Chapitre 1

Retrait de sommets d'un DST (suite des travaux)

1.1 Présentation de différents cas de figure

1.1.1 Retraits avec fusion

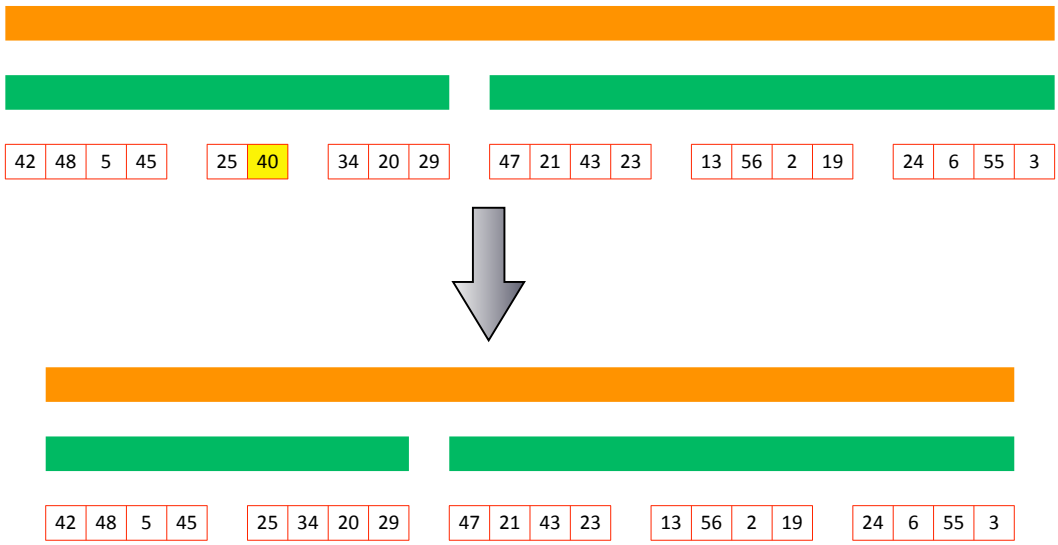

FIGURE 1.1 – Exemple de retrait : le sommet 40 quitte un DST [2,4]

Ici, le départ de 40 laisse un nœud orphelin, le 25, que le groupe à sa droite peut accueillir.

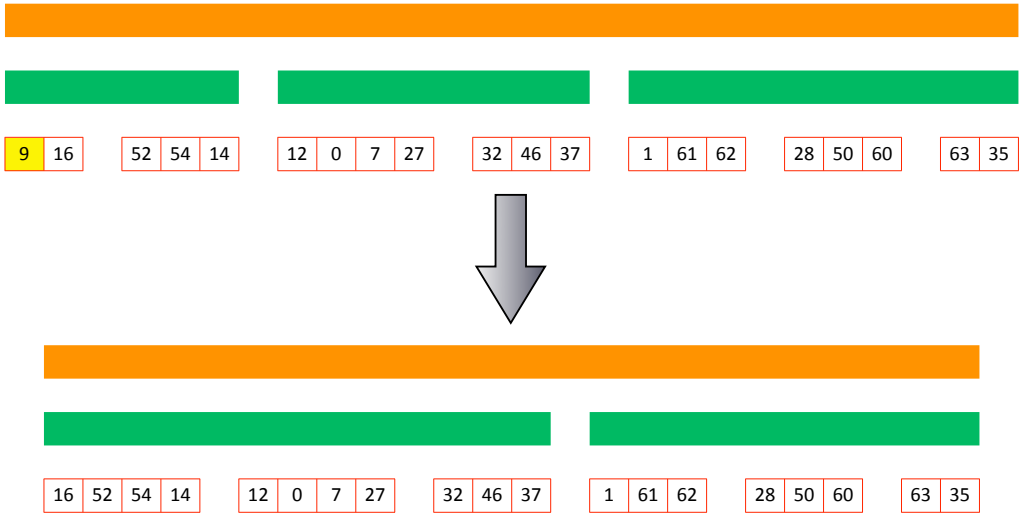

FIGURE 1.2 – Exemple de retrait : le sommet 9 quitte un DST [2,4]

Cette fois, la fusion de 16 avec le groupe [52 . . . 14] provoque la fusion de l'étage supérieur (vert) qui a de la place.

