TD d'algorithmes Distribués

Année 2020-21

Version 2.1

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

Rodolphe Giroudeau
161, rue Ada
34392 MONTPELLIER Cedex 5
TEL : 04-67-41-85-40
Mail : rgirou@lirmm.fr

Université de Montpellier 2018/2019
HMIN213 Durée: 4.5h

Horloge
TD – Séance n° 1

Exercice 1 – Robustesse d'un protocole par rapport au réseau

1. Soit p la probabilité de perte d'un message et e le nombre d'échanges nécessaires pour réaliser le protocole. Calculer le taux d'échecs d'un protocole en fonction de p et de e sachant que la probabilité $P(X = k)$ suit une loi binomiale. Evaluer la probabilité quand $p = 0.001$ et $e = 1000$, et $p = 0.01$ et $e = 1000$.
2. Regardons maintenant combien de fois il faut relancer le protocole pour qu'il ait $x\%$ de se terminer sans perte. Soit T le nombre de tentatives. Soit p_t la probabilité qu'aucun message ne se perde lors d'une tentative. Soit $x\%$ la probabilité tel que le protocole se termine correctement. Donner une formule en fonction de T et x qui la probabilité que le protocole se termine correctement avec une probabilité d'au moins de $x\%$. Evaluer la probabilité quand $x = 95\%$ et $p_t = 0,368$ et $x = 99\%$ et $p_t = 0,368$.

Exercice 2 – Horloges de Lamport, de Martell et matricielle

Nous considérons les horloges de Lamport définies de la manière suivante :

Une horloge est une fonction H définie à partir des événements vers un ensemble ordonné tel que $a \prec b \Rightarrow H(a) < H(b)$. L'horloge de Lamport HL affecte à chaque événement a la longueur k de la plus longue chaîne de causalité $b_1 \prec \ldots \prec b_k = a$.

Un algorithme distribué permet de calculer HL et sera basé sur les caractéristiques suivantes :

— Si a est un événement interne ou l'envoi de messages, soit k la valeur de l'horloge de l'événement précédent ($k = 0$ si il n'y a pas de événement précédent) $HL(a) = k + 1$;
— Si a est l'événement de réception, k la valeur de l'horloge de l'événement précédent ($k = 0$ si il n'y a pas de événement précédent), et $HL(b)-$ la valeur de l'horloge de l'événement b, alors $HL(a) = \max\{k, HL(b)\} + 1$.

1. Donner l'algorithme qui permet de calculer les horloges logiques.
2. Donner les valeurs des horloges pour la figure 1. Avons-nous un ordre total ?
3. Que devons rajouter à la définition précédente pour créer un ordre total ?
4. Montrer que si $e \rightsquigarrow e'$ alors $H(e) \prec H(e')$. Utiliser par récurrence sur la longueur n de la suite d'évènement reliant e à e'

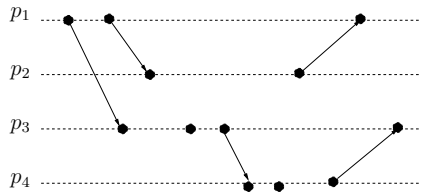

FIGURE 1 – *Sur les horloges de Lamport*

5. L'ordre causal est l'ordre partiel sur les événements : Si a et b sont deux événements a précède $b, (a \rightsquigarrow b)$ si et seulement si l'une des trois conditions est vraie :
   — a et b ont lieu sur le même site avec a avant b,
   — $a = ENVOYER(< M >)$ et $b = RECEVOIR(< M >)$ du même message,
   — Il existe un évènement c tel que $a \rightsquigarrow c$ et $c \rightsquigarrow b$

   (a) Soit la figure 2

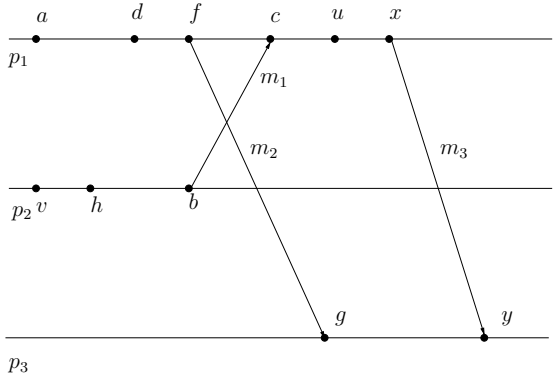

FIGURE 2 – *Détermination des relation de causalité*

   Donner les relations de causalité

   (b) A un événement e trois ensembles d'événements sont associés :
   — $PASSE(e)$: ensemble des événements antérieurs à e dans l'ordre causal (e appartient à cet ensemble)
   — $FUTUR(e)$: ensemble des événements postérieurs à e dans l'ordre causal (e appartient à cet ensemble) ;

— $CONCURRENT(e)$: ensemble des événements concurrents avec e.

Notation : $e||e'$ deux événements e et e' sont concurrents

Sur la figure 2 donner $PASSE(u), FUTUR(u), CONCURRENT(u)$.

Délivrance causale : La délivrance causale assure que si l'envoi du message m_1 par le site S_i à destination du site S_k précède (causalement) l'envoi du message m_2 par le site S_j à destination du site S_k, le message m_1 sera délivré avant le message m_2 sur le site S_k. Formellement,

$$send_i(m_1, k) \rightsquigarrow send_j(m_2, k) \rightarrow del_k(m_1) \rightsquigarrow del_k(m_2)$$

6. Considérons maintenant la notion de vectorielle proposé par Mattern.
   — Chaque site i gère un vecteur d'entiers de n éléments (une horloge HV_i avec n le nombre de sites ;
   — Chaque message m envoyé est estampillé (daté) (EV_m) par la date de son évènement :
   — Si un événement est locale, alors $HV_i[i] \leftarrow HV_i[i] + 1$;
   — Si l'événement est l'envoi d'un message m alors

$$\begin{cases} HV_i[i] \leftarrow HV_i[i] + 1 \\ EV_m \leftarrow HV_i \end{cases}$$

   — Si l'événement est la réception du message m alors

$$\begin{cases} HV_i[i] \leftarrow HV_i[i] + 1 \\ HV_i[j] \leftarrow \max(HV_i[j], EV_m[j], j \neq i \end{cases}$$

   (a) Donner l'évolution des horloges vectorielles pour le graphe de la figure 3.

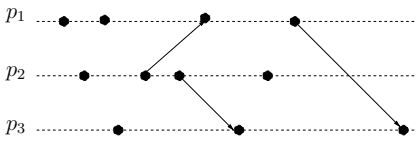

FIGURE 3 – *Sur les horloges vectorielles*

   (b) Montrer que la valeur de la $i$ème composante de EV_e correspond au nombre d'évènements du site S_i appartenant au passé de e

$$\forall i, EV_e[i] = card(\{e' : e' \in S_i \wedge e' \rightsquigarrow e\})$$

   (c) Compléter la relation d'ordre suivante :

$$EV_e \preceq EV_{e'} \text{ ssi}$$

   (d) Montrer que $e \rightsquigarrow e'$ ssi $EV_e \preceq EV_{e'}$.

   (e) Montrer que $e||e'$ ssi $EV_e||EV_{e'}$ avec $e||e'$ désigne deux événements e et e' concurrents.

   (f) Cet ordre permet-il la délivrance causale ?

7. Sur l'estampille matricielle :
   — Chaque site i gère une horloge (notée HM_i) matricielle ($n \times n$) avec n le nombre de sites
   — Chaque message m envoyé est estampillé (daté) (EM_m) par la valeur courante de HM_i
   — Que signifie $HM_i[j, k]$?
      — $HM_i[j, k]$ = nombre de messages issus de p_j vers p_k dont p_i a connaissance ;
      — $HM_i[i, i]$ correspond au nombre d'évènements locaux du site i
   — Comment HM_i est modifiée
      — Evènement local à p_i : $HM_i[i, i] + +$
      — Emission de m vers p_j : $HM_i[i, i] + +; HM_i[i, j] := HM_i[i, j] + 1$ et m est estampillé : $EM_m = HM_i$
      — Réception de (m, EM_m) en provenance de p_j.

   (a) Donner les modifications de HM_i lors de la réception de (m, EV_m) en provenance de p_j.

   Rappel : nous voulons la délivrance causale.

   (b) Considérons un système contenant trois sites. Tous les sites possèdent des horloges logiques matricielles. Supposons que l'horloge matricielle HM_3 du site 3 en B est

$$HM_3 = \begin{pmatrix} HM_3[1,1] & HM_3[1,2] & HM_3[1,3] \\ HM_3[2,1] & HM_3[2,2] & HM_3[2,3] \\ HM_3[3,1] & HM_3[3,2] & HM_3[3,3] \end{pmatrix} = \begin{pmatrix} 6 & 2 & 2 \\ 1 & 5 & 1 \\ 1 & 2 & 7 \end{pmatrix}$$

      i. Comment sait-on le nombre d'événements interne d'un site i ?

      ii. A quoi correspond l'élément de l'horloge matricielle $HM_3[3, 1]$, pour le site 3 ?

      iii. Même question pour les éléments $HM_3[1, 3], HM_3[2, 3]$ pour le site 3 ?

      iv. Que peut déduire le site 3 par rapport aux éléments $EM_m[1, 3], EM_m[2, 3]$?

      v. Le site 3 peut-il délivrer le message m (délivrance causale) ? Justifier votre réponse.

vi. B reçoit

$$EM_m = \begin{pmatrix} 8 & 2 & 3 \\ 2 & 9 & 2 \\ 1 & 1 & 3 \end{pmatrix}$$

. Que doit-on faire ?

vii. Donner la valeur des horloges matricielles pour la figure 4.

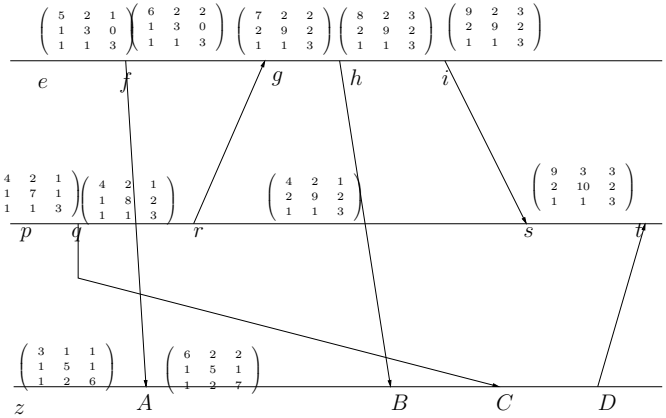

FIGURE 4 – *Début de datation avec l'horloge matricielle : attention à la délivrance des messages*

viii. Avez-vous détecté une erreur dans la matrice en e, p ou z ?

