Définition et implémentation d'un modèle causal d'exécution temps-réel distribuée

Sébastien Moutault

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 : SMI - Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur de

L'École Nationale Supérieure des Mines de Paris

Spécialité « Informatique temps réel, robotique et automatique »

présentée et soutenue publiquement par

Sébastien MOUTAULT

le 16 décembre 2011

DÉFINITION ET IMPLÉMENTATION D'UN MODÈLE CAUSAL D'EXÉCUTION TEMPS-RÉEL DISTRIBUÉE

Directeur de thèse : Claude LAURGEAU
Encadrant de thèse : Bruno STEUX

Jury :

M. Serge DULUCQ, Professeur des universités	Président
M. Christian FRABOUL, Professeur des universités	Rapporteur
M. Claude LAURGEAU, Professeur des universités	Examinateur
M. Francis LEPAGE, Professeur des universités	Rapporteur
M. Bruno STEUX, Maître assistant	Examinateur

MINES ParisTech
CAOR – Centre de Robotique, Mathématiques et Systèmes
60, Boulevard Saint-Michel 75272 PARIS cedex 06

THÈSE

À Cécile.

REMERCIEMENTS

Ces travaux ont été réalisés au sein du centre de robotique de l'École des Mines de Paris, en collaboration avec l'INRIA et les entreprises VALEO et Intempora.

Je tiens tout d'abord à remercier M. Claude Laurgeau, créateur du centre de robotique, qui m'a fait le grand honneur de diriger cette thèse. Je suis très fier d'avoir été le dernier doctorant qu'il ait dirigé durant son immense carrière.

Mes remerciements vont ensuite à M. Arnaud de la Fortelle, directeur du centre, pour m'avoir accueilli au sein de son laboratoire.

Je remercie ensuite M. Bruno Steux, mon encadrant et ami, avec qui travailler a été un réel plaisir. Malgré la distance, il a été extrêmement présent et disponible tout au long de ces travaux.

Ma gratitude va ensuite aux membres de mon jury pour avoir accepté d'évaluer mon travail, en particulier à M. Francis Lepage et M. Christian Fraboul, mes rapporteurs ainsi qu'à M. Serge Dulucq, président de mon jury.

Merci à l'ensemble de mes collègues du CAOR pour leur accueil toujours sympathique. Un merci tout particulier à mon ami Joël Senpo-Roca dont l'écoute à été d'un grand réconfort en temps de crise. Merci également à Bogdan Stanciulescu dont j'ai si souvent envahi le bureau ainsi qu'à Christine Vignaud et Christophe Kotfila qui m'ont rendu de grands services lorsque l'administration exigeait mon impossible présence à l'École.

Je ne remercierai jamais assez Mme Valérie Tainturier et Mme Béatrice Pellé – et à travers elles ARMINES – pour l'efficacité avec laquelle elles se sont chargées de mon dossier de détachement.

En parallèle de ces travaux, j'ai exercé mon métier d'enseignant à l'IUT de Bordeaux. J'adresse donc mes remerciements à Messieurs Pierre Lafond, ancien directeur de l'IUT Bordeaux 1, Serge Dulucq, directeur, Dominique Rebière, chef de département, Florent Arnal, chef adjoint, et Benjamin Caillard, responsable des emplois du temps, pour avoir simplifié ma double vie professionnelle, autant que cela leur était possible. Merci également à l'ensemble de mes collègues de l'IUT pour leur compréhension durant ces cinq années.

Faire une thèse à Paris en travaillant à Bordeaux demande, en plus d'une logistique bien rodée, des amis fidèles. Merci à Céline et Guillaume (et Arthur), Charlotte et Christophe ainsi qu'à Veronique de m'avoir hébergé si souvent et avec autant de gentillesse, de mets succulents et de breuvages d'exceptions.

Merci à mes Parents de leur soutien inflexible tout au long de ces années. Ça y est, j'ai enfin terminé mes études ! Un merci particulier à ma mère qui a relu mon manuscrit.

À Cécile enfin, ma compagne, qui, d'une thèse à l'autre, a eu la patience d'attendre si souvent, l'énergie de m'encourager toujours et pour finir le courage de lire les 220 pages de mon indigeste manuscrit, plus qu'un merci...

TABLE DES MATIÈRES

INTRODUCTION.....1

PREMIÈRE PARTIE
UN MODÈLE D'EXÉCUTION TEMPS-RÉEL DISTRIBUÉ DÉTERMINISTE

CHAPITRE 1
L'ORDONNANCEMENT ÉVÈNEMENTIEL TEMPS-RÉEL PEUT-IL ÊTRE DÉTERMINISTE ?.....15

1 De l'origine du non-déterminisme.....17
1.1 Où l'on analyse l'origine du non-déterminisme comportemental.....17
1.2 Où l'on fixe la durée apparente d'exécution des tâches – Le temps logique d'exécution.....19
1.3 Le temps, la cause et la conséquence.....21

2 La causalité au cœur de l'exécution – La simulation distribuée à évènements discrets.....22
2.1 Parallélisme et causalité – Principe de la simulation à évènements discrets.....23
2.2 La simulation distribuée conservative.....25
2.3 L'optimisme de Jefferson et la distorsion du temps.....27

3 De la simulation à l'exécution temps-réel.....32
3.1 Un modèle d'exécution distribuée conservatif.....32
3.2 Les protocoles optimistes sont-ils ordonnançables ?.....34
3.3 La synchronisation temps-réel des simulateurs distribués optimistes.....38

CHAPITRE 2
VERS UN MODÈLE ÉVÈNEMENTIEL TEMPS-RÉEL DÉTERMINISTE.....43

1 Principe de synchronisation temps-réel.....44
1.1 Le cadre exécutif : des tâches, des signaux, des processus et des messages. .44
1.2 Deux types de tâches : les tâches temps-virtuel et les tâches temps-réel.....46
1.3 Rôle de la latence.....47
2 Politique d'ordonnancement.....49
2.1 Comment trop d'avance peut compromettre le respect des contraintes temps-réel 49
2.2 Le délai de garde.....51
2.3 Estimation du délai de garde en temps-réel.....53
3 Vers l'ordonnancement temporel.....56
3.1 Une comparaison avec l'ordonnancement temporel.....56
3.2 Vers l'ordonnancement temporel.....57

CHAPITRE 3
LE PROTOCOLE DE SYNCHRONISATION.....59

1 Des tâches au temps.....59
1.1 Du modèle structurel au modèle d'exécution.....60
1.2 Les problèmes de propagation, de convergence et de causalité : deux cas tordus 62
1.3 De la définition du temps.....64
2 Le graphe causal et le protocole de synchronisation.....66
2.1 Topologie du graphe causal.....67
2.2 Le retour arrière.....68
2.3 Nettoyage de la mémoire.....70
3 Quand rien ne va plus.....73
3.1 Découpler les niveaux hiérarchiques de l'application.....73
3.2 Respecter la latence des tâches temps-réel.....74
3.3 Fiabiliser l'exécution des tâches critiques.....75

DEUXIÈME PARTIE
UN MODÈLE STRUCTUREL DYNAMIQUE DISTRIBUÉ

CHAPITRE 4
COMPOSANTS, TÂCHES, VARIABLES – LE MODÈLE STRUCTUREL. .81

1 Des composants qui hébergent des variables et des tâches.....81

1.1 Deux familles d'objets : les conteneurs et les terminaux.....82
1.2 Des propriétés et des paramètres communs.....82
1.3 Portée et durée de vie des objets.....83

2 Le composant est l'objet structurel de base.....85

2.1 Le port d'entrées/sorties relie le composant à son environnement.....85
2.2 Les propriétés et paramètres renseignent sur l'état du composant.....86
2.3 Le corps décrit la structure ou le comportement du composant.....87
2.4 Héritage entre composants.....87

3 Les variables, leur type et leur comportement.....88

3.1 Le typage des variables est dynamique.....89
3.2 Comportement temporel de l'affectation des variables.....89
3.3 Connectivité et variable de port d'entrées/sorties.....89
3.4 Propriétés et paramètres des variables.....90

4 Les propriétés et les paramètres.....91

4.1 Définition et politique d'accès des propriétés et paramètres.....91
4.2 Association d'une propriété à une variable.....92
4.3 Définition de paramètre de composant ou de canal.....93

5 Les tâches portent le code actif de l'application.....93

5.1 Quatre variantes de tâches : les tâches temps-réel, les tâches temps-virtuel, les boites d'horloges et boites de découplage.....94
5.2 Déclenchement sur évènement interne.....94
5.3 Déclenchement sur évènement externe.....95
5.4 Déclenchement temporel : Timeout et Timeslot.....96
5.5 Un corps en trois blocs : initial, final et always.....97
5.6 La latence des tâches temps-réel.....98

6 Des méthodes pour abstraire les échanges d'informations.....99

6.1 Définition.....99
6.2 Topologie.....100
6.3 Appel.....101

7 Les canaux structurent les interconnexions.....102

7.1 Principe topologique et définition.....102
7.2 Les protocoles.....104

CHAPITRE 5
CONSTRUCTION DES OBJETS, DISTRIBUTION ET GESTION DYNAMIQUE DE L'APPLICATION – LE MODÈLE DE RECONFIGURABILITÉ ET LE MODÈLE DE DISTRIBUTION.....109

1 La construction des objets, les paquetages et les bibliothèques.....110
1.1 Modèle et instance.....110
1.2 Paquetages et recherche des modèles.....111
1.3 Bibliothèques et gestion des dépendances.....112

2 La reconfiguration dynamique de l'application.....113
2.1 Les objets, leur ligne de vie et les retours arrière.....113
2.2 De njn_new() à njn_delete() – la durée de vie des objets.....114
2.3 Assurer la continuité de service.....115

3 Vecteurs et algorithmes structurels.....119
3.1 Objets et vecteurs d'objets.....120
3.2 Auto-dimensionnement, pseudo-composants mux et demux.....120
3.3 Algorithmies structurelles.....121

4 La distribution.....123
4.1 Connexions point à point – Les prises et les fiches.....124
4.2 Distribution – Les composants et leur ombre.....125

CHAPITRE 6
ÉTUDE DE CAS : LE PROJET COREBOTS.....129

1 Présentation du projet et du robot.....130
1.1 Le défi CAROTTE.....130
1.2 L'équipe CoreBots.....132
1.3 Le robot CoreBot M.....132

2 Analyse du problème.....134
2.1 L'architecture logicielle.....134
2.2 Fiabilité.....136
2.3 Synchronisation.....137

3 La solution avec NJN.....138
3.1 Architecture et distribution.....138
3.2 Fiabilité.....139
3.3 Synchronisation.....140

TROISIÈME PARTIE
L'IMPLÉMENTATION

CHAPITRE 7
ABSTRACTION DE LA MÉMOIRE, DE LA PLATEFORME ET DE LA COMMUNICATION – LES BIBLIOTHÈQUES SUPPORTS.....145

1 Le double arbre à cames en tête.....145

1.1 Les objets DOHC de base et les conteneurs.....146
1.2 L'embrayage (la couche objet).....148
1.3 La sérialisation.....150
1.4 La gestion de la mémoire.....151

2 Le châssis.....153

2.1 La gestionnaire des évènements.....153
2.2 La gestion du temps.....154
2.3 La gestion des entrées/sorties.....154

3 Les câbles.....157

3.1 Déclarer un service et démarrer le serveur.....157
3.2 Établir la connexion et faire un appel de service.....157

CHAPITRE 8
LA MACHINE (INFRASTRUCTURE).....159

1 Les signaux, les évènements et les gestionnaires de tâches.....160

1.1 Les évènements et la structure du graphe causal.....160
1.2 Le gestionnaire de tâche, les évènements d'exécution et leur état.....162
1.3 La mise en œuvre du retour arrière.....164

2 Les messages, les paquets et le gestionnaire de services.....165

2.1 Un simple appel de méthode.....165
2.2 (qu'est-ce qu'un service NJN ?).....166
2.3 … à distance.....167

3 Ordonnancement général et nettoyage de la mémoire.....168

3.1 Les choses à faire et leur priorité.....168
3.2 Le nettoyage des évènements et des messages.....169

CHAPITRE 9
LA MACHINE (SUPERSTRUCTURE).....171

1 Fabriques, bibliothèques de modèles et paramètres.....171

1.1 Les objets, leur modèle et leur fabrique.....172
1.2 Mécanisme de construction de l'objet.....173
1.3 Chargement des bibliothèques.....175

2 Structure hiérarchique.....177
2.1 Les objets et leur ligne de vie.....178
2.2 Les composants et leur contenu.....180
2.3 Les variables, les ports et les connecteurs.....181
3 La distribution, les aller/retours et les ombres.....183
3.1 Tant de choses à faire (en un aller/retour).....183
3.2 La construction des composants et de leur ombre.....184
3.3 La vie d'un composant distant vue de son ombre et vice-versa.....186

CONCLUSION ET PERSPECTIVES.....189

BIBLIOGRAPHIE.....199

INTRODUCTION

Cette thèse s'inscrit dans le cadre du projet AROS (*Automotive Robust Operating Services*), projet coordonné par le centre de robotique de Mines-ParisTech, en partenariat avec l'INRIA et les entreprises Valéo et Intempora. Il a pour objectif de proposer un outil de prototypage rapide d'applications distribuées temps-réel. Bien que son domaine d'application puisse être relativement large – les systèmes embarqués et/ou distribués – AROS s'intéresse spécifiquement aux problématiques du développement logiciel dans les secteurs de **l'automobile** et de **la robotique mobile**.

L'environnement et la sécurité sont deux préoccupations actuelles majeures du secteur automobile. La première pousse les industriels vers des solutions de motorisation toujours plus élaborées : moteurs hybrides, contrôle de l'injection [38], récupération d'énergie au freinage, *stop and start*, etc. Concernant la seconde, on peut noter que sur les routes de France, l'ONISR déplorait encore en 2009 plus de 74 000 accidents et près de 4 500 décès. Or, dans 95% des cas, la responsabilité du conducteur est engagée. La logique voudrait qu'à l'avenir l'industrie et les pouvoirs publics travaillent au remplacement de l'élément défaillant du système – le conducteur – par un système automatisé plus fiable [47]. La multiplication des systèmes d'aide à la conduite participent de cet effort : freinage d'urgence (BAS), antipatinage (ABS), contrôle de trajectoire (ESP), protection en cas d'accident (Airbag), contrôle de distance de sécurité (ACC) etc. Mais pour aller plus loin, c'est-à-dire pour automatiser la conduite, les véhicules devront non seulement communiquer avec l'infrastructure routière et des serveurs de trafic mais également échanger des informations entre eux.

L'an 2000 annonçait une révolution robotique dans notre quotidien. Dix ans après, deux constats s'imposent : l'humanoïde grand public a encore un peu de difficultés

à tenir debout [61] mais la « technologie » avance tout de même à pas de géant. Tandis que les aspirateurs automatiques envahissent petit à petit les maisons individuelles [70], on parvient à cartographier un territoire inconnu à l'aide de robots totalement autonomes [8]. Pour le futur, les idées d'applications robotiques ne manquent pas : applications chirurgicales, militaires, spatiales, industrielles, etc. L'actualité nous fait rêver à des robots capables d'interventions en milieu fortement radioactif ; les militaires réfléchissent à des robots capables d'intervenir sur le champ de bataille ; on envisage des systèmes capables de prendre en partie en charge la dépendance des personnes, etc. Notre environnement s'automatise et se robotise de plus en plus. A l'avenir, il ne s'agira plus seulement de concevoir un robot capable d'effectuer une tâche donnée mais d'inscrire chaque nouveau système dans un schéma d'interaction global ou interviennent l'environnement, l'être humain et les autres systèmes robotiques. Plus encore, certaines tâches complexes seront prises en charge par des systèmes robotisés coopératifs [6].

Les applications logicielles dans ces deux domaines, automobile ou robotique, ont en commun plusieurs caractéristiques : leur caractère **distribué**, **temps-réel** et leur **exigence de fiabilité**.

Ce sont d'abord des applications distribuées sur des cibles hétérogènes et à travers des réseaux hétérogènes. Le nombre de processeurs embarqués dans les véhicules va grandissant [47], de même, il n'est pas rare de concevoir un robot équipé de plusieurs processeurs. Ces unités communiquent via des bus dédiés (réseaux I2C, CAN, etc.). A une échelle plus réduite, les architectures multi-cœurs se généralisent, y compris chez les micro-contrôleurs de taille modeste. Cette tendance pourrait tendre à rendre, à terme, le problème de la distribution prépondérant vis-à-vis de celui de l'ordonnancement. A une plus grande échelle, la gestion du trafic routier est confiée à des serveurs en réseau qui seront, dans un avenir proche, capables de communiquer avec les véhicules (réseau WAVE ou CALM). Sans une abstraction forte des architectures d'une part et de la topologie du réseau d'autre part, le développement des applications est difficile à appréhender de façon globale et hiérarchique.

Ce sont ensuite des applications réactives associées à des contraintes temps-réel de dureté plus ou moins grande. La mise en œuvre d'un contrôle moteur demande une grande rigueur dans le respect des contraintes temps-réel. A l'inverse, la gestion du trafic constitue une famille d'applications moins exigeante de ce point de vue. Par ailleurs, les algorithmes mis en œuvre en robotique, par exemple, sont d'une complexité telle qu'il est difficile d'en prévoir le comportement temporel avec certitude. Dans le même ordre d'idée, les performances des réseaux sont à l'heure actuelle considérables (Gigabits) mais leur latence n'est pas déterministe. Sauf pour des sous-ensembles très localisés, les systèmes distribués dont il est question ici s'accommodent donc mal de l'orthodoxie traditionnelle du développement d'applications temps-réel distribuées basée sur le pire temps d'exécution. Il faut donc envisager les choses de manière plus souple.

Enfin, certaines de ces applications doivent être en fonctionnement permanent avec un niveau élevé de sécurité. La fiabilité des applications est donc capitale. Elle peut être obtenue en mettant en œuvre des techniques de redondance, de monitoring, etc. Il faut par ailleurs être capable de maintenir ces applications sans qu'il soit nécessaire d'en interrompre le fonctionnement.

L'objectif d'AROS est d'offrir une solution cohérente et homogène aux problématiques soulevées par la robotique et l'automobile, sous la forme d'un outil de prototypage d'applications à la fois **temps-réel**, **distribuées**, **reconfigurables dynamiquement**, **fiables** et **agiles**[1] au regard de la complexité de l'application.

AROS doit permettre de développer des applications temps-réel plus ou moins dures, en tout cas temporellement maitrisables[2]. Le domaine privilégié visé par AROS concerne les systèmes distribués dont la complexité rend la preuve difficile à atteindre bien que le système montre expérimentalement sa fiabilité. Une grande variété d'exigence temps-réel existe que l'on peut exprimer de manière probabiliste. Cette marge de manœuvre entre le temps-réel mou et le temps-réel dur permet d'envisager des systèmes plus souples à développer que ceux basés sur un ordonnancement temporel et sur le pire temps d'exécution. Sous certaines conditions, AROS devra tout de même permettre de développer des applications temps-réel strictement dures, mais il n'a pas pour vocation de déterminer l'ordonnancement temporel d'une application ni d'apporter la preuve d'ordonnançabilité de cette application. Au minimum, l'ordonnancement étant déterminé, AROS doit permettre sa mise en œuvre moyennant certains conditions d'ordre architecturales. Les zones dures devront être sécurisées et isolées du reste de l'application.

AROS doit permettre de mettre en œuvre des systèmes distribués hétérogènes hiérarchisés. Du contrôle moteur au serveur de trafic, en passant par la communication inter-véhicule, AROS doit gérer tous les niveaux hiérarchiques et toutes les plateformes matérielles. Rappelons qu'il est courant aujourd'hui d'utiliser des plateformes matérielles mutli-cœurs. La distribution devient de ce fait un élément prépondérant vis-à-vis du problème de l'ordonnancement des tâches. Pour cette raison, AROS ne doit pas se substituer au système d'exploitation – éventuellement temps-réel – qui a en charge l'ordonnancement des processus. Pour AROS, un processus système est un organe distribué. Pour faciliter cette approche AROS doit gérer la distribution de façon totalement transparente pour le développeur. L'application a la même forme, qu'elle soit distribuée ou non, indépendamment de la topologie du réseau.

L'architecture d'une application AROS doit être une structure dynamique. Pendant l'exécution, il doit non seulement être possible d'ajuster certains paramètres, mais aussi de modifier les algorithmes, de supprimer ou d'ajouter des composants ou

1 C'est-à-dire faciles à utiliser et à maintenir pour le développeur.

2 Ce manuscrit tente de respecter les règles de la nouvelle orthographe (Journal officiel de la République française du 6 décembre 1990, www.orthographe-recommandee.info) qui font, entre autres choses, disparaitre les accents circonflexes sur les « i ».

des fonctionnalités. L'objectif est de pouvoir intervenir sur l'application pendant son fonctionnement pour la corriger, la mettre à jour ou l'adapter à un changement d'environnement. De même, il doit être possible de modifier la distribution de l'application pendant l'exécution pour, par exemple, ajuster la charge en cours de fonctionnement ou prendre en compte les connexions ou déconnexions d'opérateurs distants. Dans la gestion d'une infrastructure routière par exemple, il peut être intéressant d'associer à chaque véhicule un composant logiciel. Les différentes instances de ce composant devront apparaitre ou disparaitre en fonction de l'évolution du trafic routier.

Bien que dynamique et distribué, AROS doit fournir un cadre d'exécution fiable pour les applications. Il doit donc être possible d'isoler des parties critiques de l'application, de faire du diagnostique et du monitoring pendant l'exécution. En cas de dysfonctionnement grave de certaines parties du code, il doit être possible de relancer les composants défectueux sans que l'application dans son ensemble soit mise en péril. Les capacités dynamiques d'AROS sont ici exploitées pour sécuriser le fonctionnement des applications [87].

Une application AROS doit se développer de façon très simple à partir de bibliothèques de composants réutilisables. Dans ce cadre de développement, un composant est un objet structurel qui héberge du code actif ou d'autres composants et qui dispose d'entrées et de sorties. L'interconnexion de composants entre eux construit une application. Le développement, la simulation, le débugge et l'exécution doivent être réalisés dans le même instant. L'application est développée pendant qu'elle s'exécute, sans être interrompue, éventuellement avec des parties de l'application simulées. En temps-réel, le développeur doit pouvoir diagnostiquer les dépassements de latence, les problèmes de synchronisation, les déséquilibres de charge, les dysfonctionnements sporadiques, etc. et les corriger sans arrêter le système. Il doit également pouvoir revenir en arrière sur une exécution pour voir l'effet d'une modification algorithmique sur le comportement de l'application dans une configuration singulière clairement identifiée. Le flot de conception dans AROS doit être naturellement continu. Il ne doit pas être nécessaire de redévelopper l'application pour la cible visée. Tout au plus une couche d'abstraction minimale et générique doit être développée pour rendre une nouvelle cible matérielle compatible avec AROS.

Pour résumer, AROS doit permettre de développer des applications temps-réel avec divers degrés de dureté, à partir de composants réutilisables et dans des contextes de distribution hétérogènes, tant du point de vue des cibles matérielles, des systèmes d'exploitation que des réseaux de communication. Une application AROS doit pouvoir être modifiée dynamiquement mais offrir une grande sureté de fonctionnement en permettant de mettre en œuvre de la redondance et des outils de débugge et de diagnostic élaborés.

Pour répondre à ce cahier des charges, on envisage une plateforme qui s'articule autour de quatre modèles : un **modèle structurel**, un **modèle d'exécution**, un

modèle de distribution et un **modèle de reconfigurabilité**. Les paragraphes qui suivent en exposent les idées générales.

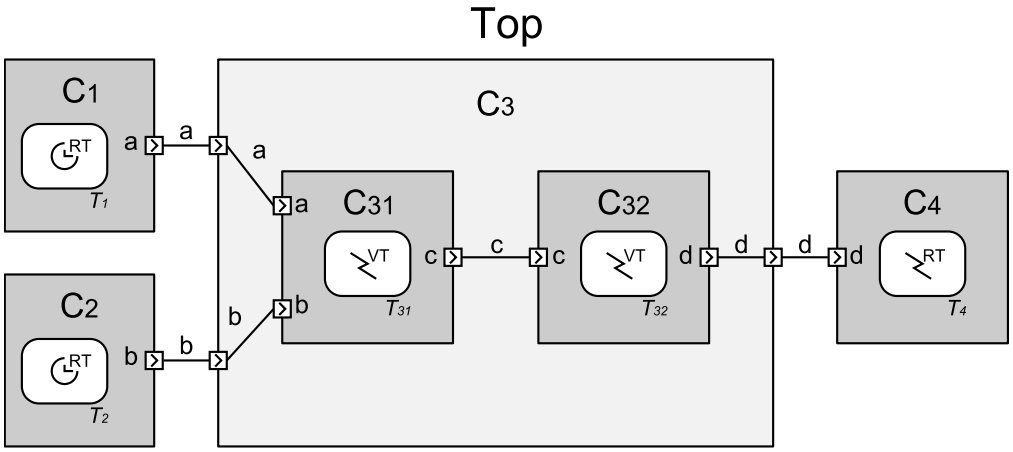


Figure 1. Le modèle structurel. Les composants hébergent des tâches et sont reliés entre eux par des variables.

Comme nous l'avons énoncé plus haut, AROS doit s'appuyer sur une approche orientée composant. La figure 1 montre un exemple d'application. Les composants sont disponibles en bibliothèques ou peuvent être créés pendant le processus de développement. Ils sont instanciés pour former une application. Un composant comporte des ports d'entrées et de sorties et héberge des tâches. Les tâches constituent le code actif de l'application. Un composant peut également être constitué d'un assemblage d'autres composants dont les ports sont interconnectés par l'intermédiaire de variables. Les composants sont donc à la base des possibilités de conception hiérarchisée des applications. L'application elle-même est définie par le composant de niveau le plus haut dans la hiérarchie de la structure de l'application.

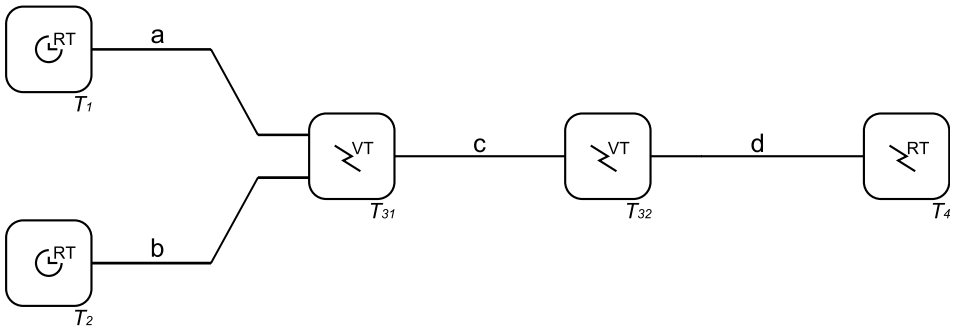


Figure 2. Le modèle d'exécution. Les tâches communiquent entre elles à travers des signaux.

Dans l'exécutif ne subsistent de cette structure hiérarchique que les tâches et les chemins de données qui les lient. La figure 2 montre ce qu'il reste de l'application de la figure 1. Les tâches sont restées identiques tandis que les variables et les

ports d'entrées/sorties ont permis de constituer des signaux par lesquels les tâches échangent leurs données.

Le choix du composant comme élément structurant d'une application conduit à un exécutif morcelé en un grand nombre de tâches. Parmi elles, on distinguera deux catégories de tâches : les tâches d'entrées/sorties d'une part, et les tâches de traitement d'autre part. Les tâches d'entrées/sorties échangent des informations avec le monde extérieur. Schématiquement, les tâches d'entrées échantillonnent l'état de capteurs et les tâches de sorties pilotent des actionneurs.

Pour ces tâches, le contrôle des instants d'exécution est capital puisqu'il s'agit d'instants d'interaction avec le monde réel. Les tâches de traitement opèrent des transformations des données de manière à établir un lien entre captures et commandes. Dans la mesure où les opérations sont effectuées dans un laps de temps raisonnable, l'instant exact d'exécution de ces tâches de traitement n'est pas essentiel, pourvu que le résultat du traitement n'en dépende pas. La figure 3 montre ce qu'il en est du modèle d'exécution pour l'exemple simple de l'application de la figure 1.

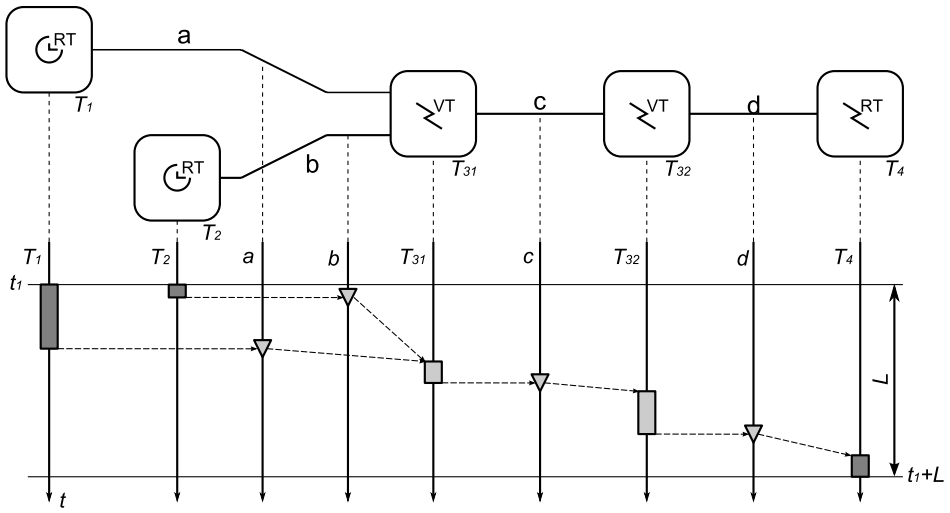


Figure 3. Exécution d'une application. Le temps est représenté du haut vers le bas. Chaque tâche inscrivant une valeur sur le signal qu'elle pilote provoque l'exécution de la tâche suivante.

L'instant t_1 d'exécution de la tâche d'entrée est déterminé par un évènement extérieur ou une périodicité d'échantillonnage. La tâche de sortie est exécutée au bout d'un délai fixe noté L par rapport à l'instant t_1. Entre les instants t_1 et t_1+L, les tâches de traitement sont exécutées sans qu'il soit nécessaire de fixer les instants exacts d'exécution. La production de données d'une tâche entraine l'exécution de la tâche suivante, et ainsi de suite jusqu'à la tâche de sortie. Le modèle d'exécution définit parfaitement la latence du système, c'est-à-dire son délai de réaction, tout en offrant une certaine souplesse d'ordonnancement pour les tâches de traitement. Cette souplesse pourra être mise à profit pour gérer les latences de communication lorsque l'application est distribuée. Cependant, on veillera à garantir la

reproductibilité du comportement de l'application quelque soit l'ordonnancement et la distribution. Ce dernier point est l'un des éléments essentiels du modèle élaboré pour AROS.

L'élément structurant d'AROS étant le composant, il est assez naturel d'en faire l'élément de base du modèle de distribution. Ainsi pour distribuer une application sur un ensemble de processus et/ou de cibles matérielles, il suffit de mapper chaque composant sur l'un des processus identifié dans la topologie réseau. AROS doit se charger d'établir les communications de façon transparente. Il n'est donc pas nécessaire de modifier la structure hiérarchique de l'application pour effectuer la distribution. Dans le cas d'un réseau de type Ethernet, il est en outre inutile de préciser la topologie exacte du réseau. Les choses pourraient être moins transparentes lorsque le support de communication est un réseau spécifique (CAN, RS232, I2C, etc.). Aucune limite à priori n'existe du moment que la cible est accessible et qu'elle dispose des ressources matérielles nécessaires (mémoire, entrées/sorties spécifiques, etc.). Au besoin, il est possible de créer de nouveaux processus sur une cible matérielle identifiée dans le réseau pour y héberger des composants que l'on souhaite isoler du reste de l'application. Notons enfin que l'on doit pouvoir modifier la distribution pendant l'exécution afin, par exemple, d'équilibrer la charge entre les différentes cibles matérielles sans interrompre l'exécution.

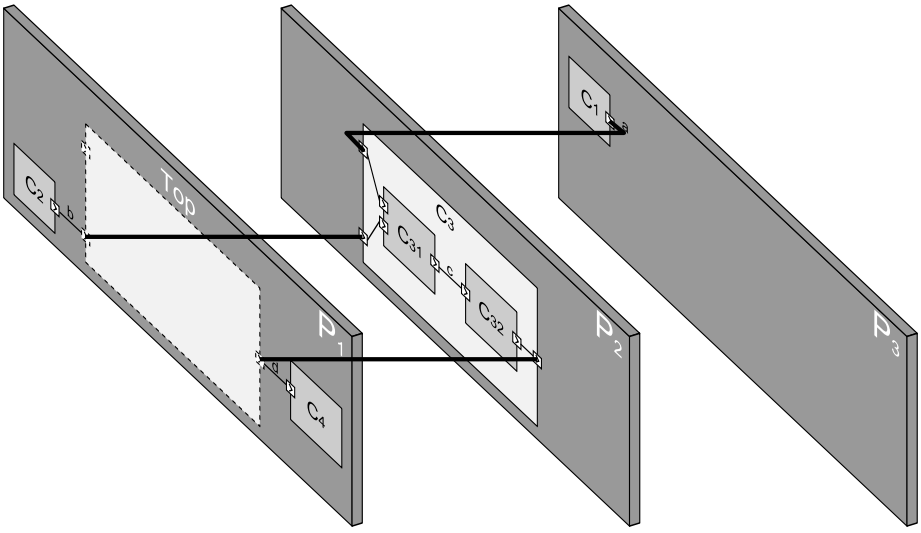


Figure 4. Modèle de distribution. Les composants de l'application sont distribués sur plusieurs processus. Les interconnexions entre composants sont établies à l'aide de communications entre processus.

Les possibilités de reconfiguration dynamique d'AROS doivent cependant aller bien au delà de la simple répartition de charge. Il doit être en effet possible de reconfigurer l'ensemble de l'application pendant son exécution ce qui interdit toute analyse statique de la topologie ou de l'ordonnancement de l'application. Les opérations suivantes sont envisagées : ajouter une tâche, supprimer une tâche,

instancier un composant, connecter ses entrées et ses sorties à des variables, effectuer une déconnexion, supprimer un composant, etc. Ces opérations doivent être possibles localement mais également sur des cibles distantes.

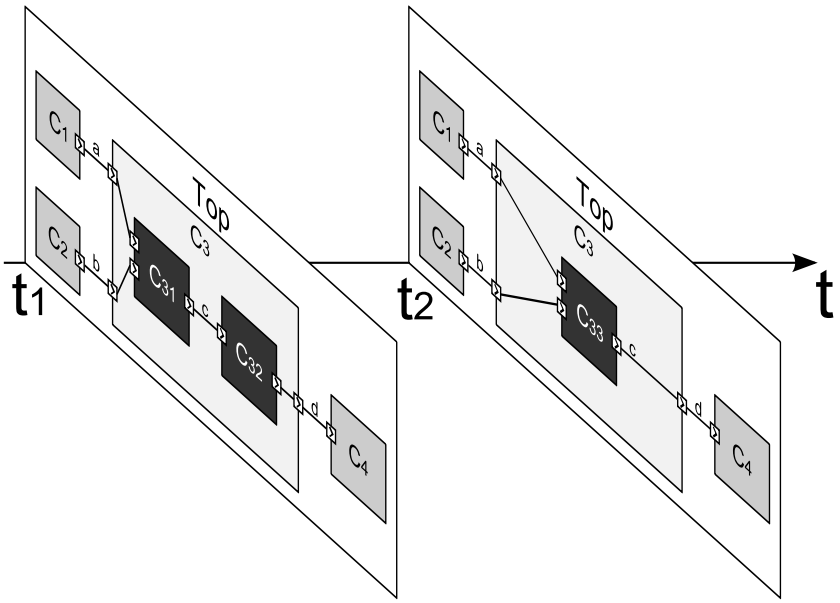


Figure 5. Modèle de reconfigurabilité dynamique. Les composants peuvent être créés ou détruits pendant l'exécution. Ici les composants C_{31} et C_{32} sont supprimés au profit de C_{33}.

Ses opérations de modification structurelle de l'application seront vraisemblablement à l'initiative d'un opérateur/développeur humain dans la majorité des cas via une interface de développement. Mais il doit également être possible de confier ce travail à des tâches de manière à ce que l'application soit en mesure de s'adapter seule aux différents contextes qu'elle rencontre au cours de son exécution.

Le modèle d'AROS étant dévoilé, examinons à présent les outils existants sur le marché de prototypage rapide. Plusieurs outils existent. Le tableau I donne les principales caractéristiques d'un échantillon représentatif du marché.

Matlab/Simulink [75] (The Mathworks) et LabVIEW [45] (National Instruments) pour ne citer que les plus connus, sont dotés d'un moteur d'exécution synchrone de type évènementiel parfaitement déterministe d'un point de vue comportemental mais difficilement contrôlable dans un contexte temps réel. Ils offrent une bonne plage de granularité de composants et permettent d'intégrer (via DLL) des bibliothèques externes. Des outils spécifiques permettent de produire du code temps réel sur plate-formes prioritaires ou sur PC, voire sur FPGA (LabVIEW). Leurs principaux défauts concernent leurs limites en termes de distribution et l'absence totale de dynamisme architectural des applications produites. Matlab/Simulink offre, en outre, des possibilités de simulation multi-domaine

	RT-Maps	Simulink	LabView	Op.Mod.	Ptolemy-II	SynDEx
Facilité d'utilisation	++ C++ (dév. composants)	+++	+++	++	+	-
Granularité	Grosse (thread)	Fine (niveau fonction)	Très fine (opération élémentaire) à Grosse	Fine (équat.)	Fine (opérateur)	Fine (opérateur)
Intégration librairies externes	+++ (C/C++ natif + support Win32 et linux)	+ (chargement DLLs. Interface type matrice)	+ (chargement DLL)	-	- (Java)	Non (pas Win32 ni Linux)
Exécution temps-réel	Mou. Plateforme PC.	Real-Time workshop. Génération de code, OS propriétaire.	LabView Real-Time. OS propriétaire.	Non.	Génération C/C++ limitée.	Dur. OS-less. Pas Linux ni Win32.
Synthèse matérielle	Non	Non	Oui, sur plateforme spécifique	Non	Non	Oui
Exécutif pour Automobile	Non. Prototypage sur PC seulement.	Génération de code.	Non	Non	Non	Support de OSEK
Distribution	Sockets	Sockets (DS toolbox)	Mémoire partagée. Sockets implicites	Non	Sockets	Transparente
Dynamique	Partiel	Non	Non	Non	Non	Non
Simulation synchrone	Non	Oui	Non (programmation graphique)	Oui	Oui	Simulation d'horloges only
Front Panel	Support .NET + visu bimaps	Third parties.	Oui	Non	Non	Non
Multi-process	Multi-threads only	Non	Non	Non	Non	Oui
BDD	Oui	Non	Non	Non	Non	Non

Tableau I. Comparatif des outils de prototypage.

(numérique, analogique, etc.) tandis que LabVIEW est plus un environnement de programmation graphique qui reprend les concepts de base de la programmation procédurale.

En marge de ces deux succès commerciaux, quelques outils universitaires offrent des possibilités intéressantes. SynDEx [84] est l'un d'eux dont le but est d'optimiser un *mapping* d'opérateurs logiciels sur un réseau de processeurs. Le calcul de la distribution est fait statiquement en fonction d'un critère de latence. L'outil produit le code exécutable destiné à chaque cible du réseau. Bien que la distribution soit ici beaucoup plus élaborée que dans les deux environnements présentés précédemment, l'outil souffre d'une absence totale de possibilités dynamiques. OpenModelica [62] est un langage déclaratif de modélisation multi-domaine (mécanique, électrique, hydraulique, etc.) orienté composant. Il s'agit d'un langage de simulation continue qui n'a aucune propriété temps-réel. Dans le cadre du projet AROS, il est envisagé de pouvoir importer dans une application des composants OpenModelica. Ptolemy II [66] est une plateforme d'expérimentation orientée acteurs. Un acteur est un composant logiciel qui s'exécute en parallèle d'autres acteurs et qui communique avec ceux-ci à l'aide de messages envoyés sur un réseau de ports et d'interconnexions. La sémantique d'un modèle Ptolemy II dépend d'un composant logiciel spécifique intégré au modèle, le directeur, qui peut être choisi parmi différents domaines (temps discret, temps continu, synchrone/réactif, etc.). Les domaines peuvent être mélangés au sein de la hiérarchie de l'application pour former un modèle multi-domaine.

Le logiciel RTMaps [72] a ceci de particulier qu'il a été développé au centre de robotique de Mines-ParisTech [82]. Il est donc l'ascendant historique du projet AROS. Il fonctionne avec un moteur d'exécution asynchrone. Chaque composant RTMaps fonctionne dans sa propre tâche ce qui simplifie grandement le développement d'applications *multi threads*. La conception de composants, écrits en C++, est d'une grande simplicité, ce qui offre des possibilités d'intégration de bibliothèques externes illimitées. Bien que distribué, RTMaps ne fonctionne que sur plate-forme PC et il n'est pas possible de générer du code *OS-less* à partir d'une application RTMaps. L'un de ses principaux défauts est son moteur asynchrone puisqu'il est impossible de faire fonctionner un modèle de simulation causal dans RTMaps à moins de l'encapsuler intégralement dans un composant. La reproductibilité comportementale n'est donc pas garantie. En revanche, les fonctionnalités d'enregistrement et de rejeu de données réelles de tous types ont fait le succès de RTMaps. L'utilisation de bases de données datées (DBB) simplifient le développement des applications et le débugge.

Hérité de l'expérience de RTMaps, AROS doit en reprendre les fonctionnalités qui ont fait son succès, en particulier les bases de données datées et la possibilité de rejeu. Pour autant, AROS devrait être en forte rupture technologique vis-à-vis de cette filiation historique.

D'autres outils existent bien sûr, que nous n'avons pas cités. L'histoire de chaque outil démarre bien souvent d'un besoin spécifique, d'un domaine d'applications particulier ou d'une idée originale pour être ensuite rendu compatible avec les

fonctionnalités des outils concurrents. Il est maintenant possible, par exemple, de faire cohabiter Matlab/Simulink et RTMaps au sein d'une même application. Les exigences du prototypage rapide sont variées et évoluent sans cesse. Un seul outil ne fera sans doute jamais tout. La place pour de nouveaux outils tels qu'AROS existe donc bel et bien. Pour profiter pleinement de la richesse que peuvent apporter l'ensemble de ces outils, il est impératif de prévoir, entre eux, des interfaces de communication. AROS devra lui aussi satisfaire cette exigence.

Les travaux qui font l'objet de ce manuscrit ont porté sur l'élaboration et la mise au point du **modèle d'exécution** ainsi que des **modèles structurel, de distribution et de reconfigurabilité** du projet AROS d'une part, et sur le **développement d'NJN**[1], la plateforme logicielle issue de cette réflexion. Ce manuscrit est organisé de la façon suivante :

La première partie expose le modèle d'exécution d'NJN, la plateforme logicielle du projet AROS. Elle est découpée en trois chapitres et s'attache à montrer que les principes sur lesquels repose l'exécution d'NJN offrent une réponse originale et pertinente au problème d'ordonnancement des applications temps-réel distribuées. Le chapitre 1 montre en quoi les approches traditionnelles d'ordonnancement temps-réel ne sont pas satisfaisantes dans notre contexte et ouvre la voie vers un mode d'ordonnancement évènementiel inspiré des techniques de simulation distribuée. Le chapitre 2 montre comment s'appuyer sur un moteur de simulation distribué pour construire un ordonnancement temps-réel à la fois flexible, déterministe et qui offre des garanties temps-réel acceptables dans notre contexte. Le chapitre 3 expose le fonctionnement du moteur d'exécution d'NJN .

La seconde partie porte sur le modèle structurel d'NJN, ses aspects dynamiques et distribués, et son exploitation dans la conception et le développement d'application. Le point de vue adopté dans cette partie se rapproche de celui de l'utilisateur d'NJN, c'est-à-dire le développeur d'applications. Le chapitre 4 énonce l'ensemble des objets structurels qui composent une application NJN. Il s'agit des composants, des tâches et des variables bien sûr, mais aussi d'objets plus complexes tels que les canaux ou les méthodes. Le chapitre suivant porte sur les mécanismes de création d'objets et plus particulièrement la reconfigurabilité dynamique et la distribution. Pour clore cette partie, une étude de cas est proposée dans le chapitre 6 et porte sur *CoreBot M,* le robot proposé par Mines ParisTech pour le concours CAROTTE [8].

La troisième et dernière partie est consacrée à l'implémentation d'NJN. Le premier chapitre de cette partie porte sur les bibliothèques logicielles qui offrent à NJN les services dont il a besoin en terme de gestion mémoire, d'abstraction du système d'exploitation et de gestion des communications. Le chapitre 8 entre dans le noyau d'NJN pour en expliquer les principes et les originalités d'implémentation. Le chapitre 9 montre comment les capacités structurelles d'NJN sont implémentées et comment sont gérées les bibliothèques de composants.

1 Prononcer « *engine* »... à la française !

Le dernier chapitre conclut ce manuscrit en rappelant les contributions importantes de ces travaux et en exposant les perspectives et développements futurs.

Première partie
Un modèle d'exécution temps-réel distribué déterministe

CHAPITRE 1
L'ORDONNANCEMENT ÉVÈNEMENTIEL TEMPS-RÉEL PEUT-IL ÊTRE DÉTERMINISTE ?

Pour Kopetz [43], les systèmes temps-réel distribués sont répartis en deux mondes selon que leur ordonnancement est de type temporel ou de type évènementiel.

L'ordonnancement temporel (*Time Triggered*), [44], [9], consiste à partager statiquement le temps disponible entre les tâches allouées à une même cible matérielle en tenant compte des communications nécessaires entre les différentes cibles. Dans ce type d'ordonnancement, toutes les tâches sont ordonnancées de façon périodiques selon une horloge globale à toutes les cibles du système. Tout (tâches, communications, utilisation des ressources, etc.) est alors planifié de façon statique sur une échelle de temps globale et cyclique, si bien que, moyennant une allocation suffisante de temps à chaque tâche et à chaque échange d'information, le comportement du système est parfaitement prévisible et ses délais de réaction sont totalement maitrisés. Le système peut donc être qualifié de sûr. La difficulté principale de ce type d'ordonnancement réside justement dans la détermination du temps à allouer à chaque tâche. Pour garantir le fonctionnement du système dans tous les cas, un majorant du temps d'exécution dans le pire cas de figure (*Worst Case Execution Time*) doit être connu pour chaque tâche et chaque échange de données. Ce temps est particulièrement difficile à déterminer à cause, d'une part, des architectures matérielles actuelles compte-tenu des caches mémoire, des pipelines et de l'architecture des réseaux de communications [90], [67], [5] et, d'autre part, de la complexité des algorithmes mis en oeuvre. Par ailleurs, l'ajout d'une tâche suppose de revoir la synchronisation globale du

système. Rappelons que dans notre contexte, toute étude statique de l'ensemble de l'application est proscrite du fait du caractère dynamique de l'architecture des applications que l'on vise. Ce point exclut, de fait, un ordonnancement purement temporel.

L'ordonnancement évènementiel (*Event Triggered*) consiste à associer le déclenchement de l'activité des tâches à des évènements internes (sémaphore, message, etc. [22]) ou externes (interruption) au système. Pour permettre aux actions les plus critiques d'être effectuées sans délai, l'ordonnancement mis en œuvre est dynamique. La politique adoptée repose en général sur l'attribution d'une priorité constante à chaque tâche [52]. Lorsqu'une tâche est déclenchée par un évènement, elle est exécutée si sa priorité est plus importante que la tâche en cours d'exécution, cette dernière est donc préemptée. Dans le cas contraire, elle est mise en attente d'exécution jusqu'à ce qu'elle devienne la tâche active la plus prioritaire. Ajouter une tâche conduit à intercaler la priorité de cette nouvelle tâche dans la table des priorités de la cible. La contrepartie de cette relative simplicité[1] est qu'il est bien difficile de prévoir avec certitude le comportement de l'application [50], tant en termes de comportement que de respect des contraintes temps-réel, en particulier dans les cas de charges exceptionnelles du système. L'ordre d'exécution des tâches ainsi que les instants d'accès aux données ne sont pas prévisibles à priori compte-tenu des évènements externes, des préemptions possibles et des exclusions mutuelles. Il en résulte une forte indétermination sur la disponibilité temporelle exacte des différentes données du système. Autrement dit, la causalité de la production des données n'est pas assurée ce qui conduit au non-déterminisme comportemental global du système. Concernant le respect des contraintes temps-réel, les cas de charges exceptionnelles du système montrent de façon évidente les limites de ce type d'approche. A moins de prendre des marges colossales dans le dimensionnement des cibles matérielles, le dépassement de capacité est toujours une éventualité à envisager. La préoccupation de pouvoir prouver le respect des contraintes temps-réel a fait émerger une classe particulière de systèmes où toutes les tâches critiques sont de nature périodique. Il est alors possible de prouver que, sous certaines conditions, les contraintes temps réel sont garanties [73]. Ce type de système domine largement les applications de contrôle car il est à la fois souple et relativement sûr, bien qu'il ne soit pas déterministe.

Kopetz conclut que l'ordonnancement temporel sera donc préféré dans des contextes où la principale exigence est la sûreté de fonctionnement, c'est-à-dire le respect strict des contraintes temps-réel et le déterminisme comportemental, tandis que l'ordonnancement évènementiel trouvera naturellement sa place dans des systèmes moins critiques où l'on recherche plus d'agilité dans la conception et la maintenance du système.

La comparaison frontale que Kopetz effectue entre les deux approches laisse entrevoir l'intérêt de trouver le meilleur compromis entre souplesse et sûreté de fonctionnement. Il est entendu que le respect des contraintes temps-réel ne peut plus être prouvé en pratique dès lors que le déclenchement des tâches s'écarte d'un

1 Les choses ne sont jamais simples : bien des problèmes se posent dans un système à ordonnancement évènementiel comme l'inversion de priorité par exemple [15].

modèle périodique [73]. S'orienter vers un système réellement évènementiel, c'est-à-dire acceptant des tâches critiques non-périodiques, condamne de ce point de vue à l'expression de contraintes temps-réel tout au plus probabilistes. Le temps réel dur n'est véritablement accessible qu'aux systèmes périodiques, quelque soit la nature de l'ordonnancement. Pour autant, on peut se demander si le déterminisme comportemental, au sens des données produites par les traitements, ne reste pas néanmoins accessible. Lee [49] pense l'objectif tout a fait atteignable à condition de remettre en cause radicalement les fondements même de la discipline. Le problème sous-jacent est celui du respect de la causalité dans l'ordonnancement des tâches et la mise à jour des données. C'est donc plus un problème de principe d'ordonnancement que de nature (évènementielle ou temporelle) de l'activation des tâches.

Comme le rappelle Lee [49], le problème de la causalité est résolu depuis fort longtemps dans le domaine de la simulation à évènements discrets [16], [86], y compris dans des contextes distribués [11], [42]. Le moteur d'exécution de ce type de simulateurs repose sur la gestion de listes d'évènements qui conserve l'ordre partiel des exécutions de tâches imposé par les relations de cause à effet du système. La causalité native du moteur garantit la prévisibilité du comportement de l'application. Cependant, ce sont les aspects temps-réel qui posent ici problème. S'il existe des simulateurs interactifs capables de fournir des données parfaitement prévisibles en suivant grosso modo le rythme de l'horloge murale (WCT[1]) [79], [35], les exigences temporelles de ces systèmes se limitent bien souvent à celles de la perception humaine, ce qui est loin d'être suffisant dans notre cas.

