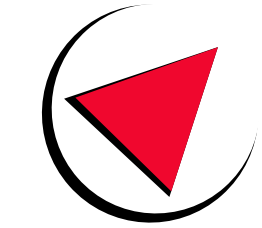


L I F C

Université de Franche-Comté

Contruction d'un DST : autres propositions

par Christophe Enderlin

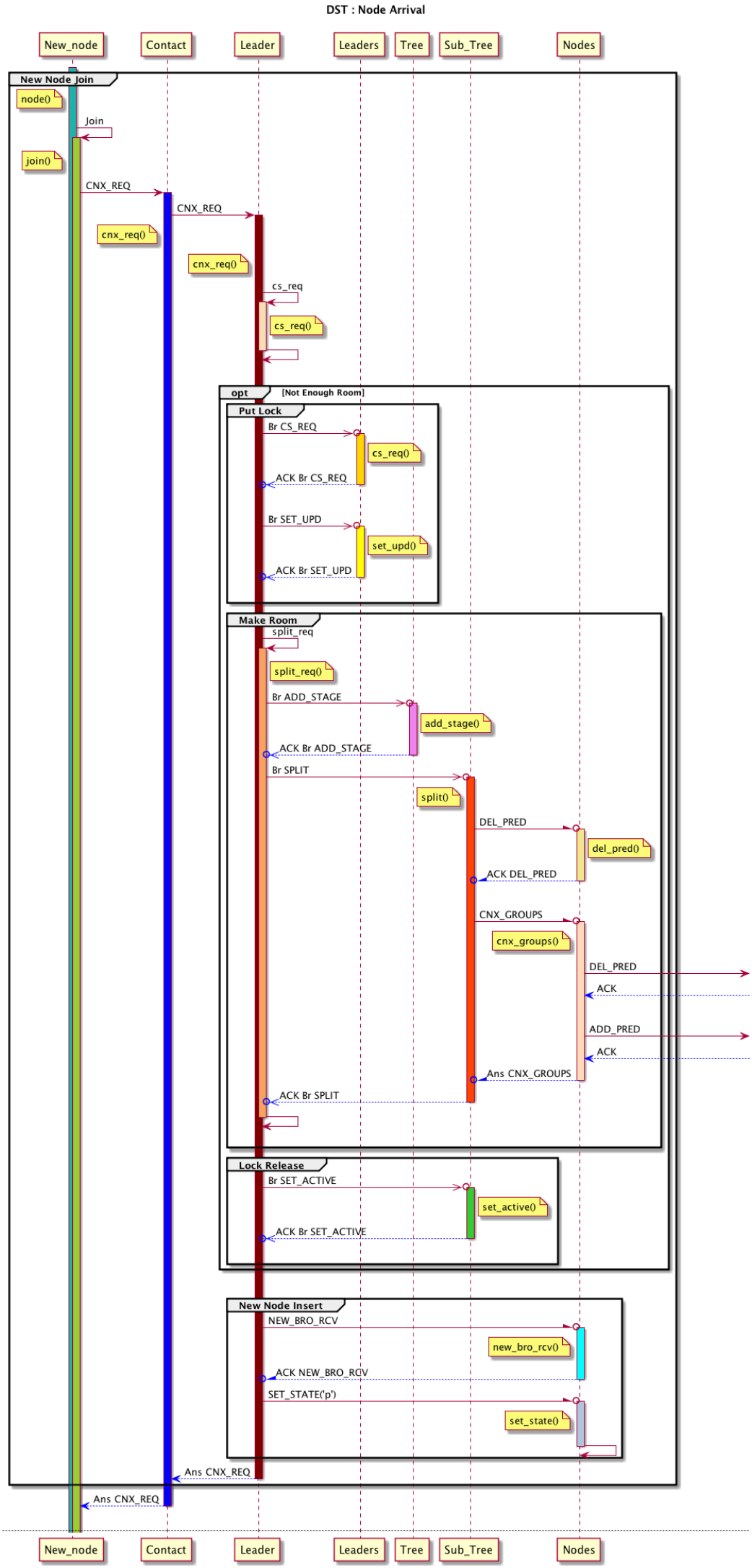
le 29 septembre 2014

Chapitre 1

CONTEXTE

1.1 RAPPEL

Pour mémoire, voici une représentation graphique du déroulement des opérations lors de l'insertion d'un nouveau nœud dans le DST, sans équilibrage de charge, pour simplifier. (voir page suivante)