Exercice 3 – La diffusion

Nous considérons le problème de la diffusion d'un sommet r_0 vers tous les autres.

`Si` $i = r_0$ `alors Envoyer(`$< message, v >$`) avec` $v \in voisin_{r_0}$;

`Lors de la réception (`$< message, v >$`) de` i ;
`Envoyer(`$< message, v >$`) avec` $v \in voisin_j \backslash \{i\}$;

1. Quel est l'inconvénient de l'algorithme précédent ?
2. Proposer une solution pour éviter la non terminaison.
3. Donner une borne inférieure en fonction de n pour tout algorithme de diffusion.

4. De même en fonction de m.
5. Donner la borne supérieure de votre algorithme.

Exercice 4 – Calcul de la barrière causale

Lorsqu'un site P_i reçoit un message M avec de telles données, sous quelles conditions peut-il délivrer M en prenant en compte les contraintes de précédence immédiates ?
La condition est :

$$\forall (k, d) \in CB_M : d \leq DEL_i[k]$$

Cela exprime que tous les prédécesseurs immédiats de M ont déjà été délivrés. Le protocole de diffusion décrit pour un site i quelconque est alors :

Au départ, $CB_i := \emptyset$; $num\_envoi_i := 0$;

`Procédure diffuser(`M`)`
  $num\_envoi_i := num\_envoi_i + 1$;
  `Pour tout (`$x_i \in V$`) faire Envoyer(`$< M, num\_envoi_i, CB_i >$`) à` x_i ;
  $CB_i := (i, num\_envoi_i)$;

`Lors de la réception de` $< M, num_M, CB_M >$ `de` P_j
  `Attendre(`$\forall (k, d) \in CB_M : d \leq DEL_i[k]$`)` ;
  $DEL_i[j] := num_M$;
  $CB_i := (CB_i - CB_M) \cup \{(j, num_M)\}$; `(*)`
  `Délivrer(`M`)` ;

(*) A faire de manière atomique.
NB : le stockage (la destruction) des messages lors (après) l'attente n'est pas indiqué(e) ici mais est implicite.

Scénario : Le site numéro 1 procède à deux envois successifs. Ensuite le site 1 procède à un nouvel envoi, et en parallèle le site 2 effectue également un premier envoi. Les messages arrivent sur le site 2 (resp. 1) après l'émission des messages d'envois. Ensuite le site 1 procède à un dernier envoi.

Exercice 5 – Minimisation des échanges de messages pour les horloges vectorielles

Nous souhaitons minimiser la quantité d'information qui circule sur le réseau. Pour cela, nous proposons une gestion incrémentale des horloges vectorielles.

— Dans un message envoyé par j à i n'inclut que les composantes de son horloge qui ont été modifiées depuis le dernier message qu'il a envoyé à i.
— L'estampille du message est une liste de couple (id, val) tels que :
  — $VC_j[id]$ a été modifié depuis le dernier message envoyé par j à i.
  — La valeur courante de $VC_j[id]$ est val.
— Pour tout couple (id, val) contenu dans le message de j :

$$VC_i[id] = \max(VC_i[id], val)$$

1. Appliquer ce principe sur la figure 5.

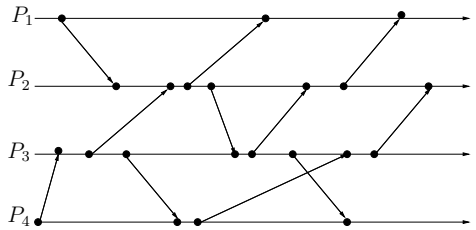

FIGURE 5 – *Diagramme à compléter*

2. Implémentation de la méthode décrite précédemment.

   La question importante porte sur quelle information à maintenir sur chaque site. Il semble que l'information qui permet à j de savoir quelles composantes de son vecteur d'horloge à envoyer à i porte sur l'horloge vectorielle de dernière émission vers i.

   Nous considérons deux vecteurs LS et LU définis ci-après :
   — Gestion des dates d'émission : vecteur (Last Sent) LS avec $LS_j[i]$ = valeur de $VC_j[j]$ lors du dernier envoi de j à i.
   — Gestion des dates de modifications : vecteur LU (Last Update) avec $LU_j[k]$ = valeur de $VC_j[j]$ lors de la dernière modification de $VC_j[k]$.
   — Identification des valeurs émises : j envoie à i toutes les composantes k de VC_j telles que :

$$LU_j[k] > LS_j[i]$$

   sous la forme $(k, VC_j[k])$.

   On supposerons que les canaux sont FIFO. Donner l'évolution des vecteurs LU et LS pour la figure 6 sur le site 3 en partant de l'état e_1 (avant l'émission d'un message vers le site 2).

Index	VC_3	LU_3	LS_3
1	3	2	10
2	10	5	6
3	10	10	
4	4	4	7
5	20	9	3

TABLE 1 – Etat 1

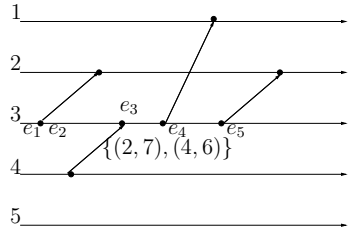

FIGURE 6 – *Diagramme à compléter*

Exercice 6 – Asservissement mutuel d'horloges logiques pour deux sites

Le problème est le suivant : on veut contraindre les deux processus de telle faÃ§on qu'à tout instant chacun d'eux ne puisse pas émettre plus de δ requêtes d'avance par rapport à l'autre. Vue l'association des valeurs d'horloges aux requêtes, le problème consiste à maintenir invariante la relation suivante : $|h_i - h_j| \leq \delta$.

Cette relation est vraie : $h_i = h_j = 0$.

Les messages relatifs à ces échanges vont être véhiculés par des lignes que nous supposons dotées du comportement suivant : les messages peuvent être dupliqués, déséquencés et perdus.

Pour cela on introduit à tout instant dans chaque processus P_i une variable locale $\max_i$ qui lui indique à tout instant la valeur jusqu'à laquelle il peut faire croître son horloge h_i sans que cela n'invalide l'invariant ; la condition associée à l'appel à dater est donc $h_i < \max_i$.

Conditions initiales

—

`Lors de l'appel à dater`
`Si` $h_i < \max_i$
$h_i := h_i + 1$
`Envoyer(<` h_i `>) à` P_j

1. Donner la procédure de réception.
2. Montrer que nous avons bien $|h_i - h_j| \leq \delta$ tout le long du processus.
3. Nous supposons que les messages peuvent être perdus et dupliquées mais ne peuvent être déséquencés. Simplifier le code de la réception d'un message.
4. Comment peut-on limiter la croissante des horloges ? Soit x la valeur envoyée du site h_i à un instant donné. Donner un encadrement de x par rapport δ et h_i. Donner maintenant l'algorithme qui prend en compte le non dérive des horloges.

Université de Montpellier 2018/2019
Master Durée: 7.5h

Algorithme distribué
TD – Séance n° 2

Exercice 1 – Exclusion mutuelle dans les arbres

Nous utiliserons les messages $REQUEST$ pour demander à utiliser la ressource et le message $JETON$ qui représente la ressource (ou l'autorisation de l'utiliser). L'idée de ce protocole est que chaque site p a toujours un «pointeur» ($Racine_p$) qui est dirigé vers le jeton dans l'arbre.

```
Procédure acquisition
    Si (non Avoir_jeton_p) alors
        Si (File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p;
        Ajouter(Request_p,p);
        Attendre(Avoir_jeton_p);
    En_SC_p := vrai;

Procédure libération
    En_SC_p := faux;
    Si (non File_vide(Request_p)) alors
        Racine_p := Défiler(Request_p);
        Envoyer(< JETON >) à Racine_p;
        Avoir_jeton_p := faux;
        Si (non File_vide(Request_p)) alors
            Envoyer(< REQUEST >) à Racine_p;

Lors de la réception de < REQUEST > de q
    Si (Avoir_jeton_p) alors
        Si (En_SC_p) alors Ajouter(Request_p,q);
        Sinon
            Racine_p := q;
            Envoyer(< JETON >) à Racine_p;
            Avoir_jeton_p := faux;
```


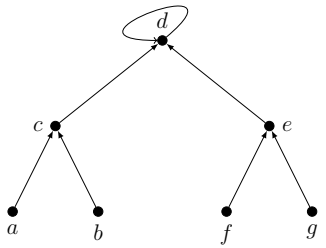

FIGURE 1 – Anti-arborescence.

```
        Sinon
            Si (File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p;
            Ajouter(Request_p,q);

Lors de la réception de < JETON > de q
    Racine_p := Défiler(Request_p);
    Si (Racine_p = p) alors Avoir_jeton_p := vrai;
    Sinon
        Envoyer(< JETON >) à Racine_p;
        Si (non File_vide(Request_p)) alors Envoyer(< REQUEST >) à Racine_p;
```

La file *Request* sert à ordonner les demandes en chaque sous-arbre. Il est important de noter les points suivants dans l'algorithme.

Appliquer cet algorithme sur la figure 1, avec le scénario suivant :

— e fait une demande.
— g fait une demande
— b fait une demande et arrive avant la demande de g.
— a fait une demande qui arrive après celle de g.

Exercice 2 – L'algorithme de Ricart-Agrawala

Dans l'algorithme suivant, chaque site désirant la ressource va demander la permission à tous les autres. On départage les conflits en étiquetant chaque demande par l'heure (logique) à laquelle on a fait cette demande. Les demandes les plus anciennes sont les plus prioritaires.

```
Procédure acquisition
```

$demandeur_i := vrai$;
$horloge_i := horloge_i + 1$; $heure\_demande_i := horloge_i$;
$rep\_attendues_i := n - 1$;
`Pour tout` $(x_i \in V - \{i\})$ `faire`
`Envoyer(`$< REQUEST, heure\_demande_i >$`) à` x_i;
`Attendre(`$rep\_attendues_i = 0$`);`

`Procédure libération`
...
`Procédure reception de request`
...
`Lors de la réception de` $< REPONSE >$ `de` j
$rep\_attendues_i := rep\_attendues_i - 1$;

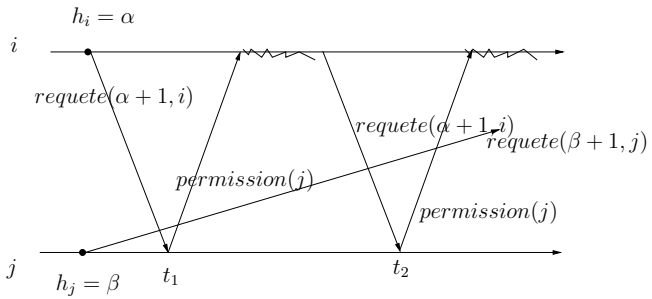

FIGURE 2 – *Effets du déséquencement*

1. Donner le rôle des variables $heure\_demande_i$, $rep\_attendues_i$, $demandeur_i$ et $differe_i$.
2. Donner la procédure de libération et de reception.
3. Donner la complexité en nombre de messages.
4. Montrer que l'algorithme de Ricart-Agrawala respecte le principe de sûreté c'est à dire que il existe au plus un site en section critique.
5. Montrer que l'algorithme de Ricart-Agrawala respecte le principe de vivacité c'est à dire que qu'un site pourra toujours rentré en section critique.
6. Donner l'intervalle de temps de non utilisation de la section critique sachant que les messages ont un temps de transit borné par T.

