Université de Franche-Comté

Construction d'un DST : autres propositions

par Christophe Enderlin

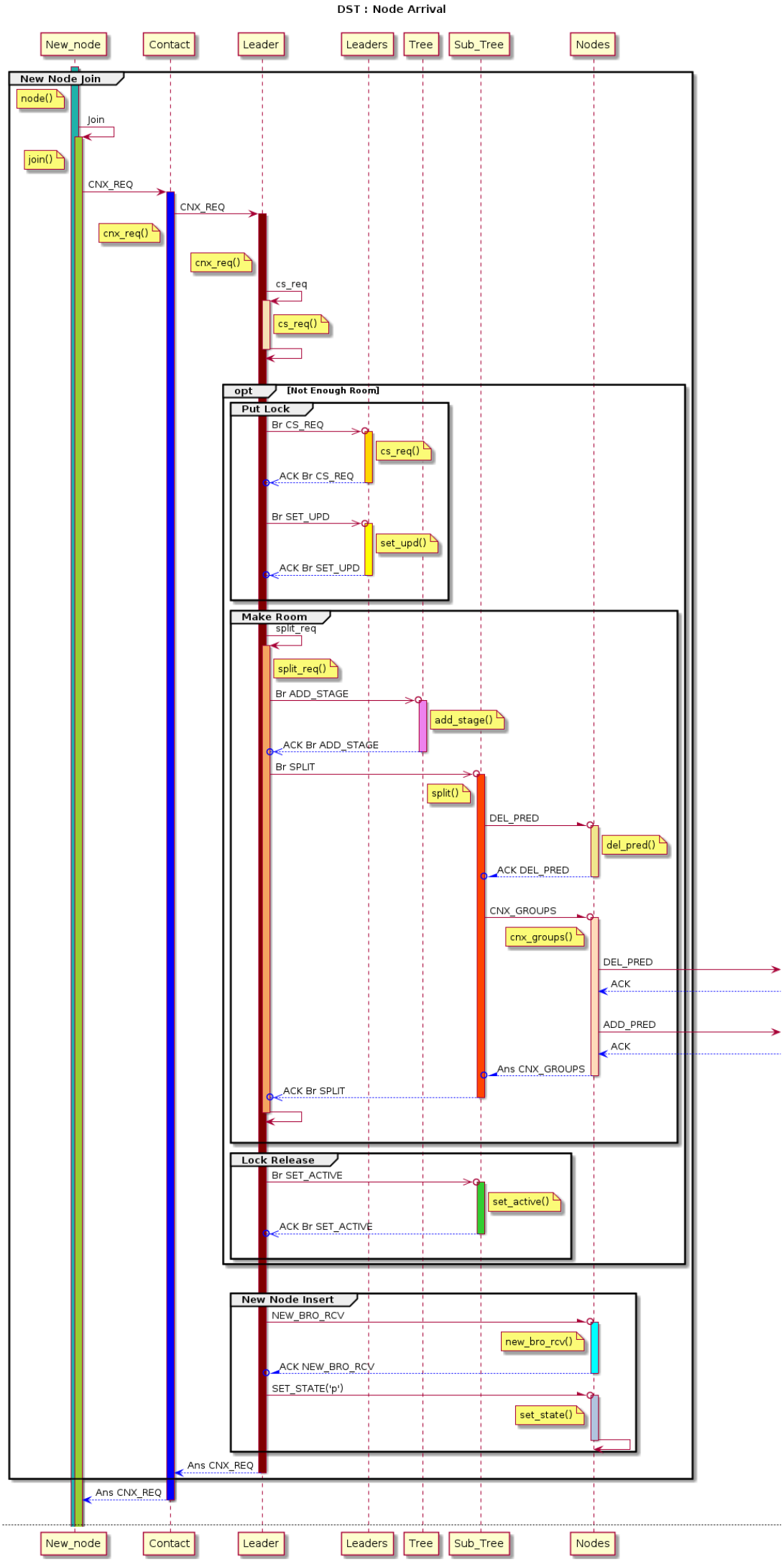
le 12 octobre 2014

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).

Lorsque cette diffusion échoue (parce qu'on tombe sur une partie déjà verrouillée pour un autre arrivant, par ex.), 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 à générer des *deadlocks*, les deux sous-arbres se bloquant mutuellement.

Il faut donc trouver des solutions pour ces deux problèmes.

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

Chapitre 2

Foisonnement des messages : solution

Principes de base :

- Chaque demande d'insertion de nouveau nœud (`CNX_REQ`) 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 peut (et doit) 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 figure 2.1 - page 5.

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.

Explications : On utilise au plus trois process par nœud.

1. `Main_Proc`
   - 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` reçues en attente de traitement) 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*.
   - 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*.

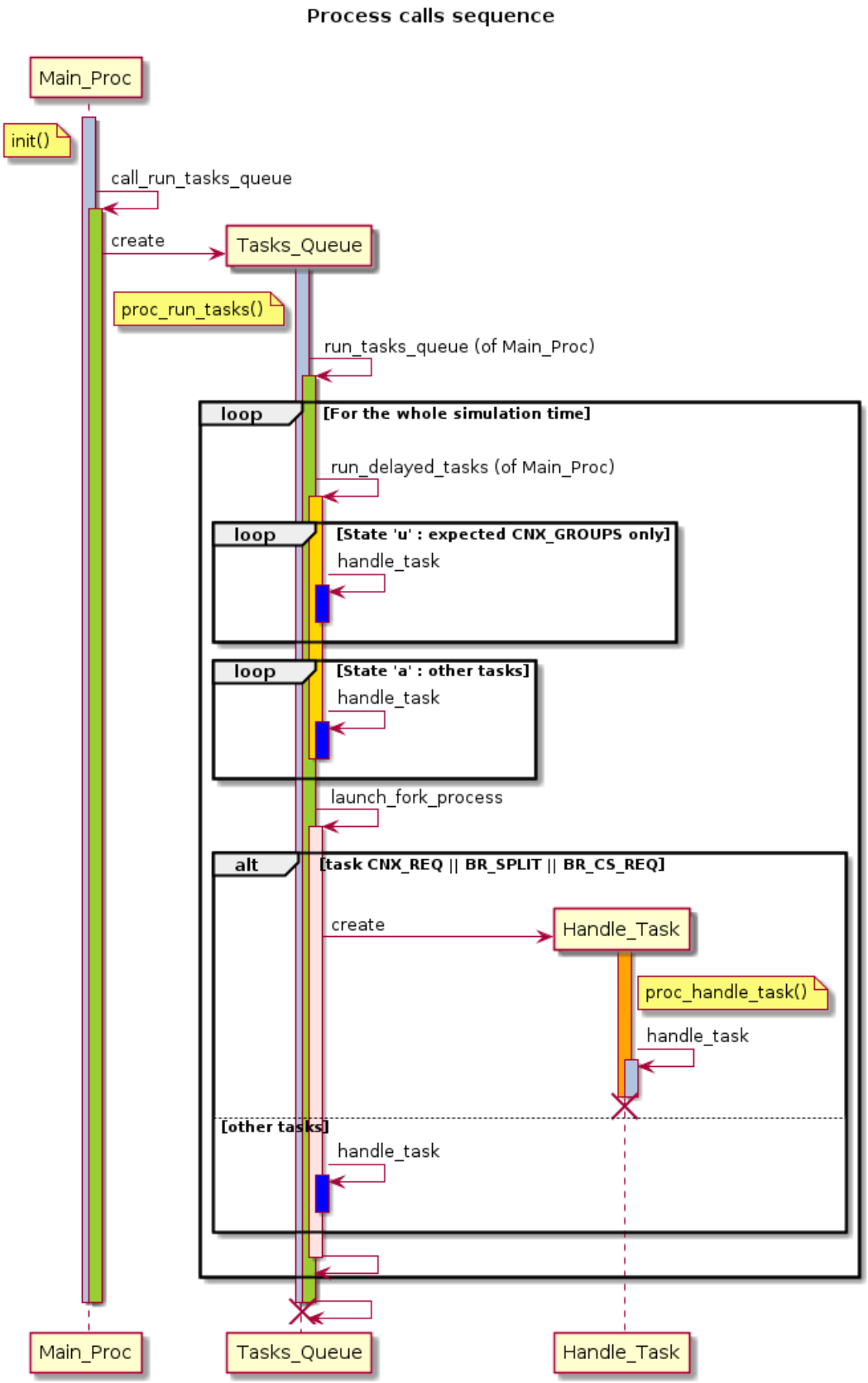


FIGURE 2.1 – Séquence d'appels des sous-process

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*.

2.1 Les fonctions de base

2.1.1 run_tasks_queue()

Voir figure 2.2 - page 7.

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. [1]

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.)