La première partie de ce chapitre se penche sur l'origine du non déterminisme des exécutifs temps-réel évènementiels et analyse quelques solutions proposées dans la littérature. La deuxième partie est consacrée à la simulation à évènements discrets distribuée. La dernière partie analyse les solutions proposées dans la littérature pour synchroniser en temps-réel un moteur à évènements discrets. Enfin la conclusion tente d'ouvrir la voie vers un nouveau genre de système temps-réel distribué inspiré des techniques de simulation.

1 De l'origine du non-déterminisme

Les systèmes à ordonnancement évènementiel ne sont pas prévisibles. Ce point a été évoqué dans l'introduction de ce chapitre. Ce non-déterminisme trouve sa source dans le principe même de l'ordonnancement par priorité utilisé classiquement dans les systèmes temps-réel évènementiels. La première section de cette partie revient plus en détail sur ce point.

1.1 Où l'on analyse l'origine du non-déterminisme comportemental

Les systèmes temps-réel évènementiels, tels qu'ils sont classiquement implémentés, reposent sur un ordonnancement préemptif à priorités. Chaque tâche

1 *Wall Clock Time* dans le texte.

se voit attribuer une priorité d'exécution qui lui permet d'être exécutée préférentiellement aux tâches de niveau inférieur, éventuellement en interrompant l'exécution de celles-ci.

Raisonnons sur le cas simple à trois tâches présenté figure 6. La tâche *B* consomme des données produites par les tâches *A* et *C*. Elle est synchronisée sur la tâche *A* ; la relation entre *C* et *B* est supposée non bloquante. Les priorités respectives de ces trois tâches sont définies de la façon suivante : la tâche *A* est la plus prioritaire, vient ensuite la tâche *B*, puis la tâche *C*. Examinons le cas de figure où la tâche *C* est activée avant la tâche *A*. La durée d'activation de la tâche *C* n'étant pas constante[1], deux cas peuvent se présenter : dans le premier cas, la tâche *C* se termine avant l'activation de la tâche *A* ; dans le second cas, la tâche *C* n'est pas terminée lorsque la tâche *A* est activée, elle est donc préemptée par cette dernière, puis par la tâche *B*.

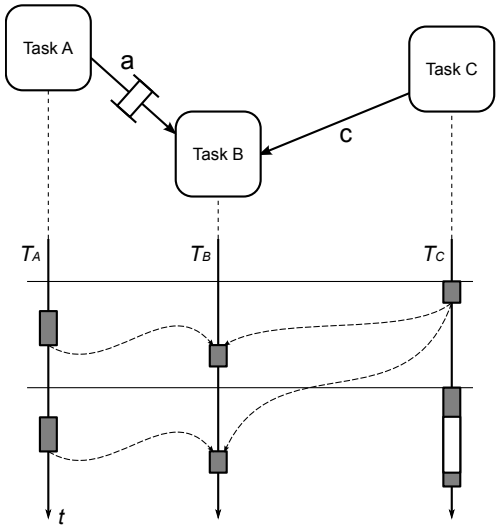


Figure 6. Trois tâches dans un système évènementiel ordonnancé par priorités.

La conséquence de cette indétermination d'exécution sur la tâche *B* est immédiate : dans le premier cas la tâche *B* exploitera les données issues de la dernière exécution de la tâche *C ;* dans le second cas la tâche *B* exploitera les données de l'occurrence d'exécution précédente.

Dans un ordonnancement temporel, il est prévu dès la conception une durée d'exécution pour chaque tâche. La figure 7 présente schématiquement ce mécanisme d'ordonnancement pour les trois tâches précédentes.

1 Compte tenu par exemple des préemptions possibles par d'autres tâches.

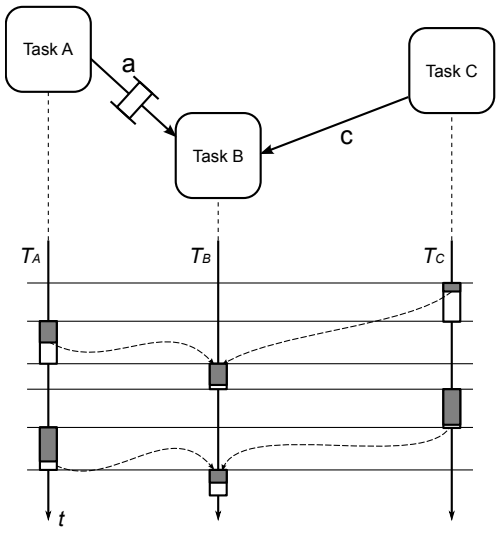


Figure 7. Trois tâches dans un système ordonnancé temporellement.

Quelque soit la durée effective d'exécution des tâches, tant qu'elle ne dépasse pas l'intervalle de temps alloué, la relation entre le consommateur et le producteur d'une donnée est toujours la même. L'indétermination rencontrée dans le cas précédent est complètement levée.

L'analyse de ses deux situations montre bien l'origine du non-déterminisme de l'ordonnancement par priorité. Il provient de l'association du mécanisme de préemption et de la variabilité de la durée d'exécution des tâches. Un idée simple pour lever le non-déterminisme serait donc de fixer la durée apparente d'exécution.

1.2 Où l'on fixe la durée apparente d'exécution des tâches – Le temps logique d'exécution

C'est ce que proposent Ghosal [33] et Liu [53]. Pour résoudre le problème du non-déterminisme comportemental, ils fixent la durée apparente d'exécution de chaque tâche du système depuis l'origine de son activation.

Ghosak appelle cette durée le temps logique d'exécution (LET[1]) tandis que Liu la nomme plus simplement échéance (*deadline*). Les deux concepts sont cependant très similaires.

Pour l'un comme pour l'autre, trois règles capitales régissent l'ordonnancement des tâches :

- Lorsqu'une tâche est activée par un évènement, les données qu'elle exploite sont immédiatement échantillonnées, même si la tâche n'est pas immédiatement exécutée. Le décompte de l'échéance ou du LET commence à cet instant.

1 *Logical Execution Time*

- Entre cet instant et l'expiration de l'échéance, la tâche doit être exécutée intégralement, quelque soit sa durée effective d'exécution et les préemptions qu'elle subit.
- A l'issue de l'échéance, et seulement à cet instant, les données produites par la tâche sont mises à disposition des autres tâches du système.

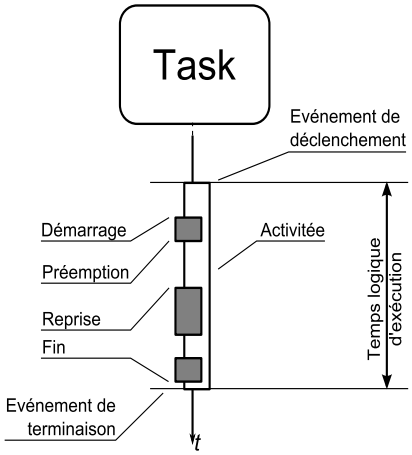


Figure 8. Le temps logique d'exécution (LET). Source [33].

Vu de l'extérieur, la tâche considérée semble donc avoir débuté à sa date d'activation et s'être achevée à l'issue de l'échéance.

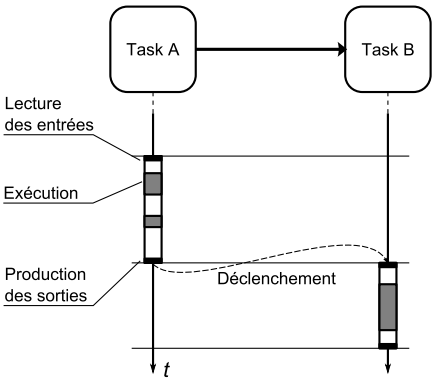


Figure 9. Ordonnancement à échéance. Source [53].

Ghosal met en œuvre des mécanismes d'échantillonnage des données[1] qui garantissent l'isolation du contexte d'exécution de la tâche pendant le LET.

1 On retrouve un peu ce mécanisme dans le *Simulink Real-Time Workshop* de *The MathWorks* [75]. L'échantillonneur bloqueur séparant deux blocs de périodicité différente permet à chaque bloc de travailler sur des données stables.

L'astuce de Liu consiste à n'autoriser l'accès aux données produites qu'une seule fois, à travers une file d'attente et seulement à échéance du producteur.

En attribuant à chaque tâche un temps logique d'exécution, Ghosal et Liu induisent implicitement des relations de cause à effet stables entre les données du système. L'échéance d'une tâche active la tâche suivante et lui délivre ses dernières données produites.

Pour que le système fonctionne, la priorité attribuée à chaque tâche doit être établie dynamiquement en fonction de la proximité de l'échéance. Plus l'échéance d'une tâche est proche, plus cette tâche devient prioritaire. La mise en œuvre de ce type d'ordonnancement ne permet pas de faire l'économie de la détermination du pire cas d'exécution. Les choses sont même un peu plus complexes puisque l'échéance attribuée à chaque tâche doit englober les préemptions éventuelles.

1.3 Le temps, la cause et la conséquence

Ce qu'il y a de capital dans les travaux de Ghosal [33] et Liu [53] c'est l'établissement de relations de dépendances stables entre les données, bien que le déclenchement des tâches soit de nature évènementielle. Ils parviennent à ce résultat en fixant la durée d'exécution de chaque tâche, un peu comme cela est fait avec un ordonnancement temporel.

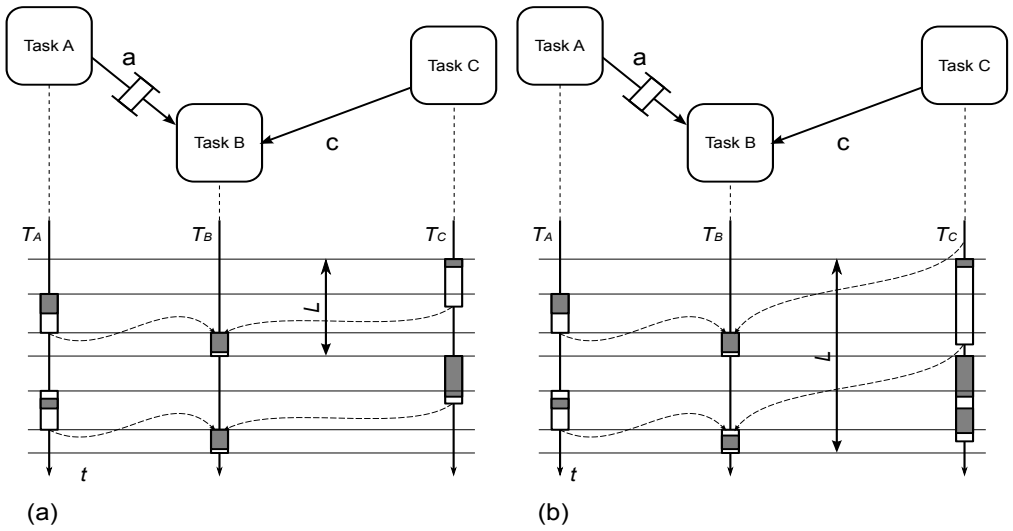


Figure 10. Ordonnancement de trois tâches au temps logique d'exécution.

Mais le comportement déterministe de l'application est obtenu de façon indirecte. Pour s'en convaincre, examinons le cas de la figure 10 qui reprend l'exemple étudié au paragraphe 1.1. Il est vrai que la dépendance entre les tâches *C* et *B* est établie définitivement. Dans le second cas présenté, par exemple, la tâche *B* exploiterait la donnée produite par la dernière activation de la tâche *C* arrivée à échéance, même si une activation plus récente s'est produite. Le comportement de l'exécution est donc parfaitement reproductible si les LET sont fixés.

Cependant, les relations causales dépendent de la durée du LET attribuée à chaque tâche et non d'un choix explicite de conception vis-à-vis des relations entre les données. Un changement de LET peut donc modifier les relations de dépendance entre les échantillons de données ce qui induit potentiellement de gros bouleversements sur la latence du système. Si on observe les deux cas présentés figure 10, il est facile de constater l'influence du LET de la tâche *C* sur la latence *L* du traitement dans son ensemble. Selon la durée du LET, la tâche *B* exploite l'échantillon produit par la dernière occurrence d'exécution de la tâche *C* ou bien celui de l'occurrence précédente. L'accroissement de la latence est remarquable.

Dans les contextes statiques où la granularité des tâches est assez grossière, cette influence du LET sur la latence du traitement peut ne pas se manifester du fait du nombre réduit de dépendance entre les tâches et de l'aspect totalement figé des choses. Mais il en est tout autrement dans notre contexte où la granularité des tâches est très fine et où la structure de l'application peut être modifiée dynamiquement. Il semble plus raisonnable de conserver la trace des relations causales entre les données au cours de l'exécution et cela de façon totalement indépendante de la durée d'exécution, logique ou réelle, des tâches.

Le but à atteindre est à présent clairement défini : élaborer un mécanisme d'ordonnancement qui conserve les relations causales entre les données dans un contexte d'exécution distribuée. Comme nous l'avons dit dans l'introduction de ce chapitre, ce type d'ordonnancement existe [29]. Les moteurs d'exécution utilisés dans les simulateurs distribués à évènements discrets conservent nativement les relations causales entre les données du système. Le type d'ordonnancement qu'ils mettent en œuvre exécute chaque tâche au plus tôt en fonction de l'ordre partiel imposé par les règles de causalité. La prochaine partie se penche sur le fonctionnement et les caractéristiques de ces simulateurs.

2 La causalité au cœur de l'exécution – La simulation distribuée à évènements discrets

Le principe de la simulation évènementielle n'est pas récent. Il trouve ses origines dans les années 60 [60] avec des langages tels que GPSS [37], SIMSCRIPT [55], SIMULA [16] ou GASP [65]. L'objectif est de pouvoir simuler de façon efficace des systèmes dont le comportement peut être modélisé par une fonction d'état discrète. Il a été démocratisé avec l'émergence des langages de description du matériel tels que VHDL principalement en Europe [58], *Verilog* et, plus récemment, *SystemVerilog* de l'autre côté de l'Atlantique [57].

La simulation distribuée reprend les mêmes principes, en y ajoutant des mécanismes de synchronisation, de manière à pouvoir répartir le travail au sein d'un réseau de processeurs. Elle répond d'abord à un besoin de puissance de calcul, en particulier dans le domaine autogène de la simulation des réseaux. Deux algorithmes ont marqués le champ disciplinaire de la simulation distribuée :

Chandy [11] d'abord, Jefferson [42] ensuite. Le domaine du jeu en réseau [10] et de la simulation interactive [28] profiteront des avancées dans le domaine et imposeront l'idée d'une simulation synchronisée avec le temps réel.

La première section de cette partie pose les bases de la simulation évènementielle. Elle montre qu'elle correspond à un ordonnancement purement évènementiel et qu'elle conduit à un comportement parfaitement déterministe, indépendant de la durée d'exécution des tâches. Les deux sections suivantes présentent respectivement les algorithmes de Chandy et de Jefferson qui étendent ce type de simulation à des contextes d'exécution distribués.

2.1 Parallélisme et causalité – Principe de la simulation à évènements discrets

Formellement, dans un simulateur évènementiel, le système est modélisé par des signaux et des tâches[1]. Les signaux renferment les données du système tandis que les tâches encapsulent les algorithmes qui en modélisent le comportement. On associe à chaque tâche une liste dite de sensibilité qui est constituée de l'ensemble des signaux que la tâche doit surveiller. Lorsqu'un évènement[2], tel qu'un changement d'état, se produit sur l'un de ces signaux, la tâche est activée en vue de son exécution. Toutes les tâches activées à une date donnée sont ensuite exécutées au cours d'un cycle de simulation. On reproduit ainsi une apparente simultanéité d'exécution. Chaque cycle produit à son tour de nouveaux évènements qui modifient l'état des signaux du système, évènements qui activeront à leur tour de nouvelles tâches, donnant naissance à un nouveau cycle, et ainsi de suite.

L'action des tâches sur les signaux est régie par deux postulats simples : seul le passé peut être consulté ; seul l'avenir peut être modifié. Chaque opération effectuée au cour d'un cycle s'inscrit sur une ligne temporelle globale où l'on distingue clairement ce qui est passé de ce qui constitue l'avenir [59]. Au cours d'un cycle, toutes les tâches actives du système puisent leurs données dans un état stable et connu appelé « état présent », et inscrivent leur production dans un nouvel état, appelé « état futur »[3]. Ce nouvel état ne sera accessible en lecture qu'au prochain cycle, lorsque le temps propre du simulateur aura atteint la date du dit état, conférant à celui-ci le statut d'état présent. Conformément aux deux postulats énoncés plus haut, d'une part, l'état présent du système, ainsi que tous les états antérieurs, ne peuvent pas être modifiés et, d'autre part, il est impossible d'avoir la moindre information sur l'état futur du système ainsi que sur les

1 Dans la littérature, on parle plutôt de processus. Nous préférons réserver le mot processus pour désigner ceux du système d'exploitation.

2 La notion d'évènement varie un peu selon les auteurs. Pour certains, il s'agit de l'exécution d'une tâche, pour d'autres, il s'agit des changements d'état opérés sur les signaux. Cela n'a guère d'importance en définitive : le changement d'état d'un signal à une certaine date est le résultat de l'exécution d'une tâche au même instant ; les deux « évènements » existent et sont irrémédiablement liés l'un à l'autre.

3 A moins que soit explicitement spécifié un délai lors de l'affectation, auquel cas l'état affecté se situe plus loin sur la ligne temporelle.

éventuels états postérieurs à celui-ci. Ces deux derniers points garantissent la causalité du système.

En pratique, l'ordonnancement des tâches est non préemptif. Au cours d'un cycle, les tâches actives sont donc exécutées les unes après les autres. La distinction nette entre l'état présent et l'état futur permet alors de traduire l'apparente simultanéité d'exécution des tâches. Sous certaines conditions[1], elle rend en effet le résultat produit par un cycle indépendant de l'ordre effectif d'ordonnancement des tâches, ce qui garantit le déterminisme comportemental du système. Ce résultat se comprend aisément. Les données produites par un cycle de simulation dépendent uniquement des algorithmes exécutés et des données qu'ils exploitent. Or, les données exploitées par les tâches actives au cours d'un cycle proviennent exclusivement de l'état présent du système qui, par définition, n'est pas modifiable dans le cycle courant.

Le module ci-dessous, écrit en *SystemVerilog*, illustre ce point :

```
module exemple1;
    logic a = 1, b;
    always begin    // Une tache
        a <= !a;
        b <= a;
        #5;
   end
endmodule
```

A la lecture du code, on est tenté de croire que les signaux *a* et *b* prennent les mêmes valeurs aux mêmes instants. Il n'en est rien. La tâche désignée par le mot clé `always` détermine les états futurs de *a* et de *b* en exploitant la valeur de *a* relative à l'état présent du système. Les signaux *a* et *b* sont donc compléments l'un de l'autre et l'ordre des affectations n'a aucune influence sur ce résultat.

Le même comportement peut être obtenu avec deux tâches simultanées :

```
module exemple2;
    logic a = 1, b;
    always #5 a <= !a;   // Une première tache
    always #5 b <= a;    // Une seconde tache
endmodule
```

Là encore, les affectations de *a* et de *b* sont opérées à partir du même état présent. Là encore, l'ordre dans lequel sont opérées les affectations n'a aucune importance.

Pour obtenir le comportement souhaité, l'activation de la seconde tâche doit être la conséquence d'un évènement affectant le signal *a*. C'est ce que traduit, dans ce dernier exemple, la liste de sensibilité `@(a)` associée à la seconde tâche :

1 La condition nécessaire et suffisante est que toutes les modifications apportées à un signal au cours d'un cycle doivent avoir pour résultat la même valeur, quelque soit l'ordre dans lequel sont effectuées les affectations. Pour s'en assurer, VHDL et *SystemVerilog* adoptent deux stratégies selon les cas : soit ils interdisent à un signal déjà piloté par une tâche d'être modifié par une autre ; soit ils utilisent des fonctions de résolution qui arbitrent les conflits engendrés par les affectations multiples. Voir [58] à ce sujet.

```
module exemple3;
    logic a = 1, b;
    always #5 a <= !a;
    always @(a) b <= a;
endmodule
```

On illustre ici clairement le rôle joué par la liste de sensibilité dans l'établissement des relations causales entre les tâches. Par rapport à l'exemple précédent, l'exécution de la seconde tâche sera effectuée un cycle de simulation plus tard, après que le signal *a* a été mis à jour.

Le temps d'un simulateur évènementiel ne partage pas beaucoup plus avec le sens commun que son irréversible évolution croissante. Il est de nature discrète et est en général représenté par un entier positif. Il permet d'établir un ordre total des états du système. Il s'agit donc plus de compter les cycles que de se raccrocher à une évolution temporelle réaliste. Afin de gérer convenablement les problèmes de relaxation des données engendrés par les interactions entre tâches, le temps est parfois défini selon plusieurs dimensions interdépendantes (en général deux, quelques fois trois). La dimension la plus fine est nommée *delta* (*Δ*). La dimension la plus grande est plus simplement appelée date.

```
Tant qu'il y a des tâches actives
    T <- date de la prochaine tâche active
    Δ <- 0
    Tant qu'il y a des tâches actives à T
        Tant qu'il y a des tâches actives à (T, Δ)
            Exécuter une tâche active
        Fin tant que
        Δ <- Δ + 1
    Fin tant que
Fin tant que
```

La convergence à une date donnée est assurée par une boucle pour laquelle chaque itération correspond à un cycle *Δ* de simulation. Lorsque la relaxation est enfin atteinte, la date est avancée pour correspondre à l'état futur le plus proche et les cycles *Δ* reprennent leur course à partir de zéro.

Le type d'ordonnancement adopté repose finalement sur le maintien d'une liste des évènements à traiter dans l'ordre croissant de leur datation. A chaque itération de l'ordonnanceur, l'évènement en tête de cette liste en est retiré pour être traité. Le temps propre du simulateur correspond à la date de cet évènement, sans aucun rapport, ni avec l'évolution de l'horloge murale, ni avec le temps effectif mis par les tâches pour être exécutées.

2.2 La simulation distribuée conservative

Cet algorithme général pose d'énormes difficultés dans un contexte distribué. La répartition du système sur plusieurs cibles matérielles[1] conduit à fractionner l'état interne entre les différentes cibles en fonction de la localité des signaux. Compte tenu des latences réseau et de l'absence d'horloge absolue, il est difficile de garantir qu'un état localement présent – et donc accessible – ne soit pas un état

1 Ou dans plusieurs processus systèmes.

futur – et donc modifiable – sur l'une ou l'autre des cibles. Cette division compromet donc grandement l'intégrité de l'état et le déterminisme temporel du système. En terme d'évènements on peut formuler le problème ainsi : Comment s'assurer, lorsqu'on exécute une tâche, que tous les évènements antérieurs dont elle dépend ont bien été traités ?

K. Mani Chandy et Jayadev Misra proposent une solution à ce problème à la fin des années 70 [11] avec ce qu'on appellera par la suite la méthode conservative [29].

Ils formalisent le problème comme suit. Le système est constitué d'un ensemble de processus logiques[1] qui communiquent par l'intermédiaire de messages à travers un réseau de communication ou des mémoires partagées. Chaque processus logique du système traite une liste d'évènements ordonnés temporellement. Les évènements sont produits de façon interne ou résultent de la réception de messages en provenance des autres processus logiques du système. Un évènement ne peut alors être traité que si l'on est certain qu'aucun message antérieur à cet évènement ne sera reçu après l'exécution. Sans cette contrainte, un tel message provoquera à coup sûr une violation de causalités.

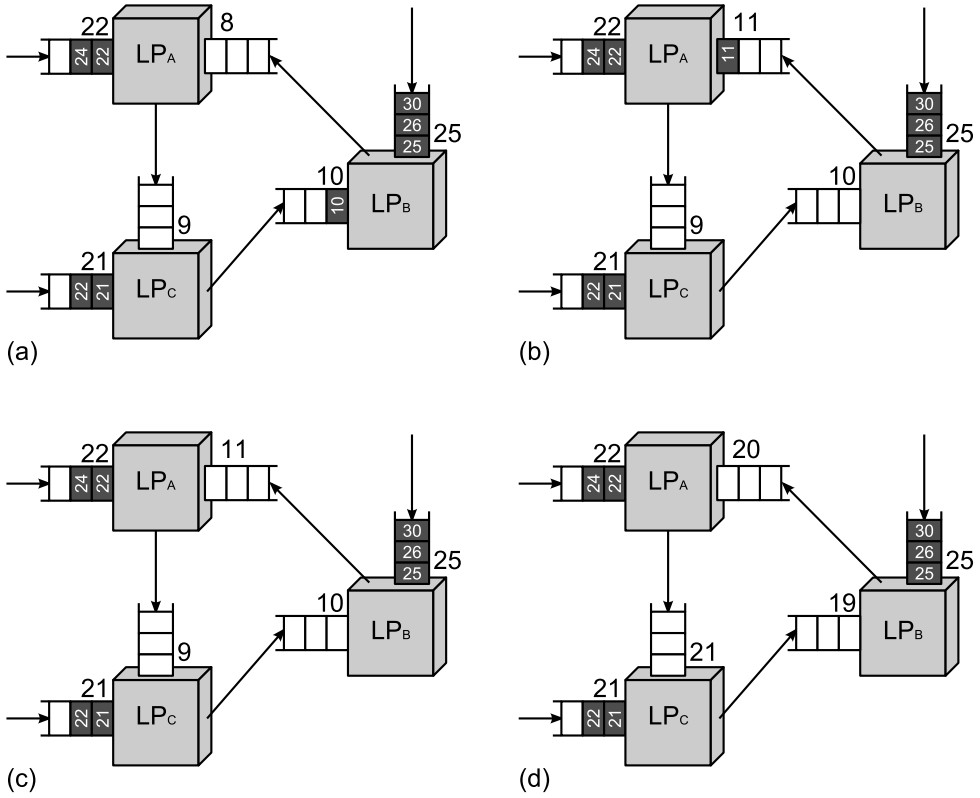


Figure 11. La méthode conservative de Chandy et Misra.

1 Désigne ici un processus du système d'exploitation ou le processus unique d'une cible matérielle.

La solution de Chandy et Misra repose sur l'horodatage des messages et la connaissance de la topologie locale du système. Chaque processus doit en effet identifier l'ensemble des sources à l'origine des messages qu'il reçoit.

Chaque canal de communication entre processus logiques est matérialisé par une file d'attente côté consommateur où viennent s'empiler, dans l'ordre de leur date, les messages non-encore consommés (figure 11.a). Le canal se voit attribuer la date de son message de tête qui est par conséquent le plus ancien. À défaut de message, il conserve la date du dernier message qui a été émis.

À chaque cycle de simulation, le processus consomme le message de tête du canal le plus ancien (figure 11.a : pour le processus *B*, il s'agit du message de date 10*)*. L'exécution de l'évènement correspondant produit éventuellement de nouveaux messages dont la date est nécessairement postérieure au message initial (figure 11.b). Si le canal le plus ancien est vide, le processus est bloqué jusqu'à réception d'un nouveau message (figure 11.c, cas du processus *C*).

Ce mécanisme prémunit totalement contre les violations de causalité. Un processus ne peut en effet progresser que si ses sources sont temporellement en avance par rapport à lui. La fâcheuse contrepartie est qu'il suffit d'une dépendance circulaire entre les processus pour bloquer le système dans son ensemble, ce que montre la figure 11.c.

Chandy et Misra proposent une première solution à ce problème par l'intermédiaire de messages de synchronisation appelés « messages nuls ». Leur rôle est d'offrir à chaque processus impliqué dans une dépendance circulaire les garanties dont il a besoin pour se sortir de l'impasse temporelle dans laquelle il se trouve. Un processus qui envoie un tel message à une certaine date prend en effet l'engagement de ne jamais plus envoyer de message antérieur cette date. De messages nuls en messages nuls, les processus bloqués peuvent ainsi progresser (figure 11.d) jusqu'à retrouver un message non nul à traiter.

Le trafic dû aux messages nuls s'avère relativement important pour un bénéfice somme toute assez relatif. Chandy et Misra proposent donc, quelques années plus tard, une seconde solution beaucoup plus élégante [12]. Ils observent en effet que, parmi tous les messages coincés dans les processus d'une dépendance circulaire, il y en a au moins un qui peut être consommé sans crainte : le plus ancien (le message de date 21 sur la figure 11.c). Le problème est alors de repérer les cycles funestes et d'identifier les messages salutaires, problème sur lequel se focalisera la communauté scientifique pendant de nombreuses années.

L'étude topologique nécessaire pour identifier les dépendances circulaires rend délicat l'usage du protocole de Chandy et Misra dans notre contexte. En effet, cela vient en contradiction avec les caractéristiques architecturales dynamiques de notre modèle.

2.3 L'optimisme de Jefferson et la distorsion du temps

David R. Jefferson propose au milieu des années 80 une nouvelle solution qui prend le contre-pieds de la méthode de Chandy et Misra. Dans son article intitulé

Virtual time [42], il présente un algorithme appelé « distorsion du temps »[1] qui marquera une rupture radicale dans les recherches sur la simulation évènementielle distribuée.

Jefferson parie sur le fait qu'il est préférable d'avancer en semant des cailloux plutôt que d'attendre un évènement improbable. L'algorithme sera qualifié à juste titre de méthode optimiste [29]. Toutes les contraintes de synchronisation temporelle sont relâchées afin d'optimiser l'utilisation des ressources de calcul. Les processus n'attendent plus de messages nuls pour exécuter un évènement. Ils prennent le risque d'anticiper son exécution. Dans le pire cas de figure, il faudra exécuter l'évènement à nouveau. Dans les autres cas, le temps d'exécution aura été gagné sur des plages temporelles où Chandy et Misra attendent les messages de synchronisation.

L'originalité du cadre fixé par Jefferson est qu'il repose sur une définition locale et flexible du temps. Chaque processus logique dispose de sa propre horloge virtuelle, indépendante de celle des autres processus logiques. Localement, la course du temps peut être déformée, voire inversée. L'évolution des horloges n'est donc pas nécessairement croissante. Jefferson s'appuie sur les conditions d'horloges définies par Lamport [46] pour donner à son cadre temporel déformable les bonnes propriétés de causalité : d'une part, la date d'émission d'un message est toujours strictement inférieure à sa date de réception ; d'autre part, un évènement[2] a toujours une date strictement inférieure à l'évènement qui le suit au sein du même processus[3]. La progression de l'information va donc toujours dans le sens croissant du temps, même lorsque ce dernier fait marche arrière.

D'un point de vue pratique, les processus traitent toujours les messages reçus, sans attendre d'avoir la certitude qu'aucun autre message ne mettra en péril la causalité du système. Si une telle chose arrive, la violation de causalité est détectée par comparaison entre la date du message et l'horloge locale. Les processus mettent alors en œuvre un algorithme de retour arrière[4] pour retrouver un état satisfaisant.

Voyons cela un peu plus en détail en examinant les quatre situations de la figure 12. Chaque processus est associé à une pile de messages d'entrée, une pile de messages de sortie et une pile d'états. A chaque évènement, c'est-à-dire à chaque fois que le traitement associé au processus est exécuté, l'état présent du processus est empilé dans la pile d'états (processus *A*, figure 12.a). Si cet évènement nécessite l'envoi de messages à d'autres processus, l'opération est effectuée sur le champ. Une copie « négative » de chaque message, appelée anti-message, est conservée dans la pile de sortie.

Lorsqu'un message est réceptionné, il est stocké dans la pile d'entrée et classé en fonction de sa date. Deux cas de figures peuvent alors se présenter :

1 *Time Warp* dans le texte.
2 Au sens de l'exécution d'une tâche.
3 Dans le contexte temporel posé par Jefferson, la seconde condition d'horloge de Lamport engendre le mécanisme de retour arrière (*rollback*) dont on parlera plus loin.
4 *Rollback* dans le texte.

- Si la date du message est postérieure à l'horloge locale, il sera traité lorsque cette horloge aura atteint la bonne valeur. Notons dès à présent qu'une fois traité le message ne sera pas pour autant retiré de la pile d'entrée.
- Si la date du message est antérieure à l'horloge locale, il y a violation de causalité et le processus tente de retrouver un état cohérent en mettant en œuvre l'algorithme de retour arrière.

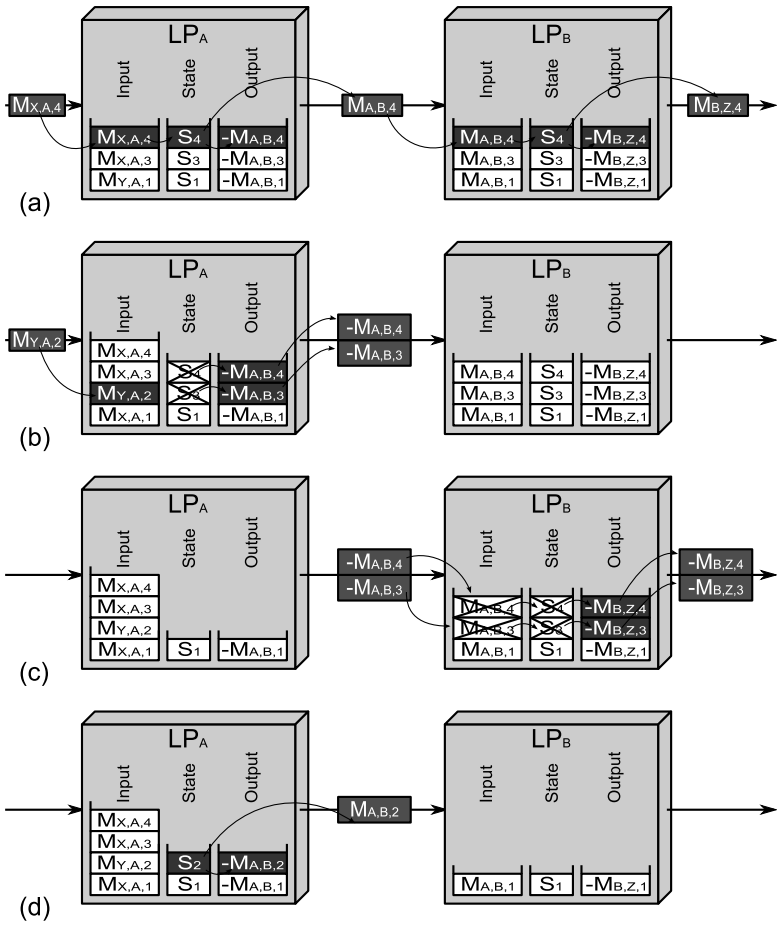


Figure 12. Mécanisme de retour arrière de l'algorithme de Jefferson.

Dans cette dernière situation, la pile d'état du processus est dépilée jusqu'à retrouver un état antérieur à la date du message nouvellement reçu (processus *A*, figure 12.b). Les anti-messages correspondant à des messages obsolètes sont envoyés à leurs destinataires et supprimés de la pile de sortie. Le processus retrouve alors un état qui lui permet de traiter le nouveau message et les suivants dans des conditions causales (processus *A*, figure 12.c).

La réception d'un anti-message (processus *B*, figure 12.c) provoque grosso modo le même effet que la réception de son homologue « positif », y compris le retour arrière si nécessaire, à ceci près que :

- il ne provoque pas d'exécution du processus cible ;
- la présence d'un anti-message et de son message associé dans la même file d'entrée provoque leur annihilation mutuelle.

Message et anti-message disparaissent donc de la file d'entrée après avoir remis le processus dans l'état adéquate.

D'un point de vue plus global, le phénomène se propage de processus en processus, provoquant des retours en arrière en cascade, jusqu'à ce que l'ensemble du système ait retrouvé un état satisfaisant. Chaque processus reprend ensuite le traitement des messages de sa pile d'entrée dans le bon ordre (figure 12.d).

Bien que rapidement adopté par la communauté scientifique, l'algorithme de Jefferson posera quelques épineux problèmes, en particulier concernant la gestion de la mémoire et le coût temporel des retours arrière. Les deux problèmes sont en fait intimement liés.

Une première difficulté est de savoir à quel moment les différentes piles d'un processus peuvent être nettoyées de leur contenu fossile, c'est-à-dire des objets dont aucun retour arrière n'aura plus jamais besoin. Jefferson définit pour cela un temps appelé temps virtuel global (GVT) qui marque la limite des objets fossiles pour l'ensemble des processus du système. Ce temps correspond à l'évènement non traité le plus ancien de l'ensemble du système. Tous les objets (messages, états) antérieurs au GVT peuvent être supprimés du système. Cependant, la littérature abondante sur le sujet témoigne de la difficulté qu'il y a à déterminer ce temps de manière efficace [18], [36], [81], [31], [24], [4], [14]…

Afin d'optimiser la gestion de la mémoire, certains auteurs proposent de ne plus systématiquement empiler l'intégralité de l'état du processus à chaque évènement [63]. Deux techniques sont proposées. La première consiste à réduire la fréquence des empilements d'états. On se contente de sauvegarder un état tous les *n* évènements. En cas de retour arrière, le processus revient au dernier étant valide de la pile et doit reconstruire l'historique manquant [51], [27], [69]. La seconde idée consiste à mettre en œuvre des sauvegardes incrémentales de l'état [3]. Ces techniques permettent par ailleurs d'améliorer les performances en fonctionnement normal en réduisant le surcoût temporel lié à la sauvegarde de l'état.

Le coût temporel des retours arrière peut s'avérer extrêmement pénalisant dans certains cas pathologiques où les processus alternent de façon hystérique avances considérables et retours en arrière brutaux [54], si bien que les recherches se concentrent sur l'idée de les rendre aussi exceptionnels que possible.

Certaines recherches visent à rendre les retours arrière plus rapides. Par exemple, la technique connue sous le nom d'annulation directe, [1], [30] et [92], consiste à matérialiser par des liens directs[1] les relations de cause à effet entre évènements. La suppression d'un évènement entraine alors la suppression des évènements qu'il a engendrés, sans nécessiter l'envoi d'anti-message. L'avantage de cette technique est qu'elle permet de ne supprimer que les évènements effectivement altérés par la

1 Par exemple par des pointeurs sur les architectures multiprocesseurs à mémoire partagée.

violation de causalité sans qu'il soit nécessaire de remettre en cause ni les évènements simultanés ni les évènements postérieurs qui sont sans relation avec l'évènement annulé.

Les auteurs tentent aussi d'éviter les retours arrière sans effet, en retardant le dépilage de la pile d'état [89] ou en différant l'envoi des anti-messages [32]. L'objectif est d'éviter de provoquer une réaction en chaine tant que l'on est pas certain que le retour arrière a bien modifié le résultat des traitements. Ces techniques sont respectivement connues sous les noms de réévaluation et annulation paresseuse. Elles induisent un surcout qui ne les rend pas intéressantes dans tous les contextes d'exécution. Dans le même ordre d'idées, d'autres auteurs tentent de corriger l'état de la pile sans revenir en arrière [13].

D'autres proposent encore de limiter l'avance prise par chaque processus logique vis-à-vis de l'ensemble du système en lui associant une fenêtre temporelle qui limite sa visibilité sur les évènements futurs [76].

A l'extrême, on tente de supprimer les retours arrière. Deux approches originales sont à relever. Dickens [21] propose d'abord de restreindre l'usage des retours arrière aux évènements locaux. Les messages envoyés entre processeurs sont retardés de manière à garantir leur pérennité. Les anti-messages ne sont donc plus nécessaires [78], [35]. Ensuite Ferschal [25] et Som [77] proposent de réserver les retours arrière aux situations statistiquement exceptionnelles. Leur solution est basée sur l'analyse statistique des caractéristiques temporelles des messages échangés entre processus. L'avance que peut prendre un processus est alors déterminée de manière à accepter un risque raisonnable de retour arrière.

Les algorithmes de Chandy et de Jefferson font encore aujourd'hui l'objet de nombreuses optimisations [64] – anticipation [13], exécution inverse [7], [85], clonage [40], tolérance aux fautes [17], [23], etc. – quelques fois relatives à des domaines d'application très spécifiques. Le cout de ces optimisations font qu'il n'est pas toujours intéressant de les mettre en œuvre. Suivant l'application, certaines seront plus efficaces que d'autres. La frontière entre les méthodes optimistes et concervatives n'est par ailleurs plus si nette. Nombreux compromis existent entre l'approche purement optimiste et l'approche strictement conservative [68]. Est-il seulement envisageable d'exploiter les mécanismes d'ordonnancement qu'ils mettent en œuvre dans des contextes temps-réel raisonnablement dur ? Cette question fait l'objet de la prochaine partie de ce chapitre.

3 De la simulation à l'exécution temps-réel

Une simulation est dite temps-réel si l'horloge du simulateur et l'horloge murale (WCT) avancent de concert, avec un rapport d'avancement, un STAR[1], constant et égal à l'unité [28]. Pour atteindre ce ratio unité, il faut bien évidemment que les performances du simulateur lui permettent d'avancer à une cadence supérieure au WCT. Car, comme le fait remarquer Steinman [79], on peut ralentir un simulateur, il est en revanche impossible de le faire avancer au-delà de ses capacités.

Deux domaines ont fait émerger des simulateurs synchronisés en temps-réel : celui des modèles hybrides introduits par Bagrodial [2] où interagissent des composants réels et des composants simulés et celui de la simulation interactive qui fait intervenir par exemple un opérateur humain. Les contraintes temps-réel de ces systèmes sont en général molles puisque mises en œuvre dans des contextes de test, de vérification ou d'entrainement. Ce sont donc avant tout des simulateurs. Ils n'ont pas pour vocation de contrôler des processus temps-réel dur, encore moins dans des systèmes en exploitation.

Au moins une exception existe cependant, celle proposée par Zhao [93] en 2006. Elle est basée sur un protocole de synchronisation conservatif. L'ordonnançabilité de la simulation conservative est en effet simple à démontrer à condition, bien évidemment, que les évènements d'entrée soient de nature périodique.

Les protocoles optimistes quant à eux se heurtent à une incompatibilité structurelle avec le temps-réel dur, liée au cout des retours arrière. Certains auteurs ont toutefois cherché à en démontrer l'ordonnançabilité moyennant des contraintes fortes sur les caractéristiques de l'application et du protocole.

La première section de cette partie est consacrée aux travaux de Zhao et son modèle d'exécution conservatif temps-réel. La seconde section portera sur l'ordonnançabilité des protocoles optimistes. La dernière section montrera les techniques qui permettent de synchroniser un simulateur interactif optimiste avec l'horloge murale, indépendamment de toute considération de preuve.

3.1 Un modèle d'exécution distribuée conservatif

Zhao [94] parvient à construire un modèle d'exécution temps-réel en s'appuyant sur deux éléments fondamentaux : une synchronisation temps-réel localisée en certains points de l'application et un relâchement des contraintes temporelles le long des chemins de données. Le modèle d'exécution qu'il propose nécessite une analyse topologique des relations temporelles reliant les tâches de l'application. Cette analyse permet de choisir localement l'ordre d'exécution des évènements au sein de chaque processus logique de la distribution.

De façon un peu simplifiée, les choses se formalisent ainsi :

Une application est définie par un ensemble de composants/tâches, munis de ports d'entrées/sorties, et interconnectés entre eux par des signaux. Chaque signal

1 *Simulation Time Advancement Rate*.

transfère les évènements issus d'un port de sortie vers un ou plusieurs ports d'entrées. L'application est distribuée sur plusieurs processus logiques.

La figure 13 donne un exemple d'application comportant 6 composants notés *A*, *B*, *C*, *D*, *E* et *F* distribués sur deux processus logiques notés LP_1 et LP_2. Chaque composant héberge une tâche.

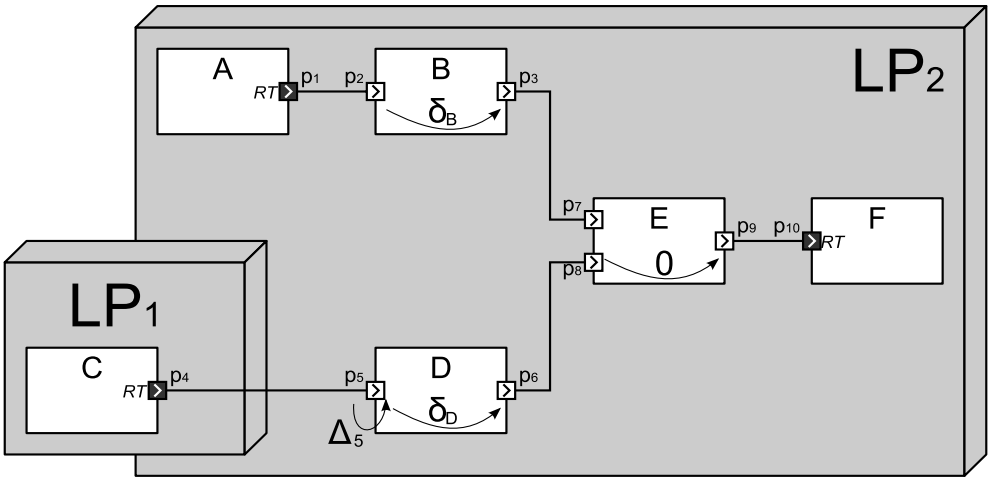


Figure 13. Une application de 6 composants distribués sur deux processus logiques.

Un composant est caractérisé par un délai purement virtuel, éventuellement nul, correspondant au temps minimum qu'il faut à l'information pour le traverser. Ce temps ne correspond à aucune réalité physique. Il a pour rôle de relâcher les contraintes temporelles de l'application en donnant à chaque tâche le temps dont elle a besoin pour être traitée. Par exemple, le composant *B* de l'application de la figure 13 est caractérisé par un délai δ_B ; si l'évènement $e_2 = (v_2, t_2)$ est délivré sur le port p_2 et produit, après traitement par le composant *B*, l'évènement $e_3 = (v_3, t_3)$ sur le port p_3, alors les dates respectives de e_2 et e_3 sont nécessairement telles que $t_3 \geq t_2+\delta_B$. Ces délais caractéristiques permettent d'établir la « longueur » du chemin reliant deux ports situés sur un même processus logique, en fonction des composants traversés. Par exemple, le chemin reliant le port p_1 au port p_{10} a pour longueur : $d_{1,10} = \delta_B+\delta_E = \delta_B$. Grosso modo, cette longueur correspond au délai dont dispose le système pour exécuter toutes les tâches situées sur le chemin considéré et absorber les latences réseaux.

Cette mesure de longueur permet de construire une relation d'ordre partielle entre les évènements du système. Cette relation d'ordre traduit le fait qu'un évènement situé sur un port de l'application peut être traité dès lors que les chemins de données reliés à ce port n'apporteront plus d'évènement plus ancien. Formellement, soit $e_i = (v_i, t_i)$ et $e_j = (v_j, t_j)$ deux évènements associés respectivement aux ports p_i et p_j. Alors $e_i < e_j$ si et seulement si $t_i+d_{i,j} < t_j$. A un instant donné, les évènements non encore traités par le processus logique sont classés dans une liste du plus « petit » au plus « grand ». Le processus traite alors les évènements en commençant par les plus « petits » de la liste.

La latence de communication entre processus logiques est modélisée par un type de port d'entrée particulier dit « port réseaux » auquel est associé un délai. Ce délai majore le retard avec lequel les évènements sont délivrés, par rapport à la date qui leur est associée. Par exemple, la latence de communication entre les processus LP_1 et LP_2 de la figure 13 est modélisée par le port réseau p_5. Celui-ci est caractérisé par le délai de délivrance Δ_5 qui garantit que tout évènement $e_5 = (v_5, t_5)$ destiné à ce port sera délivré avant que l'horloge murale n'indique la date $t_5+\Delta_5$.

Afin de garantir la causalité du traitement des évènements, chaque processus logique doit s'assurer que l'évènement qu'il traite ne pourra être remis en cause par un évènement associé à un port réseau. Prenons le cas concret d'un évènement $e_5 = (v_5, t_5)$ associé au port p_5 de la figure 13. Cet évènement ne pourra être traité par le processus logique LP_2 que si l'horloge murale a dépassé la date $T = t_5 + \Delta_5$. De même, un évènement $e_9 = (v_9, t_9)$ associé au port p_9 ne pourra être traité que si l'horloge murale a dépassé la date $T = t_9 – d_{5,9} + \Delta_5$. On s'assure ainsi que, compte tenu de la latence réseau, le port p_5 ne délivra pas d'évènement susceptible de remettre en cause le traitement de l'évènement e_9. Si tel n'est pas le cas, le processus LP_2 s'interrompt en attendant que les conditions temporelles de traitement de e_9 soient remplies.

Les capteurs et actionneurs sont modélisés par des composants disposant de port d'entrées/sorties dits « temps-réel ». De tels ports établissent une relation temporelle entre la date des évènements qu'ils traitent et l'horloge murale. C'est par leur intermédiaire que l'application se synchronise en temps-réel. Par exemple, le composant F représente un actionneur et dispose d'un port temps-réel d'entrée nommé p_{10}. Si $e_{10} = (v_{10}, t_{10})$ est un évènement traité par le port p_{10} alors cet évènement doit être délivré avant que l'horloge murale n'atteigne la date t_{10}. De cette manière, l'action engendrée sur l'actionneur pourra être effectuée à temps.

De façon plus globale, l'application est ordonnançable à condition que la contrainte temporelle associée à chaque port temps-réel soit respectée.

Le cadre élaboré par Zhao conduit à un ordonnancement temps-réel évènementiel parfaitement reproductible puisqu'il est basé sur les relations causales entre les évènements. De ce point de vue, il est pleinement satisfaisant. Cependant, il nécessite une analyse statique de la topologie de l'application extrêmement poussée. Notre cadre de travail nous interdit cette analyse. Contrairement à Zhao, il nous faut donc nous orienter vers les protocoles optimistes qui sont capables de produire un ordonnancement causal en toute circonstance. Reste à évaluer leur capacités exactes du point de vue des contraintes temps-réel.

3.2 Les protocoles optimistes sont-ils ordonnançables ?

Afin de montrer qu'un protocole optimiste est ordonnançable, Ghosh [34] part de l'hypothèse que l'on connait un ordonnancement conservatif satisfaisant pour l'application visée.

Il impose ensuite deux principales conditions :

- D'abord, la simulation doit être sans fausse date (NFT[1]). Dans ce type de simulation, si un évènement est créé à une date donnée au cours de la simulation, un évènement *valide* (pas forcément le même) est créé à cette même date au cours de la simulation. Des évènements invalides – qui seront supprimés du système par le mécanisme de retour arrière – peuvent donc être générés, pour peu qu'ils soient remplacés plus tard par des évènements valides.
- Ensuite, le protocole de synchronisation optimiste choisi doit mettre en œuvre le principe d'annulation paresseuse [32]. Rappelons que dans ce type de synchronisation, les retours arrière ne sont pas immédiatement propagés. On s'assure d'abord qu'ils remettent bien en cause les données émises vers d'autres processus logiques.

Sa démonstration aboutit au théorème de *R-ordonnançabilité* suivant :

> *Si l'on connait un ordonnancement conservatif valide, au sens temps-réel, pour une application donnée et si cette application est sans fausse date, l'ordonnancement par un protocole optimiste avec annulation paresseuse est possible, au sens temps-réel, moyennant l'ajout d'une durée R au pire temps d'exécution de chaque tâche.*

La durée *R* correspond au temps nécessaire à l'accomplissement d'un retour arrière. Elle englobe le temps de mémorisation de l'état du processus logique, le temps de restauration de cet état et le temps de transfert des anti-messages.

