Systèmes et Algorithmiques Répartis

Master 1
Informatique des Organisations - MIDO

Joyce EL HADDAD
joyce.elhaddad@dauphine.fr

Chapitre 2 : La Communication

- Contrôle de flux
- Communication synchrone
- Qualité de service

Contrôle de flux

- Lors d'une réception d'un message, si l'application n'est pas immédiatement intéressée par l'arrivée d'un message celui-ci doit être stocké pour un usage ultérieur.

- Quelque soit la taille prévue du tampon des messages, un débordement est toujours possible si un mécanisme adéquat n'est pas prévu. Ce mécanisme s'appelle le contrôle de flux.

- Contrôle point-à-point :
  - Modèle Producteur/Consommateur .

Contrôle de flux

Le modèle Producteur-Consommateur

- Deux sites, P le producteur et C le consommateur.
- Le producteur envoie des messages au consommateur.
- Lorsque l'application du producteur veut envoyer un message au consommateur elle appelle la primitive Produire(m) où m est un paramètre par valeur de message.
- Lorsque l'application du consommateur veut recevoir un message du producteur, elle appelle la primitive Consommer(m) où m est un paramètre par référence de message.

Contrôle de flux

Le modèle Producteur-Consommateur

- Asynchronisme des sites P et C : la vitesse d'exécution de P n'étant assujettie à aucune contrainte, il est possible que le rythme de production et d'émission de messages de P soit supérieur au rythme avec lequel C 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.

- Le nouveau message est soit rejeté, soit l'un des messages stockés est écrasé par ce nouvel arrivant.

Contrôle de flux

Le modèle Producteur-Consommateur

- Mise en place d'un contrôle de flux : asservir la production et l'émission des messages par le site P à des informations de consommation fournies par C.
- Le producteur disposera d'autorisations de produire *(initialement égales à la taille du tampon du consommateur).*
- A chaque envoi, le producteur dépense une autorisation.
- 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.

Contrôle de flux

Le modèle Producteur-Consommateur

- Les variables du producteur P :
  - $Nbcell_P$: la variable du producteur indiquant le nombre d'autorisations, initialisé à N.
- Les variables du consommateur C :
  - T_C : le tampon de taille N contenant les messages à consommer. T_C est géré de manière circulaire *(lorsqu'un indice est en fin de tableau, il est remis à zéro)* par deux indices :
    - in_C : indice d'insertion dans T_C, initialisé à 0.
    - out_C : indice d'extraction de T_C, initialisé à 0.
  - $Nbmess_C$: la variable indiquant le nombre de messages dans le tampon, initialisé à 0.

Contrôle de flux

❑ Le modèle Producteur-Consommateur

Algorithme du producteur P

Produire(m)

Début

Attendre ($Nbcell_P > 0$) ;

Envoyer_à (C, m) ;

$Nbcell_P = Nbcell_P - 1$;

Fin

Sur_réception_de(C,Ack)

Début

$Nbcell_P = Nbcell_P + 1$;

Fin

Algorithme du consommateur C

Consommer(m)

Début

Attendre ($Nbmess_C > 0$) ;

$m = T_C[out_C]$;

$out_C = (out_C + 1) \% N$;

$Nbmess_C = Nbmess_C - 1$;

Envoyer_à (P, Ack) ;

Fin

Sur_réception_de(P,m)

Début

$T_C[in_C] = m$;

$in_C = (in_C + 1) \% N$;

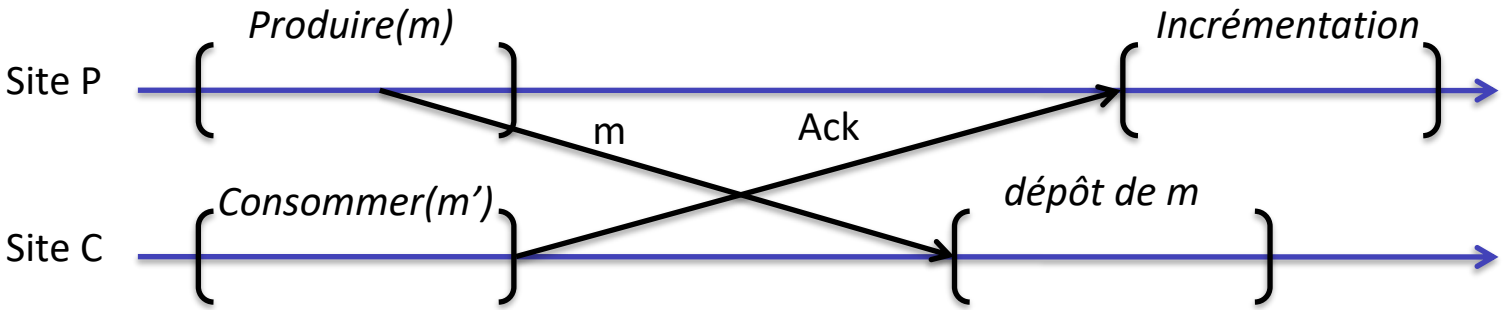
$Nbmess_C = Nbmess_C + 1$;

