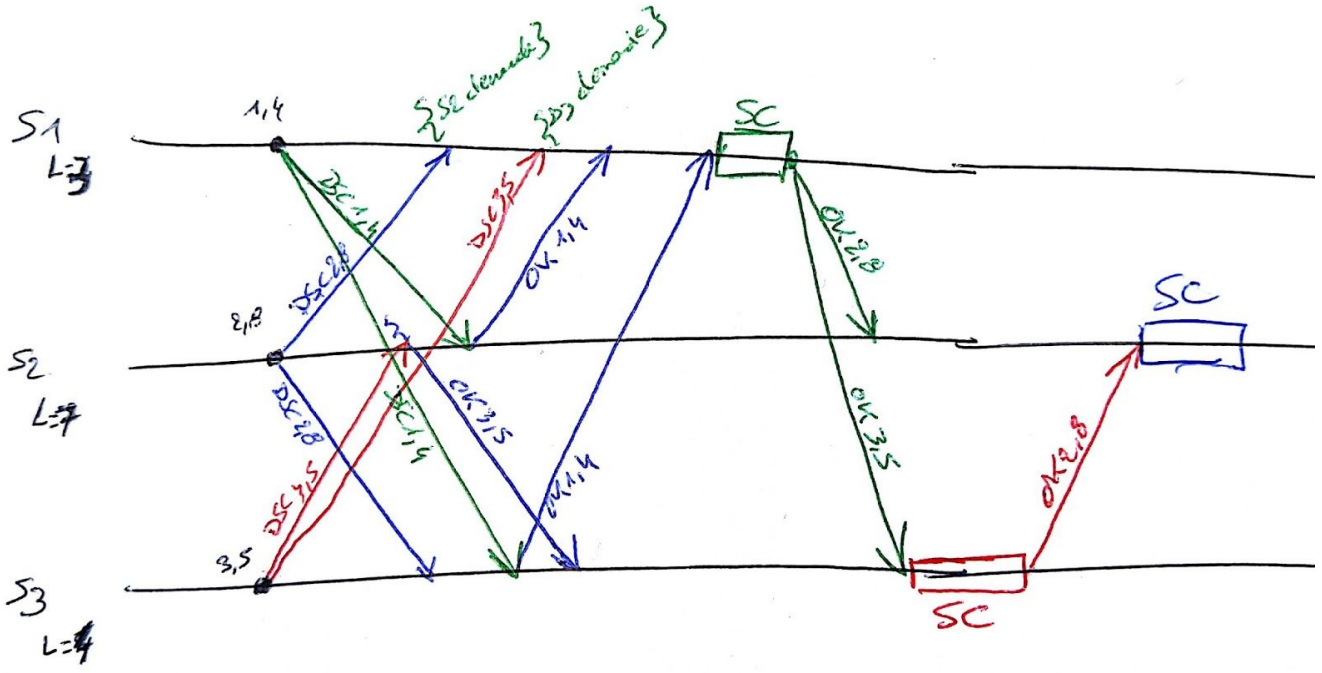
- Entrée en SC (Section Critique)
    - Broadcast avec l'estampille (numéro de machine, Horloge Lamport).
    - Si tout le monde répond OK après comparaison de l'estampille, le demandeur récupère la ressource.
- Réception demande d'entrée en SC
    - 1- pas de demande locale en cours de receveur
        - réponse de OK
    - 2- demande locale
        - estampille plus petite (local est prioritaire) on stocke la demande pour y répondre après
            - Attente de tous les OK des autres receveur
            - Entrée en SC
            - Sortie du SC
            - Réponse OK à la demande reçue
        - estampille plus grande (remote est prioritaire)
            - réponse OK



