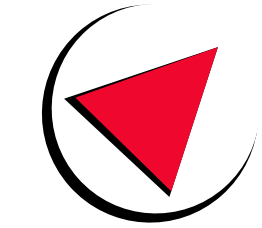


L I F C

Université de Franche-Comté

# Contruction d'un DST : autres propositions

*par* Christophe Enderlin

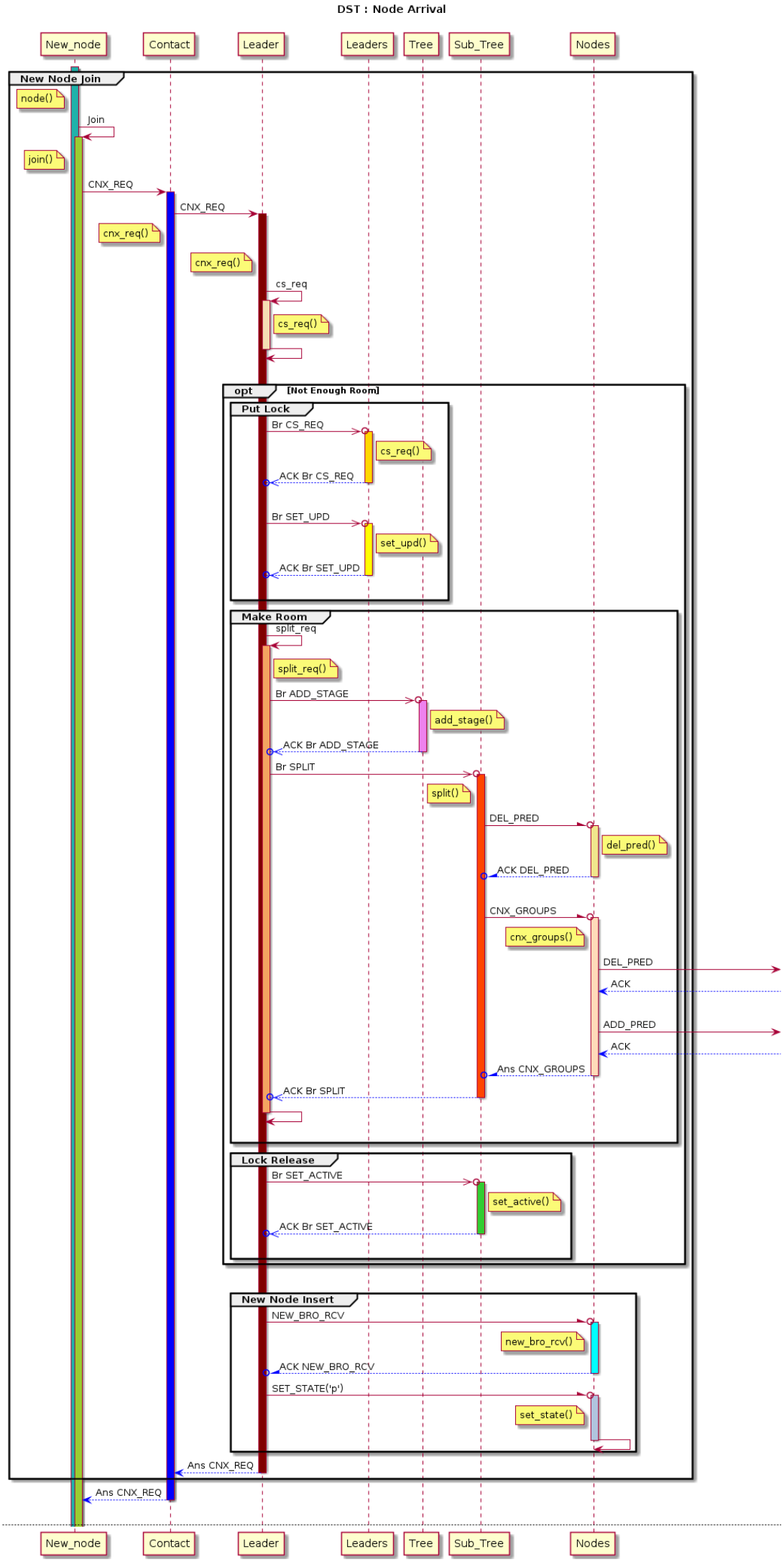
*le* 20 septembre 2014

# Chapitre 1

# CONTEXTE

## 1.1 RAPPEL

Pour mémoire, voici une représentation graphique du déroulement des opérations lors de l'insertion d'un nouveau noeud 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 noeuds 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 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 : c'est un DEADLOCK

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

Un seul process (hébergé sur Node 121) chargé de l'intégration de deux nouveaux noeuds 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 : 1) Utiliser la fonction MSG_comm_waitany() de Simgrid (elle permet de réagir à n'importe quelle réponse parmi celles qui sont attendues) 2) 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 noeud 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 des ces noeuds. Ainsi, ils ne peuvent plus s'occuper que des requêtes concernant 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 exemple), on se retrouve dans la situation où une partie du sous-arbre a été verrouillée pour le nouveau noeud courant, et une autre pour un autre nouveau noeud. 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 puiqu'on arrive malgré tout à générer des deadlocks. (ces deux sous-arbres se bloquent mutuellement)