DST : Node Arrival
New_node
Contact
Leader
Leaders
Tree
Sub_Tree
Nodes
New Node Join
node()
Join
join()
CNX_REQ
CNX_REQ
cnx_req()
cnx_req()
cs_req
cs_req()
opt
[Not Enough Room]
Put Lock
Br CS_REQ
cs_req()
ACK Br CS_REQ
Br SET_UPD
set_upd()
ACK Br SET_UPD
Make Room
split_req
split_req()
Br ADD_STAGE
add_stage()
ACK Br ADD_STAGE
Br SPLIT
split()
DEL_PRED
del_pred()
ACK DEL_PRED
CNX_GROUPS
cnx_groups()
DEL_PRED
ACK
ADD_PRED
ACK
Ans CNX_GROUPS
ACK Br SPLIT
Lock Release
Br SET_ACTIVE
set_active()
ACK Br SET_ACTIVE
New Node Insert
NEW_BRO_RCV
new_bro_rcv()
ACK NEW_BRO_RCV
SET_STATE('p')
set_state()
Ans CNX_REQ
Ans CNX_REQ
New_node
Contact
Leader
Leaders
Tree
Sub_Tree
Nodes

1.2 SITUATION

À l'issue de nombreux tests, la première version de mon simulateur a montré son manque de robustesse : lorsqu'un grand nombre de nouveaux arrivants impactent la même zone du DST à un même moment, les choses se passent mal. L'ensemble des problèmes observés semble pouvoir être ramené aux deux points suivants :

1.2.1 Le foisonnement des requêtes/réponses se passe mal

Voici un exemple de cas de figure qui pose problème :

2 nœuds intègrent le DST via un même contact :

- *Node* 14 → *Node* 121 → *Node* 42 (intégration rapide, pas de scission requise)
- *Node* 249 → *Node* 121 → *Node* 42 (scissions et ajout d'étage requis)
- Alors que *Node* 121 est en attente de `ACK_CNX_REQ/249` [1] de 42, il reçoit `ACK_CNX_REQ/14` de 42. Il le stocke donc puisque ce n'est pas la réponse qu'il attend.
- *Node* 14 ne recevant pas la réponse de 121 reste en '`b`'. À ce stade de l'intégration, il fait déjà partie du DST et il reçoit donc `ADD_STAGE/249` mais il ne peut y répondre $\Longrightarrow$ *deadlock*

Comme on le voit, le fait que `ACK_CNX_REQ/14` soit stocké par 121 pose problème ici.

Un seul process (hébergé par *Node* 121) chargé de l'intégration de deux nouveaux nœuds différents attend deux réponses. Le problème vient alors du fait qu'il ne peut pas traiter les réponses du premier pendant qu'il est occupé avec le deuxième. Ce mécanisme n'est donc pas correct.

On pense alors à deux solutions possibles : *a*) utiliser la fonction `MSG_comm_waitany()` de Simgrid (elle permet de réagir à une réponse parmi celles attendues) *b*) utiliser plusieurs process dédiés, chacun gérant ses propres requêtes/réponses .

La deuxième solution semble permettre de bien séparer les tâches, ce qui aurait un double avantage : arrêter de mélanger les requêtes/réponses pour différents nouveaux arrivants, et présenter une trace d'exécution plus lisible.

1.2.2 L'échec d'une diffusion d'un `SET_UPDATE` est mal géré

Pour rappel, lorsque l'arrivée d'un nouveau nœud requiert des scissions, on diffuse un `SET_UPDATE` sur l'ensemble du sous-arbre impacté dans le but de placer un verrou (l'état '`u`') sur l'ensemble de ses nœuds. Ainsi, les seules requêtes qu'ils accepteront seront celles qui concernent cet ajout (les autres sont soit refusées, soit différées).

1. Autrement dit, un accusé réception d'une requête de demande de connexion pour le nouveau nœud 249

Lorsque cette diffusion échoue (parce qu'on tombe sur une partie déjà verrouillée pour un autre arrivant, par exemple), on se retrouve dans la situation où une partie du sous-arbre a été verrouillée pour le nouveau nœud courant, et une autre pour un autre nouveau nœud. Il est donc important de remettre les choses en état (ôter les verrous posés par la diffusion en échec) pour ne pas bloquer les choses.

Les tests ont montré que la méthode utilisée pour cela n'est pas correcte puisqu'on arrive malgré tout à générer des *deadlocks*, les deux sous-arbres se bloquant mutuellement.