1. voir plus loin les remarques à ce sujet

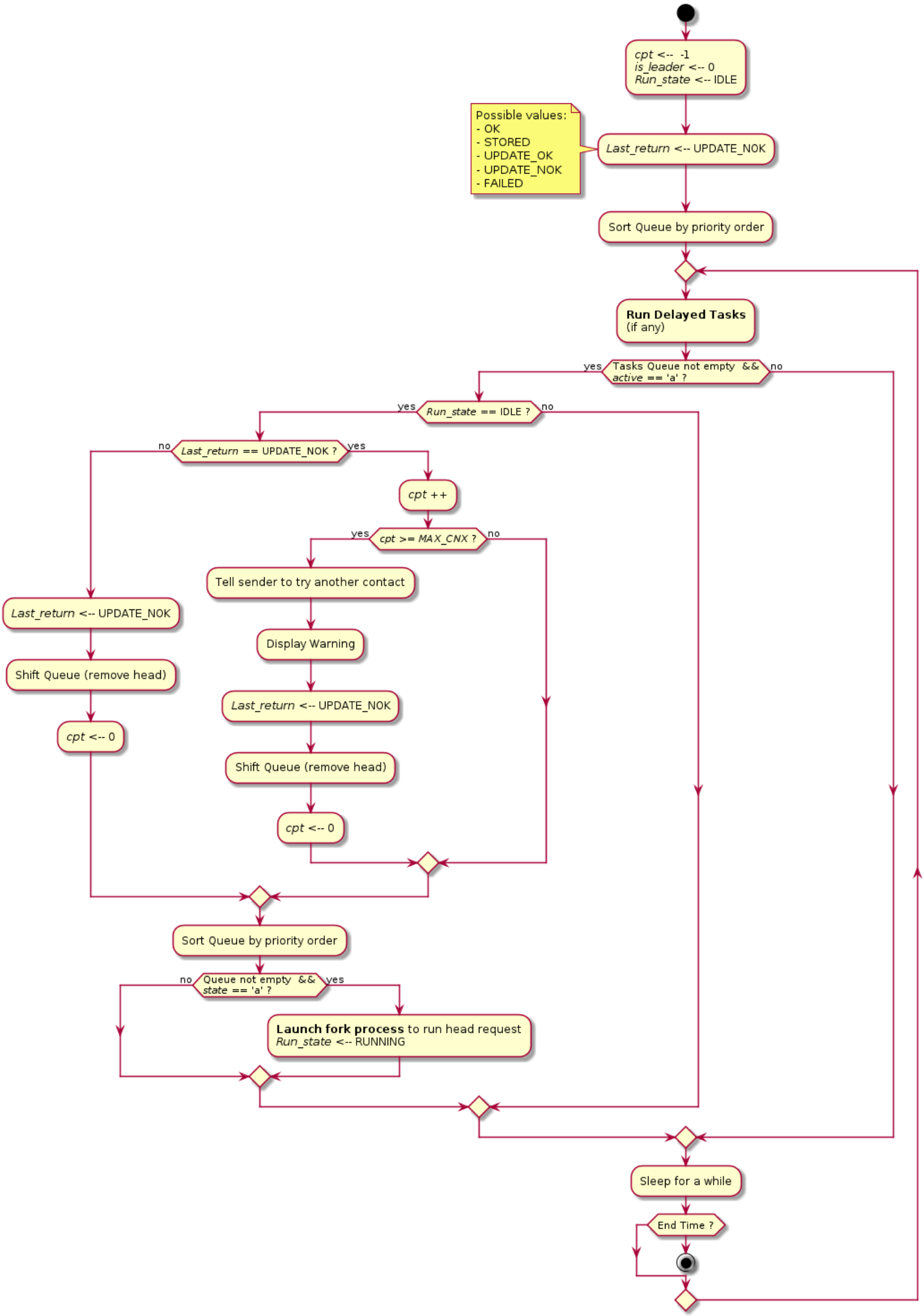


FIGURE 2.2 – Algo `run_tasks_queue()`

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. [2])

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)

À propos de l'ordre d'exécution des tâches dans *tasks_queue*

Dans le cas particulier de la simulation réalisée avec Simgrid, le contact de chaque nouveau nœud est choisi aléatoirement parmi les nouveaux nœuds précédents. Mais au moment où ils sont utilisés comme contact, rien ne garanti qu'ils soient déjà intégrés au DST. S'ils ne le sont pas, les demandes d'insertion sont remises à plus tard et chaque insertion retardée retarde d'autant l'insertion des nœuds qui en dépendent. On aboutit alors à un empilement important des requêtes qui peut se traduire par un temps de simulation trop long, voire même à des blocages.

Afin d'éviter au maximum ce cas de figure, il est donc important, pour une file donnée, de traiter les demandes de connexion des nouveaux nœuds par ordre de priorité. Pour réaliser cela, un numéro de priorité est attribué à chaque nouveau nœud lors de son arrivée. Cette priorité est utilisée à deux moments : lors du tri des files *tasks_queue* selon cet ordre, et lors d'un conflit (deux requêtes pour deux nouveaux nœuds différents sont reçues par le même nœud), on laisse passer en priorité celui qui a le numéro le plus petit.

Un peu de centralisation

Dans le même ordre d'idée, on surveille le nombre de tentatives d'insertion d'un nouveau nœud. S'il devient trop important, on fourni à ce nouveau nœud un nouveau contact, provenant, cette fois, d'une liste de nœuds déjà intégrés au DST. Cette liste devant être accessible à l'ensemble des nœuds, il s'agit d'une donnée globale, donc centralisée.

Le but de ce mécanisme étant juste d'améliorer le comportement du simulateur même et pas de valider les algorithmes du DST proprement dits, je pense qu'on peut raisonnablement le laisser en place sans remettre en cause le caractère décentralisé de ces algorithmes.

2. Voir détails plus bas

2.1.2 run_delayed_tasks()

Voir figures 2.3 et 2.4 - pages 9 et 11.

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 (charge alors à l'émetteur de réitérer sa demande plus tard), soit elle est stockée dans cette file pour être exécutée plus tard. Cette fonction est donc appelée périodiquement. (depuis `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 2.3 - page 9)

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)`)

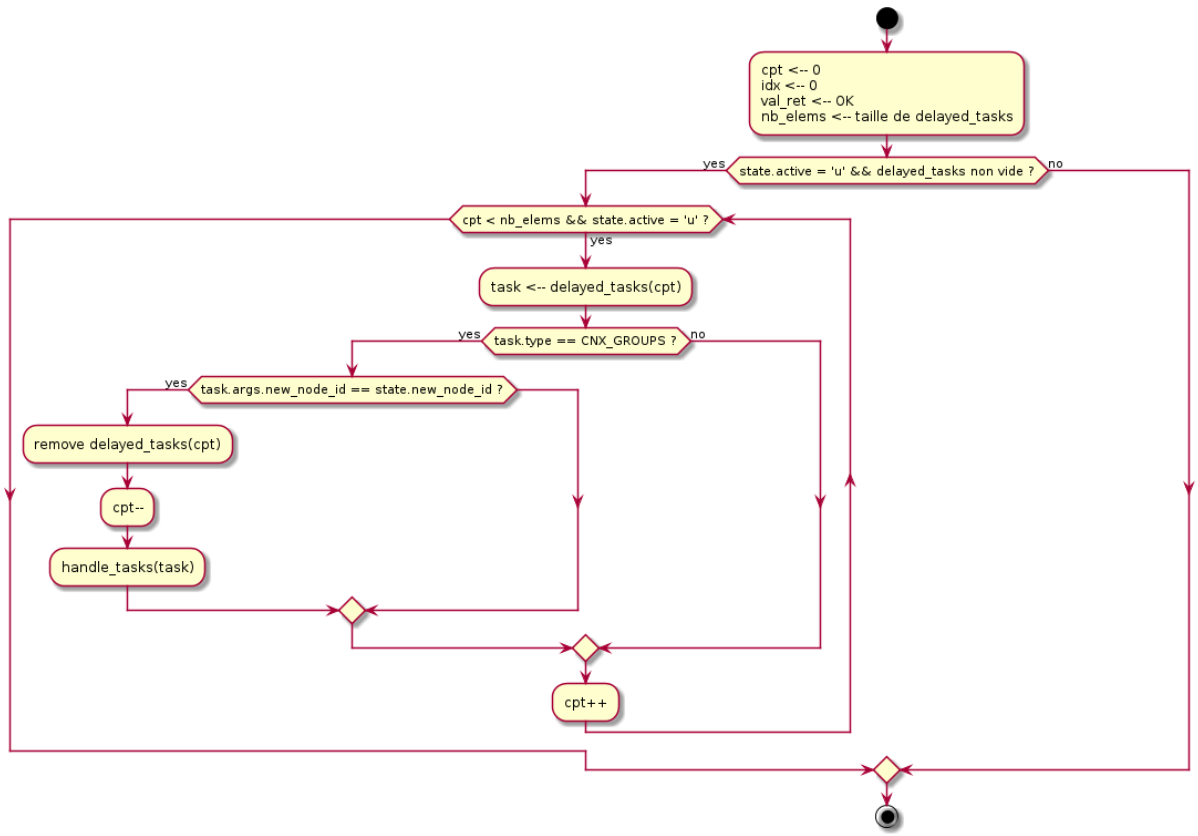


FIGURE 2.3 – Algo `run_delayed_tasks()` - Partie 1

Lorsqu'on en trouve une, on peut l'ôter de la file sans se soucier si 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 2.4 - page 11)

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. [3]

À l'issue de l'exécution de cette boucle (`idx == nb_elems`), si tout s'est bien passé (`val_ret == OK`), on choisi de parcourir à nouveau la file jusqu'à ce que *nb_ elems* soit nul. C'est à dire jusqu'à ce qu'on ait ôté de la file autant de requêtes qu'elle en contenait au début de l'exécution de `run_delayed_tasks()`. Comme déjà indiqué, les tâches qui auraient éventuellement été stockées dans la file entre temps seront traitées lors d'une prochaine exécution de `run_delayed_tasks()`.

