Algorithmique Répartie

Communication

Université François Rabelais de Tours
Faculté des Sciences et Techniques
Antenne Universitaire de Blois

Master 1 Informatique
Option Systèmes d'Information et Aide à la Décision

Mohamed Taghelit
taghelit@univ-tours.fr

Introduction

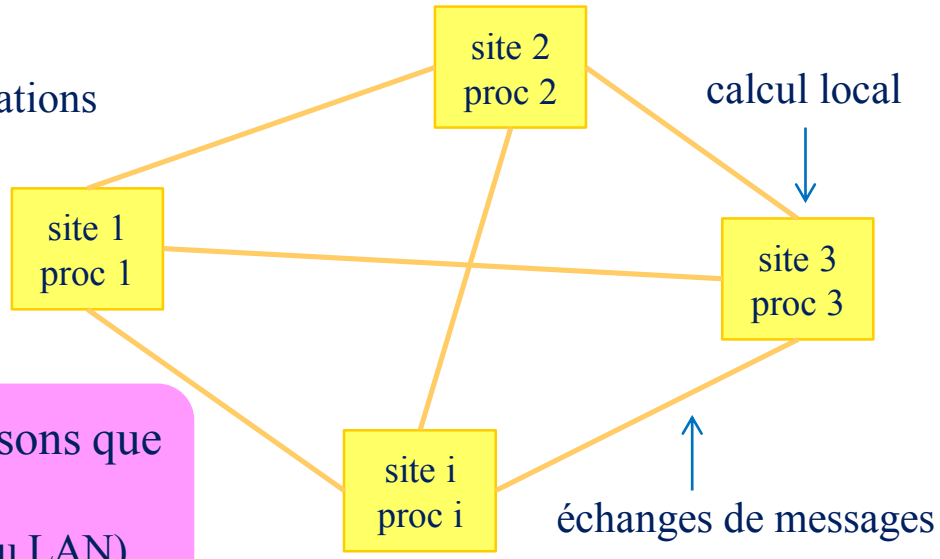
Tendance majeure des systèmes informatiques

- Répartition des traitements entre des "entités" coopératives
  - processeurs d'une machine multiprocesseurs,
  - stations de travail d'un réseau local,
  - serveurs d'application connectés par l'Internet.
- Avantages de la répartition
  - augmentation de la puissance de calcul (par ajout de nouvelles entités),
  - tolérance aux pannes,
  - répartition de charge, migration,
  - découpage logique des applications (comme dans le modèle client-serveur),
  - …
- Difficultés algorithmiques
  - spécifiques à l'activité concurrente des entités (problèmes liés à son application : messagerie, forum, …)
  - absence de mémoire partagée (cohérence des données, détection de la terminaison, l'exclusion mutuelle entre sections de code critiques, …)
- Construire l'application en couches
  - chaque couche s'appuyant sur l'interface de la couche inférieure et fournissant une interface de service à la couche supérieure
  - l'application sera divisée entre une couche applicative (dont les traitements sont spécifiques aux fonctionnalités attendues) et une couche service qui résout des problèmes génériques et récurrents dans le domaine de la répartition)

Modèle de Répartition

L'environnement réparti :

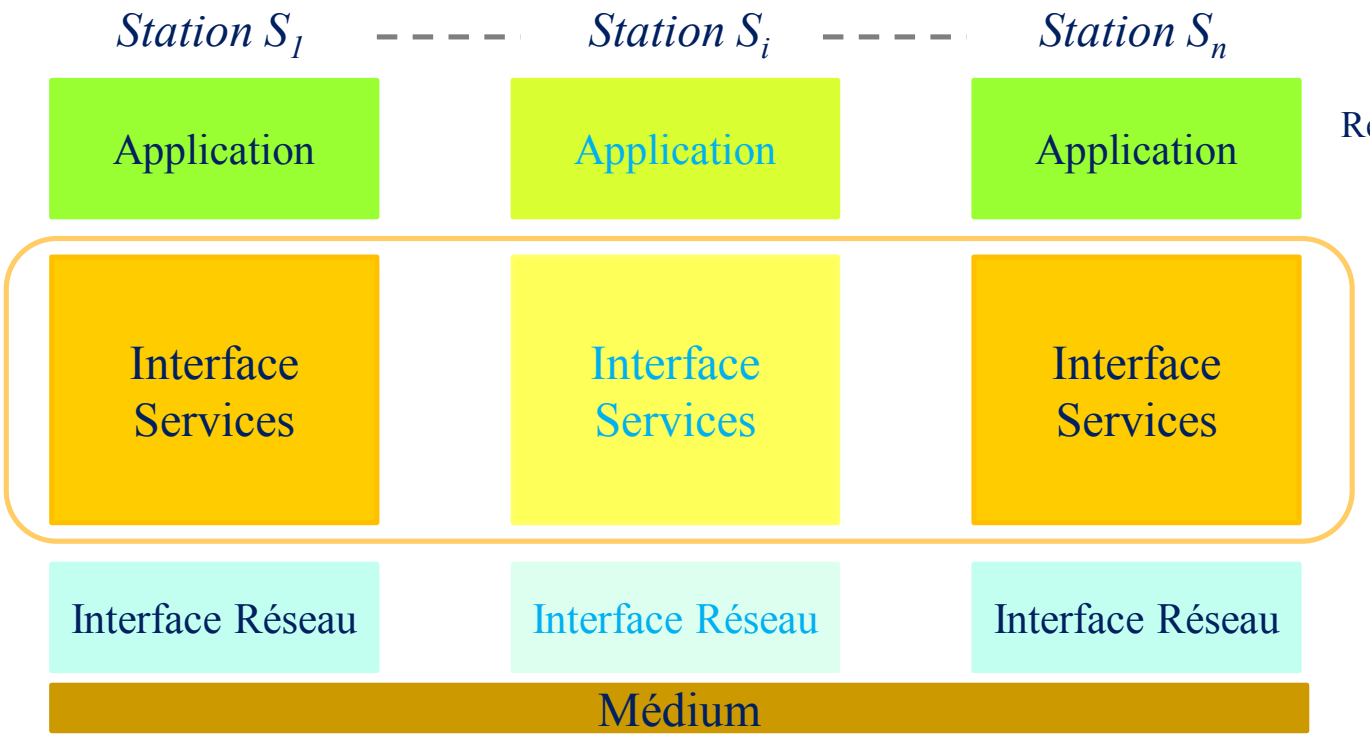
- Entités actives
  - sites, stations
  - mémoire dédiée non accessible aux autres entités
  - un (ou plusieurs) processeurs(s) qui partage son temps entre l'exécution de plusieurs processus
  - une adresse unique et numérique (adresse MAC ou IP)
- Lignes de communication bipoints (reliant deux sites) et bidirectionnelles (transmission d'information dans les 2 directions)
  - une direction = canal
  - seul moyen d'échange d'information entre 2 stations
- Graphe de communication non orienté
  - nœuds → stations
  - arêtes → liaisons



Hypothèse 1 : sauf mention contraire, nous supposons que ce graphe de communication est une clique
  - toute paire de nœuds est reliée par une arête (cas du LAN).

Modèle de Répartition

Nous nous intéressons à chaque site qu'en tant que composant d'une application répartie.



Réalise le traitement spécifique et fait appel à la couche services

Ensemble des codes associés
=
Algorithme Réparti

Qualité de service dépendant de l'environnement

Modèle de Répartition

Le réseau et la couche réseau :

- Un canal de communication reliant un site S_i à un autre site S_j a les propriétés suivantes :
  - les messages ne sont pas altérés,
  - les messages ne sont pas perdus,
  - le canal est FIFO,

- Délai de transit d'un message
  - quelconque (Ethernet avec un médium partagé) → réseau asynchrone
  - borné par un délai maximum connu du concepteur de l'application (Token Ring) → réseau synchrone

Hypothèse 2 : sauf mention contraire, nous supposons que le réseau est asynchrone
- une application conçue pour un environnement asynchrone fonctionne aussi en environnement synchrone
- l'exploitation de la borne nécessite un choix souvent délicat de valeurs temporelles (délai des chiens de garde ou "time out")

Modèle de Répartition

La couche réseau :

- Fournit un mécanisme de transfert de messages pour la couche service :
  - Primitives synchrones : bloquant éventuellement le processus qui les appelle
    - une émission est synchrone si le processus émetteur est bloqué jusqu'à la réception d'un acquittement de son message (acquittement signifiant que l'application distante l'a pris en compte).
    - une réception est synchrone si le processus appelle explicitement une primitive pour recevoir un message et bloque éventuellement en attente de ce message.
    - applications plus simples à vérifier et donc à concevoir.
  - Primitives asynchrones : non bloquantes
    - applications plus flexibles où le concepteur peut choisir lui-même les points d'attente

- Choix d'une communication asynchrone
  - Il est aisé d'émuler des primitives synchrones par des primitives asynchrones,
  - Dans certains cas, l'asynchronisme de communication est plus adapté au problème considéré.

Modèle de Répartition

La couche réseau :

- Primitives d'émission
  - **`envoyer_à(destinataire, message)`**

    Permet d'envoyer un message au destinataire indiqué et de poursuivre son exécution.

    Le destinataire est désigné par son identité et le message peut être structuré selon les besoins de la couche service.

  - **`diffuser(message)`**

    Permet d'envoyer un message à tous les autres sites participant à l'application.

    L'utilisation de cette primitive n'est possible que si le graphe de communication est une clique.

  (appelées dans l'algorithme réparti par la couche service)

- Primitive de réception
  - **`sur_réception_de(émetteur, message)`**

    Spécifie le traitement à effectuer sur réception d'un message.

    L'émetteur est désigné par son identité et le message peut être structuré selon les besoins de la couche service.

  (écrite par la couche service mais appelée par la couche réseau)

La Couche Service

La couche service sera composée de :

- Variables qui lui sont propres

  Inaccessibles à la couche application (sauf peut-être en lecture).

  Une variable **`var`** d'une station **`i`** sera généralement désignée dans l'algorithme par $\mathbf{var}_{\mathbf{i}}$ pour insister sur le caractère local de la mémoire.

- Les primitives descendantes de l'interface application-service.

  Sont en général la transcription de la fonctionnalité recherchée.

- Des primitives internes

  Utilisées pour factoriser des traitements communs à plusieurs primitives.

- Occasionnellement, un processus interne au service lorsqu'une tâche devra être exécutée de manière concurrente au fonctionnement de l'application (par souci d'efficacité par exemple).

- Les primitives montantes **`sur_réception_de()`** de l'interface service-réseau.

La Couche Service

- Langage de programmation

  Pseudo-C, mais tout autre langage peut également être utilisé.

- Quelques extensions
  - Un processus d'application ou de service peut être suspendu

    Attente passive, cad allocation du processeur à un autre processus, jusqu'à ce qu'une condition soit remplie (elle peut l'être immédiatement).

    Primitive utilisée : **`Attendre(Expression booléenne)`**.

  - La suspension peut également être due à l'attente de l'expiration d'un `time-out`

    Primitives utilisées : **`Attendre()`** et **`Armer(délai)`**.

  - Extension des expressions booléennes avec les quantificateurs ∀ et ∃

    Utilisés principalement lorsque l'indice du quantificateur portera sur un ensemble de sites ou de messages.