Il faut donc trouver des solutions pour ces deux problèmes.

1.3 Proposition de solutions

Solution au premier point, principes de base

- Chaque demande d'insertion de nouveau nœud est traitée par un process distinct créé pour l'occasion
- Ce sous-process ne peut pas recevoir de réponses à des requêtes qu'il n'a pas émises. Dit autrement, il doit (et peut) traiter immédiatement toutes les réponses reçues. (plus besoin de différer ou de refuser)
- Plusieurs nouveaux nœuds utilisant le même contact ne pouvant pas être traités simultanément, un mécanisme de file d'attente pour les traiter séquentiellement est mis en place.

De plus, j'ai fait le choix d'utiliser aussi un tel sous-processus pour les diffusions de `SPLIT` et de `CS_REQ` parce que là encore, cela semble être un avantage de ne pas mélanger les réponses attendues dans le cas de diffusions croisées.

Le schéma général de cette solution à process multiples est présenté en figure 1.1.

Remarque préliminaire : `launch_fork_process()` est une fonction chargée de choisir comment exécuter les requêtes qui lui sont transmises. Elle les confie soit à un nouveau process (cas de `CNX_REQ`, `BR_SPLIT`, `BR_CS_REQ`), soit au process courant.

On utilise au plus trois process par nœud :

1. `Main_Proc`
   Comme son nom l'indique, c'est le process principal :
   - il se charge des initialisations et de la création du process `Tasks_Queue` (qui tourne tout le temps de la simulation).
   - il héberge les files *tasks_ queue* (les tâches `CNX_REQ` en attente) et *delayed_ tasks* (les tâches différées)
   - lorsqu'il reçoit une requête, *a*) soit il la place dans la file *tasks_ queue* (cas des `CNX_REQ`), *b*) soit il la transmet à `launch_fork_process()`, *c*) soit il la place dans la file *delayed_ tasks*.

