LA MACHINE (INFRASTRUCTURE)

L'implémentation d'NJN est constituée en deux parties : l'infrastructure et la superstructure.

L'infrastructure assure l'ordonnancement général de l'application et fournit à la superstructure les primitives qui lui permettent de fonctionner dans un cadre causal. Trois actions principales sont à la charge de l'infrastructure : la gestion des évènements du système, la gestion des messages entre processus logiques et le nettoyage de la mémoire.

Pour gérer les communications et la mémoire, l'infrastructure utilise les bibliothèques support abordées au chapitre précédent. La gestion des évènements repose sur l'implémentation du graphe causal vue dans la première partie (voir page 66).

Variables et tâches sont des objets de la superstructure. Au niveau infrastructure, NJN ne voit des variables que les signaux qui en mémorisent l'état et il ne voit des tâches que les gestionnaires de tâches qui prennent en charge leur fonctionnement.

Ce chapitre s'intéresse à l'infrastructure et présente d'abord le graphe causal et la gestion des évènements, ensuite la gestion des messages de service, enfin l'ordonnancement général et la gestion de la mémoire.

1 Les signaux, les évènements et les gestionnaires de tâches

Le graphe causal est à la base du comportement causal d'NJN. Il trace les relations de cause à effet entre toutes les opérations effectuées dans l'application.

1.1 Les évènements et la structure du graphe causal

La figure 62 fournit le diagramme de classes simplifié du graphe causal.

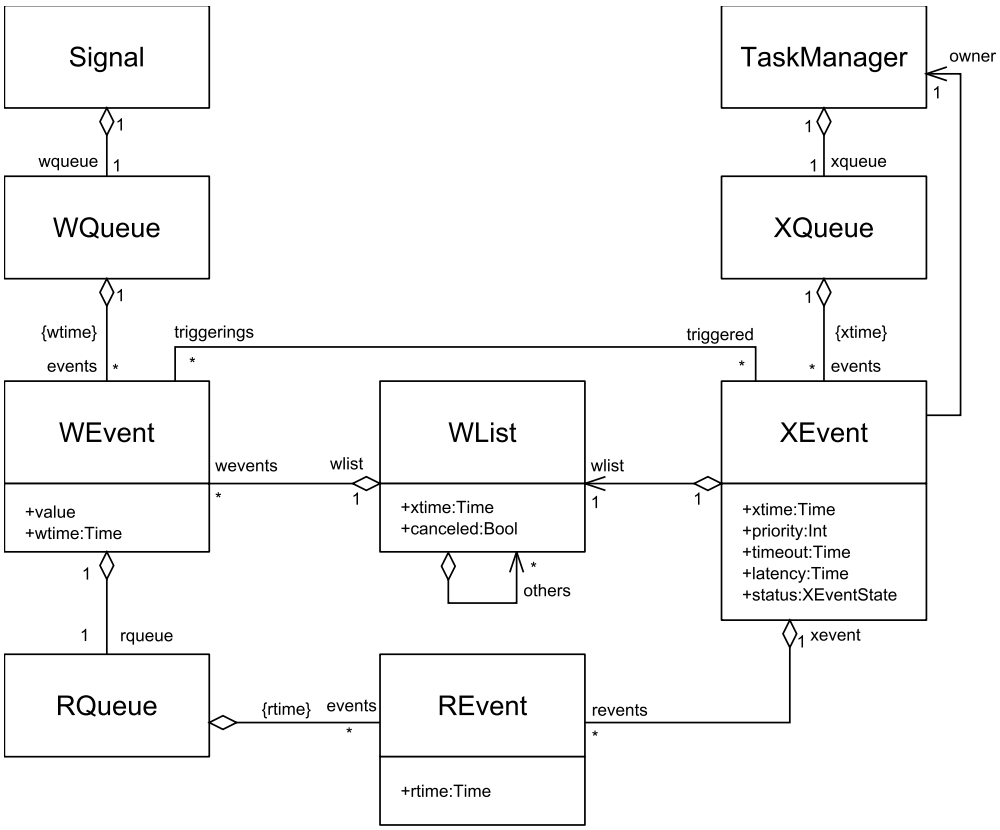

Figure 62. Diagramme de classes du graphe causal.