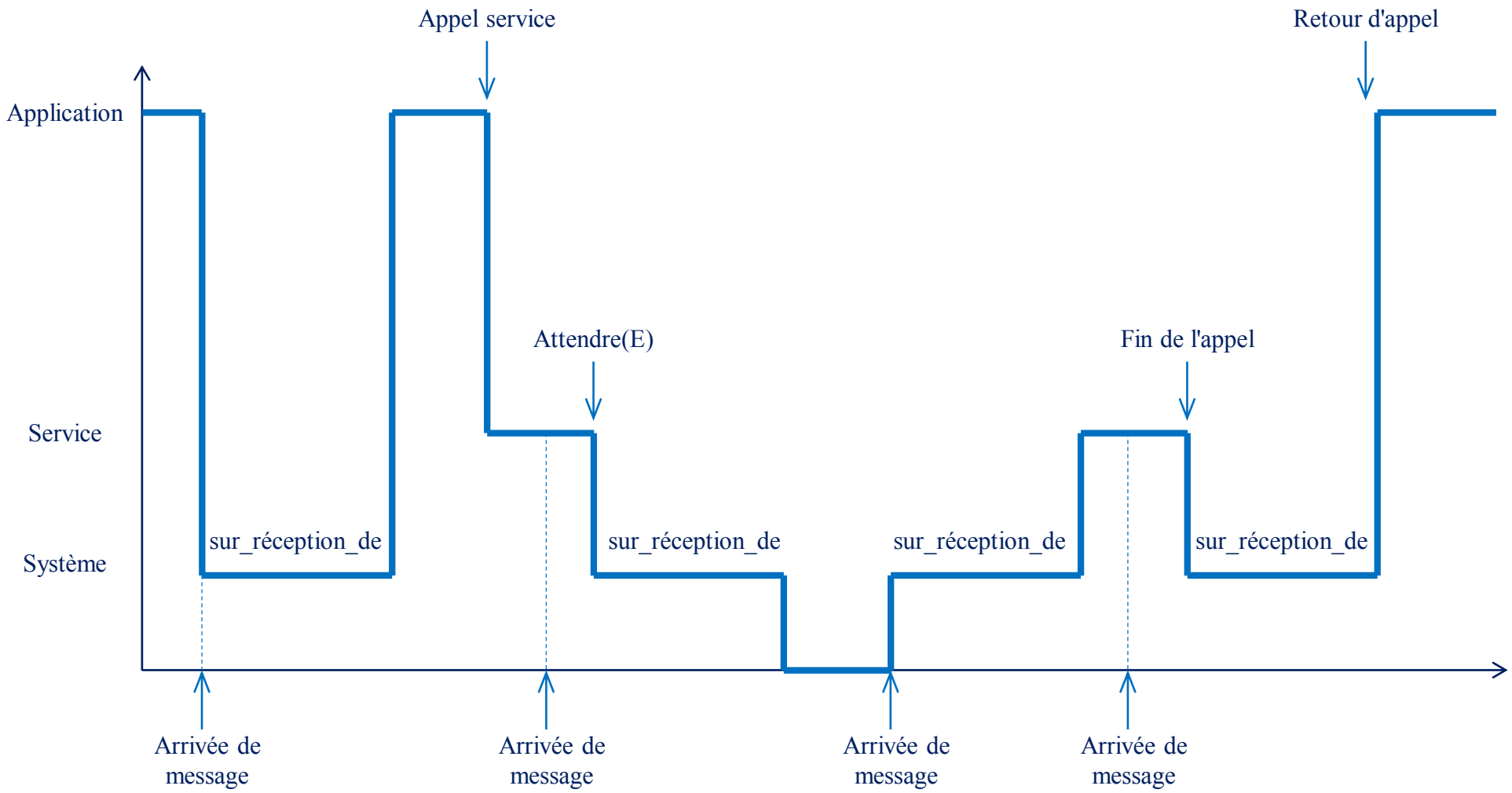
La Couche Application

Hypothèse 3 : sauf mention contraire, nous supposons que l'application est exécutée sur un site par un unique processus
- simplification de la gestion du service
- si nécessaire, on peut généraliser la plupart des algorithmes au cas où l'application est exécutée par plusieurs processus (dont le nombre est connu) en dupliquant la couche service

Interaction entre les différentes couches

- **Règle 1** : Si le processus applicatif exécute du code de la couche application, toute réception d'un message donne lieu à un déroutement et à l'exécution de la primitive `sur_réception_de` correspondante. Le traitement applicatif se poursuit ensuite.
- **Règle 2** : Si le processus applicatif (ou de service) exécute du code de la couche service, l'exécution de la primitive `sur_reception_de` suite à une réception de message est retardée jusqu'au blocage du processus (par la primitive `Attendre`) ou jusqu'au retour en couche application.
- **Règle 3** : Le code des primitives `sur_reception_de` ne comporte pas d'appel à la primitive `Attendre`. En effet ce code est exécuté par le système en interruption et ne correspond à aucun processus particulier.
- **Règle 4** : Si le système exécute une primitive `sur_reception_de` suite à une réception de message, toute exécution de `sur_reception_de` suite à une autre réception de message est retardée jusqu'à la fin de l'exécution de la première primitive.

Déroulement d'une Application sur un Site



La Communication

- Contrôle de flux
  - Producteur/Consommateur
  - Producteurs multiples/Consommateur
  - Dans les réseaux étendus
- Communication synchrone
  - Le rendez-vous
- Qualité de service
  - Réseau FIFO
  - Émulation d'un réseau FIFO

Contrôle de Flux

Contrôle point à point – le modèle producteur/consommateur

- Description du modèle
  - Deux sites **`Prod`** (le producteur) et **`Cons`** (le consommateur)
  - Le producteur envoie des messages au consommateur via un canal unidirectionnel
  - Lorsque l'application du producteur veut envoyer un message au consommateur elle appelle la primitive **`produire(m)`**
  - Lorsque l'application du consommateur veut recevoir un message du producteur, elle appelle la primitive **`consommer(m)`**

- Description du problème
  - La vitesse d'exécution de **`Prod`** n'est assujettie à aucune contrainte
    - le rythme de production et d'émission de messages par **`Prod`** peut être supérieur au rythme avec lequel **`Cons`** les consomme
    - Quelque soit le nombre d'emplacements prévus pour stocker les messages reçus et non encore consommés, ceux-ci peuvent se trouver occupés alors qu'un nouveau message arrive
    - Soit le nouveau message est rejeté, soit l'un des messages stockés est écrasé par le nouvel arrivant

Contrôle de Flux

Contrôle point à point – le modèle producteur/consommateur

- Description de la solution
  - Problème du à l'asynchronisme des sites **`Prod`** et **`Cons`**
  - Une solution consiste à asservir la production et l'émission des messages par le site **`Prod`** à des informations de consommation fournies par **`Cons`**
  - D'un côté, le producteur disposera d'autorisations de produire (initialement égales à la taille du tampon du consommateur)

    A chaque envoi, le producteur consomme une autorisation
  - De l'autre côté, le consommateur envoie une autorisation à la fin de chaque appel à **`consommer`** (signifiant qu'une place s'est libérée dans le tampon)

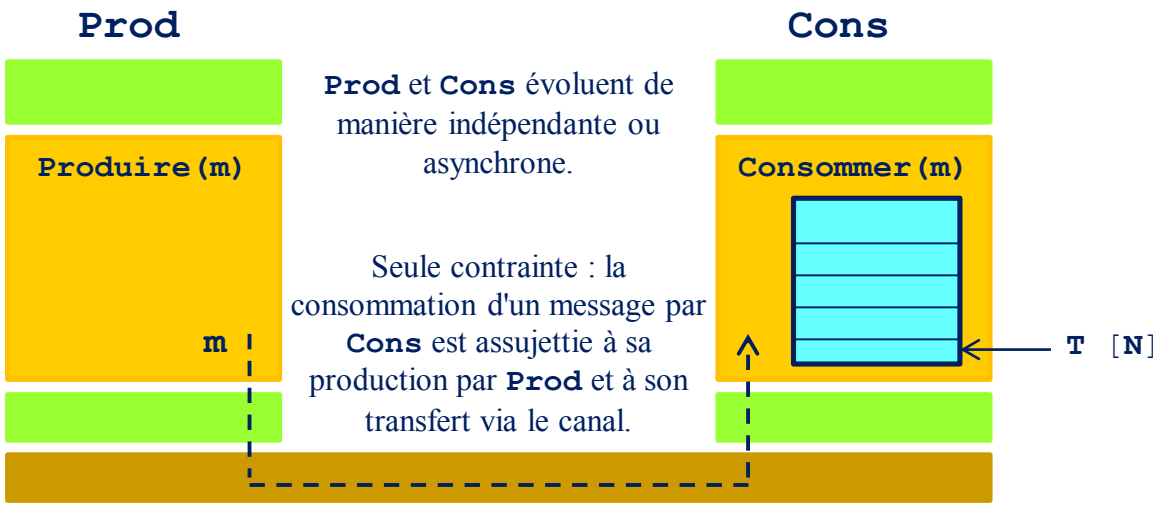
    Le consommateur ne peut consommer que si son tampon n'est pas vide

Producteur - Consommateur

Algorithme

- Variables de contrôle de chacun des sites
  - **`Nbmess`** : variable du consommateur indiquant le nombre de messages (non encore consommés) dans le tampon
  - **`Nbcell`** : variable du producteur indiquant le nombre d'autorisations
- Variables du consommateur pour la gestion du tampon
  - **`T`** : tampon de taille **`N`** contenant les messages à consommer
  - **`in`** : indice d'insertion dans le tampon
  - **`out`** : indice d'extraction du tampon

Le tampon **`T`** est géré de manière circulaire : lorsqu'un indice est en fin de tableau, il est remis à zéro.



Contrôle de Flux

Algorithme du producteur

```
Var Nbcell : entier initialisé à N;

Produire(m){
    Attendre(Nbcell > 0);
    envoyer_à(Cons, m);
    Nbcell = Nbcell - 1;
}

sur_réception_de(Cons, Ack){
    Nbcell = Nbcell + 1;
}
```

Remarque : lorsque la valeur de l'indice **in** est égale à celle de **out**, ceci signifie soit que le tampon est plein soit qu'il est vide. D'où la nécessité d'introduire la variable **Nbmess** pour tester l'existence d'un message dans le tampon.

Algorithme du consommateur

Var **Nbmess** : entier initialisé à **0**;

T[0..N-1] : tableau contenant les messages reçus du site producteur.

in, out : **0..N-1**, indices respectivement d'insertion et d'extraction dans le tableau **T**, initialisés à **0**.

```
consommer(m){
    Attendre(Nbmess > 0);
    m = T[out];
    out = (out+1)%N;
    Nbmess = Nbmess - 1;
    envoyer_à(Prod, Ack);
}

sur_réception_de(Prod, m){
    T[in] = m;
    in = (in + 1)%N;
    Nbmess = Nbmess + 1;
}
```

Vérification de l'Algorithme

Preuve de non débordement du tampon:

- Démontrer que pour tout état accessible

  `Nbmess + Nbcell + Nbt = N` // `Nbt` = nombre de messages en transit
  // (application ou acquittement)

  Ce qui garantit qu'un message d'application arrivant ne trouve jamais le tampon plein.

- Démonstration par induction

  L'égalité est vérifiée initialement

  `N + 0 + 0 = N`

  Supposons la vraie dans un état donné et examinons l'effet de chacune des primitives

  Après appel à **`produire`** :

  `Nbcell` est décrémenté et `Nbt` est incrémenté // l'équation reste vraie

  Après appel à **`consommer`** :

  `Nbmess` est décrémenté et `Nbt` est incrémenté // l'équation reste vraie

  Après une réception d'un message :

  `Nbmess` est incrémenté et `Nbt` est décrémenté // l'équation reste vraie

  Après une réception d'un acquittement :

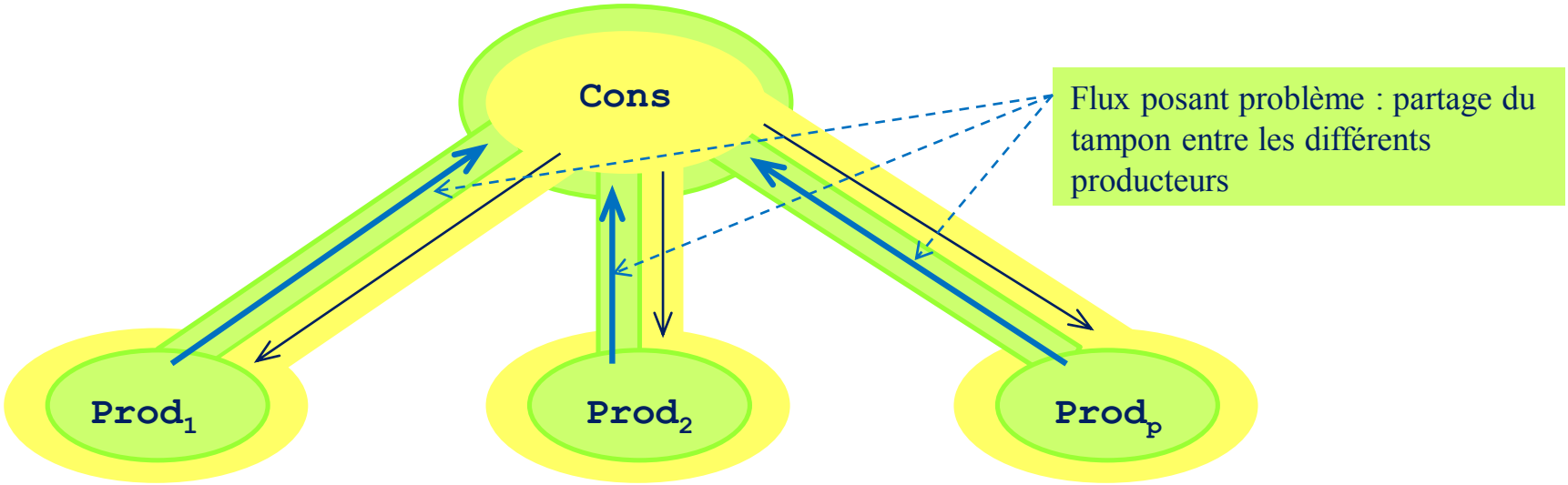
  `Nbcell` est incrémenté et `Nbt` est décrémenté // l'équation reste vraie

- Quelque soit l'évolution de l'algorithme, l'équation est vraie.

Généralisation aux Producteurs Multiples

Généralisation du modèle producteur-consommateur :

- p sites de production $Prod_1$, $Prod_2$, …, $Prod_p$
- un seul site consommateur disposant d'un tampon de stockage de N cellules



- Les interfaces qui définissent le service sont inchangés : `produire(m)` et `consommer(m)`
- Application de ce schéma : gestion des requêtes dans un modèle client-serveur (les producteurs sont des clients et le consommateur est le serveur).