La principale difficulté est que le cadre des NFT imposé aux applications est très restrictif et difficilement appréciable en pratique. Ghosh montrera plus tard que les protocoles de type « agressifs, sans risques » (ANR[2]) suggérés par, Reynolds [68] constituent un cas extrême de protocole paresseux et qu'ils sont également *R-ordonnançables* [35]. Ils ont l'avantage de rendre toute application compatible avec le cadre de la démonstration de [34]. Ces protocoles restreignent les retours arrière à des usages exclusivement locaux. Seuls des messages valides peuvent être échangés entre les processus logiques.

Plusieurs simulateurs mettront en œuvre un protocole ANR [21]. Steinman [78] proposera deux ans avant la démonstration de Ghosh un protocole ANR mis en œuvre dans un simulateur interactif appelé SPEEDES. Ghosh proposera sa version, le système PORTS, trois ans plus tard sans véritablement en préciser les mécanismes internes [35]. SPEEDS et PORTS restent encore aujourd'hui les deux principales références d'implémentation d'un protocole ANR dans un cadre temps-réel [56].

1 *No False Timestamps* dans le texte
2 *Agressive No Risk* dans le texte.

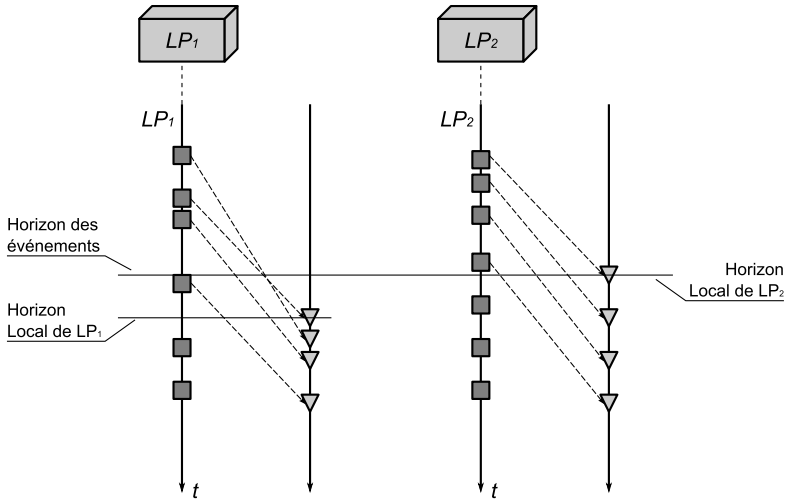


Figure 14. L'horizon des évènements

Le principe du protocole ANR de SPEEDES repose sur la notion d'horizon des évènements [80]. Steinman s'appuie sur le fait que les évènements engendrent des messages toujours positionnés dans le futur sur l'axe temporel de la simulation. Chacun de ces messages est susceptible d'engendrer un évènement à sa date propre, c'est-à-dire à une date postérieure à celle de l'évènement qui en est la cause[1]. Partant de l'évènement non encore évalué le plus ancien, on peut identifier la date avant laquelle aucun message ne sera plus généré. Cette date est par définition l'horizon des évènements. Tous les évènements situés avant l'horizon peuvent être évalués sans risque d'être annulés par un retour arrière. Cependant, la détermination de cet horizon suppose un calcul global sur l'ensemble des processus logiques tout aussi difficile que de déterminer le GVT d'une simulation optimiste quelconque.

Le protocole de Steinman repose sur un cycle de simulation en deux phases :

- Lors de la première phase, chaque processus logique exécute les évènements de sa liste jusqu'à identifier l'horizon des évènements locaux. Lorsque la date du prochain évènement à évaluer est inférieure à la date du message généré le plus ancien, l'horizon local est atteint. Chaque processus diffuse la date trouvée de manière à identifier l'horizon global, c'est-à-dire le minimum des horizons locaux sur l'ensemble des processus logiques du système.
- Lors de la deuxième phase les messages qui ont été générés lors de la phase précédente sont envoyés à leurs destinataires. Seuls les messages générés par des évènements antérieurs à l'horizon global sont envoyés. Cette phase peut engendrer localement des retours arrière pour les évènements situés temporellement entre l'horizon global et l'horizon local. Mais les messages diffusés ne seront eux, jamais remis en cause.

1 Principe de causalité, ni plus ni moins.

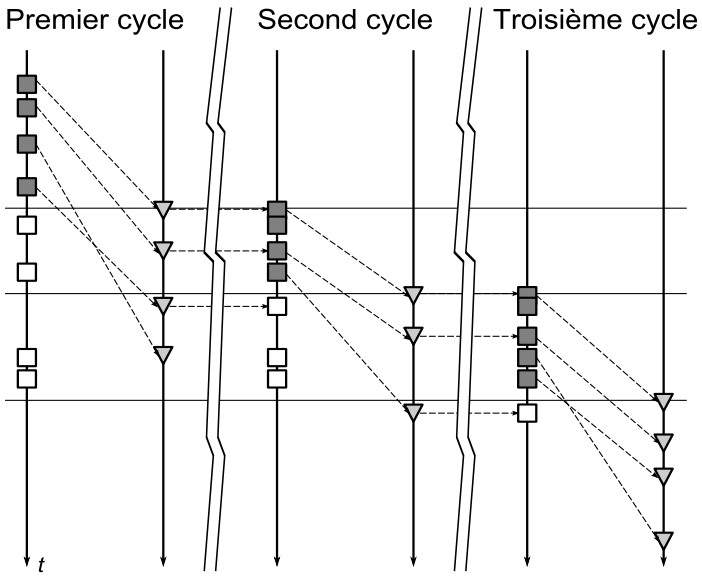


Figure 15. Trois cycles du protocole de SPEEDES.

Le mécanisme de SPEEDES est loin de convenir à tous les cas de figure. Les possibilités d'anticipation étant réduites, l'algorithme ne sera efficace que si la charge de calcul est bien répartie entre les processus logiques. Par ailleurs, l'efficacité de l'algorithme se trouve rapidement détériorée si l'horizon des évènements ne dépasse pas le cycle Δ. De façon plus générale, ce protocole impose aux différents processus logiques d'avancer de façon synchrone, cycle par cycle, afin de pouvoir déterminer l'horizon global des évènements. Cette nécessité algorithmique établit des relations de dépendance entre tous les processus logiques là où l'application proprement dite n'en n'impose pas nécessairement. C'est là le défaut principal de ce protocole.

Avant de clore cette section, rappelons que le cadre fixé par Ghosh [34] est celui des protocoles de synchronisation paresseux. L'ANR implémenté dans SPEEDES est une version extrême de cette famille de protocoles. Il est donc beaucoup plus restrictif que nécessaire. En pratique, certains simulateurs temps-réel sortent du cadre fixé par Ghosh et utilisent l'algorithme de Jefferson sans trop de restriction, par exemple pour tirer avantage de la capacité de Jefferson à prendre de l'avance [41]. Les calculs longs peuvent ainsi être anticipés.

Il faut donc faire la part des choses. Prenons acte que d'un point de vue théorique, le protocole de Jefferson n'est pas prouvable à moins d'être sévèrement encadré ; et autorisons-nous à une réflexion plus intuitive et pragmatique. Les propriétés d'ordonnançabilité des protocoles paresseux proviennent du fait qu'ils évitent les réactions en chaine lorsqu'un retour arrière est engagé. L'idée fondamentale est donc celle-ci : les protocoles optimistes sont d'autant plus compatibles avec des contraintes temps-réel qu'ils évitent la propagation de retours arrière en cascade. Plus le contexte est dur, plus les retours arrière sont à proscrire. Reste à savoir comment procéder en pratique pour synchroniser un simulateur avec l'horloge murale.

3.3 La synchronisation temps-réel des simulateurs distribués optimistes

Le mécanisme employé par SPEEDS est assez particulier [79]. Celui-ci progresse cycle de simulation par cycle de simulation. Au cours d'un cycle, plusieurs évènements peuvent être traités par un même processus logique. Les données qui entrent dans le système sont injectées au cycle qui suit le cycle en cours d'exécution tandis que les données qui sortent du système sont lues à partir du cycle précédant le cycle courant. Le contrôle temporel des cycles permet au simulateur de maintenir un STAR correct pendant l'exécution.

Cette technique impose à tous les processus logiques de suivre plus où moins le WCT ce qui est parfaitement inutile et contraint abusivement la durée effective de traitement de chaque évènement. En toute rigueur, les seuls processus logiques qui ont l'obligation de suivre l'horloge murale sont ceux qui gèrent des interactions avec le monde extérieur.

Une deuxième approche, plus subtile, consiste, d'une part, à exploiter les interactions pour synchroniser les processus logiques en interaction avec le monde extérieur et, d'autre part, à laisser le protocole de synchronisation se charger des autres processus logiques.

Selon ce principe, la gestion des sorties est assez simple à assurer [79]. Il suffit de placer les messages dans un tampon d'attente. Lorsque que le WCT atteint son échéance le message est livré au monde extérieur. Cette idée est reprise par Das [18] et Hybinette [39] dans un simulateur optimiste à travers le concept d'évènement d'entrée/sortie non bloquant. Si une annulation est engagée avant l'échéance du message, il est simplement retiré du tampon de sortie et ne sera jamais délivré au monde extérieur.

Le problème des entrées est un peu plus délicat. Simmonds [74] et Xiao [91] proposent d'utiliser des processus logiques particuliers : les *RT-task*. La particularité de ces processus temps-réel est que leur temps propre n'est pas lié aux messages qu'ils traitent mais à l'horloge murale temps-réel. Les mécanismes de synchronisation du protocole mis en œuvre se chargent de synchroniser de façon naturelle les autres processus logiques de l'application au travers des échanges de messages. C'est cependant dans un cadre conservatif que Simmonds et Xiao mettent en œuvre cette idée. Hybinette [41] transposera cette technique dans le cadre d'un noyau de simulation optimiste. Il définit une classe d'évènements particuliers appelés évènements d'entrées/sorties bloquants et ajoute à son simulateur un processus logique spécifique dit « temps-réel » destiné au traitement de ces évènements. La progression temporelle du processus temps-réel est liée à l'horloge murale. Lorsqu'il traite un évènement bloquant, le processus se place en attente de l'échéance de l'évènement. Lorsque l'horloge murale coïncide avec la date de l'évènement, l'évènement est traité. Les messages produits provoquent, en cas de besoin, un retour arrière des autres processus logiques du système et ramènent le temps du simulateur – c'est-à-dire le GVT – à une date appropriée proche du WCT.

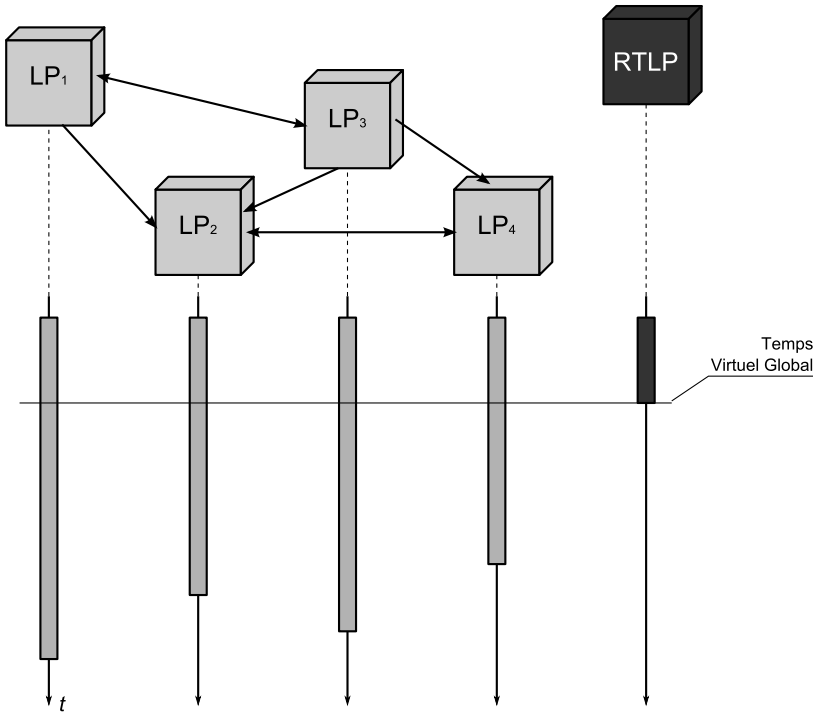


Figure 16. Avance temporelle des différents processus logiques. Le processus logique temps-réel impose le GVT.

Comme le souligne Jefferson [42], le temps virtuel global (GVT) tient une place fondamentale dans la synchronisation temps-réel des simulateurs optimistes. Le GVT est défini par la date de l'évènement non évalué le plus ancien. Il marque la limite temporelle avant laquelle les messages produits par le simulateur ne peuvent plus être annulés. Comme le monde extérieur peut difficilement revenir en arrière, il n'est pas acceptable de lui fournir des données qui peuvent être remises en causes. En conséquence, seuls les messages dont la date est antérieure au GVT peuvent être délivrés sans risque.

Mais contrairement au cas d'un simulateur non temps-réel, les messages en question ne doivent pas être détruits du système avant leur livraison. Ces conditions imposent une relation entre le GVT, l'horloge murale et la limite des objets que l'on peut supprimer. Pour clarifier ces relations temporelles, Franks [28] définit deux nouveaux temps globaux :

- le temps virtuel des objets fossiles (FVT[1]) qui délimite l'ensemble des objets définitivement inutiles ;
- le temps virtuel universel (WVT[2]) qui correspond à la date du dernier message délivré au monde extérieur.

Les trois temps ainsi définis doivent alors être tels que :

$$FVT < WVT \leqslant GVT$$

1 *Fossil-collect Virtual Time* dans le texte.
2 *World Virtual Time* dans le texte.

Le caractère temps-réel du simulateur pourra être apprécié par le ratio d'avancement calculé entre le WCT et le WVT. En pratique, tant que le GVT est postérieur au WCT, le simulateur est en mesure de répondre aux exigences temps-réel.

On peut voir dans les contraintes précédentes une contradiction. D'un côté, une donnée en provenance de l'extérieur du simulateur est nécessairement injectée dans le système à une date antérieure au WCT, au mieux égale à celui-ci. Conséquemment, chaque nouvelle donnée provoque potentiellement un retour arrière qui reconduit irrémédiablement le GVT derrière le WCT. D'un autre côté, pour qu'une donnée de sortie puisse être délivrée à temps au monde extérieur, elle doit être, au moment de sa sortie, postérieure au WCT, au pire égale à celui-ci, ce qui suppose de la part du simulateur de travailler avec un peu d'avance. Le GVT doit donc être postérieur au WCT, comme nous l'avons dit plus haut. Le GVT se trouve donc coincé de part et d'autre du WCT ce qui n'est pas possible. Pour résoudre cette contradiction, il faut nécessairement relâcher les contraintes temporelles et donner au simulateur le temps dont il a besoin pour traiter les différents évènements qui séparent l'entrée de la sortie. Steinman [79], par exemple, impose que les données injectées dans le système le soient au prochain cycle d'exécution et que les données de sortie délivrées correspondent au dernier cycle achevé ; Zhao [93] quant à lui associe aux différentes tâches un délai virtuel qui permet à chaque tâche de délivrer ses données avec un peu d'avance sur l'horloge murale ; etc. De manière générale, on trouvera dans les simulateurs temps-réel deux mécanismes complémentaires : une synchronisation en temps-réel en certains points de l'application et des délais de latence sur les chemins de données pour accorder aux traitements le temps dont ils ont besoin pour être effectués.

*
* *

Nous avons vu que les pratiques classiquement mises en œuvre dans les systèmes temps-réel distribués reposent sur deux approches. La première, l'ordonnancement temporel, offre une sécurité de fonctionnement exemplaire au prix d'une analyse temporelle statique poussée qui interdit toute adaptation dynamique de l'architecture de l'application en cours de fonctionnement. La seconde, l'ordonnancement évènementiel, repose bien souvent sur un ordonnancement préemptif à priorité, beaucoup plus souple que l'ordonnancement temporel. Le prix à payer est que ce type d'ordonnancement conduit à des applications dont le comportement n'est pas déterministe. L'étude menée dans ce chapitre a montré qu'une troisième voie était envisageable, basée sur les principes d'ordonnancement utilisés dans le domaine de la simulation distribuée. Ces systèmes prennent en compte nativement les relations causales entre les tâches. Tout en étant de nature évènementielle, l'ordonnancement garantit donc le déterminisme comportemental de l'application. Reste à évaluer les capacités temps-réel de ces algorithmes. Deux familles d'algorithmes existent : l'une qualifiée de conservative, l'autre d'optimiste. Les protocoles conservatifs ne permettent pas tout à fait d'atteindre la souplesse attendue en terme de dynamisme de l'architecture de l'application mais peuvent

déboucher sur des systèmes temps-réel durs. Les protocoles optimistes offrent une grande souplesse du fait des mécanismes de retour arrière et de ré-exécution qu'ils utilisent pour se synchroniser. Ils sont cependant plus délicats à contenir dans un cadre temps-réel dur. Sous certaines conditions, Ghosh [34] a néanmoins montré que cela était possible si l'application évitait les réactions en chaine de retours arrière. On peut donc envisager la construction d'un système temps-réel distribué sur la base de ses algorithmes. Le chapitre suivant tentera de poser les bases d'un tel système.

CHAPITRE 2
VERS UN MODÈLE ÉVÈNEMENTIEL TEMPS-RÉEL DÉTERMINISTE

Le chapitre précédent a montré que le protocole conservatif de Chandy [11] et le protocole optimiste de Jefferson [42] pouvaient, sous certaines conditions, s'inscrire dans un cadre temps-réel dur. Ces algorithmes distribués offrent à la fois un cadre d'exécution évènementiel et un comportement déterministe. C'est cette double caractéristique qui conduit Zhao [94] à utiliser ces mécanismes de distribution pour élaborer un modèle d'exécution temps-réel. L'algorithme qu'il propose s'appuie sur un protocole conservatif. Une analyse topologique statique de l'application permet de classer les évènements du système de manière à pouvoir les traiter sans avoir recours à un quelconque mécanisme de retour arrière. Malheureusement, les spécifications d'AROS sont incompatibles avec une telle analyse. L'architecture d'une application AROS est une structure dynamique qui peut être modifiée à tout instant pendant l'exécution. On cherche donc un mécanisme de synchronisation beaucoup plus souple capable d'assurer localement un cadre temps-réel sans pour autant le garantir sur l'ensemble de l'application. Notre algorithme doit être capable d'encaisser les bouleversements structurels sans remettre en cause le fonctionnement du système et sans nécessiter d'analyse topologique. Un protocole conservatif n'est donc pas envisageable. C'est vers un protocole optimiste que nous allons nous tourner.

L'intérêt majeur des protocoles de type Jefferson est d'accepter l'imprévisible. Le mécanisme de retour arrière permet en effet de revenir sur n'importe quelle situation incorrectement traitée pour effectuer les opérations en respectant la causalité des évènements. Potentiellement, toute situation architecturale nouvelle

peut être prise en compte même si elle est détectée tardivement. Reste à contenir ce mécanisme dans un cadre temporel acceptable. Or, on sait que les retours arrière compromettent fortement l'ordonnançabilité de l'exécution. Il faut donc limiter leur usage aux situations exceptionnelles.

L'idée maîtresse du moteur d'exécution d'NJN est d'accepter des phases transitoires non-temps-réel pendant lesquelles les paramètres de l'application s'ajustent afin d'atteindre la relaxation du système. En fonctionnement normal, l'architecture de l'application est stable et chaque processus logique du système est synchronisé sans qu'il soit fait grand usage des retours arrière. On peut alors s'approcher d'un fonctionnement temps-réel dur. Si une modification structurelle majeure survient dans l'application, un nouveau point d'équilibre doit être trouvé. Les retours arrière sont alors utilisés pour assurer la causalité du système en attendant le nouveau point d'équilibre. Pour atteindre ce point d'équilibre, NJN s'appuie sur une analyse statistique des propriétés temporelles des tâches.

La première partie de ce chapitre montre les mécanismes de synchronisation temps-réel mis en œuvre dans NJN. Nous verrons qu'ils reposent, comme pour Zhao, sur des points de synchronisation localisés et sur le relâchement des contraintes temporelles le long des chemins de données. La partie suivante porte sur la politique d'ordonnancement dont l'objectif est de limiter l'usage des retours arrière. On abordera dans la troisième partie les possibilités de fonctionnement d'NJN dans des contextes temps-réel dur.

1 Principe de synchronisation temps-réel

Le fonctionnement temps-réel d'NJN repose, comme chez Zhao, sur deux éléments fondamentaux : des éléments en interaction avec le monde extérieur qui sont synchronisés en temps-réel et un relâchement des contraintes temporelles le long des chemins de données. Après un bref rappel des différents acteurs du cadre exécutif, on revient dans cette partie sur ces deux éléments.

1.1 Le cadre exécutif : des tâches, des signaux, des processus et des messages

Un exécutif NJN peut être observé selon deux niveaux. Le premier est relatif à la l'architecture algorithmique de l'exécutif et le second concerne la manière dont cette architecture est distribuée.

Du point de vue de l'exécutif, une application NJN est composée de tâches et de signaux (figure 17). Les tâches renferment le code actif de l'application tandis que les signaux assurent la communication entre les tâches. D'un point de vue fonctionnel, les tâches lisent l'état des signaux et y inscrivent de nouvelles valeurs. Chaque tâche est contrôlée par une liste de sensibilité qui énonce ses conditions d'activation. Il s'agit en pratique de la liste des signaux que la tâche doit surveiller. Si un évènement se produit sur l'un des signaux de cette liste, la tâche est programmée pour être exécutée.

Lorsqu'une tâche est activée, un évènement d'exécution est créé qui associe la tâche et la date à laquelle elle doit être exécutée. Lorsque son exécution a lieu, chaque opération effectuée sur un signal donne lieu à un évènement d'écriture qui associe la nouvelle valeur affectée au signal, l'évènement d'exécution qui est à l'origine de l'opération et la date à laquelle elle a été effectuée.

Les relations causales entre évènements sont représentées temporellement par des cycles Δ. Une tâche exécutée à une date t consulte l'état des signaux du système à cette même date et produit des évènements sur d'autres signaux à une date $t+\Delta$. Il n'est donc pas possible pour une tâche d'observer les modifications qu'elle produit sur l'application.

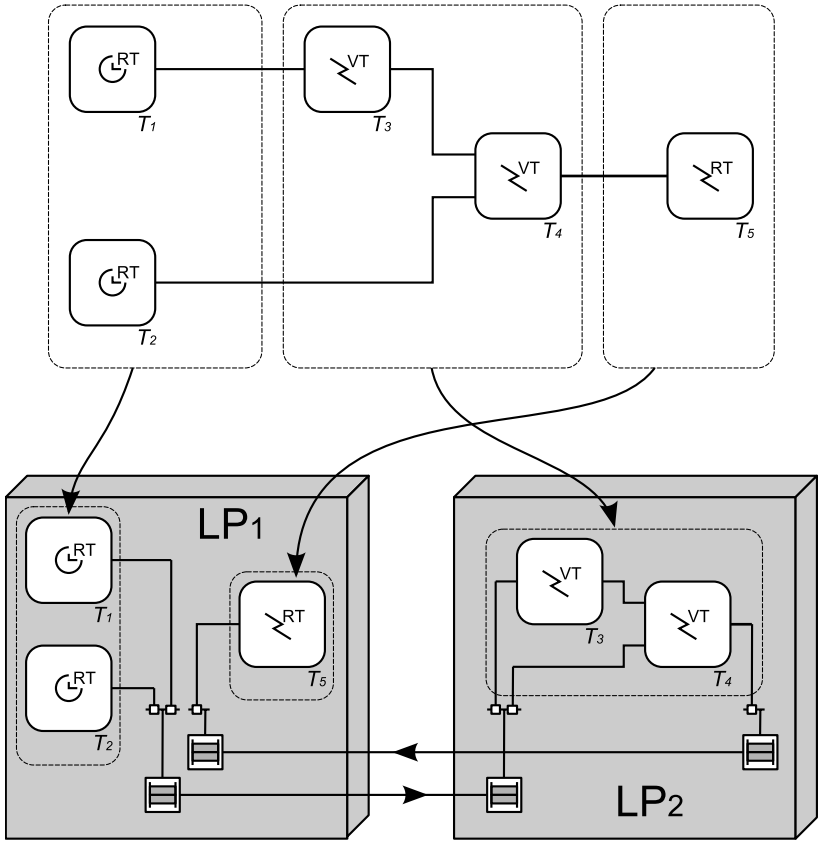


Figure 17. Un exécutif NJN distribué. Les tâches échangent des données via les signaux de l'application. Elles sont distribuées sur plusieurs processus logiques. Des messages assurent la communication entre les processus logiques.

Les tâches de l'application sont distribuées sur différents processus logiques (figure 17). Un processus logique héberge en général plusieurs tâches. Les évènements qui doivent être transférés entre deux tâches situées sur des processus logiques différents sont acheminés sous la forme de messages.

L'ordonnancement des processus logiques et l'acheminement des messages sont pris en charge par le système d'exploitation hôte. NJN se charge de gérer les évènements et l'ordonnancement des tâches au sein même des processus logiques.

1.2 Deux types de tâches : les tâches temps-virtuel et les tâches temps-réel

Le modèle d'exécution d'NJN s'appuie sur deux types de tâches (figure 17) : les premières sont dites « temps-virtuel » et sont d'ordinaire des tâches de traitement tandis que les secondes sont qualifiées de « temps-réel » et sont en principe chargées des entrées/sorties avec le monde extérieur.

Les tâches temps-virtuel correspondent à celles que l'on va trouver dans un simulateur. Elles sont déclenchées par les évènements qui se produisent sur les signaux de leur liste de sensibilité. La date qui est associée à leur exécution est sans rapport avec l'horloge murale. Elle correspond à la date de l'évènement qui les a activées. Un ordonnanceur spécifique est chargé d'établir l'ordre dans lequel doivent être exécutées ces tâches, indépendamment de toute considération temps-réel. En cas de nécessité, l'exécution de ces tâches peut être remise en cause ce qui donne lieu à un retour arrière.

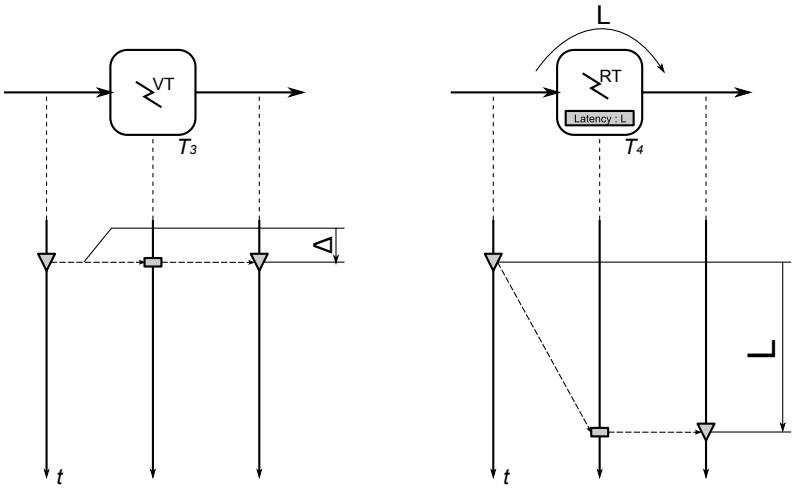


Figure 18. Comportement temporel des tâches temps-virtuel et temps-réel.

Les tâches temps-réel assurent la synchronisation temporelle du système dans son ensemble. Contrairement aux tâches temps-virtuel, elles ne sont pas exécutées à n'importe quel moment. C'est lorsque l'horloge murale arrive en concordance avec leur date d'exécution que l'ordonnanceur leur donne la main. Leur exécution étant liée à une interaction avec l'extérieur, il est impossible de remettre en cause leur exécution. Les retours arrière ne les concernent donc pas.

Comme pour leur homologue virtuel, la date d'exécution qui leur est associée est déterminée à partir de la date de l'évènement qui est à l'origine de leur activation. En revanche, un retard appelé latence est systématiquement introduit entre

l'activation et l'exécution. Ce paramètre est caractéristique de chaque tâche et peut être ajusté en cours d'exécution. Concrètement, si un évènement d'écriture se produit à la date *t* sur l'un des signaux de la liste de sensibilité associée à la tâche, celle-ci sera exécutée à la date *t+L*, *L* étant la latence associée à cette tâche.

Une tâche temps-réel observe les signaux de l'application tels qu'ils étaient au moment de son déclenchement et non tels qu'ils sont au moment de l'exécution. Elle voit donc le système avec un retard correspondant à la valeur de sa latence. Elle modifie l'état des signaux sur lesquels elle écrit un cycle Δ après la date de son exécution, de la même manière que son homologue virtuel. Ainsi, une durée correspondant à la latence sépare ses lectures de ses écritures.

1.3 Rôle de la latence

La latence associée aux tâches temps-réel donne à NJN la marge temporelle dont il a besoin pour fonctionner correctement, tout en permettant de conserver la cohérence temporelle des données durant toute la chaine de traitement.

Examinons la chaine de traitement de la figure 19. Elle est constituée de quatre tâches T_1, T_2, T_3 et T_4. La première, T_1, et la dernière, T_4, sont des tâches temps-réel. La latence de T_4 est notée L. La latence de T_1 n'a ici que peu d'importance. La tâche T_1 produit des évènements qui sont traités successivement par T_2, T_3 et enfin T_4.

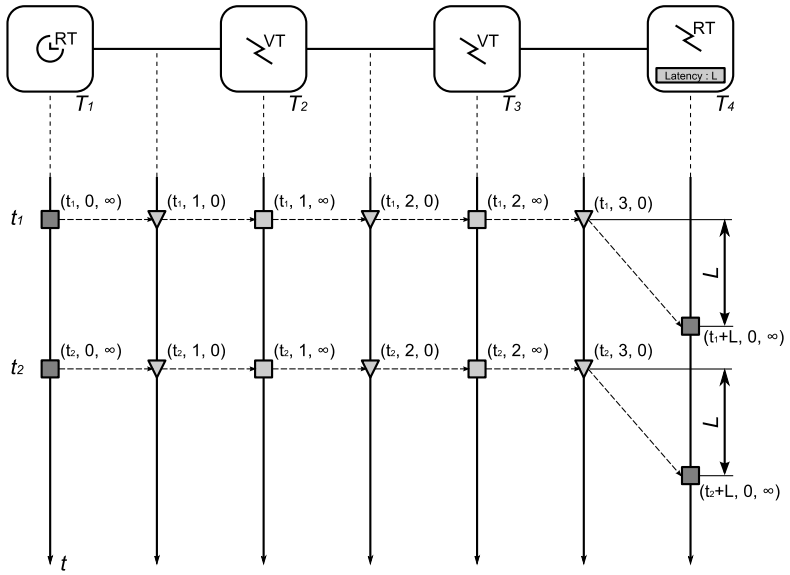


Figure 19. Chronogramme d'exécution d'une chaine de traitement selon une échelle virtuelle du temps.

La figure 19 montre l'enchainement de deux traitements successifs selon un axe temporel virtuel, c'est-à-dire un axe temporel relatant les dates associées aux différents évènements. On voit que les données consommées et produites par toutes les tâche de la chaine ont, au Δ près, la même date.

La figure 20 montre une exécution possible du même enchainement de traitements mais cette fois selon un axe temps-réel. Les évènements sont positionnés selon la date effective à laquelle ils sont traités. Par ailleurs, la durée de traitement est également représentée par la hauteur du symbole qui figure l'évènement. On voit que la position des évènements d'exécution associés à des tâches temps-réel n'a pas bougé. Les évènements des tâches temps-virtuel se sont positionnés au plus tôt sur la ligne temporelle et dans l'ordre induit par les relations causales naturelles.

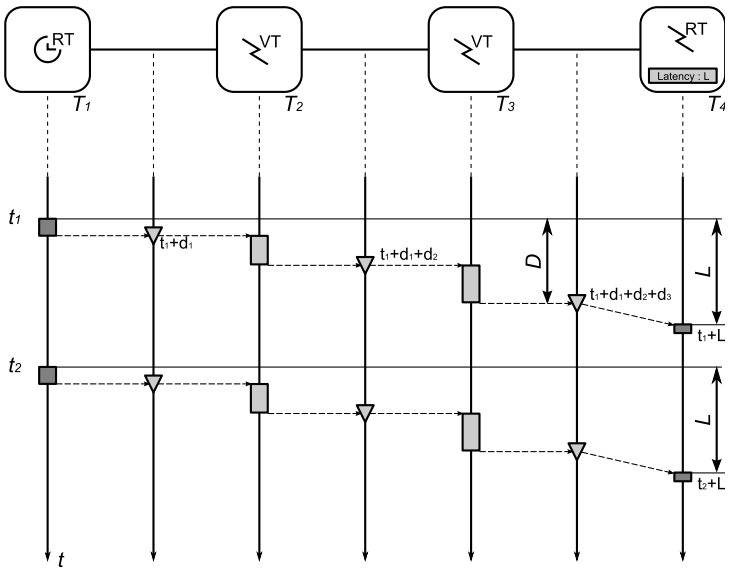


Figure 20. Chronogramme d'exécution de la chaine de traitement de la figure 19 selon une échelle réelle du temps.

Ce petit exemple montre le rôle fondamental joué par la latence des tâches temps-réel. En provoquant un retard d'exécution des tâches temps-réel de sortie vis-à-vis de l'origine temporelle de la chaine de traitement, la latence donne au système la marge temporelle nécessaire pour exécuter la chaine dans son ensemble.

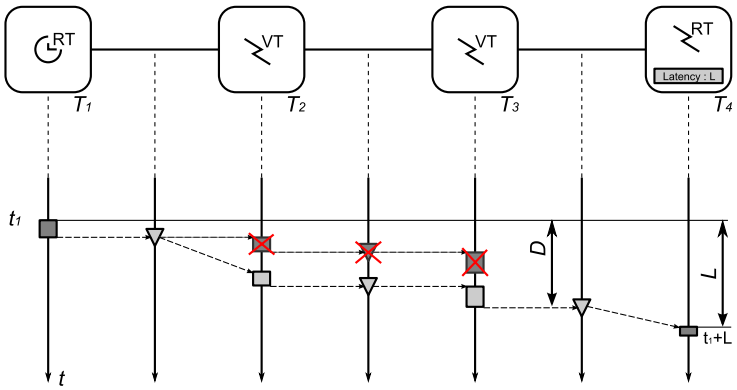


Figure 21. Un retour arrière de la tâche T_2.

Toutes les tâches qui entrent en jeu dans la chaine doivent être exécutées dans cet intervalle de temps.

Dans un contexte distribué, la latence des tâches temps-réel doit être fixée de manière à ce que tous les éléments de la chaine puissent être traités, c'est-à-dire l'exécution de chaque tâche mais aussi les échanges de message ainsi que les retours arrière éventuels (figure 21). Une latence trop faible conduit au non-respect de la contrainte temps-réel fixée. C'est donc le paramètre le plus déterminant du système.

On a montré comment se synchronisait NJN par l'intermédiaire des tâches temps-réel. Rappelons que les contraintes temps-réel sont d'autant plus faciles à respecter que le système évite les retours arrière. Voyons comment la politique d'ordonnancement des tâches au sein d'un processus logique parvient à restreindre l'usage des retours arrière.

2 Politique d'ordonnancement

Le rôle des tâches temps-réel est capital dans la synchronisation d'NJN, nous venons de le voir. L'ordonnancement doit donc garantir le fonctionnement correct de ces tâches.

2.1 Comment trop d'avance peut compromettre le respect des contraintes temps-réel

La politique adoptée vise deux objectifs : premièrement, donner une priorité absolue à l'exécution des tâches temps-réel arrivées à échéance ; deuxièmement, réserver les retours arrière aux situations exceptionnelles puisqu'on sait qu'ils compromettent le respect des contraintes temps-réel.

Le premier point n'est pas difficile à résoudre. Chaque processus logique gère deux files d'évènements d'exécution : la première concerne les évènements relatifs aux tâches temps-réel ; la seconde se charge des évènements relatifs aux tâches temps-virtuel. Les évènements de la première file sont prioritaires dès qu'ils arrivent à échéance de manière à garantir l'aspect temps-réel de leur exécution.

Le second point est plus délicat et nécessite de bien analyser les mécanismes qui sont à l'origine du retour arrière ainsi que ses conséquences sur le comportement temporel du système. La figure 22 montre un exemple d'application où le phénomène peut apparaitre.

L'application comporte cinq tâches, nommées T_1 à T_5, distribuées sur deux processus logiques, nommés LP_1 et LP_2. La tâche T_1 est hébergée par le processus LP_1, toutes les autres tâches sont hébergées par le processus LP_2. Les tâches T_1 et T_2 sont des tâches temps-réel d'entrée. La tâche T_5 est une tâche temps-réel de sortie et comporte une latence L.

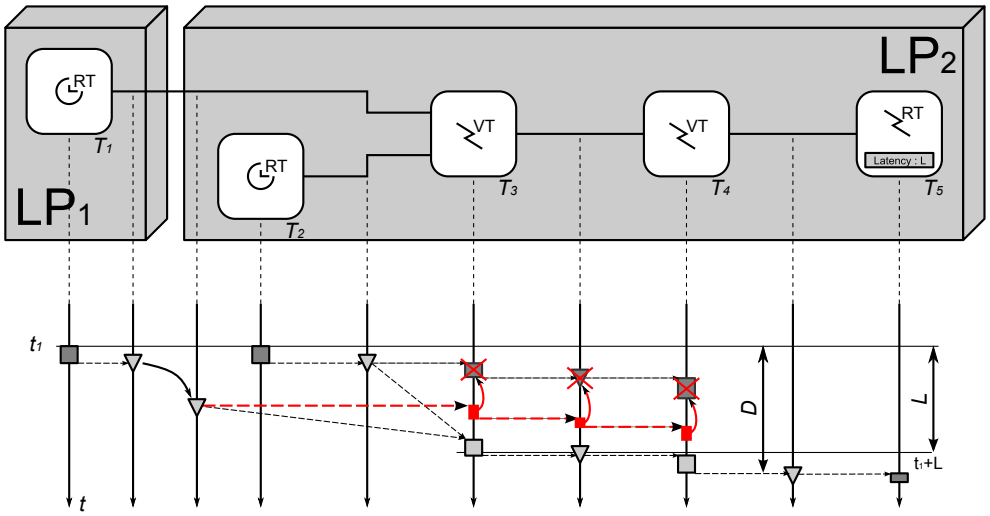


Figure 22. Une application distribuée sur deux processus logiques. Le retour arrière met en défaut les contraintes temps-réel.

La tâche T_3 consomme des données issues des deux tâches T_1 et T_2. C'est sur cette tâche qu'apparaissent potentiellement les violations de causalité. En effet, les évènements consommés en provenance du processus LP_1 doivent être acheminés via le réseau si bien qu'ils sont livrés au processus LP_2 avec un retard provoqué par la latence du réseau. La violation de causalité se produit si les évènements en provenance de T_2 sont consommés sans délai. Chaque nouvel évènement en provenance de T_1 provoque alors un retour arrière de T_3 et T_4. En plus du temps nécessaire pour effectuer le retour arrière, la ré-exécution des deux tâches retarde la production de l'évènement à destination de la tâche de sortie. Dans l'exemple de la figure 22, ce retard est tel que la tâche T_5 n'est pas exécutée à temps.

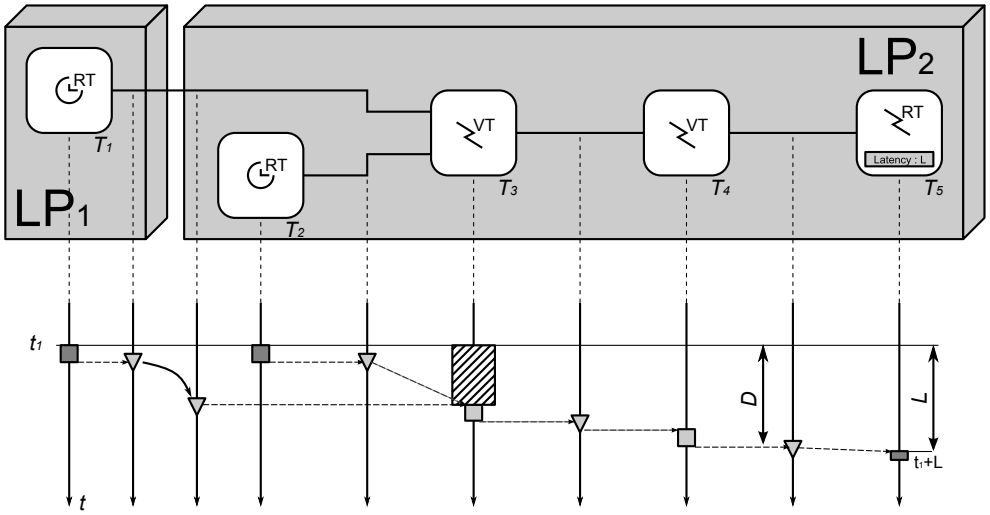


Figure 23. Effet du délai de garde sur l'exécution de la tâche de sortie.

Exécuter une tâche trop tôt risque paradoxalement de provoquer une violation de latence.

En retardant artificiellement l'exécution de la tâche T_3, on évite ce phénomène. La figure 23 montre ce qu'il en est. Un délai, appelé délai de garde, a été introduit avant l'exécution de la tâche T_3 de manière à ce que les évènements en provenance du processus LP_1 aient le temps d'être acheminés. Le retour arrière est évité ce qui permet cette fois de respecter la latence de la tâche de sortie.

L'idée maîtresse de la politique d'ordonnancement des tâches temps-virtuel s'inspire de [26]. Elle consiste à associer à chaque tâche un délai de garde qui lui évite, dans les situations normales, d'être exécutée trop tôt. Ce délai est déterminé de façon statistique de manière à satisfaire un risque de retour arrière inférieur à une certaine limite.

L'ordre d'exécution des tâches temps-virtuel n'est donc pas régi uniquement par la date qui est associée à leur évènement d'exécution. La valeur de la garde est ajoutée à cette date de manière à permettre à NJN d'exécuter les tâches qui peuvent l'être dès l'échéance de leur délai de garde. Notons que par construction, il ne peut pas y avoir d'inversion dans l'ordre d'exécution des tâches puisque la garde ne définit pas une date d'exécution mais une date limite avant laquelle l'exécution est bloquée. Quelque soit la valeur de sa garde, pour qu'une tâche soit exécutée, il faut qu'un évènement la réveille. Et cet évènement provient nécessairement des tâches qui précèdent celle considérée dans la chaine de traitement.

2.2 Le délai de garde

Le délai de garde correspond au retard, mesuré vis-à-vis de l'horloge murale, entre la date de l'évènement d'exécution et la date d'exécution au plus tôt de la tâche. Si on souhaite éviter à la tâche un retour arrière, il convient de fixer ce délai de manière à ce que toutes les tâches antérieures aient été exécutées.

Supposons un système de n tâches, numérotées de 1 à n, constituant le chemin critique d'une chaine de traitement séquentielle. Seules la première et la dernière tâche interagissent avec l'extérieur ce qui leur impose un positionnement temporel fixé ; ce positionnement n'est pas nécessaire pour les tâches intermédiaires. On fixe la probabilité p qu'aucune tâche du chemin critique ne subisse un retour arrière, typiquement de l'ordre de 99,999%. On fait, par ailleurs, les hypothèses simplificatrices suivantes :

- Les durées d'exécution t_i des tâche peuvent être modélisées par les variables aléatoires T_i deux à deux indépendantes, distribuées suivant des lois normales d'espérance μ_i et d'écart type σ_i ;
- Chaque tâche est exécutée dès l'échéance de son délai de garde.
- Si une tâche i subit un retour arrière, les tâches $i+1$ à n en subissent un également.

Soit la variable aléatoire X_i vraie si la tâche i ne subit aucun retour arrière. La troisième hypothèse faite plus haut conduit à ce que chaque variable aléatoire X_i soit liée à toutes les variables X_j d'indice j inférieur à i :

$$P(X_i) = P\left(X_i | X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\right) \cdot P\left(X_{i-1} \cap X_{i-2} \cap \cdots \cap X_1\right)$$

Il vient alors pour la dernière tâche :

$$P\left(X_n\right) = \prod_{i=1}^{n} P\left(X_i | X_1 \cap \cdots \cap X_{i-1}\right)$$

Or, pour qu'une tâche i ne subisse pas de retour arrière, sachant que les tâches précédentes n'en ont pas subies, il faut que sa garde g_i soit telle que la tâche précédente ait pu être exécutée, ce qui s'exprime par :

$$P\left(X_i | X_1 \cap \cdots \cap X_{i-1}\right) = P\left(g_i \geq g_{i-1} + T_{i-1}\right)$$

La probabilité p que la dernière tâche ne soit pas annulée s'exprime donc par :

$$P\left(X_n\right) = \prod_{i=1}^{n} P\left(g_i \geq g_{i-1} + T_{i-1}\right) \geq p$$

En normalisant cette expression, il vient :

$$P\left(X_n\right) = \prod_{i=1}^{n} P\left(\frac{g_i - g_{i-1} - \mu_{i-1}}{\sigma_{i-1}} \geq \frac{T_{i-1} - \mu_{i-1}}{\sigma_{i-1}}\right) \geq p$$

Où μ_i et σ_i sont respectivement la moyenne et l'écart-type de la durée d'exécution de la tâche i.

Par hypothèse : $\forall\, i \{1\,;2\,;\cdots;n\}\quad T_i \sim N\left(\mu_i, \sigma_i^2\right)$

En introduisant Φ_0, fonction de répartition de la variable normale centrée réduite, il vient :

$$\prod_{i=1}^{n} \Phi_0\left(\frac{g_i - g_{i-1} - \mu_{i-1}}{\sigma_{i-1}}\right) \geqslant p$$

En définissant $t_{a,i}$ le temps alloué à la tâche i, tel que :

$$t_{a,i} = g_{i+1} - g_i$$

On peut écrire :

$$\prod_{i=1}^{n-1} \Phi_0\left(\frac{t_{a,i} - \mu_i}{\sigma_i}\right) \geqslant p \qquad (1)$$

Ne pouvant déterminer de façon exacte chacun des $t_{a,i}$, on va chercher m tel que :

$$\forall\, i \quad m \leqslant \frac{t_{a,i} - \mu_i}{\sigma_i} \qquad (2)$$

En effet, la fonction Φ_0 étant strictement croissante,

$$m \leqslant \frac{t_{a,i} - \mu_i}{\sigma_i} \Rightarrow \Phi_0(m) \leqslant \Phi_0\left(\frac{t_{a,i} - \mu_i}{\sigma_i}\right)$$

Il en résulte que :

$$(\Phi_0(m))^n = p \Rightarrow \prod_{i=1}^{n} \Phi_0\left(\frac{t_{a,i} - \mu_i}{\sigma_i}\right) \geqslant p$$

Or :

$$(\Phi_0(m))^n = p \Leftrightarrow \Phi_0(m) = \sqrt[n]{p} \Leftrightarrow m = \Phi_0^{-1}\left(\sqrt[n]{p}\right) \qquad (3)$$

De (1), (2) et (3) il vient alors :

$$\forall\, i \quad t_{a,i} \geqslant \mu_i + \Phi_0^{-1}\left(\sqrt[n]{p}\right)\sigma_i$$

où Φ_0^{-1} est la fonction réciproque de la fonction de répartition de la loi normale centrée réduite.

D'où la garde g_i de la tâche i :

$$\forall\, i \quad g_i \geqslant g_{i-1} + \mu_{i-1} + \Phi_0^{-1}\left(\sqrt[n]{p}\right)\sigma_{i-1}$$

En respectant les temps trouvés pour la garde de chaque tâche et en fixant judicieusement la latence de la tâche de sortie, on s'assure, en théorie, de respecter les contraintes temps-réel fixées avec une probabilité arbitraire. Le choix de la latence de la tâche de sortie n'est pas différent de celui de la garde des tâches intermédiaires.

Ce résultat est bien évidemment à relativiser. Le modèle gaussien pris pour la durée d'exécution des tâches ne correspond pas exactement à la réalité. En pratique, il est intéressant car il rend l'estimation de la garde assez simple à mettre en œuvre.

2.3 Estimation du délai de garde en temps-réel

La valeur minimale de la garde de chaque tâche est estimée en temps-réel. Cette estimation est obtenue en mesurant le retard avec lequel sont inscrits les évènements des signaux utilisés par la tâche. Une estimation de la moyenne et de la variance de ce retard est déterminée pour chaque signal. Notons que le calcul correspondant doit être extrêmement rapide à effectuer, quitte à ne donner qu'un résultat très approximatif, afin de réduire son cout temporel sur l'exécution. On ajuste alors la garde de la tâche de manière à ce que son exécution démarre, dans la grande majorité des cas, après l'arrivée des évènements du signal le plus lent.

Chaque évènement correspondant au changement d'état d'un signal est doublement daté. Pour le $i^{ème}$ évènement $e_{s,i} = (v_i, t_{w,s,i}, t_{wc,s,i})$ du signal s, la première date $t_{w,s,i}$ est appelée date d'écriture. Il s'agit d'une date virtuelle. Elle correspond en général à la date d'exécution de la tâche responsable de l'écriture[1]. La seconde date $t_{wc,s,i}$ correspond à l'état qu'avait l'horloge murale lorsque

1 A laquelle on a ajouté un cycle Δ.

l'évènement a été créé dans la structure de donnée d'NJN. On est ainsi en mesure de connaître le retard $\Delta t_{w,s,i}$ avec lequel chaque évènement est créé (figure 24).

$$\Delta t_{w,s,i} = t_{wc,s,i} - t_{w,s,i}$$

On suppose que $\Delta t_{w,s,i}$ suit une loi normale de moyenne $\mu_{s,i}$ et de variance $\sigma_{s,i}^2$. On détermine alors une estimation de $\mu_{s,i}$ et $\sigma_{s,i}^2$ en prenant en compte majoritairement les derniers évènements inscrits sur le signal.

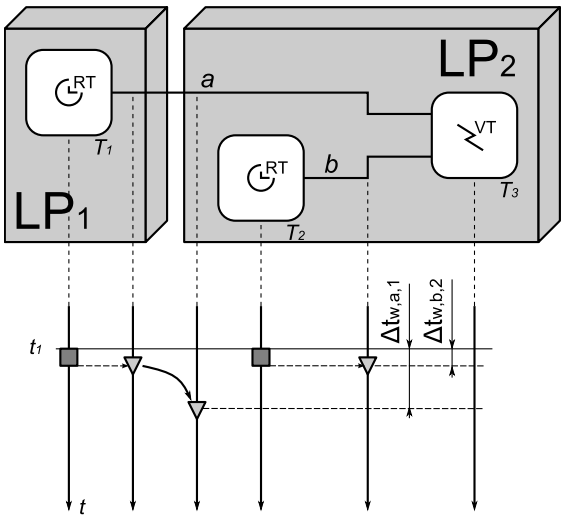


Figure 24. Estimation du retard à l'écriture de chaque évènement.