Le graphe est constitué de trois principaux objets :

- des évènements d'exécution (*XEvent*) qui matérialisent l'exécution d'une tâche (champ *owner*) à une date donnée (*xtime*) ;
- des évènements d'écriture (*WEvent*) qui représentent l'écriture de donnée (champ *value*) à une date (*wtime*) sur un signal ;
- des évènements de lecture (*REvent*) qui représentent la lecture de l'état d'un signal à un instant donné (*revent*).

Des objets intermédiaires implémentent les relations entre ces trois objets principaux :

- les objets *WList* permettent de relier l'exécution d'une tâche (un objet *XEvent*) à toutes les opérations d'écriture (objets *WEvent*) que cette exécution a provoquées ;
- les objets *RQueue* classent selon l'ordre temporel toutes les lectures d'une même donnée (objet *WEvent*) effectuées lors de divers exécutions de tâches (objets *Xevent*) ;
- les objets *WQueue* permettent de classer selon l'ordre temporel les évènements d'écriture se rapportant à un même signal ;
- les objets *XQueue* permettent de classer selon l'ordre temporel les évènements d'exécution se rapportant à une même tâche.

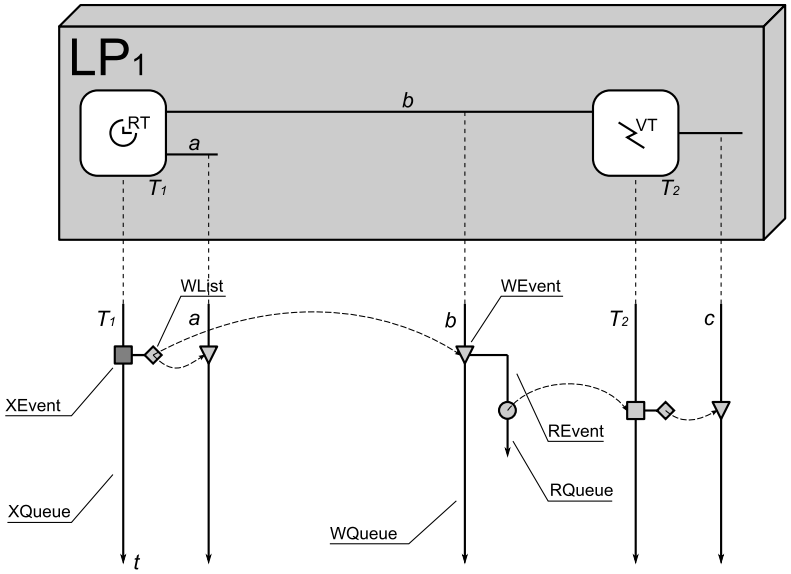

Figure 63. Le graphe causal.

Lorsqu'une tâche écrit une donnée sur une variable, un objet *WEvent* est créé et ajouté au signal associé à cette variable. La relation causale de l'opération est matérialisée par l'objet *WList* dans lequel est également placé ce nouvel évènement.

Lorsqu'il s'agit d'écriture sur des signaux distants, un objet *WList* est construit du côté de la cible et est référencé par le champ *others* de la liste localement associée à l'évènement d'exécution. Ce lien distant est implémenté par un objet DOHC de type *Handle*.

Lorsqu'une tâche lit l'état d'une variable, NJN cherche sur le signal correspondant l'objet *WEvent* le plus récent antérieur à la date de lecture. Un objet *REvent* est créé pour matérialiser l'opération et est ajouté à la liste *rqueue* du *WEvent* concerné.