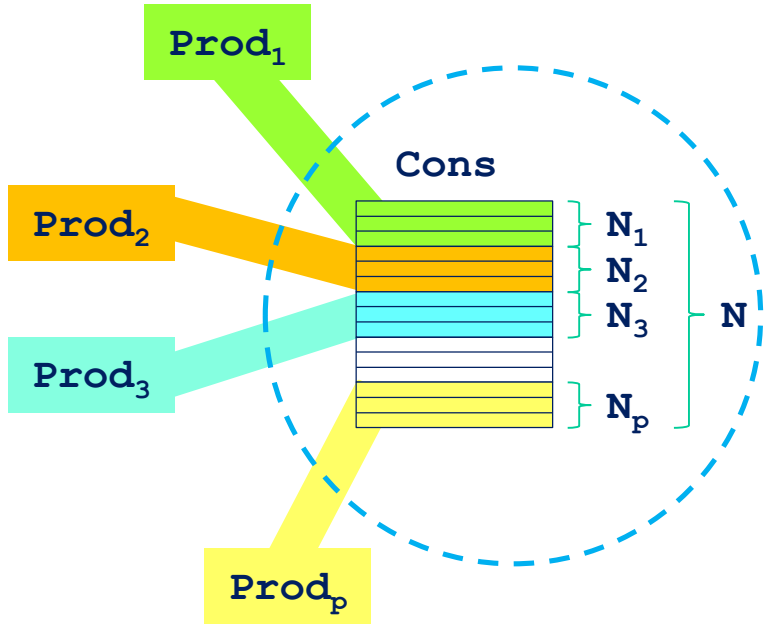
Solution par Partage Statique

Solution la plus simple :

- Partager a priori (d'où le caractère statique de la solution) les **N** cellules entre les **p** producteurs
  Attribuer N_i cellules au site $Prod_i$, avec $\Sigma N_i = N$ (ce qui implique que $N \geq p$)
- Plus de compétition entre les producteurs
  Les producteurs ne partagent plus de ressources

Remarques

- Solution consistant à appliquer la solution précédente pour p canaux de communication moyennant un multiplexage des messages lors de la consommation
- Chaque site producteur ne communique qu'avec le consommateur
  Au niveau du contrôle, le réseau logique est une étoile dont le site Cons est le centre
- Sous-utilisation des ressources
  Situation où seuls q (<<p) producteurs désirent produire
  Inconvénient majeur lié à toute solution statique



Distribution d'Autorisation sur un Anneau Logique

Solution avec établissement d'un anneau logique:

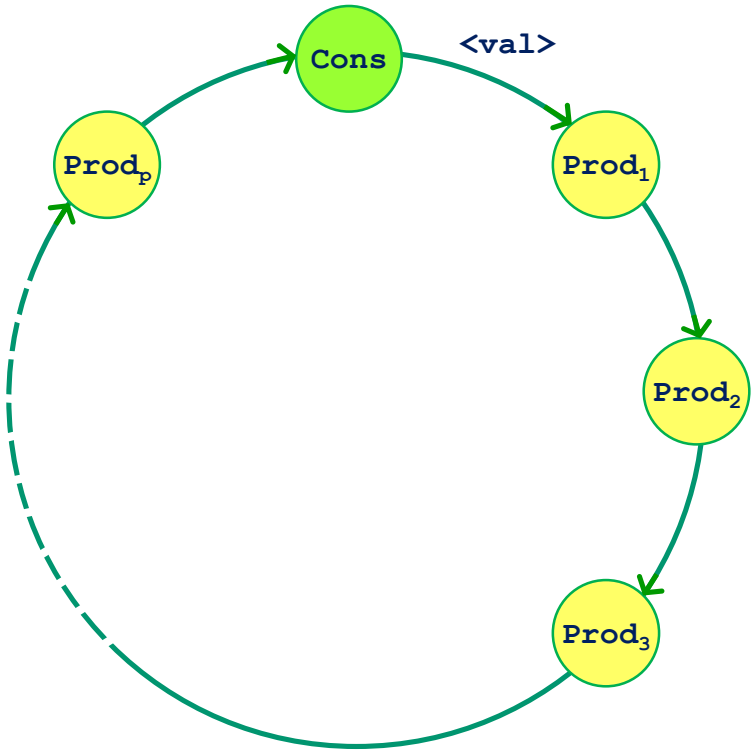
- Faire circuler des autorisations (correspondant à des cellules disponibles) sur un anneau logique
- L'anneau logique est constitué de la façon suivante :

  `Cons, Prod`$_1$`, Prod`$_2$`, …, Prod`$_p$`, Cons`
- Association d'un jeton (message de contrôle spécial)

  Le jeton parcours l'anneau de manière unidirectionnelle

  Il visite donc tous les sites et ceci de façon répétitive

  Au jeton est associé une valeur `val` indiquant le nombre de cellules disponibles



Remarques

- De quel nombre d'autorisations a besoin un producteur lorsque passe le jeton ?

  1 seule si on fait une généralisation immédiate de la solution précédente

  L'appel à `produire` est bloqué jusqu'au passage du jeton

  Solution inefficace car le débit maximum de productions est d'un message par passage de jeton
- Désynchroniser la production de messages du passage du jeton

Désynchronisation de la production de messages de leur envoi

Mise en œuvre

– Ajout d'un tampon local au producteur

L'application dépose les messages dans le tampon local et ne bloque que lorsque ce dernier est plein

– Ajout d'un processus de service : `facteur`

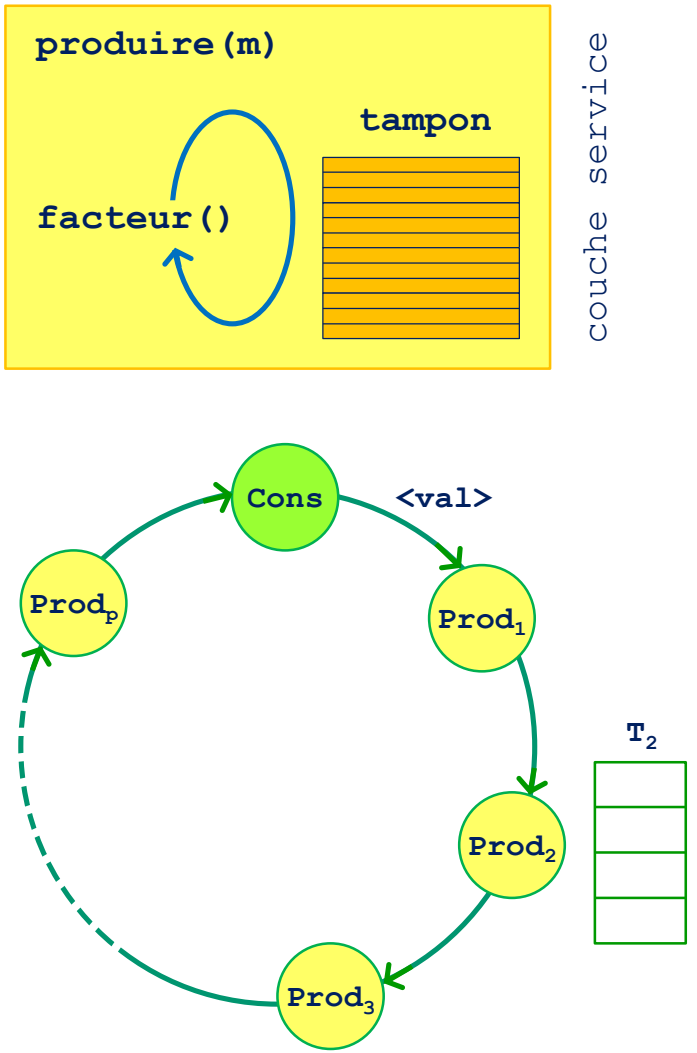
Il envoie les messages du tampon local à destination du consommateur

Se sert de deux variables indiquant :

- le nombre de messages dans le tampon local
- le nombre d'autorisations

Tant qu'il a une autorisation, il envoie un message

Lorsque le jeton arrive, combien d'autorisations le producteur cherche-t-il à obtenir ?



Algorithme du Producteur - Anneau

Variable locales du producteur **$Prod_i$**:

- **$T_i[0..N_i-1]$** : tableau contenant les messages produits par le producteur $Prod_i$.
- **in_i** : indice d'insertion du tampon T_i, initialisé à 0.
- **out_i** : indice d'extraction du tampon T_i, initialisé à 0.
- **$Nbmess_i$** : nombre de messages (non encore envoyés) stockés dans T_i, initialisé à 0.
- **$Nbaut_i$** : nombre d'autorisations d'envoi de messages, initialisé à 0.
- **$Succ_i$** : identificateur du site successeur du site i.
  si $i < p$ alors $Succ_i = Prod_{i+1}$
  sinon $Succ_i$ = identificateur du site consommateur.
- **$Temp_i$** : variable de calcul temporaire. Elle prend la valeur minimale entre le nombre de cellules que le site désire réserver et le nombre de cellules libres associé au jeton.

```
sur_réception_de(j,(jeton,val)){
    temp_i = Min((Nbmess_i-Nbaut_i),val);
    Nbaut_i += temp_i;
    val -= temp_i;
    envoyer_à(Succ_i,(jeton,val));
}
```

```
produire(m){
    Attendre(Nbmess_i < N_i);
    T_i[in_i] = m;
    in_i = (in_i + 1)%N_i;
    Nbmess_i++;
}
```

```
facteur_i(){
Tant que (vrai)
    Attendre(Nbaut_i > 0)
    envoyer_à(Cons, (app, T_i [out_i]));
    out_i = (out_i+1)%N_i;
    Nbaut_i--;
    Nbmess_i--;
Fin tant que
}
```

Remarque :
d'après la gestion de l'algorithme, on aura à tout instant $Nbmess_i \geq Nbaut_i$.

Algorithme du Consommateur- Anneau

Variable locales du consommateur **Cons**:

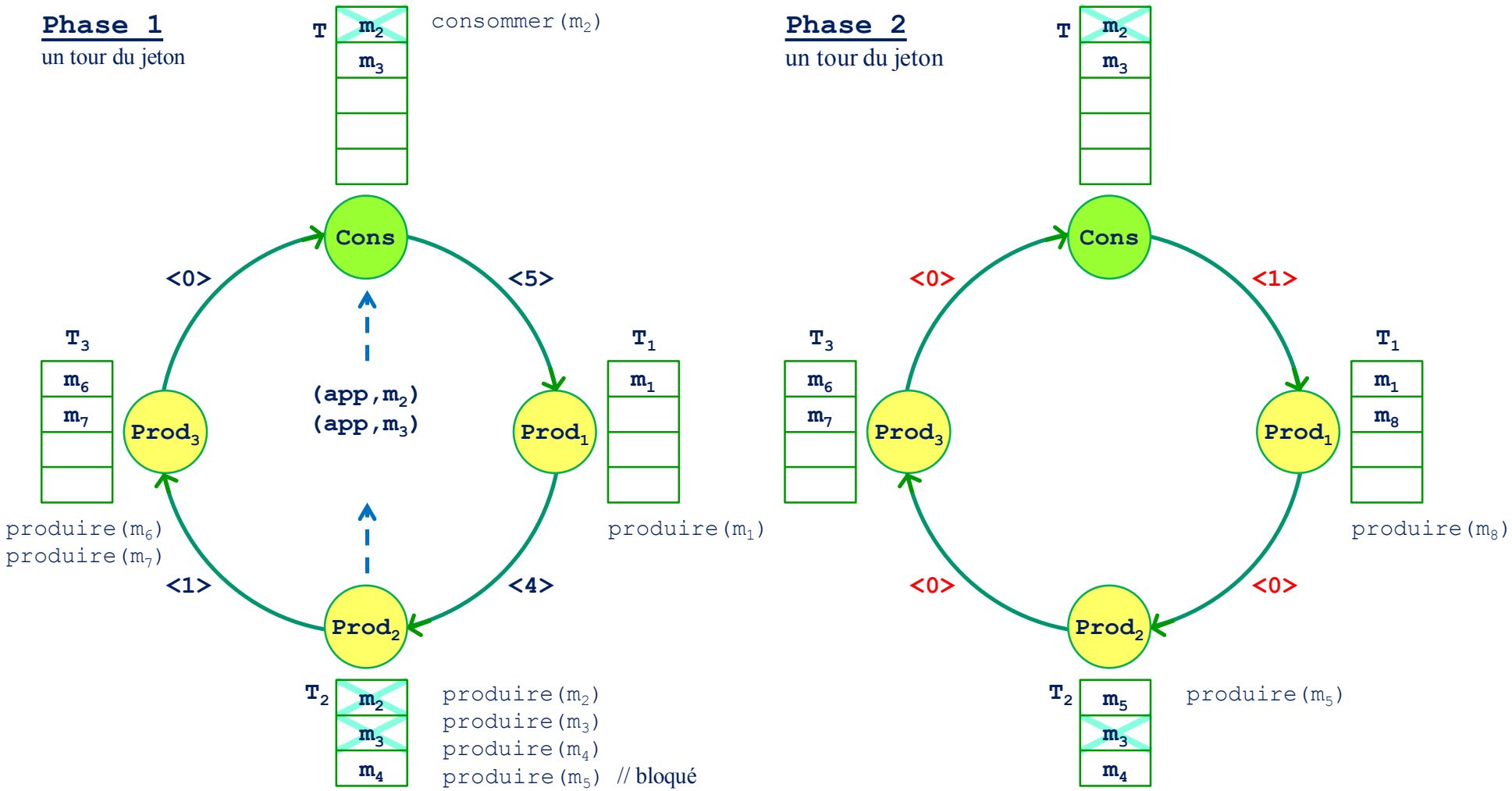
- **T[0..N-1]** : tableau contenant les messages reçus des sites producteurs (tampon du consommateur).
- **in** : indice d'insertion du tampon T, initialisé à 0.
- **out** : indice d'extraction du tampon T, initialisé à 0.
- **Nbmess** : nombre de messages (non encore consommés) stockés dans T, initialisé à 0.
- **Nbcell** : nombre de cellules libérées entre deux passages du jeton, initialisé à 0.

```
consommer(m){
   Attendre(Nbmess > 0);
   m = T[out];
   out = (out + 1)%N;
   Nbmess--;
   Nbcell++;
}
```

```
sur_réception_de(j,(jeton,val)){
   val += Nbcell;
   Nbcell = 0;
   envoyer_à(Prod1,(jeton,val));
}
```

```
sur_réception_de(j,(app,m)){
   T[in] = m;
   in = (in + 1)%N;
   Nbmess++;
}
```