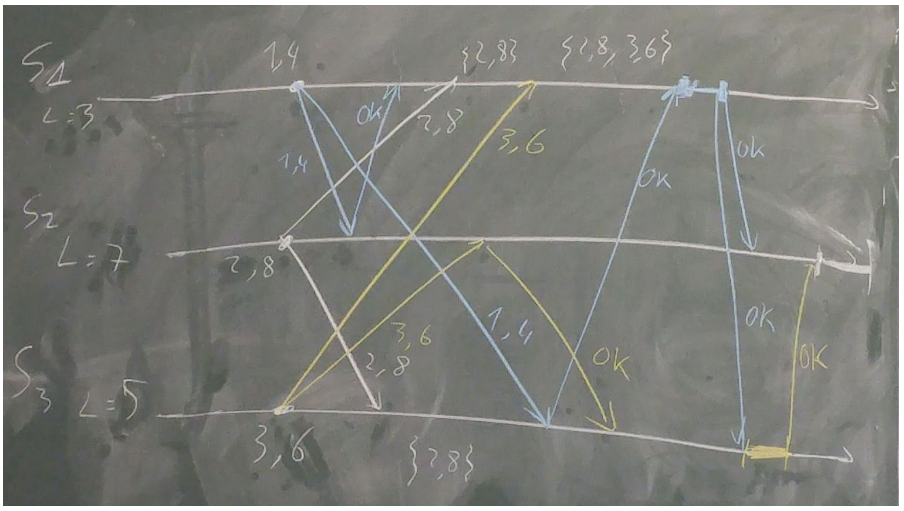
Exemple de demande simultanée d'entrée en Section Critique :



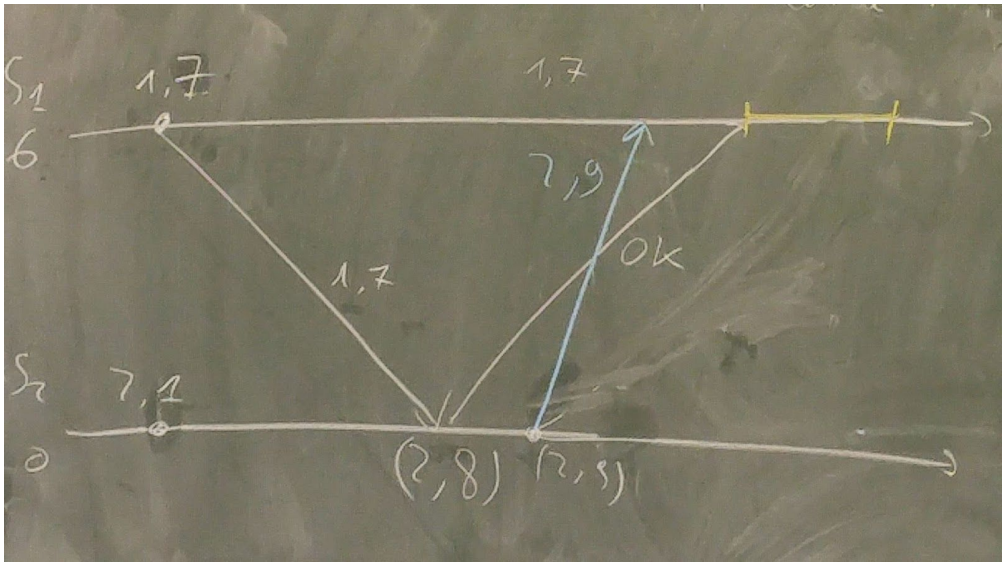
Exercice à 3 demandes simultanées :