1.2 Retraits avec transfert

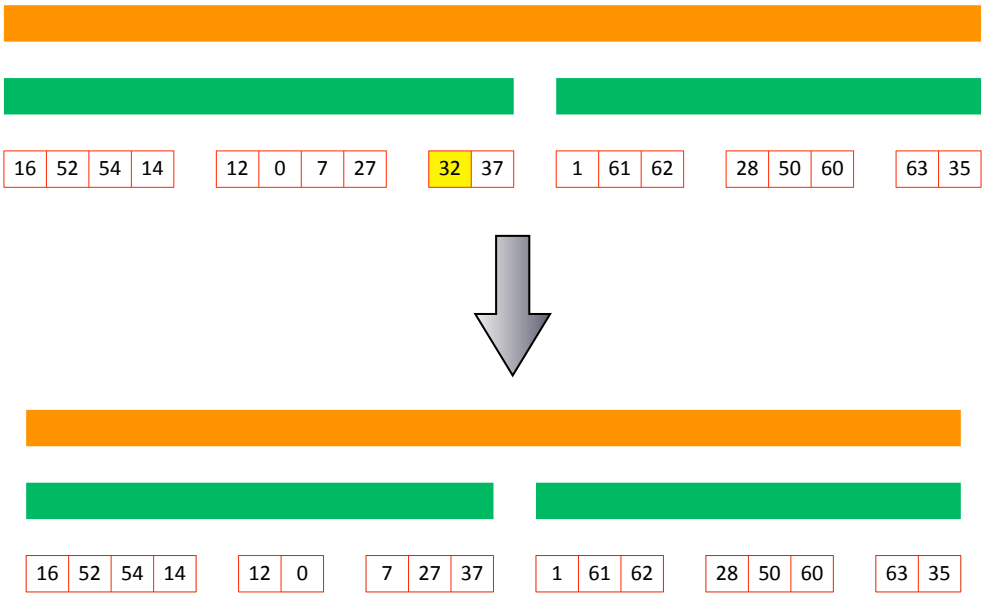

FIGURE 1.3 – Exemple de retrait : le sommet 32 quitte un DST [2,4]

Les groupes [16 . . . 14] et [12 . . . 27] n'ont pas de place pour accueillir 37. Il va donc y avoir un transfert de [7, 27] vers 37. À noter qu'un transfert ne provoque pas de propagation aux étages supérieurs puisqu'il ne réalise qu'une nouvelle répartition des membres d'un étage donné, sans modifier leur nombre, contrairement à la fusion.

1.3 Détail des opérations de retrait

1.3.1 La fonction LEAVE()

Lorsqu'un nœud quitte le DST, il doit exécuter la fonction LEAVE()

Algorithme 1 : Le départ

```
1: procedure LEAVE( )
                                          ▷ travaille sur des copies des tables de routage
2:     cpy_brothers ← me.brothers
3:     cpy_preds ← me.preds
```

Cette fonction modifiant les tables de routage et de prédécesseurs, on travaille sur des copies de ces tables.

Algorithme 1 : LEAVE (suite)

```
4:     for stage ← 0 to (height(me.brothers) − 1) do
                                          ▷ INFORME MES PRÉDÉCESSEURS DE MON DÉPART
5:         for pred ← 0 to (size(cpy_preds[stage]) − 1) do
6:             if (cpy_preds[stage][pred].id ≠ me.id) then
7:                 if (stage = 0) then
8:                     SEND_MSG_ASYNC(cpy_preds[stage][pred].id, del_bro(stage, me.id))
9:                 else
10:                    new_rep_id ← un de mes frères de l'étage 0 choisi aléatoirement
11:                    SEND_MSG_ASYNC(cpy_preds[stage][pred].id,
                           replace_bro(stage, new_rep_id))
12:                end if
13:            end if
14:        end for                                          ▷ prédécesseur suivant
```

À chaque étage, le nœud courant doit prévenir chacun de ses prédécesseurs de son départ :

- à l'étage 0, le prédécesseur doit simplement ôter ce nœud de sa table de routage. (fonction DEL_BRO())
- aux autres étages, le prédécesseur doit utiliser un autre représentant que le nœud courant. (fonction REPLACE_BRO()) Ce sera un de ses frères de niveau 0 choisi aléatoirement.

Algorithme 1 : LEAVE (suite)

```
                                                    ▷ INFORME MES FRÈRES DE MON DÉPART
15:     for brother ← 0 to (size(cpy_brothers[stage]) − 1) do
16:         if (cpy_brothers[stage][brother].id ≠ me.id) then
17:             SEND_MSG_ASYNC(cpy_brothers[stage][brother].id,
                    del_pred(stage, me.id))
18:         end if
19:     end for                                                          ▷ frère suivant
20: end for                                                              ▷ étage suivant
```

Ensuite, chacun des frères doit également être prévenu : ils ne peuvent plus avoir le nœud courant comme prédécesseur. (fonction DEL_PRED())