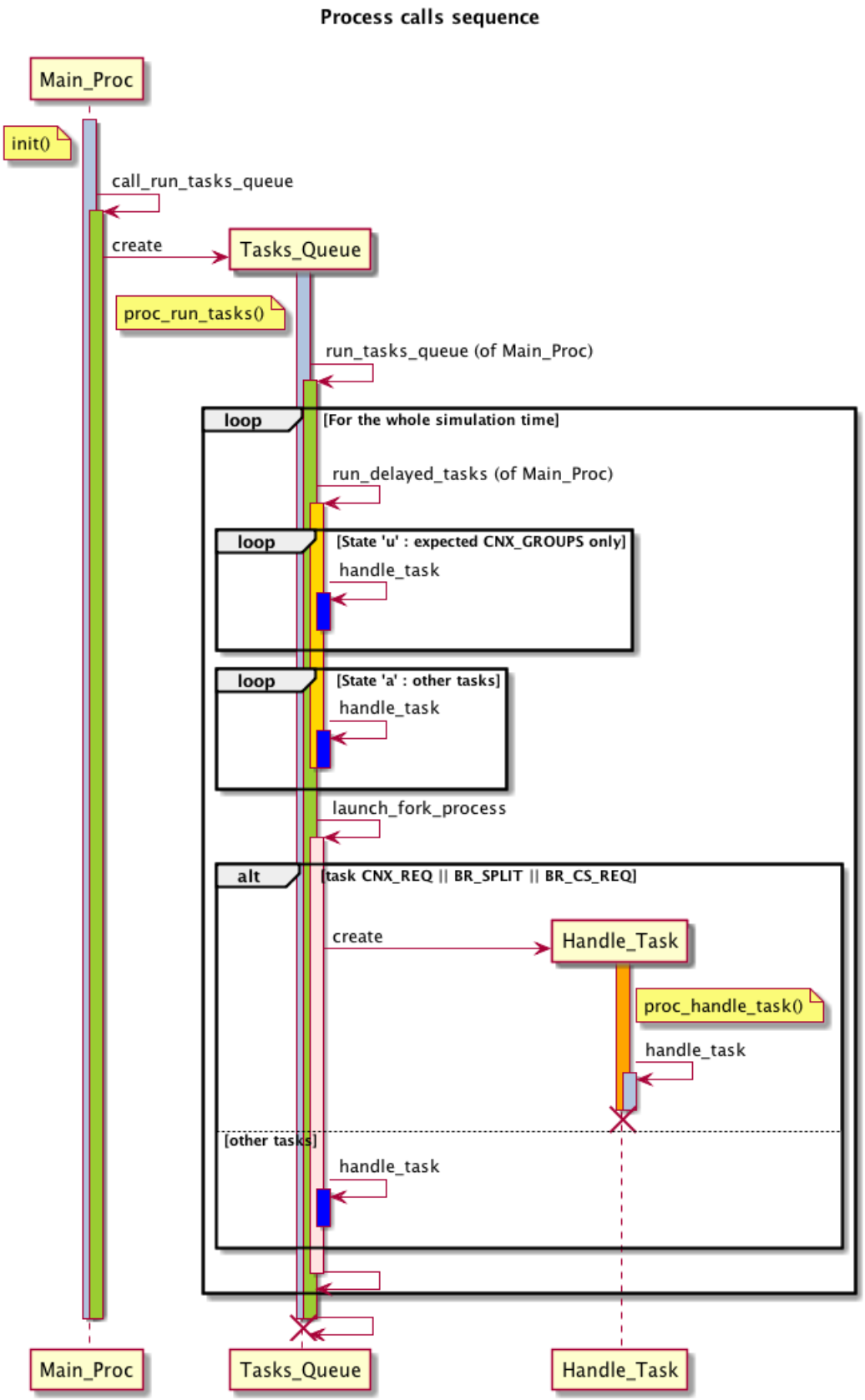


FIGURE 1.1 – Séquence d'appels des sous-process

– il héberge ses propres files *async_ answers* et *sync_ answers* (les réponses asynchrones et synchrones attendues aux requêtes qu'il a émises)

2. `Tasks_Queue`

   Ce process est chargé d'exécuter la fonction `run_tasks_queue()` qui traite les files *tasks_ queue* et *delayed_ tasks*. Il héberge aussi ses propres files *async_ answers* et *sync_ answers*.

3. `Handle_Task`

   C'est ce process qui est éventuellement créé par `launch_fork_process()` depuis `run_tasks_queue()` pour traiter les requêtes concernées. Il héberge aussi ses propres files *async_ answers* et *sync_ answers*.

1.3.1 Dans le détail

run_tasks_queue()

La file *tasks_ queue* contient toutes les demandes de connection (c'est à dire toutes les tâches `CNX_REQ`) reçues par le nœud qui héberge cette file. `run_tasks_queue()` est donc chargée de traiter les requêtes présentes dans cette file, par ordre de priorité. [2]

Après avoir traité les tâches différées (voir détails `run_delayed_tasks()` plus loin), on s'occupe de la file *tasks_ queue* proprement dite, à condition d'être actif (à l'état '`a`').