Fin

Contrôle de flux

Le modèle Producteur-Consommateur

- Exemple de déroulement :



- Vérification de l'algorithme :
  - Nous nous limiterons à la preuve du non débordement du tampon.
  - Démontrer que pour tout état accessible, l'égalité suivante est vérifiée : $Nbmess_C + Nbcell_P + Nbt = N$ *(Nbt désigne le nombre de messages en transit).*
  - Démonstration par induction.

Contrôle de flux

Le modèle Producteur-Consommateur

- Vérification de l'algorithme :
  - L'égalité est vérifiée initialement.
  - Supposons la vraie dans un état donné et examinons l'effet de chacune des primitives :
    - après un appel à produire, $Nbcell_P$ est décrémenté et Nbt est incrémenté,
    - après un appel à consommer, $Nbmess_C$ est décrémenté et Nbt est incrémenté,
    - après une réception d'un message, $Nbmess_C$ est incrémenté et Nbt est décrémenté,
    - après une réception d'un acquittement, $Nbcell_P$ est incrémenté et Nbt est décrémenté.
  - Dans tous les cas, l'égalité reste vérifiée.

Contrôle de flux

Le modèle Producteurs-Consommateur

- Une généralisation du modèle Producteur-Consommateur avec p sites de production P_1,…,P_p et toujours un seul site de consommation C
- Une application de ce schéma est la gestion des requêtes dans un modèle clients-serveur *(les producteurs sont les clients et le consommateur est le serveur).*
- Le site C dispose d'un tampon de stockage de N cellules.
- Les interfaces qui définissent le service sont inchangées : Produire(m) et Consommer(m).
- Problème : le partage du tampon entre les différents producteurs.

Contrôle de flux

Le modèle Producteurs-Consommateur

Solution 1 : Partage Statique

- Partager a priori les N emplacements entre les p producteurs.
- Un site P_i se voit attribuer N_i places avec $\sum N_i = N$ *(ce qui implique que $N \geq p$).*
- Plus compétition entre les producteurs puisqu'ils ne partagent plus de ressources.
- Chaque site producteur ne communique qu'avec le consommateur : au niveau du contrôle, le réseau logique est une étoile dont le site C est le centre.
- Cette solution conduit à une sous-utilisation des ressources.

Contrôle de flux

Le modèle Producteurs-Consommateur

- Solution 2 : Distribution d'autorisations sur un anneau logique
  - Faire circuler des autorisations (*correspondant à des cellules disponibles)* sur un anneau logique constitué de la façon suivante : C, P_1, P_2, ..., P_p ,C
  - A l'anneau est associé un jeton = un message de contrôle spécial qui parcourt l'anneau de manière unidirectionnelle; il visite tous les sites et ceci de façon répétitive.
  - Associer au jeton une valeur val indiquant le nombre de cellules utilisables par les producteurs qui se serviront au passage du jeton.

Contrôle de flux

Le modèle Producteurs-Consommateur

- Solution 2 : Distribution d'autorisations sur un anneau logique
  - La valeur du jeton décroît au cours des visites chez les producteurs et croît lors de la visite chez le consommateur *du nombre de cellules consommées depuis le dernier passage du jeton.*
  - Question : de quel nombre d'autorisations a besoin un producteur lorsque passe le jeton ?

Contrôle de flux

- Le modèle Producteurs-Consommateur
  - Solution 2 : Distribution d'autorisations sur un anneau logique
    - Réponse : Si on en reste à une généralisation immédiate de la solution précédente, l'application qui appelle Produire(m) se trouve bloquée jusqu'au passage du jeton. Une seule autorisation est donc nécessaire.
    - Cette solution est très inefficace car la production est synchroniser avec le passage du jeton !

Contrôle de flux

Le modèle Producteurs-Consommateur

- Solution 2 : Distribution d'autorisations sur un anneau logique
  - Désynchroniser production et passage du jeton : ajout chez chaque producteur *un tampon local* et *un processus de service* appelé le facteur.
  - L'application dépose les messages dans le tampon local et ne se bloque que lorsque le tampon est plein.
  - Le facteur envoie les messages du tampon local à C. Il se sert de deux variables : nb de messages de son tampon et nb d'autorisations dont il dispose. Tant qu'il a une autorisation, il envoie un message.
  - Lorsque le jeton arrive, un producteur cherche à obtenir autant d'autorisations que de messages à envoyer pour lesquels il ne dispose pas d'autorisation.