Algorithme 1 : LEAVE (suite)

```
                                    ▷ CHARGE UN DE MES FRÈRES DE TRAITER LES FUSIONS OU TRANSFERTS
21: if (size(me.brothers[0]) ≤ a) then
22:     idx ← index d'un de mes frères de l'étage 0
23:     SEND_MSG_SYNC(me.brothers[0][idx].id, merge_req())
24: end if
25: end procedure
```

Ceci fait, si le groupe courant est devenu trop petit, alors l'un de ses membres est chargé de traiter ce cas en exécutant la fonction MERGE_REQ(). (voir algo 2)

1.3.2 La fonction MERGE_REQ()

Cette fonction parcourt le DST de bas en haut, tant que l'étage courant est trop petit. À chaque étage, elle regarde si on doit réaliser une fusion ou un transfert et transmet les ordres en conséquence.

Algorithme 2 : Traite les fusions ou transferts consécutifs à un départ

```
1: procedure MERGE_REQUEST( )
2:     stage ← 0
3:     size_last_stage ← 0
                                    ▷ parcourt l'ensemble des étages si nécessaire
4:     while (size(me.brothers[stage]) < a) and (stage < height(me.brothers) − 1) do
5:         pos_contact ← merge_or_transfer(me, stage)
```

C'est la fonction MERGE_OR_TRANSFER() (voir algo 3) qui est chargée d'examiner si une fusion est possible. Si c'est le cas, elle retourne la position d'un contact de l'étage supérieur qu'on pourra joindre pour réaliser cette fusion. Sinon, elle retourne -1, auquel cas on réalisera alors un transfert.

Algorithme 2 : MERGE_REQUEST (suite)

```
6:    if (pos_contact > −1) then                          ▷ UNE FUSION EST POSSIBLE
7:        transfer ← 0
8:        pos_me ← index(stage + 1, me.id)
                                          ▷ demande au contact de réaliser une première fusion ...
9:        if (pos_me > pos_contact) then
10:           right ← 11
11:       else
12:           right ← 10
13:       end if
14:       SEND_MSG_SYNC(me.brothers[stage + 1][pos_contact].id,
              merge(stage, pos_me, pos_contact, right, me.brothers[stage]))
                                          ▷ ... puis lui demande de diffuser une tâche de fusion
15:       if (pos_me > pos_contact) then
16:           right ← 1
17:       else
18:           right ← 0
19:       end if
20:       SEND_MSG_SYNC(me.brothers[stage + 1][pos_contact].id,
              broadcast_merge(stage, pos_me, pos_contact, right, me.brothers[stage]))
```

Une fusion est possible. On commence par demander au contact de réaliser une première fusion, puis, sa table de routage étant alors correcte, on lui demande de lancer une diffusion de cette tâche de fusion pour mettre à jour tout son voisinage.

Ligne 14 : la fonction MERGE() chargée de la fusion proprement dite a besoin de savoir dans quel sens elle doit être réalisée. C'est le rôle de la variable *right*. [1] On lui fournit également l'ensemble des membres de l'étage à fusionner. (voir algo 4)

1. voir les détails de la fonction MERGE() pour les différences entre 1, 11 et 0, 10.

Algorithme 2 : MERGE_REQUEST (suite)

21: ▷ après la fusion, l'étage supérieur contient deux représentants du même groupe

22: CLEAN_UPPER_STAGE($stage$, pos_me, $pos\_contact$) ▷ exécution locale

▷ diffusion

23: BROADCAST(me, $stage$+1, `clean_upper_stage`($stage$, pos_me, $pos\_contact$))

Après la fusion, l'étage supérieur contient deux représentants du même groupe et il faut donc le "nettoyer". La fonction CLEAN_UPPER_STAGE() s'en charge. Elle est premièrement exécutée en local pour que sa diffusion se déroule ensuite correctement.

Algorithme 2 : MERGE_REQUEST (suite)

24: **else**

25: ▷ FUSION IMPOSSIBLE - IL FAUT FAIRE UN TRANSFERT

26: $pos\_me\_up \leftarrow$ `index`($stage + 1$, $me.id$)

27: on peut sortir cette variable puisqu'elle est utilisée dans les deux cas du If

29: **if** ($pos\_me\_up = 0$) **then**

30: $pos\_contact \leftarrow 1$

31: $right \leftarrow 0$

32: $cut\_pos \leftarrow b - a - 1$

33: **else**

34: $pos\_contact \leftarrow pos\_me\_up - 1$

35: $right \leftarrow 1$

36: $cut\_pos \leftarrow a$

37: **end if**

38: $contact\_id \leftarrow me.brothers[stage + 1][pos\_contact].id$

39: $answer \leftarrow$ SEND_MSG_SYNC($contact\_id$,
`transfer`($stage$, $right$, cut_pos, $me.id$))