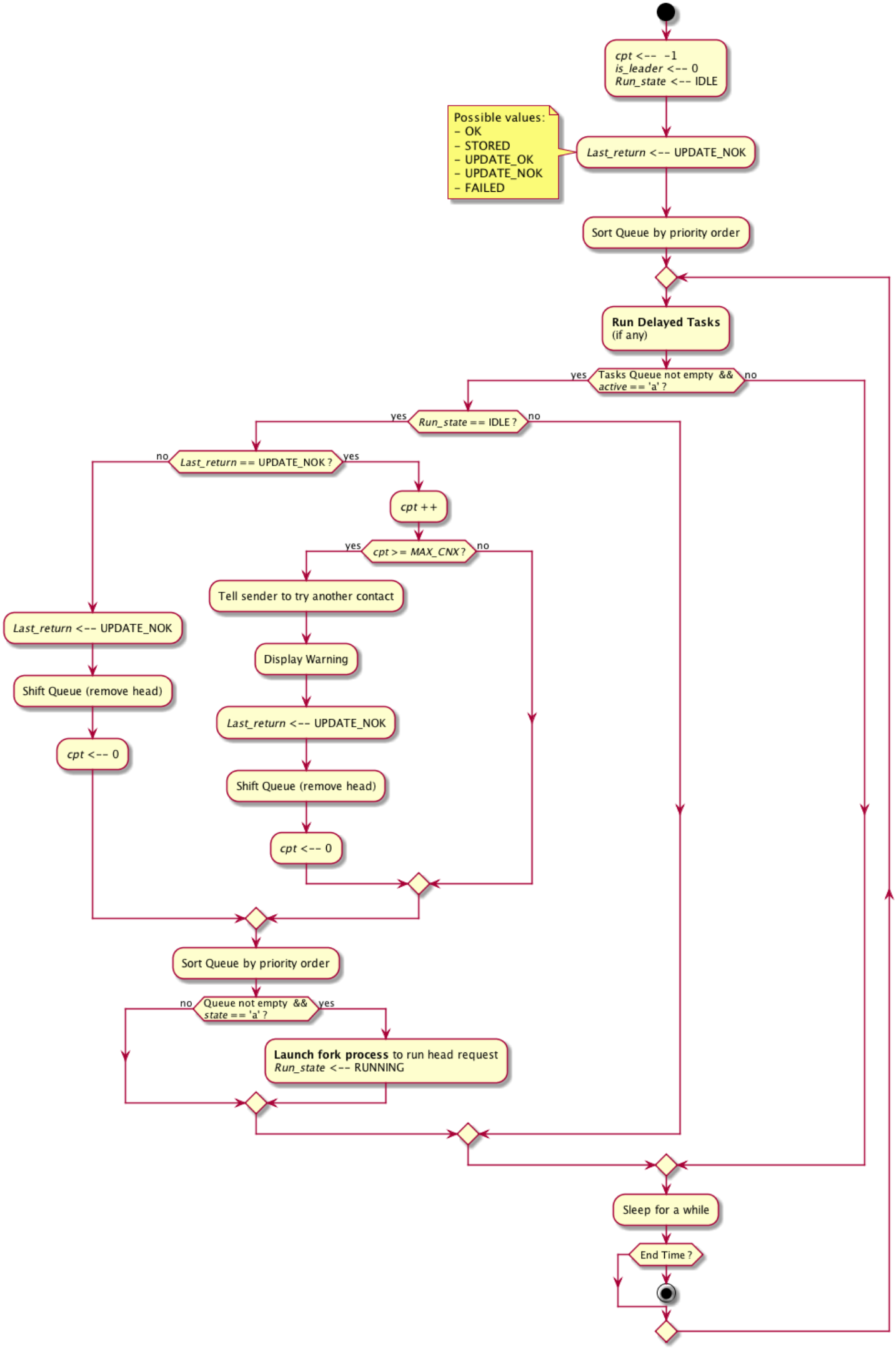
Le nœud courant possède deux variables "de nœud" (c'est à dire globales à tous les process hébergés par ce nœud) :

- *Run_ state* : Chaque requête de cette file sera traitée par un process distinct, mais il n'est pas possible ici d'insérer plus d'un nouveau nœud à la fois et il faut donc que ces process s'exécutent séquentiellement. C'est la raison d'être de cette variable *Run_ state* : pour s'occuper de la tâche suivante, la tâche courante doit être terminée. (valeur *IDLE*)
- *Last_ return* : Cette variable contient la valeur de retour de l'exécution courante de `connection_request()`.

  Il y a deux sortes d'échec d'insertion possibles. Lors de l'arrivée d'un nouveau nœud, on a la séquence d'appels suivante : *nouveau nœud* → *contact* → *leader*. En cas d'échec, le contact doit refaire une tentative plus tard, mais son leader peut avoir changé entre temps. Le leader doit donc dépiler la requête alors que le contact doit la conserver dans sa file *tasks_ queue*. Deux valeurs de retour en cas d'échec sont alors requises : un leader retourne la valeur *FAILED* alors qu'un contact retournera la valeur *UPDATE_ NOK*.

Comme on peut le voir, dans le cas général (i.e. $cpt < MAX\_CNX$), une valeur de retour *UPDATE_ NOK* laisse la requête dans la file pour une nouvelle tentative un peu plus tard ("*Sleep for a while*"). Dans le cas contraire, on dépile pour passer à la requête suivante. (On voit donc qu'en cas de retour *FAILED*, la requête est bien dépilée.)

2. voir plus loin les remarques à ce sujet


cpt <-- -1
is_leader <-- 0
Run_state <-- IDLE
Possible values:
- OK
- STORED
- UPDATE_OK
- UPDATE_NOK
- FAILED
Last_return <-- UPDATE_NOK
Sort Queue by priority order
Run Delayed Tasks
(if any)
Tasks Queue not empty && active == 'a' ?
yes
no
Run_state == IDLE ?
yes
no
Last_return == UPDATE_NOK ?
yes
no
cpt ++
cpt >= MAX_CNX ?
yes
no
Tell sender to try another contact
Display Warning
Last_return <-- UPDATE_NOK
Shift Queue (remove head)
cpt <-- 0
Last_return <-- UPDATE_NOK
Shift Queue (remove head)
cpt <-- 0
Sort Queue by priority order
Queue not empty && state == 'a' ?
no
yes
Launch fork process to run head request
Run_state <-- RUNNING
Sleep for a while
End Time ?