Contrôle de flux

- Le modèle Producteurs-Consommateur
  - Les variables d'un producteur P_i :
    - $T_i[0..N_i-1]$: tableau contenant les messages produits par le producteur P_i
      - in_i : indice d'insertion dans T_i, initialisé à 0.
      - out_i : indice d'extraction de T_i, initialisé à 0.
    - $Nbmess_i$: nombre de messages 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 P_i. Si $i < p$ alors $Succ_i$ prend la valeur *i+1* sinon la valeur de l'identificateur du site consommateur.

Contrôle de flux

Le modèle Producteurs-Consommateur

Algorithme d'un producteur P_i

Produire(m)

Début

Attendre ($Nbmess_i < N_i$) ;

$T_i [in_i] = m;$

$in_i = (in_i + 1) \% N_i$;

$nbmess_i$++ ;

Fin

Sur_réception_de(j, (jeton, val))

Début

/$temp_i$ = variable temporaire contenant la valeur minimale entre le nombre de cellules que le site désire réserver et le nombre de cellules libres associé au jeton*/*

$temp_i = Min((nbmess_i - nbaut_i) , val);$

$nbaut_i$ += $temp_i$;

val - = $temp_i$;

Envoyer_à ($Succ_i$, (jeton, val)) ;

Fin

Contrôle de flux

- Le modèle Producteurs-Consommateur
  - Algorithme d'un producteur P_i

```
Facteur ( )
    Début
        Tant que ( vrai)
            Attendre ( Nbaut_i > 0 );
            Envoyer_à( C, (app, T_i[out_i] ) );
            out_i = (out_i + 1) % N_i;
            nbaut_i--;
            nbmess_i--;
        Fin tant que
    Fin
```

Contrôle de flux

- Le modèle Producteurs-Consommateur
  - Les variables du consommateur C:
    - $T_C[0...N-1]$: un tableau contenant les messages des sites producteurs (le tampon du consommateur)
      - in_C : indice d'insertion dans T_C , initialisé à 0.
      - out_C : indice d'extraction de T_C , initialisé à 0.
    - $Nbmess_C$: le nombre de messages stockés dans T_C et non encore consommés, initialisé à 0.
    - $Nbcell_C$: le nombre de cellules libérées entre deux passages du jeton, initialisé à 0.

Contrôle de flux

Le modèle Producteurs-Consommateur

Algorithme du consommateur C

Consommer(m)

Début

Attendre ($Nbmess_C > 0$) ;

$m = T_C[out_C]$;

$out_C = (out_C + 1) \% N$;

$Nbmess_C$ -- ;

$Nbcell_C$ ++ ;

Fin

Sur_réception_de (j, (app, m))

Début

$T_C[in_C] = m$;

$in_C = (in_C + 1) \% N$;

$Nbmess_C$ ++ ;

Fin

Sur_réception_de (j, (jeton, val))

Début

val += $nbcell_C$;

$nbcell_C = 0$;

envoyer_à(1,(jeton,val));

Fin

Contrôle de flux

Le modèle Producteurs-Consommateur

- Inconvénients :
  - Bien que chaque producteur voit passer le jeton une infinité de fois, il se peut que le jeton passe systématiquement devant lui à partir d'un certain tour avec la valeur nulle. Dans ce cas, le producteur restera en attente infinie d'autorisations.
  - Seul les premiers producteurs (comme P1) sont assurés de voir passer le jeton avec une valeur non nulle une infinité de fois.

Communication synchrone

Le schéma du rendez-vous

- Dans sa forme la plus simple, le schéma du rendez-vous met en jeu deux sites.
- La communication se déroule en deux phases :
  - Phase de synchronisation : durant cette phase, le premier processus qui appelle la primitive de rendez-vous est bloqué jusqu'à l'appel correspondant par le deuxième processus.
  - Phase d'échange : suite la phase de synchronisation s'engage la phase d'échange de données qui ne présente pas d'intérêt algorithmique.
- Nous nous limiterons à l'étude de la phase de synchronisation.

Communication synchrone

Le schéma du rendez-vous

- Dans une forme plus élaborée, un site peut souhaiter un rendez-vous avec un site choisi parmi un ensemble fixé lors de l'appel à la primitive
  - *ex1 : pour un service assuré par un ensemble de serveurs redondants, un client souhaite un rendez-vous avec l'un quelconque des serveurs pour soumettre sa requête*
  - *ex2 : pour un service assuré par un serveur sécurisé qui n'accepte des requêtes que d'un ensemble de clients identifiés, le serveur souhaite un rendez-vous avec l'un quelconque de ces clients pour traiter sa requête*
- C'est la forme plus élaborée que nous allons étudier

Communication synchrone

Le schéma du rendez-vous

- Les propriétés attendues de notre algorithme :
  - Propriété de sûreté : A aucun instant, le service ne peut engager l'application dans deux rendez-vous simultanément.
  - Propriété de vivacité : 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.