La fusion n'étant cette fois pas possible, on demande au groupe voisin (contacté via $contact\_id$) d'exécuter la fonction TRANSFER(). Celle-ci le coupe (à la position cut_pos) et retourne les membres extraits dans $answer$. [2]

2. Voir le descriptif de cette fonction pour plus de détails

Algorithme 2 : MERGE_REQUEST (suite)

```
40:                                                          ▷ AJOUTE LES NŒUDS REÇUS
41:         current_bro ← me.brothers[stage]                 ▷ sauvegarde l'étage courant
42:                                          ▷ chaque membre de la branche courante ajoute les nœuds reçus
43:         BR_ADD_BRO_ARRAY(stage, answer.rep_array, mod((right + 1), 2)
44:                                          ▷ chaque membre du groupe reçu ajoute les nœuds courants
45:         SEND_MSG_SYNC(answer.rep_array[0].id,
                br_add_bro_array(stage, current_bro, right))
```

Il s'agit maintenant de fusionner le groupe extrait du voisin avec le groupe courant.

Étudier s'il aurait été possible de réutiliser la fonction MERGE() de la partie fusion. voir aussi la réutilisation de NOUVEAU_FRÈRE_REÇU()

La fonction BR_ADD_BRO_ARRAY() diffuse une tâche d'ajout d'un ensemble de membres donné dans le groupe courant.
Il faut donc réaliser deux diffusions : une sur le groupe courant pour qu'il ajoute les membres extraits et une autre sur le groupe extrait pour qu'il ajoute les membres du groupe courant.

Algorithme 2 : MERGE_REQUEST (suite)

```
46:                                          ▷ nettoyage de l'étage supérieur
47:                                          ▷ exécution locale
48:         UPDATE_UPPER_STAGE(stage, pos_contact, answer.stay_id)
49:                                          ▷ diffusion
50:         BROADCAST(me, stage + 1,
                update_upper_stage(stage, pos_contact, answer.stay_id))
51:       end if
52:       stage ← stage + 1                  ▷ étage suivant
53:     end while
```

Tout comme dans la partie fusion, l'étage supérieur contient maintenant deux représentants du même groupe et il faut corriger cela. La fonction UPDATE_UPPER_STAGE() s'en charge.
pourquoi ne pas avoir utilisé CLEAN_UPPER_STAGE() ? expliquer

Elle est premièrement exécutée en local pour que sa diffusion se déroule ensuite correctement.
on voit qu'au final, on réalise une fusion dans les deux cas (fusion et transfert). Harmonisation possible ?

Algorithme 2 : MERGE_REQUEST (suite)

54: ▷ TRAITEMENT DE LA RACINE

55: $i \leftarrow 0$
56: **while** $(me.brothers[0][i].id = me.id)$ **do**
57: $i \leftarrow i + 1$
58: **end while**

59: $size\_last\_stage \leftarrow$ SEND_MSG_SYNC$(me.brothers[0][i].id,$
$\texttt{get\_size}(\texttt{height}(me.brothers) - 1))$
60: **if** $(size\_last\_stage = 1)$ **then** ▷ détruit la racine

61: BROADCAST$(me,\ \texttt{height}(me.brothers) - 1,\ \texttt{del_root}(\texttt{height}(me.brothers)))$
62: **end if**
63: **end procedure**

Une fois l'ensemble des étages parcouru, on examine la racine. Si elle ne comporte qu'un enfant, c'est qu'il faut la supprimer. On diffuse pour cela la fonction DEL_ROOT() sur l'ensemble du DST.

1.3.3 La fonction MERGE_OR_TRANSFER()

Cette fonction indique si un groupe voisin du groupe courant trop petit a de la place pour l'accueillir en totalité. Si c'est le cas, elle retourne la position d'un représentant de ce groupe, qu'on pourra joindre pour réaliser la fusion. Sinon, elle retourne -1.