Une bonne estimation de la moyenne sur n évènements peut être obtenue par l'expression :

$$\hat{\mu}_s = \frac{1}{n}\sum_{i=1}^{n} \Delta t_{w,s,i}$$

De même, un estimateur sans biais de la variance est donné par :

$$\hat{\sigma}_s^2 = \frac{1}{n-1}\sum_{1}^{n} \left(\hat{\mu}_s - \Delta t_{w,s,i}\right)^2$$

Ces deux estimateurs nécessitent d'être recalculés intégralement pour chaque nouvel évènement, ce qui demande un temps de calcul proportionnel au nombre d'évènements pris en compte. Ce temps pourrait être fortement dommageable aux performances du système puisqu'il doit être calculé pour chaque nouvel évènement inscrit sur chaque signal de l'application.

Pour cette raison, on préfère estimer les deux paramètres statistiques de façon très approchée en privilégiant l'efficacité de l'exécution. On procède en pondérant l'estimation du paramètre obtenue à l'itération précédente par la valeur du retard correspondant au nouvel évènement. La moyenne est ainsi estimée par :

$$\hat{\mu}_{s,i} = \alpha\, \hat{\mu}_{s,i-1} + (1-\alpha)\, \Delta t_{w,s,i}$$

où α est un coefficient pondérateur compris entre 0 et 1, typiquement de l'ordre de 0,99.

De même, une estimation très approximative de la variance est déterminée par :

$$\widehat{\sigma^2}_{s,i} = \alpha \, \widehat{\sigma^2}_{s,i-1} + (1-\alpha)\left(\widehat{\mu}_{s,i} - \Delta\, t_{w,s,i}\right)^2$$

D'un point de vue strictement théorique, la garde doit être déterminée à partir des caractéristiques statistiques de l'ensemble des signaux consultés par la tâche. En pratique, il est assez difficile de déterminer cet ensemble avec exactitude sans analyser les opérations effectuées pendant l'exécution du corps de la tâche. De plus, certains de ces signaux – ceux destinés à gérer les aspects structurels de l'application par exemple – sont modifiés très rarement et ne sont donc pas très pertinents pour l'objectif fixé. A l'inverse, les signaux de la liste de sensibilité sont naturellement ceux qui sont les plus sollicités puisqu'ils correspondent aux éléments étroitement surveillés par la tâche. Ils ont en plus l'avantage d'être parfaitement connus dès la construction de l'application. Pour cette raison, on se limite à l'ensemble des signaux qui sont énoncés dans cette liste.

Soit S_T l'ensemble de signaux de la liste de sensibilité de la tâche T. On détermine la garde de T à partir des paramètres statistiques du signal $s \in S$ le plus lent :

$$g_T = max_{s \in S_T}\left(\widehat{\mu}^2_{s,i} + k \cdot \sqrt{\widehat{\sigma^2}_{s,i}}\right)$$

avec k, un paramètre de sécurité typiquement de l'ordre de 3 ou 4.

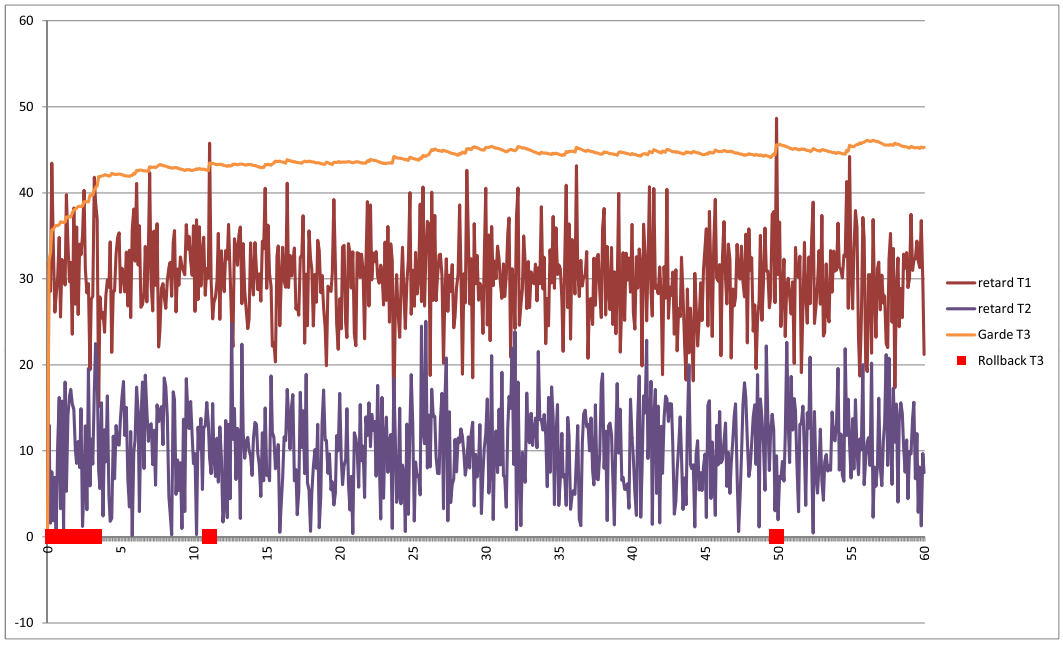


Figure 25. Simulation d'une minute d'exécution des trois tâches de la figure 24. Les tâches sont exécutées avec une périodicité de 100 ms. Sont représentés en ms le retard d'écriture de T_1 et T_2 vu de T_3 ainsi que la garde appliquée à l'exécution de T_3. Les carrés rouges signalent les retours arrière de la tâche T_3.

La figure 25 montre le résultat d'une minute de simulation d'une application telle que celle présentée figure 24. Les évènements en provenance de la tâche T_1 sont caractérisés par un retard moyen de 30 ms et d'écart type 5 ms. Les évènements en provenance de T_2 ont un retard moyen de 10 ms avec un écart type de 5 ms. Le coefficient α est ajusté à 0,99 tandis que le coefficient de sécurité k est fixé à 3. On observe la stabilisation de la garde de T_3 à une valeur d'environ 45 ms. A partir de 4 s d'exécution, la tâche T_3 ne subit plus que des retours arrière exceptionnels.

En théorie, la garde offre certaines garanties que les tâches du système ne subissent pas de retour arrière en régime permanent. Mais la validité de cette approche dépend de la qualité du modèle statistique et de celle des estimateurs. Or, les estimateurs choisis ne sont pas sans biais, loin s'en faut, et les échantillons utilisés comportent les évènements annulés par les retours arrière ce qui fausse l'estimation. Devant l'exigence de performance, la solution adoptée semble pourtant être la plus satisfaisante mais nécessite une analyse plus poussée[1].

3 Vers l'ordonnancement temporel

Le mécanisme de garde prémunit en principe des retours arrière lorsque l'application a trouvé un point de fonctionnement adapté à la durée d'exécution de chaque tâche. Dans ces conditions de fonctionnement et sous réserve que les sollicitations du système soient périodiques, NJN peut présenter certaines garanties d'ordonnançabilité. La première section de cette partie montrera que le principe de la garde peut être rapproché d'un ordonnancement temporel.

3.1 Une comparaison avec l'ordonnancement temporel

Examinons, sous le même angle statistique que la section 2.2 page 25, un ordonnancement de type temporel (*time triggered*) appliqué à un système de même structure. L'étude porte sur le chemin critique de l'application. On fixe la probabilité p que les contraintes temps-réel soient respectées. On fait, par ailleurs, les hypothèses simplificatrices suivantes :

- Les tâches s'exécutent successivement sans préemption ;
- Les durées d'exécution t_i des tâches peuvent être modélisées par les variables aléatoires T_i deux à deux indépendantes, distribuées suivant des lois normales d'espérance μ_i et d'écart type σ_i ;
- La complexité de l'algorithme d'ordonnancement est négligée.

Le respect des contraintes temps-réel avec une probabilité p s'exprime par :

$$P\left((T_1 < t_{a,1}) \cap (T_2 < t_{a,2}) \cap \cdots \cap (T_n < t_{a,n})\right) \geqslant p$$

Où $t_{a,i}$ est le temps alloué à la tâche i.

L'indépendance des variables T_i permet alors d'écrire :

1 Voir conclusion et perspectives p 189.

$$\prod_{i=1}^{n} P(T_i < t_{a,i}) \geqslant p \qquad (4)$$

L'inéquation (4) exprime le fait que pour respecter les contraintes temps-réel, aucune tâche ne doit dépasser l'intervalle de temps qui lui est alloué.

On cherche finalement les $t_{a,i}$ tels que :

$$\prod_{i=1}^{n} P\left(\frac{T_i - \mu_i}{\sigma_i} < \frac{t_{a,i} - \mu_i}{\sigma_i}\right) \geqslant p$$

Par hypothèse : $\forall\, i\{1;2;\cdots;n\}\quad T_i \sim N(\mu_i, \sigma_i^2)$

En introduisant Φ_0, fonction de répartition de la variable normale centrée réduite, il vient :

$$\prod_{i=1}^{n} \Phi_0\left(\frac{t_{a,i} - \mu_i}{\sigma_i}\right) \geqslant p \qquad (5)$$

Cette dernière expression est identique, à l'indice final du produit près, à l'équation (1) de la page 52.

Ce que montre cette démonstration c'est que, dans un cas simple, la garde répond à la même préoccupation que l'ordonnancement temporel. En théorie, elle nous assure en régime permanent des garanties similaires à celles d'un système à ordonnancement temporel tout en nous accordant la souplesse nécessaire aux exigences structurelles dynamiques des spécifications. Il faut bien évidemment relativiser ce résultat dès lors que la taille du système dépasse la dizaine de tâches.

3.2 Vers l'ordonnancement temporel

Pour certaines applications, il peut être nécessaire d'atteindre un niveau de sureté plus important sur un ensemble restreint de tâches. La structure de l'application doit nécessairement être localement statique. En effet, toute modification dynamique de l'application conduit à remettre en cause l'ordonnancement établi. De plus, les tâches concernées doivent être isolées du reste de l'application dans des processus spécifiques. Enfin la preuve d'ordonnançabilité est plus accessible dans le cas d'un ordonnancement temporel.

NJN offre cette possibilité de façon très rustique à travers la notion d'intervalle d'exécution (*timeslot*). Il est en effet possible de définir, pour chaque tâche, un intervalle de temps périodique pendant lequel la tâche doit être exécutée.

L'intervalle d'exécution se substitue à la liste de sensibilité. Il est défini par trois paramètres :

- La période T de répétition ;
- L'instant t_b de démarrage ;
- La durée Δt de fin l'intervalle ;

L'origine de l'axe des temps est absolue. Il correspond au 1er janvier 1970 à 0h00 pour toutes les tâches. Chaque tâche est exécutée une fois par période T, après un délai t_b par rapport à l'origine de chaque période. Son exécution doit être achevée Δt après t_b.

Lorsque ce mécanisme de déclenchement est appliqué à l'ensemble des tâches hébergées par un processus logique et que ces tâches sont toutes temps-réel, NJN réalise un ordonnancement temporel. Il est naturellement indispensable d'avoir déterminé le pire temps d'exécution de chacune des tâches pour que le résultat soit garanti.

*
* *

Ce chapitre a montré les fondations théoriques sur lequel repose l'ordonnancement d'NJN. Le dynamisme structurel de l'application rend difficile toute analyse topologique ce qui compromet de facto la mise en œuvre de protocoles conservatifs. NJN met donc en œuvre un protocole de synchronisation de type optimiste. Le principe de synchronisation temps-réel repose sur des objets de synchronisation spécifiques, les tâches temps-réel, dont l'exécution est opérée en fonction de l'horloge murale. Ces éléments sont associés à une latence qui permet de relâcher les contraintes temporelles le long des chemins de données de l'application [93]. Un second type de tâches, les tâches temps-virtuel, se charge d'assurer les traitements entre les entrées et les sorties du système.

Afin d'éviter les retours arrière, néfastes pour le respect des contraintes temps-réel fixées, NJN utilise un mécanisme de garde temporelle. Celui-ci consiste à retarder l'exécution des tâches temps-virtuel de manière à ce que, dans la majorité des cas, l'ensemble des messages qui leur sont destinés soient acheminés avant l'exécution. Un point important restera à étudier. Il s'agit de la stabilité de l'estimateur de garde et des conséquences sur la stabilité du système dans son ensemble.

Lorsque des garanties temps-réel plus fortes sont nécessaires, nous avons vu qu'NJN proposait un mécanisme rudimentaire d'ordonnancement temporel. Même si ce n'est pas là l'intérêt majeur d'NJN, il permet d'intégrer dans l'application un sous ensemble de tâches pour lesquelles le comportement temporel est parfaitement déterministe.

CHAPITRE 3
LE PROTOCOLE DE SYNCHRONISATION

Le chapitre précédent a posé les bases théoriques sur lesquelles peut reposer l'ordonnancement d'NJN. Il nous faut à présent en établir le fonctionnement interne et élaborer les mécanismes mis en œuvre dans le cœur du système : le protocole de synchronisation.

La première partie de ce chapitre aborde la définition du temps. Nous verrons que cette définition découle, entre autres, des propriétés dynamiques des applications.

Le protocole de synchronisation sur lequel s'appuie l'exécutif repose sur la construction d'un graphe causal. Cette structure de données matérialise les relations entre toutes les opérations effectuées au sein de l'application. La seconde partie décrit la structure de ce graphe et la manière dont NJN l'utilise pour gérer les retours arrière.

La dernière partie traite des situations particulières qui font qu'il n'est pas toujours possible de garantir la causalité de l'exécution si l'on veut respecter les contraintes temps-réel (et réciproquement).

1 Des tâches au temps

L'élaboration du temps dans NJN est intimement liée aux problèmes de propagation des données à travers la structure hiérarchique dynamique de l'application. La première section extrait du modèle structurel d'NJN les éléments qui participent à la structure de l'exécutif.

1.1 Du modèle structurel au modèle d'exécution

L'exécutif d'une application NJN se résume essentiellement à des tâches et aux signaux qui les interconnectent, nous avons déjà parlé de ce point brièvement. Les tâches proviennent des éléments structurels de même nom (*RTTask*, *VTTask, ClockBox* et *DecouplingBox*) et les signaux sont les objets qui supportent l'information véhiculée par les variables.

De la structure hiérarchique d'origine, c'est-à-dire des composants, subsistent cependant quelques traces dans l'exécutif, des traces liées aux facultés de reconfigurabilité dynamique d'NJN. La structure de l'application peut en effet évoluer au cours de l'exécution si bien que les connexions entre variables ne sont pas établies de façon statique. Il est par conséquent difficile de réduire le chemin matérialisé par un ensemble de variables connectées entre elles à un seul objet en faisant disparaitre tout aspect structurel au niveau de l'exécutif. Les variables et les ports de communication d'une part et les connexions d'autre part, laissent donc leur empreinte dans le modèle exécutif : chaque variable se traduit par l'existence d'un signal au niveau de l'exécutif ; chaque connecteur associant deux variables fait apparaitre dans l'exécutif une tâche particulière appelée répéteur qui a pour fonction de faire transiter l'information entre les deux signaux correspondants.

Voyons par exemple l'application représentée figure 26. Cinq tâches sont hébergées au sein d'une structure hiérarchique de six composants. Les tâches sont reliées entre elles à travers un réseau de douze variables dont certaines sont des ports d'entrée ou de sortie. La connexion entre T_1 et T_3 par exemple nécessite quatre variables dont l'une est un port de sortie et deux sont des ports d'entrées. Trois connecteurs assemblent ces variables.

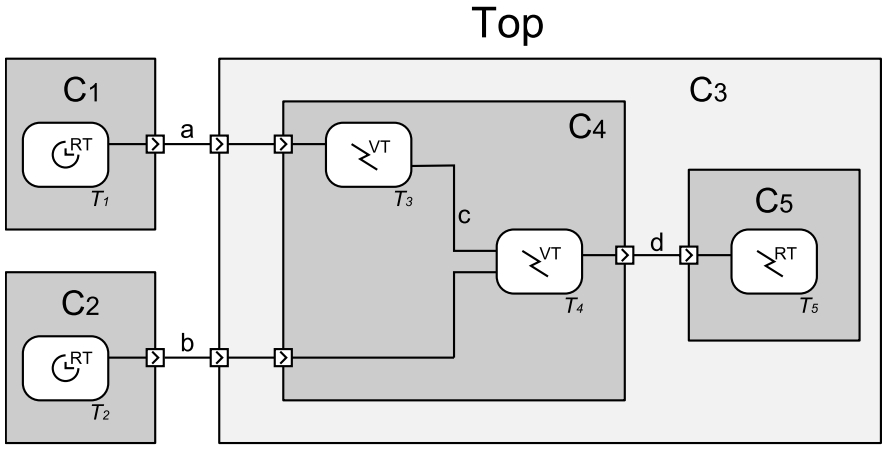


Figure 26. Une application NJN simple.

La figure 27 montre l'exécutif correspondant à l'application de la figure 26. La hiérarchie de composants a disparu. Les quatre variables associant entre elles les tâches T_1 et T_3 ont fait apparaitre quatre signaux reliés par des répéteurs.

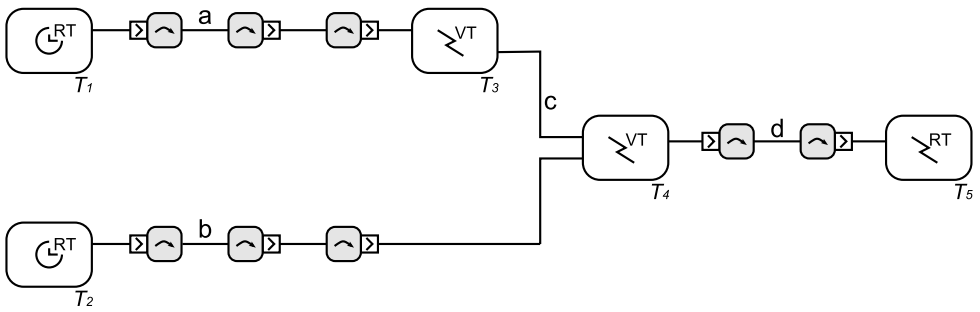


Figure 27. Exécutif correspondant à l'architecture de la figure 26.

Lorsque l'exécutif est distribué, les tâches de l'application, répéteurs compris, sont réparties sur différents processus logiques (figure 28). La structure hiérarchique d'origine joue ici un rôle important : l'élément distribuable dans NJN est en effet le composant. Il en résulte que les chemins de données sont interrompus au niveau des ports des composants distribués. Du point de vue de l'exécutif, c'est aux abords des répéteurs que la distribution va avoir lieu. Le répéteur reste accroché à son signal source. Les données transmises sont adressées au signal cible sous la forme de messages par l'intermédiaire d'un service distant. Ce type de service permet à un processus logique d'émettre des messages ou des requêtes à destination d'un autre processus logique. Une file de messages est associée au service côté émetteur et côté récepteur de manière à assurer la synchronisation des deux processus.

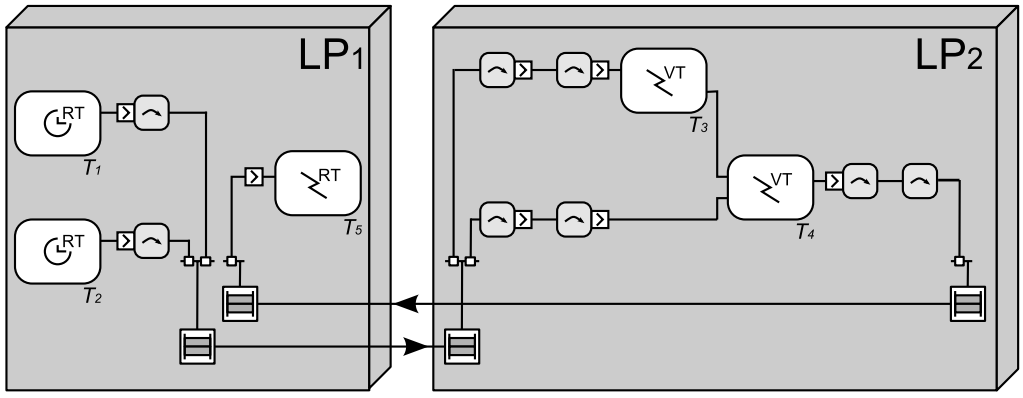


Figure 28. Distribution de l'application de la figure 26 dans deux processus logiques distincts.

La figure 28 montre le résultat de la distribution de notre application sur deux processus logiques. Les composants C_1, C_2 et C_5 ont été placés dans le processus logique LP_1 tandis que le composant *Top* et par conséquent les composants C_3 et C_4 ont été placés dans le processus logique LP_2. La connexion entre les tâches T_1 et T_3 s'en trouve affectée. La cible du premier répéteur est en effet distante. C'est donc à travers un service distant que l'information transitera vers le signal cible.

Notre exemple montre bien les différents éléments que le modèle d'exécution doit gérer. Le moteur d'NJN devra assurer l'ordonnancement des tâches et des

répéteurs mais également assurer l'envoi et l'exécution de requêtes de services distants.

Un élément manque à notre modèle d'exécution, élément nécessaire dans des classes d'applications particulières. Certaines applications séparent en effet l'acheminement de l'information de la synchronisation des opérations. Elles utilisent des signaux spécifiques pour déclencher les différentes tâches de l'application indépendamment des signaux de données. Ces signaux sont appelés horloges et sont placés dans la liste de sensibilité des tâches concernées. Des opérations sur ces horloges peuvent être nécessaires : classiquement multiplication ou division de leur fréquence, fusion entre deux horloges, etc. Afin d'implémenter ces opérations particulières, NJN définit une classe de tâches spécifiques, les boites d'horloges, dont le comportement temporel est à rapprocher des répéteurs. On notera cependant deux différences entre les boites d'horloges et les répéteurs : premièrement, les boites d'horloges sont définies par le programmeur ; deuxièmement, elles peuvent être déclenchées par plusieurs signaux.

1.2 Les problèmes de propagation, de convergence et de causalité : deux cas tordus

Dans NJN, les processus logiques ont pour rôle de supporter la distribution de l'application sur plusieurs cibles matérielles ou d'isoler des sections critiques de code dans des processus. On peut considérer en première approche que chaque processus logique est hébergé par un processeur différent ce qui place le problème de son ordonnancement au second plan. En pratique, le système d'exploitation hôte se chargera de cet aspect de l'exécutif selon des principes d'ordonnancement classiques. Notre problème se situe plutôt au sein même d'un processus logique : gérer l'ordonnancement de l'ensemble des tâches hébergées par un processus.

Comme l'a montré la section précédente, l'exécutif est constitué de tâches et de signaux. On appelle évènement d'exécution l'objet rattaché à l'exécution d'une tâche par le processus logique à une date donnée. Cette tâche, si elle effectue des opérations d'écritures, produit des évènements d'écriture sur les signaux de l'exécutif. Les évènements sont datés de manière à garantir les propriétés causales de l'exécution, la règle de causalité étant la suivante : un évènement d'exécution ne peut produire des évènements d'écriture qu'à des dates qui lui sont strictement postérieures.

Comme nous l'avons vu, afin de gérer convenablement les chaines de causalité, NJN utilise la notion de cycle Δ [58][1]. Il s'agit d'une durée strictement nulle qui établit un ordre partiel des évènements de l'application. Un évènement traité au premier Δ d'une date donnée est antérieur à un évènement exécuté au second Δ de cette même date, etc. L'écart temporel entre l'exécution d'une tâche et ses écritures est arbitrairement fixée à un cycle Δ. Autrement dit, sauf mention explicite contraire au moment de l'écriture, les messages sont postérieurs d'un cycle Δ aux évènements qui les ont produits.

1 Voir page 23.

Ce principe tout-à-fait classique en simulation évènementielle pose un redoutable problème de cohérence temporelle vis-à-vis de la propagation des données à travers les répéteurs et les boites d'horloges.

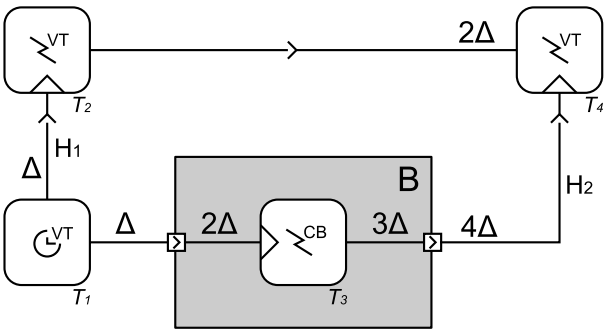


Figure 29. Une application mettant en évidence le problème de cohérence posé par les répéteurs.

La figure 29 met en évidence le problème. Dans cette application, deux tâches de traitement, nommées T_2 et T_4 sont sensées fonctionner un peu à la manière d'un *pipeline*. Un système de deux horloges H_1 et H_2 gère l'activation de deux tâches. Une boite d'horloges T_3, placée dans un composant, génère l'horloge H_2 à partir de l'horloge H_1. Si on accorde à chaque répéteur et chaque boite d'horloges un cycle Δ pour propager le signal qu'il a en charge, un problème de propagation se pose. A l'arrivée, la tâche T_4 reçoit le signal de données 2 deltas avant le signal d'horloge sensé synchroniser le *pipeline*. La conséquence est immédiate : au lieu de traiter les données en décalage, les deux tâches traitent les mêmes données successivement à la même date, avec un décalage de deux Δ.

Ce phénomène est très dommageable. En effet, le comportement de l'application se trouve altéré par sa structure hiérarchique. Pour y remédier, il faut assurer la propagation des données à travers le réseau de signaux et de répéteurs avant tout nouveau cycle d'exécution des tâches classiques. On est donc tenté de supprimer les cycles Δ pour les répéteurs et les boites d'horloges. Un autre problème se pose alors que l'exemple de la figure 30 montre de façon évidente.

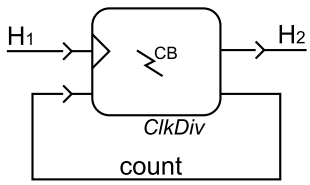


Figure 30. Un diviseur d'horloge.

Il s'agit du cas classique du diviseur d'horloges. A partir d'une horloge primaire H_1, la boite d'horloges *ClkDiv* génère une horloge secondaire H_2 dont le nombre d'évènements par unité de temps est divisé par un facteur entier constant. Pour

compter les évènements de l'horloge primaire, la tâche utilise un signal intermédiaire. Le signal *count* est incrémenté à chaque nouvel évènement constaté sur H_1. Lorsque le compteur est égal à zéro modulo le rapport de division, un évènement est généré sur l'horloge de sortie.

Assurer le fonctionnement d'un tel opérateur sans cycle Δ est tout simplement impossible. Si tel était le cas, le signal *count* serait en effet modifié dans le même cycle que le signal d'horloge H_1 qui a provoqué l'opération. Du point de vue de la tâche, cette mise à jour de *count* est interprétée comme une violation de causalité ce qui provoque un retour arrière. Le temps reste donc figé indéfiniment. Le seul moyen de résoudre ce blocage est d'introduire un cycle Δ entre chaque opération.

Pour lever la contradiction mise en évidence par ces deux exemples, NJN introduit deux cycles d'exécution. Le premier, appelé *micro-delta* (δ), assure la propagation et la convergence des signaux à travers les répéteurs et les boites d'horloges. Le second, appelé *delta* (Δ), est associé à l'exécution des tâches de traitement classiques. La section suivante élabore le modèle temporel d'NJN basé sur ce principe de double cycle.

1.3 De la définition du temps

NJN définit trois dates courantes : une date de lecture t_R, une date d'exécution t_E et une date d'écriture t_W. La date d'exécution correspond à la date de l'évènement en cours de traitement. La date de lecture est antérieure ou égale à la date d'exécution. En pratique, ces deux dates sont le plus souvent égales. La date d'écriture quant à elle est strictement postérieure aux deux autres. La relation ci-dessous est donc vérifiée à chaque instant :

$$t_R \leqslant t_E < t_W$$

Lorsqu'un évènement est traité, l'état observable du système est celui défini à la date de lecture. Il est éventuellement possible de connaitre l'état qu'avait un signal à une date antérieure. En revanche, aucune lecture à une date postérieure n'est possible. De façon analogue, les écritures sur les signaux de l'application sont effectuées à la date d'écriture. Il est éventuellement possible de retarder une écriture en précisant un délai explicite. Mais toute tentative d'écriture à une date antérieure est prohibée[1].

Une tâche est composée de deux éléments : un corps qui renferme le code exécutable de la tâche et des conditions d'activation. Les conditions d'activation peuvent être de nature temporelle. On peut par exemple vouloir qu'une tâche soit exécutée à intervalles de temps réguliers (*timeslot*) ou au bout d'un certain temps d'inactivité (*timeout*). Les conditions d'activation peuvent également prendre la forme d'une liste de sensibilité qui énonce l'ensemble des signaux du système qu'il faut surveiller. L'exécution du corps de la tâche sera alors programmé chaque fois qu'une modification affectera l'un des signaux de la liste.

Les règles d'exécution des tâches sont les suivantes : lorsque les conditions d'activation d'une tâche sont remplies, un évènement d'exécution est programmé.

1 Nous verrons un peu plus loin qu'il existe une exception avec les boites de découplage.

La date associée à cet évènement dépend de la nature des conditions d'activation. S'il s'agit d'un déclenchement temporel, la datation de l'évènement est immédiate. Si le déclenchement est dû à un évènement d'écriture de l'un des signaux de la liste de sensibilité, la date d'activation correspond à la date de l'évènement d'écriture[1]. Lorsque l'évènement d'exécution est traité, c'est-à-dire lorsque la tâche est exécutée, la date d'exécution t_E est fixée à la date de l'évènement.

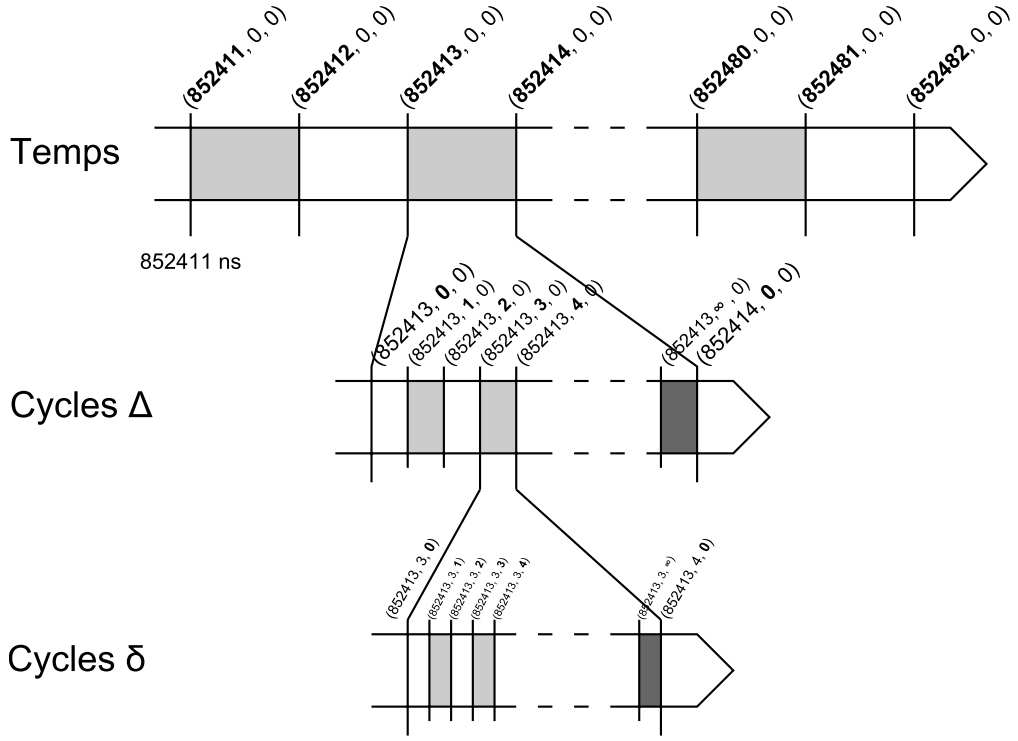


Figure 31. Modèle temporelle d'NJN.

Deux cycles sont définis dans NJN (figure 31) : les *delta,* notés Δ et les *micro-delta* notés δ. La relation d'ordre entre deux dates $(t_1, \Delta_1, \delta_1)$ et $(t_2, \Delta_2, \delta_2)$ est définie de la façon suivante : Si t_1 est égal à t_2 alors la relation d'ordre est établie en comparant Δ_1 et Δ_2. Si ces derniers sont également identiques, c'est la relation entre δ_1 et δ_2 qui est examinée. L'écart temporel entre t_E et t_W dépend alors de la nature de la tâche. Pour une tâche classique (*RTTask* ou *VTTask*), il est d'un Δ. Pour un répéteur ou une boite d'horloges (*ClockBox*), il est d'un δ. Les cycles Δ ne sont incrémentés que lorsque les cycles δ ont permis de propager les données à travers le réseau de signaux.

Le tableau II résume les relations temporelles entre les trois dates définies par NJN en fonction du type d'évènement traité.

1 Nous verrons que pour les tâches temps-réel un délai est introduit entre l'évènement d'écriture et l'évènement d'exécution qu'il produit.

Tableau II. Relations temporelles entre les dates de lecture, d'exécution et d'écriture en fonction du type de tâche exécuté.

Nature de la tâche	Lecture	Exécution	Écriture
VTTask	$(t_E, \Delta_E, +\infty)$	$(t_E, \Delta_E, +\infty)$	$(t_E, \Delta_E + 1, 0)$
RTTask	$(t_E - L, +\infty, +\infty)$	$(t_E, \Delta_E, +\infty)$	$(t_E, \Delta_E + 1, 0)$
ClockBox, Repeater	$(t_E, \Delta_E, \delta_E)$	$(t_E, \Delta_E, \delta_E)$	$(t_E, \Delta_E, \delta_E + 1)$

On note que les tâches classiques (*VTTask*) observent l'état des signaux après relaxation du réseau ($\delta = +\infty$) et placent les messages qu'ils produisent au début d'un nouveau cycle Δ ($\delta = 0$). Entre un cycle Δ et le suivant, les répéteurs et les boites d'horloges (*ClockBox*) assurent la propagation des messages en gérant les relations causales au moyen de cycle δ.

Les tâches temps-réel (*RTTask*) observent le monde avec retard en repoussant en arrière la date de lecture d'une durée égale à leur latence L. On peut noter que cette observation est faite sur un passé relaxé. Cycles Δ et δ sont en effet à une valeur infinie. L'écriture est réalisée quant à elle au cycle Δ suivant, comme cela est le cas pour les tâches temps-virtuel.

Le cadre temporel défini ci-dessus permet d'établir des relations causales entre évènements tout en assurant la convergence de la propagation des données à travers l'application avant que les tâches soient exécutées. Si l'on veut garantir la causalité de l'ensemble de l'application, les évènements devront être traités en respectant l'ordre partiel établi par leur date, en apparence du moins. Cette responsabilité est celle du protocole de synchronisation, objet des sections suivantes de ce chapitre.

2 Le graphe causal et le protocole de synchronisation

Le protocole de synchronisation choisi est de type optimiste. Il est donc nécessaire de mettre en œuvre une sauvegarde de l'état du système ainsi qu'un mécanisme de restauration. Notre approche fait appel à une technique d'annulation directe inspirée de Zhang [92]. Elle ne fait pas appel à une sauvegarde globale de l'état du système mais à des sauvegardes atomiques. Toutes les opérations d'écriture et de lecture effectuées par les tâches d'un processus logique sont archivées de façon atomique. Les relations entre les différentes opérations sont également représentées dans la structure de données du système. De cette manière il est possible, en cas de retour arrière, d'annuler une opération quelconque ainsi que ses conséquences. Il n'est pas nécessaire d'annuler l'ensemble des opérations du système. Seules les opérations réellement affectées par une violation de causalité sont annulées.

2.1 Topologie du graphe causal

Toutes les actions entreprises dans NJN donnent naissance à des évènements reliés entre eux par des relations matérialisant les liens de cause à effet. Trois types d'évènements sont représentés dans le protocole (figure 32) :

- Un évènement d'exécution représente l'exécution d'une tâche à une date donnée.
- Un évènement d'écriture représente la modification de l'état d'un signal par une tâche à une date donnée.
- Un évènement de lecture représente l'accès, par une tâche, au contenu d'un signal à une date donnée.

Au cours de l'exécution de l'application, un graphe de dépendance appelé graphe causal, est construit. Il matérialise les relations de cause à effet entre les différents évènements du système. Les règles de construction de ce graphe reposent à la fois sur les relations de causalité entre évènements et sur leurs relations temporelles. Les règles de construction du graphe sont les suivantes :

Tout d'abord, chaque évènement appartient à une file ordonnée selon la date des évènements. On distingue trois types de listes :

- La liste des évènements d'écriture est représentée tout naturellement par le signal affecté par ces écritures.
- Une file d'exécution renferme la liste des évènements d'exécution se rapportant à une tâche donnée.
- Une file de lecture renferme la liste des évènements de lecture relatifs à un évènement d'écriture – autrement dit relatif au message qui lui est associé.

Ensuite, les relations de cause à effet sont représentées par des liens entre évènements. Lors de son exécution par exemple, une tâche consulte l'état de certains signaux et en modifie d'autres. Un évènement d'exécution pointe donc vers l'ensemble des évènements de lecture et d'écriture dont il est à l'origine. De même, l'apparition d'un message sur un signal active les tâches qui référencent ce signal dans leur liste de sensibilité. Un évènement d'écriture pointe donc vers les évènements d'exécution qu'il provoque.

Enfin, le graphe causal gère les relations distantes entre évènements. Il s'agit concrètement de créer un lien représentant l'action d'un répéteur sur son signal cible, y compris lorsque ce dernier n'est pas localisé sur le même processus logique. Un lien entre un évènement d'exécution et un évènement d'écriture peut donc être un lien local ou distant selon le contexte.

Pour garantir l'aspect causal de l'application, il est nécessaire de prendre certaines précautions avec les évènements simultanés. Par exemple, si deux évènements d'écriture provoquent l'activation d'une même tâche à une même date, un seul évènement d'exécution est créé. Les deux évènements d'écriture à l'origine de l'activation y feront référence.

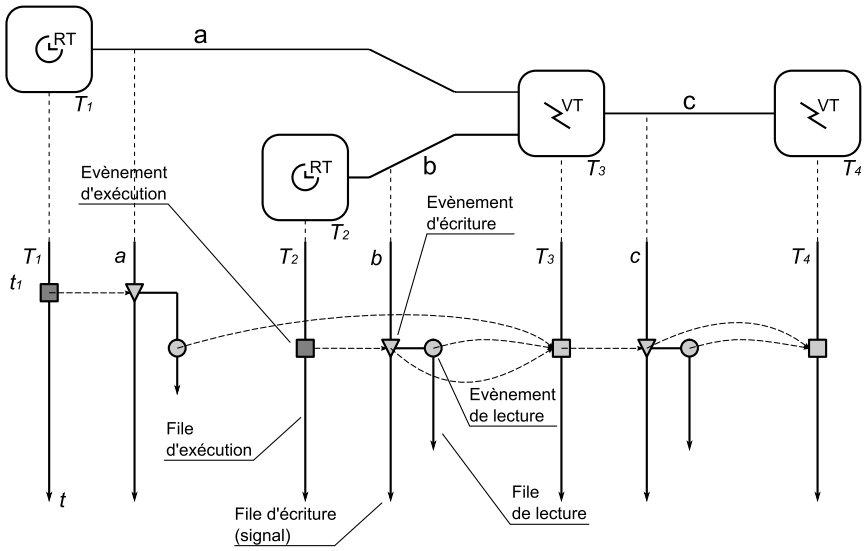


Figure 32. Le graphe causal.

Concernant les évènements d'écriture, les choses sont un peu plus délicates. Sans précaution, si deux tâches tentent d'écrire des valeurs différentes sur un même signal au même instant, la dernière écriture inscrite physiquement sur le signal risque de l'emporter. Et rien n'indique que ce soit la même tâche qui l'emporte à tous les coups lorsque la scène est rejouée.

Pour éviter tout risque, une bonne démarche consiste à ne jamais tenter le diable. Sauf qu'à la faveur d'une modification structurelle, le cas peut facilement se présenter. Par exemple, lorsqu'on substitue une tâche par une autre, une zone de recouvrement temporelle peut subsister si la substitution n'est pas réalisée convenablement[1].

Le mécanisme employé consiste à analyser l'ensemble des évènements inscrits au cours d'un même cycle Δ (ou δ selon le type de tâche) et de signaler la probable perte de causalité si on constate des valeurs différentes parmi les évènements concernés.

2.2 Le retour arrière

Dans l'algorithme de Jefferson, l'origine du retour arrière provient d'une violation de causalité dans la lecture d'un message. Dans notre approche, cette violation est facilement identifiable en analysant le graphe causal.

Lorsqu'un évènement d'écriture est inséré dans un signal, NJN consulte la file de lecture associée à l'évènement d'écriture immédiatement précédent (figure 33). Si cette file contient des évènements antérieurs à l'écriture insérée, alors les lectures en question n'ont pas été faites sur la bonne information. Il y a donc violation de causalité.

1 Voir le problème de la continuité de services page 115

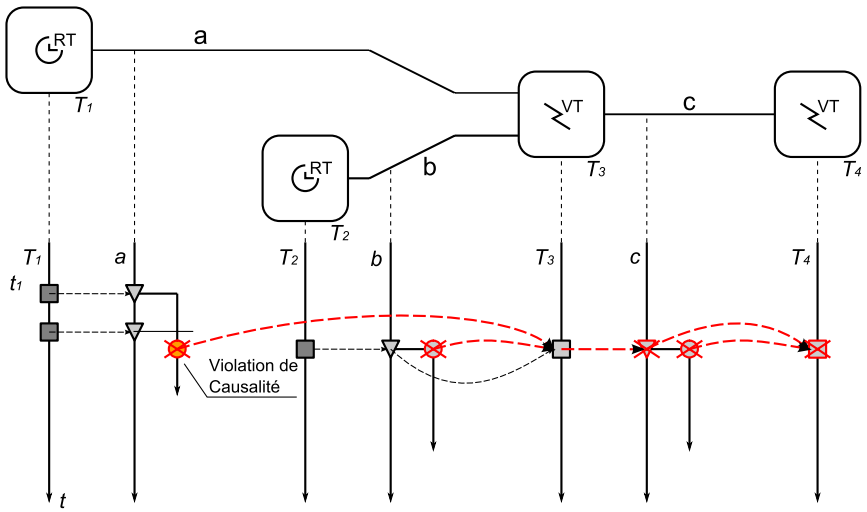


Figure 33. Un graphe causal. Les rectangles représentent les évènements d'exécution, les triangles les évènements d'écriture et les ronds les évènements de lecture.

NJN détache du graphe causal les évènements de lecture concernés. Le graphe est ensuite parcouru en partant de ces évènements. On retrouve ainsi l'ensemble des évènements de lecture, d'écriture et d'exécution affectés par la violation temporelle. Ces évènements sont tous supprimés ou réinitialisés suivant les cas, ainsi que les sous-graphes qui en dépendent.

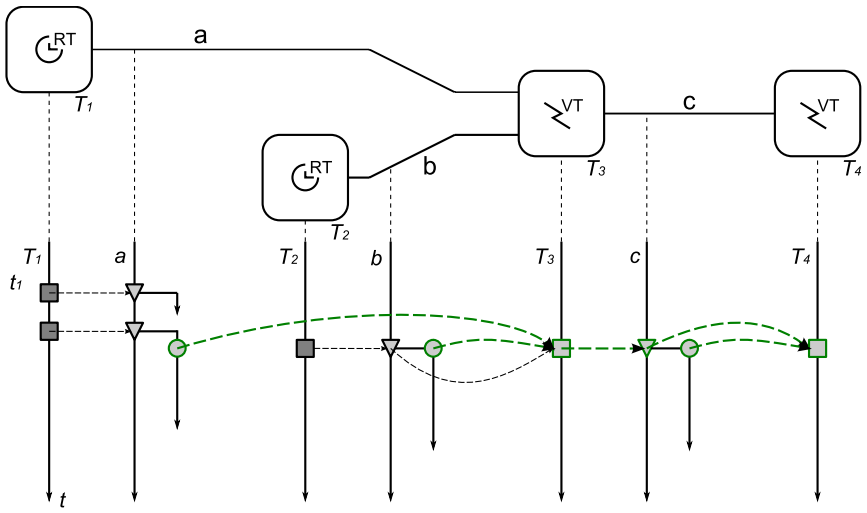


Figure 34. Le graphe causal de la figure 33 après retour arrière.

Les évènements d'écriture et de lecture sont purement et simplement supprimés de la structure de données. En revanche, un traitement particulier est opéré sur les évènements d'exécution. L'exécution en elle-même, si elle à déjà eu lieu, est évidemment remise en cause. Avec elle sont annulés les évènements d'écriture et de lecture qu'elle a engendrés. Mais l'évènement d'exécution lui-même n'est pas nécessairement supprimé de la structure de donnée. Il se peut en effet que deux

évènements simultanés soient à l'origine du déclenchement. Dans ce cas, l'évènement subsiste et sera à nouveau ordonnancé en vue d'une ré-exécution.

Lorsque le graphe causal référence un évènement distant, un message d'annulation est construit et envoyé au processus logique sur lequel l'évènement à supprimer est localisé.

Un fois toutes ces opérations effectuées, le système retrouve un état satisfaisant et le traitement des évènements d'exécution peut reprendre normalement (figure 34).

2.3 Nettoyage de la mémoire

Pour qu'une application NJN fonctionne dans de bonnes conditions, le temps virtuel universel (WVT), c'est-à-dire la date d'exécution courante des tâches non temps-réel, doit être positionné dans un intervalle de temps majoré par l'horloge murale (WCT) et de longueur égale à la latence de la chaine de traitement en cours d'évaluation. Concrètement, si une tâche temps-virtuel est en cours d'exécution et qu'elle participe d'un traitement pour lequel la tâche temps-réel de sortie à une latence L, on doit avoir nécessairement :

$$WCT - L < WVT < WCT$$

sans quoi il sera impossible de satisfaire la latence de la tâche de sortie.

Cette particularité de fonctionnement facilite la détermination de la limite des objets fossiles (FVT) qui permet de savoir quels évènements peuvent être supprimés de la structure de données.

En supposant que le système satisfasse les contraintes temps-réel, autrement dit que tous les processus temps-réel du système soient exécutés à temps, il suffit de connaitre la latence la plus grande pour déterminer le FVT. Par exemple, si l'on adopte une approche globale, soit R l'ensemble des tâches temps-réel du système, le FVT est tel que :

$$FVT = WCT - max_{T \in R}\left(L_T \right)$$

La latence maximale trouvée représente alors le délai de fossilisation des évènements, c'est-à-dire le temps nécessaire pour que ces objets soient devenus sans intérêt pour le système.

Chaque fois qu'un paramètre de latence est modifié, qu'une nouvelle tâche temps-réel est ajoutée ou qu'une tâche temps-réel est supprimée de l'application, le délai de fossilisation est réévalué. Mais NJN procède de façon locale en tenant compte des dépendances de données entre processus logiques. On détermine d'abord le maximum de la latence en considérant les tâches temps-réel hébergées localement par chaque processus logique. Cette valeur est ensuite propagée aux processus logiques avec qui des messages sont échangés. Chaque processus compare ensuite les valeurs reçues avec la latence maximale trouvée localement et conserve la plus grande valeur.

Localement, une structure de type *heap* permet de trouver L_{max} la latence maximale des tâches hébergées localement.

L'exemple de l'application distribuée de la figure 35 montre comment NJN procède, pour déterminer le délai de fossilisation *FT*,

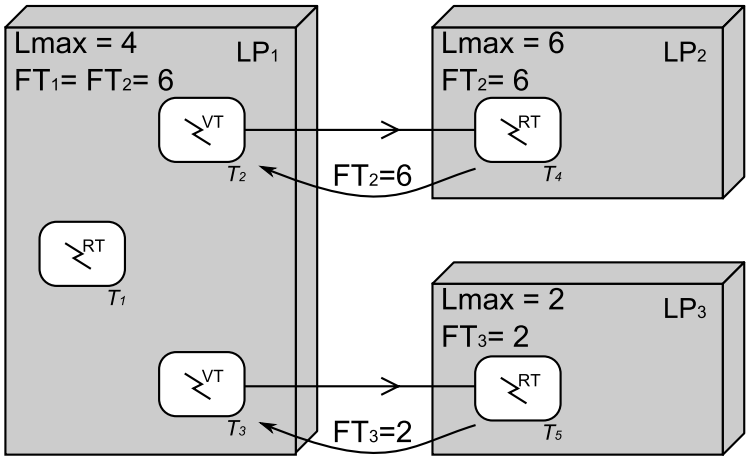


Figure 35. Détermination du délai de fossilisation.

Trois processus sont représentés : LP_1, LP_2 et LP_3. Les tâches du processus LP_1 produisent des données à destination de tâches hébergées par LP_2 et LP_3. Pour déterminer son délai de fossilisation, LP_1 devrait donc tenir compte des délais de fossilisation trouvés pour LP_2 et LP_3. En revanche, ces deux derniers n'ont pas besoin du FT de LP_1.

LP_2 et LP_3 propagent vers LP_1 leur délai de fossilisation, en indiquant la provenance de ce dernier, c'est-à-dire l'identifiant du processus logique qui héberge la tâche à partir de laquelle a été déterminé ce délai. Le processus LP_1 déterminera alors le délai *FT* en comparant la valeur locale dont il dispose et les valeurs reçues des processus LP_2 et LP_3 dont il dépend.

Les choses se compliquent un peu lorsque les processus logiques sont liés par des dépendances circulaires. NJN procède alors comme dans le cas précédent, à ceci près que chaque processus logique ignore les délais de fossilisation dont il est l'auteur au profit de la latence locale maximale. La figure 36 présente un tel cas de figure. Le processus LP_1 héberge la tâche de latence maximale (36.a). Celle-ci est propagée aux deux autres processus logiques (36.b).

Lorsque LP_1 voit sa valeur L_{max} modifiée (figure 37.a), une nouvelle diffusion des délais de fossilisation est engagée. La valeur de *FV* associée à LP_1 ayant pour origine LP_1 lui-même, ce dernier ne diffuse pas sa valeur de *FV* mais la valeur de sa latence maximale L_{max}. Rapidement, la diffusion met à jour la valeur des délais de fossilisation des différents processus (37.b).

Le comportement temps-réel d'NJN n'étant pas totalement garanti, la limite des objets fossiles est calculée avec une marge de sécurité. Si *FT* est le délai de fossilisation déterminé localement, FVT est tel que :

$$FVT = WCT - (1+k)\,FV$$

où *k* représente la marge relative de sécurité dont la valeur est de l'ordre de 1, ce qui permet grosso modo à chaque tâche de subir un retour arrière et d'être ré-exécutée avant que le système ne soit pris en défaut par la suppression trop hâtive d'un évènement.