- Notons que rien n'est précisé sur l'identité des participants au rendez-vous

Communication synchrone

Le schéma du rendez-vous

- Nous allons d'abord exhiber deux problèmes que tout algorithme de rendez-vous doit traiter. Nous illustrons ces problèmes à l'aide d'exemples.

- Problème 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. Pour obtenir un rendez-vous, le service de chaque site interroge le service d'un autre site pour savoir si le rendez-vous est aussi souhaité : *S_1, envoie une requête de rendez-vous au premier site de sa liste S_2. De manière similaire, S_2 a envoyé sa requête à S_3, celui ayant fait de même avec S_1.*

Communication synchrone

Le schéma du rendez-vous

- Les différents choix qui s'offrent à S_2 lors de la réception de la requête de S_1 :
  - Comportement 1 : S_2 est provisoirement engagé par sa requête à S_3, alors il rejette la requête du site S_1. Par symétrie, S_1 et S_3 rejettent les requêtes reçues. Bien que des messages peuvent continuellement être échanger, l'algorithme ne progresse pas vers un rendez-vous. Ce type de blocage est appelé livelock
  - 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

Communication synchrone

Le schéma du rendez-vous

- Les différents choix qui s'offrent à S_2 lors de la réception de la requête de S_1 :
  - Comportement 3 : 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. Par symétrie, les autres sites font de même et tous les sites restent bloqués en attente d'une réponse. On appelle cette situation un deadlock

- Un paradoxe 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.

Communication synchrone

Le schéma du rendez-vous

- Solution : Il faut introduire à faible dose l'asymétrie
  - Avec un code identique en s'appuyant sur ce qui distingue chaque site, *son identité.*
  - Selon l'identité de l'émetteur d'une requête reçue que le site qui attend une réponse à sa propre requête, choisira son comportement.
  - Supposons que S_i attende la réponse de S_k et qu'il reçoive une requête venant de S_j :

si (j > i) alors S_i retarde sa réponse à S_j

sinon il rejette cette requête

Communication synchrone

Le schéma du rendez-vous

- Problème 2 : Soit une application répartie sur deux sites S_1 et S_2. Supposons que l'application de S_1 désire un rendez-vous avec S_2. Le service de S_1 émet une requête. A la réception, le service de S_2 dont l'application n'est pas à cet instant intéressée par un rendez-vous rejette la requête. Que doit faire le service de S_1 à la réception de ce rejet?
- Après un délai, réémettre sa requête. Ceci soulève le 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.

Communication synchrone

Le schéma du rendez-vous

- Solution : Il est inutile de retransmettre une requête, il faut être deux pour le rendez-vous, il suffit alors qu'à tout instant l'un des deux ait l'initiative du rendez-vous.
  - La mise en œuvre de cette prise d'initiative se fait par l'intermédiaire d'un *jeton par paire de sites {i,j}.*
  - Le site qui possède le jeton a le droit d'envoyer une requête et il perd ce droit avec l'envoi de telle sorte qu'il n'y aura jamais de réémission et ceci sans perdre la possibilité du rendez-vous.
  - La *répartition initiale* des jetons sur les sites est arbitraire; par commodité, on donnera *un jeton associé à une paire de sites au site de plus grande identité.*

Communication synchrone

Le schéma du rendez-vous

- Graphe d'état d'un service de rendez-vous de S_i
  - Les états sont représentés par les nœuds du graphe et les arcs correspondent aux transitions d'états.
  - Chaque transition respecte la syntaxe : « Garde → Actions » où la garde est une expression booléenne traduisant les conditions d'occurrence de la transition et les actions indiquent les modifications des variables locales du service.
  - Afin de rendre plus compacte la représentation :
    - j ? m : la réception d'un message m venant du site S_j
    - j ! m : l'envoi par S_i au site S_j du message m

Communication synchrone

Le schéma du rendez-vous

- Graphe d'état d'un service de rendez-vous de S_i
  - Lorsque l'application du site S_i n'a pas appelé la primitive de rendez-vous, le service est au *repos.*
  - A l'appel de la primitive de rendez-vous RV, la variable E_i est initialisée avec le sous-ensemble des sites, possibles correspondants du rendez-vous. Le site passe à l'état *encours.*
  - Dans l'état encours, le site recherche un $candidat_i$ (c'est à dire un site de E_i) pour lequel il possède le jeton associé à la paire {i, $candidat_i$}.
  - La présence des jetons est mémorisée dans le tableau de booléens $jeton_i$