Même schéma mais avec utilisation de la même couleur pour chaque requête



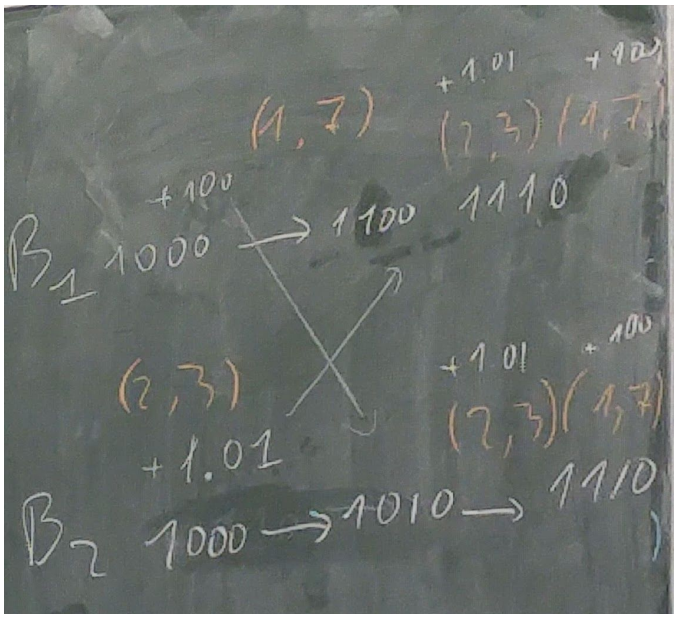
L’horloge de Lamport est augmentée à chaque communication, donc cela permet de rester cohérent même avec une machine qui fait une demande d’entrée en SC en cours de route, puisque sont horloge sera augmentée de 1 lors de la communication.
La garantie de cohérence est due aux règles de causalités



S2 part avec (2,1) mais après communication se retrouve en (2,8) donc sa demande de SC en (2,9) reste inférieure à la demande de S1 en (1,7) donc cohérente.

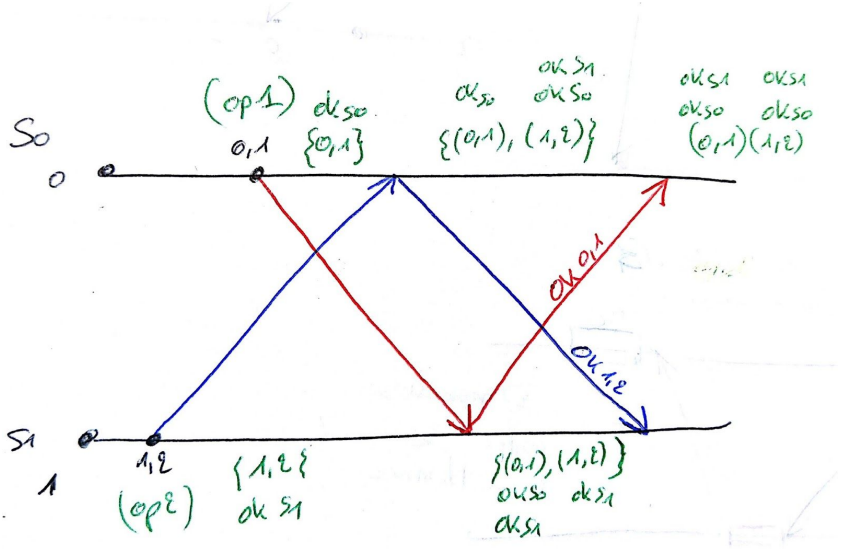
5.4 Files d'attentes distribuées

Exemple de la vie réelle : Les systèmes bancaires
Synchronisation d'opération dans plusieurs banques