3. Il s'agit d'ôter ce verrou au plus tôt, voir explications sur le verrou plus loin.

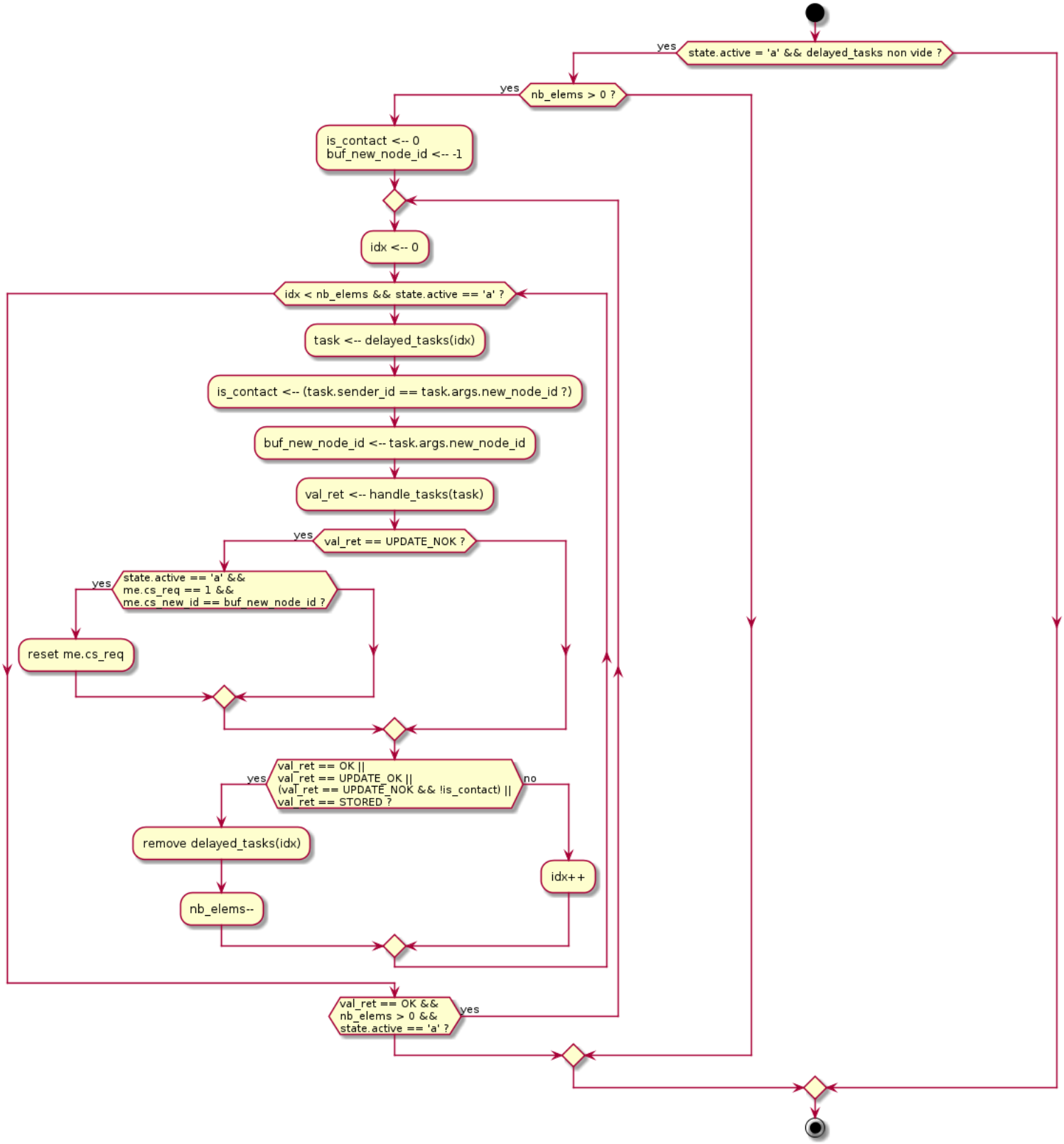


FIGURE 2.4 – Algo `run_delayed_tasks()` - Partie 2

2.1.3 launch_fork_process()

Voir figure 2.5 - page 13.

Si la tâche transmise à cette fonction n'est ni une demande de connexion (`CNX_REQ`), ni une diffusion de `SPLIT` ou de `CS_REQ`, alors elle est exécutée localement, c'est à dire par le process courant. Sinon, elle est confiée à un nouveau process créé pour l'occasion.

Ce nouveau process doit posséder ses propres files d'attente de réponses attendues (*async_answers* et *sync_answers*) de sorte qu'elles ne puissent plus être mélangées avec celles requises pour l'insertion d'un autre nouveau nœud.

Il est labelisé ainsi : `xxx-nom_process-yyy` où :

- `xxx` est l'id du nœud courant
- `nom_process` est le nom attribué par cette fonction (tel que `"Cnx_req"` ou `"Br_split"`, etc)
- `yyy` est l'id du nouveau nœud en cours d'insertion

On s'assure ainsi de l'unicité de l'étiquette de ce process.

Dans le cas où la tâche est une diffusion de `CS_REQ` (demande d'entrée en section critique), on commence par s'assurer que le nœud courant est disponible (c'est à dire répondrait favorablement à un `CS_REQ`[4]) pour ne pas créer de process fils inutilement. En cas de non-disponibilité, il faut en informer l'émetteur de la requête.

2.2 Les fonctions de communication

2.2.1 synchrone - `send_msg_sync()`

La requête synchrone est utilisée lorsqu'une réponse est attendue pour poursuivre l'exécution du programme.

Cas d'emplois principaux : [5]

- `CNX_REQ` (fonction `join()`) La réponse attendue est la table de routage du contact qui a réussi l'insertion du nouveau nœud
- `BROADCAST` (fonction `handle_task(): BROADCAST`) La réponse attendue ici est la valeur de retour de la fonction `handle_task()`, c'est à dire le succès ou l'échec de la requête diffusée.

De plus, on utilise aussi ce type de requête pour retransmettre au leader les requêtes `CNX_REQ` et `SPLIT_REQ`. [6]

L'attente de la réponse ne devant pas être bloquante, toutes les requêtes ou autres réponses arrivant dans l'intervalle doivent être traitées.

4. Voir explications sur les verrous plus loin.
5. Pour simplifier, la phase d'équilibrage de charge est ignorée ici.
6. Pas forcément utile dans le cas de `SPLIT_REQ`, à voir.