Communication synchrone

Le schéma du rendez-vous

- Graphe d'état d'un service de rendez-vous de S_i
  - Après l'envoi d'une requête à $candidat_i$, le service passe alors à l'état *attente-sans-retarder*
    - A la réception d'un acquittement, le site passe à l'état *succès* puis au retour de la primitive (ayant renvoyé à l'application le site retenu) de nouveau au *repos*
    - En cas de rejet, le site retourne à *encours* pour rechercher un autre $candidat_i$.
  - Alors que le site est dans l'état *attente-sans-retarder*, celui-ci peut recevoir une requête qui lui conviendrait; il passe à l'état *attente-en-retardant* où le site retardé est indiqué par la variable $retarde_i$

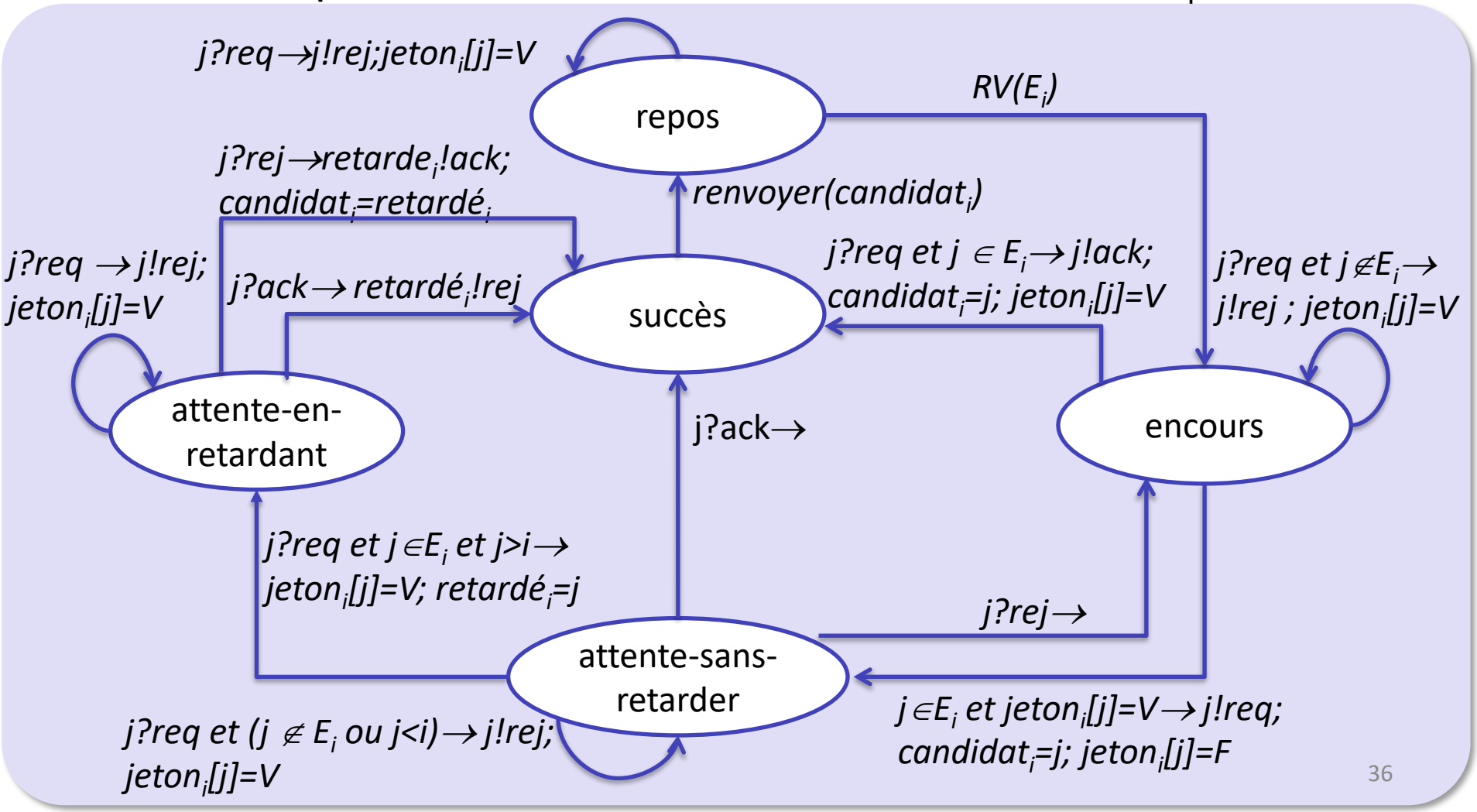
Communication synchrone

Le schéma du rendez-vous

- Graphe d'état d'un service de rendez-vous de S_i
  - Dans l'état *attente-en-retardant*, quelque soit la réponse du $candidat_i$, S_i passera à l'état *succès* avec éventuellement un changement d'identité de $candidat_i$. Dans tous les cas, S_i enverra une réponse à $retarde_i$
  - Dans un état où le site n'est pas en cours de service (*repos*) ou éventuellement bloqué (*encours, attente-sans-retarder, attente-en-retardant*), il faut prendre en compte les éventuelles réceptions de requêtes :
    - S'il reçoit une requête dans l'état *attente-en-retardant*, il la rejette car il est assuré d'un RV
    - S'il reçoit une requête dans l'état *encours,* d'un site de E_i, il l'accepte et passe directement à l'état *succès*