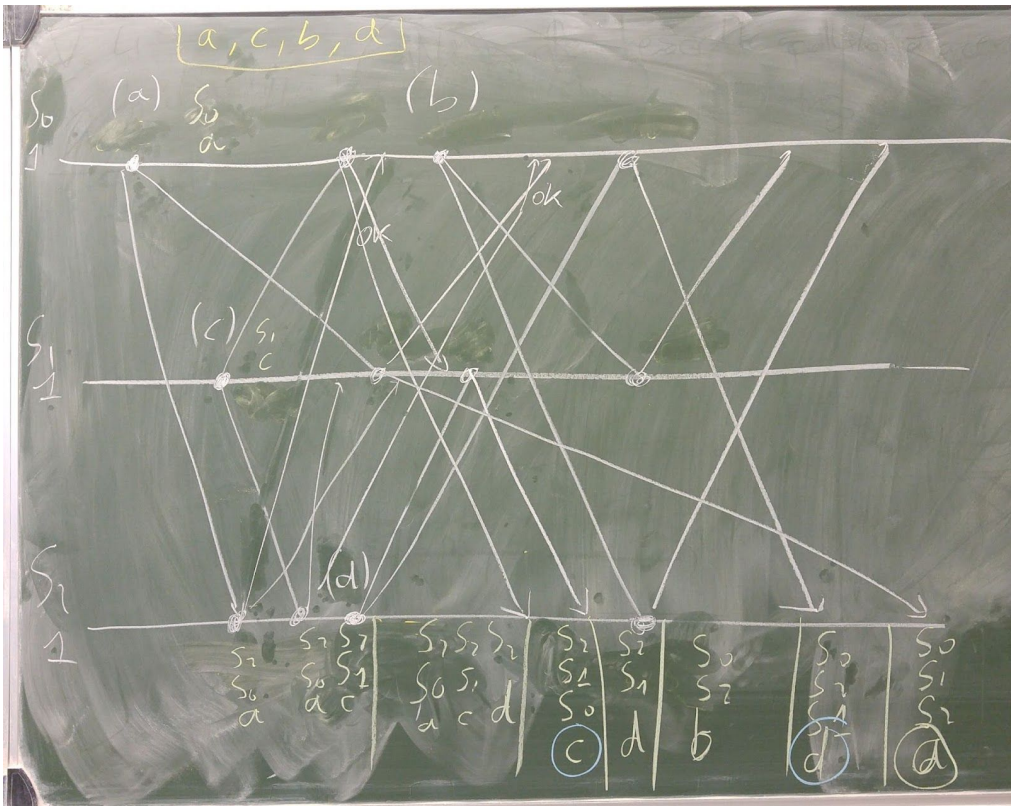


- Ajout d'un élément dans la file
    - Broadcast de l'opération avec l'estampille
- Réception d'un message d'ajout
    - ajout dans file ordonée au sens de Lamport locale
    - Broadcast du OK
- Réception d'un OK
    - Si en Tête et tous les OK reçu => Execution Tête

Les demandes d'ajout sont acquittées au fur et à mesure jusqu'à ce que tous les OK soit reçus.



