



UNIVERSITÉ DE FRANCHE-COMTÉ

Construction d'un DST : autres propositions

par Christophe ENDERLIN

le 21 septembre 2014

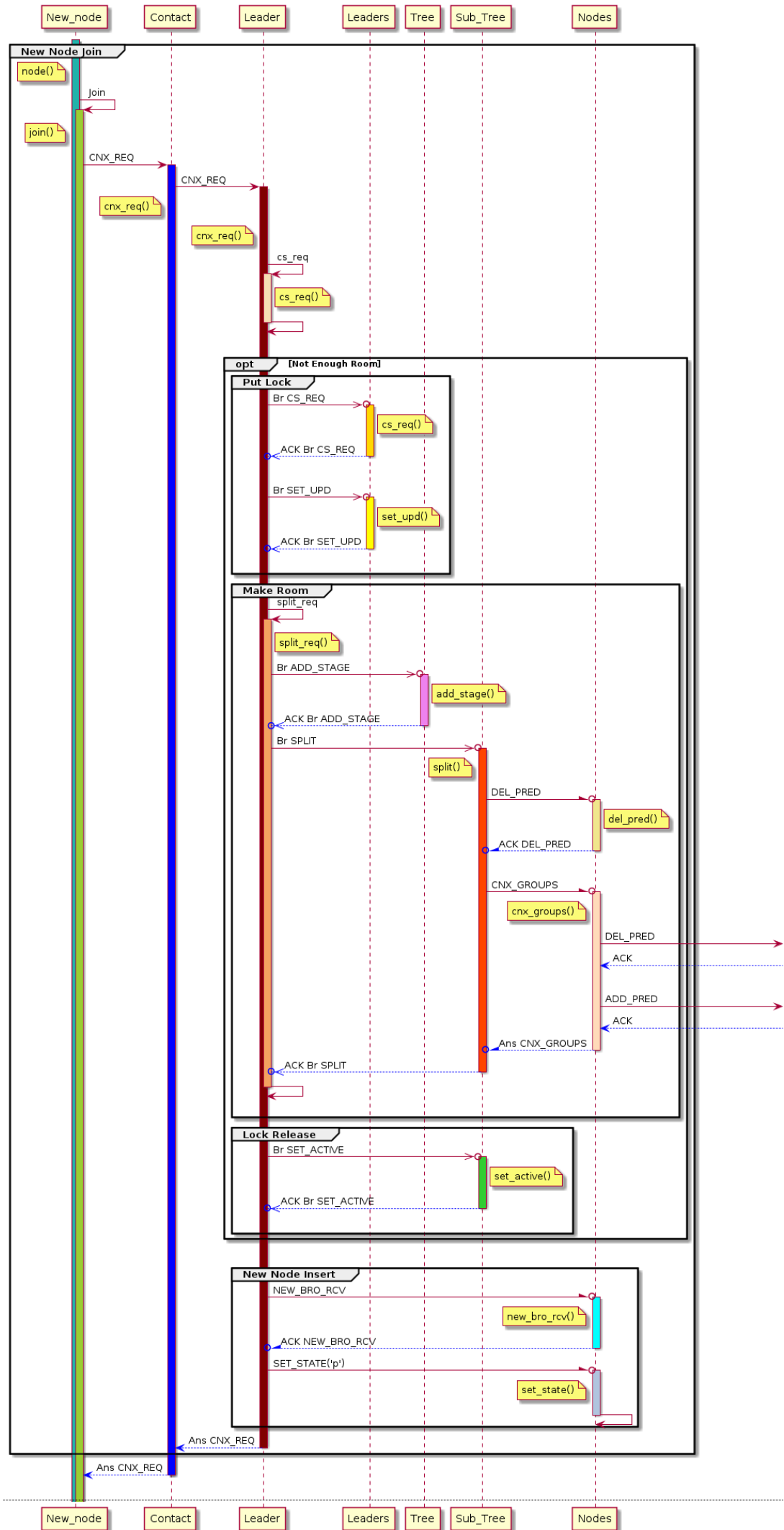
Chapitre 1

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



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 \implies *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 **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*.