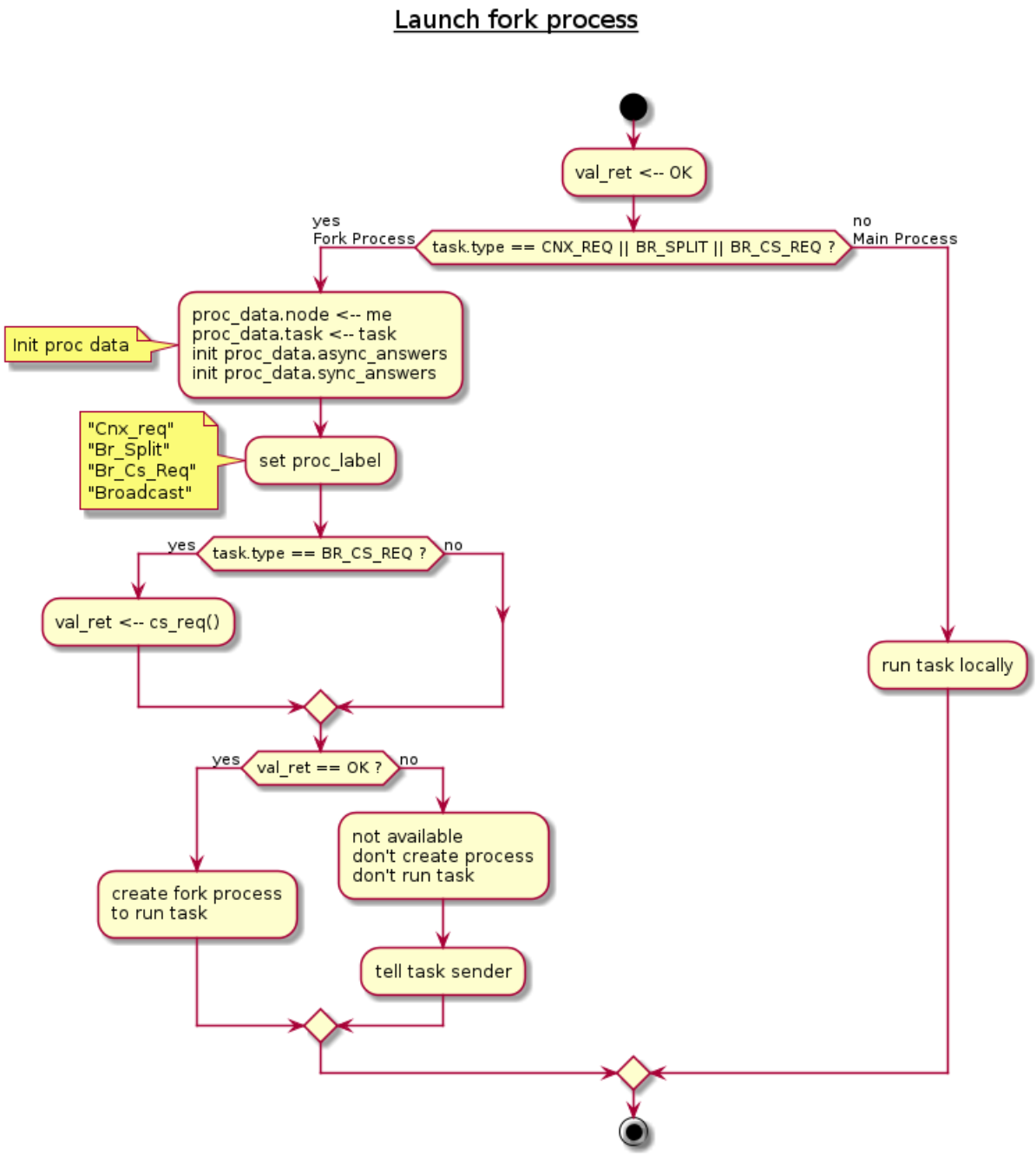


FIGURE 2.5 – Algo `launch_fork_process()`

Déroulement simplifié des opérations

L'émetteur de la requête appelle `send_msg_sync()` qui exécute les tâches suivantes :

- crée la requête
- encapsule la requête dans une tâche
- empile la requête sur *sync_ answers*
- envoie la tâche au destinataire
  (c'est à dire au process principal *Main proc*[7] du nœud destinataire)
- attend la réponse
  (sur le process courant qui peut être l'un des process présentés sur la figure 2.1, page 5)
- traite tout ce qui est reçu qui n'est pas la réponse
- dès que la réponse est reçue, dépile la requête de *sync_ answers*
- retourne la réponse à l'appelant

Dans le détail

La fonction `send_msg_sync()` est chargée d'envoyer une requête à un autre nœud, puis d'en attendre la réponse.

Chaque process hébergé par un nœud possède deux files : *sync_ answers* et *async_ answers*.

Ces deux files servent à stocker les requêtes synchrones et asynchrones (respectivement) émises, le temps qu'elles reçoivent une réponse et qu'elle soit lue. Ces requêtes sont dépilées dès que leur réponse a été prise en compte. [8]

Lorsqu'une réponse est reçue, il y a trois possibilités :

1. il s'agit de **la réponse attendue** : Dans ce cas, la réponse est prise en compte, la requête correspondante est dépilée et le programme se poursuit.
2. il s'agit d'**une autre réponse attendue** : Elle est alors enregistrée dans la bonne pile, avec la requête correspondante, et on se remet en écoute.
3. il ne s'agit d'**aucune réponse attendue** : Elle est simplement ignorée (cas de certains accusés réception, par exemple)

Première partie envoi de la requête (Voir figure 2.6 - page 16)

Après avoir construit la tâche contenant la requête, celle-ci est envoyée au moyen de la fonction `isend()` de Simgrid, puis stockée dans la pile *sync_ answers* du process courant. On attend ensuite la fin de la communication.

Deux traitements différents des cas d'échecs :

- la communication ne se termine pas (cas du **process** destinataire arrêté) ou elle a le statut `MSG_TRANSFER_FAILURE` (cas de l'**hôte** destinataire arrêté) : La fonction s'arrête en retournant une erreur de transmission.

7. voir figure 2.1, page 5
8. Pour mémoire, la réponse à une requête synchrone contient les données demandées alors que la réponse à une requête asynchrone est simplement un accusé réception.

– la communication se termine avec le statut `TIMEOUT` (cas d'un destinataire trop occupé, par exemple) : On recommence l'émission un certain nombre de fois (*loop_ cpt*). Si *max_ loops* est atteint, la fonction est également arrêtée avec une erreur de transmission.

Deuxième partie réception de la réponse
(Voir figures 2.7 et 2.8 - pages 17 et 18)

Dans cette partie, le *process courant* se met en écoute de la réponse. En cas d'erreur de communication (`res != MSG_OK`), on la signale et on se remet en écoute, sauf s'il s'agit d'un `TIMEOUT` pour une requête `GET_REP`.[9] Dans ce cas, on arrête la fonction là en retournant simplement l'erreur pour la signaler à l'appelant.

Deux types de tâches peuvent être reçus : soit une requête, soit une réponse.

Requête : Si c'est une demande de connexion (`CNX_REQ`), elle est simplement stockée dans *tasks_ queue*.[10] Sinon, on l'exécute avec `handle_task()`.

Ensuite, on regarde dans la pile *sync_ answers* pour voir si la réponse attendue n'a pas été reçue entre temps. Si oui, elle est dépilée et on quitte la boucle de réception. Sinon, on se remet en écoute.

Réponse : On commence par regarder s'il s'agit de la réponse synchrone attendue. Si oui, la requête correspondante est dépilée et on quitte la boucle de réception. Sinon, il peut s'agir d'une autre réponse attendue (synchrone ou asynchrone). Autrement dit, la requête correspondante à cette réponse est trouvée dans une des deux piles *sync_ answers* ou *async_ answers*. Dans ce cas, la réponse est enregistrée avec sa requête dans sa pile et on se remet en écoute. S'il ne s'agit d'aucune réponse attendue, on peut l'ignorer simplement et se remettre en écoute.

Remarques

1. Dans cette fonction, la requête en attente de réponse se trouve toujours au sommet de la pile du fait de ces trois éléments :
   – une pile *sync_ answers* donnée n'est associée qu'à un seul process.
   – un process donné ne peut exécuter qu'une seule instance de `send_msg_sync()` à la fois
   – `send_msg_sync()` est la seule fonction susceptible d'empiler une requête dans *sync_ answers*

   De ce fait, lorsqu'on reçoit une réponse à une requête synchrone attendue, il est facile de déterminer s'il s'agit de la réponse attendue ou d'une autre : la requête est au sommet de la pile ou pas.
2. Si `send_msg_sync()` n'est pas appelée par le process principal, elle ne peut pas recevoir d'autre message que celui qui est attendu.

9. `GET_REP` demande à un nœud de fournir un autre représentant moins chargé pour un étage donné. S'il met trop de temps à répondre et qu'on abandonne, ça n'a pas d'autre conséquence que de conserver le même représentant. On évite ainsi un éventuel *deadlock*
10. On rappelle que cette file est hébergée par le nœud courant.