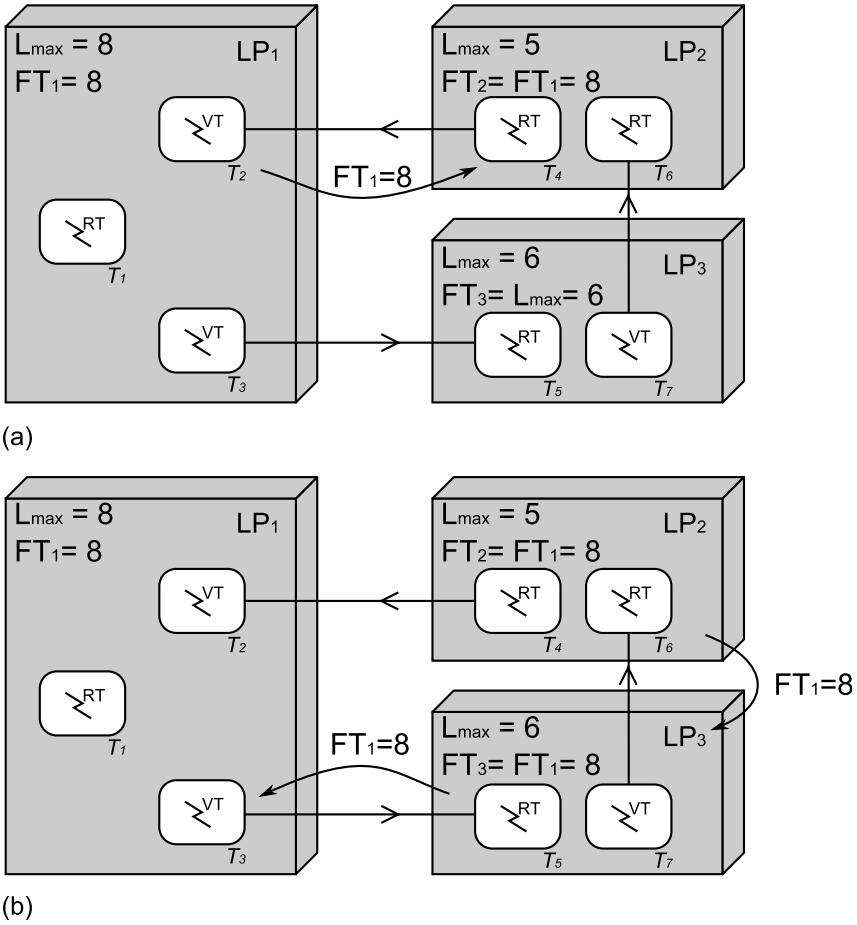


Figure 36. Cas d'une dépendance circulaire.

Le nettoyage de la structure de données est une opération de priorité très faible. Elle est effectuée lorsqu'aucune tâche ne peut être exécutée. Les évènements d'écriture dont la date est antérieure à FVT sont supprimés du système. Chaque suppression d'un évènement d'écriture s'accompagne de la destruction de la file de lecture qui lui était associée ainsi que des évènements d'exécution qu'il a déclenchés.

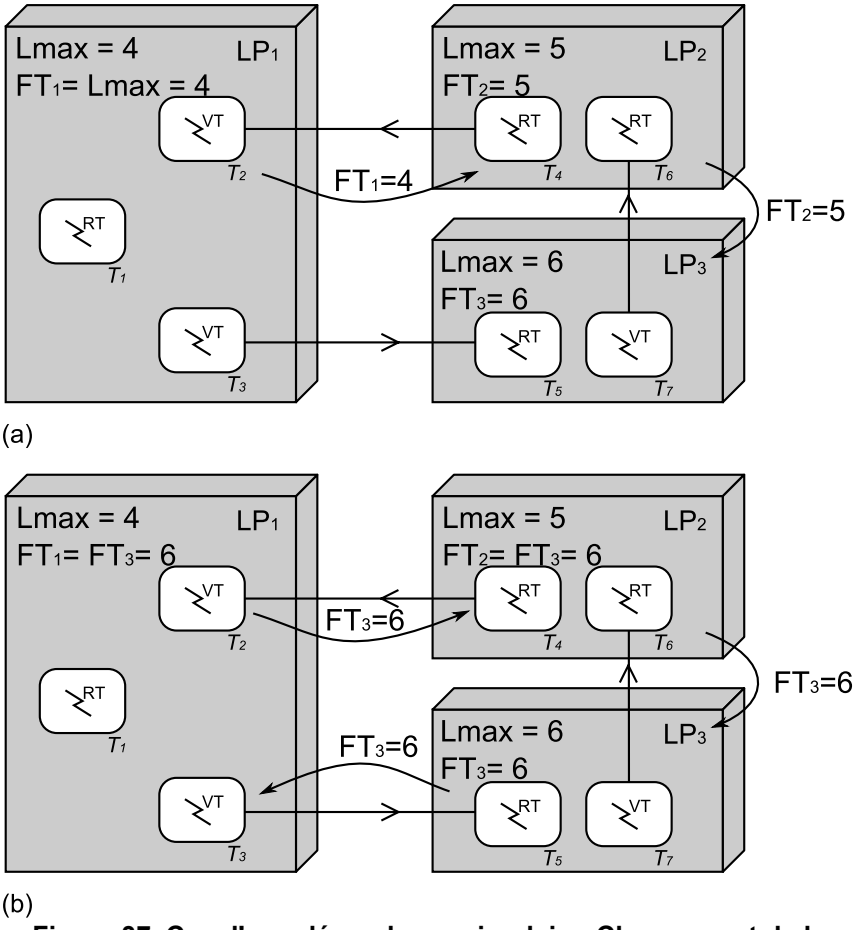


Figure 37. Cas d'une dépendance circulaire. Changement de la latence maximale de LP_1.

3 Quand rien ne va plus...

La précédente partie a dévoilé le cœur du protocole de synchronisation d'NJN. Lorsque l'application est convenablement construite et qu'un régime de fonctionnement stable est établi, ce protocole assure la reproductibilité comportementale du système. Les choses ne sont cependant pas toujours idéales et NJN doit faire face, comme tout système temps-réel, à des situations exceptionnelles ou des configurations plus ou moins tordues. Cette troisième partie se penche sur ces cas particuliers où les principes généraux d'NJN sont mis en défaut. On montrera alors comment, et avec quels outils, NJN tente de s'adapter pour assurer l'essentiel.

3.1 Découpler les niveaux hiérarchiques de l'application

Schématiquement, une application NJN est une construction hiérarchique dont les différents niveaux n'ont en commun ni les supports matériels, ni les contraintes. Au plus profond de la hiérarchie, les opérateurs gèrent les interactions avec les

éléments physiques du système – capteurs et actionneurs – et sont soumis à des contraintes temps-réel très contraignantes. A ce niveau, la distribution est relativement figée. Elle est réalisée sur des cibles dédiées généralement de type micro-contrôleur. Les réseaux employés ont des latences bornées et relativement faibles. Au plus haut de l'application, la gestion est plus globale et vise à orienter le comportement général des couches inférieures. La structure est distribuée sur des serveurs et les communications font appel à des réseaux, éventuellement sans fil, dont la latence est plus aléatoire et non-bornée.

Entre ces différents niveaux, les échelles de temps ne sont donc pas les mêmes. Dans une telle structure, assurer la causalité de l'ensemble de l'application entraine un couplage temporel entre les différents niveaux hiérarchiques du système qui peut être très dommageable aux contraintes temps-réel des couches basses de l'application.

Pour bien comprendre le problème, examinons le cas d'une décision prise au plus haut niveau hiérarchique. Pour parvenir aux couches basses de l'application, cet ordre va nécessiter un temps d'acheminement relativement long causé par les réseaux empruntés. Le retard produit à l'arrivée va engendrer localement un retour arrière conséquent qui risque de mettre en péril le respect des contraintes temps-réel locales.

De façon générale, plus le système est gros, plus les latences associées aux tâches temps-réel doivent être conséquentes pour prendre en compte ce phénomène de couplage. Or, d'un point de vue strictement fonctionnel, cette relation entre taille du système et latence des opérations de contrôle est contre-nature. Il est donc impératif de pouvoir découpler les différents niveaux hiérarchiques en délimitant les zones où la causalité est impérative.

C'est le rôle joué par des tâches particulières appelées « boites de découplage » (*DecouplingBox*) dont la particularité est d'autoriser la re-datation des évènements qu'elles produisent. Une boite de découplage comporte des signaux d'entrée et de sortie. Les dates associées aux évènements de sortie peuvent être fixées arbitrairement si bien que la relation causale entre les évènements consommés et produits n'est pas conservée. La source de temps employée pour dater les évènements provient en général de l'horloge murale mais peut provenir d'informations extraites des données consommées elles-mêmes. Comme les relations causales entre données sont ignorées, les retours arrière n'ont plus véritablement de sens. Les boites de découplage les ignorent donc.

Pour être efficaces, de telles tâches doivent être placées sur les chemins de données descendant de la hiérarchie, dans les niveaux les plus bas, avant que les paramètres de commande concernés ne soient pris en compte.

3.2 Respecter la latence des tâches temps-réel

Localement, l'objectif d'NJN est de garantir au mieux le respect de la latence des tâches temps-réel. Pour ce faire, il s'appuie sur deux éléments : l'attribution d'un

niveau de priorité aux tâches et une estimation statistique rudimentaire de leur durée d'exécution.

En fonctionnement normal, NJN donne priorité aux tâches temps-réel. Les deux types de tâches sont traités selon deux files d'évènements séparées. Lorsqu'une tâche temps-réel arrive à échéance, elle est exécutée. Si aucune n'est dans ce cas de figure, NJN tente de traiter les tâches temps-virtuel. Connaissant l'échéance de la prochaine tâche temps-réel et l'estimation de la durée d'exécution de la prochaine tâche temps-virtuel, NJN détermine si l'échéance de la première laisse suffisamment de temps à la seconde pour être traitée. Le cas échéant, la prochaine tâche temps-virtuel est exécutée. Dans le cas contraire, NJN se place en attente de l'échéance de la prochaine tâche temps-réel.

Il peut arriver que deux tâches temps-réel doivent être exécutées simultanément par le même processus logique. Il est alors impossible de respecter la latence des deux tâches. C'est dans cette situation que le niveau de priorité attribué aux tâches est utilisé. NJN choisit naturellement d'exécuter la tâche de priorité la plus haute en premier.

En cas de surcharge ou de modification structurelle importante, NJN peut se trouver dans une situation où il n'est plus en mesure de respecter la latence des tâches temps-réel. Là encore, une stratégie basée sur les niveaux de priorité est mise en œuvre. Concrètement, si des dépassements de latence importants sont constatés sur les tâches temps-réel, NJN cherche à gagner du temps en ignorant l'exécution des tâches de niveau de priorité faible. Le seuil de priorité, qui fixe le niveau en dessous duquel les tâches sont ignorées, est remonté tant qu'une situation satisfaisante n'est pas retrouvée.

Bien évidemment, l'usage récurrent de cette stratégie conduit à un fonctionnement anormal de l'application. Il faut garder à l'esprit que l'apparition chronique des dépassements de latence témoigne d'un mauvais dimensionnement de l'application, d'une mauvaise conception ou d'une mauvaise répartition de la charge qu'aucune stratégie ne saurait combler.

3.3 Fiabiliser l'exécution des tâches critiques

Les tâches hébergées par un même processus logique ne sont pas exécutées de manière isolée. D'un point de vue pratique, si l'exécution d'une tâche provoque une erreur grave, c'est l'ensemble des tâches hébergées qui est touché car le processus logique dans son ensemble risque d'être interrompu.

Pour contourner cette difficulté, une solution consiste à isoler les tâches dont le fonctionnement n'est pas éprouvé dans des processus logiques séparés. Si une telle tâche vient à provoquer une erreur sérieuse, le processus qui l'héberge sera bel et bien détruit mais il n'affectera pas les autres tâches de l'application.

Parallèlement, on peut mettre en place facilement une surveillance du fonctionnement d'un processus logique et identifier ses arrêts de fonctionnement pour relancer les éléments défaillants. La surveillance est réalisée en plaçant une tâche à déclenchement périodique dans le processus logique à surveiller. Cette

tâche génère à chaque exécution un évènement sur un signal spécifique appelé signal de battement. Une seconde tâche, placée dans un processus logique différent, est chargée de surveiller ce signal de battement. Elle prendra en charge la reconstruction de l'application si aucun évènement ne se produit sur le signal pendant une durée supérieure à la périodicité de la tâche surveillée. Un nouveau processus logique sera créé dans lequel les composants détruits seront ré-instanciés.

Cette stratégie d'isolement conduit à multiplier les processus logiques. Or, ces derniers sont plus lourds à gérer que de simples tâches. Cette méthode ne peut donc être employée qu'avec parcimonie.

C'est pourquoi NJN propose une autre stratégie basée sur la gestion d'exceptions par sauts longs (*setjump* et *longjump*). Il n'est plus nécessaire d'isoler les tâches dangereuses dans des processus spécifiques. Cependant, cette technique ne prémunit pas de toutes les défaillances d'exécution. Pour conserver en toutes circonstances une structure de données cohérente, NJN procède à l'exécution en deux temps.

La première phase commence par une sauvegarde de l'état de la pile du système[1]. Le corps de la tâche est ensuite exécuté. Cette phase est la plus critique car les erreurs d'exécution qui peuvent apparaitre à cet instant découlent principalement du code de l'application, c'est-à-dire d'une erreur de conception d'un composant ou d'une utilisation non-conforme de celui-ci. Aucune modification de la structure de donnée d'NJN n'est opérée pendant cette phase ce qui préserve son intégrité. NJN se contente de lister les opérations appelées par la tâche. Si une erreur grave se produit pendant cette phase, l'état de la pile du système est restitué[2] et la liste des opérations est détruite. L'échec d'exécution est signalé et NJN passe à l'exécution de la tâche suivante.

La seconde phase est engagée si l'exécution s'est passée sans encombre. Toutes les opérations listées sont effectivement opérées sur la structure de données d'NJN. Les erreurs qui peuvent se produire pendant cette seconde phase sont plus rares car elles découlent plus vraisemblablement d'un bogue dans le code d'NJN que d'une erreur de l'application. Par ailleurs, chaque opération peut être effectuée sous un contrôle beaucoup plus étroit que pendant la première phase.

*
* *

Les propriétés dynamiques d'NJN ont conduit à une définition du temps en trois dimensions. La première correspond à la représentation commune du temps. Les deux suivantes, les cycles Δ et δ, assurent la préservation des relations causales entre les données pendant les phases de relaxation du système. Les cycles Δ sont relatifs à l'exécution des tâches tandis que les δ permettent la propagation des données à travers le réseau de signaux. Cette représentation du temps permet d'établir les relations temporelles entre les différents évènements de l'application.

1 Concrètement, il s'agit d'un appel à *setjump*.
2 Par un appel à *longjump*.

Trois types d'évènements sont définis : les évènements d'écriture et de lecture qui représentent les opérations de même nom effectuées par les tâches sur les signaux et les évènements d'exécution qui représentent l'exécution d'une tâche. Ces évènements sont liés par une structure de données, appelée graphe causal, qui matérialise leurs relations de cause à effet. Ce graphe, construit au fur et à mesure de l'exécution, conserve la trace de toutes les opérations effectuées par l'application. En cas de nécessité, il permet de réaliser les retours arrière.

Nous avons à présent terminé la construction du modèle d'exécution d'NJN. Bien qu'étant évènementiel, notre modèle produit des applications dont le comportement peut être parfaitement déterministe. Le principe d'ordonnancement qu'il met en œuvre tranche radicalement avec les techniques évènementielles traditionnelles. Il s'inspire des techniques de simulation distribuées qui traitent nativement les relations causales entre les évènements du système. C'est cette particularité qui confère à notre modèle ses propriétés déterministes. Deux types de protocoles sont classiquement employés dans les simulateurs : les premiers sont dits conservatifs, les seconds optimistes. Ces derniers donnent aux processus logiques de l'application une grande souplesse d'exécution. Les situations de violation de causalité sont détectés à postériori. Lorsque cela est nécessaire, un mécanisme de retour arrière est mis en œuvre pour retrouver une état du système cohérent. Le protocole de synchronisation utilisé par NJN est de ce type et lui permet de s'adapter aux modifications structurelles opérées sur le système pendant l'exécution sans qu'il soit nécessaire d'effectuer d'analyse topologique de l'application.

Dans cette partie, nous avons raisonné sur le modèle structurel rudimentaire élaboré en introduction. Il est en réalité beaucoup plus complexe. La partie suivante en décrira tous les détails.

Deuxième partie
Un modèle structurel dynamique distribué

CHAPITRE 4
COMPOSANTS, TÂCHES, VARIABLES – LE MODÈLE STRUCTUREL

Le modèle structurel d'NJN a été choisi pour faciliter le développement rapide d'applications distribuées. Il est orienté composant. Ce qui signifie qu'une application NJN est une construction hiérarchique élaborée à partir de composants standards, disponibles dans des bibliothèques. Les composants hébergent des tâches et des variables, seuls éléments qui laisseront leur marque dans l'exécutif. A ce modèle de base s'ajoutent quelques objets plus complexes à appréhender, les méthodes et les canaux, destinés à structurer les interconnexions ou à abstraire les communications au sein de l'application.

Ce chapitre expose les différents objets présents dans le modèle structurel d'NJN. Après une première partie générale où quelques repères sont posés, chaque famille d'objet est présentée avec ses caractéristiques essentielles. Le point de vue adopté est celui du développeur d'application. Le rôle des objets et leur comportement est décrit en détail. Quelques exemples de code viennent illustrer de façon concrète le propos.

1 Des composants qui hébergent des variables et des tâches

Une application NJN est représentée par l'instance de composant de plus haut niveau dans la hiérarchie structurelle. Cette instance, souvent appelée « *top* », est elle-même composée d'instances d'autres composants interconnectées entre elles

par des variables et des ports d'entrées/sorties. Les composants situés tout en bas de la hiérarchie, hébergent des tâches qui portent le code actif de l'application.

1.1 Deux familles d'objets : les conteneurs et les terminaux

Les composants, les canaux ou les vecteurs sont appelés des conteneurs et peuvent héberger d'autres objets. Les autres classes d'objets sont des éléments terminaux de la hiérarchie. On trouve dans cette seconde catégorie les variables, les tâches, etc.

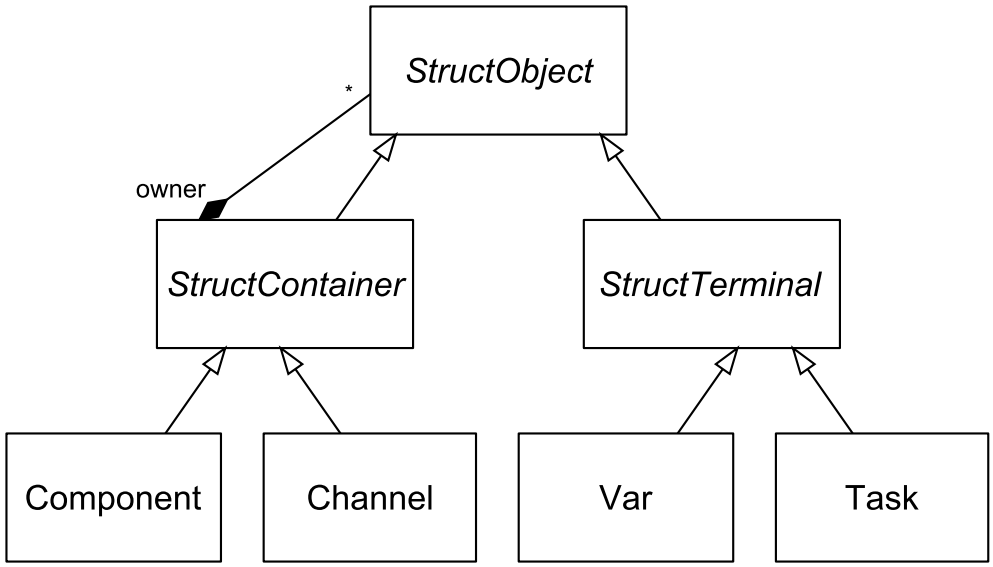


Figure 38. Diagramme de classe simplifié du modèle structurel d'NJN.

Certains éléments de cette topologie rudimentaire se déclinent en plusieurs variantes qui cachent des différences de comportement ou de caractéristiques.

Enfin, chaque classe d'objet est associée à un modèle qui lui est spécifique, un *pattern*, qui renferme les informations nécessaires à la création de l'objet. Les *patterns* sont rangés dans des paquetages et des bibliothèques. Nous reviendrons dans le chapitre suivant sur ces trois dernières constructions.

1.2 Des propriétés et des paramètres communs

Tous les objets du modèle structurel ont en commun trois paramètres et deux propriétés :

- Le paramètre *owner* permet de remonter au conteneur père d'un objet.
- Le paramètre *pattern* fournit une référence sur le modèle qui a permis de construire l'objet.
- Le paramètre *package* fournit quant à lui une référence sur le paquetage auquel appartient le modèle précédant.

- La propriété *name* correspond au nom donné à l'objet au moment de sa création. Il s'agit d'une chaine de caractères quelconque qui peut changer au cours de la vie de l'objet. Cette propriété peut être consultée et modifiée.
- La propriété *lifeline* renseigne sur l'activité de l'objet. Une valeur *nil* indique que l'objet n'existe pas encore ou a été détruit[1]. Une valeur différente de *nil* indique au contraire que l'objet est actif. Cette propriété peut être consultée ou modifiée. Cependant, on ne peut lui affecter que la valeur *nil*, ce qui conduit à la destruction de l'objet à laquelle elle est rattachée.

Notons que les modifications de la propriété *lifeline* se transmettent d'un conteneur à son contenu. La destruction d'un conteneur détruit tous les objets qu'il contient de façon récursive.

1.3 Portée et durée de vie des objets

Une application NJN est une structure hiérarchique de composants. Les composants hébergent des tâches, des variables, des propriétés, des canaux, d'autres composants, etc. Chacun des éléments de cette architecture porte un nom visible localement.

La portée des noms est définie par le conteneur (composant, canal ou vecteur) directement englobant. Par exemple, une variable définie dans un composant est accessible, via son nom, dans toutes les tâches hébergées par ce composant au même niveau hiérarchique.

Dans l'extrait de code ci-dessous, une variable nommée *v* est définie dans le composant *comp1*. On en récupère la référence dans la tâche *task1* définie dans le même composant.

```
NJN_DEF_ALWAYS(task1){
    njn_ref_t v = njn_get(local, "v");
    /*...*/
}

NJN_DEF_COMPONENT(comp1){
    njn_ref_t v = njn_new(self, "var", "v");
    njn_new(self, "rt_task", "task1", NJN_ALWAYS(task1),
        NJN_WHEN(v));
    /*...*/
    return NJN_SUCCESS;
}
```

La méthode *njn_get()* renvoie l'objet qui a pour nom la chaine transmise en second argument. Le premier argument définit la portée de recherche.

Compte tenu des mécanismes de synchronisation d'NJN, il n'est pas possible de récupérer un objet par la méthode *njn_get()* dans le bloc de code où il a été créé[2].

1 A ce stade de l'exposé, on peut se demander comment un objet qui n'existe pas encore peut avoir une propriété.

2 Pour que la recherche aboutisse, il faut nécessairement attendre la date de création effective de

Appeler cette méthode depuis un composant pour récupérer un objet défini localement est donc la plupart du temps une erreur. La seule possibilité est de mémoriser dans une variable locale l'objet renvoyé par la fabrique *njn_new()*. Bien que pointant sur un objet temporairement invalide, la référence renvoyée est suffisante pour configurer l'objet futur.

Appelée depuis une tâche, la méthode *njn_get()* aboutira plus sûrement[1]. La variable *local* représente l'espace de noms de la tâche. Il correspond au composant ou au canal qui l'héberge.

Rien n'interdit de prime abord d'accéder à un espace de noms quelconque si l'on dispose d'une référence sur le module qui lui correspond. En effet, tout module défini peut être transmis en premier argument à la méthode *njn_get()*. C'est ce qui est exploité dans l'exemple ci-dessous.

```
NJN_DEF_COMPONENT(comp1){
    njn_ref_t v = njn_new(self, "var", "v");
    /*...*/
}


NJN_DEF_ALWAYS(task2){
    njn_ref_t inst1 = njn_get(local, "inst1");
    njn_ref_t v     = njn_get(inst1, "v");
    /*...*/
}


NJN_DEF_COMPONENT(comp2){
    njn_new(self, "comp1", "inst1");
    njn_new(self, "rt_task", "task2", NJN_ALWAYS(task2));
    /*...*/
    return NJN_SUCCESS;
}
```

Depuis la tâche *task2*, on demande à la méthode *njn_get()* la référence du composant *inst1* qui se situe dans la même portée. Cette référence est ensuite utilisée pour récupérer la variable *v* qui elle, est au niveau hiérarchique inférieur.

Complément de la méthode *njn_get()*, la méthode *njn_get_owner()* renvoie le conteneur du module qu'on lui fournit en argument. En associant ces deux méthodes, on accède à l'ensemble de l'application, bien que la technique soit un peu fastidieuse.

La durée de vie des objets structurels dans une application NJN est parfaitement explicite. Par défaut, ils sont créés et détruits en même temps que leur conteneur. Mais il est toujours possible de raccourcir leur durée de vie. Dans l'exemple ci-dessous, la tâche *task3* est créée en même temps que le composant *comp3* qui l'héberge et est détruite 10 secondes après sa création, si son conteneur n'est pas détruit d'ici-là.

---

l'objet qui est située au mieux un Δ cycle après l'appel de la fabrique.

1 Le problème précédent ne se pose pas dans ce cas puisque, par définition, si le corps de la tâche est exécuté, c'est que la tâche existe bel et bien ainsi que tout ce qui a été créé en même temps.

```
NJN_DEF_COMPONENT(comp3){
    njn_ref_t task3 = njn_new(self, "rt_task", "task3",
        NJN_ALWAYS(task3),
        NJN_DYNAMIC);
    njn_delete(task3, NJN_AFTER(10 SEC));
    /*...*/
    return NJN_SUCCESS;
}
```

La tâche est présente dans l'espace de nom du composant entre l'instant de sa création et celui de sa destruction, à un Δ cycle près.

2 Le composant est l'objet structurel de base

Un composant est défini par son interface et son corps. L'interface rassemble ses caractéristiques visibles et/ou modifiables de l'extérieur. Il s'agit de son port d'entrées/sorties d'une part, et de ses propriétés et paramètres d'autre part. Le corps est quant à lui privé et définit la structure interne ou le comportement du composant.

2.1 Le port d'entrées/sorties relie le composant à son environnement

Le port est constitué des variables (et par extension des objets de type *vector* ou *channel*) déclarées en entrée ou en sortie. Par exemple, le composant *Add* défini ci-dessous dispose de deux entrées nommées *a* et *b* et d'une sortie nommée *out*.

```
NJN_DEF_COMPONENT(Add){
    njn_ref_t a = njn_new(self, "var", "a",
        NJN_INPUT, NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t b = njn_new(self, "var", "b",
        NJN_INPUT, NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "out", NJN_OUTPUT);
    /*...*/
    return NJN_SUCCESS;
}
```

La connexion d'un composant avec son environnement extérieur est établie lors de l'instanciation. Chaque élément du port d'entrées/sorties est mis en relation avec un objet de classe identique situé au niveau hiérarchique supérieur.

```
NJN_DEF_COMPONENT(Filter1){
    njn_ref_t in  = njn_new(self, "var", "in",
        NJN_INPUT, NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t o1  = njn_new(self, "var", "o1",
        NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t out = njn_new(self, "var", "out", NJN_OUTPUT);
    njn_new(self, "Add",   "op1",
        NJN_ASSIGN("a", in),
        NJN_ASSIGN("b", o1),
        NJN_ASSIGN("out", out));
    njn_new(self, "Shift", "op2",
        NJN_ASSIGN("in", out),
```

```
            NJN_ASSIGN("out", o1));
        return NJN_SUCCESS;
    }
```

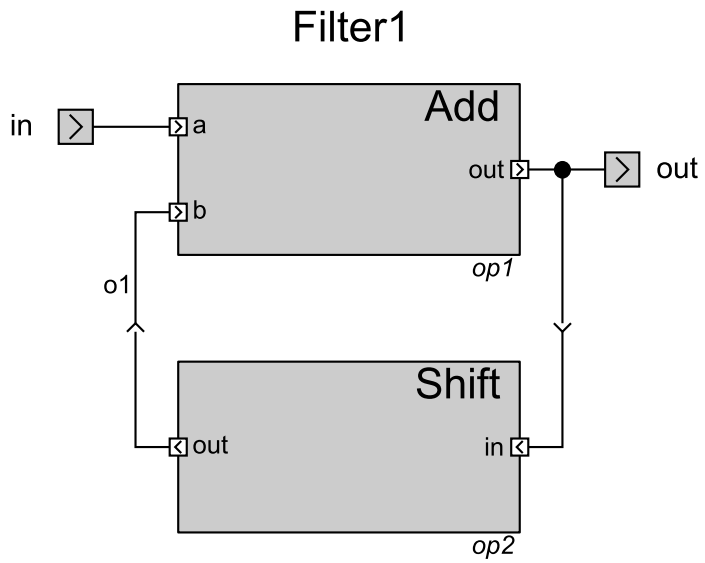


Figure 39. Un composant structurel nommé *Filter1* composé de deux instances de composants.

Notons que les connexions établies entre variables peuvent être supprimées ou modifiées au cours de l'exécution.

2.2 Les propriétés et paramètres renseignent sur l'état du composant

Les propriétés et paramètres des composants sont publiques. On peut donc les consulter depuis l'extérieur du composant.

La propriété *content* vient s'ajouter aux propriétés *name* et *lifeline* déjà rencontrées. Elle signale les changements structurels infligés au corps de l'instance de composant. L'ajout d'une tâche ou d'une variable, le changement de nom d'une variable, la suppression d'un composant interne, etc. se traduisent par l'inscription d'un évènement sur cette propriété. La propriété *content* ne peut être que consultée.

A l'exception des paramètres définis pour tous les objets, c'est-à-dire *owner*, *pattern* et *package*, il n'y a pas de paramètre supplémentaire associé aux composants.

Néanmoins, le concepteur de l'application peut définir des paramètres utilisateurs et les associer à un composant. La section 4 page 91 aborde ce point.

2.3 Le corps décrit la structure ou le comportement du composant

Le corps d'un composant définit sa structure interne, c'est-à-dire l'ensemble des objets qu'il renferme et la manière dont ils entrent en interaction pour produire le comportement attendu. Parmi les éléments du corps, on retrouve les signaux du port d'entrées/sorties.

Classiquement, il y a deux manières d'écrire le corps d'un composant :

- La première est dite structurelle. Elle correspond à un composant constitué d'instances d'autres composants. Le comportement du composant résulte alors de celui des instances qu'il héberge et de leur agencement au sein de leur conteneur commun.
- La seconde est dite comportementale et correspond à un composant dont le comportement est le résultat d'une ou plusieurs tâches. Les tâches décrivent alors les algorithmes à appliquer aux données.

Bien évidemment, une description mixte, faisant appel à des instances de composant et à des tâches, est toujours possible.

```
NJN_DEF_COMPONENT(Add){
    njn_ref_t a = njn_new(self, "var", "a", NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t b = njn_new(self, "var", "b", NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "out", NJN_OUTPUT));
    njn_new(self, "vt_task", "tk",
        NJN_ALWAYS(Add),
        NJN_WHEN(a, b));
    return NJN_SUCCESS;
}
```

Le contenu du corps d'un composant peut évoluer au cours de l'exécution. Tous les objets créés dans le composant portent un nom d'instance (propriété *name*) et ont une ligne de vie (propriété *lifeline*). A un instant donné, ne sont visibles dans le corps du composant que les objets dont la ligne de vie atteste de l'activité. Chacun de ces objets est accessible par son nom d'instance.

2.4 Héritage entre composants

Les composants peuvent être construits en s'appuyant sur la notion d'héritage.

```
NJN_DEF_COMPONENT(I2O1){
    njn_new(self, "var", "a", NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "b", NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "out", NJN_OUTPUT);
}
```

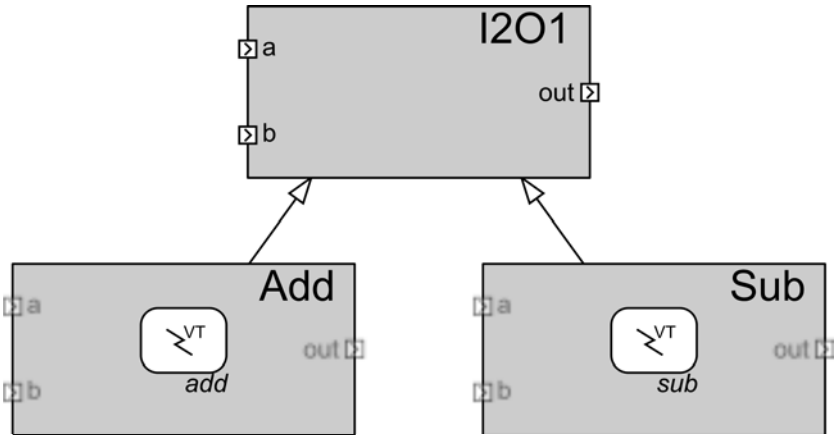


Figure 40. Relation d'héritage entre composants. Le composant *I2O1* est spécialisé en deux classes : *Add* et *Sub*.

Les définitions des caractéristiques communes à plusieurs composants peuvent ainsi être mutualisées.

```
NJN_DEF_COMPONENT(Add){
    NJN_EXTENDS(I2O1);
    njn_ref_t a = njn_get(super, "a");
    njn_ref_t b = njn_get(super, "b");
    njn_new(self, "vt_task", "tk",
        NJN_ALWAYS(Add),
        NJN_WHEN(a, b));
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(Sub){
    NJN_EXTENDS(I2O1);
    njn_ref_t a = njn_get(super, "a");
    njn_ref_t b = njn_get(super, "b");
    njn_new(self, "vt_task", "tk",
        NJN_ALWAYS(Sub),
        NJN_WHEN(a, b));
    return NJN_SUCCESS;
}
```

3 Les variables, leur type et leur comportement

Les objets de la classe *var* sont les données manipulées par les tâches de l'application. Leur type est géré dynamiquement. Elles pourraient être des variables au sens commun, à ceci près que leur comportement temporel diffère des variables classiquement rencontrées dans la plupart des langages de programmation. Les variables peuvent être déclarées en entrée ou en sortie. Elles deviennent dans ce cas des éléments du port d'entrées/sorties du composant qui les héberge.

3.1 Le typage des variables est dynamique

Dans une application NJN, le type des données n'est pas spécifié. Une variable peut donc renfermer un type de base *Int*, *Float*, *String*, un conteneur de type *Array* ou *HTable*, une structure de données complexe, un objet[1], etc. Son type peut même évoluer au cours de l'application.

```
njn_ref_t v = njn_new(self, "var", "v");
njn_write(v, dohc_int(10));
njn_write(v, dohc_float(0.1), NJN_AFTER(100 MSEC));
```

Il est toutefois possible de préciser un type ou plus généralement une contrainte en fournissant au constructeur une fonction de contrainte. Une valeur d'initialisation est dans ce cas requise pour garantir le respect de la contrainte dès la création de la variable.

```
njn_ref_t v = njn_new(self, "var", "v",
    NJN_CONSTRAINT(dohc_is_float), NJN_INITVAL(dohc_float(0.0)));
/*...*/
NJN_ASSERT(dohc_get_error(
    njn_write(v, dohc_int(1)));        /*Returns an error*/
NJN_ASSERT(dohc_is_nil(
    njn_write(v, dohc_float(1.0)));    /*Returns nil*/
```

La tentative d'écriture d'une valeur non conforme génère une erreur mais ne modifie pas la valeur de la variable.

3.2 Comportement temporel de l'affectation des variables

Comme nous l'avons évoqué plus haut, le comportement temporel des variables NJN est différent de ce qu'on observera dans la majorité des langages de programmation. Il se rapproche de celui des signaux du langage VHDL [57] ou des variables du langage SystemVerilog [58][2]. Afin de garantir la causalité des évènements qui se produisent dans l'application, l'affectation d'une nouvelle valeur à une variable n'est jamais immédiatement observable.

```
njn_ref_t v = njn_new(self, "var", "v", NJN_INITVAL(dohc_int(0)));
/*...*/
njn_ref_t tmp = njn_read(v);                  /*tmp <= v      */
njn_write(v, dohc_inc(tmp));                  /*v   <= tmp + 1*/
NJN_ASSERT(dohc_is_equal(tmp, njn_read(v)));  /*v   == tmp    */
```

En pratique, on ne peut modifier ni le présent, ni le passé, on ne peut que décrire ce qu'on attend du futur.

3.3 Connectivité et variable de port d'entrées/sorties

Les variables peuvent être connectées les unes aux autres. De cette façon, les changements d'état d'une variable sont reportés sur toutes celles qui lui sont connectées.

1 Au sens de la programmation orientée objet.
2 Dans le cas d'une affectation non bloquante.

```
njn_ref_t v1 = njn_new(self, "var", "v1", NJN_INITVAL(dohc_int(0)));
njn_ref_t v2 = njn_new(self, "var", "v2");
njn_connect(v2, v1);          /*v2 <- v1*/
/*...*/
NJN_ASSERT(dohc_is_same(njn_read(v1), njn_read(v2)));
```

La syntaxe de connexion précise explicitement le sens de transit de l'information : on connecte une variable cible à une variable source. L'écriture sur la cible par un autre biais qu'à travers la source n'affectera nullement cette dernière[1].

La remarque précédente en amène une seconde. Pour garantir la causalité de l'application, il est capital d'éviter toute affectation simultanée d'une même variable. Une manière de respecter cette règle est de faire en sorte qu'il n'existe qu'une seule source (une tâche ou une connexion) susceptible de modifier l'état de chaque variable. Les sources multiples sont donc permises mais à éviter autant que possible.

Lorsqu'une variable est déclarée en entrée ou en sortie, elle participe à la définition du port de son composant. Elle peut alors faire l'objet d'une assignation de port, ce qui revient à établir une connexion entre elle et une variable située à l'extérieur du composant.

```
NJN_DEF_COMPONENT(Shift){
    njn_ref_t in = njn_new(self, "var", "in", NJN_INPUT,
       NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "out", NJN_OUTPUT);
    /*...*/
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(Filter1){
    njn_ref_t in  = njn_new(self, "var", "in",
        NJN_INPUT, NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t o1  = njn_new(self, "var", "o1",
        NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t out = njn_new(self, "var", "out", NJN_OUTPUT);
    /*...*/
    njn_new(self, "Shift", "op2",
        NJN_ASSIGN("in", out),
        NJN_ASSIGN("out", o1));
    return NJN_SUCCESS;
}
```

Dans la syntaxe d'assignation, il n'est pas nécessaire de préciser le sens des échanges d'informations. Le mode d'accès de la variable de port permet de statuer sur l'orientation de la connexion.

3.4 Propriétés et paramètres des variables

Les variables disposent des propriétés et paramètres communs à tous les objets. Il s'agit des paramètres *owner*, *pattern* et *package* et des propriétés *name* et *lifeline*.

A cette liste s'ajoutent trois paramètres constants :

1 Les deux variables n'auront potentiellement plus la même valeur dans ce cas de figure.

- Le paramètre *access* renseigne sur l'appartenance ou non de la variable au port d'entrées/sorties de son composant. Une valeur égale à *NJN_INPUT_MODE* (respectivement *NJN_OUTPUT_MODE*) indique que la variable est déclarée en entrée (respectivement en sortie). Une valeur égale à *NJN_NO_ACCESS_MODE* indique quant à elle que la variable ne participe pas à la définition du port.
- Le paramètre *constraint* renvoie la fonction de contrainte fournie lors de la définition de la variable.
- Le paramètre *initval* conserve la valeur initiale donnée à la variable.

Et deux propriétés :

- La propriété *sources* tient à jour une liste des sources actives auxquelles la variable est connectée.
- La propriété *targets* fait de même avec les variables qui sont connectées à elle et pour lesquelles elle est une source.

Ces deux dernières propriétés changent d'état à chaque fois que la configuration de connexion d'une variable change. En un sens, elles sont à rapprocher de la propriété *changes* des composants.

4 Les propriétés et les paramètres

Propriétés et paramètres ont en commun de caractériser l'état ou le comportement d'un objet de l'application. Ce point justifie leur présentation commune au sein de cette section.

Leur différence est cependant capitale : les premières peuvent modifier en cours d'exécution le comportement de l'objet auquel elles sont rattachées ou relater un changement d'état de celui-ci ; les seconds ne le peuvent en aucune manière et sont le plus souvent des constantes[1].

4.1 Définition et politique d'accès des propriétés et paramètres

La liste des paramètres et des propriétés associés à un objet dépendent en général de sa classe. Nous avons vu qu'un petit nombre d'entre eux sont communs à tous les objets[2]. Les autres sont spécifiques. En outre, le concepteur peut définir de nouveaux paramètres. Cette possibilité est restreinte aux paramètres de composants et de canaux.

C'est en principe le constructeur d'objets qui se charge d'attribuer une valeur aux paramètres et propriétés de l'objet. Dans l'exemple ci-dessous, le constructeur définit un nouvel objet de type *var*.

1 Il y a une exception : la priorité des tâches est un paramètre non constant.
2 Voir section 1.2 page 82.

```
njn_new(self, "var", "a", NJN_INPUT, NJN_INITVAL(dohc_int(0)));
```

Le premier argument fourni au constructeur est la portée dans laquelle est placée le nouvel objet, il s'agit donc du paramètre *owner* ; le second argument désigne le nom du modèle utilisé, il est à rapprocher du paramètre *pattern* ; le troisième argument correspond au nom de l'instance créée, c'est-à-dire la propriété *name* ; l'argument *NJN_INPUT* placera le paramètre *access* dans l'état *NJN_INPUT_MODE*, etc.

On accède à chaque paramètre et propriété par des méthodes de type *get* et *set* spécifiques. Selon les droits d'accès associés à l'élément manipulé, la méthode de type *set* est définie ou non. Par exemple pour la propriété *name*, les méthodes *get* et *set* sont définies comme ceci :

```
njn_ref_t   njn_get_name(njn_ref_t self);
const char *NJN_GET_NAME(njn_ref_t self);                  /*macro*/

njn_ref_t njn_set_name(njn_ref_t self, njn_ref_t value);
njn_ref_t NJN_SET_NAME(njn_ref_t self, const char *value); /*macro*/
```

Les paramètres étant en général des constantes, ils ne disposent pas de méthode d'écriture. Par exemple, pour le paramètre *owner*, seule la méthode *get* est implémentée :

```
njn_ref_t njn_get_owner(njn_ref_t self);
```

4.2 Association d'une propriété à une variable

NJN donne la possibilité de lier une variable à une propriété, de cette manière, la propriété en question peut être manipulée par son intermédiaire : on pourra appeler les méthodes de lecture (*njn_read()*) et d'écriture (*njn_write()*), on pourra placer la variable dans le port d'entrées/sorties d'un composant, réveiller une tâche lorsque elle change d'état, etc.

```
NJN_DEF_COMPONENT(Mutable){
    njn_new(self, "var", "content", NJN_OUTPUT,
        NJN_BIND_TO_CONTENT(self));
    /*...*/
}

NJN_DEF_COMPONENT(Control){
    njn_ref_t c = njn_new(self, "var", "c");
    njn_new(self, "Mutable", "m",
        NJN_ASSIGN("content", c), /*...*/);
}
```

Les opérations légales sur la variable ainsi définie dépendent bien évidemment des caractéristiques de la propriété à laquelle elle est liée. Par exemple, une propriété qui n'est que consultable ne pourra pas être associée à une variable définie en entrée d'un composant.

Cette construction est capitale. Elle fournit à NJN la base des outils de monitoring. La section 5.4 en montrera un exemple.

4.3 Définition de paramètre de composant ou de canal

Le choix des valeurs des paramètres et propriétés n'est pas toujours facile à fixer au moment de leur définition. On aimerait bien par exemple pouvoir décider de la latence d'une tâche lorsqu'on instancie le composant qui l'héberge. On peut pour cela exploiter la technique développée au paragraphe précédent : lier à la propriété *latency* de la tâche une variable d'entrée du composant et ajuster la valeur de la latence à postériori.

NJN offre une autre solution basée sur la définition de paramètres par le concepteur de l'application. Ainsi, le composant *Sampler* ci-dessous comporte un paramètre nommé *period* dont la valeur est utilisée pour initialiser la latence et le *timeout* de la tâche nommée *t*.

```
NJN_DEF_COMPONENT(Sampler){
    njn_ref_t period = njn_new(self, "param", "period",
        NJN_CONSTRAINT(dohc_is_int), NJN_INITVAL(dohc_int(10 MSEC)));
    njn_new(self, "var", "in",  NJN_INPUT);
    njn_new(self, "var", "out", NJN_OUTPUT);
    njn_new(self, "rt_task", "t",
        NJN_ALWAYS(Sampler),
        NJN_LATENCY(DOHC_TO_INT(period)),
        NJN_TIMEOUT(DOHC_TO_INT(period)));
    return NJN_SUCCESS;
}
```

Au moment de l'instanciation, on pourra donner à ce paramètre une valeur appropriée.

```
NJN_DEF_COMPONENT(Filter1){
    njn_ref_t in   = njn_new(self, "var", "in",  NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t i1   = njn_new(self, "var", "i1");
    /*...*/
    njn_new(self, "Sample", "op1",
        NJN_ASSIGN("in", in),
        NJN_ASSIGN("out", i1),
        NJN_PARAM("period", dohc_int(100 MSEC));
    /*...*/
    return NJN_SUCCESS;
}
```

Ce type de paramètre est nécessairement constant. Passée l'instanciation du composant, il n'est donc plus possible d'en changer la valeur.

Pour finir, notons que seuls les composants et les canaux supportent ce mécanisme.

5 Les tâches portent le code actif de l'application

Les tâches sont au modèle d'exécution d'NJN ce que sont les composants à son modèle structurel : les éléments fondamentaux. Elles portent le code actif des applications, et sont à l'origine de tous les changements d'état de tous les objets de

l'application. Leur organisation structurelle découle donc directement des nécessités du modèle d'exécution.

L'objet de cette section n'est pas de revenir sur le modèle d'exécution. Néanmoins, il faut garder à l'esprit que quatre familles de tâches sont nécessaires au bon fonctionnement de ce modèle : les tâches temps-réel, les tâches temps-virtuel, les boites d'horloges et les boites de découplage. Le premier paragraphe rappellera quelques caractéristiques de ses quatre familles. Ces quatre types de tâche ont en commun une structure qui s'appuie sur deux éléments essentiels : un ensemble de règles de déclenchement et un corps. Les règles de déclenchement établissent la liste des évènements ou des conditions qui provoquent l'exécution de la tâche. Le corps renferme le code correspondant aux actions à entreprendre si l'une des règles de déclenchement est activée. Nous verrons plus en détail le rôle de ses deux éléments dans les paragraphes suivants. Nous nous intéresserons ensuite à la définition de la latence, propriété spécifique des tâches temps-réel. Le dernier paragraphe récapitulera les différents paramètres et propriétés associés aux tâches.

5.1 Quatre variantes de tâches : les tâches temps-réel, les tâches temps-virtuel, les boites d'horloges et boites de découplage

Quatre variantes de tâches existent :

Les tâches dites « temps-réel » (*RTTask*) sont exécutées dans un contexte temporel contraint. Chaque exécution est programmée à une date temps-réel qui doit impérativement être respectée. Le rôle de ces tâches est de gérer les interactions de l'application avec l'extérieur.

Les tâches dites « temps-virtuel » (*VTTask*) n'ont pas cette contrainte. Elles sont exécutées au plus tôt et, dans la mesure du possible, lorsque toutes les données qu'elles consomment sont à jour. Ces tâches sont destinées à implémenter les traitements de données de l'application.

Les boites d'horloges (*ClockBox*) ont un comportement proche de celui des tâches temps-virtuel. Leur existence est justifiée par les problèmes de causalité que peuvent engendrer les mécanismes de propagation de données dans NJN[1].

Les boites de découplage (*DecouplingBox*) ont la particularité d'être insensibles aux retours arrière et d'autoriser les datations explicites par la méthode *njn_write_at()*. Elles ont pour fonction de rompre les liens temporels indésirables entre les différents niveaux hiérarchiques de l'application.

5.2 Déclenchement sur évènement interne

A chaque tâche est associée une sensibilité qui liste les variables de l'application qu'il faut surveiller. Lorsque l'une de ces variables est mise à jour, le noyau d'NJN programme l'exécution de la tâche.

1 Voir Chapitre 3.

```
NJN_DEF_COMPONENT(Add){
    njn_ref_t a   = njn_new(self, "var", "a",
        NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_ref_t b   = njn_new(self, "var", "b",
        NJN_INPUT,
        NJN_INITVAL(dohc_float(0.0)));
    njn_new(self, "var", "out",
        NJN_OUTPUT);
    njn_new(self, "vt_task", "Add",
        NJN_ALWAYS(Add),
        NJN_WHEN(a, b));
return NJN_SUCCESS;
}
```

Pour des raisons liées à l'implémentation, la liste de sensibilité est assignée à la tâche de façon définitive. Elle ne peut pas être modifiée.

5.3 Déclenchement sur évènement externe

Il est également possible de déclencher l'exécution d'une tâche lorsqu'un évènement externe à l'application se produit comme la réception de données sur un port de communication.

Dans l'exemple ci-dessous, la tâche *t* est sensibilisé à l'objet *tic* de type *handle* qui encapsule une communication par socket *tcp*.

```
NJN_DEF_INITIAL(t){
    njn_ref_t tic = njn_get(local, "tic");
    njn_open(tic);
    return NJN_SUCCESS;
}

NJN_DEF_ALWAYS(t){
    njn_ref_t w   = njn_get(local, "w");
    njn_ref_t tic = njn_get(local, "tic");
    njn_ref_t buff;
    switch(event){
    case NJN_HANDLE_EVENT:
        buff = njn_read(tic);
        njn_write(w, dohc_string_new2(self,
            DOHC_TO_RAW(buff, char), NJN_SIZE(buff)));
        break;
    }
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(tcp){
    njn_ref_t w   = njn_new(self, "var", "w");
    njn_ref_t tic = njn_new(self, "handle", "tic",
        NJN_TCP("localhost", 2000));
    njn_new(self, "rt_task", "t",
        NJN_INITIAL(t),
        NJN_ALWAYS(t),
        NJN_WHEN(tic),
        NJN_LATENCY(100 MSEC));
    njn_new(self, "vt_task", "r",
```

```
            NJN_ALWAYS(helloworld_r),
            NJN_WHEN(w));
        return NJN_SUCCESS;
    }
```

Chaque fois que des données sont reçues via ce socket, la tâche *t* est réveillée.

Les objets *handle* supportent les sockets de type *tcp* et *udp* mais aussi les ports de communication séries, les *pipes*, les fichiers, les services *CABLES*[1], etc.

5.4 Déclenchement temporel : *Timeout* et *Timeslot*

Il est enfin possible de programmer le déclenchement temporel de l'exécution d'une tâche. Deux possibilités existent, la programmation d'un *timeout* et l'attribution d'un *timeslot*.

La propriété *timeout* associée à une tâche définit le temps maximum entre deux exécutions de celle-ci. Si la tâche n'est pas réveillée par un autre moyen (évènement interne ou externe), elle sera automatiquement déclenchée à l'issue du temps spécifié.

L'extrait de code ci-dessous montre tout l'intérêt d'un tel mode de déclenchement. Deux tâches sont instanciées dans cet exemple : la tâche *task* et la tâche *monitor*. Cette dernière a pour fonction de surveiller la première.

```
NJN_DEF_ALWAYS(monitor){
    switch(event){
    case NJN_TIMEOUT_EVENT:
        printf("The task is probably dead!\n");
        break;
    }
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(Monitoring){
    /*...*/
    njn_ref_t task    = njn_new(self, "rt_task", "task", /*...*/);
    njn_ref_t beat    = njn_new(self, "var", "beat",
        NJN_BIND_TO_BEAT(task));
    njn_new(self, "vt_task", "monitor",
        NJN_ALWAYS(monitor),
        NJN_WHEN(beat),
        NJN_TIMEOUT(1 SEC));
    return NJN_SUCCESS;
}
```

Lorsque le signal *beat* est à l'origine du réveil de la tâche *monitor*, aucune action n'est entreprise. En revanche, lorsqu'elle est réveillée par l'échéance de son *timeout*, un message est envoyé sur la console. Autrement dit, tant que la durée entre deux exécutions de la tâche *task* n'excède pas le *timeout* spécifié pour la tâche *monitor*, aucun message n'est diffusé.