Process calls sequence

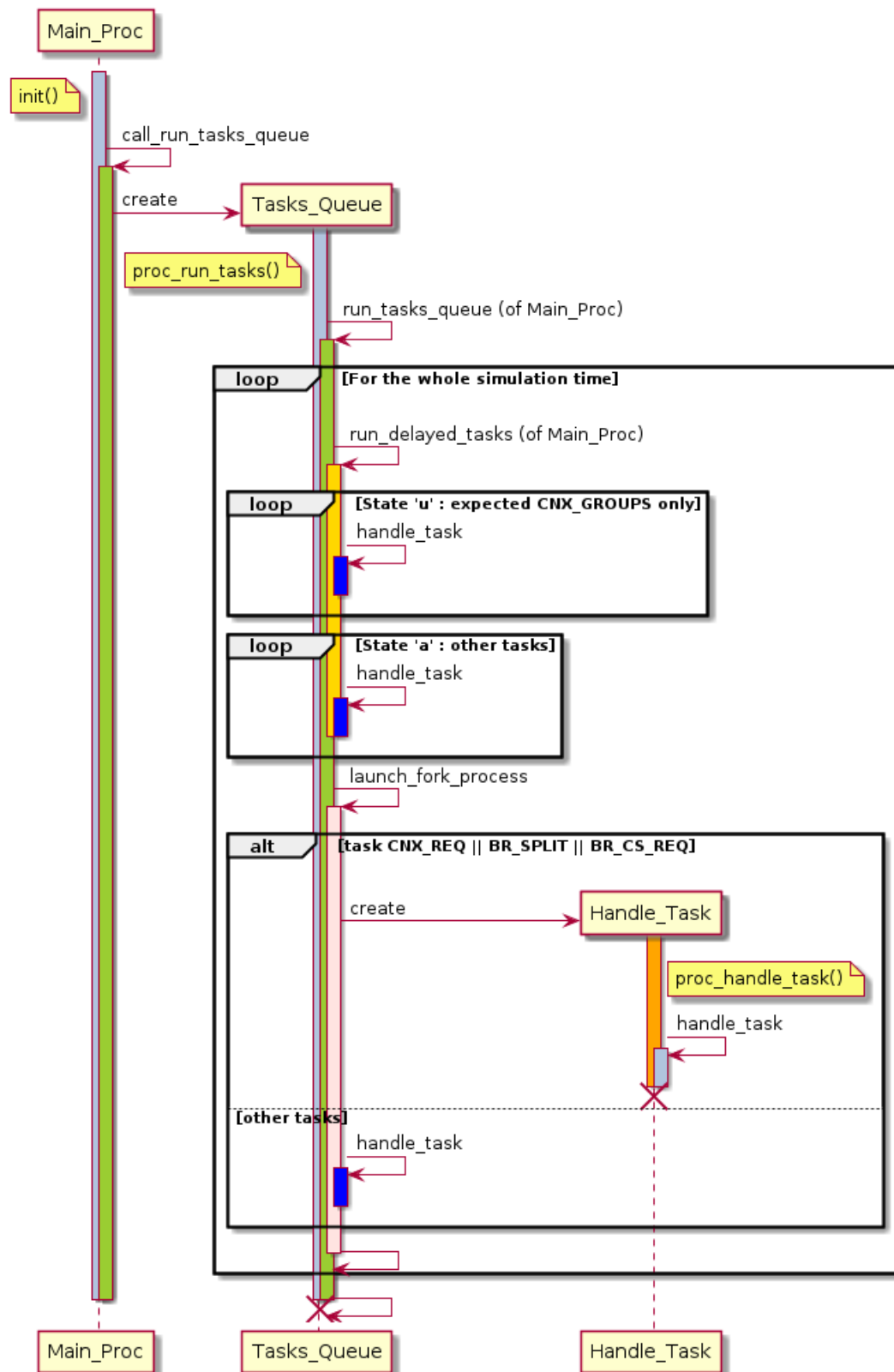


FIGURE 1.1 – Séquence d'appels des sous-process

- 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()` 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()`