Un Scénario d'Exécution

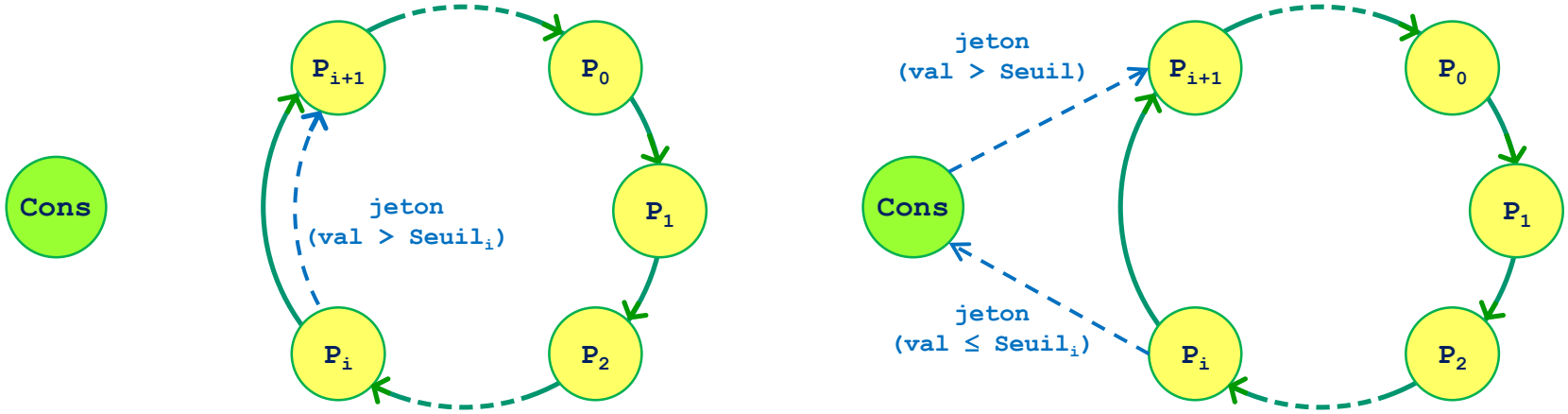


Seul le producteur $\texttt{Prod}_1$ est assuré de voir passer le jeton avec une valeur non nulle une infinité de fois : solution non équitable ni efficace.

Une solution Équitable

Proposition d'une solution équitable et plus efficace :

- L'anneau est réduit aux seuls producteurs qu'on numérote à présent de **0** à **p-1**.
- Chaque producteur voit passer le jeton avec une valeur supérieure à un seuil (**$Seuil_i$** = valeur commune à tous les producteurs).
- Lorsqu'un producteur s'aperçoit qu'après sa mise à jour le jeton contient une valeur inférieure au seuil, il le renvoie au consommateur.
- Le consommateur conserve le jeton jusqu'à ce que sa valeur soit supérieure à un deuxième seuil (supérieur ou égal au premier) et le renvoie au producteur suivant sur l'anneau.



- L'équité est garantie : le jeton est toujours reçu avec une valeur supérieur au premier seuil ($seuil_i$) et il visite autant de fois chacun des producteurs.
- Les deux seuils peuvent être ajustés au comportement de l'application, par exemple :

  $Seuil_i$: peut correspondre à une production moyenne entre deux passages du jeton,

  $Seuil$: peut prendre en compte le nombre de producteurs pour éviter un retour trop rapide du jeton.

Algorithme Équitable - Consommateur

On ajoute aux variables du consommateur, les variables suivantes :

- **Seuil**: constante entière. Elle correspond au seuil au delà duquel le consommateur envoie le jeton vers un producteur
- **Présent**: booléen. Elle prend la valeur VRAI si le consommateur retient le jeton, FAUX dans le cas contraire.
  Elle est initialisée à FAUX.
- **Prochain**: identificateur du prochain producteur auquel le consommateur va expédier le jeton.
  A chaque fois que le site **Cons** consomme un message, il incrémente **Nbcell**. Au cas où il possède le jeton, **Cons** teste si celle nouvelle valeur est supérieure **Seuil**. Si c'est le cas, il envoie le jeton au **Prochain** dans l'anneau.

```
consommer(m){
    Attendre(Nbmess > 0);
    m = T[out];
    out = (out + 1)%N;
    Nbmess--;
    Nbcell++;
    Si (Présent et Nbcell > Seuil) Alors
    |   envoyer_à(Prochain,(jeton,Nbcell));
    |   Présent = FAUX;
    |   Nbcell = 0;
    Fsi
}
```

```
sur_réception_de(j,(jeton,val)){
    Prochain = (j+1)%p;
    Nbcell += val;
    Si (Nbcell > Seuil) Alors
    |   envoyer_à(Prochain,(jeton,Nbcell));
    |   Nbcell = 0;
    Sinon
    |   Présent = VRAI;
    Fsi
}

sur_réception_de(j,(app,m)){
    T[in] = m;
    in = (in + 1)%N;
    Nbmess++;
}
```

Algorithme Équitable - Producteurs

On ajoute aux variables du producteur, la constante suivante :

- **$Seuil_i$**: initialisée à la même valeur pour tous les producteurs

  La valeur choisie doit vérifier : $Seuil_i \leq Seuil$

Seule la réception de jeton est différente de la solution précédente.

```
sur_réception_de(j,(jeton,val)){
   Temp_i = Min(Nbmess_i - Nbaut_i, val);;
   val -= Temp_i;
   Nbaut_i += Temp_i;
   Si (val > Seuil_i) Alors
   |   envoyer_à((i+1)%p,(jeton, val));
   Sinon
   |   envoyer_à(Cons,(jeton, val));
   Fsi
}
```

// Banalisation des producteurs
// Plus de test sur "dernier" ou pas

Contrôle de Flux dans les Réseaux Étendus

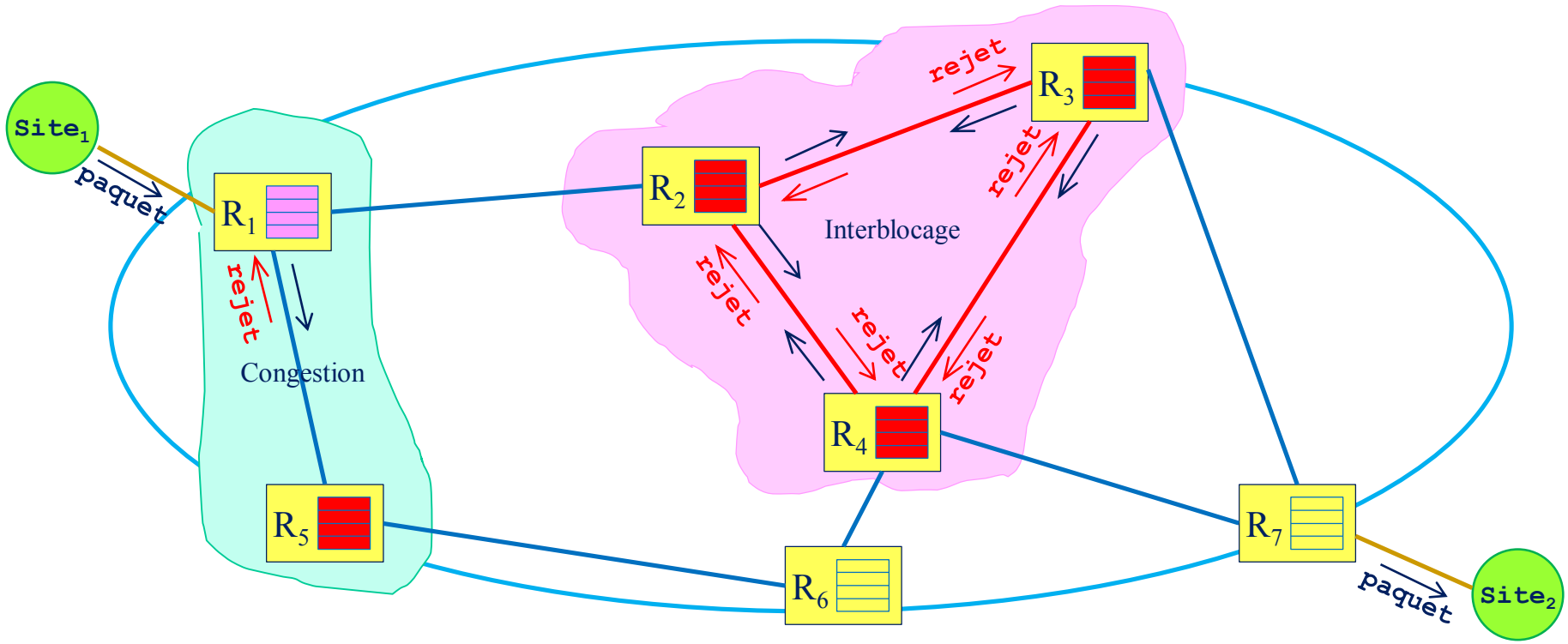
- Le transfert d'un paquet entre deux stations passe par des nœuds intermédiaires spécialisés : les routeurs (déterminent le prochain nœud à qui envoyer un paquet en fonction de la destination finale de celui-ci.
- Le routage peut être statique (établi une fois pour toute) ou adaptatif (de nouvelles routes peuvent être calculées en fonction du trafic ou de l'opérationnalité des lignes de communication ou des routeurs voisins).

Hypothèse générale : Dans la suite, nous supposons que

- un routeur (ou une station pour un paquet entrant) conserve un paquet jusqu'à ce qu'il soit accepté par le prochain routeur.
- si le paquet est arrivé à destination d'une station, le dernier routeur ne le conserve qu'un temps fini en absence de réponse de la station.

- Problèmes liés à la circulation des paquets
  - **La congestion** : paquet reçu par un routeur dont le tampon est plein (rejet). Risque de saturation du tampon émetteur. Le réseau fonctionne au ralenti.
  - **L'interblocage** : cas extrême de congestion qui se produit lorsqu'un sous-ensemble de routeurs se retrouve avec ses tampons pleins et que le prochain nœud de n'importe quel paquet d'un de ces tampons appartient à ce sous-ensemble.

Contrôle de Flux dans les Réseaux Étendus



- Stratégies de traitement des congestion et interblocage :
  - Stratégies optimistes : adjoindre aux réseaux un mécanisme de détection qui reconnaît de telles situations (calculer le taux moyen de saturation du tampon sur un intervalle donné et tester qu'il est au-dessus d'un certain seuil). Lorsque la congestion est détectée, un mécanisme de guérison se met en place (supprimer des paquets et envoyer vers les sources des flux des demandes de ralentissement (feedback)).
  - Stratégies pessimistes : mise en œuvre d'un mécanisme de prévention de l'interblocage qui réduit par voie de conséquence les risques de congestion.

Prévention par Structuration des Tampons

- Technique de structuration des tampons :
  - Exploitation des informations sur le routage connues a priori (routage statique)
  - Nombre maximum **`N`** de routeurs traversés par un paquet
    - Routage statique optimal : **`N`** = diamètre du graphe
    - Routage statique quelconque : **`N`** = nombre de routeurs
  - Le tampon d'un routeur est partitionné en **`N`** sous-tampons indicés de **`1`** à **`N`**
  - Règles définissant les contraintes de placement lors de la réception d'un paquet en fonction de sa provenance (parmi les informations de contrôle d'un paquet se trouve l'indice de son sous-tampon d'origine) :

    `Règle 1` : un paquet entrant (c'est à dire provenant d'une station connectée au réseau) est inséré dans le sous tampon d'indice **`1`**.

    `Règle 2` : un paquet provenant d'un sous-tampon d'indice **`i`** est placé dans le sous-tampon d'indice **`i+1`**.