7. Avons-nous utilisé pour la démonstration que les canaux sont fifos ou non ? Une question se pose : Quelle incidence peut avoir le caractère non fifos des canaux sur le comportement de l'algorithme ? Pour répondre à cela nous allons nous limiter à 2 sites i et j qui obéissent au comportement suivant :

   `Boucler sur Acquérir ;`
   $< utilisationdelasectioncritique >$
   $librer$;
   `Fin de boucle`

   et nous supposons que $h_i = \alpha$ et $h_j = \beta$ avec $(\alpha, i) < (\beta, j)$. Une exécution possible est décrite par la figure 2 (la zone hachurée indique que le site correspondant est en section critique). Dans ce scénario les deux sites ont initialement fait des requêtes estampillées respectivement $(\alpha + 1, i)$ et $(\beta + 1, j)$ en t_1. A la réception de la requête, le site j renvoie sa permission qui double la requête qu'il avait envoyée précédemment. A la réception du message *permission*, le site i rentre en section critique, puis en sort et invoque à nouveau *acquerir*.
   (a) Quel est la valeur de l'horloge h_i à cet instant t_2 (que vaut x ? Quel conséquence ?
   (b) Que se passe-t-'il concernant l'horloge de i quand le site i reçoit la requête de j ?
   (c) Un site peut-il rentrer un nombre de fois arbitraire en section critique alors qu'un autre site en a fait la demande ? Ce nombre peut-il être non borné ?
   (d) Donner une exécution dans le cas où les canaux seraient fifos.
   (e) Comment obtenir, avec des canaux non fifos, un ordre de satisfaction des demandes analogues à celui que l'on obtiendrait avec des canaux fifos sans utiliser des messages supplémentaires ?
8. Quel est le défaut de cet algorithme concernant les demandes de permissions.
9. On suppose maintenant que le temps de transfert est connu. Soit δ le majorant sur les temps de transfert entre deux sites.
   (a) Modifier l'algorithme de Ricart-Agrawala pour prendre en compte cette nouvelle hypothèse.
   (b) Evaluer le nombre de messages par utilisation de la section critique.
10. On a fait l'hypothèse implicite d'un maillage complet pour les communications c'est-à-dire d'un canal bidirectionnel entre tout couple de sites. Comment adapter l'algorithme si le maillage entre les sites est un anneau unidirectionnel ? Montrer que le nombre de messages pour une utilisation de la section critique est $\frac{n(n+1)}{2}$. Proposer une idée afin que le nombre de messages soit n pour une utilisation de la section critique.

11. La valeur des horloges peut croître indéfiniment. Donner l'intervalle de temps entre deux demandes pour un même site i. Soit h_i l'horloge d'un site i. Quelle est la valeur de l'intervalle des horloges des autres sites ? Quel est la longueur de l'intervalle. Comment éviter la dérive des horloges ?

Exercice 3 – L'algorithme de Carvalho et Roucairol Nous allons voir maintenant l'algorithme de Carvalho et Roucairol qui peut-être vu comme une amélioration du précédent. Considérons pour illustrer cela la situation suivante :deux sites, i et j tels que i veut utiliser la section critique plusieurs fois de suite alors que j n'est pas intéressé par celui-ci et reste donc dans l'état *dehors*. Avec l'algorithme précédent le site i demande la permission de j à chaque nouvelle invocation de l'opération acquérir. Le principe utilisé ici est le suivant : puisque j a donné sa permission à i, ce dernier la considère comme acquise jusqu'à ce que j demande à i sa permission ; dans la situation décrite le site i ne demande donc qu'une fois la permission à j.

1. Donner l'intervalle du nombre de messages nécessaires pour une utilisation de la section critique.
2. Donner une façon d'implémenter ce principe.
3. Montrer que la sûreté est garantie.
4. Montrer que la vivacité est garantie.
5. Donner l'algorithme.
6. Donner l'intervalle de temps de non utilisation de la section critique sachant que les messages ont un temps de transit borné par T.
7. Est-il possible de borner le domaine des valeurs d'horloges ?
8. Est-ce que le délai borné apporte quelque chose ?

Exercice 4 – Exclusion mutuelle sur un graphe complet Algorithme de Raymond

On suppose que l'on a un réseau logique complet à n sommets V. Pour tous les utilisateurs, une *ressource* à $M < n$ entrées est disponible. Nous proposons de définir un protocole d'accès à cette ressource. Voici une partie de ce protocole pour le sommet i. Les principales variables sont :

$permatt_i : array[1 \dots n]$ d'entiers positifs ;
$differe_i : array[1 \dots n]$ d'entiers positifs ;
$etat_i := dehors \in \{demandeur, dedans, dehors\}$;
(* Les deux tableaux sont initialisés à 0 *)
h_i est l'horloge associé au site i

`Procédure Demander-SC`
$etat_i := demandeur$;
$last_i := h_i + 1$;
$nbperm_i := 0$;
`Pour tout` $(j \in V - \{i\})$ `faire`
`Envoyer(`$< REQUEST, last_i, i >$`)` `à` j;
$permatt_i[j] := permatt_i[j] + 1$
`Attendre(`$nbperm_i \geq n - M$`)`;
$etat_i := dedans$;

< SECTION CRITIQUE >

`(* Phase de libération *)`
$etat_i := dehors$;
`Pour tout` $(j : j \neq i, differe_i[j] \neq 0)$ `faire`
`Envoyer(`$< PERMISSION, i, differe_i[j] >$`)` `à` j;
$differe_i[j] := 0$;

`Lors de la reception de` $< REQUEST, k, j >$
...

`Lors de la reception de` $< PERMISSION, j, x >$;
$permatt_i[j] := permatt_i[j] - x$;
`Si` $(etat_i = demandeur)$ `alors`
`Si` $(permatt_i[j] = 0)$ `alors` $nbperm_i := nbperm_i + 1$;

1. Si $M = 1$, que retrouve-t'on ?
2. Donner la signification de $permatt_u[v]$, $differe_u[v]$ et $nbperm_u$
3. Pourquoi dans la procédure `Demander-SC` attend on $n - M$ permissions avant d'entrer en section critique ? Que représente la donnée associée au message $< REQUEST, ., . >$? Comment déterminer des niveaux de priorité grâce à ces données ? Si un sommet u reçoit un message $< REQUEST, ., . >$ d'un sommet v plus prioritaire, et que lui même demande une ressource, que doit il faire ?
4. Donner un exemple à trois sites, pour lequel la procédure $< PERMISSION, j, x >$ avec $x \geq 2$ est utilisée.

5. Ecrire la procédure de réception de $< REQUEST, ., . >$.
6. Montrez (par contradiction) qu'au plus M sommets peuvent simultanément utilisez la ressource.
7. Avec un tel algorithme, un site peut-il attendre infiniment avant d'entrer en section critique ?
8. Donner la complexité concernant le nombre de messages pour l'utilisation d'une ressource.

Exercice 5 – Algorithme de Maekawa et ses variantes Dans l'algorithme de Ricart et Agrawala un site désirant la ressource doit demander la permission à tous les autres sites ce qui entraîne la génération d'un grand nombre de messages. Au lieu de demander un si grand nombre de permissions on peut imaginer que chaque site i a un ensemble de sites S_i à qui il va demander la permission d'entrer en section critique. Lorsque i obtient *toutes* les permissions de S_i il peut utiliser la ressource. Un tel ensemble S_i sera appelé *quorum*.

Nous souhaitons donc obtenir un algorithme réparti « symétrique » dans lequel chaque site joue un rôle équivalent au x autres au sens suivant :

$$
\begin{array}{lll}
(c_0) & i \in R_i & \\
(c_1) & \forall i : & Card(R_i) = K \\
(c_2) & \forall i & i \text{ est contenu dans } D \text{ ensemble } R_j
\end{array}
$$

1. Donner la signification des contraintes c_i.
2. Est-ce que l'algorithme de Ricart-Agrawala respecte les deux contraintes c_1 et c_2. Si oui, avec quelles valeurs de K et de D. Si nous utilisions l'algorithme de Ricart-Agrawala avec les valeurs données ci-avant, quel est le nombre de messages nécessaires pour accéder à la section critique ? Conclusion ?
3. Donner la règle d'intersection, pour qu'on ait la propriété de sureté.
4. Construire des quorums pour le cas où $K = 2, 3$.
5. On constate cependant qu'il peut y avoir des blocages si on ne met en place que ce mécanisme. Il faut rajouter un ordre de priorité pour s'assurer qu'en cas de multiples demandes simultanées au moins un site puisse recevoir toutes les permissions. Que va t'on utiliser ?
6. Toutes ces propriétés sont indispensables pour assurer la sûreté et la vivacité. Cependant, d'autres critères peuvent être pris en compte. Parmi ceux ci on peut distinguer la *charge induite par le mécanisme pour chaque site* ainsi que la performance en nombre de messages. Le premier point est relatif à la charge de travail que chaque site du système doit accomplir pour la communauté, même s'il n'a pas besoin de la ressource. Le deuxième point est la complexité de l'algorithme. Nous voudrions que chaque site i fasse partie de D quorums S et que pour tout j, $|S_j| = k$. Dans ce cas, la complexité de l'algorithme est en $O(k)$. Il faudrait donc minimiser k et D. Donner le nombre maximum de quorum qui peuvent être construits.
7. Construire effectivement des quorums avec ces propriétés est difficile (plans projectifs). Par contre, si $n = p^2$ il est facile de construire n quorums avec $\sqrt{n} = p$ sites dans chacun d'eux.
8. Dans ce qui suit nous décrivons les grandes lignes de l'algorithme d'allocation de ressources. Avant de mettre en place l'algorithme il faut au préalable construire les quorums et les affecter aux sites. Chaque site i connaît son quorum S_i. Chaque fois que i demande la ressource, il demande la permission, estampillée par son horloge, à tous les sites de S_i. Lorsqu'un site i reçoit une demande de permission de la part de j on peut être dans un des trois cas.
   — i ne demande pas la ressource. Il envoie sa permission à j.
   — i a demandé la ressource. Si sa demande a une estampille plus ancienne que celle de j il lui envoie un message pour lui dire non, sinon il lui donne la permission.
   — i a déjà donné sa permission à un site k dont l'heure de la demande avait une estampille plus récente que celle de j. Dans ce cas, i va essayer de récupérer la permission qu'il avait donné à k pour la donner à j. Si k a reçu un message de refus de permission, il redonne la permission à i qui va la donner à j.