En détails : À l'issue de l'exécution de `connection_request()`, une réponse est envoyée à l'émetteur seulement dans les cas où la requête est ici dépilée.

Lors d'un échec, voici alors ce qu'il se passe dans la séquence de retour *leader* → *contact* → *nouveau nœud* : le leader reçoit *FAILED* comme valeur de retour de `connection_request()`, il dépile donc sa requête et répond *UPDATE_ NOK* à son contact. Celui-ci ne dépile rien et ne répond pas à *nouveau nœud*. La requête restant dans la file s'exécutera alors au plus *MAX_ CNX* fois.

Si ce nombre est atteint, on décide alors de dépiler tout de même la requête et de répondre *UPDATE_ NOK* au nouveau nœud. Il peut ainsi détecter l'échec et refaire d'autres tentatives avec d'autres contacts. (ces contacts sont alors choisis aléatoirement parmi les nœuds déjà intégrés au DST. Il s'agit d'un tableau global, donc introduisant un peu de centralisation dans cet algorithme. [3])

Que la file soit dépilée ou pas, elle est à nouveau triée par ordre de priorité à ce moment-là. En effet, pendant le temps d'exécution de `connection_request()`, d'autres demandes d'insertion ont pu arriver dans la file et il faut s'assurer que la prochaine à exécuter soit bien la suivante en termes de priorité.

L'exécution d'une requête de la file est confiée à `launch_fork_process()` (voir plus haut)

run_delayed_tasks()

Cette fonction est chargée de traiter la file des tâches différées (*delayed_ tasks*). Lorsqu'une tâche ne peut pas être exécutée par un nœud, soit elle est refusée (et retournée à l'émetteur), soit elle est stockée dans cette file pour être exécutée plus tard. Cette fonction est donc appelée périodiquement. (Voir `run_tasks_queue()`)

On distingue deux cas : soit le nœud courant est à l'état '`u`', soit il est à '`a`'.

état '`u`' (voir partie 1 - figure 1.2)

Variables :

- *nb_ elems* : contient le nombre d'éléments de la file au lancement de `run_delayed_tasks()`
- *cpt* : itérateur (de 0 à *nb_ elems*)

Un nouveau nœud est alors en cours d'insertion et il faut examiner ce cas en premier pour minimiser les risques de blocages. Les seules tâches exécutables ici sont celles qui pourraient permettre de terminer cette insertion, c'est à dire les `CNX_GROUPS` pour le même nouveau nœud que celui en cours d'insertion. (c'est le test `task.args.new_node_id == state.new_node_id`)

On parcourt donc la file à la recherche de ces tâches pour les exécuter. (`handle_task(task)`)