Algorithme 3 : indique si une fusion est possible.

```
1: procedure MERGE_OR_TRANSFER(stage)

2:     idx_bro ← 0
3:     merge ← 0

4:     while (merge = 0 and idx_bro < size(me.brothers[stage + 1])) do

5:         if (me.brothers[stage + 1][idx_bro].id ≠ me.id) then

6:             size ← SEND_MSG_SYNC(me.brothers[stage + 1].[idx_bro].[id],
                          get_size(stage))
7:             if (size ≤ b − size(me.brothers[stage])) then
8:                 merge ← 1
9:             end if
10:        end if

11:        idx_bro ← idx_bro + 1
12:    end while

13:    if (merge = 1) then
14:        return idx_bro − 1
15:    else
16:        return −1
17:    end if
18: end procedure
```

Il s'agit d'interroger chaque membre de l'étage supérieur à l'étage courant donné *stage*, pour connaître sa taille. Si l'un d'eux a suffisamment de place (ligne 7) pour accueillir l'ensemble du groupe courant (trop petit, donc), on s'arrête en retournant la position de ce membre, sinon on retourne -1. (lignes 13 à 17)

1.3.4 La fonction MERGE()

Cette fonction permet de fusionner un groupe donné (le petit groupe "à fusionner") avec le groupe courant (le groupe "accueillant").
nodes_array est l'ensemble des membres de l'étage concerné *stage* du groupe à fusionner.
nodes_array_size est sa taille.
pos_me et *pos_contact* sont, respectivement, les positions de représentants des groupes à fusionner et accueillant, à l'étage supérieur à celui de la fusion *stage* (Voir l'appel de cette fonction dans l'algo 2, ligne 14)
right est le sens d'arrivée du groupe à fusionner. Il vaut 1 si le groupe à fusionner est à droite du groupe accueillant.

Algorithme 4 : incorpore des nœuds "orphelins" (*source*) dans le groupe courant (*cible*)

1: **procedure** MERGE(*nodes_array*, *nodes_array_size*, *stage*, *pos_me*, *pos_contact*, *right*)

2: **if** (*nodes_array_size* = `size`(*me.brothers*[*stage*])) **then** ▷ déjà fait
3: **return**
4: **end if**

Cette fonction étant diffusée, il est possible qu'un même nœud la reçoive deux fois. Il faut donc s'assurer qu'on ne l'a pas déjà exécutée.
S'il n'y a pas de place pour accueillir le groupe donné, c'est qu'on a déjà exécuté cette fonction.

expliquer ça autrement, ce n'est pas une question de place

Algorithme 4 : MERGE (suite)

5: **if** (`size`(*me.brothers*[*stage*]) < *a*) **then** ▷ change le sens d'arrivée si 'moi' est dans la source
6: *right* ← mod((*right* + 1), 2)
7: **end if**

Cette fonction est diffusée des deux côtés de la fusion. C'est à dire qu'elle doit être exécutée du côte du groupe à fusionner comme du côté du groupe accueillant. Le sens de la fusion doit donc basculer en conséquence.

Algorithme 4 : MERGE (suite)

```
8:                                                  ▷ nombre de nœuds à incorporer
9:    if (nodes_array_size ≥ size(me.brothers[stage])) then
10:       loc_nodes_array_size ← nodes_array_size − size(me.brothers[stage])
11:   else
12:       loc_nodes_array_size ← nodes_array_size
13:   end if

14:   if (loc_right = 0) then                  ▷ prend la partie gauche de la liste des nœuds fournie
15:       if (loc_nodes_array_size > 0) then
16:           for i ← 0 to (loc_nodes_array_size − 1) do
17:               loc_nodes_array[i] ← nodes_array[i]
18:           end for
19:       end if
20:   else                                     ▷ prend la partie droite de la liste des nœuds fournie
21:       if (loc_nodes_array_size > 0) then
22:           for i ← (nodes_array_size−loc_nodes_array_size) to ((nodes_array_size − 1)) do
23:               loc_nodes_array[i − (nodes_array_size − loc_nodes_array_size)] ←
                  nodes_array[i]
24:           end for
25:       end if
26:   end if
```

On détermine ensuite le nombre de membres à inclure, puis on constitue l'ensemble de ces membres (*loc_nodes_array*).

voir s'il arrive que la condition de la ligne 9 se vérifie, c'est à dire s'il arrive qu'on ne prenne pas la totalité des membres fournis

Pour finir, on insère les nouveaux membres en début de liste lorsqu'ils arrivent de la gauche, ou on les ajoute en fin de liste lorsqu'ils arrivent de la droite.

1.3.5 La fonction CLEAN_UPPER_STAGE()

À la suite d'une fusion à un étage donné, l'étage supérieur contient deux représentants d'un même groupe. Cette fonction permet de corriger cela.