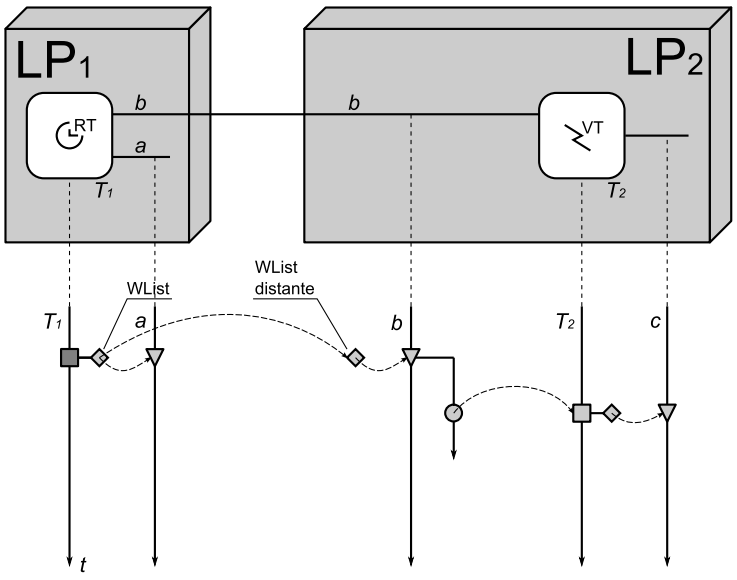

Figure 64. Lien entre XEvent et WEvent distants.

Le déclenchement d'une tâche est provoqué par l'apparition d'un évènement associé à une des variables placées dans la liste de sensibilité de la tâche. Le graphe causal crée, dès le déclenchement l'objet *XEvent* qui servira de support à l'exécution de la tâche. La relation de cause à effet entre l'écriture sur la variable et l'exécution est matérialisée par une relation bidirectionnelle entre les objets *WEvent* et *XEvent* impliqués. L'objet *XEvent* voit les écritures qui l'ont déclenché (champ *triggerings*) tandis que l'objet *WEvent* voit les exécutions qu'il a provoquées (*triggered*).

1.2 Le gestionnaire de tâche, les évènements d'exécution et leur état

Chaque tâche est associée à un gestionnaire spécifique qui assure son bon fonctionnement et gère ses exécutions. Concrètement, le gestionnaire de tâche gère l'état des évènements d'exécution associés à la tâche dont il a la charge.

Un évènement d'exécution peut être placé dans six états distincts : déclenché (*scheduled*), exécuté *(executed),* abandonné *(aborted),* invalidé (*invalidate*), annulé *(cancelled)* ou détruit (*destroyed*) :

- L'état *scheduled* est l'état de l'évènement à sa création. Il correspond au déclenchement de la tâche. L'évènement se trouve alors en attente d'exécution.
- L'état *executed* témoigne de l'exécution effective de la tâche correspondante. Sauf si un retour arrière rend l'exécution invalide, l'évènement restera dans cet état jusqu'à ce qu'il soit supprimé de la mémoire.

- L'état *aborted* correspond à une exécution déclenchée par *timeout* pour laquelle la condition temporelle n'est plus remplie.
- L'état *invalidate* correspond à un évènement rendu invalide par une violation de causalité. Si l'évènement a déjà été exécuté, il devra être déclenché à nouveau.
- L'état *cancelled* correspond à un évènement qui, à la suite d'un retour arrière ou une violation de causalité, n'a plus d'évènement d'écriture déclencheur. Il n'a donc plus de raison d'exister.
- Enfin l'état *destroyed* correspond à un évènement supprimé du graphe causal. Il sera nettoyé par le ramasse-miette de DOHC.

Au moment du déclenchement d'une tâche, le gestionnaire de tâche place l'objet *XEvent* nouvellement créé dans une liste globale d'évènements d'exécution à traiter[1]. L'objet y reste tant qu'il est dans l'état *scheduled.* Une fois traité – c'est-à-dire une fois que la tâche aura été exécutée – l'évènement sera retiré de la liste globale mais sera conservé dans le graphe causal.