3. Voir commentaires en conclusion

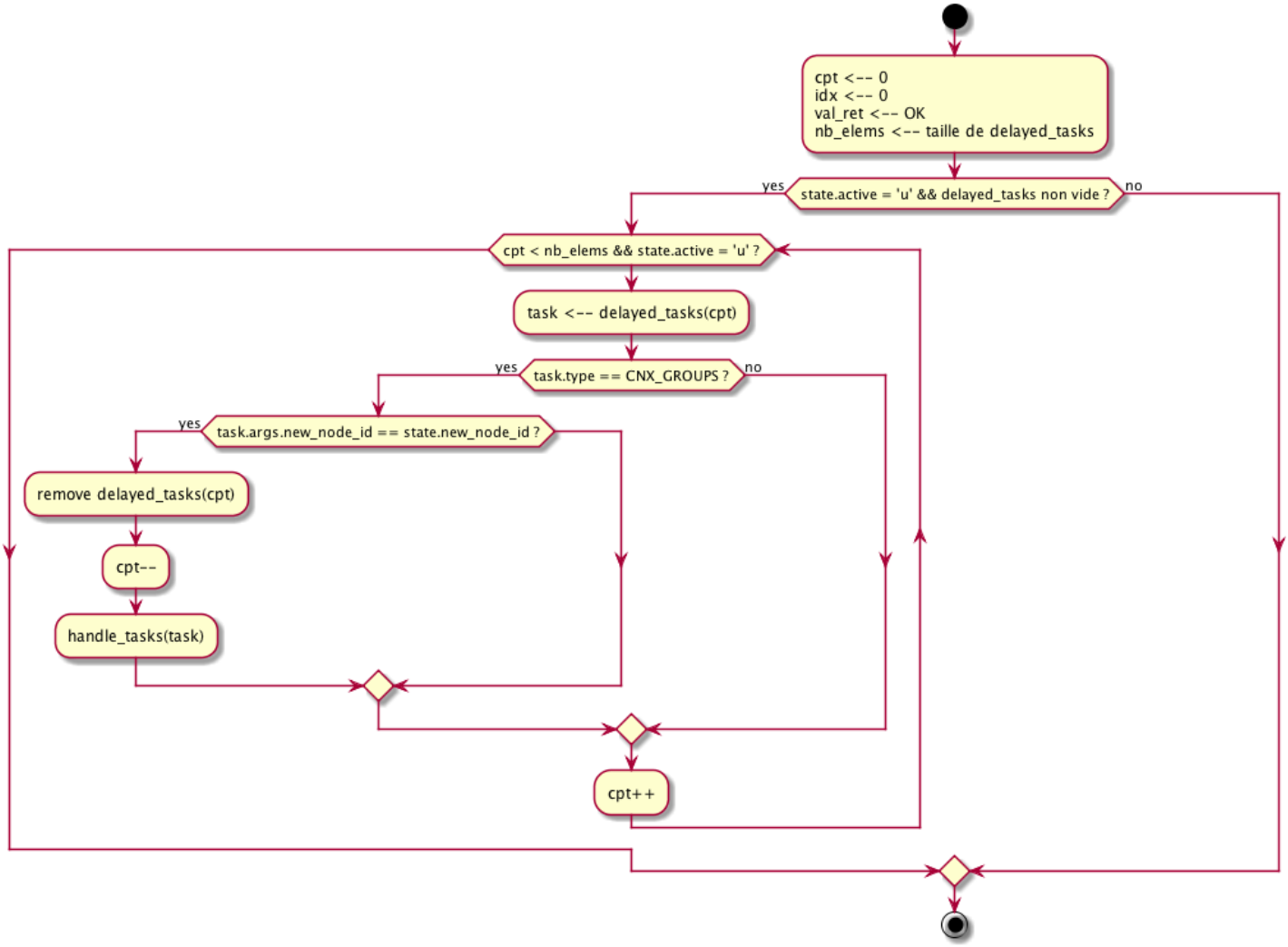


FIGURE 1.2 – Traitement des tâches différées - Partie 1

Lorsqu'on en trouve une, on peut l'ôter de la file sans s'assurer que son exécution a réussi ou pas puisque si elle échoue, elle est à nouveau stockée dans la file.

À noter : à l'issue de la fonction `handle_tasks()`, le nombre de tâches de la file a pu augmenter. Pourtant, on choisi de ne pas mettre à jour la variable *nb_ elems* ici pour ne pas risquer une boucle sans fin. Si des tâches ont été ajoutées entre temps, elles seront traitées lors d'une prochaine exécution de `run_delayed_tasks()`.

état 'a' (voir partie 2 - figure 1.3)

Ici, le nœud courant est prêt à exécuter n'importe quelle autre tâche et on peut traiter le reste de la file.

Variables :

- *nb_ elems* : le nombre d'éléments restants de la file.
- *idx* : itérateur (de 0 à *nb_ elems*)
- *is_ contact* : vaut 1 si le nœud courant est le contact direct du nouveau nœud. (autrement dit, si l'émetteur de la tâche à exécuter est le nouveau nœud)
- *buf_ new_ node_ id* : `task.args.new_node_id` est mémorisé dans cette variable parce qu'on en a besoin après la destruction de task.

On ôte la tâche courante de la file dans les cas suivants :

- l'exécution a réussi (*OK* et *UPDATE_ OK*)
- la tâche a été stockée à nouveau (*STORED*)
- l'exécution a échoué mais le nœud courant n'est pas le contact direct du nouveau nœud. (`UPDATE_NOK && !is_contact`) Il s'agit du cas *FAILED* décrit plus haut.

À noter : si le nœud courant était verrouillé pour le même nouveau nœud que celui dont la tâche vient d'échouer, alors on ôte le verrou.