Algorithme 4 : MERGE (suite)

```
27:     if (loc_nodes_array_size > 0) then
28:         if (loc_right = 0) then                    ▷ insère les nouveaux frères au début (gauche)
29:             for i ← (loc_nodes_array_size − 1) to (0) do
30:                 INSERT_BRO(stage, loc_nodes_array[i])
31:                 SEND_MSG_ASYNC(loc_nodes_array[i], add_pred(stage, me.id))
32:             end for
33:         else                                       ▷ ajoute les nouveaux frères à la fin (droite)
34:             for i ← 0 to (loc_nodes_array_size − 1) do
35:                 ADD_BROTHER(stage, loc_nodes_array[i])
36:                 SEND_MSG_ASYNC(loc_nodes_array[i], add_pred(stage, me.id))
37:             end for
38:         end if
39:     end if
40: end procedure
```

Algorithme 5 : Corrige l'étage supérieur après une fusion

```
1: procedure CLEAN_UPPER_STAGE(stage, pos_me, pos_contact)
2:     recp_id ← me.brothers[stage + 1][pos_me].id          ▷ nœud à détruire
3:     if (recp_id = me.id) then                  ▷ échange pos_me et pos_contact
4:         buf ← me.brothers[stage + 1][pos_me]
5:         me.brothers[stage + 1][pos_me] ← me.brothers[stage + 1][pos_contact]
6:         me.brothers[stage + 1][pos_contact] ← buf
7:     end if
```

pos_me et *pos_contact* sont les positions des représentants des deux groupes qui ont fusionnés. On choisit arbitrairement de supprimer la position *pos_me*.[3]

S'il se trouve que c'est la position du nœud courant (qu'on ne peut donc pas détruire), alors on échange les nœuds se trouvant aux positions *pos_me* et *pos_contact*. Ce cas de figure peut se produire lors de la diffusion de cette fonction.

On procède ensuite à l'effacement de ce nœud, puis on met à jour les prédécesseurs.

1.3.6 La fonction UPDATE_UPPER_STAGE()

Après qu'un transfert ait eu lieu, l'étage supérieur peut contenir un nœud qui pointe désormais vers la partie qui a quitté le groupe. Il s'agit donc de le remplacer par l'un des nœuds du groupe restant.

3. Lors de l'appel de la fonction depuis MERGE_REQUEST() (voir algo 2, ligne 22), *pos_me* contient un représentant du petit groupe à fusionner et *pos_contact*, celui du groupe accueillant.

Algorithme 5 : CLEAN_UPPER_STAGE (suite)

```
8:      if (recp_id > −1) then
9:          DEL_BRO(stage + 1, recp_id)
10:         SEND_MSG_ASYNC(recp_id, del_pred(stage + 1, me.id))
11:     else
12:                                          ▷ la fonction a déjà été exécutée
13:     end if
14: end procedure
```

donner un exemple et mentionner SHIFT_BRO dans CUT_NODE

Algorithme 6 : Corrige l'étage supérieur après un transfert

```
1: procedure UPDATE_UPPER_STAGE(stage, pos2repl, new_id)
2:     if (new_id > −1) then
3:         if (me.brothers[stage + 1][pos2repl] ≠ me) then
4:             REPLACE_BRO(stage + 1, pos2repl, new_id)
5:         end if
6:     end if
7: end procedure
```

1.3.7 La fonction TRANSFER()

En diffusant la fonction CUT_NODE() (voir algo 8), cette fonction coupe l'étage *st* du groupe courant à la position *cut_pos*[4]. Puis elle retourne la partie droite du nœud courant si *right* vaut 1, gauche sinon.

préciser l'utilité de *stay_id*

Algorithme 7 : Transfert de nœuds du groupe courant vers un groupe appelant

```
1: procedure TRANSFER(st, right, cut_pos)
2:     if (right = 1) then
3:         start ← cut_pos
4:         end ← size(me.brothers[st]) − 1
5:         answer.stay_id ← me.brothers[st][cut_pos − 1].id
6:     else
7:         start ← 0
8:         end ← cut_pos
9:         answer.stay_id ← me.brothers[st][cut_pos + 1].id
10:    end if

11:    for i ← start to (end) do
12:        answer.rep_array[i − start] = me.brothers[st][i]
13:    end for

14:    BROADCAST(me, st, cut_node(st, right, cut_pos))

15:    return answer
16: end procedure
```

4. *cut_pos* fait partie de la partie coupée

1.3.8 La fonction CUT_NODE()

Pour le nœud courant, cette fonction permet de couper la partie *right* de l'étage *stage* à la position *cut_pos*, incluse dans la partie coupée.

Algorithme 8 : Scinde un nœud lors d'un transfert

```
1:  procedure CUT_NODE(stage, right, cut_pos)
2:      pos_me ← index(me.brothers[stage], me.id)

3:      if (right = 0) then
4:          start ← 0
5:          end ← cut_pos
6:          new_node ← me.brothers[stage][cut_pos + 1]
7:      else
8:          start ← cut_pos
9:          end ← size(me.brothers[stage]) − 1
10:         new_node ← me.brothers[stage][0]
11:     end if

12:     if (pos_me ≥ start and pos_me ≤ end) then
13:         SHIFT_BRO(stage + 1, new_node, right)
14:     end if
```

Lignes 3 à 11, on commence par déterminer *start* et *end* qui délimitent la partie de l'étage *stage* qui doit être coupée.