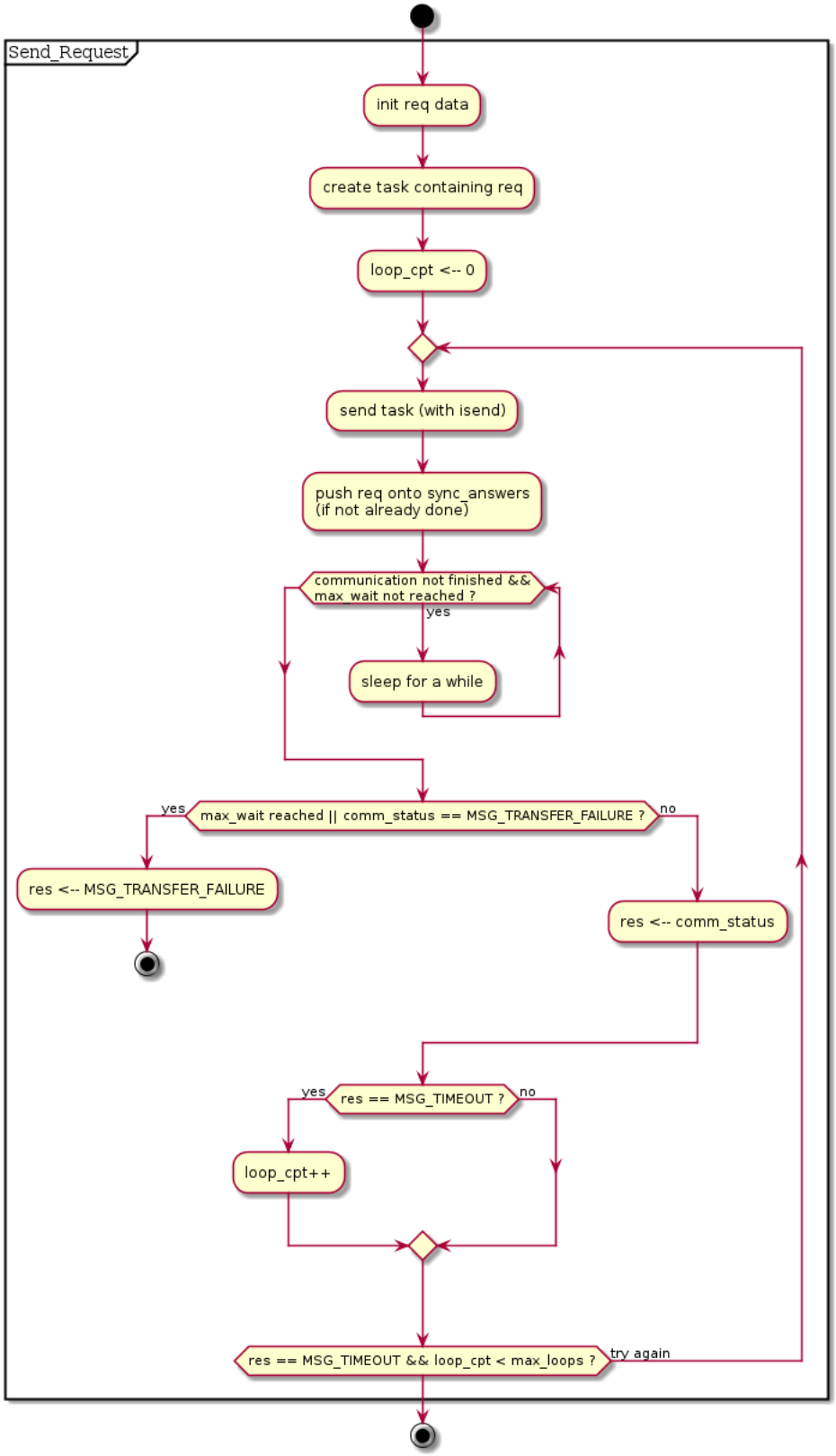
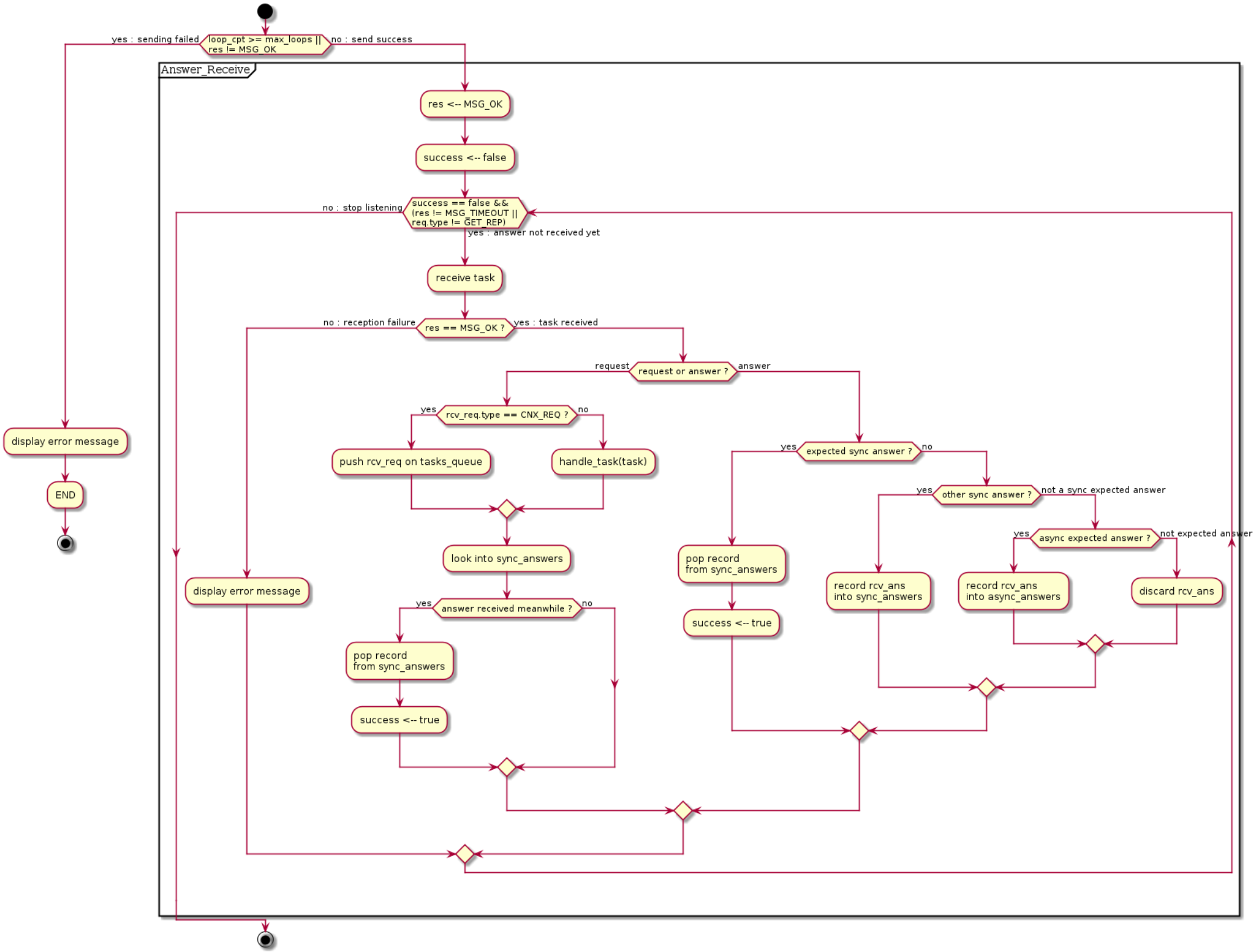


FIGURE 2.6 – Algo `send_msg_sync()` - Partie 1


yes : sending failed
loop_cpt >= max_loops ||
res != MSG_OK
no : send success
Answer_Receive
res <-- MSG_OK
success <-- false
no : stop listening
success == false &&
(res != MSG_TIMEOUT ||
req.type != GET_REP)
yes : answer not received yet
receive task
no : reception failure
res == MSG_OK ?
yes : task received
request
request or answer ?
answer
yes
rcv_req.type == CNX_REQ ?
no
push rcv_req on tasks_queue
handle_task(task)
look into sync_answers
yes
answer received meanwhile ?
no
pop record
from sync_answers
success <-- true
yes
expected sync answer ?
no
pop record
from sync_answers
success <-- true
yes
other sync answer ?
not a sync expected answer
record rcv_ans
into sync_answers
yes
async expected answer ?
not expected answer
record rcv_ans
into async_answers
discard rcv_ans
display error message
display error message
END

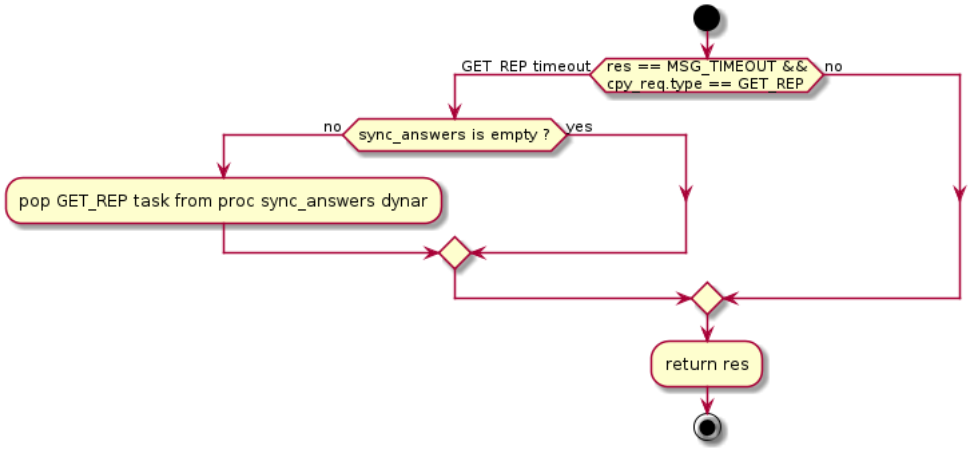


FIGURE 2.8 – Algo `send_msg_sync()` - Partie 3

2.2.2 asynchrone - `send_msg_async()`

Dans ce type de requête, le destinataire renvoie un simple accusé réception à l'émetteur une fois la requête demandée effectuée. Ces accusés réception seront pris en compte ou pas selon les besoins de synchronisation.