Communication synchrone

- Le schéma du rendez-vous
  - Graphe d'état d'un service de rendez-vous de S_i



Communication synchrone

Le schéma du rendez-vous

- Les variables d'un site S_i
  - E_i : ensemble des identités des sites avec lesquels S_i désire un rendez-vous
  - $état_i$: état du site à valeur dans {repos, encours, attente, succès}, initialisé à repos
  - $retarde_i$: identité du site retardé par S_i. Par convention lorsque S_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, $jeton_i[j]$ = Vrai pour i > j
  - $candidat_i$: identité du site candidat courant

Communication synchrone

- Le schéma du rendez-vous
  - Algorithme d'un site S_i

Sur_réception_de (j, ack)

Début

$état_i$ = succès;

Si $retarde_i$!= i Alors

Envoyer_à ($retarde_i$, rej);

$retarde_i$ = i;

Fsi

Fin

Sur_réception_de (j, rej)

Début

Si ($retarde_i$ = i) Alors

$état_i$ = encours;

Sinon

$état_i$ = succès;

Envoyer_à ($retarde_i$, ack) ;

$candidat_i$ = $retarde_i$;

$retarde_i$ = i;

Fsi

Fin

Communication synchrone

- Le schéma du rendez-vous
  - Algorithme d'un site S_i

```
Sur_réception_de ( j, req )
    Début
        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; candidat_i=j;
                    Envoyer_à(j,ack);
            sinon
                    si (retarde_i != i) ou ( j < i ) Alors
                            Envoyer_à(j,rej);
                    sinon   retarde_i = j ;
    Fin
```

Communication synchrone

- Le schéma du rendez-vous
  - Algorithme d'un site S_i

RV(E_i)

```
Début
        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);
        renvoyer(candidat_i); état_i = repos;
Fin
```

Qualité de service

Un réseau FIFO

- Supposons qu'une base de données soit dupliquée sur trois sites afin d'assurer une tolérance aux pannes et de diminuer les temps de réponse par répartition de la charge.

- Une lecture ne pose aucun problème de gestion tandis qu'une mise à jour de la base nécessite un mécanisme de *synchronisation* pour garantir la cohérence entre les différentes copies.

- Mécanisme simple : la circulation d'un jeton entre les sites.

Qualité de service

Un réseau FIFO

- Lorsqu'un site S_i désire effectuer une mise à jour, il attend le passage du jeton. Puis S_i modifie sa copie et diffuse un message de mise à jour MAJ contenant les modifications.
- Chaque site S_j, qui reçoit un message MAJ, enregistre les changements dans sa propre copie et renvoie un acquittement.
- Lorsque le site initiateur de la modification a reçu tous les acquittements, il libère le jeton.
- Manque de performances : l'initiateur doit attendre tous les acquittements. *Le délai d'une mise à jour est au moins celui du serveur le plus lent.*

Qualité de service

- Un réseau FIFO
  - Mécanisme alternatif : libérer le jeton sans attendre d'acquittement
    - S_1 effectue sa mise à jour, envoie le message de mise à jour à S_2 et S_3 et expédie ensuite le jeton à S_2
    - Quand S_2 reçoit le message de mise à jour, il enregistre les changements. A la réception du jeton, S_2 effectue une autre mise à jour sur la copie locale qu'il maintient et diffuse son message de mise à jour à S_1 et S_3
    - S_3 reçoit le jeton. Supposons que la mise à jour de S_2 arrive à S_3 avant la mise à jour de S_1. S_3 appliquera les changements de S_2 puis ceux de S_1, ceci conduit à une incohérence de la base de données.

Qualité de service

Un réseau FIFO

- Mécanisme alternatif : libérer le jeton sans attendre d'acquittement
  - Trois messages à l'origine du problème : la mise à jour de S_1 vers S_2 qui précède l'envoi du jeton dont la réception précède l'envoi de la mise à jour de S_1 vers S_3

- Formaliser les contraintes qui interdisent ces comportements pathologiques
  - Chaque site dispose d'une horloge qui progresse avec le temps. Un message m est défini par les attributs : m.emet (id. du site émetteur), m.dest (id. du site destinataire), m.hem (heure "locale" de l'émission), m.hdest (heure "locale" de réception)