   Expliquer pourquoi il n'y a pas de blocage et que la vivacité est assurée.
9. Donner l'algorithme.
10. Donner un scénario pour lequel il existe un interblocage avec $R_1 = \{1, 2\}$; $R_2 = \{2, 3\}$ et $R_3 = \{1, 3\}$, et chaque site gère une seule permission (pour entrer en SC il faut deux permissions).
11. Comment résoudre le problème de l'interblocage ?
12. Dans quel cas il y a déséquencement ?,
13. Donner le délai pour lequel la section critique bien que demandée n'est pas utilisé ?
14. Quel est le nombre de messages relatifs à une utilisation de la section critique ?

Exercice 6 – Un algorithme mixte dû à Singhal

Dans les algorithme à permissions d'arbitres l'interblocage peut nécessiter, comme nous venons de la voir, la préemption de permissions. Peut-on dans cette classe d'algorithmes, une permission une fois attribuée, ne jamais la retirer ?

1. Proposer une idée en vous basant sur les algorithmes précédents.
2. Appelons par CR_i l'ensemble $CR_i = \{j \text{ tel que } i \in R_j\}$. Quel est la signification ? Peut-on avoir $R_i \cap CR_i \neq \varnothing$?
3. Voici une partir de l'algorithme. Compléter les parties manquantes (parties *retourperm* et *requte*) et expliquer le fonctionnement de l'algorithme.

```
Lors d'un appel à acquérir
    etat_i = demandeur;
    h_i = h_i + 1; last_i := h_i;
    attendus_i := R_i;
    ∀j ∈ R_i : envoyer requête(last_i, i) à j;
    insérer(last_i, i) dans file_i (la file est triée α)
    attendre (R_i = ∅) et ((last_i, i) est en tête de file_i; (β)
    etat_i = dedans;

Lors d'un appel à libérer :
    supprimer (last_i, i) de file_i
    etat_i = dehors;
    ∀j ∈ R_i : envoyer retourpermission(h_i, i) à i;
    ∀m tel que (x, m) ∈ file_i : envoyer permission(h_i, i) à m (γ)

Lors de la reception de permission(k, j);
    h_i := max(h_i, k)
    attendus_i := attendus_i − {j}
```

4. Quel est le rôle de $file_i$?
5. Comment pouvez-vous définir les ensembles R_i ?
6. Quel est le délai de non utilisation de section critique ?

Exercice 7 – L'algorithme de Naimi et Tréhel Dans cet exercice nous nous plaçons dans le cas d'une ressource à une seule entrée (au plus un sommet peut utiliser la ressource à un instant donné). Réaliser l'exclusion mutuelle en réparti revient à proposer une mise en oeuvre distribué de l'objet informatique qu'est une file des sites. plusieurs solutions sont envisageables allant de la duplication de l'objet sur chaque site (avec un protocole qui assure la cohérence mutuelle des copies) à son partitionnement ; L'algorithme que nous présentons maintenant appartient à cette dernière catégorie. Une file va donc être implémentée en plaçant sur chaque site i un pointeur $suivant_i$, qui, s'il est différent de nil, indique le site successeur de i dans la file d'attente. La file étant définie deux problèmes se posent :

1. Comment un site sait-il qu'il est en tête de la file ?
2. Comment un site qui n'est pas dans la file peut-il venir se placer en queue de file ?

C'est la réponse à ces deux questions qui va définir l'algorithme.

1. Proposez un système qui permet de savoir si un site est en tête de file ?
2. Proposez une structure qui permet de se reconfigurer dynamiquement et qui permet de savoir qui se trouve en dernier (dernier élément de la file).
3. Proposez un protocole de gestion de la structure
4. Nous avons raisonné jusqu'à maintenant comme si la file n'est pas vide. Proposez une solution simple pour éviter que la file soit vide.
5. Donner un exemple d'utilisation de l'algorithme où s est la racine de l'arbre possèdant le jeton et p et q désirent entrer dans la file. p est traité avant q et p reçoit le jeton de s. Puis s veut rentrer en section critique.
6. Ecrire la phase de libération.
7. Donner un exemple d'exécution de cet algorithme. A qui correspondent les variables $pere_p$ et $suivant_p$?
8. Pourquoi un sommet demandant le jeton l'obtiendra t'il au bout d'un temps fini (s'il n'y a pas de panne) ?
9. Quel est le nombre de messages échangés dans le pire des cas lors d'une demande ?
10. Est-ce que pour un site i son père diffère durant l'exécution ? Qu'est-ce que cela implique pour tous les sites ?

Voici le protocole que nous proposons. Le code est décrit pour le sommet p.

```
Procédure Demander-SC;
    demandeur_p := vrai;
    Si (pere_i ≠ nil) alors
        Envoyer(< REQUEST >) à pere_p;
        pere_p := nil;
        Attendre(AvoirJeton_p);

  <SECTION CRITIQUE>

(* Phase de libération *)
```

Lors de la réception de $< REQUEST >$ de q (q demandeur);
    ($Demandeur_p$) alors
        ($Suivant_p = nil$) alors $suivant_p := q$
    Si ($pere_p \neq nil$) alors
        Envoyer($< REQUEST >$ à $pere_p$
    Sinon Si ($AvoirJeton_p$) alors
        $AvoirJeton_p := faux$
        Envoyer($< TOKEN >$) à q
    $pere_p := q$

Procédure recevoir $< TOKEN >$ de q;
    $AvoirJeton_p := vrai$;

Initialisation : choisir un sommet S quelconque et

$AvoirJeton_S := vrai$; Pour tout $X \neq S$, $AvoirJeton_X := faux$;
Pour tout sommet Y, $pere_Y := S$; $suivant_Y := NIL$; $Demandeur_Y := faux$;

Université de Montpellier 2017/2018
HMIN213 Durée: 4.5h

Election
TD – Séance n° 1

Exercice 1 – Anneau, Algorithme de Chang-Roberts

Nous proposons maintenant un autre algorithme pour l'élection dans un cycle unidirectionnel. Chaque site i maintient un booléen $participant_i$ qui est initialisé à $Faux$. Le code pour le site i est le suivant.

```
En cas de réveil spontané
    participant_i := Vrai;
    Envoyer(< ELECTION, val_i >);

Lors de la réception d'un message < ELECTION, V >
    Si (V > val_i) alors
        participant_i := Vrai;
        Envoyer(< ELECTION, V >);
    Si (V < val_i) et (non participant_i) alors
        participant_i := Vrai;
        Envoyer(< ELECTION, val_i >);
    Si (V = val_i) alors Envoyer(< ELU, val_i >);

Lors de la réception d'un message < ELU, V >
    Le gagnant est le site V;
    participant_i := Faux;
    si (val_i ≠ V) alors Envoyer(< ELU, V >);
```

1. Proposez une exécution dans la configuration de la figure 1. Qui détecte le gagnant et comment est-il propagé ?
2. Est-ce que cet algorithme fonctionne en mode asynchrone ?
3. Supposons que tous les sites se réveillent simultanément. Evaluez la complexité en nombre de messages échangés dans les configurations particulières suivantes en mode asynchrone et synchrone :

— Les valeurs sont ordonnées croissantes dans le sens du cycle.
— Les valeurs sont ordonnées décroissantes dans le sens du cycle.

4. Nous allons étudier la complexité en moyenne.
   (a) Montrer qu'un site actif possédant la $j$ième plus grande identité parcours en moyenne n/j liens.
   (b) Avec x initiateurs, donner la formule.
   (c) Conclure que la complexité en $O(n \log n)$.

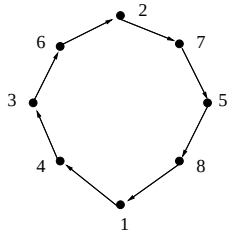

FIGURE 1 – Une configuration dans C_8.

Exercice 2 – Anneau bi-directionnel, Algorithme de Hirschberg et Sinclair (80)

Nous allons maintenant travailler sur des cycles bi-directionnels (on peut envoyer des messages dans les deux sens sur le cycle). L'algorithme proposé marche par *phase* et est basé sur le filtrage. Les sites peuvent être dans deux états $actif$ ou $perdant$. Lorsqu'un site reçoit un message contenant une valeur plus grande que la sienne il devient $perdant$. Sinon il reste $actif$.

A la phase i, $(i = 0, 1, \dots)$ les sites encore actifs envoient leur valeur à une distance de 2^i d'eux. Lorsqu'un site $actif$ reçoit un tel message il devient $perdant$ si la valeur reçue est supérieure à la sienne sinon il passe à la phase suivante.

Nous allons supposer ici que le réseau est *synchrone par phase*. C'est-à-dire que les sites encore actifs au début de la phase i commencent tous *en même temps* à exécuter les opérations de cette phase.

1. Donner un exemple d'exécution sur la graphe de la figure 1
2. Après la phase i, quelle est la distance minimale entre deux sites $actifs$?
3. Combien y a t'il de phases ? En déduire le nombre de messages échangés pendant l'algorithme.
4. Qui connaît le gagnant ?

5. Donner l'algorithme.
6. Quel est l'inconvénient sur le nombre de messages ?

Exercice 3 – L'algorithme de Franklin L'algorithme proposé dans cet exercice est un raffinement du précédent.. Le code pour un site i est :

```
Lors du réveil du site i
    etat_i := actif;
    Tant que (etat_i = actif) faire
        Envoyer(val_i) à droite;
        Envoyer(val_i) à gauche;
        Attendre un message (V_d) venant de la droite;
        Attendre un message (V_g) venant de la gauche;
        V := max{V_d, V_g};
        Si (V ≥ val_i) alors etat_i := passif;
    Si (val_i = V) alors
        gagnant_i = i;
        Prévenir les autres sites
    Sinon faire suivre tous les messages.
```

1. En quoi cet algorithme est-il différent du précédent ? Peut on remplacer $\geq$ par $>$?
2. Combien de messages sont générés ?