Lignes 12 à 14, si le nœud courant est compris dans la partie coupée, alors à l'étage supérieur, il ne peut plus être utilisé comme représentant du groupe restant.
Dans le DST, un nœud devant être utilisé comme représentant de son groupe, [5] on ne peut pas simplement le supprimer. Il faut donc le décaler d'un cran – du côté de la partie coupée et en écrasant son voisin – et insérer à sa place un représentant du groupe restant : *new_node*. C'est la fonction SHIFT_BRO() qui est chargée de ces opérations.

5. Ici, son groupe sera le groupe résultant de la fusion de la partie coupée avec le groupe voisin (trop petit), au retour de la fonction TRANSFER().

Algorithme 8 : CUT_NODE (suite)

```
15:     if (right = 0) then
16:         if (pos_me ≤ cut_pos) then
17:             start ← cut_pos + 1
18:             end ← size(me.brothers[stage] − 1)
19:         else
20:             start ← 0
21:             end ← cut_pos
22:         end if
23:     else
24:         if (pos_me < cut_pos) then
25:             start ← cut_pos
26:             end ← size(me.brothers[stage] − 1)
27:         else
28:             start ← 0
29:             end ← cut_pos − 1
30:         end if
31:     end if

32:     DEL_MEMBER(stage, start, end)
33: end procedure
```

Ensuite, on calcule à nouveau les bornes *start* et *end* pour réaliser la coupure proprement dite (avec la fonction DEL_MEMBER()). Si le nœud courant fait partie de la partie coupée, alors on coupe l'autre partie.

du coup, ça contredit l'utilité de la partie d'avant. il faut reprendre des exemples pour étudier ce cas de plus près

1.3.9 La fonction SHIFT_BRO()

Comme indiqué dans la fonction CUT_NODE() (voir algo 8), SHIFT_BRO() sert à corriger l'étage $s+1$ lors d'une coupure à l'étage s.
À partir de la position du nœud courant, elle décale l'étage *stage* d'un cran du côté indiqué par *right*, puis insère *new_node* à la place. S'il y a un nœud en trop, il est détruit.

Algorithme 9 : Décale les membres du groupe pour en accueillir un nouveau à la position de '*moi*'. Le membre en trop est détruit.

```
1: procedure SHIFT_BRO(stage, new_node, right)
                                ▷ s'assure de n'exécuter cette fonction qu'une fois
2:     pos_new_node ← index(me.brothers[stage], new_node.id)
3:     if (pos_new_node > −1) then
4:         return
5:     end if
```

SHIFT_BRO() est appelée par CUT_NODE() qui est diffusée par TRANSFER(). On doit donc s'assurer de ne pas exécuter cette fonction deux fois.
Si *new_node* figure dans l'étage considéré, c'est que la fonction a déjà été exécutée et on arrête là.

Algorithme 9 : SHIFT_BRO (suite)

```
6:    pos_me ← index(me.brothers[stage], me.id)

7:    if (right = 1) then
8:        if (pos_me < b) then
9:            lost_id ← me.brothers[stage][pos_me + 1].id
10:           me.brothers[stage][pos_me + 1] ← me.brothers[stage][pos_me]
11:       else
12:           Affiche un message d'erreur et stoppe la procédure
13:       end if
14:   else
15:       if (pos_me > 0) then
16:           lost_id ← me.brothers[stage][pos_me − 1].id
17:           me.brothers[stage][pos_me − 1] ← me.brothers[stage][pos_me]
18:       else
19:           Affiche un message d'erreur et stoppe la procédure
20:       end if
21:   end if
```

Dans cette partie, on réalise le décalage après avoir mémorisé le nœud qui va disparaître, *lost_id* qui sera utilisé pour la mise à jour de ses prédécesseurs (ligne 24).

Pour l'instant, on se protège si le nœud courant se trouve aux extrémités, mais il faut se pencher sur ce cas.

Algorithme 9 : SHIFT_BRO (suite)

```
22:   me.brothers[stage][pos_me] ← new_node

                                  ▷ 'moi' doit être un prédécesseur de new_node
23:   SEND_MSG_ASYNC(new_node.id, add_pred(stage, me.id))
                                  ▷ 'moi' ne doit plus être un prédécesseur de lost_id
24:   SEND_MSG_ASYNC(lost_id, del_pred(stage, me.id))
25: end procedure
```

Pour finir, on insère *new_node*, et on met à jour les prédécesseurs.