1 Voir Chapitre 7

En l'absence de toute autre source de déclenchement, l'attribution d'une valeur de *timeout* conduit à une exécution périodique de la tâche. Cette façon de faire n'est cependant pas idéale.

Si un tel comportement périodique est attendu, il est préférable d'utiliser la propriété *timeslot*. Celle-ci permet d'attribuer à la tâche à laquelle elle est associée une fenêtre temporelle périodique d'exécution.

Un *timeslot* est défini par trois informations : la première est la période de répétition, la seconde correspond au début de la fenêtre temporelle au sein de cette période, enfin la troisième donne la durée maximum attribuée à l'exécution de la tâche. L'origine de la référence de temps est fixée à l'instant 0 du système (1er janvier 1970 à 0h00).

```
NJN_DEF_COMPONENT(tic){
    njn_new(self, "rt_task", "t",
        NJN_ALWAYS(t),
        NJN_TIMESLOT(1 SEC, 100 MSEC, 10 MSEC));
    /*...*/
    return NJN_SUCCESS;
}
```

Dans l'exemple ci-dessus, la tâche *t* doit se répéter toutes les secondes. Son exécution est calée entre les instants *0,100s* et *0,110s* de chaque seconde.

Le déclenchement par *timeslot* est exclusif. Il n'est pas possible d'y associer une liste de sensibilité ou un *timeout*.

5.5 Un corps en trois blocs : *initial*, *final* et *always*

Trois blocs de code distincts peuvent être associés aux tâches de type temps-réel ou temps-virtuel : *initial*, *final* et *always*.

Le bloc *initial* est exécuté lorsque la tâche est crée. Bien qu'il soit autorisé dans la définition des tâches temps-virtuel, il est surtout utilisé pour la configuration des tâches temps-réel. Il permet d'effectuer certaines initialisations de paramètres ou de configurer des ressources matérielles avant l'exécution de la tâche.

Le bloc *final* est exécuté lors de la destruction de la tâche. Là aussi, il trouvera surtout sa raison d'être dans le contexte des tâches temps-réel. Il permet d'assurer la libération d'éventuelles ressources matérielles monopolisées par la tâche pendant son exécution.

Enfin le bloc *always* contient le code réellement actif de la tâche. Lorsque la tâche est exécutée, c'est ce bloc de code qui est appelé.

Dans l'extrait de code ci-dessous, l'unique tâche du composant *Bogus* comporte une implémentation de chacun de ces blocs. Pour l'exemple, ces trois blocs se contentent de produire un affichage dans la console. Le bloc *initial* se charge, en outre, d'attribuer à la propriété *timeslot* une valeur.

```
NJN_DEF_INITIAL(bogus_task){
    dohc_printf(self, "Birth at   %o!\n", njn_now(self));
    NJN_SET_TMESLOT(self, 1 SEC, 0, 0); /*Sets timeslot property*/
    return NJN_SUCCESS;
}

NJN_DEF_FINAL(bogus_task){
    dohc_printf(self, "Death at   %o!\n", njn_now(self));
    return NJN_SUCCESS;
}

NJN_DEF_ALWAYS(bogus_task){
    dohc_printf(self, "Running at %o!\n", njn_now(self));
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(Bogus){
    njn_new(self, "rt_task", "bogus_task",
        NJN_INITIAL(bogus_task),
        NJN_FINAL(bogus_task),
        NJN_ALWAYS(bogus_task));
    return NJN_SUCCESS;
}
```

L'exécution de ce code pendant 5 secondes produit l'affichage suivant :

```
Birth at   14:25:02,097!
Running at 14:25:02,097!
Running at 14:25:03,197!
Running at 14:25:04,197!
Running at 14:25:05,197!
Running at 14:25:06,197!
Death at   14:25:06,998!
```

Le bloc *initial* est d'abord exécuté puis le bloc *always* une première fois. La propriété *timeslot* provoque ensuite la répétition du bloc *always* à quatre reprises en espaçant chaque occurrence d'une durée d'une seconde. Enfin le bloc *final* avertit de la destruction du composant. Il permet de libérer les ressources matérielles éventuellement monopolisées par la tâche.

Deux remarques pour terminer ce paragraphe : les boites d'horloges et les boites de découplage ne supportent que le bloc *always ;* les blocs *initial* et *final* n'ont véritablement de sens que dans le cadre de tâches temps-réel.

5.6 La latence des tâches temps-réel

Afin de garantir la faisabilité de l'exécution des tâches temps-réel, celles-ci sont caractérisées par une propriété de latence qui est définie de la façon suivante : la latence d'une tâche temps-réel définit la durée qui sépare la date de lecture des données consommées par la tâche de la date d'exécution de cette même tâche ; cette date d'exécution correspond également à la date d'écriture des données produites par la tâche.

Prenons un exemple : Si une tâche est paramétrée avec une latence de *1 s*, elle verra la structure de données de l'application (structure de l'application, état des variables, etc.) telle qu'elle était *1 s* avant son exécution. En revanche, comme

c'est le cas pour toutes les tâches, les modifications qu'elle opèrera sur l'état des variables ne prendront effet qu'à l'issue de son exécution.

La latence a une autre conséquence : si un évènement interne est à l'origine du déclenchement d'une tâche temps-réel, l'exécution de cette tâche sera retardée de la valeur de sa latence.

```
NJN_DEF_ALWAYS(t1){
    njn_ref_t a = njn_get(local, "a");
    njn_printf(self, "t1 %o!\n", njn_now(self));
    njn_write(a, dohc_int(rand()));
    return NJN_SUCCESS;
}

NJN_DEF_ALWAYS(t1){
    njn_printf(self, "t2 at %o!\n", njn_now(self));
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(c1){
    njn_ref_t a = njn_new(self, "var", "a");
    njn_new(self, "vt_task", "t1",
        NJN_ALWAYS(t1),
        NJN_TIMESLOT(1 SEC, 0, 0));
    njn_new(self, "rt_task", "t2",
        NJN_ALWAYS(t2),
        NJN_WHEN(a),
        NJN_LATENCY(1 SEC));
    return NJN_SUCCESS;
}
```

6 Des méthodes pour abstraire les échanges d'informations

Les méthodes permettent d'abstraire les communications entre composants sous la forme d'appels de services asynchrones. Le principe est le suivant : un composant (ou un canal) implémente une méthode et en fournit une interface d'accès sur son port d'entrées/sorties. La méthode peut alors être appelée depuis l'extérieur du composant sans que son implémentation réelle soit connue de l'extérieur.

6.1 Définition

Une méthode est un objet qui peut être hébergé par un composant ou un canal. La méthode porte un nom et est associée à une implémentation. Pour être accessible de l'extérieur du composant, une méthode doit être exportée. Elle devient dans ce cas un élément du port d'entrée/sortie du composant.

```
NJN_DEF_COMPONENT(slave){
    njn_new(self, "method", "hello",
        NJN_BODY(slave_hello),  // method implementation
        NJN_EXPORT);
    return NJN_SUCCESS;
}
```

Ici un composant nommé *slave* héberge la méthode *hello*. Elle est associée à l'implémentation *slave_hello*.

L'implémentation de la méthode, son corps, s'apparente à une fonction. Les arguments lui sont transmis sous la forme d'une liste d'objets.

```
NJN_DEF_BODY(slave_hello){
    njn_ref_t name = dohc_shift(args);
    return dohc_sprintf(self, "Hello %o!", name);
}
```

La valeur de retour est de type quelconque et sera retournée à l'appelant.

Côté appelant, la méthode doit également être définie mais aucune implémentation ne doit être fournie.

Pour qu'un composant puisse appeler une méthode qui lui est extérieure, elle doit être importée. Elle est alors associée au port d'entrées/sorties du composant.

```
NJN_DEF_COMPONENT(master){
    njn_ref_t hello = njn_new(self, "method", "hello",
        NJN_IMPORT);
    ...
    return NJN_SUCCESS;
}
```

Le composant *master* ci-dessus importe la méthode *hello*. Elle n'est pas implémentée localement.

6.2 Topologie

Les méthodes sont des objets qui peuvent se connecter entre eux de la même manière que les variables. Cependant, une seule implémentation doit être présente sur l'ensemble du chemin constitué par les interconnexions.

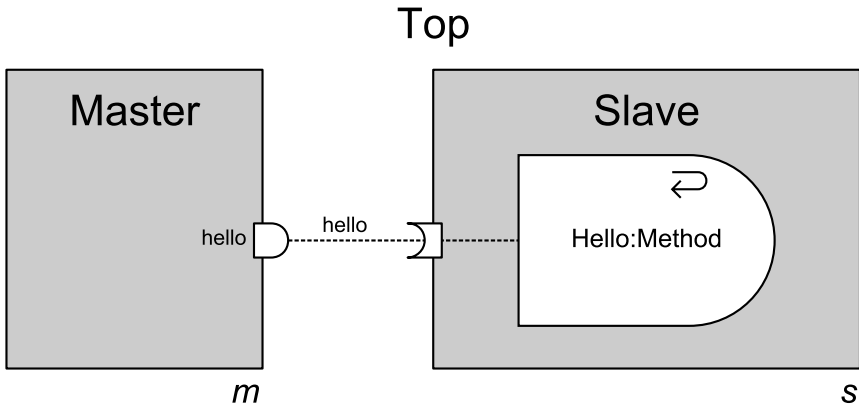


Figure 41. Une méthode.

Dans l'exemple ci-dessous, le composant *top* instancie les composants *master* et *slave* précédents. Un objet méthode connecte l'implémentation fournie par le composant *slave* à la méthode non-implémentée du composant *master*.

```
NJN_DEF_COMPONENT(top){
    njn_ref_t hello = njn_new(self, "method", "hello");
    njn_new(self, "master", "m",
        NJN_ASSIGN("hello", hello));
    njn_new(self, "slave", "s",
        NJN_ASSIGN("hello", hello));
    return NJN_SUCCESS;
}
```

Ainsi le composant *master* pourra faire appel à la méthode *hello* du composant *slave* sans en connaitre les détails d'implémentation.

6.3 Appel

Pour appeler le service, il suffit de fournir à la fonction *njn_call()* la référence de la méthode ainsi que les arguments d'appel. NJN se charge d'acheminer la requête jusqu'à la méthode implémentée.

Dans l'exemple ci-dessous, le composant *master* héberge une tâche nommée *c*, dont le corps appelle chaque seconde la méthode *hello*.

```
NJN_DEF_ALWAYS(master_call){
    njn_ref_t hello = njn_get(local, "hello");
    njn_call(hello, dohc_string_new(self, "M. Steux"));
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(master){
    njn_ref_t hello = njn_new(self, "method", "hello", NJN_IMPORT);
    njn_new(self, "vt_task", "c",
        NJN_ALWAYS(master_call),
        NJN_TIMESLOT(1 SEC, 0, 100 MSEC));
    ...
    return NJN_SUCCESS;
}
```

La récupération de la valeur de retour n'est évidemment pas immédiate car plusieurs cycles d'exécution sont nécessaires pour acheminer la requête et récupérer le résultat. Une tâche spécifique doit donc attendre la réponse.

Pour ce faire, la tâche en question doit comporter dans sa liste de sensibilité la référence de la méthode. Elle sera réveillée dès que la valeur de retour sera disponible.

```
NJN_DEF_COMPONENT(master){
    ...
    njn_new(self, "rt_task", "r",
        NJN_ALWAYS(master_return),
        NJN_WHEN(hello), NJN_LATENCY(100 MSEC));
    return NJN_SUCCESS;
}
```

Le composant *master* ci-dessus comporte une tâche, nommée *r*, qui remplit ce rôle. Son corps, *master_return*, récupère l'information attendue. Un sélecteur de cas permet d'identifier l'origine du déclenchement. S'il s'agit d'une valeur de retour

de méthode, il suffit d'appeler la fonction *njn_get_return()* pour obtenir le résultat de l'appel de la méthode *hello*.

```
NJN_DEF_ALWAYS(master_return){
    njn_ref_t hello = njn_get(local, "hello");
    switch(event){
    case NJN_METHOD_RETURN_EVENT:
        dohc_printf(self, "%o\n", njn_get_return(hello));
        break;
    }
    return NJN_SUCCESS;
}
```

L'exécution de l'application exemple donne finalement :

```
Hello M. Steux!
Hello M. Steux!
Hello M. Steux!
Hello M. Steux!
...
```

7 Les canaux structurent les interconnexions

Le premier rôle des canaux est de fournir une structuration des interconnexions entre composants et de généraliser la notion de variable.

7.1 Principe topologique et définition

Un canal est un objet hybride situé entre le composant et la variable. Comme un composant, il est capable d'héberger d'autres objets tels que des variables, des instances de canaux ou de composants, etc. et comporte un port d'entrées/sorties. Comme une variable, il participe du réseau d'interconnexions de l'application.

Voici par exemple la définition d'un canal nommé *ch_wire*.

```
NJN_DEF_CHANNEL(ch_wire){
    njn_new(self, "var",    "req",   NJN_MAINDIR);
    njn_new(self, "var",    "ack",   NJN_REVERSEDIR);
    return NJN_SUCCESS;
}
```

L'architecture d'interconnexion des canaux est une topologie de type point à point orientée. Il est toutefois possible de réaliser une topologie type maitre/esclave. Il faut alors veiller à ce que deux esclaves ne soient pas actifs au même instant.

Le canal comporte une orientation privilégiée qui va en principe de l'objet qui contrôle le canal (le maitre) vers les objets qui répondent aux sollicitations de celui-ci (les esclaves). Les variables qui en composent la structure peuvent suivre cette orientation (*NJN_MAINDIR*) ou être orientées dans le sens opposé (*NJN_REVERSEDIR*). Cette attribution d'un sens relatif aux objets permet à NJN de construire les répéteurs internes qui supportent les échanges d'informations à travers les interconnexions.

La conception d'un nouveau canal nécessitera bien souvent trois définitions : un canal maitre, un canal esclave et un canal de transport, les deux premiers héritant des propriétés du troisième.

```
NJN_DEF_CHANNEL(ch_master){
    NJN_EXTENDS(self, "ch_wire");
    njn_new(self, "var", "req", NJN_MAINDIR,    NJN_INPUT);
    njn_new(self, "var", "ack", NJN_REVERSEDIR, NJN_OUTPUT);
    return NJN_SUCCESS;
}

NJN_DEF_CHANNEL(ch_slave){
    NJN_EXTENDS(self, "ch_wire");
    njn_new(self, "var", "req", NJN_MAINDIR,    NJN_OUTPUT);
    njn_new(self, "var", "ack", NJN_REVERSEDIR, NJN_INPUT);
    return NJN_SUCCESS;
}
```

Un canal sera constitué d'une seule instance maitre et d'éventuellement plusieurs instances esclaves. Des instances de transport assurent si nécessaire la connexion entre le maitre et ses esclaves à travers la hiérarchie de l'application.

Voici par exemple un composant faisant office de maitre :

```
NJN_DEF_COMPONENT(master){
    njn_ref_t req   = njn_new(self, "var", "req");
    njn_ref_t ack   = njn_new(self, "var", "ack");
    njn_new(self, "ch_master", "ch",
        NJN_OUTPUT,
        NJN_ASSIGN("req", req),
        NJN_ASSIGN("ack", ack));
    ...
    return NJN_SUCCESS;
}
```

Les variables internes *req* et *ack* sont connectées aux ports de même nom définis dans l'interface du canal maitre. L'orientation privilégiée du canal est déclarée sortante (*NJN_OUTPUT*) vis-à-vis de l'interface du composant. Il figure donc dans l'interface du composant. Sa variable interne *req*, déclarée dans la direction principale, sera orientée à la manière d'une sortie vis-à-vis composant tandis que sa variable *ack*, déclarée dans le sens opposé, s'apparentera à une entrée pour le composant.

De même, voici la définition d'un composant esclave :

```
NJN_DEF_COMPONENT(slave){
    njn_ref_t req   = njn_new(self, "var", "req");
    njn_ref_t ack   = njn_new(self, "var", "ack");
    njn_new(self, "ch_slave", "ch",
        NJN_INPUT,
        NJN_ASSIGN("req", req),
        NJN_ASSIGN("ack", ack));
    ...
    return NJN_SUCCESS;
}
```

On retrouve grosso modo la même structure de code que pour le maitre. Le canal est cette fois de type *ch_slave* et est orienté entrant (*NJN_INPUT*) vis-à-vis de l'interface du composant.

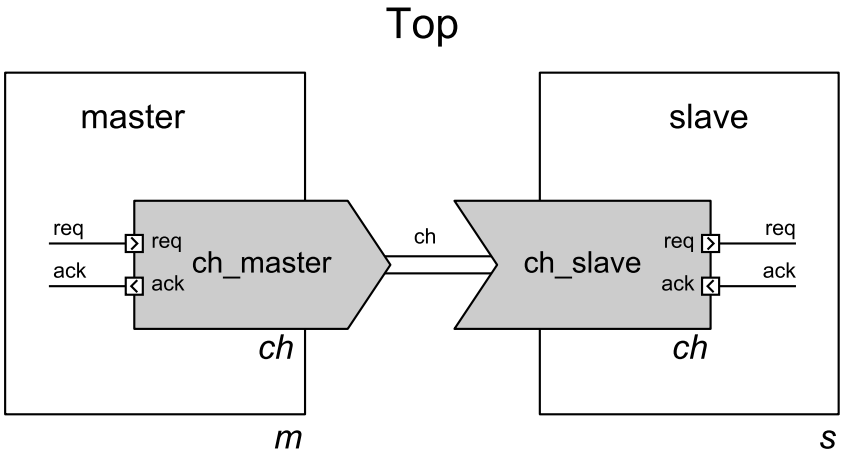


Figure 42. Un canal.

Au niveau hiérarchique supérieur, un canal de transport vient réaliser la connexion entre le canal de sortie du composant *master* et le canal d'entrée du composant *slave*.

```
NJN_DEF_COMPONENT(top){
    njn_ref_t ch = njn_new(self, "ch_wire", "ch");
    njn_new(self, "master", "m",
        NJN_ASSIGN("ch", ch));
    njn_new(self, "slave", "s",
        NJN_ASSIGN("ch", ch));
    return NJN_SUCCESS;
}
```

Cette façon d'utiliser les canaux n'est pas particulièrement avantageuse. Elle ne fait guère économiser d'éléments d'interconnexion et confie le contrôle de la communication au maitre et à ses esclaves. NJN propose une méthode bien plus abstraite de manipulation des canaux à travers les protocoles.

7.2 Les protocoles

Un protocole est constitué d'un ensemble de méthodes qui assurent la gestion abstraite de la communication à travers le canal. Ces méthodes se définissent de la même manière que les méthodes de composants et suivent les mêmes règles d'interconnexions.

Un protocole particulier, le protocole standard, permet de manipuler un canal à la manière d'une variable, par l'intermédiaire des fonctions *njn_read()* et *njn_write()*. Il est constitué de trois méthodes qu'il faut redéfinir pour chaque nouveau type de canal : les méthodes *read*, *write* et *event*.

Voici par exemple une nouvelle version du canal *ch_master* vu précédemment.dohc_int

```
NJN_DEF_CHANNEL(ch_master){
    NJN_EXTENDS(self, "ch_wire");
    njn_new(self, "method", "event", NJN_BODY(ch_master_event));
    njn_new(self, "method", "read",  NJN_BODY(ch_master_read));
    njn_new(self, "method", "write", NJN_BODY(ch_master_write));
    return NJN_SUCCESS;
}
```

Le canal est constitué des signaux *req* et *ack* de sa classe parent *ch_wire*. Les trois méthodes du protocole standard disposent d'une implémentation.

La méthode *event* est un filtre d'évènement utilisé pour implémenter la liste de sensibilité des tâches. Elle renvoie un entier non nul lorsque l'objet qu'elle reçoit en argument correspond à un objet à surveiller. Vis-à-vis d'un maitre, ce sont les évènements en provenance du signal interne d'acquittement *ack* qui nous intéressent.

```
NJN_DEF_BODY(ch_master_event){
    njn_ref_t ack   = njn_get(local, "ack");
    njn_ref_t net   = dohc_shift(args);
    return dohc_int(dohc_is_same(net, ack));
}
```

La méthode *read* abstrait la lecture de l'état du canal. Dans notre cas, elle renvoie l'état du signal *ack*.

```
NJN_DEF_BODY(ch_master_read){
    njn_ref_t ack = njn_get(local, "ack");
    return njn_read(ack);
}
```

La méthode *write* abstrait l'écriture sur le canal. Pour notre exemple, elle lance une requête en modifiant l'état du signal *req*.

```
NJN_DEF_BODY(ch_master_write){
    njn_ref_t req   = njn_get(local, "req");
    njn_ref_t value = dohc_shift(args);
    njn_ref_t after = dohc_shift(args);
    if (dohc_is_nil(after)) return njn_write(req, value);
    else return njn_write(req, value, after);
}
```

On retrouve bien évidemment les implémentations réciproques côté esclave.

```
NJN_DEF_CHANNEL(ch_slave){
    NJN_EXTENDS(self, "ch_wire");
    njn_new(self, "method", "event", NJN_BODY(ch_slave_event));
    njn_new(self, "method", "read",  NJN_BODY(ch_slave_read));
    njn_new(self, "method", "write", NJN_BODY(ch_slave_write));
    return NJN_SUCCESS;
}
```

```
NJN_DEF_BODY(ch_slave_event){
    njn_ref_t req = njn_get(local, "req");
    njn_ref_t net = dohc_shift(args);
    return dohc_int(dohc_is_same(req, net));
}

NJN_DEF_BODY(ch_slave_read){
    njn_ref_t req = njn_get(local, "req");
    return njn_read(req);
}

NJN_DEF_BODY(ch_slave_write){
    njn_ref_t ack   = njn_get(local, "ack");
    njn_ref_t value = dohc_shift(args);
    njn_ref_t after = dohc_shift(args);
    if (dohc_is_nil(after)) return njn_write(ack, value);
    else return njn_write(ack, value, after);
}
```

Pour exploiter le canal, il suffit alors d'appeler les méthodes classiques *njn_read()* et *njn_write()* utilisées pour manipuler les variables.

```
NJN_DEF_COMPONENT(master){
    njn_ref_t ch  = njn_new(self, "ch_master", "ch",
        NJN_OUTPUT);
    ...
    njn_new(self, "rt_task", "s",
        NJN_ALWAYS(master_s),
        NJN_TIMESLOT(1 SEC, 0, 100 MSEC));
    njn_new(self, "rt_task", "r",
        NJN_ALWAYS(master_r),
        NJN_WHEN(ch));
    return NJN_SUCCESS;
}

NJN_DEF_ALWAYS(master_s){
    njn_ref_t ch = njn_get(local, "ch");
    ...
    njn_write(ch, dohc_int(rand()));
    return NJN_SUCCESS;
}

NJN_DEF_ALWAYS(master_r){
    njn_ref_t ch  = njn_get(local, "ch");
    njn_ref_t data = njn_read(ch);
    ....
    return NJN_SUCCESS;
}
```

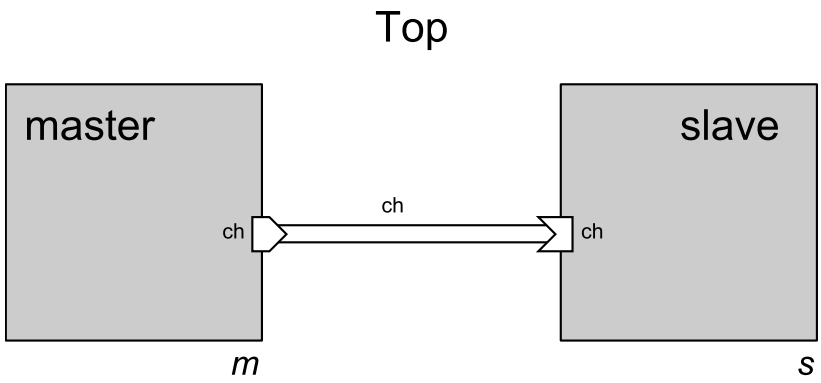


Figure 43. Un canal.

Les éléments internes du canal deviennent invisibles à son utilisateur. Il n'est plus utile de leur connecter quoi que ce soit. Le canal se comporte de la même façon qu'une variable à ceci près qu'il peut mettre en œuvre des échanges de données complexes et surtout bidirectionnels.

D'autres protocoles peuvent être construits. Ils devront s'appuyer sur les objets méthodes et sur les fonctions d'API associées : *njn_call()* et *njn_get_return()*. En effet, NJN ne permet pas de les prendre en charge de façon totalement transparente.

*
* *

Les nombreux objets d'NJN permettent au modèle structurel de couvrir une grande variété d'approches. Il est principalement orienté composants ce qui facilite la réutilisation du code et permet de construire des applications à partir de composants de bibliothèques. D'ailleurs, la plupart des applications n'exploiteront que les objets les plus simples : composants, tâches et variables. Il convient également aux approches orientées service grâce à des objets tels que les méthodes. Mais la véritable originalité de ce modèle réside plus dans ses propriétés dynamiques, propriétés qui font l'objet du prochain chapitre.

Une remarque en guise de transition : le modèle structurel d'NJN comporte un type d'objets dont nous n'avons pas encore parlé : il s'agit des vecteurs. Ils sont à la base des capacités dynamiques génériques des applications. Pour cette raison, nous avons préféré les présenter dans le chapitre suivant.

CHAPITRE 5
CONSTRUCTION DES OBJETS, DISTRIBUTION ET GESTION DYNAMIQUE DE L'APPLICATION – LE MODÈLE DE RECONFIGURABILITÉ ET LE MODÈLE DE DISTRIBUTION

La structure d'une application NJN n'est pas nécessairement figée. Le modèle de reconfigurabilité permet, partant d'une page blanche, de construire une application, de la modifier, d'en adapter les paramètres, etc., pendant que la plateforme est en fonctionnement, sans qu'il ne soit jamais nécessaire d'en arrêter l'exécution. De même, le modèle de distribution permet d'ajuster dynamiquement, en cours d'exécution, la répartition de charges entre les différentes cibles de l'application.

Concernant la reconfigurabilité, deux grandes approches sont proposées par NJN. La première est assez rudimentaire. L'API fournit les fonctions nécessaires pour créer des composants, les détruire, interconnecter des variables entre elles, configurer les paramètres des tâches, etc. La seconde repose sur la notion de vecteur d'objets et permet de décrire des structures répétitives qu'NJN va pouvoir adapter automatiquement aux besoins de l'application.

Quoi qu'il en soit, tout commence par un modèle d'objet disponible dans une bibliothèque. NJN fabrique en effet les objets de l'application à partir de modèles. Les modèles sont rangés dans des paquetages et les paquetages sont compilés dans

des bibliothèques. La première opération effectuée par NJN lorsqu'il démarre est de charger les bibliothèques dont il a besoin.

La première partie de ce chapitre explique les mécanismes de création d'objet et la manière dont sont organisés les modèles. La seconde porte sur les mécanismes de reconfigurabilité dynamiques standards. Dans la troisième partie, nous aborderons les structures génériques et les vecteurs. Enfin dans la dernière partie sera traité le modèle de distribution.

1 La construction des objets, les paquetages et les bibliothèques

Comme nous l'avons dit dans l'introduction, tout commence avec un modèle.

1.1 Modèle et instance

Pour élaborer l'application, NJN utilise un constructeur générique nommé *njn_new()*.

```
njn_new(self, "var", "times",
    NJN_CONSTRAINT(dohc_is_int),
    NJN_INITVAL(dohc_int(0)));
```

La nature de l'objet construit est décrit par un modèle. Pour construire l'objet, le constructeur a besoin de connaitre également la portée locale dans laquelle créer l'objet (premier argument), son nom (troisième argument) et une liste de paramètres destinés à le configurer (quatrième argument et suivants). Le modèle renferme toutes les caractéristiques communes aux objets d'une même classe : la classe proprement dite, les paramètres de configuration acceptés, la procédure d'initialisation à appeler, etc.

Par exemple, l'appel de constructeur ci-dessus construit un nouvel objet appelé *times*, en utilisant le modèle *var*. C'est donc un objet de type *Var* qui sera créé. Il sera configuré à l'aide de deux paramètres : le premier est une contrainte de type et le second une valeur initiale.

Il existe un modèle pour chaque type d'objet structurel : composants, canaux, tâches, signaux, etc.

Dans ce second exemple, le modèle est désigné par un chemin, fourni en deuxième argument. Il s'agit du chemin *welcome::helloword*.

```
NJN_DEF_COMPONENT(test_helloword){
    NJN_LIBRARY(self, starting);
    njn_new(self, "welcome::helloworld", "hello");
    return NJN_SUCCESS;
}
```

Le second élément du chemin désigne le modèle proprement dit, le premier, le paquetage où il se situe. On trouvera ce paquetage dans la bibliothèque *starting* chargée juste avant l'appel du constructeur.

Les modèles sont donc rangés dans des paquetages, et les paquetages, dans des bibliothèques. Paquetages et bibliothèques ont pour rôle d'organiser les modèles afin d'en faciliter la gestion.

1.2 Paquetages et recherche des modèles

Un paquetage est constitué d'une liste de modèles. Pour l'essentiel, il s'agit de modèles de composants ou de canaux. Voici à titre d'exemple la définition du paquetage *welcome* qui ne comporte que le modèle *helloworld* :

```
NJN_DEF_PACKAGE(welcome){
    NJN_COMPONENT(self, helloworld);
    return NJN_SUCCESS;
}
```

La macro *NJN_DEF_PACKAGE* construit le prototype de la fonction d'initialisation du paquetage dont on lui donne le nom en argument. La macro *NJN_COMPONENT* ajoute au paquetage désigné par le premier argument le composant dont on lui fournit le nom en second argument.

Les objets de base d'NJN sont rassemblés dans un paquetage particulier, le paquetage standard, qui est chargé au démarrage de l'application. On y trouve les modèles suivants : *rt_task*, *vt_task, clock_box, decouling_box*, *var*, *param*, *handle*, *method*, etc. Chacun de ces modèles peut être appelé par le constructeur *njn_new()*. Il n'est pas utile de rappeler le paquetage *std* dans le chemin d'accès au modèle.

Pour les autres modèles, c'est-à-dire les modèles de composant ou de canaux, le chemin d'accès est constitué du nom du paquetage puis du nom du modèle séparé par l'opérateur « :: » .

Le chemin d'accès peut également être relatif. La recherche d'un modèle commence en effet dans les paquetages correspondant à la portée local. Il s'agit du paquetage d'où est issu le modèle de l'objet accueillant la future instance. Par exemple, si le modèle de composant instancié (*helloworld*) est déclaré dans le même paquetage que le composant conteneur (*test_helloworld*), le constructeur *njn_new()* se contentera du nom du modèle à instancier.

```
NJN_DEF_PACKAGE(welcome){
    NJN_COMPONENT(self, helloworld);
    NJN_COMPONENT(self, test_helloworld);
    return NJN_SUCCESS;
}

...

NJN_DEF_COMPONENT(test_helloword){
    njn_new(self, "helloworld", "hello");
    return NJN_SUCCESS;
}
```

Si la recherche dans le paquetage local n'aboutit pas, elle se poursuit dans le paquetage standard *std*. Si aucun modèle ne correspond au chemin fourni, l'instanciation n'aboutit pas.

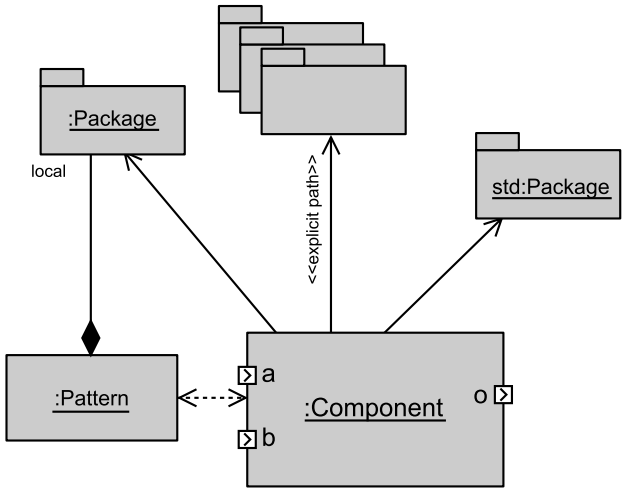


Figure 44. Paquetages accessibles depuis une instance de composant. La recherche commence dans le paquetage *local,* elle se termine dans le paquetage *std*.

1.3 Bibliothèques et gestion des dépendances

Une bibliothèque NJN est une unité de compilation. Elle prend la forme d'une *DLL* sous *Windows* ou d'un s*hared object* sous les systèmes *UNIX-Like*. Le point d'entrée d'une librairie NJN est spécifiée par la macro *NJN_DEF_LIBRARY*.

```
/* ------------------------------------------------------ */
/* starting library                                       */
/* ------------------------------------------------------ */

#include <njn.h>

NJN_DEF_PACKAGE(welcome){
   ...
}

NJN_DEF_LIBRARY(starting, 1.0){
    NJN_PACKAGE(self, welcome);
    return NJN_SUCCESS;
}
```

La macro `NJN_PACKAGE` ajoute le paquetage désigné par le second argument dans la bibliothèque représentée par le premier argument[1]. Dans l'exemple ci-dessus, le paquetage *welcome* est compilé dans la bibliothèque *starting.dll*.

Le chargement de la bibliothèque est effectué par la commande *NJN_LIBRARY*. Tous les paquetages qu'elle contient sont alors chargés dans le gestionnaire de modèles d'NJN. On peut de cette façon s'assurer que les bibliothèques dont pourrait avoir besoin un paquetage particulier sont bien chargées.

1 Le fichier de librairie doit être placé dans un dossier référencé dans le PATH global du système. A défaut, son chemin complet doit être précisé.

```
NJN_DEF_PACKAGE(test){
    NJN_LIBRARY(self, starting);
    NJN_COMPONENT(self, test_helloworld);
    return NJN_SUCCESS;
}
```

Notons que la nature du premier argument de la macro *NJN_LIBRARY* n'est pas réellement déterminante. Il peut s'agir de n'importe quel objet participant à la structure de l'application (composant, paquetage, variable, etc.). Il permet uniquement de retrouver la racine de l'arborescence de paquetages[1].

La construction des objets étant à présent définie, nous pouvons aborder la gestion de leur durée de vie.

2 La reconfiguration dynamique de l'application

La durée de vie de chaque objet de l'application est contrôlée dynamiquement. Dans un contexte de programmation classique, ce genre de choses ne poserait guère de difficulté. Mais dans NJN, le contexte distribué et plus particulièrement le protocole de synchronisation posent d'épineuses questions sur la relation entre le temps et la structure du système.

2.1 Les objets, leur ligne de vie et les retours arrière

Selon la nature de la tâche observée, la notion du temps peut être extrêmement différente. Le chapitre 2 a éclairci ce point. Pour une tâche temps-réel, le temps est assez semblable au sens commun. Pour une tâche temps-virtuel, la politique d'ordonnancement d'une part, et le protocole de synchronisation d'autre part, font qu'il en va tout autrement. Le temps propre d'une tâche temps-virtuel avance de façon croissante jusqu'à ce qu'elle soit concernée par un retour arrière. Les conséquences sur la gestion structurelle de l'application sont problématiques.

Premier problème, le même objet peut être « vu » par plusieurs tâches au même instant (du point de vue de l'horloge murale) à des dates différentes (du point de vue du temps propre des tâches). NJN doit donc conserver l'état de la structure de l'application correspondant à toutes ses dates possibles d'observation et fournir aux différentes tâches la vision structurelle de l'application qui correspond à son temps propre.

Second problème, les modifications structurelles de l'application sont opérées par des tâches temps-virtuel dont l'exécution peut être remise en cause par un retour arrière. Les modifications opérées doivent dans ce cas être effacées de la structure de données de l'application, comme si elles n'avaient jamais été effectuées. Il faut donc conserver la trace des relations causales entre l'exécution des tâches et les opérations structurelles pour pouvoir revenir sur les opérations correspondantes.

1 Il n'y a aucune variable globale dans NJN, mais on peut accéder à n'importe quel point du système à partir d'un objet qui en fait partie.

Pour résoudre ces deux difficultés, NJN doit gérer la structure de l'application comme il gère les données transportées par les variables, c'est-à-dire au moyen de signaux et d'évènements d'écriture.

Ainsi chaque objet de la structure hiérarchique de l'application est associé à un signal particulier, sa ligne de vie, dont le rôle est de dire si l'objet existe ou non à l'instant d'observation. Ce signal est visible à travers la propriété *lifeline* de chaque objet. Il contrôle le comportement de l'objet tout au long de sa durée de vie. Une variable ne peut être reliée à une autre que si elle existe bel et bien à la date d'observation. La connexion elle-même peut être active ou non en fonction du temps. Lorsque l'objet considéré est une tâche, celle-ci ne peut être exécutée que lorsque sa ligne de vie est valide. Etc.

La ligne de vie est donc la propriété fondamentale du modèle de reconfigurabilité dynamique d'NJN. Voyons concrètement comment en manipuler l'état.

2.2 De *njn_new()* à *njn_delete()* – la durée de vie des objets

En règle générale, lorsqu'on veut créer un objet, on place l'appel au constructeur *njn_new()* dans la fonction d'initialisation du conteneur dans lequel il sera situé. La durée de vie d'un objet créé de cette façon est liée à celle de son conteneur. L'objet est créé au moment de la construction du conteneur et est détruit lorsque celui-ci est détruit.

```
NJN_DEF_COMPONENT(helloworld){
    njn_new(self, "var", "times",
        NJN_CONSTRAINT(dohc_is_int),
        NJN_INITVAL(dohc_int(0)));
    njn_new(self, "rt_task", "t",
        NJN_ALWAYS(helloworld_t),
        NJN_TIMESLOT(1 SEC, 0, 0));
    return NJN_SUCCESS;
}
```

Mais NJN permet aux tâches temps-virtuel de modifier la structure de l'application, avec certaines limites cependant. L'objet à modifier doit être déclaré *dynamic,* afin qu'NJN lui associe une ligne de vie qui lui soit propre. De plus, une tâche ne peut modifier que les éléments qui se trouvent dans la même portée qu'elle, c'est-à-dire, les tâches, les variables et les composants ou canaux situés dans le même composant qu'elle et au même niveau hiérarchique.

Les opérations possibles peuvent être classées en trois catégories selon qu'elles portent sur un objet quelconque, spécifiquement sur les tâches ou spécifiquement sur les objets appartenant au réseau (variables et canaux).

Voici les fonctions de manipulation générales :

- *njn_new()* : créer un objet quelconque ;
- *njn_delete()* : détruire un objet quelconque ;
- *njn_set_name()* : modifier le nom d'un objet quelconque.

Les propriétés des tâches peuvent être modifiées par les fonctions suivantes :

- *njn_set_latency()* : modifier la valeur de la latence ;
- *njn_set_timeout()* : modifier la valeur du *timeout* ;
- *njn_set_timeslot()* : modifier le *timeslot.*

Bien que celle-ci ne soit pas une propriété, rappelons qu'il est également possible de modifier la priorité d'une tâche. En revanche, la liste de sensibilité ou le corps d'une tâche sont des éléments statiques. Ils ne peuvent pas être modifiés après la construction de la tâche.

Les variables et les canaux peuvent être connectés ou déconnectés dynamiquement :

- *njn_connect()* : connecter une variable ou un canal à une source ;
- *njn_disconnect()* : déconnecter une variable ou un canal d'une source ;
- *njn_assign()* : associer à une variable de la portée locale le port d'un composant situé dans la même portée ;
- *njn_unassign()* : dissocier d'une variable de la portée locale le port d'un composant situé dans la même portée ;

Lorsqu'on engage des modifications structurelles, il faut rester extrêmement prudent et se rappeler que les tâches temps-virtuel sont toujours exécutées en retard par rapport à l'horloge murale.

2.3 Assurer la continuité de service

Les modifications structurelles ne peuvent être effectuées que par les tâches temps-virtuel. Or, celles-ci opèrent toujours avec retard vis-à-vis de l'horloge murale. Un problème se pose donc si l'on tente de faire disparaitre ou apparaitre une tâche temps-réel. L'opération doit en effet être faite dans la structure de données d'NJN avant que l'horloge murale atteigne la date de la modification sans quoi un paradoxe temporel apparait. Il faut donc anticiper.

L'exemple ci-dessous montre le remplacement d'une tâche temps-réel par une autre.

Le composant décrit comporte deux tâches. La première se nomme *h* et est temps-réel. Sa latence est d'une seconde et elle se déclenche périodiquement toutes les secondes. La seconde se nomme *m* et se déclenche au bout d'un *timeout* de 4 secondes.

```
NJN_DEF_COMPONENT(helloworld){
    njn_new(self, "rt_task", "h",
        NJN_ALWAYS(hello_t),
        NJN_TIMESLOT(1 SEC, 0, 0),
        NJN_LATENCY(1 SEC),
        NJN_DYNAMIC);
    njn_new(self, "vt_task", "m",
        NJN_ALWAYS(manager_t),
```

```
            NJN_TIMEOUT(4 SEC));
        return NJN_SUCCESS;
    }
```

Le corps de la tâche *m* est le suivant :

```
NJN_DEF_ALWAYS(manager_t){
    njn_ref_t h;
    switch(event){
    case NJN_TIMEOUT_EVENT:
        h = njn_get(local, "h");
        njn_delete(h);
        njn_new(local, "rt_task", "b",
            NJN_ALWAYS(byby_t),
            NJN_TIMESLOT(1 SEC, 0 MSEC, 0),
            NJN_LATENCY(1 SEC),
            NJN_DYNAMIC);
        njn_delete(self); //arakiri
    }
    return NJN_SUCCESS;
}
```

Lorsqu'elle est exécutée, elle détruit la tâche *h* et crée une nouvelle tâche temps-réel nommée *b*, puis s'autodétruit. La figure 45 montre le déroulement des opérations selon un axe temporel virtuel.

On observe sur le chronogramme une discontinuité de service. Un délai de 2 secondes sépare la dernière exécution de la tâche *h* de la première exécution de la tâche *b*. Cette discontinuité est due au fait que pour les tâches temps-réel, une durée égale à la latence de la tâche sépare la première activation de la première exécution. Le déclenchement de la tâche *b* intervient au premier *slot* de temps inclus de la zone active de la tâche mais ne provoque son exécution qu'une seconde plus tard.

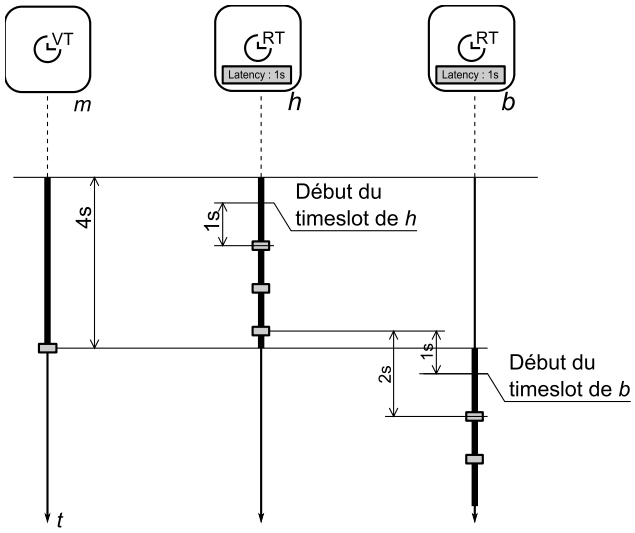


Figure 45. Remplacement d'une tâche par une autre, observé selon une échelle temps-virtuel.

Pour assurer la continuité de service, il faut donc anticiper la création de la seconde tâche *b* d'une seconde, de manière à ce que sa latence soit écoulée avant la première occurrence d'exécution souhaitée. Il apparait un chevauchement d'activité entre les tâches *h* et *b* correspondant à la valeur de leur latence.

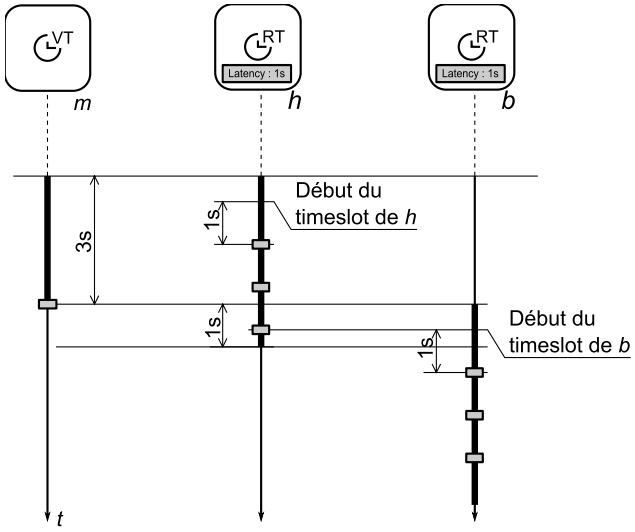


Figure 46. Création anticipée de la seconde tâche.

Si l'on regarde le déroulement des opérations d'un point de vue temps-réel, un second problème apparait (figure 47).

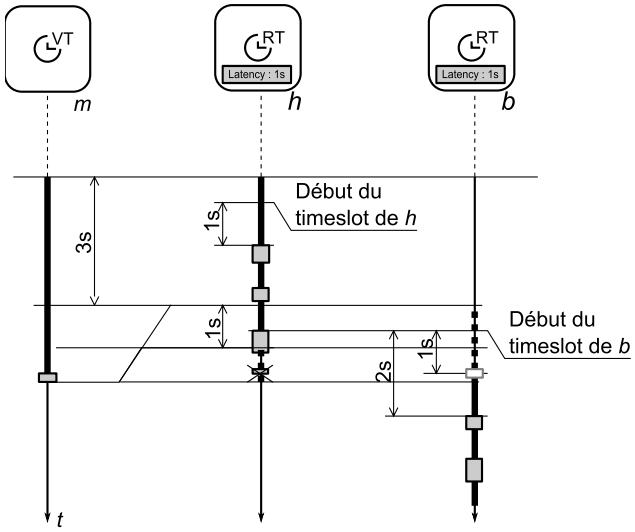


Figure 47. En temps-réel.

La tâche *m* est en réalité exécutée avec un certain retard (très exagéré ici pour mettre en évidence le problème) si bien que les modifications structurelles sont opérées bien au-delà de leur échéance. La tâche *h* est exécutée une fois de trop tandis que la tâche *b* manque sa première exécution.

Pour éviter ce phénomène fort fâcheux, il convient d'anticiper l'ensemble des opérations. Le code de l'application devient le suivant :

```
NJN_DEF_COMPONENT(helloworld){
    njn_new(self, "rt_task", "h",
        NJN_ALWAYS(hello_t),
        NJN_TIMESLOT(1 SEC, 0, 0),
        NJN_LATENCY(1 SEC),
        NJN_DYNAMIC);
    njn_new(self, "vt_task", "m",
        NJN_ALWAYS(manager_t),
        NJN_TIMEOUT(2 SEC));
    return NJN_SUCCESS;
}
```

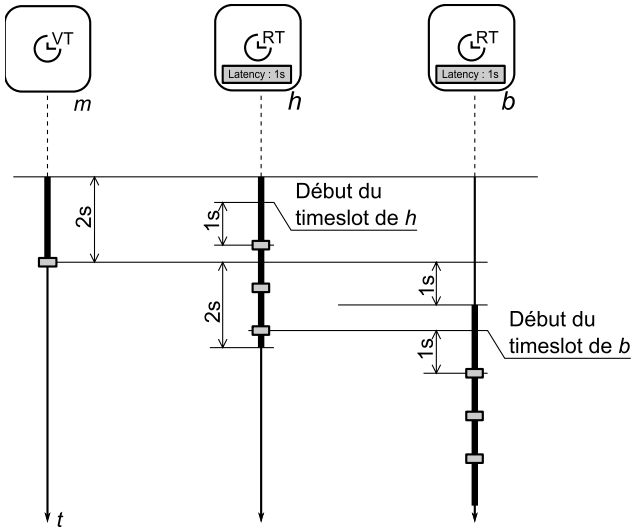


Figure 48. La solution.

La tâche *m* est exécutée 2 secondes avant l'échange effectif de tâche *h*. Le corps de la tâche *m* devient :

```
NJN_DEF_ALWAYS(manager_t){
    njn_ref_t h;
    switch(event){
    case NJN_TIMEOUT_EVENT:
        h = njn_get(local, "h");
        njn_delete(h, NJN_AFTER(2000 MSEC));
        njn_new(local, "rt_task", "b",
            NJN_ALWAYS(byby_t),
            NJN_TIMESLOT(1 SEC, 0 MSEC, 0),
            NJN_LATENCY(1 SEC),
            NJN_DYNAMIC,
            NJN_AFTER(1000 MSEC));
        njn_delete(self); //arakiri
    }
    return NJN_SUCCESS;
}
```

La destruction de la tâche *h* intervient 2 secondes après l'exécution de *m*. De même, la création de la tâche *b* est effectuée 1 seconde après l'exécution de *m*.

La figure 48 montre la séquence des opérations vues en temps-virtuel tandis que la figure 49 montre la même séquence, cette fois selon un point de vue temps-réel.

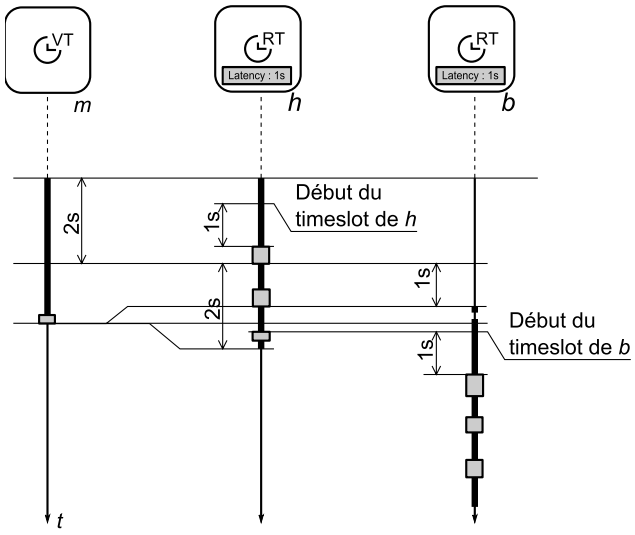


Figure 49. En temps-réel

Cette fois la continuité de service est pleinement assurée et aucun aléa n'apparait au cours de la reconfiguration de l'application.

Nous venons de voir qu'un certain nombre d'opérations sont possibles sur la structure de l'application pendant qu'elle est en exécution. Ces opérations doivent être menées selon une séquence convenablement orchestrée d'un point de vue temporel afin d'assurer la continuité de service de l'application pendant la reconfiguration. Le type d'opérations dont il est question ici est atomique ; opération après opération, on parvient à faire évoluer la structure de l'application. La construction de systèmes génériques auto-adaptatifs peut s'avérer relativement fastidieuse. NJN offre une autre possibilité basée sur des opérateurs algorithmiques structurels.