Le déclenchement par *timeout* est géré par un signal spécifique : la propriété *beat* (figure 65). A chaque exécution, un évènement est inscrit sur ce signal. Toutes les tâches déclenchées par *timeout* sont sensibles à leur propriété *beat,* ce qui a pour conséquence que chaque exécution provoque systématiquement un nouveau déclenchement. L'exécution suite à un *timeout* est conditionnée par le respect du délai correspondant depuis la précédente exécution. Si le délai est respecté, la tâche est exécutée et le *XEvent* associé passe dans l'état *executed.* Dans le cas contraire, l'évènement d'exécution passe dans l'état *aborted*.

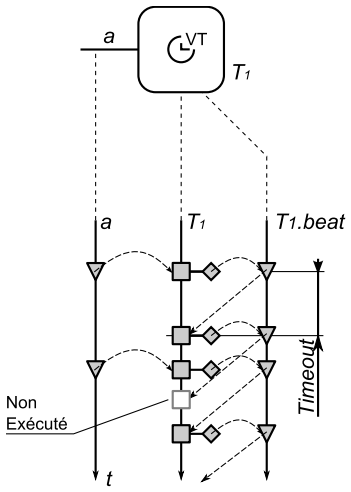

Figure 65. Implémentation du déclenchement par Timeout.

1 Selon que l'évènement correspond à une tâche temps-réel ou une tâche temps-virtuel, la liste globale n'est pas la même.

A tout moment, un évènement d'exécution peut être remis en cause par un retour arrière, qu'il soit exécuté ou simplement déclenché.

1.3 La mise en œuvre du retour arrière

Un retour arrière a toujours pour origine une violation de causalité. Tout commence donc par l'identification de cette violation.

Au moment de la création d'un objet *WEvent*, NJN consulte la liste de lecture (*RQueue*) du *WEvent* situé dans le même objet *WQueue*, à la position immédiatement antérieure à la nouvelle écriture[1]. Si, dans cette liste de lecture, NJN trouve des lectures postérieures au nouvel évènement, la violation de causalité est manifeste.

Les évènements de lecture identifiés comme invalides sont alors le point de départ du retour arrière. A partir de chaque lecture identifiée comme invalide, NJN retrouve les objets *XEvent* qui ont exploité des données potentiellement incorrectes. A partir de ces évènements d'exécution, NJN invalide l'ensemble des opérations d'écriture (objets *WEvent*) accessibles dans la *WList* de chaque *XEvent* ainsi que tous les évènements correspondant aux lectures effectuées par erreur.

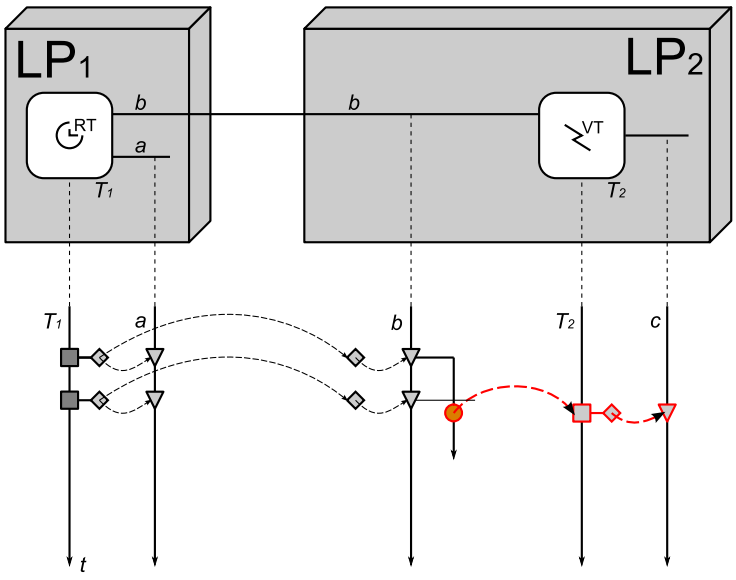

Figure 66. Identification d'une violation de causalité.