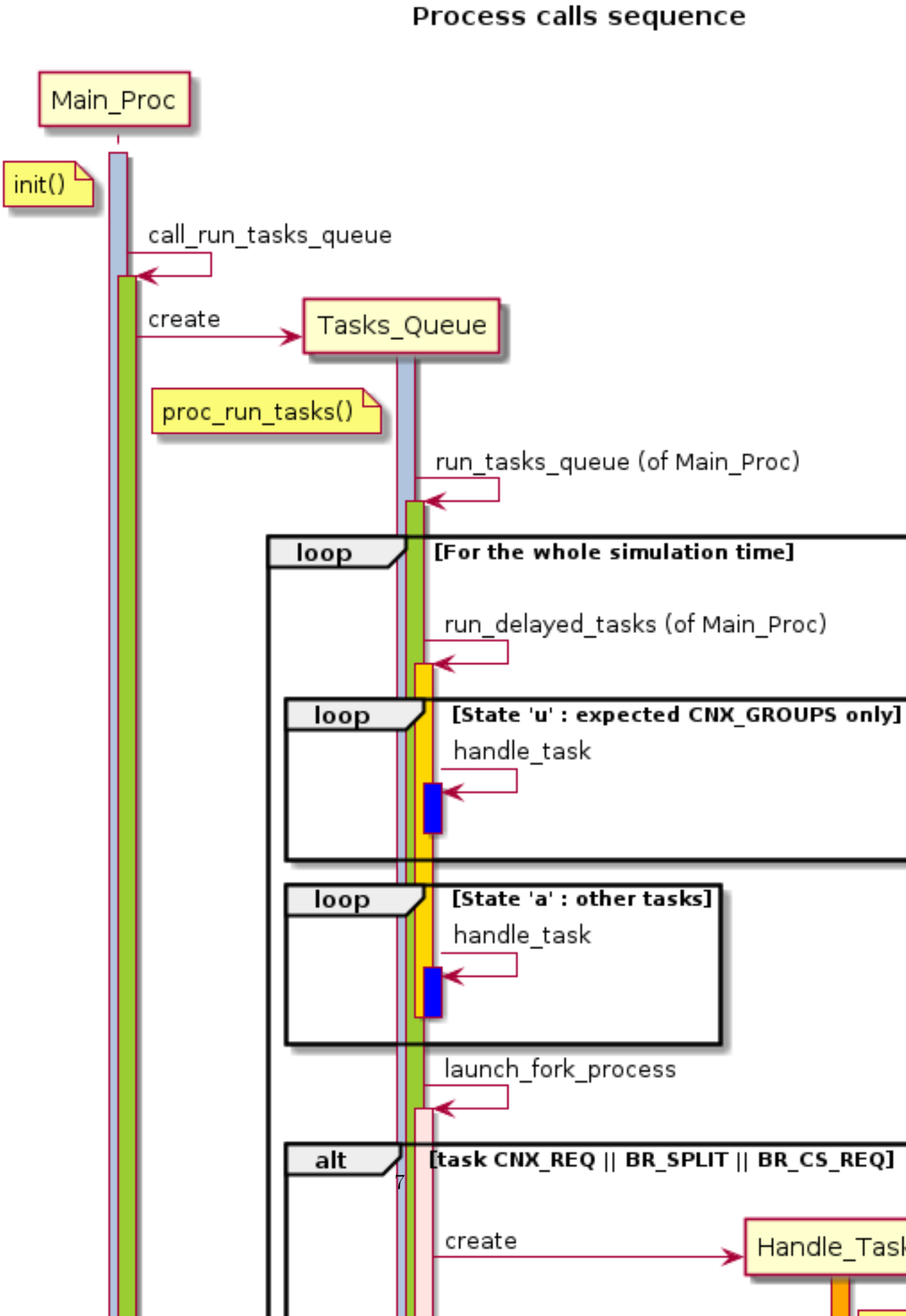
Il faut donc trouver des solutions pour ces deux problèmes.

## 1.3 Proposition de solution

Voici les principes de base de cette solution : - Chaque demande d'insertion de nouveau noeud 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 noeuds 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.

Voici donc le schéma général de cette solution à process multiples :


Process calls sequence
Main_Proc
init()
call_run_tasks_queue
create
Tasks_Queue
proc_run_tasks()
run_tasks_queue (of Main_Proc)
loop
[For the whole simulation time]
run_delayed_tasks (of Main_Proc)
loop
[State 'u' : expected CNX_GROUPS only]
handle_task
loop
[State 'a' : other tasks]
handle_task
launch_fork_process
alt
[task CNX_REQ || BR_SPLIT || BR_CS_REQ]
create
Handle_Task

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 noeud : * 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, soit il la place dans la file tasks_queue (cas des CNX_REQ), soit il la transmet à launch_fork_process(). Il peut aussi différer des requêtes en les plaçant 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)

* 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.

* 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.