    D'après l'hypothèse faite sur le routage, le sous-tampon **`i+1`** existe toujours (même s'il est éventuellement plein).

Proposition

L'application des règles précédentes prévient l'interblocage.

Amélioration de la Structuration des Tampons

- Solution relativement peu efficace
  - **N** étant le nombre maximum de routeurs traversés, peu de paquets traverseront les sous-tampons d'indice élevé
  - Sous exploitation des sous-tampons d'indice élevé
    - sous-dimensionnement des sous-tampons d'indice élevé : persistance du problème mais à un degré moindre
- Autre solution
  - nous supposons que le routage nous permet de connaître une borne sur le nombre de routeurs traversés par paire (source-destination)
  - à un paquet **p** entrant dans le réseau, on adjoint, à ses informations de contrôle, cette borne N_p.
  - très souvent, cette borne N_p sera largement inférieure à **N**
  - à chaque routeur traversé, la borne est décrémentée (à tout moment, elle indique ainsi une borne sur le nombre de routeurs qui restent à traverser, en incluant le routeur qui reçoit le paquet)
  - redéfinition des règles **1** et **2**

    Règle 1 : un paquet entrant **p** qui doit traverser au plus N_p routeurs est inséré dans un sous-tampon dont l'indice appartient à l'intervalle $[1..N-N_p+1]$

    Règle 2 : un paquet provenant d'un sous-tampon d'indice **i** et ayant encore à traverser au plus N_p' routeurs est placé dans un sous-tampon dont l'indice appartient à l'intervalle $[i+1..N-N_p'+1]$

    La règle 2 est toujours applicable car sur le routeur précédent, l'indice **i** du sous-tampon vérifie $i \leq N-(N_p'+1)+1$ (en vertu de la règle 1 ou 2) ce qui est équivalent à $i+1 \leq N-N_p'+1$.

Proposition

L'application des règles précédentes prévient l'interblocage.

Prévention par Estampillage des Paquets

Technique de prévention ne reposant pas sur une connaissance du routage

- Hypothèses
  - chaque routeur dispose d'une horloge qui progresse dans le temps
  - pas d'exigence que les horloges soient synchronisées (cependant, plus les horloges seront synchronisées, plus le traitement des paquets en cas de congestion sera équitable)

- Principe de la solution
  - Privilégier les paquets les plus âgés (un paquet demeurant dans le réseau finira par être âgé et bénéficiera de cette priorité)

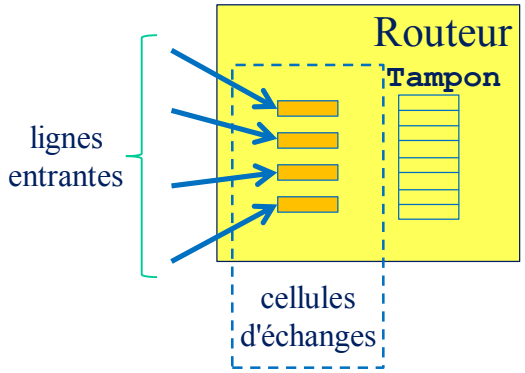
Définition : Un âge est un couple (heure, identité de site).

Soient deux âges (h_1, i_1) et (h_2, i_2) alors $(h_1, i_1) < (h_2, i_2)$ si et seulement si :

1. $h_1 < h_2$ ou
2. $h_1 = h_2$ et $i_1 < i_2$

- Lorsqu'un paquet pénètre dans le réseau, le premier routeur lui ajoute une information de contrôle appelée ***estampille*** (qui sera son âge) constituée de son heure d'arrivée et de l'identité de ce routeur.

  Deux paquets ne peuvent jamais avoir le même âge.
- Nous supposons de plus que chaque routeur dispose d'une cellule (tampon réduit à un élément) par ligne entrante. Cette cellule sera appelée cellule d'échange.

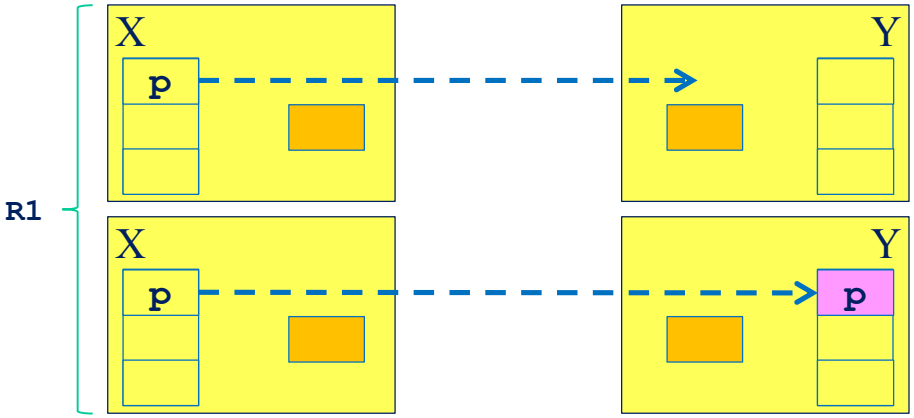


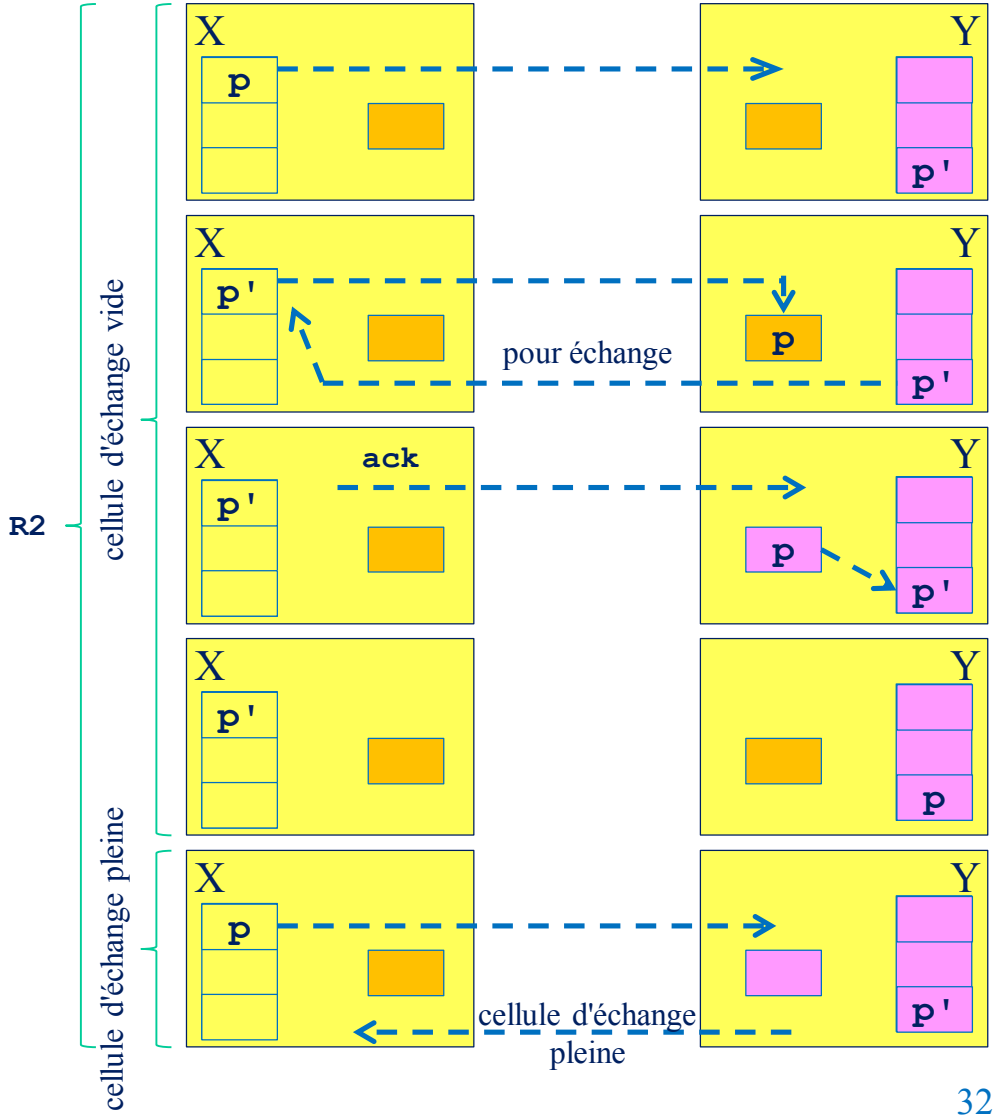
Prévention par Estampillage des Paquets

Traitement d'un paquet **p** venant du routeur **X** et arrivant sur le routeur **Y**

- **R1** : Si le tampon de **Y** n'est pas plein, alors **Y** stocke le paquet **p**
- **R2** : Si le tampon de **Y** est plein et qu'il y a un paquet **p'** à expédier à **X**, alors **Y** stocke **p** dans la cellule d'échange associée à **X** si celle-ci est vide et envoie **p'** vers **X** "pour échange" avec **p**. Lorsque **p'** arrive, **X** le place dans la cellule occupée par **p**. A la réception de l'acquittement de **p'**, **Y** place à son tour **p** dans la cellule occupée par **p'** et libère la cellule d'échange.

  Si la cellule d'échange est pleine, il rejette le paquet en indiquant "cellule d'échange pleine".





Prévention par Estampillage des Paquets

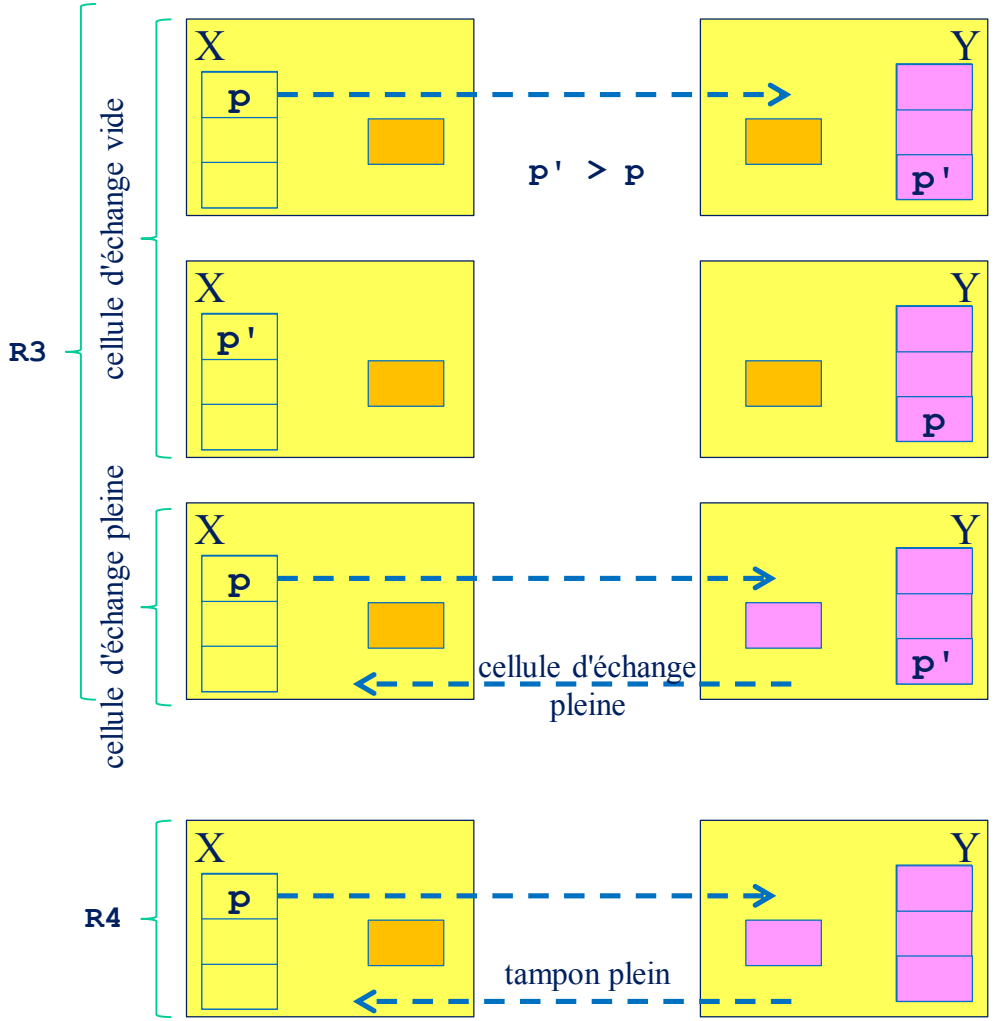
- **R3** : Si le tampon de **Y** est plein, qu'il n'a pas de paquets à expédier à **X** mais que le plus jeune des paquets de son tampon **p'** est plus jeune que **p** alors il procède comme indiqué par la règle 2. Ceci revient à détourner **p'** de sa route initiale.
- **R4** : Si aucune des règles précédentes n'est applicable alors **Y** rejette **p** pour cause de "tampon plein".