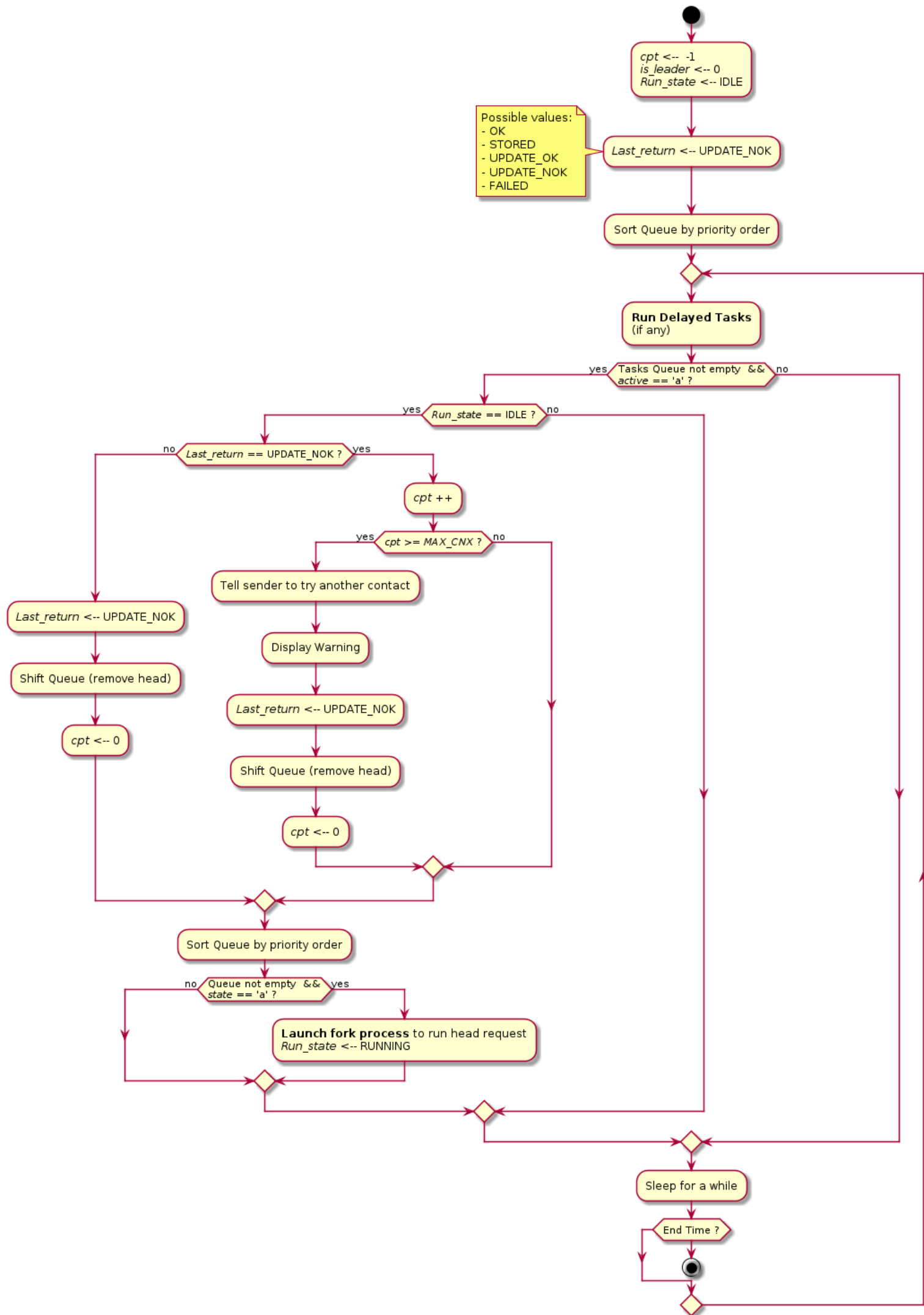
La file *tasks_queue* contient toutes les demandes de connection (c'est à dire toutes les tâches `CNX_REQ`) reçues par le nœud qui héberge cette file. `run_tasks_queue()` est donc chargée de traiter les requêtes présentes dans cette file, par ordre de priorité.

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* : 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. 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.)



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* ; il dépile 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. Voir commentaires en conclusion)

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 *CNX_REQ*, 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)