Qualité de service

- Un réseau FIFO
  - Exprimer qu'un canal est FIFO
    - Considérons deux messages m et m' émis sur un même canal. Ce canal est FIFO si l'ordre de réception de m et m' est le même que leur ordre d'émission
    - Ce qui s'exprime par :

$$\forall m, m', (m.emet = m'.emet) \text{ et } (m.dest = m'.dest) \text{ et } (m.hem < m'.hem) \Rightarrow m.hdest < m'.hdest$$

  - Ordre causal entre deux messages [Lamport 1978] : *a pour signification qu'à l'émission du deuxième message, le site émetteur pourrait avoir connaissance du premier message et donc que ce premier message a pu avoir une influence sur le second message*

Qualité de service

- Un réseau FIFO
  - Ordre causal
    - Débutons par l'ordre causal immédiat "$<_i$" qui caractérise de telles situations uniquement par l'historique d'exécution d'un site :

$\forall m, m',\ m <_i m'$ si et seulement si

$(m.emet = m'.emet)$ et $(m.hem < m'.hem)$

ou

$(m.dest = m'.emet)$ et $(m.hdest < m'.hem)$

    - La relation "$<_i$" n'est pas un ordre. Elle ne vérifie pas la propriété de transitivité.

Qualité de service

- Un réseau FIFO
  - Ordre causal
    - La fermeture transitive de l'ordre causal immédiat $\Rightarrow$ ordre causal général "$<_c$"

$\forall m, m',\ m <_c m'$ si et seulement si

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

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

    - ordre *partiel* : deux messages émis simultanément sur deux sites différents ne peuvent être en relation causale.

Qualité de service

- Un réseau FIFO
  - Caractérisons un réseau FIFO avec la relation d'ordre causal
    - 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.
    - Ce qui s'exprime par :

$$\forall m, m',\ m <_c m' \text{ et } m.dest = m'.dest \Rightarrow m.hdest < m'.hdest$$

  - Emuler un réseau FIFO

Qualité de service

Emuler un réseau FIFO

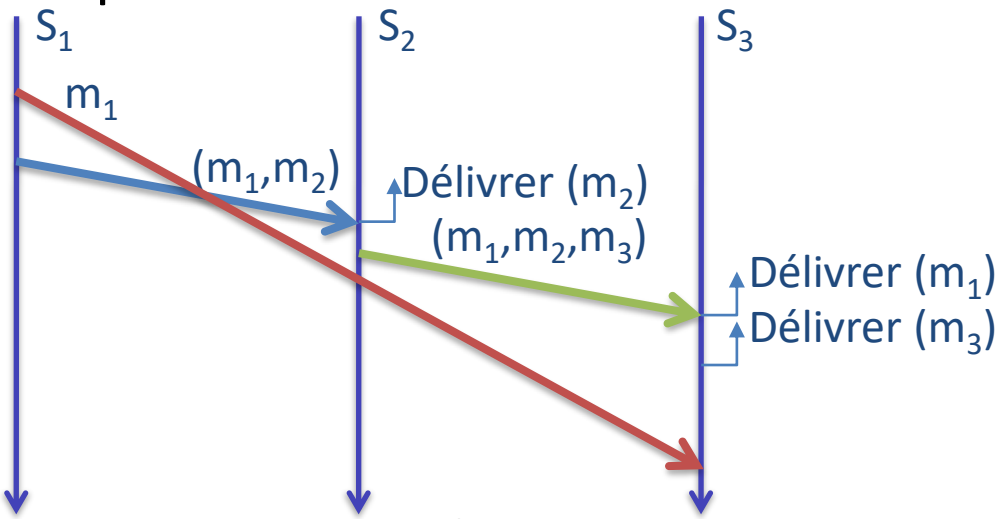
- Un réseau asynchrone n'est pas nécessairement FIFO bien que tous ses canaux soient FIFO.
- Emuler un réseau FIFO au dessus d'un réseau asynchrone.
- Solution 1 : simple mais irréaliste
  - Envoyer avec un message, la liste des messages qui le précèdent causalement. L'ordre de la liste doit respecter l'ordre causal.
  - 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.

Qualité de service

- Emuler un réseau FIFO
  - Solution 1 : simple mais irréaliste
    - 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.
    - A la réception d'une liste, le service concatène à sa liste tous les messages de la liste reçue absents de sa liste.
    - Pour chaque nouveau message de sa liste, si celui-ci est à destination de son application, il le délivre à celle-ci.

Qualité de service

- Emuler un réseau FIFO
  - Solution 1 : simple mais irréaliste



    - Le comportement vue de l'application est bien FIFO *(les messages qui précèdent causalement un message m, le précèdent dans toute liste où il apparaît. Si l'un d'entre eux a même destination que m, il sera délivré avant lui).*
    - Au fur et à mesure de l'exécution de l'application, la taille des listes de messages croît de manière considérable. Il faut adapter cette solution pour générer moins de trafic.