Règle du choix des paquets à envoyer par un routeur sur une ligne

- si celui-ci a des paquets pour échange, il les envoie sur la ligne en fonction de l'ordre d'arrivée dans le tampon,
- en dehors de ces paquets, si le routeur a des paquets rejetés pour cause de "cellule d'échange pleine", il renvoie uniquement le paquet dont le rejet est le plus ancien jusqu'à ce qu'il soit rejeté pour "tampon plein" ou accepté (en l'intercalant régulièrement si nécessaire entre des paquets pour échange).

Proposition

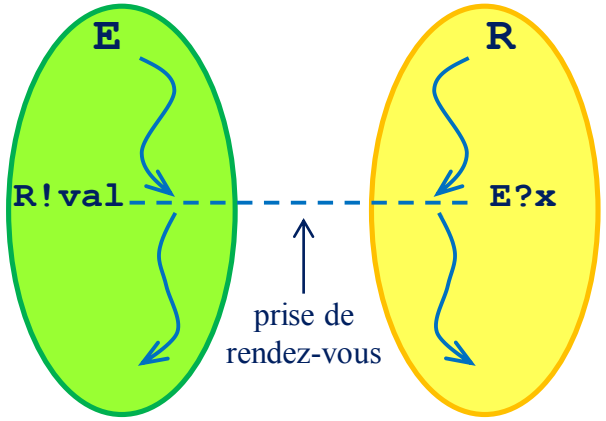
L'application des règles précédentes prévient l'interblocage



Communication Synchrone : le rendez-vous

Schéma de communication entre processus supporté par certains langages (ADA, CSP, Occam, etc.).

- Forme la plus simple : mise en jeu de deux sites
- La communication se déroule en deux phases :
  - Synchronisation : le premier processus qui appelle la primitive de rendez-vous est bloqué jusqu'à l'appel correspondant par le deuxième processus.
  - Échange de données : les modalités sont propres au langage et qui ne présente aucun intérêt algorithmique



- Forme élaborée : un site peut souhaiter un rendez-vous avec un site choisi parmi un ensemble fixé lors de l'appel à la primitive.
  - un service est assuré par un ensemble de serveurs redondants. Dans ce cas, un client souhaite un rendez-vous avec l'un quelconque des serveurs pour soumettre sa requête.
  - un service est assuré par un serveur sécurisé qui n'accepte des requêtes que d'un ensemble de clients identifiés. Dans ce cas, le serveur souhaite un rendez-vous avec l'un quelconque de ces clients pour traiter sa requête.
- Propriétés attendues de notre algorithme :
  - A aucun instant, le service ne peut engager l'application dans deux rendez-vous simultanément. ceci est un exemple de propriété de sûreté.
  - Si à un instant donné, un rendez-vous est possible en raison des différents appels en cours, alors un rendez-vous doit avoir lieu au bout d'un temps fini. Ceci est un exemple de propriété de vivacité.

Principe de la Solution

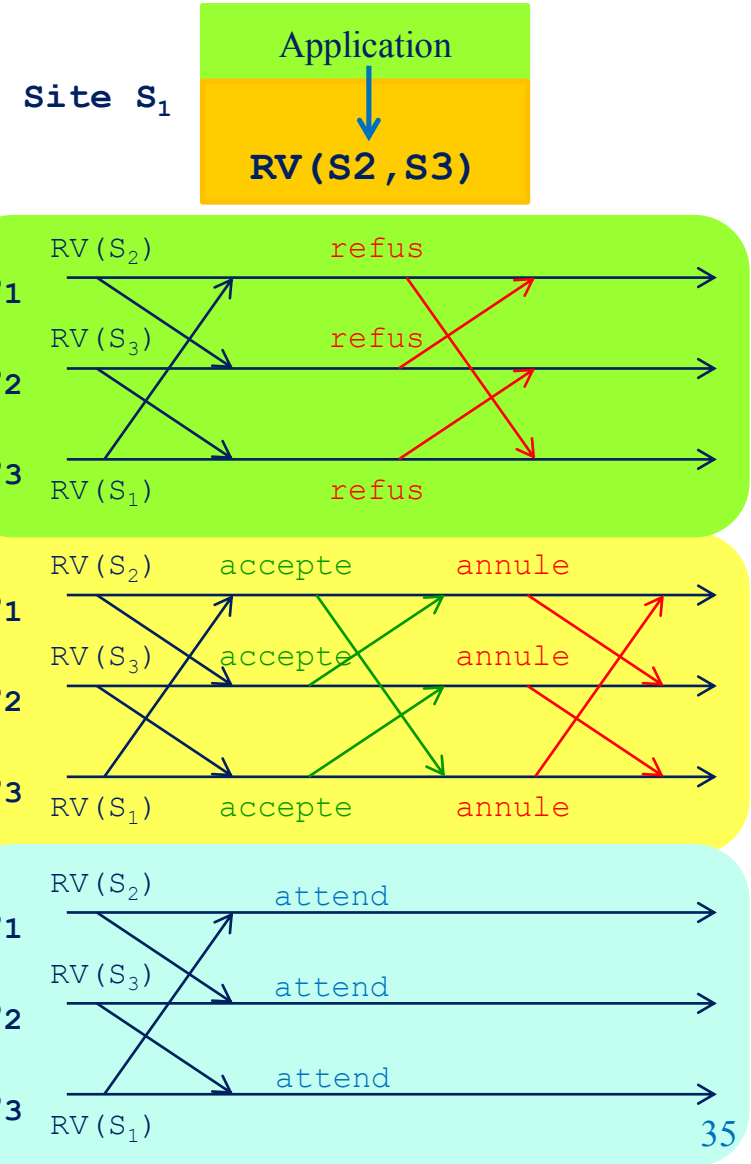
Deux problèmes que tout algorithme doit traiter

Exemple 1 : Soit une application répartie sur trois sites S_1, S_2 et S_3. L'application sur chacun des sites souhaite à peu près au même moment obtenir un rendez-vous avec l'un quelconque des deux autres sites.

Comportement 1 : Comme S_2 est "provisoirement engagé" par une requête, il rejette la requête de S_1. Par symétrie, chacun des autres sites S_1 et S_3 rejettent les requêtes reçues. La situation peut se reproduire avec le second élément de la liste. Ainsi, bien que des messages soient continuellement échangés, l'algorithme ne progresse pas vers un rendez-vous. Ce type de blocage est appelé "livelock" au sens où bien que les services soient actifs, cette activité est inutile.

Comportement 2 : S_2 accepte la requête de S_1 et annule celle faite à S_3. Par symétrie, S_1 et S_3 annulent les requêtes envoyées. La situation est similaire à la précédente avec un surcoût en nombre de messages échangés. Il s'agit ici aussi d'un "livelock".

Comportement 3 : De manière opportuniste, S_2 attend une réponse de S_3 pour rendre sa réponse à S_1 : si S_3 rejette sa requête il accepte celle de S_1 sinon il la rejette. Dans tous les cas de figure, il semble assuré d'un rendez-vous. Malheureusement, par symétrie, les autres sites font de même et tous les sites restent bloqués en attente d'une réponse. On a affaire ici à un interblocage de communication. Cette fois-ci, aucune activité n'est plus présente. On appelle cette situation un "deadlock".



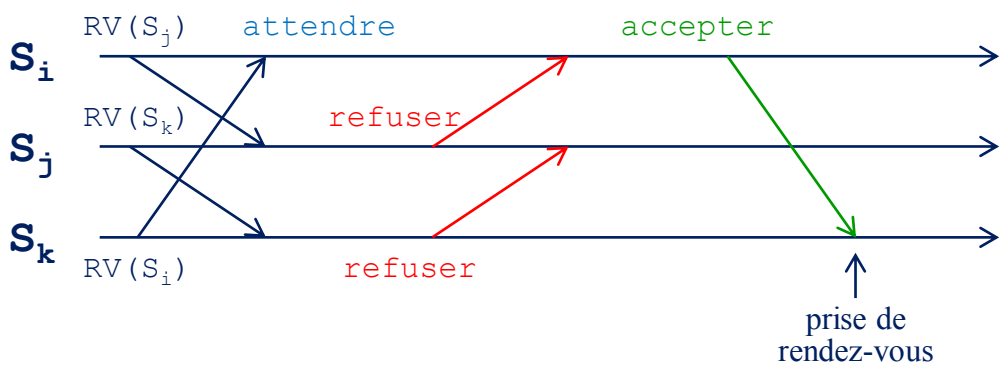
Communication Synchrone : le rendez-vous

Un des paradoxes de l'algorithmique répartie

- Pour simplifier la conception, on recherche la symétrie
- Mais, les solutions complètement symétriques ne garantissent pas la progression de l'algorithme
- Nécessité d'introduire à faible dose l'asymétrie
- Utiliser le même code en s'appuyant sur ce qui distingue chaque site : leur identité
  - Selon l'identité de l'émetteur d'une requête reçue alors que le site attend la réponse à sa propre requête , le site récepteur choisira entre le comportement 1 et le comportement 3 :

    S_i attend la réponse de S_k alors qu'il reçoit une requête de S_j
    Si j > i alors S_i retarde sa réponse à S_j
    Sinon S_i rejette la requête de S_j

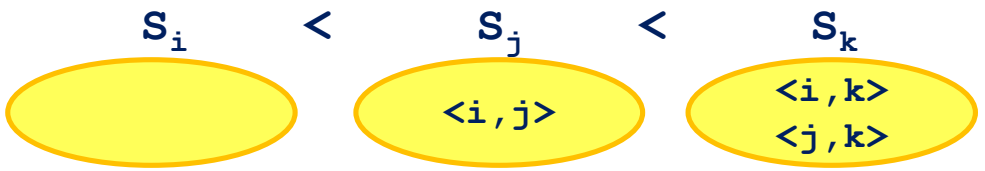
  - Exemple : trois sites S_i, S_j et S_k tels que i < j < k



Communication Synchrone : le rendez-vous

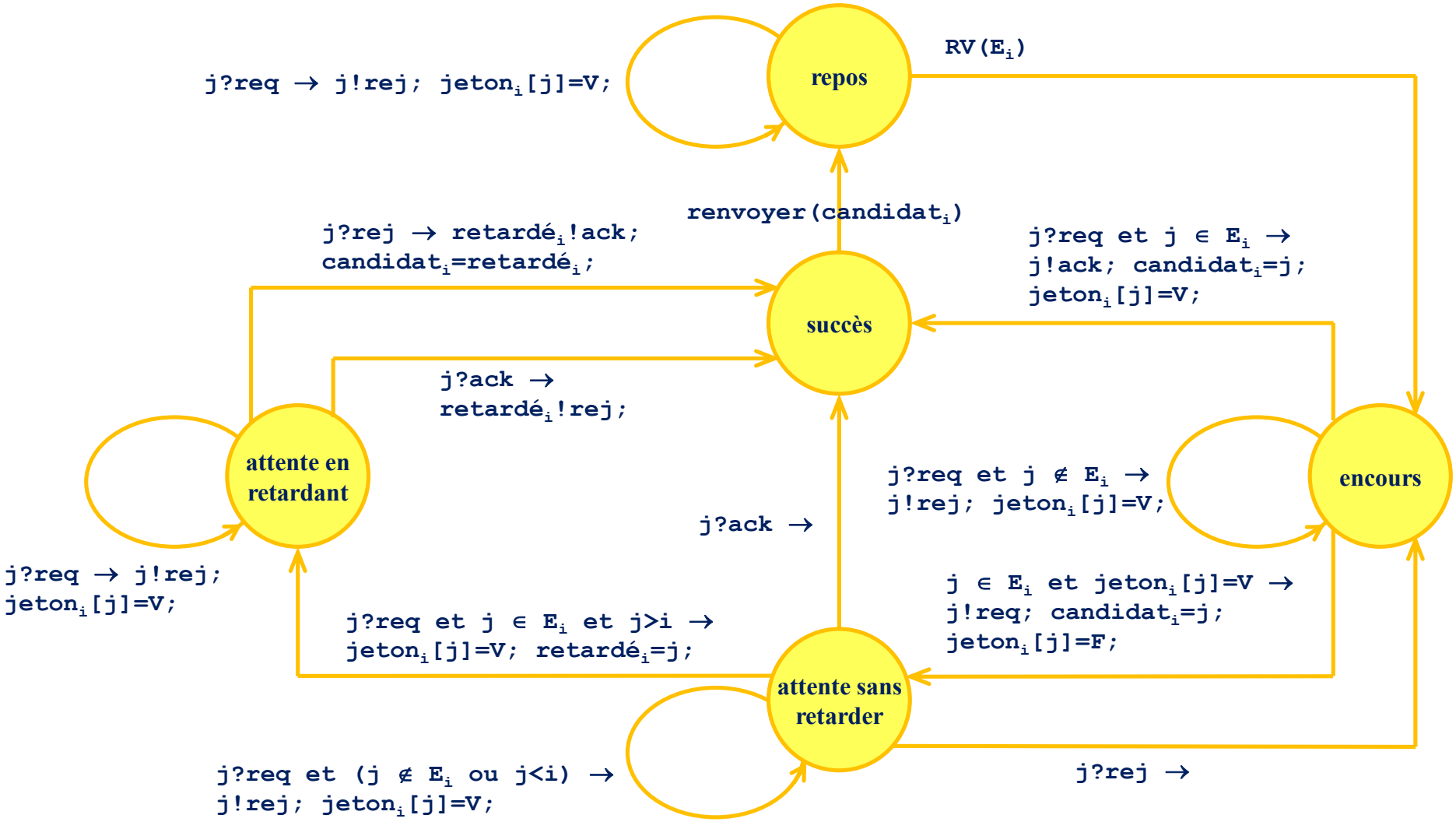
Exemple 2 : Soit une application répartie sur deux sites S_1, S_2. Supposons que l'application du site S_1 désire un rendez-vous avec S_2.