Dans les deux cas, les opération se déroulent ainsi : l'émetteur appelle `send_msg_async()` qui exécute les tâches suivantes :

- crée la requête
- encapsule la requête dans une tâche
- envoie la tâche au destinataire (*i.e.* au process principal du destinataire, comme `send_msg_sync()`)

Puis deux cas se présentent :

Un accusé réception est attendu - Cas des requêtes suivantes :

- `BROADCAST` (fonction `broadcast()`)
- `NEW_BROTHER_RCV` (fonction `connection_request()`)
- `DEL_PRED` (fonctions `connect_splitted_groups()`, `split()`)
- `ADD_PRED` (fonction `connect_splitted_groups()`)
- `CNX_GROUPS` (fonction `split()`)

L'émetteur empile alors la requête sur *async_ answers* puis, lorsque plusieurs requêtes ont été émises, il appelle `wait_for_completion()` pour arrêter le déroulement du programme jusqu'à ce que l'ensemble des accusés réception ait été reçu.[11] C'est cette fonction qui dépile les requêtes de *async_ answers* au fur et à mesure de leur réception et comme `send_msg_sync()`, elle traite aussi tout ce qu'elle reçoit entre temps.

Pas d'accusé réception - Dans ce cas, l'appelant ne fait qu'exécuter `send_msg_async()`.

En détails

Voir figure 2.9 - page 20.

11. voir détails de `wait_for_completion()` plus loin

Cette fonction réalise la même chose que la première partie de `send_msg_sync()`, à la différence qu'on n'empile pas la requête puisqu'on laisse le soin à l'émetteur de décider s'il faut le faire ou pas.

Une autre différence est qu'on ignore un éventuel défaut d'émission (`res != MSG_OK`) alors que le programme est arrêté dans `send_msg_sync()`. Ce n'est pas volontaire, je n'ai juste pas encore étudié la tolérance aux pannes.

2.2.3 asynchrone - `wait_for_completion()`

Voir figure 2.10 - page 21.

On attend *ans_ cpt* réponses [12]. Si *max_ wait* est atteint avant que toutes les réponses aient été reçues et traitées, on quitte sur erreur, ce cas ne devant pas se produire.

`wait_for_completion()` pouvant s'appeler elle-même, il faut commencer par vérifier si des réponses attendues ont déjà été traitées par ces appels récursifs, et s'il en reste encore à traiter. C'est l'appel à `check_async_nok()` du début.

Si oui, on se met en écoute, et comme dans `send_msg_sync()`, il faut traiter tout ce qui est reçu dans l'intervalle :

Requête : Si c'est `CNX_REQ`, on l'empile sur *tasks_ queue*, sinon, on l'exécute avec `launch_fork_process()`. On regarde ensuite si des réponses ont été reçues entre temps avec `check_async_nok()`, puis on se remet éventuellement en écoute.

Réponse :

- Réponse asynchrone correspondant à une requête empilée :
  - si elle fait partie de celles qui sont attendues, la requête correspondante est dépilée d'*async_ answers*, après avoir mémorisé un éventuel échec de `BROADCAST` ou `SET_UPDATE` dans *ret* (pour du log). [13] On se remet ensuite en écoute s'il reste des réponses à recevoir. (`ans_cpt > 0`)
  - sinon, elle est enregistrée dans *async_ answers*
- Réponse synchrone attendue :
  - si oui, elle est enregistrée dans *sync_ answers*
  - sinon, elle est ignorée.

12. Sur le process courant qui peut être l'un de ceux qui est présenté sur la figure 2.1 page 5
13. On ne mémorise ici que le premier échec