Exercice 4 – Anneau Retour sur les cycles unidirectionnels. On propose l'algorithme suivant pour le site i.

```
Réveil spontané du site i ou réception d'un message pour la première fois
    Envoyer(< 1, val_i >);
    etat_i := actif;
    max_i = val_i;

Lors de la réception du message < 1, V >
    Si (etat_i = actif) alors
        Si (V ≠ max_i) alors
            Envoyer(< 2, V >);
            voisin_i := V;
        Sinon max_i est élu, prévenir les autres;
```



```
    Sinon Envoyer(< 1, V >);

Lors de la réception du message < 2, V >
    Si (etat_i = actif) alors
        Si (voisin_i > V) et (voisin_i > max_i) alors
            max_i := voisin_i;
            Envoyer(< 1, max_i >);
        Sinon etat_i := passif;
    Sinon Envoyer(< 2, V >);
```

1. A quoi servent les variables locales max_i, $voisin_i$ et les deux messages ? Par qui peut être proclamé le résultat ?
2. Quelle est la complexité en nombre de messages échangés ?
3. Appliquer l'algorithme sur le cycle donné par la figure 1.

Exercice 5 – Election dans un arbre.

On suppose que l'on a un réseau d'ordinateurs connectés en *arbre* (voir exemple figure 2). Chaque canal est 'FIFO'. Les sommets (ou sites) ne communiquent directement qu'avec leurs *voisins* immédiats. Chaque sommet a un *identificateur* unique.

1. Ecrire un algorithme réparti qui :
   - "Réveille" tous les sommets à partir d'au moins un *initiateur*.
   - Calcule la plus petite étiquette de l'arbre (le gagnant étant le sommet qui possède cette étiquette). Pour cela :
     - Faire un parcours des feuilles vers l'intérieur de l'arbre en calculant la plus petite étiquette au fur et à mesure.
     - Propager le résultat de l'élection de voisin en voisin. Si un sommet reçoit son étiquette, il se déclare *vainqueur*, sinon *perdant*.

   Pour cela vous pourrez utiliser les variables suivantes :

   Un site p a les variables locales suivantes :
   $reveille_p$ booléen initialisé à $Faux$,
   $voisins\_reveilles_p$ entier initialisé à 0,
   rec_p tableau de $|voisins_p|$ booléens initialisés à $Faux$,
   val_p la valeur pour l'élection, v_p, initialisée à la valeur val_p,
   $etat_p \in \{neutre, gagnant, perdant\}$, initialisé à $neutre$.

Vous devez commenter votre algorithme.

2. Proposez une exécution possible sur l'arbre de la figure 2.
3. Quelle est la complexité en nombre de messages échangés ?
4. Pourquoi commencer par les feuilles ?
5. Que faire si l'on perd l'hypothèse de l'identificateur unique ?

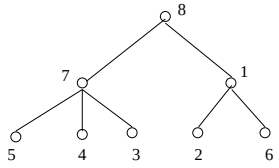

FIGURE 2 – Un réseau en arbre.

Exercice 6 – Borne inférieure pour un réseau quelconque

Nous considérons un graphe quelconque. Nous cherchons à déterminer une borne inférieure pour tout algorithme d'élection basé sur la comparaison des identités. A votre avis quelle est cette borne ? **Aide :** Aidez-vous des résultats obtenu sur l'anneau.

Exercice 7 – Un algorithme asynchrone dans un réseau complet : algorithme de Humblet

Dans le but de développer des algorithmes efficaces dans le cas d'un graphe complet, nous allons utiliser la technique d'acquisition de territoire.

Dans la procédure d'acquisition chaque candidat essaye de capturer un voisin à chaque itération ; en utilisant un message du type *capture* contenant son identité et le nombre de sommets déjà capturé avec la variable $stage(x)$. Si la tentative est un succès, le voisin attaqué devient capturé, et le sommet candidat entre dans une nouvelle itération sinon le candidat devient passif. Un sommet capturé retient l'identité, et la valeur de la variable $stage(x)$ et le lien de son maître (l'identité de celui qui l'a capturé).

1. Remplissez les spécifications suivantes :

   Nous supposons qu'un candidat x envoie un message du type *capture* à y :

   (a) Si y est candidat :

      i. Si $stage(x) > stage(y)$, alors conclure sur l'attaque.

      ii. Si $stage(x) = stage(y)$, donner les états possibles pour x et y.

      iii. Si $stage(x) < stage(y)$, alors conclure sur les états de x et de y

   (b) Si y est passif, conclure sur le résultat de l'attaque de x sur y.

   (c) Si y est déjà capturé alors x doit défaire le maître de y, noté z avant de capturer y. Dans ce cadre y envoie un message du type Warning à z avec l'identité de x et $stage(x)$.

      i. Si z est candidat. Donner les différents cas de figures.

      ii. Dans le cas contraire donner tous les cas de figures.

   (d) Si l'attaque est un succès, conclure sur les états de x et y sur la valeur de $stage(x)$.

2. Donner l'algorithme maintenant.
3. Applique l'algorithme de Humblet sur le graphe donné par la figure 3.

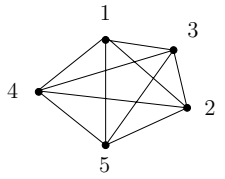

FIGURE 3 – *Graphe complet à 5 sommets*

4. Montrer que l'algorithme répond positivement au problème de l'élection.
5. Que se passe-t-'il quand un sommet a capturé au moins $n/2 + 1$ sommets ?
6. Combien de sommets peuvent être candidat à l'itération i ? Déterminer la taille maximum d'un territoire à l'itération i ? Les territoires sont-ils disjoints ?
7. Combien de messages au maximum sont nécessaires lorsque un sommet procède à une demande capture ?
8. Combien de stages sont nécessaires pour la terminaison de l'algorithme ?
9. Conclure sur la complexité en nombre de messages.
10. Donner un exemple pour illustrer le pire cas concernant la complexité.
11. Donner la complexité en temps.

Exercice 8 – Algorithme de Chan-Chin pour l'élection synchrone dans un graphe complet

Le réseau est ici supposé totalement synchrone : une horloge globale est accessible à tout processus du réseau dont chacune des actions est synchronisée cycle par cycle ; un message envoyé au temps t est donc assuré d'une réception au temps $t + 1/$ Les processus peuvent devenir actifs et déclencher l'exécution de l'algorithme d'élection à tout instant, soit spontanément, soit à la suite de la réception d'un message.

Chaque processus possède trois variables locales $id, etat, phase$:

— id : identité courante du détenteur du processus considéré et à l'initialisation $id = id_i$ du processus P_i.
— un processus actif admet deux états : *candidat* ou *detenu*. Un candidat est son propre détenteur et cherche à détenir les autres pour être élu ; dans le cas contraire être *detenu* signifie que nous ne poursuivons pas le processus d'élection. Un site peut changer plusieurs fois de détenteur mais appartient à un seul détenteur.

L'algorithme est fondé sur le principe suivant : l'unique processus qui finit par détenir tous les autres est le processus élu. *phase* est initialisé à 0 et est incrémentée d'une unité à chaque cycle pair de l'horloge globale ; la variable *phase* mesure le nombre total de processus dont le détenteur est le même que pour le processus courant considéré. Deux catégories de messages peuvent être échangés, les messages $(phase, id)$ et les messages $purge$.

A chaque incrémentation de phase, au début de tout cycle pair d'horloge, un candidat tente de doubler le nombre de ses détenus en envoyant son message $(phase, id)$ le long $2^{phase-1}$ lignes de communications non encore marquées. Ces lignes sont alors marquées pour ce processus. Un candidat ne peut continuer comme tel au cycle pair suivant que s'il n'a reçu aucun message ($purge$ entre-temps, sinon il devient détenu. En cas de succès i.e. aucun message $purge$ reçu, le candidat finira par être le seul détenteur de 2^{phase} processus.

Au début de chaque cycle impair, chaque processus P considère son couple maximal (du point de vue lexicographique) $(phase^*, id^*)$ déjà reçu. En réponse à tout autre message du type $(phase, id)$, il renvoie un message $purge$. Si le couple maximal reçu $(phase^*, id^*)$ est inférieur au couple courant $(phase, id)$ du processus P considéré, ce dernier envoie aussi un message $purge$. Sinon, le processus d'origine du couple maximal $(phase^*, id^*)$ devient détenteur du processus P : le couple $(phase, id)$ de P devient le couple maximal $(phase^*, id^*)$, P devient ou reste détenu et un message $purge$ est envoyé au précédent détenteur de P. En d'autres termes, tout processus devenant ou restant détenu ne conserve qu'un seul et unique détenteur.

Un processus arrête la procédure d'élection lorsqu'il découvre que son détenteur détient aussi tous les autres processus du réseau. L'élection est donc terminé.

1. Donner l'algorithme.
2. Donner l'exécution de l'algorithme sur la graphe de la figure 3.
3. Donner la complexité dans le pire des cas en nombre de messages.
4. Donner la complexité en temps de l'algorithme.
5. Calculons maintenant la complexité en moyenne en messages :
   (a) Soit α le nombre de processus qui est détenu de manière disjointe par chaque candidat au début de la phase j et β le nombre de messages envoyés par chaque

en début de phase j. Le nombre moyen de candidat en début de phase $j+1$ sera au plus de $1 + \frac{n}{\alpha\beta}$
   (b) Donner le nombre moyen de candidats au début de la phase 1. Au début de la phase 2, et en début de phase $j+1$? Montrer que la borne supérieure est $7n$.
   (c) Conclure sur l'optimalité en moyenne.

Exercice 9 – Election dans un graphe quelconque : l'algorithme YO-YO

1. Finir l'exemple donné en cours en utilisant l'algorithme YO-YO.

Exercice 10 – L'algorithme YO-YO

Terminer l'exemple donné en cours en utilisant le principe de l'élagage à partir du graphe donné par la figure 4

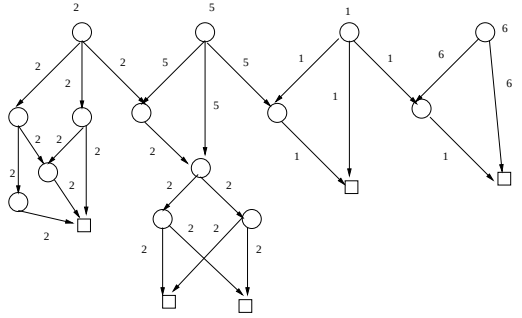

FIGURE 4 – *Graphe acyclique*

Université de Montpellier 2020/2021
Master Durée: 7.5h

Algorithme distribué
TD – Séance n° 3

Exercice 1 – Construction d'une arborescence des plus courts chemins en largeur

Nous souhaitons construire une arborescence

1. Construire une arborescence en partant d'un initiateur noté P_α. Donner la complexité de l'algorithme. Qui détecte la terminaison ?
2. Nous souhaitons construire une arborescence des plus court chemins.
   (a) En se basant sur la question précédente, donner un algorithme qui construit une arborescence des plus court chemins.
   (b) Evaluer la complexité dans le cas d'un graphe complet.
   (c) Qui détecte la terminaison ?
   (d) donner une trace dans le cas d'un graphe complet à 5 sommets avec P_1 m'initiateur. Donner l'arborescence construite.
3. Construire une arborescence des plus court chemins en largeur. Evaluer la complexité
4. Comment étendre ces résultats dans le cas d'un arbre ?