A partir des évènements d'écriture invalides, on identifie les *XEvent* dont le déclenchement est sans objet.

Par l'intermédiaire de leur *RQueue*, les écritures invalides permettent également de retrouver les lectures à remettre en cause.

Les *WEvent* correspondants deviennent les points de démarrage d'une nouvelle phase de nettoyage du graphe. L'algorithme de retour arrière est relancé à partir de

1 N'hésitez pas à relire cette phrase !

ces nouvelles lectures invalides, jusqu'à ce que l'ensemble du graphe ait retrouvé un état satisfaisant.

2 Les messages, les paquets et le gestionnaire de services

2.1 Un simple appel de méthode...

Dans NJN, un appel de méthode effectué par un objet sur un autre initie bien souvent une succession d'appels croisés entre les deux objets impliqués, voir entre eux et les objets qu'ils contiennent. La figure 67 présente par exemple la séquence de création d'un composant et l'assignation d'un de ses ports d'entrée à une variable source. Les opérations s'enchainent de la façon suivante :

- Une fabrique crée le composant et fournit à son constructeur le plan d'assignation des éléments du port d'entrées/sorties.
- Le composant construit les objets qui le composent et en particulier une variable d'entrée nommée *target*.
- Il demande ensuite l'assignation de cette entrée à une variable *source*.
- L'assignation provoque la création d'un connecteur du côté de la source.
- La variable *source* est ensuite associée à la variable *target*.

Les opérations d'assignation s'enchainent les unes à la suite des autres tant qu'il reste des ports à connecter.

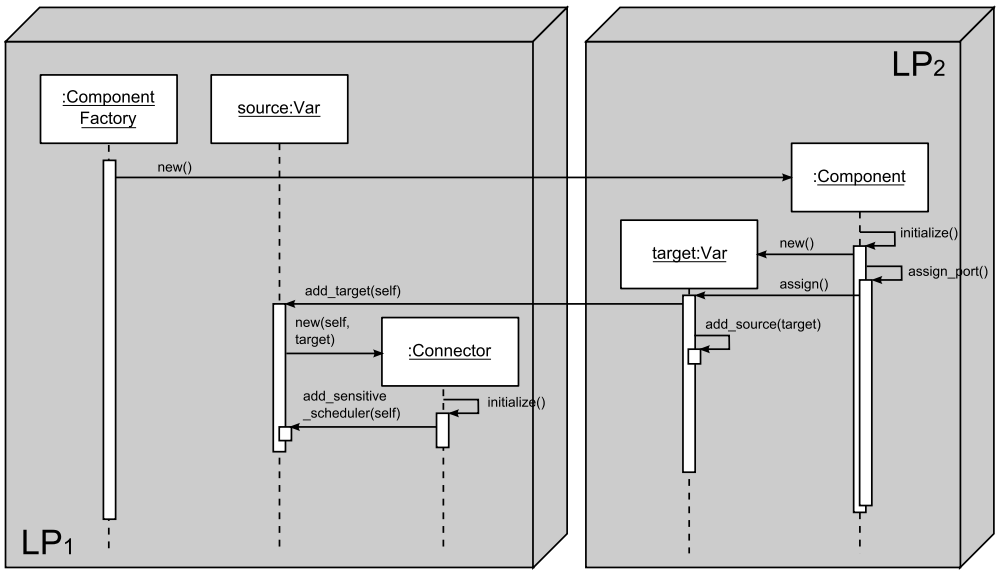

Figure 67. Création d'un composant et assignation d'un port d'entrée à une variable source.

Lorsque la création d'un composant est réalisée à distance, la situation est la suivante : la fabrique et la variable *source* sont situées dans un processus logique *(LP_1)* ; le composant créé et sa variable d'entrée *target* sont dans un autre processus logique *(LP_2)*.