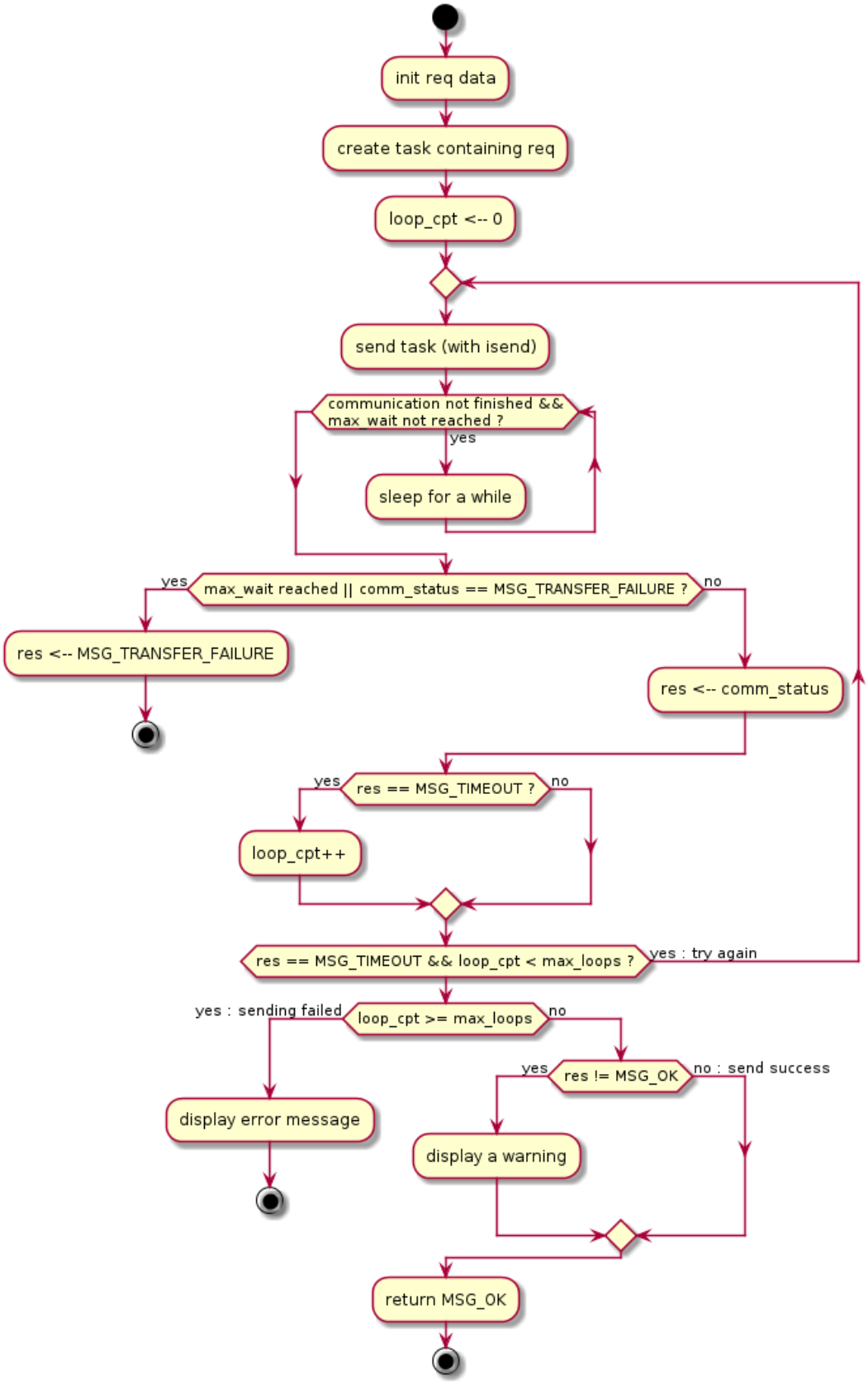


FIGURE 2.9 – Algo `send_msg_async()`

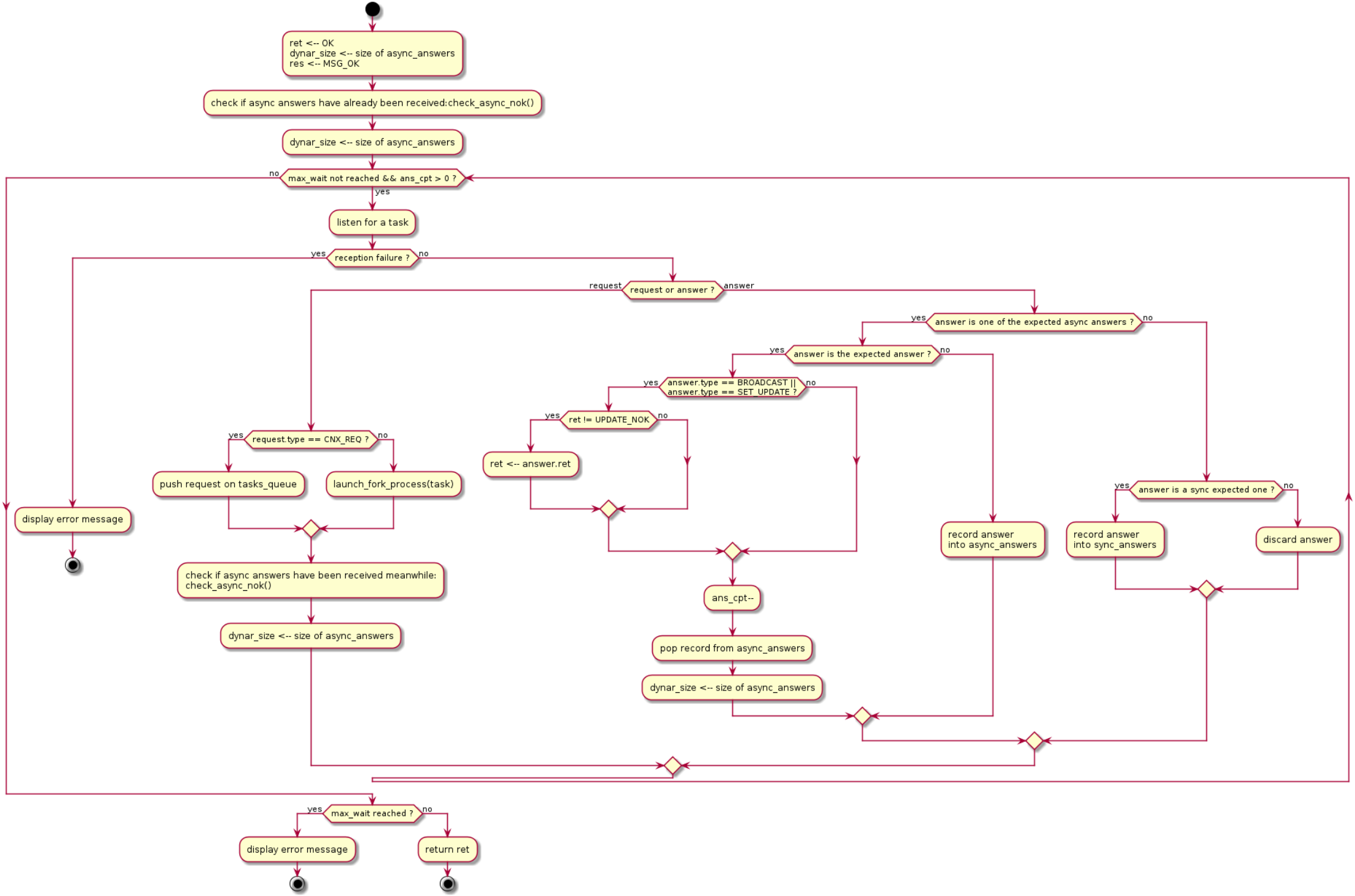


FIGURE 2.10 – Algo `wait_for_completion()`

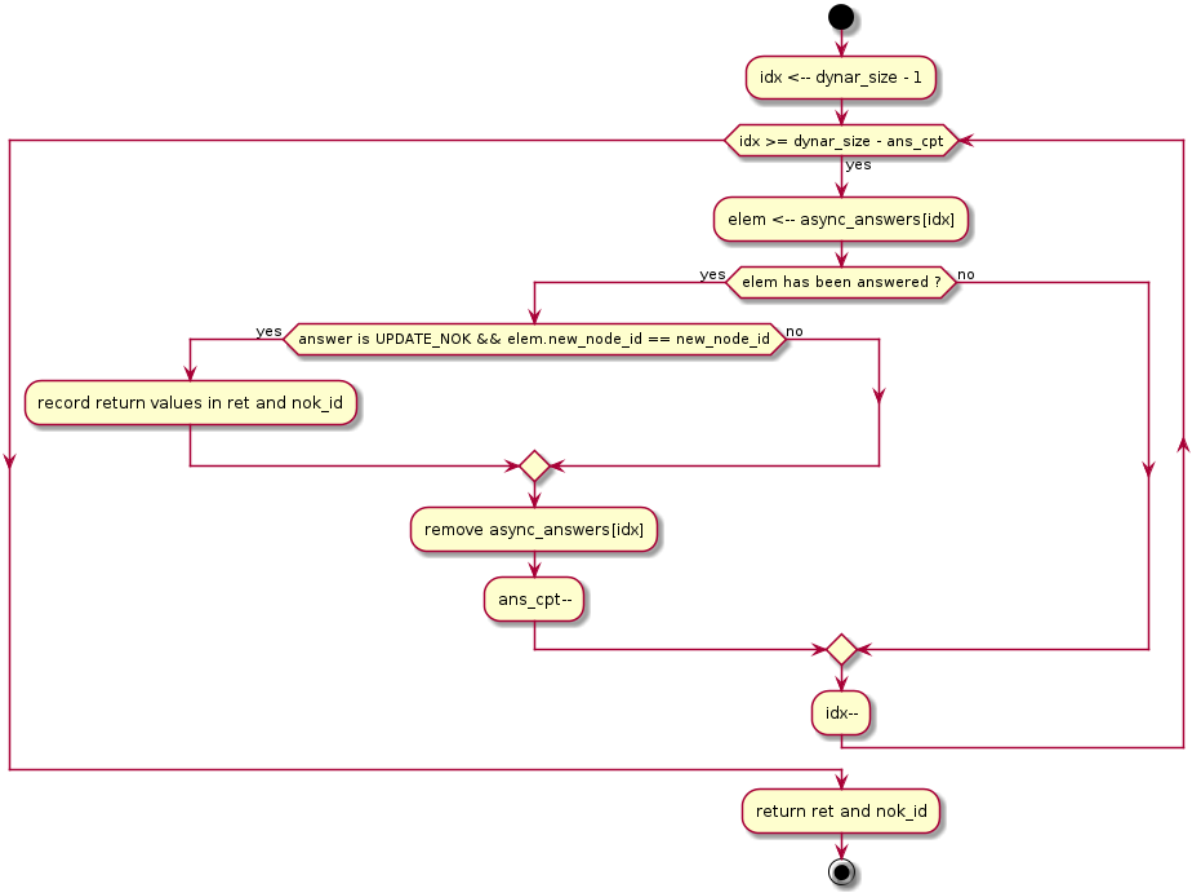


FIGURE 2.11 – Algo `check_async_nok()`

2.2.4 asynchrone - `check_async_nok()`

Voir figure 2.11 - page 22

Cette fonction auxiliaire est appelée par `wait_for_completion()`. Elle est chargée de parcourir la file *async_ answers* à la recherche de réponses reçues. Le cas échéant, la requête correspondante est retirée de la pile, le compteur de réponses attendues *ans_ cpt* est ajusté et on retourne également la dernière erreur reçue.

Chapitre 3

Échec d'une diffusion d'un verrou : solution

Rappel du problème

Lorsqu'il est nécessaire de faire de la place pour un nouveau nœud, on diffuse une requête sur l'ensemble du sous-arbre impacté pour y poser un verrou. Cette diffusion peut échouer pour différentes raisons (si elle rencontre une autre diffusion semblable pour un autre nouveau nœud, par exemple). Dans ce cas, cette diffusion doit non seulement s'arrêter, mais il faut aussi ôter les verrous qu'elle a déjà posés sous peine de blocage mutuel des diffusions en conflit.

La solution proposée dans la version précédente du simulateur consistait à relancer une nouvelle diffusion destinée cette fois à ôter les verrous posés. Dans la pratique, on constate que cette façon de faire n'est pas la bonne : elle génère un grand nombre de diffusions qui s'entrecroisent et peuvent causer des blocages, sans parler du nombre de messages que cela génère. Il faut donc trouver une autre solution.

Idée de solution

En premier lieu, pour diminuer le nombre de messages échangés lors de ces diffusions, elles ne vont plus couvrir que les leaders. En effet, il suffit d'interroger un leader pour savoir si la voie est libre ou pas.