Exercice 2 – Construction d'une arborescence des plus courts chemins en profondeur (suite) : algorithme de Awerbuch

Considérons l'algorithme suivant :

Pour l'initiateur uniquement, à exécuter une seule fois. `Le site` P_α :
$pere_\alpha := \alpha$; `choisir` $q \in voisin_\alpha$
$\forall k \in (voisin_\alpha)$ `Envoyer(`vis`) à` k;
$\forall k \in (voisin_\alpha)$ `recevoir(`ack`) de` k;
$used_p[q] := true$; `envoyer (`tok`) à` q

Pour les autres processus :

A la réception de (tok) **depuis** q_0
`Si` $pere_p = udef$ `alors`
`begin` $pere_p := q_0$;
$\forall k \in (voisin_p \backslash \{pere_p\})$ `Envoyer(`vis`) à` k;
$\forall k \in (voisin_p \backslash \{pere_p\})$ `recevoir(`ack`) de` k;
`end`
`Si` p `est l'initiateur` $\wedge$ $\forall q \in voisin(p) : used_p[q] := true$
`alors decide;`
`Sinon Si` $\exists q \in voisin_p : (i.e. q \neq pere_p \wedge not(used_p[q]))$
`Alors begin Si` $pere_p \neq q_0 \wedge not(used_p[q_0])$ `alors`
`Alors` $q := q_0$
`Sinon Choisir` $q \in voisin(p) \backslash \{pere_p\}$ `avec` $not(used_p[q])$
$used_p[q] := true$, `envoyer(`tok`) à` q
`end`
`Sinon` $used_p[pere_p] := true$; `envoyer(`tok`) à` $pere_p$

A la réception de (vis) **depuis** r
$used_p[r] := true$; `envoyer (`ack`) à` r

1. Appliquer l'algorithme un K_4.
2. Montrer que l'algorithme Awerbuch construit un arbre via un parcours en profondeur en $4n-2$ unités de temps et utilise $4m$ messages.

Exercice 3 – Construction d'un arbre en largeur

On veut un algorithme distribué pour construire un arbre en largeur enraciné en P_i. On suppose le réseau asynchrone.

On utilise le principe de l'inondation :

L'algorithme en P_i est le suivant :

Procédure construction
`envoyer <<` i `>> à tous les voisins`
$nbrep$ `:= nombre de voisins`

A la réception de <acq>
$nbrep$ `:=` $nbrep$ `-1`
`si` $nbrep = 0$ `alors fin`

L'algorithme en P_j, $i \neq j$ est le suivant :

A la réception de <i> de d

```
si participant alors envoyer <acq> sur d
  sinon participant := vrai
   nbrep := nombre de voisins -1
   pere := d
   si nbrep ≠ 0 alors envoyer <i> à tous les voisins sauf d
      sinon envoyer <acq> sur d
```

A la réception de <acq> *nbrep* := *nbrep* -1

```
si nbrep = 0 alors envoyer <acq> sur père
```

1. Pourquoi l'arbre ainsi construit n'est pas forcément un arbre en largeur ?
2. Modifier l'algorithme pour construire un arbre réellement en largeur enraciné en P_i (conseil : construire l'arbre niveau par niveau).
3. Comment baisser le nombre de messages ?

Exercice 4 – Construction d'un anneau virtuel

Un anneau virtuel peut être défini comme une relation d'ordre total sur l'ensemble X des sites, il peut donc être vu globalement comme une site circulaire dans laquelle chaque élément de X apparaît une fois et une seule, cette structure peut être distribuée entre les sites du réseau : il suffit que chacun d'eux soi doté d'une variable $succ_i$ est son suivant dans la liste (ou dans la relation d'ordre). L'exploitation de cet anneau virtuel se fait à l'aide d'un jeton qui parcourt cette liste.

Rôle de l'initiateur $pere_\alpha := \alpha$;
Si $voisins_\alpha = \varnothing$ alors
    terminé, l'anneau est réduit à un seul site
Sinon soit $k \in voisins_\alpha$
    $pv := k$;
    envoyer $explorer(\{i\}, i)$ à P_k;

A la réception de $explorer(z, d)$ **depuis** P_j
$succ_i := d$; $pere_i := j$
Si $voisins_i \subseteq z$ alors
    $R_i(j) := j$
    envoyer $retour(z \cup \{i\}, i)$ à P_j
Sinon
soit $k \in voisins_i - z$

$R_i(k) := j$
envoyer $explorer(z \cup \{i\}, i)$ à P_k

A la réception de $retour(z, d)$ **depuis** P_j
Si $voisins_i \subseteq z$ alors
    Si $pere_i = i$ alors
        P_i est l'initiateur P_α, l'anneau est bouclé
        $succ_i := d$; $R_i(pv) := j$
    Sinon $R_i(pere_i) := j$
    envoyer $retour(z, d)$ à P_{pere}
Sinon
soit $k \in voisins_i - z$
$R_i(k) := j$
envoyer $explorer(z, d)$ à P_k

1. La création de la variable $succ_i$ nécessite une information sur le suivant dans l'anneau. Que devons-nous savoir ?
2. Nous supposerons par la suite que chaque site P_i possède une table de routage R_i telle que :
   — $R_i(j)$ contient l'identité de son voisin auquel P_i doit transmettre le jeton lorsqu'il le reçoit de P_j.
   Nous devons donc définir les variables $succ_i$ et R_i $\forall i$.
   (a) Comment affecter les variables $succ_i$?
   (b) Comment définir les tables de routage ?
   (c) Comment l'initiateur crée sa table de routage ?
3. Quel est la complexité ?
4. Donner l'algorithme qui permet l'exploitation du jeton sur l'anneau virtuel

Exercice 5 – Construction d'un arbre « en profondeur d'abord »

Soit $G = (V, E)$ un graphe connexe quelconque représentant le réseau physique. On veut créer dans G un arbre de type *en profondeur d'abord* à partir d'un sommet *racine*. Chaque sommet maintiendra la connaissance de son *père* et de ses *fils* grâce à un marquage des liens correspondants (la procédure *mark* permet de changer ce marquage). Au début, tous les sommets sont dans l'état *idle* et tous les liens sont marqués *nonvisite*. Voici une partie de l'algorithme pour le sommet i :

```
Procédure chercher;
    Si (∃k: mark_i(k) = nonvisite) alors
        Envoyer(< TOKEN >) sur le lien k;
        mark_i(k) := fils;
    Sinon Si (i = initiateur) alors stop;
        Sinon Envoyer(< TOKEN >) sur le lien k tel que mark_i(k) = pere ;

Initialisation (* à exécuter seulement par l'initiateur *) :
    Si (etat_i = idle) alors
        etat_i := decouvert;
        chercher;
        Pour tout (k: mark_i(k) = visite ou mark_i(k) = nonvisite)
            Envoyer (< VISITED >) sur le lien k;

Lors de la réception de < VISITED > sur le lien j
    Si (mark_i(j) = nonvisite) alors mark_i(j) := visite;

Lors de la réception de < TOKEN > sur le lien j
...
```

1. Ecrire la procédure de réception de $< TOKEN >$.
2. Donner un exemple d'exécution.
3. On donne maintenant une autre procédure de réception de $< VISITED >$:

```
Lors de la réception de < VISITED > sur le lien j
    Si (mark_i(j) = nonvisite) alors mark_i(j) := visite;
    Si (mark_i(j) = fils) alors
        mark_i(j) := visite;
        chercher;
```

   A quoi sert le rajout par rapport à la précédente version ?

4. Evaluer la complexité en nombre de messages.

Exercice 6 – L'algorithme de Dijkstra en séquentiel

Cet exercice a pour but de vous remettre en mémoire l'algorithme de Dijkstra. Voici l'algorithme de Dijkstra :

Algorithm 1 L'algorithme de Dijkstra

$d(s) := 0$; $pred(s) := 0$
$d(i) := +\infty$; pour tout $i \neq s$
$S = \emptyset$
while $S \neq X$ **do**
  choisir $i \in \overline{S}$ avec $d(i)$ minimum
  $S = S \cup \{i\}$
  for chaque arc (i, j) **do**
    if (i, j) est un raccourci **then**
      l'emprunter
    end if
  end for
end while

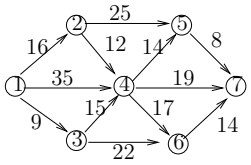

FIGURE 1 – *Graphe*

1. Appliquer cet algorithme sur le graphe donné par la figure 1 à partir du sommet 1.
2. Démontrer que l'algorithme de Dijkstra résout le problème des plus courts chemins en temps $\mathcal{O}(n^2)$.

Exercice 7 – Maintien des plus courts chemins dans un graphe connexe quelconque.

Dans cet exercice nous proposons d'écrire un algorithme réparti qui va construire entre chaque paire de sommets de $G = (V, E)$ un chemin de plus courte distance. Les communications se font entre voisins et sont asynchrones. Chaque sommet connaît la taille du graphe (n sommets) et les liens de communication sont *FIFO*. La construction des plus courts chemins est faite par "apprentissage" du réseau. Chaque sommet u maintient :

— Un tableau D_u dont la case $D_u[v]$ est la distance estimée par u de $d_G(u, v)$.
— Un tableau $Sortie_u$ dont l'entrée $Sortie_u[v]$ indique le voisin w de u qui est sur un plus court chemin de u à v.

— Un tableau $estimat_u$ dont l'entrée $estimat_u[w, v]$, avec $w \in voisins_u$, est l'estimation par u de la distance $d_G(w, v)$.

Voici une partie de l'algorithme pour un sommet u quelconque :

```
Procédure recalcul_Dist(v)
...

Procédure Initialisation
    Pour tout (w ∈ voisins_u, v ∈ V) faire estimat_u[w, v] := n;
    Pour tout (v ∈ V) faire
        D_u[v] := n;
        Sortie_u[v] := NIL;
    D_u[u] := 0;
    Sortie_u[u] := local;
    Pour tout (w ∈ voisins_u) faire Envoyer(< MADIST, u, 0 >) à w;

Lors de la réception de < MADIST, v, d > de w (w ∈ voisins_u)
    estimat_u[w, v] := d;
    recalcul_Dist(v);
```

1. Ecrire la procédure *recalcul_ Dist*.
2. Pourquoi, dans un graphe dont la topologie ne change pas, l'algorithme s'arrête au bout d'un temps fini ? Quelles sont alors les valeurs des divers paramètres ? Proposez une fonction de routage basée sur les résultats de l'algorithme précédent.
3. Que peut on faire en cas de pannes de liens sachant qu'un sommet peut être prévenu de la panne d'un de ses liens adjacent par une interruption ?