Pour réaliser l'opération, NJN utilise un service *CABLES*. Rappelons qu'un appel de service *CABLES* est une opération en trois temps :

- Le processus appelant lance la requête ;
- Le processus cible exécute le service et retourne une valeur à l'appelant ;
- La fonction *callback* du premier processus est appelée avec comme argument la valeur retournée par le service.

La difficulté est de reproduire la séquence de construction d'un composant, association des variables de port comprises, en faisant appel à un unique service *CABLES*. Autrement dit, la construction du composant distant et la connexion à son environnement doivent être achevées à l'issue de l'appel de la fonction *callback*.

2.2 (qu'est-ce qu'un service NJN ?)

Pour gérer les opérations distantes, NJN définit des objets services qui renferment la référence vers une méthode à appeler et l'ensemble des arguments nécessaires pour effectuer l'appel. Parmi la liste des arguments, ceux qui peuvent être sérialisés le sont, les autres sont transmis sous forme d'objet *Handle* DOHC , afin de pouvoir être retrouvés localement.

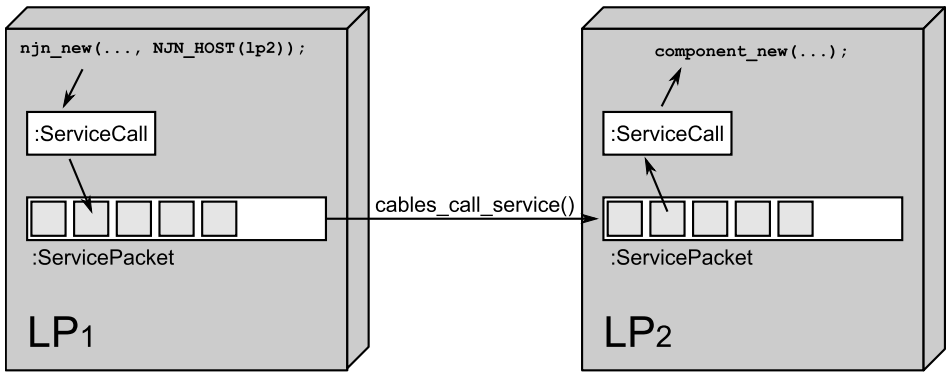

Figure 68. Appel de services pour la création d'objets à distance.

Au cours de l'exécution de l'application, les objets services sont organisés en paquets et placés dans une file d'attente (figure 68). Chaque paquet de la file contient les objets services à destination d'un processus logique. Lorsqu'un paquet est envoyé à un processus logique, il contient donc l'ensemble des objets services qui lui sont destinés. Cette façon de faire permet de réduire le nombre de requêtes *CABLES* entre processus.

Le processus récepteur reçoit l'ensemble des paquets qui lui sont destinés en provenance des autres processus à raison d'un paquet par processus émetteur. Les paquets reçus sont placés dans une file d'attente de réception. Chaque paquet est traité l'un après l'autre. Le traitement d'un paquet consiste à exécuter la méthode associée à chaque objet service contenu dans le paquet.

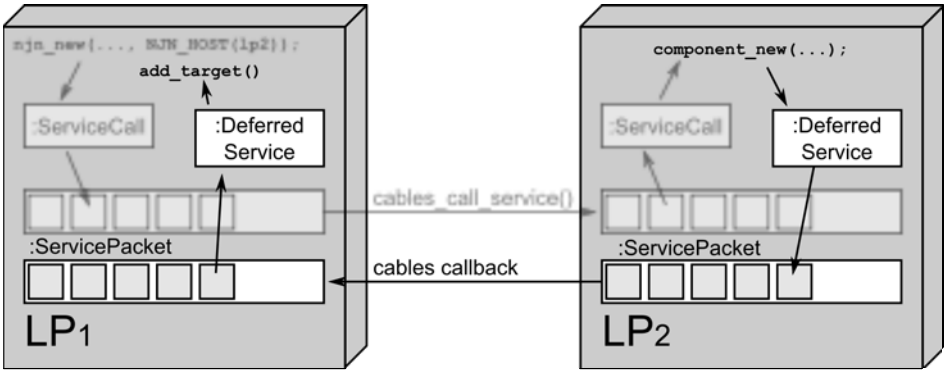

Figure 69. Retour de service.