À l'issue de l'exécution de cette boucle (`idx == nb_elems`), la file n'est pas forcément vide. Elle contient toutes les tâches dont l'exécution a échoué et celles qui ont été stockées – y compris par d'autres process – entre temps. Cette boucle est alors à nouveau exécutée jusqu'à ce que *nb_ elems* soit à 0, c'est dire jusqu'au succès de toutes les tâches qui se trouvaient dans cette file au moment de cet appel de `run_delayed_tasks()`. Comme indiqué précédemment, les tâches ajoutées entre temps ne seront pas traitées lors de cette exécution.

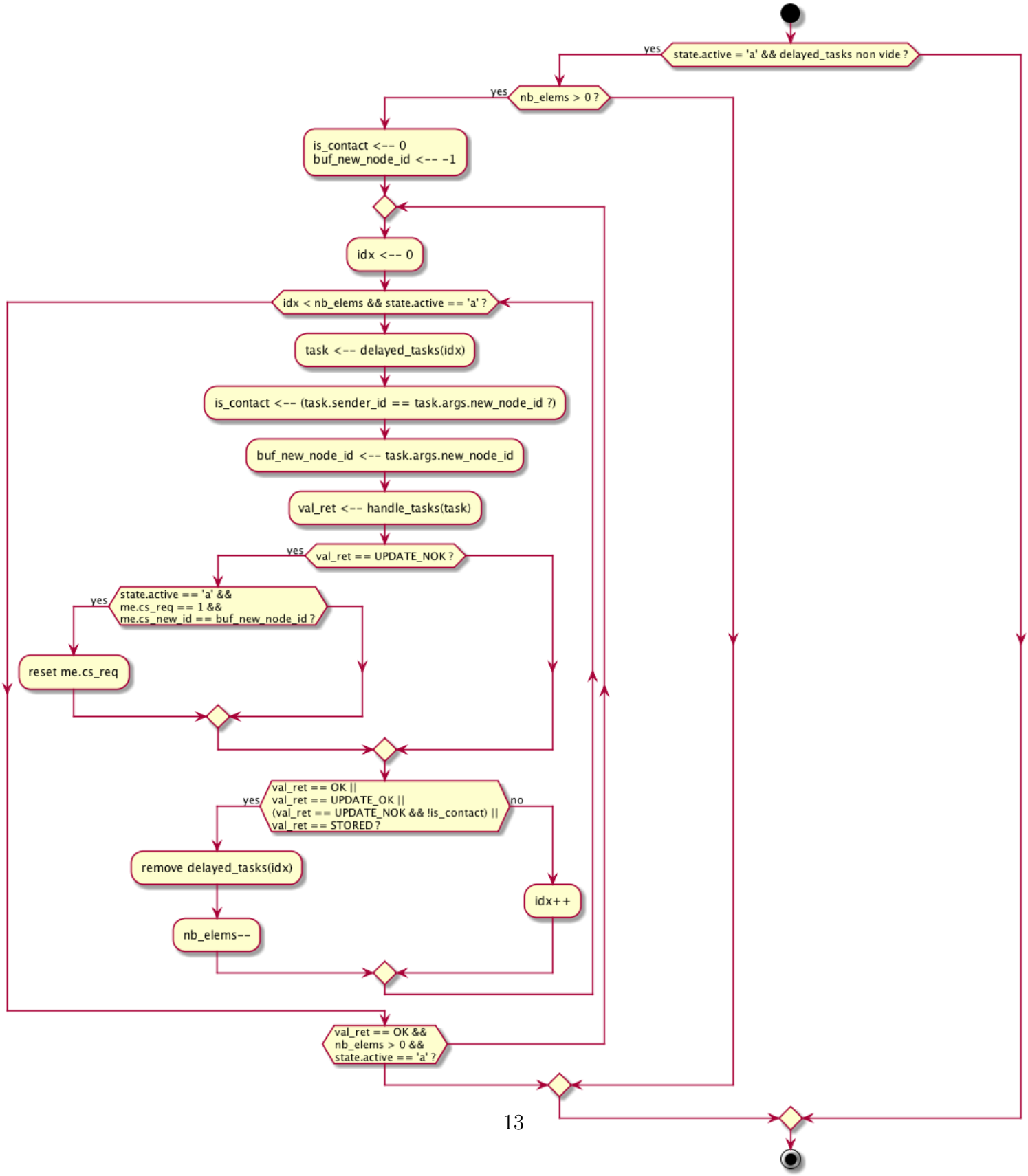


FIGURE 1.3 – Traitement des tâches différées - Partie 2