Qualité de service

- Emuler un réseau FIFO
  - Solution 2: une solution plus efficace
    - Idée : ne transporter qu'une seule fois chaque message et remplacer la liste des messages qui le précèdent causalement par l'indication de leur existence.
    - Quand un message arrive, si le service a l'information 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.
    - Quelle abstraction de la liste ?
    - La représentation doit permettre de déterminer s'il faut retarder la délivrance d'un message.

Qualité de service

- Emuler un réseau FIFO
  - Solution 2: une solution plus efficace
    - Compter le nombre de messages de la liste à destination d'un site n'est pas suffisant *(l'égalité de deux compteurs ne signifierait pas nécessairement qu'il s'agit des mêmes messages)*
    - Plus finement, compter le nombre de messages de la liste émis par un site à destination d'un autre, l'abstraction conserve suffisamment d'informations.
    - Chaque site S_i maintient un tableau de compteurs à deux entrées appelé $connaissance_i$ tel que :
      - $connaissance_i[j,k]$ = le nombre de messages émis par S_j à destination de S_k qui précèderaient causalement un message émis à cet instant par S_i

Qualité de service

- Emuler un réseau FIFO
  - Solution 2: une solution plus efficace
    - A l'émission, chaque message envoyé est accompagné de la valeur courante du tableau.
    - A la réception d'un message m, le service d'un site S_i stocke le message m et son tableau dans un tampon (pour diminuer le temps d'exécution des réceptions de message) et confie la délivrance à un processus de service appelé Facteur ().

Qualité de service

- Emuler un réseau FIFO
  - Solution 2: une solution plus efficace
    - Le processus Facteur() d'un site S_i détermine s'il a reçu tous les messages qui précèdent causalement m en comparant élément par élément la colonne i de son tableau avec celle du tableau de 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 le message délivré.
      - Sinon, il garde le message m et son tableau dans le tampon.

Qualité de service

- Emuler un réseau FIFO
  - Solution 2: une solution plus efficace
    - Le service doit émuler un réseau. Il y aura donc une primitive descendante et une primitive montante :
      - Emettre_vers (j,m) : primitive du service, appelée par l'application pour envoyer un message sur le réseau émulé.
      - Délivrer_de (j,m) : primitive de l'application pour traiter les réceptions de message sur le réseau émulé, appelée par le service lorsqu'un message doit être délivré à l'application.

Qualité de service

- Emuler un réseau FIFO
  - Les variables d'un site S_i
    - $connaissance_i[1..N,1..N]$: tableau d'entiers à deux dimensions. Initialement, tout les éléments de ce tableau sont nuls. N est le nombre de sites.
    - $tampon_i$: tampon des message 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 tableau et cond une condition portant sur T et $connaissance_i$. Elle renvoie Vrai s'il existe un élément dont le tableau T vérifie la condition,
      - extraire(élément) qui extrait un élément sélectionné.

Qualité de service

- Emuler un réseau FIFO
  - Algorithme d'un site S_i

Emettre_vers(j,m)

Début

Envoyer_à(j, ($connaissance_i$, m));

$connaissance_i$[i,j]++;

Fin

Sur_réception_de (j, (c, m))

Début

$tampon_i$.insérer (< j, c, m >);

Fin

Qualité de service

- Emuler un réseau FIFO
  - Algorithme d'un site S_i

```
Facteur ( )
    Début
        Tant que(Vrai)
            Attendre( tampon_i.tester(T,∀k,connaissance_i[k,i]≥T[k,i]) );
            tampon_i.extraire( < j, c, m > );
            /*maximum élément par élément*/
            connaissance_i=Max (connaissance_i , c );
            connaissance_i[j,i]++;
            Délivrer_de ( j, m );
        Fin tant que
    Fin
```

Références

- R.L. Bagrodia, "Synchronisation of asynchronous processes in CSP", ACM Toplas, vol 11, numero 4, pp. 585-597, 1989
- S. Toueg, J.D. Ullman, "Deadlock-free packet switching networks", SIAM Journal of Computing, volume 10, numero 3, pp. 594-611, 1981
- P.P. Merlin, P.J. Schweitzer, "Deadlock avoidance in store-and forward networks I : Store and Forward deadlock", IEEE Transactions on Communcations COM-28, pp. 345-360, 1980
- G.V. Bochmann, "Architecture of distributed computer", Springer-Verlag, LNCS 77, 1979
- L. Lamport, "Time, clocks, and the ordering of events in a distributed system", Communications of ACM 21, pp. 558-564, 1978
- A. Tanenbaum, M. van Steen, "Distributed Systems: Principles and Paradigms", Edition Prentice Hall, 2002.
- S. Krakowia, "Middleware Architecture with Patterns and Frameworks ", http://sardes.inrialpes.fr/~krakowia/MW-Book/
- L. Seinturier, "Middleware", http://www.lifl.fr/~seinturi/middleware/