Lors de l'exécution, la méthode appelée peut à son tour créer des objets services à destination du processus appelant (figure 69). Tous ces objets sont stockés temporairement dans un paquet de services spécifique appelé paquet différé qui est retourné à l'appelant comme argument de retour de service *CABLES*.

Le paquet de services différés est réceptionné par la fonction *callback* associée au service *CABLES* sollicité. La fonction *callback* traite l'exécution des objets services qu'elle reçoit pour terminer l'échange entre les deux processus.

2.3 … à distance

La figure 70 montre la séquence de création de composant à distance précédente en mettant en évidence les objets services exploités au cours de la construction.

La création du composant est prise en charge par un objet service de type *ComponentNewService* envoyé au processus cible. La connexion de la variable *target* à sa source est partiellement réalisée par un objet *AddTargetService* renvoyé au processus appelant à travers un paquet différé et exécuté par la fonction *callback* associée à l'appel du service. En un seul appel de service, l'instanciation du composant est donc finalisée.

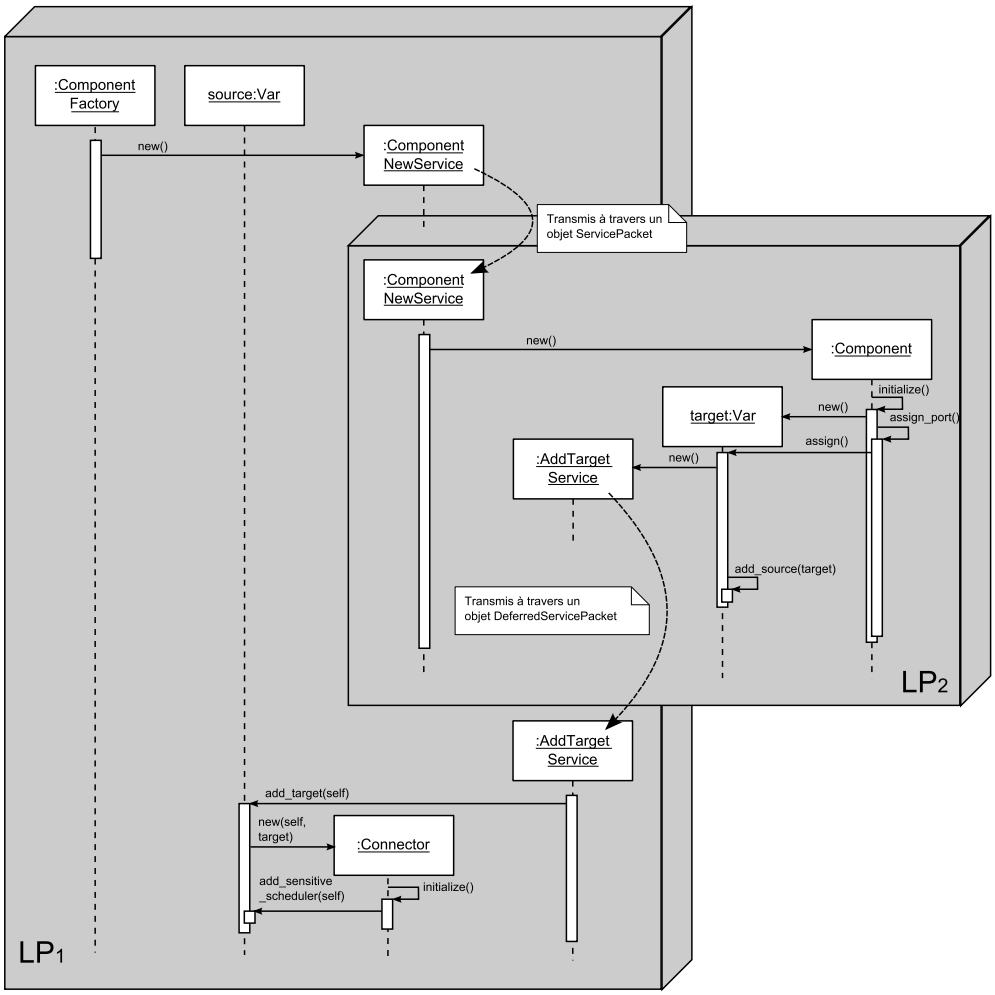

Figure 70. Création d'un composant à distance et connexion d'un port d'entrée.