Exo qui pique à reprendre



6 Ordonnancement

6.1 Définition

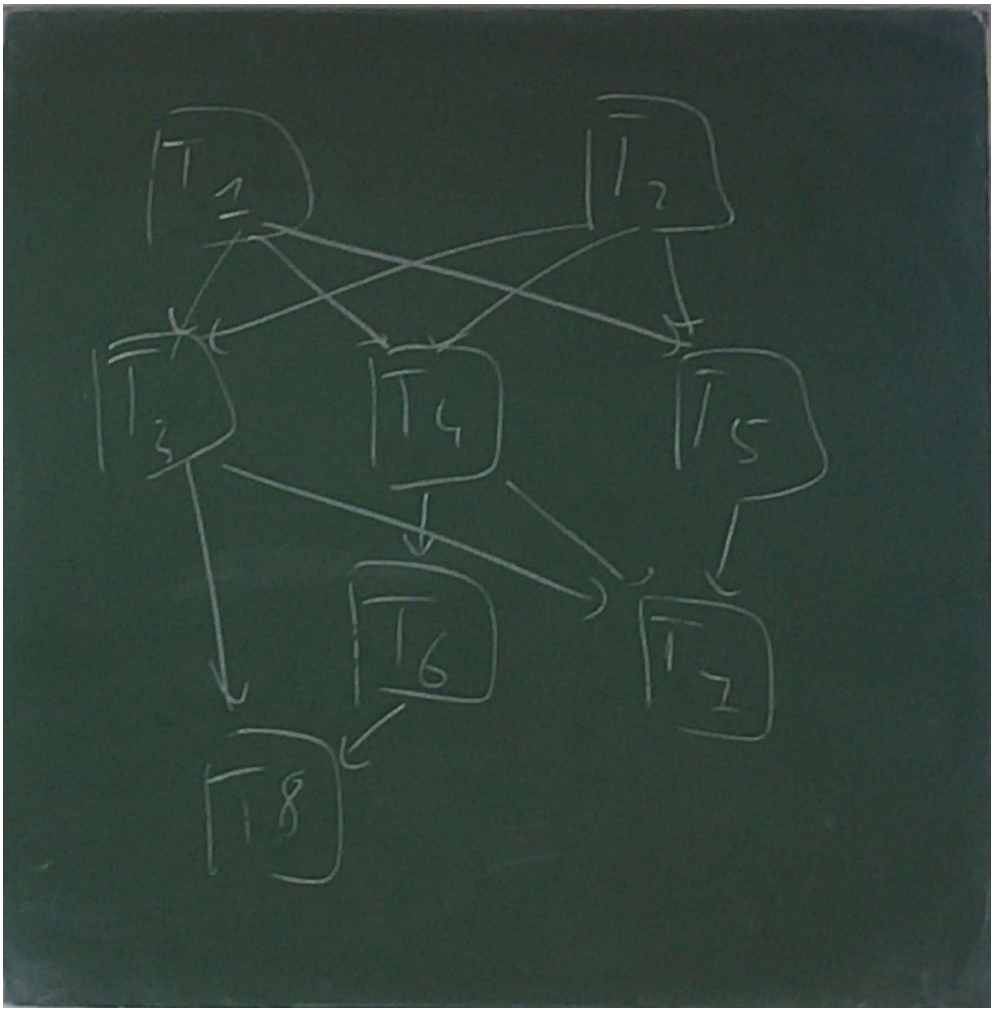
Une tache est une unité indivisible, elle est définie par
T (Et, St, f(t) Et->St, w(t))
Et = Entrée de T
St = Sortie de T
f(t) = Fonction de transformation de E en S
W(t) = fonction Oméga du temps d'execution

Deux taches sont dépendantes SSI
T et T' sont dépendante (Et ∩ Es, !=0) OU (Et' ∩ St' != 0)

6.1.1 Théorème

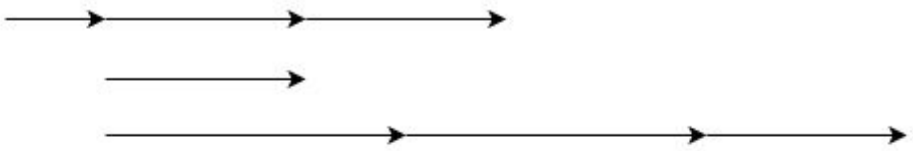
Le temps le plus court pour faire l'ensemble des taches, qui est la longueur du chemin critique, est la longueur du chemin le plus long.

6.1.2 Exemples

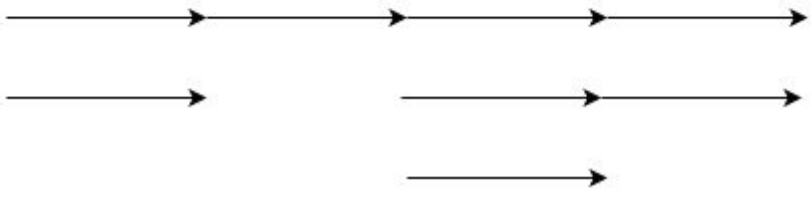


A chaque instant on exécute toutes les taches sans précédent non fini.