Dans les exercices suivants nous allons représenter un système par un graphe $G = (V, E)$. L'ensemble V des *sommets* représente les sites. On a une arête $[u, v] \in E$ entre le sommet u et le sommet v s'il existe un lien de communication entre u et v dans le réseau sous-jacent entre les sites représentés par u et v. Nous ne considérons ici que des *graphes connexes*.

Exercice 8 – Election dans un graphe quelconque

On considère un réseau quelconque (connexe) dans lequel les lignes sont bidirectionnelles et fiables ; chaque processus ne connait pas ses voisins et il connait par contre leurs identités. Au cours du calcul un processus ne doit pas apprendre d'informations globales (telle que la structure du réseau par exemple) q'il pourrait ensuite utiliser ; (une telle contrainte favorise

le rétablissement après panne). Il s'agit de concevoir un algorithme distribué, qui peut être démarré par un ou plusieurs processus, tel qu'à la fin de l'algorithme un processus unique ait été élu par les autres et que chacun des autres en connaisse l'identité et connaisse également lequel de ses voisins permet d'atteindre le processus élu.

1. Quel maximum sous-graphe (au sens de l'inclusion) pouvons nous utiliser pour reformuler le problème ?
2. Un des problèmes délicats à résoudre est le fait que l'algorithme peut être démarré par un nombre quelconque de processus. Proposer une idée pour bloquer l'exploration venant d'un processus ayant une identité i plus faible que l'exploration d'un processus j avec $i < j$.

   Chaque processus P_i est doté du contexte suivant :
   — $voisins_i$, initialisée à $\{identits\ desvoisins\ de\ P_i\}$;
   — $etat_i$: (initial, candidat,battu ,élu), initialisé à initial. Cette variable donne l'état courant de P_i ; à la fin de l'algorithme une seule de ces variables a la valeur elu, les autres ayant la valeur *battu*
   — $pgvu_i$, entier initialisé à i. Il s'agit de la plus grande identité qu'ait jamais vu passer P_i ; à la fin de l'algorithme elle contiendra la plus grande des identités.
   — $pere_i$, entier initialisé à *nil*
   — $fils_i$, ensemble d'entiers initialisé à $\varnothing$. Ces deux variables servent à placer P_i
3. Donner la procédure lancer une élection. Pour un site i donné, à quel voisin i va envoyer la procédure lancer une élection.
4. Donner la procédure réception d'explorer.
5. Donner la procédure réception de rebrousser.
6. Donner la procédure réception conclure.
7. Quel est la complexité de votre algorithme en nombre de messages ?

Exercice 9 – Rappel en théorie des graphes

Une structure de données simple pour représenter un graphe est la matrice d'adjacence M. Pour obtenir M, on numérote les sommets du graphe de façon quelconque. $X = \{x_1, \ldots, x_n\}$. M est une matrice carrée $n \times n$ dont les coefficients sont 0 et 1 telle que $M_{i,j} = 1$ si $(x_i, x_j) \in E$ et $M_{i,j} = 0$ sinon. Démontrer le la proposition suivante : **Proposition : Soit M^p la puissance $p-ime$ de la matrice M, le coefficient $M^p_{i,j}$ est égal au nombre de chemins de longueur p de G dont l'origine est le sommet x_i et dont l'extrémité est le sommet x_j.**

Dans les exercices suivants, chaque site i, initialement, possède une valeur α_i et on veut au final que tous les sites aient la somme de toutes les valeurs ce trouvant sur chaque site initialement c'est à dire pour chaque site i on veut obtenir $\sum \alpha_i$.

Exercice 10 – Routage dans l'hypercube

On appelle hypercube de dimension n et on note $H(n)$, le graphe dont les sommets sont les mots de longueur n sur un alphabet à deux lettres 0 et 1 et dont deux sommets sont adjacents si et seulement si ils différent en une seule coordonnée. Un sommet noté $x_1x_2...x_i...x_n$ est donc relié aux sommets $x_1x_2...\bar{x_i}...x_n$ avec $i = 1, 2, ..., n$.

1. Représenter $H(3)$ et $H(4)$.
2. Montrer que $H(n)$ possède 2^n sommets.
3. Montrer que $H(n)$ est n-régulier.
4. Quel est le nombre d'arêtes ?
5. Quel est le diamètre de $H(n)$ où le diamètre est le maximum pris sur toutes les paires de sommets x et y de G, de la distance entre x et y.
6. Donner la matrice d'adjacence de l'hypercube $M_{H(3)}, M_{H(4)}$ et calculer $M^k_{H(3)}, M^k_{H(4)}$ avec $k \geq 2$.
7. Proposer un algorithme qui permettent de résoudre le problème initial et le faire tourner sur un exemple.
8. Donner la complexité en nombre de messages et en temps pour le modèle synchrone.

Exercice 11 – Routage dans le graphe de De Bruijn

Le graphe de Bruijn $B(d, D)$ est le graphe orienté dont les sommets sont les mots de longueur D sur un alphabet de taille d, $d \geq 2$ et dont un sommet noté $x_1x_2...x_D$ est relié aux sommets $x_2...x_D\lambda$, λ étant une lettre quelconque de l'alphabet. On passe donc d'un sommet à un autre par un décalage à gauche avec adjonction d'une lettre.

1. Représenter le graphe de de Bruijn $B(2, 3)$.
2. Donner le nombre de sommets pour $B(d, D)$ et le nombre d'arcs.
3. Donner le diamètre de $B(d, D)$.
4. Donner la matrice d'adjacence M du graphe $B(2, 3)$. Calculer $M^k, k \geq 2$. Conclure.
5. Proposer un algorithme qui permettent de résoudre le problème initial et le faire tourner sur un exemple.

Exercice 12 – Topologie quelconque Proposer un algorithme dans le cadre où la topologie du réseau est inconnue.

Exercice 13 – Liste des sommets à une distance 2 On définit le poids d'un sommet d'un réseau comme le nombre de sommets à distance inférieur ou égale à 2 de lui.

1. Donner un algorithme distribué qui permette de calculer le poids d'un sommet x.
2. Donner un algorithme qui indique à i la liste des voisins de ses voisins.

Université de Montpellier 2020/2021
Master Durée: 4.5h

Algorithme distribué
TD – Séance n° 4

Exercice 1 – Terminaison avec maintien d'un arbre On se place dans le contexte suivant : initialement tous les processus sont passifs à l'exception de l'un d'entre eux appelé P_0. Un processus ne peut devenir actif que sur réception de messages, il peut alors envoyer à d'autres processus et les rendre donc éventuellement actifs ; un processus ne peut émettre qu'un nombre fini de messages. De plus, pour simplifier nous supposons que P_0 ne peut qu'émettre des messages et ceci au début du calcul. Ce genre de communication et de calcul est appelé *calcul diffusant*. Il permet de modéliser un grand nombre de problèmes concrets dans lesquels P_0 joue le rôle de l'environnement qui déclenche l'activité d'un calcul dans le système.

Les outils sur lesquels s'articule la solution présentée sont ,une topologie de contrôle particulière (l'arbre) et que le fait que les délais de communication sont bornés par Δ.

Lorsqu'un processus est rendu actif par la réception d'un message il se place, s'il n'a pas de père, dans un arbre de contrôle avec l'émetteur du message comme père ; il informe l'émetteur de cette relation parentale qui l'inclut dans ses fils. Lorsqu'un processus est passif et n'a pas de fils il sort de l'arbre en informant son père de la destruction de la relation parentale. Ainsi les processus actifs ou pères de processus actifs forment un arbre de racine P_0 (un processus n'existe qu'une fois au plus dans l'arbre). L'arbre de contrôle peut évoluer et se reconfigurer au cours du calcul en fonction de l'état actif ou passif dans lequel se trouve un processus à un instant donné et des messages échangés.

1. Le nombre de fils varie entre quelle valeurs ?
2. Comment le site P_0 détecte la terminaison ?
3. Comment le site P_i sait-il que le site P_j est son fils ou pas ?
4. Donner les variables nécessaire à l'algorithme.
5. Donner l'algorithme.
6. Donner un algorithme voisin qui n'utilise pas le Δ.

Exercice 2 – Algorithme inexact pour la détection de la terminaison sur un anneau logique unidirectionnel

Soit l'algorithme suivant :

- $\forall i, etat_i \in \{passif, actif\}$
- Algorithme :
  - Faire circuler un jeton (message de contrôle) selon une structure de l'anneau, envoyé initialement par P_0.
  - Lorsqu'un site est passif et possède le jeton, il l'envoie au site suivant.
  - Lorsque le jeton revient à P_0 la terminaison est détectée.

1. Donner l'algorithme de manière formel.
2. Donner un exemple pour lequel l'algorithme décrète la terminaison à tort.

Exercice 3 – Algorihtme de Rana sur un anneau logique unidirectionnel

Soit l'algorithme suivant :

A chaque fois qu'un processus reçoit soit un message applicatif soit un message de contrôle, il met son horloge logique locale à jour. Les messages de contrôle circulent sur l'anneau :

- Message decontrôle : $< H, compteur >$
- Chaque site envoie le message de contrôle à son successeur et le reçoit de son prédécesseur ;

Lorsqu'un processus devient passif, il enregistre la valeur de son horloge locale H_{pas} et envoie le message de contrôle $< H_{pas}, 1 >$ à son successeur ;. Lors de la réception d'un message de contrôle :

- si le site est actif, il ignore le message ;
- sinon
  - si compteur $\neq N$
    - Si la valeur de son passage à passif : $H_{pas} > H_{msg}$ du message de contrôle reçu, le message est ignoré.
    - Sinon, le message est envoyé à son successeur avec le compteur incrémenté $> H_{pas}, compteur + 1 >$
  - Sinon
    - Terminaison détectée
    - Le site envoie à son successeur un message de terminaison. Le message fera le tour de l'anneau.

1. Donner l'algorithme.
2. Jusitifier la terminaison de l'algorithme.
3. Illustrer l'exécution sur un exemple de votre choix.

Exercice 4 – Algorihtme de Mattern pour un système centralisé

In the algorithm each message and each process are assigned to credit value, which is always between 0 and 1 (inclusive), and the algorithm maintains the following assertions as invariants :

— S_1 : The sum of all circuits (in messages and processes) equals to 1.
— S_2 : A basic message has a positive credit.
— S_3 : An active process has a positive credit.

Process holding a positive credit when this is not prescribed according to these rules (i.e. passive processes) send their credit to the initiator. The initiator acts as a bank, collecting all credits sent to it in a variable ret (for *returned*).

When the initiator owns all credits, the requirement for active processes and basic messages to have a positive credit implies that there are no such processees and no such messages ; hence *term* holds.