3 Ordonnancement général et nettoyage de la mémoire

3.1 Les choses à faire et leur priorité

Le noyau d'NJN, gère la boucle principale d'exécution. Cette boucle lance les différentes actions gérées par le noyau selon leur ordre de priorité. Il s'agit des opérations suivantes classées de la plus prioritaire à la moins prioritaire :

- Traitement des retours arrière (réactivation des tâches) ;
- Réception et traitement des paquets de services ;
- Émission des paquets de services ;
- Exécution des tâches temps-réel à échéance ;

- Exécution des tâches temps-virtuel à échéance (en tenant compte de la garde) ;
- Nettoyage du graphe causal ;
- Mise en attente (lancement du ramasse-miette DOHC puis appel système *select)* ;

La priorité est donnée à la remise en état du système lorsqu'un retour arrière est effectué. Il s'agit principalement de réactiver les tâches dont l'exécution a été invalidée et qui doivent être exécutées avec de nouvelles données.

NJN traite ensuite les messages entre processus logiques. Les retours arrière étant provoqués par les messages échangés entre processus, traiter ces messages au plus tôt permet de réduire le nombre d'exécutions invalides.

L'exécution des tâches est ensuite lancée en donnant priorité aux tâches temps-réel arrivées à échéance.

Enfin, lorsqu'il reste du temps, NJN s'emploie à nettoyer les structures de données de l'application.

3.2 Le nettoyage des évènements et des messages

Pour effectuer le nettoyage des évènements, NJN doit disposer du FVT, la limite des objets fossiles. Les choses se passent en deux temps :

- A l'aide d'une structure de *heap*, NJN détermine la latence la plus grande parmi l'ensemble des tâches temps-réel à exécuter localement ;
- Cette information est ensuite partagée avec les processus logiques liés par des relations de dépendances (voir l'algorithme pages 70 et suivantes).

Une fois le FVT déterminé, il ne reste plus qu'à éliminer les évènements dont la date est antérieure à la limite fossile.

Lors de leur création, les évènements d'écriture sont placés dans une file globale au niveau du noyau d'NJN. Ils y sont classés en fonction de leur date. Chaque évènement d'écriture placé dans cette liste à une date antérieure au FVT est éliminée, à moins qu'il soit le dernier événement du signal. Son élimination provoque l'élimination des évènements de lecture qui lui sont associés ainsi que les évènements d'exécution qu'il a provoqués. Partant des évènements d'écriture, c'est donc toute la structure du graphe qui est nettoyée.

Une fois libéré de la structure, chaque évènement pourra être éliminé de la mémoire par le ramasse-miette de DOHC.

*
* *

Nous venons d'aborder l'infrastructure d'NJN sur laquelle repose le fonctionnement et la synchronisation des tâches de l'application. L'infrastructure assure la gestion des évènements de l'application au sein d'un graphe de causalité

qui retrace les dépendances entre les différentes opérations des tâches (exécution, lecture et écriture). Pour assurer le fonctionnement de l'application distribuée, l'infrastructure gère l'échange de messages entre processus logiques en s'appuyant sur les services *CABLES*. Enfin l'infrastructure gère l'ordonnancement général du noyau de l'application.

C'est sur les primitives de gestion des évènements et des messages que repose le fonctionnement de la superstructure, objet du chapitre suivant. Ce chapitre montrera le lien entre les éléments de la superstructure et les primitives de l'infrastructure.