AU plus tot



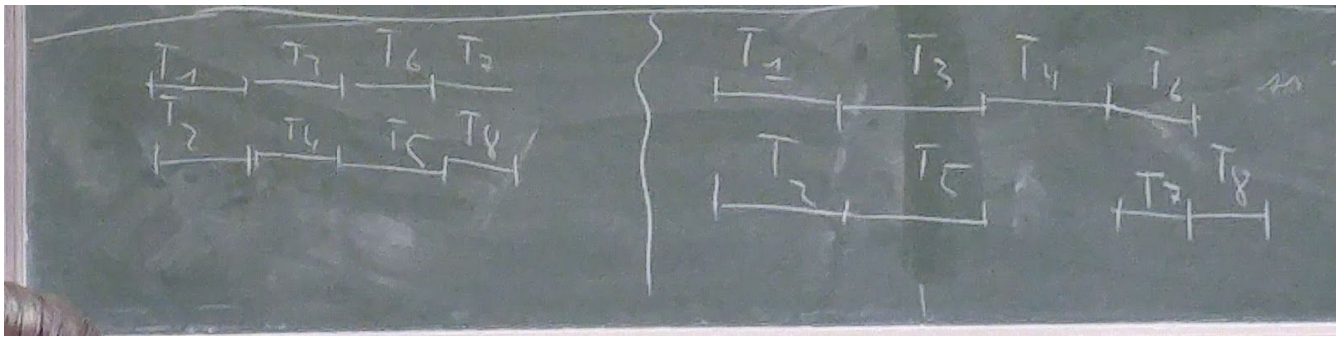
Au plus tard



6.1.3 Ordonnancement de liste

Q = {taches sans prédécesseurs}
tq reste à faire
    si processeur libre
        enlever de Q et execution
    si tache finie
        MAJ Q

enlever de Q = fonction de choix

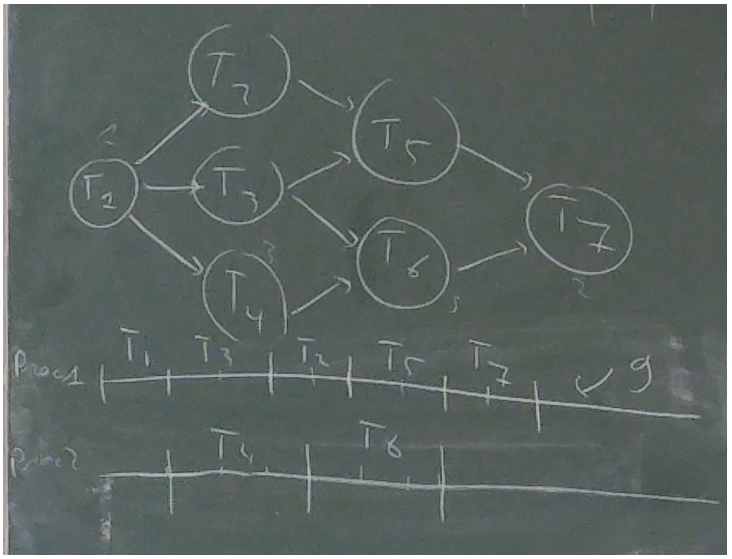


c'est pas optimal

6.2 Allocation

A : T->(proc(t), debut(T))

- si T1 précédente T2 début(T2) >= début(T1) + W(t1)
- si proc (T1) = proc(T2) alors début (T1) >= T(2) + W(t2) ou T(2) + W(t2) =< début (T1)



6.2.1 Fonction de choix

- Optimale avec un coût exponentiel
- Heuristique (fonction déterministe de choix)
    - les plus courtes
    - les plus longues
    - le plus de descendants
    - création d'une fonction dédiée

6.2.2 Contraintes

- proc(T2) = P1
- Début(T2) =< 2

6.2.3 Multi-objectifs

- Temps total : *makespan*
- minimiser le nombre de machines
    - par contrainte (minimiser le temps avec au plus 2 machines)
    - par pondération (O=nb machine + 1/10 temps)
        - valeur de pondération via l'expertise du domaine
        - mais critère random

Exemple :

1. priorité
2. heuristique longueur la plus courte
3. plus petit numéro de machine
4. contrainte : 2 machines max