- Comme précédemment, le service du site S_1 émet une requête de rendez-vous.
- A la réception, le service du site S_2 dont l'application n'est pas à cet instant intéressée par un rendez-vous rejette la requête.
- Que doit faire le service du site S_1 à la réception de ce rejet ?
  - Il pourrait, après un délai, réémettre sa requête (problème de la configuration du délai) :
    - Trop court, il peut engendrer un trafic inutile sur le réseau.
    - Trop long, il peut retarder le fonctionnement de l'application.
- Remarque : dans le contexte du rendez-vous, il est inutile de retransmettre une requête :
  - Quand l'application du site S_2 souhaitera un rendez-vous, son service enverra une requête aussitôt acceptée.
  - Puisqu'il faut être deux pour le rendez-vous, il suffit qu'à tous instant l'un des deux ait l'initiative du rendez-vous.
- Mise en œuvre de cette initiative
  - existence d'un jeton par paire de site `{i,j}`.
  - le site qui possède le jeton a le droit d'envoyer une requête et il perd ce droit après l'envoi (de telle sorte qu'il n'y aura jamais de réémission et ceci sans perdre la possibilité du rendez-vous.
  - La répartition des jetons sur les sites est arbitraire ; par commodité de programmation, on donnera un jeton associé à une paire de sites au site de plus grande identité.



Répartition des jetons entre les sites

Le rendez-vous : Graphe d'État



Le rendez-vous : L'Algorithme

Variables du site **i** :

- **E_i**: ensemble des identités des sites avec lesquels **i** désire un rendez-vous.
- **$état_i$** : état du site à valeur dans **{repos, encours, attente, succès}** initialisé à **repos**.
- **$retardé_i$** : identité du site retardé par **i**. Par convention lorsque **i** ne retarde aucun site, sa valeur est **i** (sa valeur initiale).
- **$jeton_i[1..N]$** : tableau de booléens indiquant la présence des jetons. Initialement, **$jeton_i[j] = (i>j)$**.
- **$candidat_i$** : identité du site candidat courant.

```
RV(E_i){
    état_i = encours;
    Répéter
    |   Attendre((∃ candidat_i ∈ E_i) et (jeton_i[candidat_i] == VRAI));
    |   Si (état_i == encours) Alors;
    |   |   envoyer_à(candidat_i, req);
    |   |   jeton_i[candidat_i] = FAUX;
    |   |   état_i = attente;
    |   |   Attendre(état_i != attente);
    |   Fsi;
    Jusqu'à état_i == succès;;
    état_i = repos;
    renvoyer(candidat_i);
}
```

```
sur_réception_de(j, ack){
    état_i = succès;
    Si (retardé_i != i) Alors
    |   envoyer_à(retardé_i, rej);
    |   retardé_i = i;
    Fsi
}
```

Le rendez-vous : L'Algorithme

```
sur_réception_de(j, req){
    jeton_i[j] = VRAI;
    Si ((état_i == repos) ou (j ∉ E_i)) Alors
    |  envoyer_à(j, rej);
    Sinon si (état_i == encours) Alors
    |   état_i == succès;
    |   envoyer_à(j, ack);
    |   candidat_i = j;
    |   jeton_i[j] = VRAI;
    Sinon si ((retardé_i != i) ou (j < i)) Alors
    |   envoyer_à(j, rej);
    Sinon
    |  retardé_i = j;
    Fsi
}
```

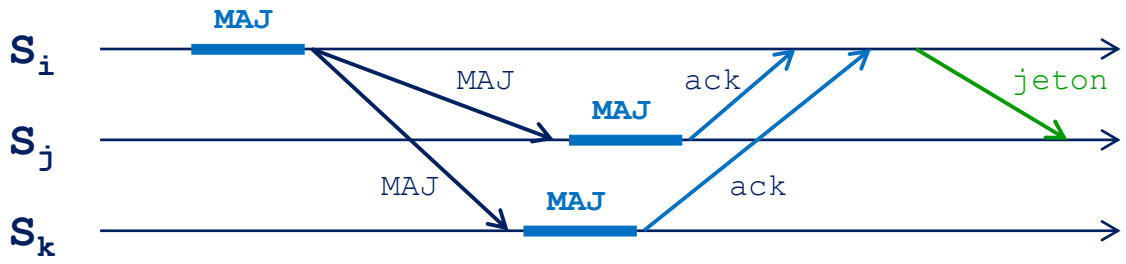
```
sur_réception_de(j, rej){
    Si (retardé_i == i) Alors
    |    état_i = encours;
    Sinon
    |   état_i = succès;
    |   envoyer_à(retardé_i, ack);
    |   candidat_i = retardé_i;
    |   retardé_i = i;
    Fsi
}
```

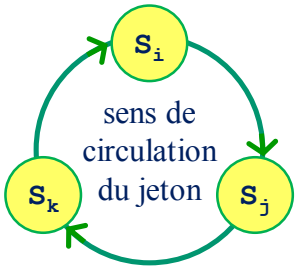
Qualité de Service : le Réseau FIFO

Exemple introductif : Soit une base de données dupliquée sur trois sites.

- assure une tolérance au pannes,
- diminue les temps de réponse par répartition de charge.,
- une lecture ne pose aucun problème de gestion,
- une mise à jour de la base nécessite un mécanisme de synchronisation pour garantir la cohérence entre les différentes copies.

- Mécanisme simple reposant sur la circulation d'un jeton entre les trois sites :
  - lorsqu'un site **S_i** désire effectuer une mise à jour, il attend le passage du **`jeton`**,
  - à la réception du **`jeton`**, **S_i** modifie sa copie et diffuse un message de mise à jour **`MAJ`** contenant la modification,
  - chaque site, qui reçoit un message **`MAJ`**, enregistre le changement dans sa propre copie et renvoie un acquittement
  - lorsque le site **S_i** initiateur de la modification a reçu tous les acquittements, il libère le jeton.



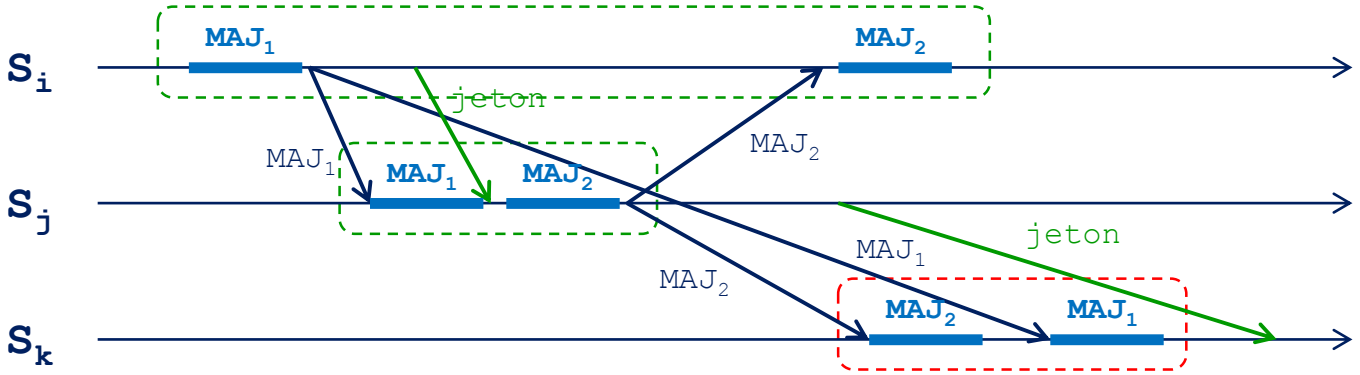


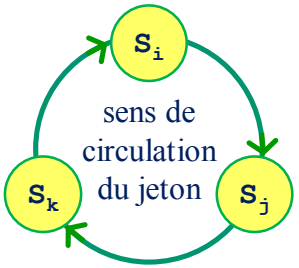
- Inconvénient de ce mécanisme
  - manque de performance : l'initiateur doit attendre tous les acquittements et par conséquent le délai d'une mise à jour est au moins celui du serveur le plus lent.

Qualité de Service : le Réseau FIFO

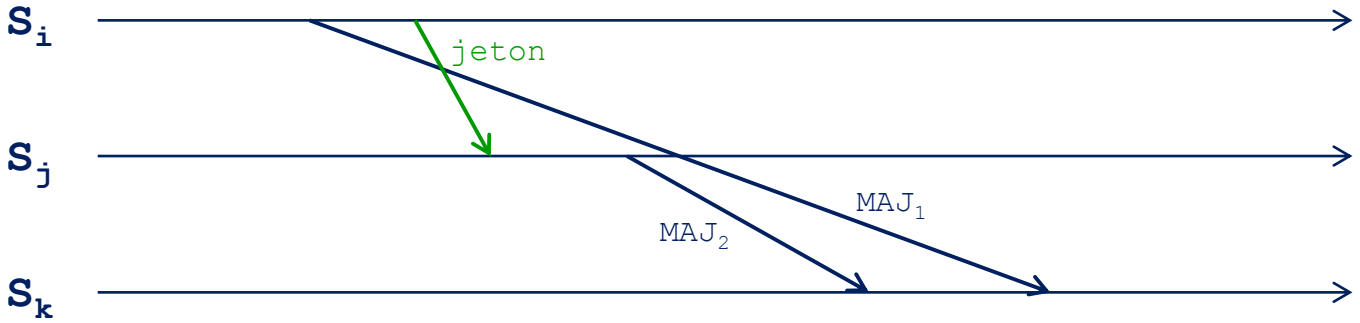
Autre alternative : libérer le jeton sans attendre les acquittements.

- Soit le scénario suivant :





- Trois messages sont à l'origine de ce problème :
  - message de mise à jour de S_i vers S_k qui précède l'envoi du **jeton** dont la réception précède l'envoi de la mise à jour de S_j vers S_k.



  - selon ces relations de précédence, il semblerai raisonnable que le troisième message soit reçu après le premier. Mais, ce n'est pas le cas.

Formalisation

Formalisation des contraintes qui interdisent ces comportements pathologiques :

- Attributs des messages : soit un message **m**
  - **m.emet** : identité du site émetteur du message
  - **m.dest** : identité du site destinataire du message
  - **m.hem** : heure "locale" de l'émission du message
  - **m.hrec** : heure "locale" de réception du message

    Aucune hypothèse n'est faite sur la synchronisation des horloges. Dans la suite, seules des heures d'un même site sont comparées.

- **Expression d'un canal FIFO** : Soient deux messages `m` et `m'` émis sur un même canal (leur ordre de réception doit être le même que leur ordre d'émission) :

  $\forall$ `m, m' m.emet = m'.emet et m.dest = m'.dest et m.hem < m'.hem`

  $\Rightarrow$ `m.hrec < m'.hrec`

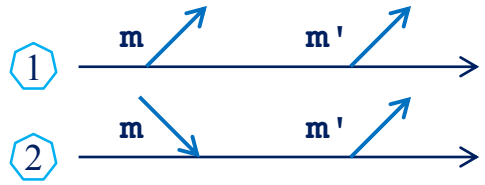
- **Ordre causal immédiat** (entre deux messages) : intuitivement, à l'émission du 2ème message, le site émetteur pourrait avoir connaissance du 1er message et donc que ce dernier a pu avoir une influence sur le second.

  Définition : Soient `m` et `m'` deux messages, alors `m` $<_i$ `m'` si et seulement si

  (1) `m.emet = m'.emet et m.hem < m'.hem`

  ou

  (2) `m.dest = m'.emet et m.hrec < m'.hem`



Du point de vue mathématique, $<_i$ n'est pas un ordre. En effet, il ne vérifie pas la propriété de transitivité.

Formalisation

- **Ordre causal général** : l'ordre causal est la fermeture transitive de l'ordre causal immédiat et, cette fois-ci, on a bien affaire à un ordre au sens mathématique. Il s'agit d'un ordre partiel (deux messages émis simultanément sur deux sites différents ne peuvent être en relation causale).