3 Vecteurs et algorithmes structurels

NJN propose un ensemble de primitives d'algorithmie structurelle qui s'appuie sur la notion de vecteur d'objets dynamique. Il est possible, par exemple, de répéter la même structure d'opérateurs de manière à ce que chaque itération structurelle traite l'un des éléments d'un vecteur de variables. Les mécanismes d'algorithmique structurelle proposés par NJN constituent de puissants outils de paramétrage dynamique d'application.

3.1 Objets et vecteurs d'objets

Les vecteurs NJN ne sont pas destinés à gérer les données vectorielles ou matricielles comme des images par exemple. Ils correspondent à un modèle hiérarchique de plus haut niveau. Leur rôle est de faciliter la gestion de collections d'objets de même nature, appartenant à un même conteneur et dont le nombre peut éventuellement évoluer dynamiquement au cours de l'exécution de l'application.

Même si un vecteur peut gérer une collection de n'importe quel type d'objet, ses éléments seront le plus souvent des variables. Voici à titre d'exemple la définition d'un vecteur de variables nommé *e* :

```
njn_ref_t e = njn_new(self, "vector", "e",
    NJN_ELEMTYPE("var",
        NJN_CONSTRAINT(dohc_is_float),
        NJN_INITVAL(dohc_float(0.0))
    )
);
```

Le modèle sollicité est le modèle *vector*. La nature de chaque élément est explicité par l'argument *NJN_ELEMTYPE*. Celui-ci rassemble tous les renseignements utiles pour créer un nouvel élément : le modèle d'abord, puis les différents paramètres de configuration.

Quelques opérations élémentaires sont possibles. On peut par exemple manipuler les éléments d'un vecteur par l'intermédiaire de son indice.

```
njn_ref_t elem0 = njn_vget(e, 0);
njn_ref_t elem1 = njn_vnew(e, 1);
njn_ref_t elem2 = njn_vnew(e, -1); //adds a new element at end
njn_vdelete(e, 1);
njn_vdelete(e, -1); //deletes the last element
```

On peut aussi manipuler directement le nombre d'éléments contenus dans le vecteur. Le dimensionnement dynamique d'un vecteur met en œuvre, de façon automatique, les mécanismes de création ou de destruction des éléments qui le composent.

```
int64_t size = NJN_GET_VSIZE(e);
NJN_SET_VSIZE(e, size + 2);
```

Cependant, ce n'est pas la manière la plus élégante de manipuler ces collections d'objets. Des opérateurs permettent d'ajuster automatiquement la taille d'un vecteur aux besoins de l'application.

3.2 Auto-dimensionnement, pseudo-composants *mux* et *demux*

Il est possible de connecter les vecteurs entre eux si la classe des éléments qui les compose le permet.

```
njn_ref_t v1 = njn_new(self, "vector", "v1", NJN_ELEMTYPE("var"));
njn_ref_t v2 = njn_new(self, "vector", "v2", NJN_ELEMTYPE("var"));

njn_connect(v1, v2); // connect v2 to source v1
```

Au moment de la connexion, la taille du vecteur le plus petit est ajustée sur la taille du vecteur le plus grand. Une fois la connexion réalisée, les tailles des vecteurs concernés sont liées. Si un élément d'un des deux vecteurs est supprimé, l'élément de même indice est supprimé également dans l'autre vecteur.

Pour contrôler les extrémités du réseau ainsi constitué, deux pseudo-composants sont fournis qui ne sont compatibles qu'avec les vecteurs de variables : un opérateur nommé *mux* et un opérateur nommé *demux*. Le premier permet d'éclater un vecteur en plusieurs variables d'entrées tandis que le second permet d'éclater un vecteur en plusieurs variables de sorties.

Voici les pseudo-définitions de ces deux opérateurs :

```
NJN_DEF_COMPONENT(mux){
    njn_new(self, "vector", "out", NJN_ELEMTYPE("var"), NJN_OUTPUT);
    njn_new(self, "var", "in0", NJN_INPUT);
    ...
}

NJN_DEF_COMPONENT(demux){
    njn_new(self, "vector", "in", NJN_ELEMTYPE("var"), NJN_INPUT);
    njn_new(self, "var", "out0", NJN_OUTPUT);
    ...
}
```

Un *mux* (respectivement un *demux*) comporte toujours une entrée (respectivement une sortie) non-connectée. Lorsqu'on lui connecte une variable, le pseudo-composant en crée une nouvelle et adapte automatiquement la taille du vecteur de sortie. La modification est répercutée sur l'ensemble du réseau.

```
njn_ref_t v = njn_new(self, "vector", "v", ...);
njn_ref_t m = njn_new(self, "mux", "m", ...);
njn_ref_t e0, e1;

e0 = njn_new(self, "var", "e0", ...);
njn_vassign(m, e0);
e1 = njn_new(self, "var", "e1", ...);
njn_vassign(m, e1);
```

De même, lorsqu'une entrée est déconnectée, l'élément correspondant est supprimé du réseau de vecteurs associés.

```
e0 = njn_get(self, "e0");
njn_vunassign(m, e0);
```

3.3 Algorithmies structurelles

NJN offre des mécanismes de description générique de code très puissants basés sur les vecteurs et des blocs de génération.

Le modèle de graphe flot de données factorisé décrit par Lavarenne [48] permet de factoriser un schéma structurel répétitif au sein d'un graphe flot de données.

L'objectif poursuivi par Lavarenne est d'optimiser l'utilisation de ressources matérielles limitées [20]. Le nôtre est bien différent puisque nous cherchons à généraliser un schéma structurel en fonction de vecteurs de données dont la taille peut évoluer dynamiquement au cours du temps. Mais le modèle qu'il propose répond parfaitement à notre problème.

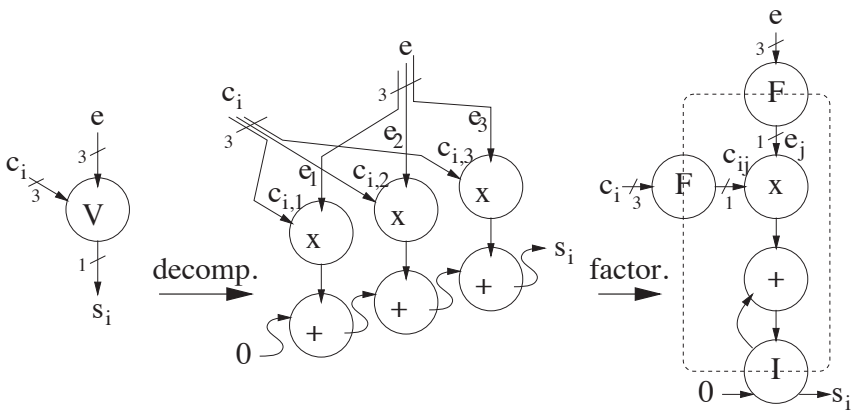


Figure 50. Décomposition et factorisation d'un produit scalaire (source [48]).

Il repose sur le placement de frontières de factorisation qui assurent la synchronisation des opérateurs factorisés (à l'intérieur de la frontière) avec l'extérieur de la frontière. Des opérateurs spécifiques gèrent la synchronisation des données à travers la frontière :

- Un opérateur *D* (*diffuse*) diffuse la même donnée pour chaque itération de la zone factorisée ;
- Un opérateur *F* (*fork*) multiplexe chaque élément d'un vecteur de données à une itération de la zone factorisée ;
- Un opérateur *I* (*iterate*) réinjecte à chaque itération une donnée de l'itération précédente ;
- Un opérateur *J* (*join*) démultiplexe les données produites par toutes les itérations au sein d'un vecteur de sortie.

Lavarenne donne l'exemple de la décomposition du produit scalaire de deux vecteurs (figure 50). L'algorithme de calcul du produit scalaire est un schéma répétitif. Les résultats de la multiplication terme à terme des vecteurs d'entrée sont sommés pour former la donnée de sortie. Sa factorisation consiste à placer un multiplexeur (opérateur *F*) sur chaque vecteur d'entrée et un itérateur (opérateur *I*) pour réaliser la somme des différents produits.

NJN propose un mécanisme similaire pour décrire des structures répétitives et auto-adaptatives. Cela passe par la définition d'un bloc (*generate*) dont le contenu peut être reproduit pour chaque élément d'un ou plusieurs vecteurs de même taille. Le schéma de factorisation est directement induit par les opérateurs de frontières

apparaissant dans la définition du bloc. L'exemple ci-dessous traite le produit scalaire de deux vecteurs selon cette approche.

Un bloc de code correspondant à une itération du schéma structurel est défini.

```
NJN_DEF_ALWAYS(sp){
    ...
    return NJN_SUCCESS;
}

NJN_DEF_BLOCK(v){
    njn_ref_t c      = njn_new(self, "var", "c", NJN_INPUT);
    njn_ref_t e      = njn_new(self, "var", "e", NJN_INPUT);
    njn_ref_t s_prev = njn_new(self, "var", "s_prev", NJN_INPUT);
    njn_new(self, "var", "s_next", NJN_OUTPUT);
    njn_new(self, "vt_task", "sp",
        NJN_ALWAYS(sp), NJN_WHEN(c, e, s_prev));
    return NJN_SUCCESS;
}
```

Les entrées et sorties du bloc correspondent aux opérateurs de la frontière de factorisation. Au niveau hiérarchique supérieur, ils sont associés à des vecteurs par l'intermédiaire d'opérateurs *NJN_FORK* ou *NJN_ITER* afin d'établir les règles de développement du schéma répétitif. De façon générale, les entrées du bloc peuvent être associées à des sommets de type *F*, *D* ou *I* et les sorties à des sommets de type *J* ou *I*.

```
NJN_DEF_COMPONENT(sp){
    njn_ref_t c = njn_new(self, "vector", "c",
        NJN_ELEMTYPE("var"),
        NJN_INPUT);
    njn_ref_t e  = njn_new(self, "vector", "e",
        NJN_ELEMTYPE("var"),
        NJN_INPUT);
    njn_ref_t s  = njn_new(self, "var", "s",
        NJN_OUTPUT);
    njn_new(self, "generate",  "v",
        NJN_BLOCK(v),
        NJN_FORK("c", c),
        NJN_FORK("e", e),
        NJN_ITER("s_prev", "s_next", dohc_float(0.0), s));
    return NJN_SUCCESS;
}
```

NJN gèrera les itérations du schéma ainsi défini en fonction des opérations faites sur les vecteurs connectés aux sommets de factorisation.

4 La distribution

Complémentaire de la reconfiguration dynamique, la distribution permet de répondre aux besoins de deux cas de figures radicalement différents : d'une part, le problème de la séparation matérielle entre les entrées/sorties du système (capteurs, actionneurs, etc.) et les moyens de gestion ou de calcul et, d'autre part, le problème de la répartition de la charge de calcul sur plusieurs unités de traitement.

Les caractéristiques de la distribution liées à la première situation sont prévisibles dès la spécification du système. Tel capteur se trouve associé à un premier calculateur, tel autre à un second, etc. de même pour les actionneurs. On peut alors prévoir pour ces dispositifs une distribution plus ou moins gravée dans l'architecture de l'application en procédant comme le font par exemple [RT]MAPS ou ou Simulink : positionner des opérateurs destinés à établir la communication entre deux dispositifs distants.

Concernant le problème de la répartition de charge, les choses sont un peu différentes. La structure de la distribution n'est pas liée à l'application elle-même mais à une question d'optimisation dynamique. Outre les risques de dépassement de latence, l'application a le même comportement quelque soit la structure de la distribution. Il est donc préférable de s'orienter vers une distribution plus transparente.

Les concepts mis en œuvre pour satisfaire ces deux visions de la distribution ne sont pas très complexes. Mais il est important de garder à l'esprit le service rendu par les objets décrits ici, lorsque l'on souhaite développer des applications distribuées.

Dans cette partie, on aborde successivement la réponse d'NJN aux deux situations décrites plus haut. Encore une fois, c'est le point de vue du développeur d'application qui est adopté ici.

4.1 Connexions point à point – Les prises et les fiches

La distribution point-à-point consiste à connecter, à travers le réseau, deux variables ou deux canaux présents physiquement sur deux processus logiques distincts. L'une des variables est appelée une fiche (*plug*) et l'autre une prise (*socket*). La prise est une connexion ouverte sur laquelle une fiche vient se connecter.

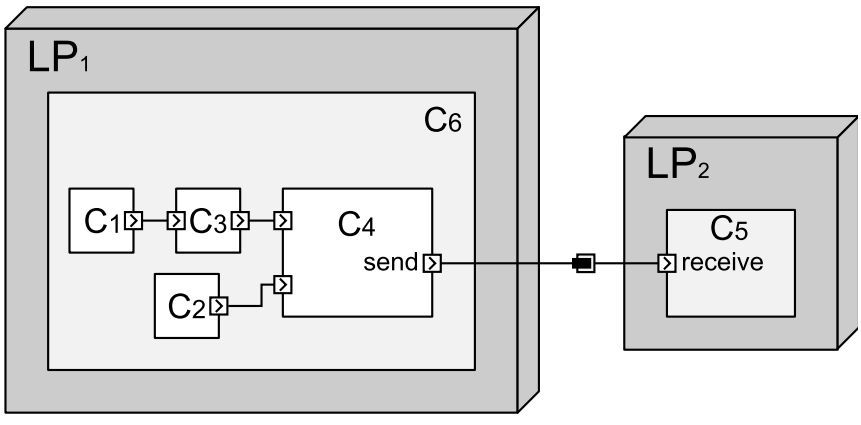


Figure 51. Une connexion point-à-point.

Le sens de transit des informations est indépendant de la sémantique de connexion. Fiches et prises sont déclarées un peu à la manière des entrées/sorties

de composants. L'une comme l'autre peuvent être déclarées en entrée ou en sortie. Pour pouvoir être connectées, elles doivent être compatibles en termes de nature d'objet et de sens de transfert des données : une fiche d'entrée de type variable peut se connecter à une prise de sortie de type variable ; une fiche de sortie de type canal peut se connecter à une prise d'entrée d'un type compatible de canal, etc.

```
// Sur LP1 (localhost:2001)
NJN_DEF_COMPONENT(C4){
    njn_ref_t lp2 = cables_get_host(njn_get_cables(self),
        "localhost:2002");
    njn_ref_t send  = njn_new(self, "var", "send",
        NJN_OUTPUT,
        NJN_PLUG);
    njn_plugin(send, lp2, "C5_receive");
    ...
    return NJN_SUCCESS;
}
// Sur LP2 (localhost:2002)
NJN_DEF_COMPONENT(C5){
    njn_new(self, "var", "receive",
        NJN_INPUT,
        NJN_SOCKET("C5_receive"));
    ...
    return NJN_SUCCESS;
}
```

Les prises sont déclarées par l'intermédiaire du paramètre de configuration *NJN_SOCKET*. Elles doivent comporter une étiquette qu'elles sont seules à porter sur le processus logique. Dans notre exemple, la variable *receive* du composant *C5* correspond à la prise *C5_receive* du processus logique *lp2*.

Une fiche n'est pas visible sur le réseau. On la déclare par le paramètre de configuration *NJN_PLUG*. On peut alors la connecter à une prise par la commande *njn_plugin*. Il faut dans ce cas fournir la référence du processus logique obtenu grâce à la bibliothèque *CABLES*[1] et le nom de la prise sur laquelle elle se connecte. La déconnexion est réalisée par un appel à *njn_unplug*.

4.2 Distribution – Les composants et leur ombre

La distribution de composants est un peu plus adaptée au travail de répartition de charge. L'opération est un peu plus transparente vis-à-vis de l'application. Elle consiste à héberger un composant entier sur un processus logique distant. Localement, l'ombre du composant (*shadow component*) permet de manipuler l'instance distante comme si elle était instanciée localement.

```
NJN_DEF_COMPONENT(C6){
    njn_ref_t a = njn_new(self, "var", "a");
    njn_ref_t b = njn_new(self, "var", "b");
    njn_ref_t lp3 = cables_get_host(njn_get_cables(self),
        "localhost:2003");
    njn_new(self, "inst3", "C3",
```

1 Voir page 157.

```
            NJN_ASSIGN("in", a),
            NJN_ASSIGN("out", c),
            NJN_HOST(lp3));
        ...
        return NJN_SUCCESS;
    }
```

Outre le paramètre de configuration *NJN_HOST* qui précise le processus hôte du composant instancié, la création du composant est identique à une instanciation classique. Les connexions entre ports et variables sont établies à travers le réseau, en utilisant les primitives d'envoi de messages d'NJN. Le constructeur *njn_new* crée localement un composant ombre sur lequel on pourra opérer les opérations classiques de manipulation d'objets : connexion ou déconnexion de variables de port, destruction de l'instance, etc. Ces opérations seront répercutées sur le composant réel.

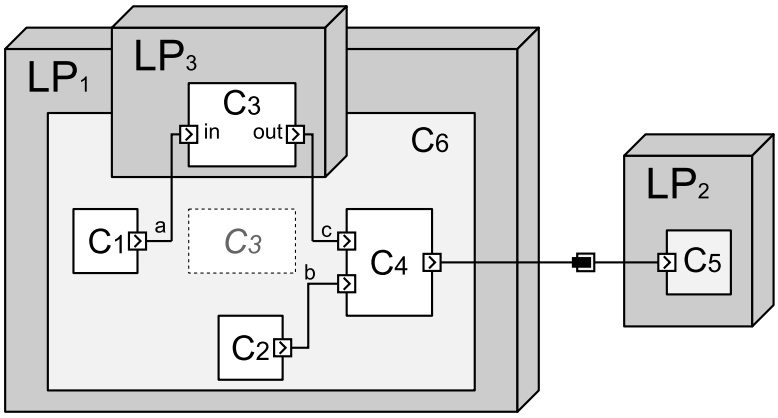


Figure 52. Une application distribuée.

La distribution peut être opérée dynamiquement. Il suffit pour cela de détruire l'instance à déplacer par un appel à *njn_delete* et de la reconstruire sur un autre processus logique en fournissant au constructeur *njn_new* le paramètre de configuration *NJN_HOST* correspondant.

Pour que l'instanciation se déroule correctement, la bibliothèque où est situé le modèle du composant doit être disponible sur la machine hôte car seule la référence du modèle et les informations d'instanciations sont communiquées au processus d'hébergement.

*
* *

Nous venons de passer en revue les différents aspects de la reconfigurabilité dynamique et de la distribution dans NJN.

A partir de modèles de composants compilés dans des bibliothèques et rangés dans des paquetages, une application NJN peut créer dynamiquement des instances de composants. Par défaut, les objets ainsi créés ont la même durée de vie que leur conteneur.

Mais nous avons vu qu'il existait des mécanismes de reconfigurabilité dynamiques qui permettaient d'appliquer d'autres règles. Deux mécanismes complémentaires existent. Le première concerne des manipulations directes de la structure de l'application. Lorsque les objets sont déclarés *dynamic*, ils peuvent être modifiés ou détruits dynamiquement. On peut connecter des variables entre elles ou les déconnecter, ajuster les paramètres des tâches, etc. Le second mécanisme repose sur la définition de schémas structurels génériques qu'NJN va pouvoir adapter automatiquement aux besoins de l'application.

La distribution d'une application est une chose relativement simple à réaliser. A partir du moment où les différents processus logiques d'hébergement sont connus et référencés, il ne reste plus qu'à préciser où doit être instancié quel composant et quelle variable doit être connectée à quelle autre. La mise en œuvre des communications est faite par le noyau d'NJN qui s'appuie pour cela sur le protocole de synchronisation vu précédemment. En définitive, dès lors qu'on dispose d'un protocole de synchronisation adapté, les concepts exposés ci-dessus sont relativement simples à mettre en œuvre.

La réelle difficulté de la reconfiguration dynamique et de la distribution réside dans l'implémentation du protocole de synchronisation qui permet à toutes ces manipulations d'objets de subir des retours arrière. Cette implémentation sera abordée dans la prochaine partie de ce manuscrit.

Notons d'ores et déjà que l'implémentation actuelle d'NJN ne supporte pas toutes les constructions décrites dans ce chapitre. Tant qu'il ne s'agit pas de vecteurs, la reconfigurabilité est fonctionnelle dans l'implémentation actuelle d'NJN. Les manipulations structurelles par vecteur ont fait l'objet d'un démonstrateur écrit en langage *Ruby* qui a permis de valider le concept mais elles ne sont pas encore implémentées dans la version actuelle de la plateforme. Concernant la distribution, nous avons préféré mettre l'accent sur ce qu'NJN propose de plus original. Ainsi la distribution de composants est implémentée. En revanche, les connexions point-à-point ne sont pas encore fonctionnelles mais ce concept n'étant pas nouveau, il ne pose aucun problème particulier d'implémentation.

CHAPITRE 6
ÉTUDE DE CAS : LE PROJET COREBOTS

Le projet *CoreBots* a été développé en parallèle du projet AROS. L'équipe *CoreBots* participe à un défi de robotique organisé par la DGA et l'ANR. Il s'agit de concevoir un robot d'exploration et de cartographie autonome.

Nous avons choisi d'exploiter les avancés du projet AROS dans la conception du robot *CoreBot M* qui concourt à ce défi car la conception logicielle du robot sollicite potentiellement presque tous les principes exposés dans les chapitres précédents. Ce choix a été à la fois un fort catalyseur pour le projet AROS et une très forte contrainte en termes de complexité d'application et de respect des délais. Au-delà de la conception d'un prototype, il s'agissait de gagner une compétition.

Ce chapitre présente la conception de l'architecture logiciel du robot *CoreBot M*. Il montrera comment NJN facilite la réalisation d'exécutifs temps-réel distribués complexes en prenant en charge les problèmes de synchronisation des données et de distribution des tâches. La première partie présente le défi CAROTTE, l'équipe et l'architecture matérielle du robot *CoreBot M*. La seconde partie porte sur son architecture logicielle, indépendamment de tout choix d'implémentation. Enfin, la dernière partie propose une solution d'implémentation exploitant le *framework* NJN.

1 Présentation du projet et du robot

1.1 Le défi CAROTTE[1]

Que ce soit en milieu naturel ou urbain, il est fréquent que l'environnement dans lequel évoluent les robots soit mal connu ou/et évolutif. Cette incertitude est préjudiciable à la réalisation des missions, notamment celles concernant l'exploration de zones dangereuses.

Dans ce contexte, de petits engins terrestres non habités (UGVs) peuvent être utilisés pour suppléer l'homme grâce à leurs capacités de reconnaissance. Une des facultés essentielles de ces systèmes robotisés est leur capacité à collecter de l'information sur leur environnement, et à l'analyser afin de fournir des informations sur la configuration des lieux (cartographie) et la reconnaissance et localisation d'objets d'intérêt. L'autonomie maximale des robots doit aller de paire avec la robustesse du système vis-à-vis par exemple des interruptions de communication.

Figure 53. Vue des arènes du défi CAROTTE.

Pour améliorer les capacités de localisation, de cartographie de bâtiments et d'analyse de terrain en milieu urbain, la DGA et l'ANR ont initié un défi intitulé CAROTTE (CArtographie par RObot d'un Territoire).

Objectifs du défi :

1 Extrait de www.defi-carotte.fr

- Faire progresser l'innovation et l'état de l'art en robotique dans le domaine perception / cognition pour des applications duales (défense et sécurité, protection civile, assistance à domicile, robot compagnon).
- Susciter des rapprochements entre chercheurs et industriels issus de la robotique et de domaines connexes (réalité augmentée, jeu, analyse image, indexation, sémantique,...).

Ce défi permettra de vérifier la capacité des petits robots terrestres pour des missions de reconnaissance en milieu fermé non totalement structuré.

Des innovations sont attendues dans le domaine de l'intelligence artificielle embarquée (perception, reconnaissance, fusion de données, cartographie sémantique, localisation en intérieur, architecture de contrôle et autonomie) avec des possibilités d'avancées dans d'autres domaines (mobilité, planification de mission et supervision, interfaces homme – machine…).

Figure 54. Le robot *CoreBot M*.

Les épreuves du concours s'appuient sur une mission de type reconnaissance d'une zone par un système robotisé autonome. Cette dernière consiste en une navigation en autonome de ce système dans une arène (une zone fermée constituée d'un ensemble de pièces). Elle doit aboutir à la reconnaissance automatique des objets présents dans l'arène. La reconnaissance de la nature des sols et des parois constituant l'arène est encouragée. Le résultat attendu en fin de mission est une cartographie de la zone accompagnée d'annotations sémantiques.

Le système robotisé devra accomplir sa mission en autonome, sans aucune intervention ni contrôle extérieur (en particulier humain) et en temps contraint.

1.2 L'équipe *CoreBots*

L'équipe *CoreBots* a été créée pour participer au défi CAROTTE. Le consortium est constitué de partenaires académiques (écoles et laboratoires de recherche) et de petites entreprises spécialisées dans la robotique. L'équipe rassemble des spécialistes d'intégration matérielle, d'architecture logicielle et d'algorithmie (SLAM, vision et contrôle).

Les partenaires sont les suivants :

- Centre de Robotique ARMINES (Mines ParisTech) ;
- La société Intempora ;
- Le centre de recherche INRIA (équipe IMARA)
- Epitech

1.3 Le robot *CoreBot M*

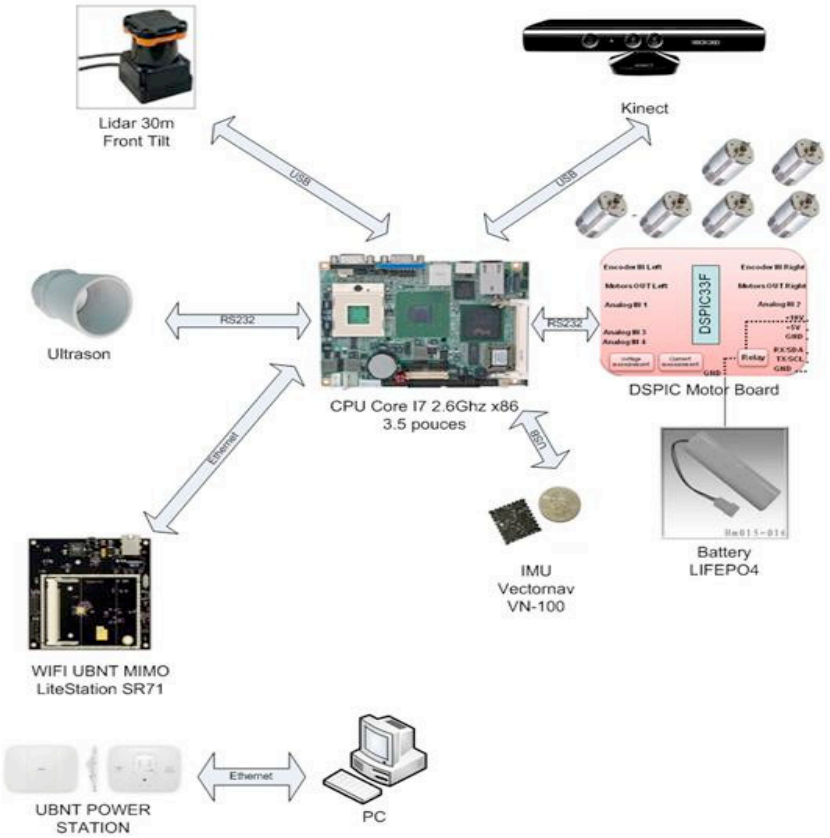


Figure 55. Architecture matérielle du robot.

Le robot *CoreBot M* (voir figure 54) est basé sur une plateforme ouverte du commerce, le *WIFIBOT LAB M Lab* (www.wifibot.com).

Il est doté de 6 roues, ce qui lui permet d'évoluer en extérieur et d'avoir de grandes capacités de franchissement. Il est équipé d'une carte *Core I7* embarqué sous Linux, d'un télémètre laser *Hokuyo 30m*, d'une caméra 3D *Kinect,* d'une centrale inertielle *VectorNav* et d'un ultrason directif.

L'organe central du robot est la carte mère à base de *Core i7 620M* faisant tourner Ubuntu 10.04 LTS.

Le système de communication est aussi sous Linux et utilise la technologie MIMO. Il permettra de déporter certains composants logiciels dont l'exécution est trop gourmande en ressources processeur.

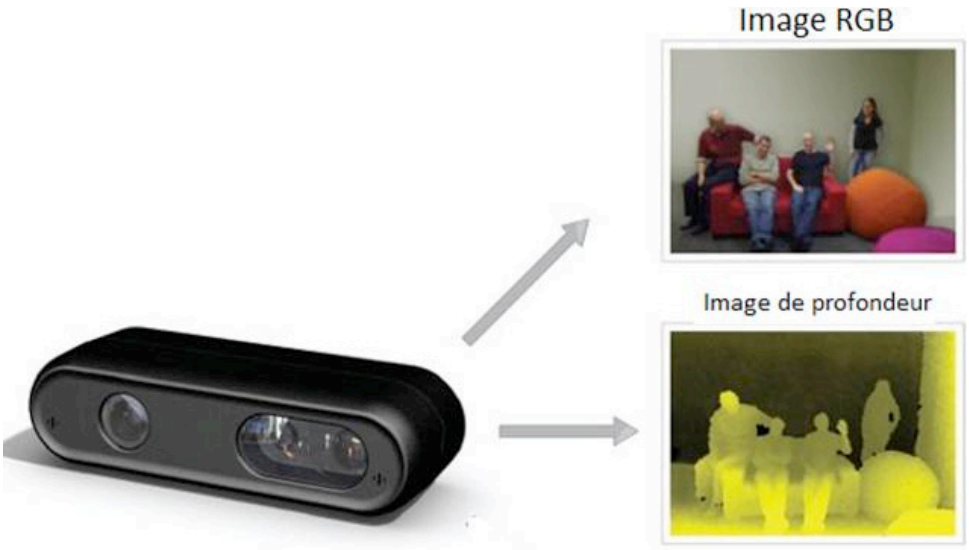


Figure 56. La caméra 3D *Kinect*.

Le télémètre laser *Hokuyo* est un scanner 2D à balayage. Il fournit au robot une coupe horizontale de l'environnement située à 40 cm du sol. Sa résolution est de plus ou moins 3 cm a 10 m de distance. La cartographie de l'arène est obtenue principalement grâce aux données fournies par le laser.

La centrale inertielle du fabriquant *VectorNav* va permettre de gérer les pentes et les changements de niveaux. Elle intègre un filtrage de Kalman qui permet d'obtenir l'attitude du robot à 200Hz.

Pour détecter la présence de surfaces transparentes ou réfléchissantes, le robot utilise un capteur ultrason directif à l'avant du robot d'une portée de 7m. Il va permettre de rajouter sur la carte des obstacles que le laser ne peut pas distinguer comme les vitres ou les miroirs.

La caméra 3D *Kinect* est utilisée pour l'évitement d'obstacles et la reconnaissance d'objets. Avec un angle de vue de 60°, elle transmet sur le même bus USB une image RGB (ou YUV) et une image de profondeur (figure 56).

A l'aide de cette architecture, le robot doit produire une cartographie de la zone d'exploration en y ajoutant des informations sur les objets repérés dans l'environnement.

2 Analyse du problème

2.1 L'architecture logicielle

Le système se décompose naturellement en trois classes de composants : les capteurs/actionneurs, les composants de traitement, et les composants de supervision.

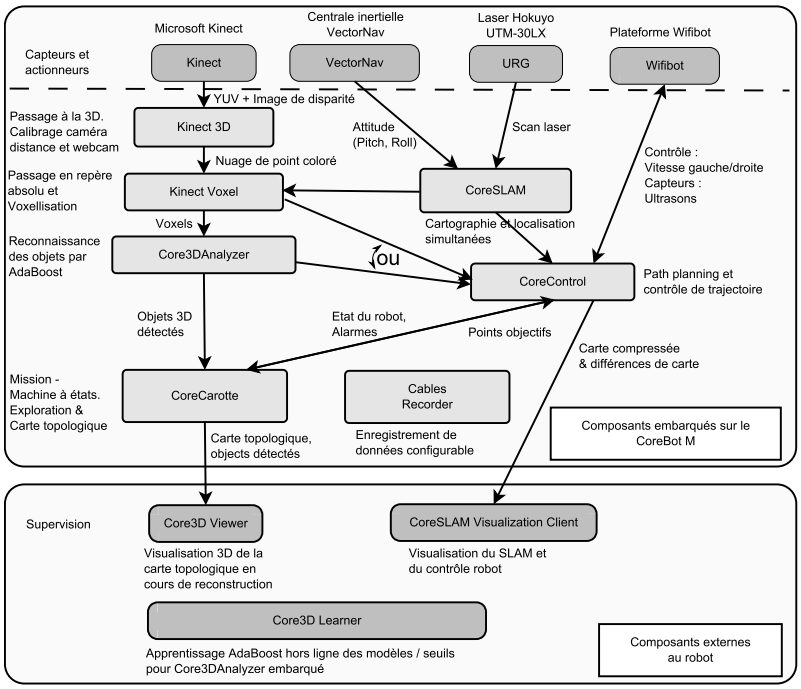


Figure 57. Architecture logicielle du robot *CoreBot M*.

Les capteurs / actionneurs comprennent :

- Le composant d'acquisition de données en provenance de la *Kinect*.
- Le composant d'acquisition de données en provenance du laser à balayage *Hokuyo*.
- Le composant d'acquisition de données en provenance de la centrale inertielle *VectorNav*, fournissant l'attitude du robot.
- Le composant de contrôle/commande de la plateforme robotique, qui collecte les données du capteur ultrason monté sur le robot, et qui assure le contrôle de la plateforme (vitesse gauche / droite, allumage/extinction des capteurs connectés à des relais commandables)

En vue de simuler et débugger le système il est nécessaire de pouvoir substituer aux composants capteurs et actionneurs des composants de simulation équivalents. Pour ce faire, nous utilisons le logiciel *Marilou* qui produit un environnement simulé très proche de la réalité (figure 58).

Figure 58. L'environnement Marilou.

Les composants de traitement comprennent :

- L'ensemble de la chaine de traitement des données issues de la *Kinect*, qui se décompose en 3 éléments :
    - *Kinect 3D* : composant de calcul générant un nuage de points 3D coloré à partir des données issues des capteurs de la *Kinect* (image couleur, image de disparité). Utilise les données de calibration propres à chaque *Kinect* (paramètres des caméras infrarouge et couleur)
    - *Kinect Voxel* : Composant assurant la voxellisation du nuage de point (regroupement des éléments du nuage de points par voxel[1]), après placement du nuage de point dans le repère absolu du robot. Ce composant utilise des paramètres statiques tels que la position du capteur sur le robot, mais aussi des données dynamiques fournies par le système de localisation (composant *CoreSLAM*) afin de placer le nuage au bon endroit dans la carte. Ces données peuvent être exploitées par la navigation (composant *CoreControl*)

1 Pour Volume Element, où élément unitaire de volume.

pour l'évitement des obstacles proches. Les données voxellisées sont également envoyées au composant de détection d'objets.

- *Core3DAnalyzer* : Il contient un algorithme de détection d'objets à base de classification AdaBoost. Il détecte également les niveaux (surélévation du sol) ainsi que les rampes, sur lesquels le robot peut naviguer : ces zones ne doivent pas être considérées comme des obstacles par la navigation. Ces données sont envoyées au composant *CoreControl* en vue de la navigation du robot. Les objets collectés sont fournis à la machine à états *CoreCarotte*.

- Le composant de localisation et cartographie simultanées (SLAM : Simultaneous Localization and Mapping) *CoreSLAM*, exploitant un algorithme original développé par le Centre de Robotique des Mines [83]. Il fournit en temps réel une carte ainsi que la position du robot *(x, y, θ)*, fournie au contrôleur du robot *CoreControl* ainsi qu'au voxelliseur des données de la *Kinect* (*Kinect Voxel*).

- Le composant de contrôle du robot, *CoreControl*, qui construit en temps réel une carte de distance aux obstacles en combinant la carte laser issue de *CoreSLAM*, les obstacles proches fournis par *Core3DAnalyzer* ou *Kinect Voxel* (voir paragraphe suivant sur la fiabilité), et les données issues du capteur ultrason monté sur le robot[1]. *CoreControl* calcule la trajectoire optimale[2] jusqu'au point objectif et calcule les commandes de robot associées. *CoreControl* permet de revenir sur ses traces en cas de problèmes (obstacle détecté au dernier moment, chemin bouché sans possibilité de se retourner).

- *CoreCarotte* est le composant contenant le code spécifique à l'application, le code de mission. Il intègre la machine à états du système, la stratégie d'exploration et la construction d'une carte topologique de l'environnement.

Les composants de supervision incluent *CablesRecorder*, un enregistreur de données de diagnostic, et des composants de visualisation : *Core3D Viewer* et *CoreSLAM Client*, qui doivent pouvoir être démarrés et arrêtés à distance, éventuellement à plusieurs exemplaires.

2.2 Fiabilité

La complexité du système induit des problèmes de fiabilité auxquels nous avons été confrontés assez rapidement.

L'utilisation de la *Kinect*, la caméra 3D de *Microsoft* – un produit tout nouveau au moment de la conception de *CoreBot M* – a nécessité l'intégration de *libfreenect*, une bibliothèque prenant en charge le protocole USB de la *Kinect* pour en acquérir les données. Or ce code open-source a connu pendant longtemps des problèmes de

1 Evitant ainsi de percuter un miroir ou une vitre, non détectées par les méthodes visuelles (laser ou caméra).

2 Au sens de l'heuristique A* utilisée dans l'algorithme.

fiabilité dont nous avons souffert directement. Au bout de quelques minutes, le composant l'intégrant s'arrêtait de manière irrévocable[1], nécessitant la déconnexion / reconnexion du capteur (possible logiciellement à travers un relais commandé) et le redémarrage du composant. L'objectif concernant la *Kinect* est de garantir la continuité de service du robot en cas d'arrêt de la *Kinect*, en tentant de la redémarrer, et en cas d'échec de redémarrage (déconnexion du capteur par exemple) d'être capable d'opérer le robot en mode dégradé (avec son laser principal seulement, en le faisant ressortir de l'arène).

Par ailleurs, le composant *Core3DAnalyzer*, intégrant l'analyse de la scène pour y détecter les objets, mais aussi les rampes et les niveaux sur lesquels le robot devait naviguer, est de haute complexité et n'a pour l'heure pas été complètement fiabilisé. En cas d'arrêt brutal de ce composant, le contrôle du robot doit pouvoir continuer, en s'appuyant sur la seule voxellisation des données de la *Kinect* : les informations issues du composant *Kinect Voxel*, plus simple et plus rudimentaire (pas de détection de niveau), sont suffisantes pour éviter les obstacles mais il ne peut plus monter les rampes puisqu'il les analyse également comme des obstacles.

2.3 Synchronisation

L'ensemble des capteurs du robot (Laser principal *Hokuyo*, Caméra 3D *Kinect*, Centrale inertielle *VectorNav*, capteurs à ultrason du *Wifibot)* fournissent des données complètement asynchrones, à des fréquences différentes (de 10Hz pour l'ultrason à 40Hz pour le laser). Ces données doivent être synchronisées car elles sont utilisées conjointement par les algorithmes. Par exemple, les données issues de la *Kinect* (détection d'obstacles proches et objets à reconnaître) doivent être correctement placés dans la carte qui est construite principalement sur la base des données du laser *Hokuyo*. Or, du fait de la vitesse de rotation assez importante du robot (il est capable de se retourner en 2 secondes), la synchronisation ne peut être ignorée.

Enfin, les différents composants n'utilisent pas les données brutes des capteurs, mais des données issues de chaines de traitement complexes. Ainsi, le composant *CoreControl* exploite la carte construite sur la base des données du laser *Hokuyo* passées à travers l'algorithme de SLAM, conjointement aux données d'obstacles proches issues de la *Kinect* et passées à travers 3 composants assez lourds (*Kinect 3D*, *Kinect Voxel* et *Core3DAnalyzer*). Tous ces algorithmes ont des temps d'exécution très variables en fonction du contexte (complexité de la scène), ce qui augmente encore la contrainte de synchronisation sur ces données.

La synchronisation, en l'absence d'environnement la prenant en charge, nécessite d'une part une datation des données à l'entrée du système, la propagation de cette datation à travers le système, et l'utilisation de files FIFO (First In, First Out) et d'un algorithme de synchronisation dédié à des points de synchronisation bien identifiés. Bien des systèmes font l'impasse sur cette contrainte de synchronisation, car cette notion est absente de la plupart des *framework*s souvent utilisés (Java, .NET, et même ROS [71], un *framework* robotique populaire).

1 A l'heure où ces lignes sont écrites, ce problème a été résolu.

3 La solution avec NJN

3.1 Architecture et distribution

L'architecture logicielle de *CoreBot M* se prête parfaitement à une implémentation sous NJN. Chaque unité logicielle est implémentée sous la forme d'un composant. Les composants sont répartis dans sept processus systèmes afin, d'une part, de profiter des deux cœurs de la cible matérielle et, d'autre part, d'isoler certains composants.

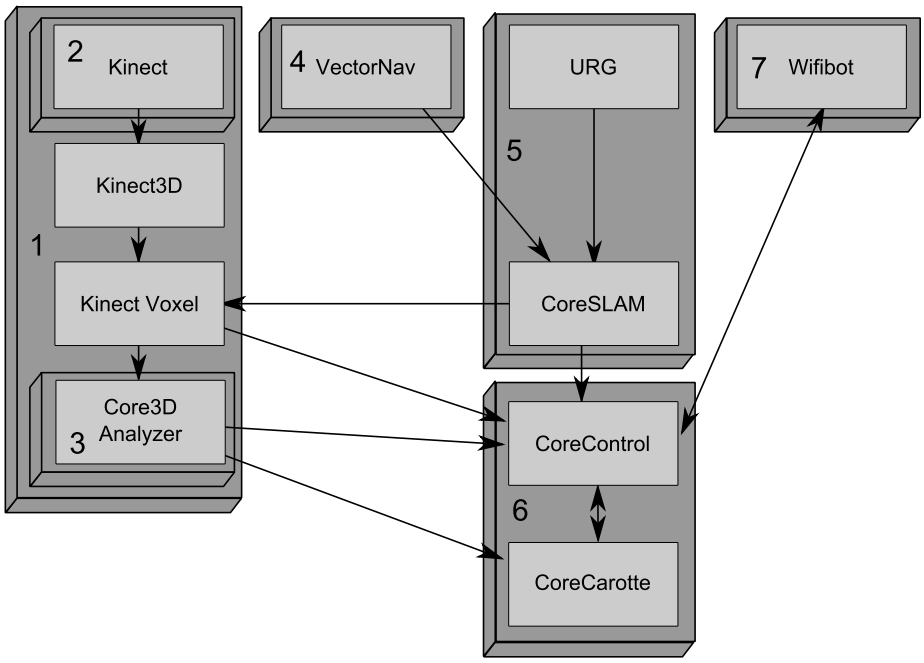


Figure 59. Répartition des composants de *CoreBot M* sur plusieurs processus systèmes.

Chaque composant de pilotage de capteur/actionneur est isolé dans son propre processus, à l'exception des composants liés au laser (URG et *CoreSLAM*). L'objectif est d'éviter la perte de données en provenance des capteurs. En donnant à ces processus une priorité maximale, on évite que des trames de données soient ignorées.

L'exécution de *CoreSLAM* est intimement liée aux données laser. Il ne peut donc fonctionner sans source en provenance du pilote du laser. Il est donc inutile de les séparer. Ils sont placés dans le même processus.

Selon cette même idée, les processus de traitement des données *Kinect* sont, à priori, à placer dans le même processus. Cependant, le pilote est séparé pour les questions de fiabilité évoqués dans la section 2.2. De même, le composant *Core3DAnalyzer* n'étant pas fiable, il est placé dans un processus qui lui est propre.

Les deux derniers composants, *CoreControl* et *CoreCarotte*, sont deux éléments de traitement algorithmique. Ils ne sont pas critiques et leur fiabilité a été éprouvée. Nous les avons donc placés dans un processus unique.

Les composants les plus gourmands en ressources sont *Core3DAnalyser* et *CoreSLAM*. Ils consomment à eux deux 70% des ressources CPU. On pourrait donc envisager de les déporter sur une ferme de processeurs. Avec NJN, ce type de manipulation est en effet très aisé et ne remet pas en cause le comportement temporel de l'application, du moment que la latence donnée au système est suffisante. L'intérêt majeur de déporter les composants gourmands est de pouvoir réduire la puissance de calcul embarquée et donc la consommation électrique.

3.2 Fiabilité

Les processus qui hébergent des composants critiques doivent être surveillés. C'est le cas par exemple du processus 3 qui héberge le composant *Core3DAnalyzer*. Une tâche supplémentaire se charge de ce travail. Elle fonctionne à la manière d'un chien de garde [88]. L'extrait de code ci-dessous en montre le principe.

Deux tâches sont instanciées dans cet exemple : la tâche *task* et la tâche *monitor*. Cette dernière a pour fonction de surveiller la première.

```
NJN_DEF_ALWAYS(monitor){
    switch(event){
    case NJN_TIMEOUT_EVENT:
        // The process is done! Restart it!
        break;
    }
    return NJN_SUCCESS;
}

NJN_DEF_COMPONENT(Monitoring){
    /*...*/
    njn_ref_t task    = njn_new(self, "rt_task", "task", /*...*/);
    njn_ref_t beat    = njn_new(self, "var", "beat",
        NJN_BIND_TO_BEAT(task));
    njn_new(self, "vt_task", "monitor",
        NJN_ALWAYS(monitor),
        NJN_WHEN(beat),
        NJN_TIMEOUT(1 SEC));
    return NJN_SUCCESS;
}
```

Lorsque le signal *beat* est à l'origine du réveil de la tâche *monitor*, aucune action n'est entreprise. En revanche un réveil par *timeout* permet d'identifier l'arrêt du processus et de le redémarrer.

Pour surveiller un processus, il suffit de lui confier l'exécution d'une tâche périodique et de placer le moniteur dans un autre processus. Lorsque le moniteur identifie un arrêt de l'exécution, il peut entreprendre de créer un nouveau processus pour y reconstituer les composants détruits. Le fonctionnement du système redevient alors normal.

Dans notre application, il suffit de surveiller la tâche hébergée par le composant *Core3DAnalyzer*. En outre, en cas de défaillance du processus hébergeur, il est possible d'extraire des données du composant *Kinect Voxel* pour assurer la continuité du service en attendant le rétablissement du *Core3DAnalyzer*. Les données extraites sont moins riches mais parfaitement suffisantes pour les opérations courantes du robot. Elles ne permettent cependant pas de différencier les rampes des simples obstacles.

3.3 Synchronisation

Le problème de la synchronisation des données est réglé de façon totalement transparente par le *framework* NJN. Il suffit d'associer à chaque tâche la liste des signaux sur lesquels elle doit être synchronisée (figure 60).

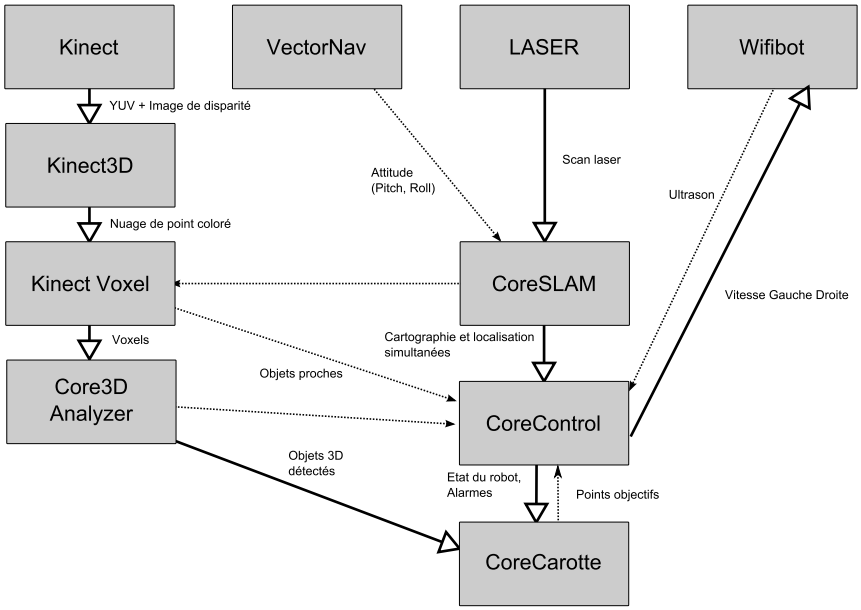


Figure 60. Synchronisation des différents composants de CoreBot M.

Les données lues sur les autres flux correspondent systématiquement aux dernières informations disponibles à l'instant de synchronisation. Par exemple, la tâche associée au composant *CoreSLAM* est synchronisée sur les données du laser. Elle consomme également les données en provenance de la centrale inertielle à travers un appel à la méthode *njn_read()*. La donnée récupérée correspond naturellement à la dernière trame de la centrale inertielle précédant la donnée laser à l'origine du traitement.

Un problème subsiste cependant : les données récupérées depuis les différents capteurs sont datées lorsque la trame correspondante est reçue par le driver logiciel correspondant et non à la date physique de la mesure. Or, à chaque capteur correspond une latence de mise en forme et de transmission des données

très différente. Il faut donc, au moment de leur introduction dans NJN tenir compte de ce délai. Deux solutions existent.

La première consiste à synchroniser les données sur celles du capteur le plus lent en ajoutant un délai différentiel aux autres données. Il suffit de réaliser les écritures avec un paramètre *NJN_AFTER()* de valeur adéquate. Par exemple, pour retarder les données laser, il suffit d'écrire :

```
data = njn_read(laser_in);
njn_write(laser_out, data, NJN_AFTER(25 MSEC));
```

Toutes les données se trouvent datées avec retard, ce qui peut être inconfortable.

La seconde solution consiste à utiliser des boites de découplage pour redater les données issues des capteurs. La méthode *njn_write_at()* permet, dans le cadre d'une *DecouplingBox* uniquement, d'attribuer une date arbitraire aux évènements. Dans le cas du laser qui introduit un retard d'environ 30 ms, il suffit alors d'écrire :

```
data = njn_read(laser_in);
njn_write_at(laser_out, data, njn_time_sub(njn_now(), 30 MSEC));
```

pour redater les données à leur instant réel de mesure.

Ne reste plus qu'à fixer la latence globale du système en attribuant à la tâche chargée du contrôle moteur une valeur suffisante pour effectuer l'ensemble des traitements et des échanges de données. Les performances de vitesse du robot sont directement liées à ce paramètre de latence. Typiquement, le contrôle est effectué à 20Hz (cadence de traitement des données laser) avec une latence de 100 ms, NJN se chargeant naturellement de pipeliner les traitements. Avec cette latence, le fonctionnement du robot est assuré pour une vitesse linéaire de 1 m/s environ et une vitesse de rotation de l'ordre de 180°/s.

Notons enfin que si l'on souhaite déporter l'algorithme de SLAM sur une ferme de calcul, la conception de l'application n'est pas affectée mais la latence globale doit être augmentée. Cela oblige à ralentir le robot pour préserver sa stabilité.

*
* *

Même si elle est empruntée au domaine de la robotique, cette étude de cas d'un système réel pose les problèmes de complexité des systèmes actuels de façon très réaliste. Les algorithmes mis en œuvre sont nombreux, très différents, complexes, de temps d'exécution fortement variable et pas toujours d'une fiabilité totale. Le nombre de capteurs est important et ils délivrent leurs données à des fréquences très différentes. La puissance de calcul nécessaire est très élevée et les contraintes temps-réel sont relativement sévères sans être critiques.

La solution proposée est générique : seul le composant *CoreCarotte* est spécifique au défi. Les autres composants sont réutilisables dans d'autres applications. Les aspects synchronisation et distribution sont totalement gérés par NJN. Seules la latence globale du système et la vitesse du robot sont à régler.