— *Rule* 1 : When $ret = 1$, the initiator calls *Announce*
In order to fulfill the liveness requirement, it must be ensured that if terminaison occurs, all credits are eventually transferred to the initiator. If te basic computation has terminated, there are no basic messages any more, and we only need to concern ourselves with the credits held by processes.
— *Rule* 2 : When a process becomes passive, it sends its credit to the initiator. In the initial configuration only the initiator is active and has a positive credit, namely 1, and $ret = 0$, which implies that S_1 through S_3 are satisfied The invariant must be maintained during the computation ; this is taken care of by the following rules. First, each basic message must be given a positive credit when it is sent ; fortunately, its sender is active, and hance has a positive credit.
— *Rule* 3 : When the active process p sends a message, its credit is divided among p and the message.
A process must be given a positive credit when it is activated ; fortunately, the message it receives at that point contains a positive credit.
— *Rule* 4 : When a process is activated it is given the credit of the message activating it.
The only situation not covered by these rules is the receipt of a basic message by an already active process. The process already has a positive credit, hence does not need the credit of the message in order to satisfy S_3 ; however, the credit may not be destroyed as this would lead to a violation of S_1. The receiving process may handle the credit in two different ways, both giving rise to a correct algorithm.
— *Rule* 5*a* : When an active process receives a basic message, the credit of that message is sent to the initiator.

— *Rule* 5*b* : When an active process receives a basic message, the credit of that message is added to the credit of the process.

1. Donner l'algorithme.
2. Appliquer l'algorithme sur les exemples données par la Figure 1.

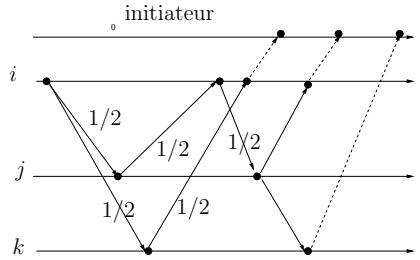

FIGURE 1 – *A compléter*

3. Montrer que l'algorithme détecte la terminaison.
4. Un problème peut apparaître lorsqu'il est plus possible de divisé (en effet, le credit est encodé sur un nombre limité de bits). Quelle solution proposez-vous ? Donner un exemple de fonctionnement.

Université de Montpellier 2019/2020
Master Durée: 4.5h

Algorithme distribué
TD – Séance n° 4

Exercice 1 – Perte d'un jeton sur un anneau unidirectionnel Supposons n sites placés sur l'anneau dans l'ordre $S_0, S_1, \cdots, S_{n-1}, S_0$. Si le jeton se perd par exemple entre S_i et S_{i+1} deux choses sont à faire :

— détecter la perte du jeton
— en générer un nouvel exemplaire.

Si le jeton se perd et que plusieurs sites le détectent il est essentiel qu'un et un seul d'entre eux le régénère ; se pose donc le problème d'exclusion.

1. Expliquer pourquoi le fait d'avoir un mécanisme d'acquitement du jeton entre deux sites ne fonctionnent pas. L'idée est valué le jeton et à utiliser cette valeur pour laisser sur chaque site une trace qui donnera les informations nécessaires pour détecter la perte éventuelle du jeton et le régénérer si tel est le cas avec la bonne valeur.
2. Appelons s_i la variable d'état locale à S_i dans laquelle celui-ci place la valeur vj du jeton lorsqu'il le possède et cela avant de la transmettre à S_{i+1}. Comment détecter la perte d'un jeton ? Quel est le critère qu'un site devra vérifier lors de la perte d'un jeton ?
3. Que se passe-t-'il pour le site S_0 ? Proposez une solution.
4. On peut de manière très générale supposer que les sites peuvent tomber en panne et qu'alors un protocole sous-jacent reconfigure l'anneau. Proposer une idée pour une réconfiguration simple.
5. Proposer maintenant l'algorithme.

Exercice 2 – Perte d'un jeton sur un anneau unidirectionnel (suite)

Supposons n sites placés sur l'anneau dans l'ordre $S_0, S_1, \cdots, S_{n-1}, S_0$. Si le jeton se perd par exemple entre S_i et S_{i+1} deux choses sont à faire :

— détecter la perte du jeton
— en générer un nouvel exemplaire.

Si le jeton se perd et que plusieurs sites le détectent il est essentiel qu'un et un seul d'entre eux le régénère ; se pose donc le problème d'exclusion. Dans cet exercice nous proposons une alternative aux horloges de gardes : nous proposons d'utiliser deux jetons *ping* et *pong* ayant pour valuation respectivement *nbi* et *nbo*.

1. Pour détecter la perte d'un jeton nous allons utiliser un invariant. Proposez un invariant. Comment un jeton arrivant sur un site S_i peut détecter la perte de l'autre.
2. Donner l'algorithme de la reception des jetons *ping* et *pong* sur un site S_i. Donner également l'algorithme de reception des jetons sur un site S_i.
3. Quel est le défaut de l'algorithme. Modifier-le

Exercice 3 – Extension de l'algorithme de Naimi-Tréhel

L'algorithme de Naimi-Trehel à pour but de gérer de façon distribuée, l'accès à une ressource critique, d'un ensemble de noeud (ou site). L'algorithme, à la manière du Token Ring, va utiliser un jeton pour symboliser l'autorisation d'accès à une ressource critique. Cet algorithme gère les noeuds du réseau en les organisant sous forme de deux structures. Tout d'abord un arbre va gérer la transmission des demandes à travers le réseau. Ensuite une file d'attente va stocker l'ordre de ces demandes et ainsi permettre la transmission du jeton d'un noeud a l'autre. Chaque noeud doit donc connaître le noeud situé juste au dessus de lui dans l'arborescence (appelé last), et le noeud qui lui succède dans la file d'attente (appelé next). Les noeuds du réseau qui veulent utiliser la ressource critique, vont utiliser un message type appelé $TOKENREQUEST$ afin de rentrer dans la file d'attente. Ainsi, lorsqu'un noeud de l'arbre a besoin de la ressource, il envoi le message $TOKENREQUEST$ à son last. A partir de la, le noeud qui reçoit la demande n'a que 2 options : - Si il n'est pas racine de l'arbre, il transmet la demande a son last puis modifie cette valeur de last pour pointer vers le demandeur. - Si il est racine de l'arbre, il est donc le destinataire du message. Il modifie la valeur de son next pour pouvoir lui transmettre le jeton dès sa sortie de la section critique. Il en fait de même avec son last pour pointer vers le demandeur.

Un invariant de cet algorithme est que le noeud racine de l'arbre sera également le dernier noeud de la file d'attente.

Cependant, une panne d'un noeud dans le réseau peut provoquer la perte irrémédiable d'information. En particulier, la perte du jeton peut survenir, si le noeud en panne est en possession du jeton, ou si il est le successeur dans la file d'attente du possesseur du jeton. Cette erreur pourrait compromettre l'intégrité du système car aucun des noeuds du réseau ne pourrait accéder à la ressource critique.

Les trois types de pannes que l'algorithme de Naïmi-Tréhel doit surporter :

1. Le sommet qui tombe en panne appartient à l'arbre.
2. Le sommet qui tombe en panne appartient à la file.
3. Le sommet qui tombe en panne possède le jeton.

1. Quel est le mécanisme peut-on introduire afin de garantir un minimum de tolérance aux pannes ?

2. Nous proposons plusieurs mécanismes selon le type de panne. Soit S_i le site qui détecte la panne :
   (a) Le mécanisme $M1$: il apparaît quand le sommet S_i a reçu le massage $< COMMIT >$ et le nombre de sites en panne est inférieur à k.
   (b) Le mécanisme $M2$: il apparaît quand le sommet S_i a reçu le message $> COMMIT >$ et le nombre de sites en panne est supérieur ou égal à k.
   (c) Le mécanisme $M3a$: il apparaît quand S_i n'a pas reçu de message $> COMMIT >$ et est le seul qui a détecté une panne.
   (d) Le mécanisme $M3b$: il apparaît quand S_i n'a pas reçu de $> COMMIT >$ et plusieurs sites ont détecté une panne.
3. Comment le site S_i peut détecter une panne pour chaque type de mécanisme ?
4. Donner les algorithmes associés à chaque mécanisme et préciser leur complexité.
5. Comment réconstruire l'arbre logique pour le mécanisme $M3$?
6. Comment géré le fait que la position augmente tout le temps ?

Université de Montpellier II — 2017/2018
Master — Durée: 3h

Algorithme distribué
TD – Séance n° 6

Exercice 1 – Généraux byzantins

On considère quatre processus répartis chacun sur un site dont au plus un est byzantin, et le protocole d'accord suivant : lors de la première phase, chaque processus diffuse sa valeur initiale à tous les autres, attend d'avoir obtenu eux réponses (le réseau est asynchrone et les messages signés) et choisit la valeur majoritaire parmi les trois valeurs (la sienne et les deux reçues, en cas d'égalité il choisit la plus petite. Les phases 2 et 3 sont identiques en partant de la valeur choisie à la phase précédente. La valeur de l'accord est la valeur choisie à la fin de la troisième phase.

1. Ce protocole est-il une solution correcte au problème de l'accord ?

Exercice 2 – Généraux byzantins

1. Appliquer l'algorithme avec $m = 2$ et $n = 7$ et le commandant admet un un comportement normal.
2. Exhiber un exemple pour $m = 2$ et $n = 6$ pour lequel nous n'arrivons pas à un consensus.
3. Six membres d'une famille interview un candidat pour un travail de cuisinier. Comment la famille peut rester sans consensus si une personne de la famille disparaît ?
4. Considérons un système synchrone de 2^n processus, l'architecture est un n-hypercube. Quel est le nombre maximum de fautes byzantins que le système peut supporter quand nous souhaitons obtenir un consensus ?
5. Prouver que, dans un réseau connecté, il est impossible de d'obtenir un consensus avec n généraux et m traitres, et la connectivité est $\leq 2M + 1$.

Exercice 3 – Calcul d'un secret

Soit s un entier secret, n le nombre de sites et t une borne supérieure du nombre de félons. On se propose de construire n entiers, appelés les ombres du secret s, qui seront distribués entre les participants de telle sorte que :

— si au moins t participants se communiquent leurs ombres, alors ils peuvent reconstruire le secret,

— tout groupe de moins de t participants ne peut reconstruire s.

1. Que proposez-vous comme fonction mathématique pour résoudre le problème ?
2. Montrer l'existence et l'unicité d'une fonction mathématique.
3. Ecrire les système d'équations linéaires sous la forme avec $A.F = Y$ où est une matrice de Vandermonde.
4. Donner la valeur du déterminant associé à la matrice A. Conclusion sur la caractère inversible de la matrice.
5. Application à $n = 12, t = 3, s_1 = 4, s_5 = 9, s_6 = 12$. Quel est le secret ?