  Définition : Soient m et m' deux messages, alors $m < m'$ si et seulement si :

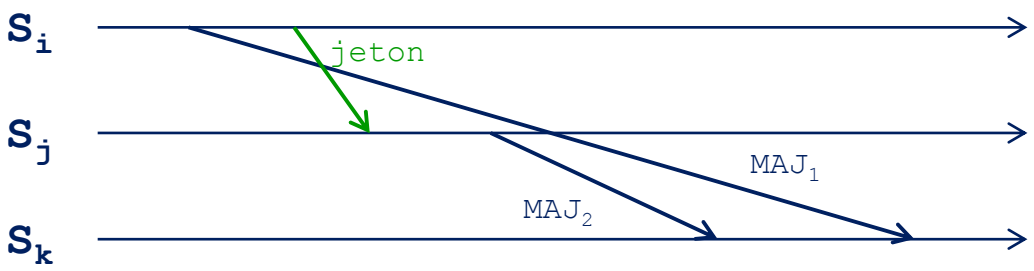
  1 $\exists\ m_1, \ldots, m_n$ tels que $m_1 = m$ et $m_n = m'$

  2 $\forall\ 0 < k < n,\ m_k <_i m_{k+1}$

- **Expression d'un réseau FIFO** : dans un tel réseau, si un message en précède causalement un autre et qu'ils ont tous deux même destination, alors le premier message sera reçu avant le second.

  Définition : un réseau est FIFO si et seulement si :

  $\forall\ m, m'\ m < m'$ et $m.dest = m'.dest \Rightarrow m.hrec < m'.hrec$



$m = MAJ_1$ et $m' = MAJ_2$
$m_1 = MAJ_1$, m_2 = jeton et $m_3 = MAJ_2$
$m_1 <_i m_2 <_i m_3$
$MAJ_1 < MAJ_2$
$MAJ_1.dest = MAJ_2.dest = S_k$
$\Rightarrow MAJ_1.hrec < MAJ_2.hrec$

Dans le cas de notre exemple, les messages MAJ_1 et MAJ_2 ont même destinataire. Comme MAJ_1 précède causalement MAJ_2, il devrait être reçu en premier.

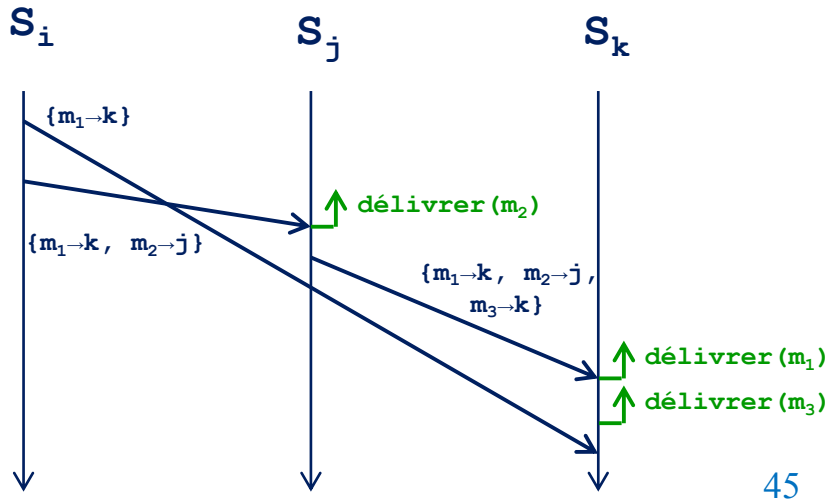
Émulation d'un Réseau FIFO

- Comme l'a illustré notre exemple, un réseau asynchrone n'est pas nécessairement FIFO (bien que ses canaux soient FIFO).
- Pour simplifier la construction d'applications :
  - nous allons définir un service qui va émuler un réseau FIFO au dessus d'un réseau asynchrone.

- Solution simple mais irréaliste
  - envoyer avec un message, la liste des messages qui le précèdent causalement,
  - l'ordre de la liste doit respecter l'ordre causale,
  - le service de chaque site maintient une liste des messages dont il a connaissance (qu'il soit ou non émetteur ou récepteur de ces messages). Initialement, cette liste est vide,
  - lorsque l'application désire envoyer un message, le service concatène ce nouveau message à la liste et envoie au service du destinataire la liste toute entière,
  - à la réception d'une liste, le service concatène à sa liste tous les messages de la liste reçue absents de sa liste dans l'ordre de leur liste,
  - pour chaque nouveau message de sa liste, si celui-ci est à destination de son application, il le délivre à celle-ci.

- Le comportement du réseau (vu de l'application) est bien FIFO car tous les messages qui précèdent causalement un message, le précèdent aussi dans toute liste où il apparaît. Par conséquent, si l'un d'entre eux a même destination que ce message, il sera délivré avant lui.

- La taille des listes croît de manière considérable : adapter cette solution pour générer moins de trafic.



Émulation d'un Réseau FIFO

- Solution plus efficace
  - transporter qu'une seule fois chaque message,
  - remplacer la liste des messages qui le précèdent causalement par l'indication de leur existence,
  - quand un message arrive, si le service a connaissance qu'un autre message à destination du même site le précède causalement et n'est pas encore reçu, alors celui-ci retarde la délivrance du message à l'application,

- Choix de l'abstraction de la liste
  - elle doit permettre de déterminer s'il faut retarder la délivrance d'un message :
    - compter le nombre de messages de la liste à destination d'un site n'est pas suffisant (l'égalité ne signifie pas nécessairement qu'il s'agit des mêmes ensembles de messages)
    - compter le nombre de messages de la liste émis par un site à destination d'un autre (ici, l'abstraction conserve suffisamment d'informations).

- Mise en œuvre
  - chaque site maintient un tableau de compteurs à deux entrées appelé **connaissance$_i$** tel que :

    connaissance$_i$[j, k] = nombre de messages émis par **j** à destination de **k** qui précèdent causalement un message qui serait émis à cet instant par **i**.
  - chaque message envoyé est accompagné de la valeur du tableau à l'instant d'émission,
  - à la réception d'un message **m**, le service du site **i** détermine s'il a reçu tous les messages qui précèdent causalement **m**. Si c'est le cas, il le délivre et met à jour son tableau en prenant le maximum, élément par élément, des deux tableaux et en prenant compte du message délivré.

Algorithme d'Émulation d'un Réseau FIFO

- Interface du service
  - **émettre_vers(j, m)** : primitive descendante du service, appelée par l'application pour envoyer le message m sur le réseau émulé,
  - **délivrer_de(j, m)** : primitive ascendante de l'application pour traiter les réceptions de messages sur le réseau émulé, appelée par le service lorsque le message m doit être délivré à l'application.
  - quand un message arrive, si le service a connaissance qu'un autre message à destination du même site le précède causalement et n'est pas encore reçu, alors celui-ci retarde la délivrance du message à l'application,

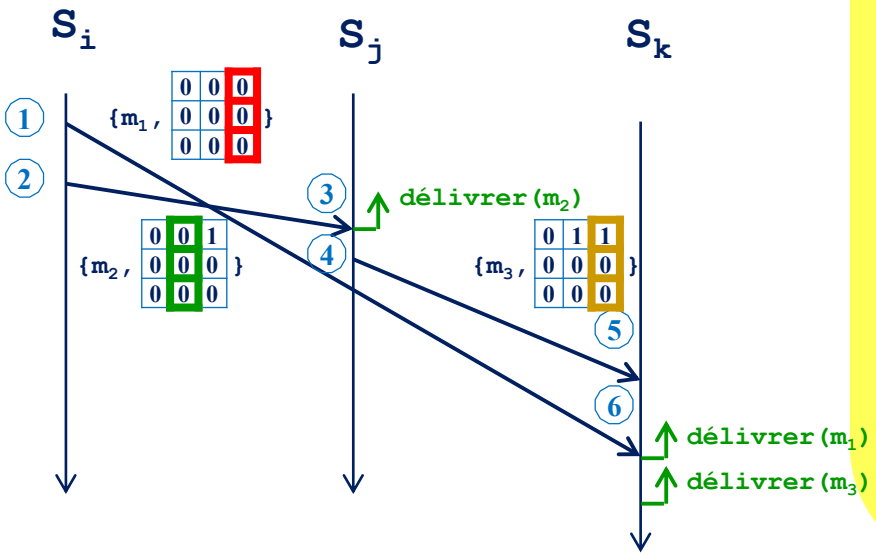
- Variable du service
  - **connaissance$_i$[1..N, 1..N]** : tableau d'entiers à deux dimensions. Initialement, tous les éléments sont nuls. **N** est le nombre de sites.
  - **tampon$_i$** : tampon des messages reçus non encore délivrés, initialement vide. Chaque message est stocké avec l'identité de son émetteur et le tableau envoyé avec lui. Le tampon dispose de trois primitives :
    - **insérer(élément)** qui ajoute un élément au tampon,
    - **tester(T, cond)** où **T** est un paramètre formel de tableau et **cond** une condition portant sur **T** et sur **connaissance$_i$**, qui renvoie `VRAI` s'il existe un élément dont le tableau jouant le rôle de **T** qui vérifie la condition,
    - **extraire(élément)** qui extrait l'élément sélectionné par la primitive précédente.

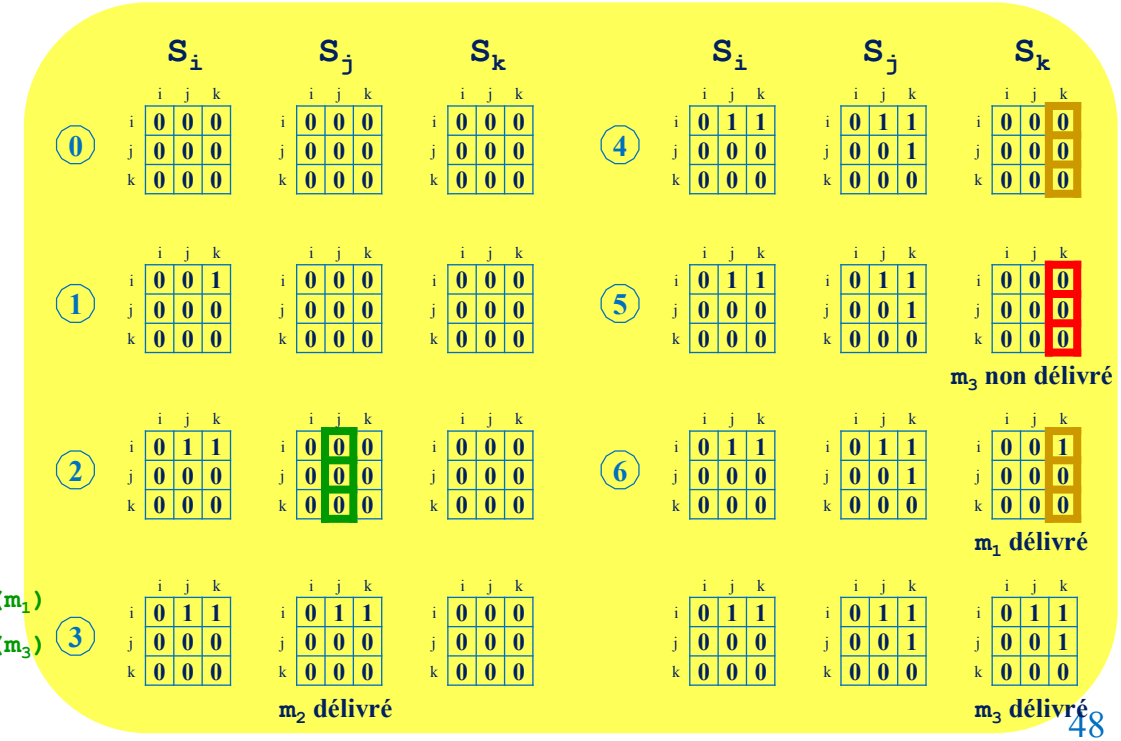
Algorithme d'Émulation d'un Réseau FIFO

```
facteur_i(){
    Tant que(VRAI)
        Attendre(tampon_i.tester(T,
                    (∀ k, connaissance_i[k,i] ≥ T[k,i]));
        tampon_i.extraire(<j, c, m>);
        connaissance_i = Max(connaissance_i, c);
        // maximum élément par élément
        connaissance_i[j, i]++;
        délivrer_de(j, m);
    Fin tant que
}
```

```
émettre_vers(j, m){
    envoyer_à(j, (connaissance_i, m));
    connaissance_i[i, j]++;
}

sur_reception_de(j, (c, m)){
    tampon_i.insérer(<j, c, m>);
}
```





Références

Cours d'Algorithmique Répartie, Joyce El Haddad et Serge Haddad, Chapitres I à VIII, Université Paris-Dauphine.

[Ray91] M. Raynal "La communication et le temps dans les réseaux et les systèmes répartis", Collection Direction des Études et des Recherches d'EDF n° 75. Hermès. 1991 ISSN 0399-4198

[Ray92a] M. Raynal "Synchronisation et état global dans les systèmes répartis", Collection Direction des Études et des Recherches d'EDF n° 79. Hermès. 1992 ISSN 0399-4198

[Ray92b] M. Raynal "Gestion des données réparties : problèmes et protocoles", Collection Direction des Études et des Recherches d'EDF n° 82. Hermès. 1991 ISSN 0399-4198

[Tan94] A. Tanenbaum "Systèmes d'exploitation. Systèmes centralisés. Systèmes distribués" Troisième édition Dunod – Prentice Hall. ISBN 2-10-004554-7. 1994

[Tel00] G. Tel "Introduction to Distributed Algorithms" Second Edition Cambridge University Press. ISBN 0-521-79483-8. 2000