1.3.10 La fonction DEL_MEMBER()

Cette fonction utilisée par CUT_NODE() (voir algo 8, ligne 32) permet de supprimer la partie comprise entre *start* et *end* de l'étage *stage*.

Algorithme 10 : Supprime une partie du groupe courant à un étage donné

```
1: procedure DEL_MEMBER(stage, start, end)
2:     nb_del ← end − start + 1
3:     if (nb_del = 0) then
4:         return
5:     end if
```

On arrête la fonction s'il n'y a rien à effacer.

Algorithme 10 : DEL_MEMBER (suite)

```
                                          ▷ mémorise les nœuds à effacer avant de commencer
6:     for i ← 0 to (nb_del − 1) do
7:         id_del[i] ← me.brothers[stage][start + i].id
8:     end for
```

On mémorise les nœuds à effacer avant de procéder aux modifications.

Algorithme 10 : DEL_MEMBER (suite)

```
                                                              ▷ boucle d'effacement
9:     for i ← 0 to (nb_del − 1) do
10:        if (id_del[i] <> me.id) then                       ▷ ne pas effacer 'moi'
11:            pos2del ← index(me.brothers[stage], id_del[i])

12:            if (pos2del < size(me.brothers[stage] − 1)) then
13:                for j ← pos2del to (size(me.brothers[stage] − 1)) do
14:                    me.brothers[stage][j] ← me.brothers[stage][j + 1]
15:                end for
16:            else
17:                j ← pos2del
18:            end if

19:            me.brothers[stage][j].id ← −1
                                          ▷ 'moi' n'est plus prédécesseur de id_del[i]
20:            SEND_MSG_ASYNC(id_del[i], del_pred(stage, me.id))
21:        end if
22:    end for
23: end procedure
```

Puis on procède aux effacements proprement dits.

1.3.11 La fonction ADD_BRO_ARRAY()

Cette fonction permet d'ajouter le groupe *bro* au groupe courant, à l'étage *stage*, du côté indiqué par *right*.

Algorithme 11 : Ajoute un groupe de nœuds donné au groupe courant

```
1:  procedure ADD_BRO_ARRAY(stage, bro, array_size, right)
2:      if (right = 0) then
3:          for i ← array_size − 1 to (0) do
4:              INSERT_BRO(stage, bro[i].id)                ▷ insertion des nœuds en début de liste
5:              SEND_MSG_ASYNC(bro[i].id, add_pred(stage, me.id))
6:          end for
7:      else
8:          for i ← 0 to (array_size − 1) do
9:              ADD_BROTHER(stage, bro[i].id)                ▷ ajout des nœuds en fin de liste
10:             SEND_MSG_ASYNC(bro[i].id, add_pred(stage, me.id))
11:         end for
12:     end if
13: end procedure
```

1.3.12 La fonction REPLACE_BRO()

Cette fonction remplace le membre situé à la position *init_idx* par *new_id*, à l'étage *stage*.

Algorithme 12 : Remplace un membre par un autre

```
1:  procedure REPLACE_BRO(stage, init_idx, new_id)
2:      if (me.brothers[stage][init_idx].id = new_id  or      ▷ new_id figure déjà à l'étage stage
3:          index_bro(stage, new_id) > −1) then
4:          return
5:      end if
```

Si *new_id* figure déjà à l'étage *stage*, c'est que la fonction a déjà été exécutée et on sort sans rien faire.

Algorithme 12 : REPLACE_BRO (suite)

```
6:      if (init_idx < size(me.brothers[stage])) then
7:          bro_id ← me.brothers[stage][init_idx].id
8:          me.brothers[stage][init_idx].id = new_id
9:      else
10:         ADD_BROTHER(stage, new_id)
11:     end if
```

Si *init_idx* désigne le prochain emplacement libre de l'étage, alors on ajoute simplement le nouveau membre avec ADD_BROTHER().

Sinon, le membre situé en $init\_idx$ est remplacé par new_id. L'ancien membre bro_id est mémorisé pour pouvoir mettre à jour ses prédécesseurs (lignes 13 et 16).

Algorithme 12 : REPLACE_BRO (suite)

```
12:     if (bro_id ≠ me.id and init_idx < size(me.brothers[stage])) then
13:         SEND_MSG_ASYNC(bro_id, del_pred(stage, me.id))
14:     end if

15:     if (new_id ≠ me.id) then
16:         SEND_MSG_ASYNC(bro_id, add_pred(stage, me.id))
17:     end if
18: end procedure
```

Les prédécesseurs sont mis à jour en conséquence.

se protéger contre le remplacement de me.id