Ensuite, l'idée générale de cette nouvelle proposition est de procéder en deux temps :

1. diffuser une première requête que j'appelle `CS_REQ`[1]
   Cette diffusion ne pose pas de verrou mais positionne simplement des variables qui sont capables de revenir à leur état initial en cas d'échec, sans nouvelle diffusion.
2. si cette diffusion s'est bien passée, diffuser la pose de verrou proprement dit – requête `SET_UPDATE`

1. Demande d'entrée en section critique

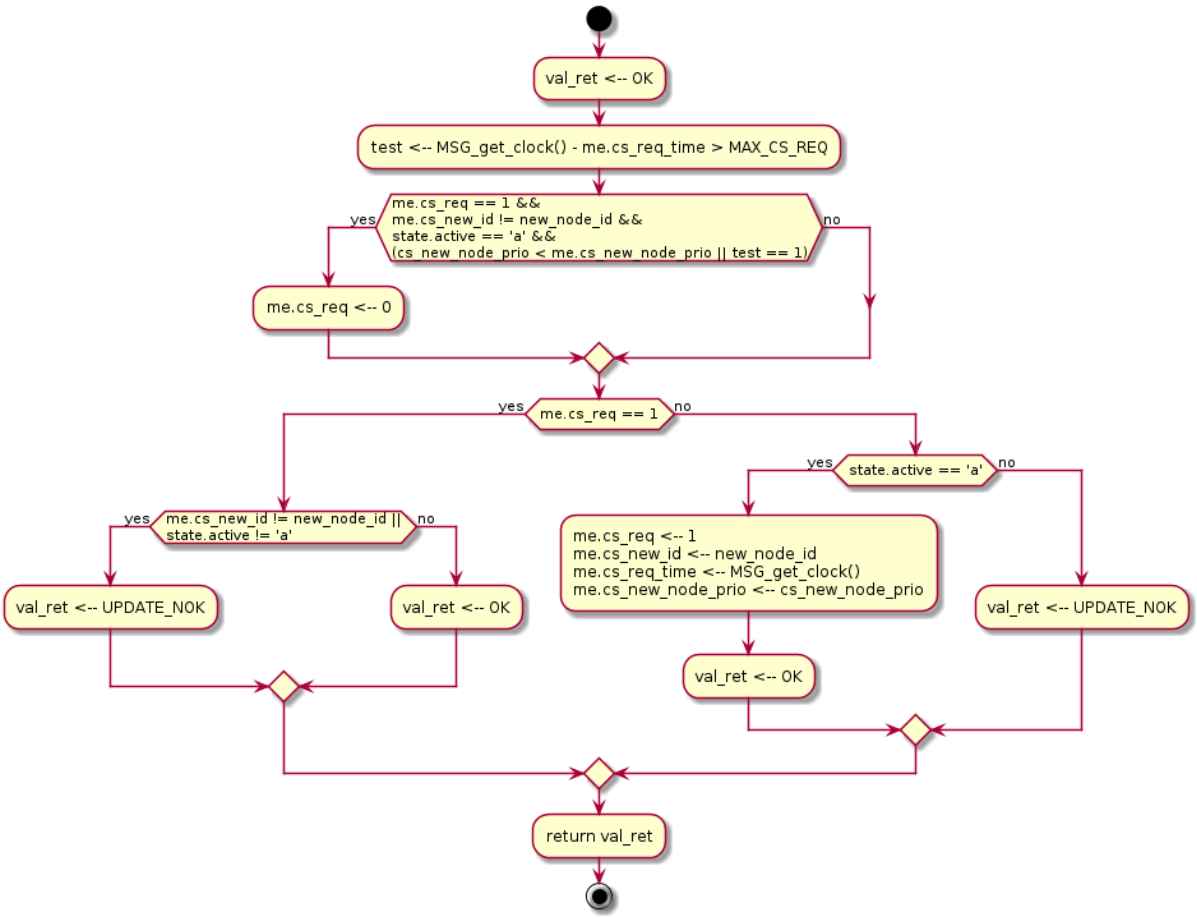


FIGURE 3.1 – Algo `cs_req()`

3.1 Solution détaillée

3.1.1 Fonction `cs_req()`

Chaque nœud possède un ensemble de variables d'état pour ce mécanisme :

cs_req **:** Flag qui indique si une demande a été reçue ou pas

cs_req_time **:** Heure à laquelle la demande a été reçue

cs_new_id **:** Id du nouveau nœud pour lequel la demande a été faite

cs_new_node_prio **:** Niveau de priorité du nouveau nœud

Cette fonction prend les arguments suivants :

new_node_id **:** C'est l'id du nouveau nœud en cours d'insertion

cs_new_node_prio **:** La priorité de ce nouveau nœud

Et elle est décrite à la figure 3.1 page 24.

test est un flag qui passe à 1 s'il s'est écoulé plus de `MAX_CS_REQ` unités de temps depuis la dernière fois qu'une demande a été vue.

- Plaçons nous dans le cas où aucune demande n'a été reçue : `me.cs_req` vaut alors `0`. Les deux premiers tests valent `non` et si on est à l'état `'a'`, on va positionner les variables de nœud indiquant qu'une demande a été reçue [2] :
  - `me.cs_req` passe à 1
  - `me.cs_new_id` reçoit l'argument `new_node_id` pour mémoriser le nouveau nœud qui a causé cette demande
  - `me.cs_req_time` reçoit l'heure courante
  - `me.cs_new_node_prio` mémorise la priorité du nouveau nœud

  On retourne alors `OK` pour indiquer que tout s'est bien passé, la demande a été acceptée, on est prêt à recevoir le verrou correspondant.

- Si une autre demande est reçue avant ce verrou, voici ce qu'il se passe :
  `me.cs_req` vaut alors `1` et on commence par regarder si ce flag doit être remis à `0`, c'est à dire si :
  - cette demande en cours concerne un autre nouveau nœud
  - on est toujours à l'état `'a'` [3]
  - soit le nouveau nœud courant est prioritaire sur celui qui a fait cette demande, soit elle est trop ancienne.

  Dans ce cas, le flag est à nouveau positionné, mais pour ce nouveau nœud courant. La diffusion du verrou pour le nouveau nœud précédent échouera donc. [4]

  Sinon, si le flag est maintenu à `1`, on se contente de lire les variables associées pour retourner la bonne réponse : on répond `UPDATE_NOK` si la demande en cours concerne un autre nouveau nœud ou si on n'est plus à l'état `'a'`. Sinon, on répond `OK` pour dire que tout va bien.

3.1.2 Diffusion du verrou `SET_UPDATE`

Tout d'abord, pour que la diffusion de `SET_UPDATE` ait lieu, la diffusion de `CS_REQ` doit avoir retourné `OK`. Ce qui signifie que l'ensemble des leaders du sous-arbre concerné doit avoir ses variables `cs_req` correctement positionnées pour qu'un `SET_UPDATE` soit accepté. [5]

Si la diffusion de `CS_REQ` échoue, aucun verrou n'est posé et comme on l'a vu plus haut, le fait que les flags aient été positionnés pour une diffusion qui n'aura finalement pas lieu n'est pas bloquant.

Mais on a également vu qu'entre la diffusion de `CS_REQ` et celle du `SET_UPDATE` correspondant, des flags pouvaient avoir changé, ce qui provoquera l'échec de la diffusion de ce `SET_UPDATE`. Dans ce cas, il faut tout de même refaire une diffusion d'une tâche `REMOVE_STATE` chargée de remettre les leaders en question dans l'état où ils étaient avant la pose du verrou.

Cette diffusion ressemble à celle de la première solution dont on avait dit qu'elle était à éviter. Mais dans les faits, il me semble que la diffusion de `CS_REQ` échoue bien plus souvent que celle

2. Autrement dit, on se met en situation d'accepter la pose de verrou qui suivra, qui doit arriver avant que `MAX_CS_REQ` soit écoulé
3. Si un verrou avait été posé entre temps, on serait à l'état `'u'`.
4. Voir détails plus loin
5. Si le `SET_UPDATE` reçu n'est pas celui qui était attendu, on exécute la fonction `cs_req()` localement pour éventuellement positionner différemment les variables `cs_req`. Il s'agit de vérifier si on peut l'accepter ou pas.

de SET_UPDATE. On peut donc penser que si ce nouveau mécanisme de CS_REQ n'était pas utilisé, on aurait beaucoup plus de diffusions de REMOVE_STATE. Il faut encore que je quantifie ces cas pour vérifier mon intuition. Si elle est juste, alors cette nouvelle solution permet effectivement d'économiser des diffusions, et donc, de la bande passante.

Chapitre 4

Comparaison avec la première solution

Avec ces nouveaux algorithmes, j'ai pu construire des DST jusqu'à 40 000 nœuds (je n'ai pas pu aller plus loin à cause de la mémoire nécessaire) et je n'ai pas constaté de *deadlocks*.

Contrairement à ma première solution, le simulateur réussi aussi ces constructions même si je change les paramètres du réseau (augmentation de la latence, par exemple, pour dégrader les performances).

Les performances du simulateurs même sont moins bonnes : durée d'exécution plus longue et mémoire nécessaire plus importante. Cela me paraît normal dans la mesure où les nœuds doivent maintenant héberger plusieurs process en parallèle. De plus, dans l'ancien simulateur, il était très fréquent (bien que j'aie cru le contraire) que lors de l'arrivée d'un nouveau nœud, les nœuds précédents avaient déjà terminé leur intégration. On était alors souvent dans un cas quasi-séquentiel ou les nœuds arrivent les uns derrière les autres. Ici, on simule bien mieux le cas de nœuds arrivant simultanément.

Reste à faire

- Comparer le nombre de fois où on doit diffuser un `REMOVE_STATE` avec le nombre d'échecs de diffusions de `CS_REQ` pour vérifier effectivement si cette nouvelle solution est meilleure que l'ancienne.
- voir s'il ne serait pas possible de diminuer la mémoire requise
- inclure les algorithmes de départ de nœuds dans ces mécanismes