Lors du développement du projet *CoreBots*, NJN n'était pas totalement fiabilisé. L'implémentation réelle sur le robot *CoreBot M* s'appuie donc uniquement sur les primitives de bas niveau du projet AROS : DOHC, CHASSIS et CABLES. Cependant, la conception des différents modules du système intègrent, en dur, les mécanismes du moteur d'NJN pour assurer la synchronisation notamment. Les principes de synchronisation, de monitoring et de distribution exposés ci-dessus ont été appliqués à tous les niveaux de la conception. Ils ont largement contribué à la réussite du projet et ont permis au robot de gagner le défi CAROTTE deux années consécutives.

TROISIÈME PARTIE
L'IMPLÉMENTATION

CHAPITRE 7
ABSTRACTION DE LA MÉMOIRE, DE LA PLATEFORME ET DE LA COMMUNICATION – LES BIBLIOTHÈQUES SUPPORTS

L'implémentation d'NJN repose sur trois bibliothèques : DOHC, qui assure la gestion de la mémoire et qui procure un ensemble de structures de données génériques ; CHASSIS, qui encapsule les services du système d'exploitation de manière à assurer la portabilité des applications ; et enfin CABLES, qui se charge de gérer les communications sous la forme d'appel de services distants asynchrone. Ce chapitre décrit brièvement les points essentiels de ces trois bibliothèques.

1 Le double arbre à cames en tête

La librairie DOHC procure à NJN des capacités de typage dynamique des données proches de celles des langages de scripts tels que *Ruby* ou *Pyton*. Le type des objets DOHC n'est pas fixé à la compilation et peut changer au cours de l'exécution. Il peut s'agir de données scalaires entières ou flottantes, mais aussi d'objets plus complexes tels que des tableaux dynamiques, des tables de hachages, etc. En associant DOHC à la librairie CLUTCH, on peut même manipuler des classes et des objets au sens de la POO.

Par ailleurs, la mémoire occupée par les objets DOHC est gérée par un ramasse-miette. Ce dernier peut travailler à espace mémoire limité et fixé, si bien que l'on peut contrôler efficacement les besoins mémoire des applications NJN, en

particulier sur les cibles matérielles disposant de peu de mémoire. Ce ramasse-miette comporte cependant quelques inconvénients. Le compromis adopté, celui de la rapidité, conduit dans certaines configurations à confier à l'utilisateur la libération mémoire de certains objets. Nous aborderons ce point plus en détail.

L'objet de cette partie est de donner un aperçu des différents types d'objets DOHC et de montrer les possibilités offertes par la bibliothèque dans son ensemble. La première section est assez générale. Elle traite de l'organisation et des principes généraux de DOHC. Dans la section suivante nous verrons comment DOHC permet de construire des structures et des objets avec la librairie CLUTCH. Nous verrons ensuite les mécanismes de sérialisation qui facilitent les échanges de données à travers fichiers et réseaux. Enfin nous aborderons la gestion mémoire proprement dite.

1.1 Les objets DOHC de base et les conteneurs

Nous abordons dans cette section la structure générale de la librairie DOHC. On tente de dégager des généralités sur son utilisation, de placer quelques repères sur les différentes familles d'objets et de fonctions qui la composent, etc.

Les objets DOHC sont manipulés à travers des références génériques (pointeurs). DOHC se charge d'allouer dynamiquement l'espace mémoire dont ils ont besoin et de gérer leurs types au cours de l'exécution. Du point de vue de l'utilisateur, les objets DOHC sont tous de type *dohc_objet_t*[1]. C'est ce qui fait leur intérêt : ce type unique permet d'écrire des fonctions polymorphes ce qui n'est, en principe, pas supporté par le langage C.

On distingue trois familles d'objets :

- La première famille est celle des petits objets. Ils occupent suffisamment peu de place en mémoire pour tenir dans une variable de type *dohc_objet_t*. Ils sont donc gérés comme n'importe quelle variable du langage C, c'est-à-dire de façon automatique ou statique selon la nature et le contexte de leur définition. On trouvera dans cette famille le singleton *nil*, les entiers, les flottants, les pointeurs, les constantes symboliques et les erreurs.
- On trouve ensuite les conteneurs qui se subdivisent en deux sous-familles : les conteneurs indexés dont la taille peut varier au cours de l'exécution (tableaux, listes, tables de hachage et chaines de caractères), et les tas dont la taille est définie à la création des objets (tas bruts et structures). Ces objets sont alloués dynamiquement.
- La troisième famille est un peu comme le troisième groupe des verbes de la langue française. On y trouve des classes d'objets inclassables, utilisés dans des contextes toujours particuliers : les *states*, les itérateurs, les symboles, les sérialisation, les *callback* de ramasse-miette, etc.

1 *njn_ref_t* lorsqu'on se trouve dans une application NJN, mais c'est rigoureusement la même chose.

Les méthodes qui manipulent ces objets sont pour la plupart polymorphes. Leur comportement n'est évidemment pas tout-à-fait le même d'un objet à l'autre mais il a toujours un sens vis-à-vis de l'objet manipulé.

Les types conteneurs indexés de DOHC sont au nombre de 4 : les tableaux, les listes, les tables de hachage et les chaines de caractères. Ce dernier conteneur est un peu particulier dans la mesure où il ne contient pas d'objet DOHC. Les méthodes de manipulation des conteneurs sont assez homogènes. Nous en présentons ici les grandes familles.

L'état d'un conteneur peut être sondé par les fonctions *dohc_size()* et *dohc_is_empty()*. La première fournit le nombre d'éléments présents dans le conteneur et la seconde signale si le conteneur est vide ou non.

Il y a grosso modo quatre grandes familles de manipulation : élément par élément, par tranche (*slice*), par l'intermédiaire d'itérateurs ou par des fonctions de manipulation spécifiques[1].

On dispose d'une collection de méthodes en *_at()* pour manipuler un conteneur élément par élément. Elles ont toutes en commun de travailler à partir d'indices : *dohc_insert_at()*, *dohc_delete_at()*, *dohc_get_at()*, *dohc_set_at()*, etc. Notons au passage que le premier élément d'un conteneur porte toujours l'indice 0, sauf lorsqu'il s'agit d'une table de hachage.

On dispose aussi de méthodes spécialisées dans les extrémités. D'abord devant : *dohc_get_first()*, *dohc_shift()*, *dohc_unshift()* ; puis derrière : *dohc_get_last()*, *dohc_push()*, *dohc_pop()*, etc. Ces méthodes sont particulièrement utiles lorsqu'on souhaite implémenter des piles, files et autres *heap*. A l'exception de *dohc_pop()* et *dohc_shift()*, elles ne sont pas supportées par le type *HTable*.

Les méthodes de manipulation par tranche (*slice*) s'apparentent aux méthodes en *_at()* précédentes à ceci près qu'elles nécessitent souvent deux indices[2] : *dohc_get_slice()*, *dohc_set_slice()*, *dohc_insert_slice()*, *dohc_delete_slice()*, etc. Ces méthodes ne sont pas supportées par les tables de hachage non plus.

Les méthodes en *_at()* ou en *_slice()* précédentes acceptent en guise d'indices des itérateurs.

```
int main(void){
    dohc_object_t state = dohc_init(0, 0, 0, 0, 0, 0, 0);
    dohc_object_t str   = dohc_string_new(state,
                "DOHC library (C) 2009 AROS Project Consortium\n");
    dohc_object_t iter  = dohc_begin(str);
    while(!DOHC_IS_END(iter)){
        putchar((int)DOHC_TO_INT(dohc_get_at(str, iter)));
        dohc_next(iter);
    }
    return 0;
}
```

Mais ces derniers procurent d'autres fonctionnalités plus intéressantes.

1 Le fameux troisième groupe ici perdu en quatrième position !!

2 Sauf la méthode *dohc_insert_slice()* qui n'a besoin que de la position d'insertion.

Par exemple, la fonction *dohc_find()* renvoie un itérateur sur la première occurrence trouvée d'un objet donné. La variante *dohc_rfind()* fait la même opération en commençant par la fin du conteneur. Les méthodes *dohc_detect()* et *dohc_rdetect()* font de même en utilisant une fonction de recherche *ad hoc*.

Enfin des fonctions spécifiques permettent de trier le conteneur (*dohc_sort()*), de le fusionner avec un second (*dohc_append()*), d'en supprimer le contenu (*dohc_clear()*) ou de le renverser (*dohc_reverse()*), d'en faire une copie (*dohc_copy()*), etc.

1.2 L'embrayage (la couche objet)

La bibliothèque CLUTCH s'appuie sur les structures DOHC pour fournir une couche objet. Les concepts de classe, d'objet, d'héritage simple, de polymorphisme, de fonction virtuelle, d'interface (au sens du langage *Java*), etc. sont pris en charge par la bibliothèque.

Dans CLUTCH, une classe est un objet statique qui comporte un lien vers la classe d'objets *Klass*[1], un lien vers la classe parente, une liste d'implémentations d'interfaces et une structure d'information à destination de DOHC.

```
typedef struct _clutch_klass_t{
    dohc_object_t           klass;
    dohc_object_t           parent;
    dohc_object_t           interfaces;
    dohc_struct_type_t      dohc_info;
}clutch_klass_t;
```

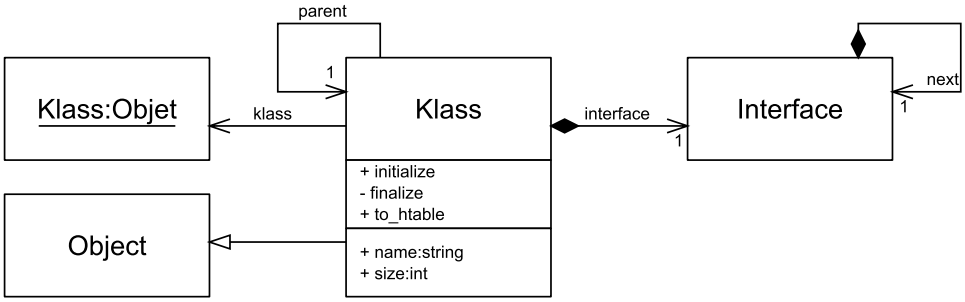


Figure 61. La structure d'un objet Klass dans CLUTCH.

La structure d'information DOHC comprend le nom de la classe, la taille des objets correspondants en octets, la fonction d'initialisation par défaut à partir d'une table de hachage, la fonction de finalisation de l'objet et une fonction de conversion en table de hachage. Initialisation et fonction de conversion en table de hachage sont complémentaires et sont indispensables pour que l'objet puisse être sérialisé.

```
typedef struct _dohc_struct_type_t{
    const char     *name;
    int             size;
```

1 L'identificateur *Klass* a été choisi pour éviter tout conflit avec le mot clé *class* du C++.

```
    dohc_object_t (*initialize)(dohc_object_t self, dohc_object_t h);
    void          (*finalize)  (dohc_object_t self);
    dohc_object_t (*to_htable) (dohc_object_t self);
} dohc_struct_type_t;
```

CLUTCH définit un objet de base duquel héritent tous les autres (y compris la classe *klass*). Sa structure est la suivante :

```
typedef struct _clutch_object_t{
    dohc_object_t klass;
}clutch_object_t;
```

Il comporte uniquement un lien vers la classe de l'objet.

A partir de cette classe de base, on peut en définir de nouvelles en fournissant un objet statique de type *Klass* et un type structuré fixant les différents champs de l'objet correspondant.

La classe *MyObject* est définie par :

```
static const base_klass_t MyObject_klass = {
    DOHC_PTR(&clutch_klass_klass),
    DOHC_PTR(&clutch_object_klass),
    DOHC_NIL, {
        "MyObject",
        sizeof(MyObject_t),
        NULL,
        MyObject_finalize,
        NULL
    },
};

static const dohc_object_t MyObject = DOHC_PTR(&MyObject_klass);
```

et la structure de l'objet correspondant ne comporte qu'un champ nommé *val* :

```
typedef struct _MyObject_t{
    clutch_object_t super;
    dohc_object_t   val;
}MyObject_t;
```

Pour pouvoir être gérée par CLUTCH, la classe doit comporter un constructeur, un initialiseur et un finaliseur.

Le constructeur demande l'allocation de l'objet à DOHC puis appel l'initialiseur de la classe.

```
dohc_object_t
MyObject_new(
    dohc_object_t state,
    dohc_object_t val
){
    dohc_object_t self;
    self = dohc_struct_new(state, &MyObject_klass.dohc_info,
        dohc_nil);
    MyObject_initialize(self, val);
    return self;
}
```

L'initialiseur appelle son homologue de la classe de base (ici la classe des objets de base de CLUTCH) puis initialise les champs de l'objet. Le compteur de référence des champs est incrémenté (voir section 1.4).

```
void
MyObject_initialize(
    dohc_object_t self,
    dohc_object_t val
){
    MyObject_t *pself;
    pself = DOHC_TO_STRUCT(self, MyObject_t);
    clutch_initialize(self, MyObject);
    pself->val = dohc_incref(val);
}
```

Le finaliseur sera appelé par le ramasse-miette de DOHC. Il décrémente les références des champs de l'objet afin qu'ils puissent à leur tour être collectés.

```
void
MyObject_finalize(
    dohc_object_t self
){
    MyObject_t *pself;
    pself = DOHC_TO_STRUCT(self, MyObject_t);
    dohc_decref(pself->val);
    return;
}
```

A partir des éléments décrits ci-dessus, CLUTCH est capable de construire et de manipuler des objets de cette nouvelle classe. Un appel à *MyObject_new()* construira un nouvel objet :

```
dohc_object_t obj = MyObject_new(dohc_state, dohc_int(10));
```

Comme nous l'avons dit, CLUTCH implémente l'ensemble des concepts de la programmation orientée objet. Pour pouvoir utiliser toutes les fonctionnalités de CLUTCH il faudrait y consacrer de nombreuses pages sans que celles-ci soient d'un intérêt fondamental pour cet exposé. Nous préférons pour cette raison nous en tenir à cette brève introduction.

1.3 La sérialisation

A l'exception des pointeurs et de *GCCallback*, tous les objets DOHC sont sérialisables. Les opérations de sérialisation et de désérialisation sont très simples à mettre en œuvre.

La sérialisation est assurée par la méthode *dohc_dump()*. Elle renvoie un objet DOHC de type *Marshal*. Il s'apparente à un *buffer* dont on peut récupérer l'adresse par la macro *DOHC_PTR()* et la taille par la macro *DOHC_SIZE()*.

Pour désérialiser, il suffit d'appeler la méthode *dohc_load()*. Elle retourne l'objet initial.

Entre les deux opérations, l'objet *Marshal* peut être envoyé à travers le réseau ou enregistré dans un fichier binaire, comme dans l'exemple ci-dessous.

```
int main(void){
    dohc_object_t state = dohc_init(0, 0, 0, 0, 0, 0, 0);
    dohc_object_t sstr  = dohc_string_new(state,
          "DOHC library (C) 2009 AROS Project Consortium");
    dohc_object_t rstr, dmarsh, lmarsh;
    int           size;
    FILE         *log;


    log = fopen("log", "w");
    dmarsh = dohc_dump(state, sstr, 0);
    size = DOHC_SIZE(dmarsh);
    fwrite(&size, sizeof(size), 1, log);
    fwrite(DOHC_TO_PTR(dmarsh, void*), size, 1, log);
    fclose(log);

    /*...*/

    log = fopen("log", "r");
    fread(&size, sizeof(size), 1, log);
    lmarsh = dohc_marshal_new(state, NULL, size, 0);
    fread(DOHC_TO_PTR(lmarsh, void*), size, 1, log);
    rstr = dohc_load(state, lmarsh);
    DOHC_ASSERT(dohc_is_equal(sstr, rstr));
    DOHC_ASSERT(!dohc_is_same(sstr, rstr));
    fclose(log);
    return 0;
}
```

Il est possible de sérialiser des structures de donneés complexes où les conteneurs se référencent les uns les autres. Lors de la désérialisation, on retrouve les structures d'origine parfaitement intactes et fonctionnelles.

Lorsque les objets ne sont pas sérialisables (par exemple un objets CLUTCH dépourvus de fonction de conversion en *HTable*), DOHC construit une référence appelée (*Handle*) qui, à défaut de permettre de transmettre l'objet à travers le réseau, assure de pouvoir le retrouver si la dite référence est désérialisée dans l'espace mémoire DOHC où est réellement situé l'objet. Cette possibilité permettra d'implémenter les communications entre objets distants dans NJN.

1.4 La gestion de la mémoire

DOHC alloue les objets dans une zone mémoire spécifique gérée par lui, que l'on initialise grâce à la fonction *dohc_init()*. En fournissant à cette fonction l'adresse de début et la taille d'une zone mémoire réservée, on place DOHC dans un mode où il se restreint à la zone mémoire fournie.

Il n'y a aucune variable globale dans la librairie qui pointe sur cette zone spécifique. Le seul moyen pour DOHC de retrouver la mémoire qu'il gère est de disposer d'un objet alloué à l'intérieur. C'est pour cette raison que le premier argument des constructeurs d'objets dynamiques est systématiquement un objet dynamique quelconque.

Quoi qu'il en soit, il faut bien amorcer la pompe !! C'est le rôle tenu par l'objet *state* retourné par la fonction *dohc_init()*.

A partir de ce noyau originel, on peut créer les premiers objets de l'application.

```
int main(void){
    dohc_object_t state = dohc_init(...);
    dohc_object_t ary   = dohc_array_new(state);

    dohc_set_at(ary, dohc_int(0), dohc_string_new(ary, "0"));
    dohc_set_at(ary, dohc_int(1), dohc_string_new(ary, "1"));

    DOHC_ASSERT(dohc_is_equal(dohc_get_at(ary, dohc_int(0)),
                              dohc_string_new(ary, "0")));
    DOHC_ASSERT(dohc_is_equal(dohc_get_at(ary, dohc_int(1)),
                              dohc_string_new(ary, "1")));

    dohc_done(state);
}
```

L'appel périodique de *dohc_gc_run()* assure la gestion incrémentale de la mémoire. DOHC essaie de compacter les objets pour éviter la fragmentation en déplaçant au besoin les objets. Le déplacement des objets est totalement transparent pour le développeur. La gestion de l'espace mémoire doit être effectuée dans les couches les plus basses de l'application sans quoi les objets de la pile risquent d'avoir des comportements fort déplaisants[1].

Notons que dans NJN, tous ces aspects sont gérés par le moteur de l'application. Le programmeur n'a donc jamais à s'en préoccuper.

Le gestionnaire de mémoire fonctionne sur le principe du comptage de référence : A chaque objet est associé un compteur qui maintient à jour le nombre de références pointant sur l'objet. Si la référence est copiée, le compteur est incrémenté. Si la référence est détruite ou est mise à jour, le compteur est décrémenté. Lorsque le compteur atteint 0, c'est-à-dire lorsque aucune référence ne permet d'accéder à l'objet, ce dernier peut être désalloué [19].

C'est un algorithme de collecte simple à mettre en œuvre, rapide à exécuter et dont le traitement est finement incrémental. La mémoire peut donc être gérée par collecte élémentaire, même lorsque les plages de repos du système sont courtes.

Mais cette simplicité se paye par un défaut majeur : le principe de comptage de références ne permet pas de collecter les objets qui se référencent eux-mêmes, soit directement, soit par l'intermédiaire d'autres objets. Les références cycles doivent donc être ouvertes manuellement pour que les objets concernés soient désalloués.

Dans la majorité des cas, la mise à jour du compteur de référence est intégralement gérée par DOHC. Lorsqu'on ajoute un objet à un conteneur indexé, ce dernier incrémente automatiquement le compteur de référence de l'objet ajouté, et inversement lorsqu'il en est supprimé.

Dans les rares cas où cela est nécessaire, comme pour la construction et la destruction des objets CLUTCH, la fonction *dohc_incref()* (resp. *dohc_decref()*)

1 Le compteur de références des objets automatiques définis dans les niveaux inférieurs de la pile doit être incrémenté pour que le ramasse-miette ne soit pas tenté de collecter les objets correspondants.

incrémente (resp. décrémente) le compteur de référence de l'objet fourni en argument.

Lorsque l'application se termine, la fonction *dohc_done()* fait le ménage et désalloue tous les objets encore présents en mémoire. Si l'espace mémoire fourni à DOHC a été créé par *malloc()*, il convient de le libérer avec *free()*, après l'appel de *dohc_done()*.

Notre tour d'horizon de la librairie DOHC est à présent terminé. Nous avons tenté de ranger un peu les choses pour présenter au lecteur une vision structurée de la bibliothèque.

2 Le châssis

La bibliothèque CHASSIS a pour fonction de fournir une implémentation abstraite des services de base du système d'exploitation. L'objectif est de rendre NJN indépendant du système. Toutes les fonctionnalités utiles du système sont encapsulées dans des fonctions CHASSIS qu'il faut bien évidemment ré-implémenter pour chaque nouvelle plateforme. De base, une implémentation est fournie pour les systèmes *Unix-Like* (Linux, Mac OS X, etc.) et *Windows* (32 et 64 bits). Il est tout-à-fait possible d'en construire une supplémentaire si l'on souhaite exécuter une application NJN sur un autre système.

2.1 La gestionnaire des évènements

Une grande part des fonctionnalités de la bibliothèque CHASSIS tourne autour de l'implémentation de la fonction d'attente évènementielle *chassis_wait()* (équivalent du *select* dans le monde *Unix*).

Lorsqu'elle est appelée, la fonction place le processus qui l'appelle en attente d'un évènement. Les évènements sont de deux types : il peut s'agir de la fin d'un délai d'attente ou d'un évènement d'entrée/sortie. Les premiers sont appelés des *timers* tandis que les seconds sont appelés des *listener*.

En attendant qu'un *timer* ou un *listener* soit activé, la priorité du processus est abaissée au minimum et le ramasse-miette de DOHC est appelé en boucle jusqu'à ce qu'il ne puisse plus optimiser la mémoire. Lorsque cette situation se produit, CHASSIS fait appel au système d'exploitation pour placer le processus en sommeil.

Chacun des évènements potentiellement attendus est associé à une fonction *callback* définie par l'utilisateur. Lorsqu'un évènement survient, la fonction *callback* qui lui est associée est appelée pour traiter l'évènement. La fonction *chassis_wait()* reprend ensuite la main pour un nouveau cycle d'attente, à moins que la fonction *callback* ait explicitement demandé le contraire. Dans ce cas, l'exécution se poursuit à l'instruction qui suit immédiatement l'appel à *chassis_wait()*.

2.2 La gestion du temps

Les *timer* permettent de mettre en place des attentes temporelles ou des actions périodiques. Voici un petit exemple où un *timer* provoque un affichage toutes les secondes.

```
int main()
{
    dohc_object_t dohc_state;
    dohc_object_t chassis_state;
    dohc_object_t retval;

    dohc_state = dohc_init(NULL, 0, 0, 0, NULL, NULL);
    chassis_state = chassis_init(dohc_state);

    chassis_add_timer(chassis_state,
        chassis_ms(1000), timer_cb, dohc_nil);

    retval = chassis_wait(chassis_state);
    dohc_printf(dohc_state,
        "Finished with return value: %o!\n", retval);

    chassis_done(chassis_state);
    dohc_done(dohc_state);
    return 0;
}
```

Un *timer* est programmé pour une attente d'une seconde. La fonction *callback* associée s'appelle *timer_cb*. Le programme appelle ensuite *chassis_wait()* en attente d'un évènement.

Voici la définition de la fonction *callback* associée au *timer* :

```
dohc_object_t
timer_cb(
    dohc_object_t timer,
    dohc_object_t userdata
){
    dohc_object_t chassis_state = chassis_get_state(timer);
    printf("I'm living!\n");
    chassis_add_timer(chassis_state,
        chassis_ms(1000), timer_cb, dohc_nil);
    return dohc_nil;
}
```

Elle se contente d'afficher un message sur la console et de reprogrammer un nouveau *timer* d'une seconde associé à la même fonction *callback*. On repart donc pour un cycle d'attente, etc.

2.3 La gestion des entrées/sorties

Les *listener* ont pour fonction de surveiller l'apparition d'évènements sur les flux d'entrée/sortie tels que fichiers, sockets réseau, liaisons séries, etc.

Voici un exemple de serveur TCP :

```
int main(){//server
    dohc_object_t dohc_state;
    dohc_object_t chassis_state;
    dohc_object_t server;
    dohc_object_t retval;

    dohc_state = dohc_init(NULL, 0, 0, 0, NULL, NULL);
    chassis_state = chassis_init(dohc_state);

    printf("Start server!\n");
    server = dohc_incref(
        chassis_init_tcp_server(chassis_state, 2000,
        connect_cb, dohc_nil));
    retval = chassis_wait(chassis_state);

    dohc_printf(chassis_state,
        "Finished with return value %o!\n", retval);
    chassis_close_tcp_server(chassis_state, dohc_decref(server));
    chassis_done(chassis_state);
    dohc_done(dohc_state);
    return 0;
}
```

Après l'initialisation de DOHC et CHASSIS, un serveur est déclaré sur le port 2000. Toute connexion d'un client se traduira par l'appel de la fonction *callback connect_cb*.

Celle-ci crée un *listener* pour le client qui a demandé la connexion. Les évènements de réception provoqueront l'appel de la fonction *callback read_cb*.

```
dohc_object_t
connect_cb(
    dohc_object_t chassis_state,
    dohc_object_t client,
    struct sockaddr_in6 *addr,
    int len,
    dohc_object_t userdata
){
    chassis_add_listener(chassis_state, client,
       CHASSIS_READ, read_cb, client);
    return dohc_nil;
}
```

La fonction *callback read_cb* fait appel à *chassis_read()* pour récupérer le message reçu du client et l'affiche sur la console.

```
dohc_object_t
read_cb(
    dohc_object_t listener,
    dohc_object_t client
){
    char buffer[256];
    dohc_object_t n;
    dohc_object_t chassis_state = chassis_get_state(listener);
    n = chassis_read(client, buffer, 255);
    dohc_printf(chassis_state, "Read %o bytes from client", n);
    if (DOHC_TO_INT(n) > 0) {
        buffer[DOHC_TO_INT(n)] = 0; // EOS
        printf(": %s\n", buffer);
```

```
        } else {
            printf("\n");
            chassis_remove_listener(chassis_state, client);
        }
        return dohc_nil;
    }
```

Côté client, un *timer* provoque l'exécution de la fonction *timer_cb*.

```
int main(){ //client
    dohc_object_t dohc_state;
    dohc_object_t chassis_state;
    dohc_object_t retval;

    dohc_state = dohc_init(NULL, 0, 0, 0, NULL, NULL);
    chassis_state = chassis_init(dohc_state);

    chassis_add_timer(chassis_state,
        chassis_ms(1000), timer_cb, dohc_nil);

    retval = chassis_wait(chassis_state);
    dohc_printf(chassis_state,
        "Finished with return value %o!\n", retval);
    chassis_done(chassis_state);
    dohc_done(dohc_state);
    return 0;
}
```

Au premier déclenchement du *timer*, la fonction *callback* initie une connexion avec le serveur associé au port 2000. Lors des déclenchements suivants, elle fait appel à la fonction *chassis_write()* pour envoyer un message au serveur.

```
dohc_object_t
timer_cb(
    dohc_object_t timer,
    dohc_object_t server
){
    dohc_object_t n;
    dohc_object_t chassis_state = chassis_get_state(timer);
    if (dohc_is_nil(server)){
        server = chassis_connect_tcp_server(
            chassis_state, "localhost", 2000);
        printf("Connected as a client!\n");
    }else{
        n = chassis_write(server, "data", 4);
        dohc_printf(chassis_state,
            "Written %o bytes to server\n", n);
    }
    chassis_add_timer(chassis_state,
        chassis_ms(1000), timer_cb, server);
    return dohc_nil;
}
```

Après chaque déclenchement, le *timer* est reprogrammé pour un nouveau cycle.

De façon similaire, CHASSIS permet de construire des *listeners* pour les *pipes*, les liaisons séries ou tout autre entrée/sortie de type caractères.

Nous venons de voir les fonctionnalités les plus importantes de CHASSIS. Mais cette bibliothèque encapsule bien d'autres services du système d'exploitation. Il permet par exemple de gérer la création et la destruction de processus systèmes. Il assure le chargement et le déchargement des bibliothèques dynamiques (*dll* sous *Windows* ou *shared object* sous *Unix*). Il permet de créer des zones de mémoire partagée. Etc.

3 Les câbles

CABLES est la bibliothèque qui gère les communications dans NJN. Le modèle abstrait adopté repose sur des appels de service distant asynchrones. Lorsqu'un client fait appel à un service, il fournit la fonction *callback* qui sera chargée de traiter la réponse. L'implémentation s'appuie sur CHASSIS pour établir la communication et sur DOHC pour assurer la sérialisation des données échangées.

3.1 Déclarer un service et démarrer le serveur

Avec *CABLES*, tout processus joue indifféremment le rôle de client et de serveur. Chaque processus est en effet associé à un port TCP.

Lors de sa déclaration, un service porte un nom et est associé à une fonction qui en implémente la fonctionnalité.

```
int main(){  //server
    dohc_object_t dohc_state = dohc_init(NULL, 0, 0, 0, NULL, NULL);
    dohc_object_t chassis_state = chassis_init(dohc_state);
    dohc_object_t cables_state  = cables_init(chassis_state,
        "test_server", 2000);
    cables_declare_service(cables_state, "hello", hello, dohc_nil);
    cables_serve(cables_state);
    return 0;
}
```

Dans l'exemple ci-dessous, le service *hello* retourne à l'appelant une chaine de caractères composée à partir de l'argument qui lui est transmis par l'appelant.

```
dohc_object_t
hello(
    dohc_object_t cables_state,
    dohc_object_t param,
    dohc_object_t context
){
    return dohc_sprintf(cables_state, "Hello %o!", param);
}
```

La fonction *cables_serve()* lance ensuite le serveur. Cette fonction se contente de faire un appel à *chassis_wait()* vue précédemment.

3.2 Établir la connexion et faire un appel de service

Côté client, la première chose à faire est d'établir la connexion avec le serveur. C'est le rôle de la fonction *cables_get_host()*.

```
int main(){ //client
    dohc_object_t dohc_state = dohc_init(NULL, 0, 0, 0, NULL, NULL);
    dohc_object_t chassis_state = chassis_init(dohc_state);
    dohc_object_t cables_state  = cables_init(chassis_state,
        "test_client", 2001);
    dohc_object_t connection = cables_get_host(cables_state,
        "localhost:2000");
```

Une fois le serveur identifié, un appel à *cables_get_service()* permet de récupérer une référence sur le service souhaité.

```
    dohc_object_t serv = cables_get_service(connection, "hello");
```

Il ne reste plus qu'à appeler le service et à se placer en attente de la réponse :

```
    cables_call_service(serv,
        dohc_string_new(dohc_state, "client"),
        hello_cb, dohc_nil);
    cables_serve(cables_state);
    return 0;
}
```

L'appel du service est associé à une fonction *callback* dont le rôle est de traiter la réponse du serveur. Ce traitement a donc lieu en différé. Dans notre exemple, la fonction *callback* se contente d'afficher la réponse retournée par le service.

```
dohc_object_t
hello_cb(
    dohc_object_t cables_state,
    dohc_object_t rtnval,
    dohc_object_t context
){
    dohc_printf(cables_state, "%o\n", rtnval);
    return dohc_nil;
}
```

La mise en œuvre de CABLES est, comme on a pu le constater plus haut, extrêmement simple. Des fonctionnalités complémentaires sont disponibles dans la bibliothèque tels que les services *push* qui permettent à un serveur de diffuser périodiquement une information à plusieurs clients inscrits à un service, ou la prise en charge de *zeroconf* qui permet de repérer au sein d'un réseau local les serveurs disponibles, etc.

*
* *

Les trois bibliothèques abordées dans ce chapitre offrent à NJN les supports de son implémentation. DOHC assure la gestion dynamique de la mémoire, CHASSIS offre une couche abstraite des services du système d'exploitation et CABLES gère les communications. Les deux prochains chapitres aborde l'implémentation d'NJN.

CHAPITRE 8
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.

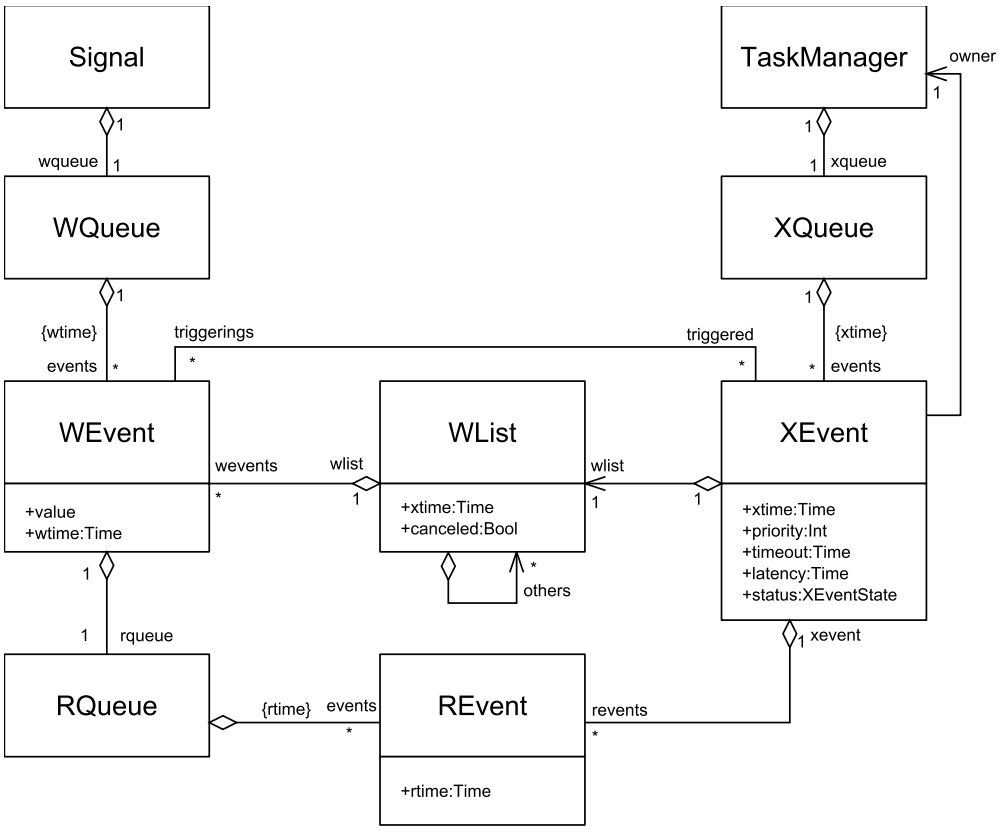


Figure 62. Diagramme de classes du graphe causal.

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.

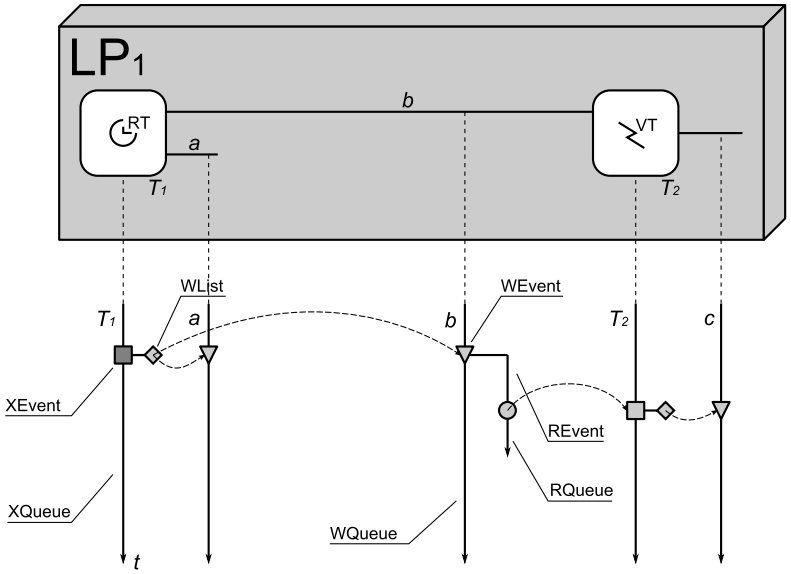


Figure 63. Le graphe causal.

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.

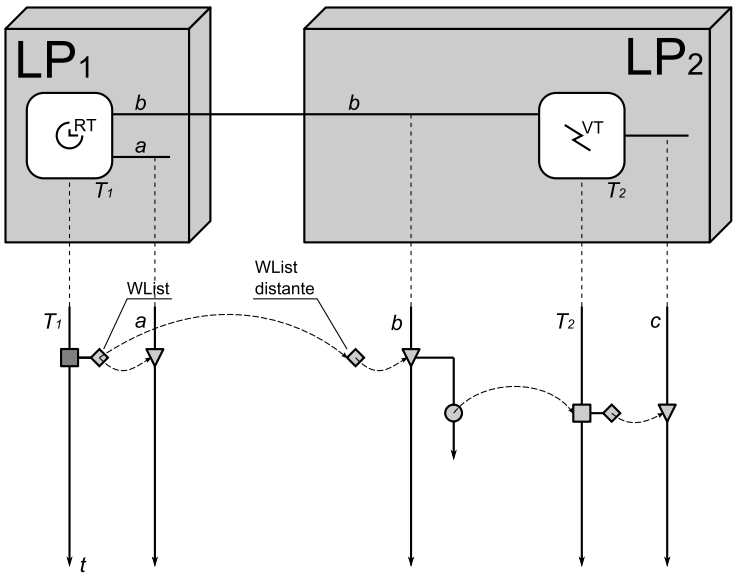


Figure 64. Lien entre XEvent et WEvent distants.

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*.

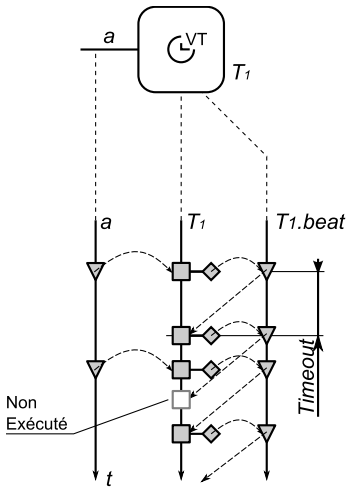


Figure 65. Implémentation du déclenchement par Timeout.

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.

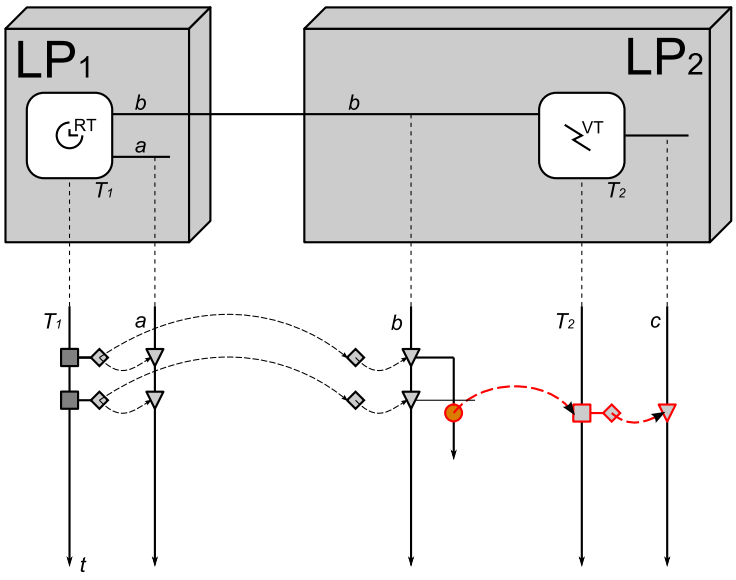


Figure 66. Identification d'une violation de causalité.

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.

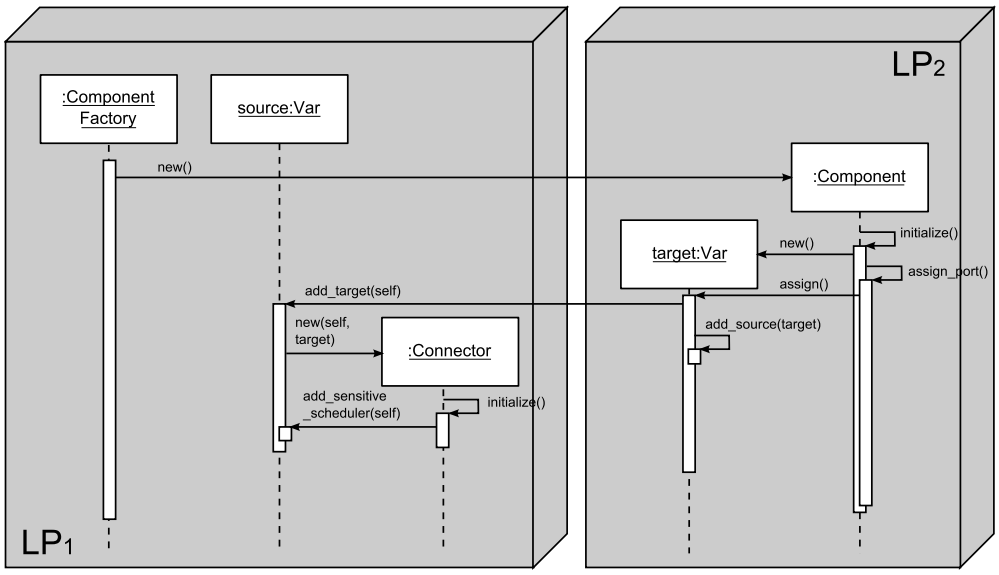


Figure 67. Création d'un composant et assignation d'un port d'entrée à une variable source.

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.

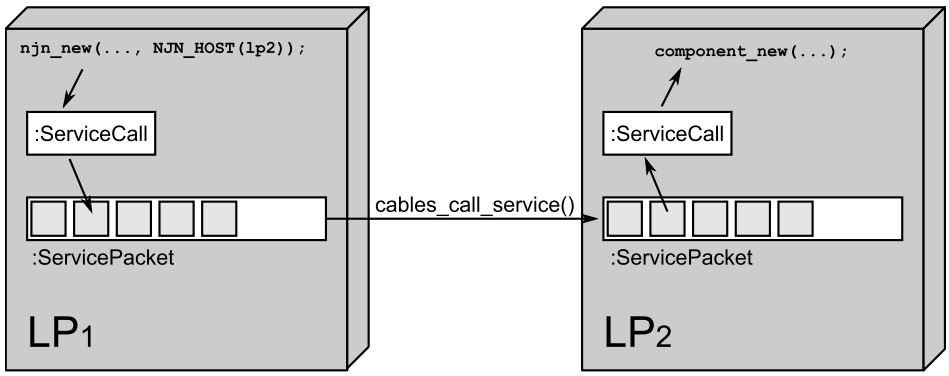


Figure 68. Appel de services pour la création d'objets à distance.

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.

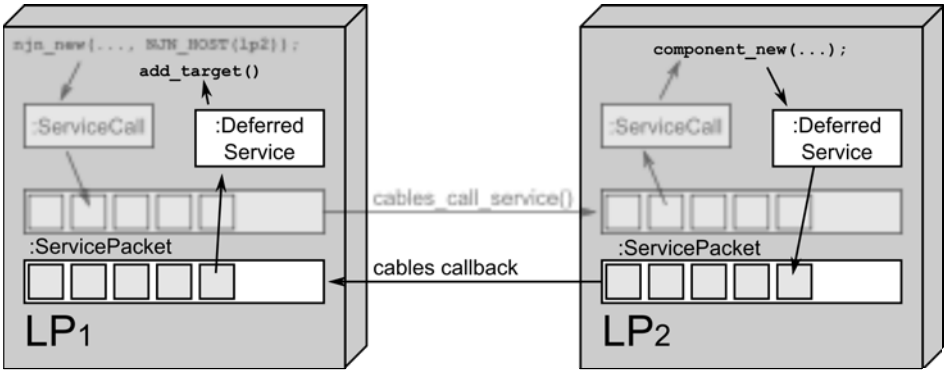


Figure 69. Retour de service.

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.

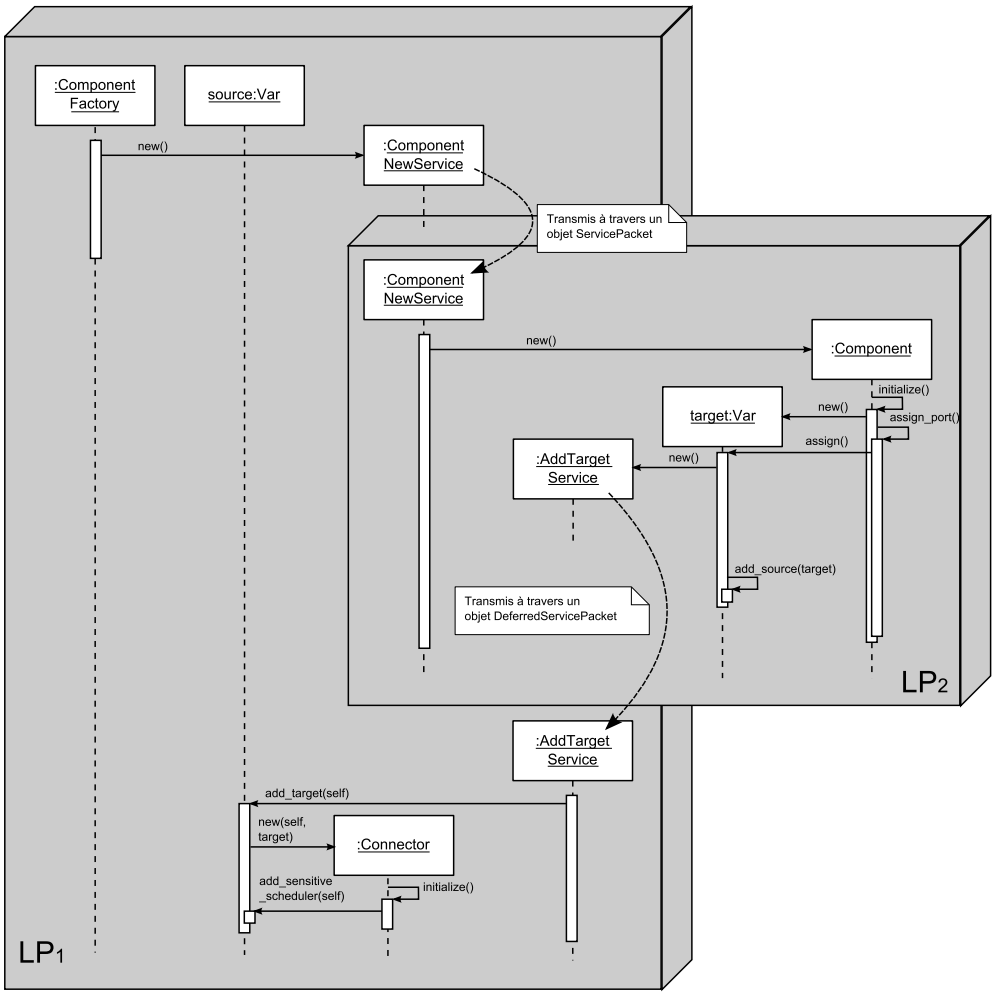


Figure 70. Création d'un composant à distance et connexion d'un port d'entrée.

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.

CHAPITRE 9
LA MACHINE (SUPERSTRUCTURE)

La superstructure gère tout ce qui concerne la création et la vie des objets de l'application. D'abord, elle assure la gestion des bibliothèques de composants et fournit une architecture de fabriques d'objets. Ces deux éléments permettent, à partir de modèles, de construire l'application. Ensuite, elle gère la structure hiérarchique constituée par l'assemblage des objets créés. Enfin elle fournit les fonctionnalités qui permettent de distribuer l'application.

Pour assurer ses fonctionnalités dynamiques, la superstructure utilise la notion de signal dont l'implémentation a été abordée au chapitre précédent. L'activité des objets, le nom des objets, la connexion entre deux variables, etc. tout dans NJN est géré par des signaux qui garantissent le caractère causal de toutes les opérations effectuées par l'application.

Ce chapitre, consacré à la superstructure, présente quelques éléments spécifiques de l'implémentation selon les trois axes : construction des objets, gestion hiérarchique et distribution.

1 Fabriques, bibliothèques de modèles et paramètres

Cette première partie dévoile les différents éléments qui interviennent dans la construction d'un objet de l'application. On commence par rappeler comment les choses s'écrivent avant de plonger dans le mécanisme de construction proprement dit.

1.1 Les objets, leur modèle et leur fabrique

La construction d'un objet est un processus en trois temps : une fabrique est construite en référence à un modèle ; elle est ensuite configurée par un ensemble de paramètres ; enfin on lui demande de créer l'objet souhaité. Ce mécanisme un peu complexe a pour but de faciliter la construction des objets, du point de vue du développeur d'application, en généralisant l'interface de programmation d'NJN.

Prenons l'exemple du composant *Sampler* déjà rencontré dans un précédent chapitre. Sa définition est la suivante :

```
NJN_DEF_COMPONENT(Sampler){
    njn_ref_t period = njn_new(self, "param", "period",
        NJN_CONSTRAINT(dohc_is_int), NJN_INITVAL(dohc_int(10 MSEC)));
    njn_new(self, "var", "in",  NJN_INPUT);
    njn_new(self, "var", "out", NJN_OUTPUT);
    njn_new(self, "rt_task", "t",
        NJN_ALWAYS(Sampler),
        NJN_LATENCY(NJN_TO_INT(period)),
        NJN_TIMEOUT(NJN_TO_INT(period)));
    return NJN_SUCCESS;
}
```

Le composant contient un objet de type *Param*, deux objets *Var* et un objet *RTTask*. Tous ces objets sont élaborés par un appel au constructeur générique *njn_new()*, quelque soit le type d'objet créé et le nombre de paramètres nécessaires pour sa construction.

Une instance de ce composant est créée en suivant la même construction :

```
    /*...*/

    njn_new(self, "DSP::Sample", "op1",
        NJN_ASSIGN("in", in),
        NJN_ASSIGN("out", i1),
        NJN_PARAM("period", dohc_int(100 MSEC));

    /*...*/
```

Le premier argument du constructeur générique *njn_new()* désigne la portée dans laquelle doit être construit l'objet.

Le second argument fourni au constructeur est le nom du modèle à utiliser. Ce modèle doit être présent dans un paquetage. S'il s'agit d'un objet de base, variable, tâche, etc., le modèle correspondant sera trouvé dans le paquetage standard d'NJN. Il n'est alors pas nécessaire de préciser le paquetage. Pour les autres, c'est-à-dire les composants et les canaux, il faudra préalablement charger une bibliothèque pour le trouver et préciser le paquetage dans lequel il se trouve. Le modèle renseigne sur la fabrique à solliciter et fournit les éléments généraux nécessaires à la construction.

Le troisième argument correspond au nom donné à l'objet créé lors de sa construction. Ce nom pourra éventuellement être modifié au cours de la vie de l'objet.

Les arguments qui suivent constituent la liste des paramètres génériques de configuration de la fabrique. Chaque macro appelée dans la liste d'arguments du constructeur retourne un objet paramètre dont le rôle sera de configurer un élément de la fabrique. Les paramètres fournissent donc les informations spécifiques de la construction d'une instance.

1.2 Mécanisme de construction de l'objet

Les objets *Pattern* implémentent les modèles. Chaque objet de cette classe pointe vers la classe de la fabrique à employer pour construire l'objet.

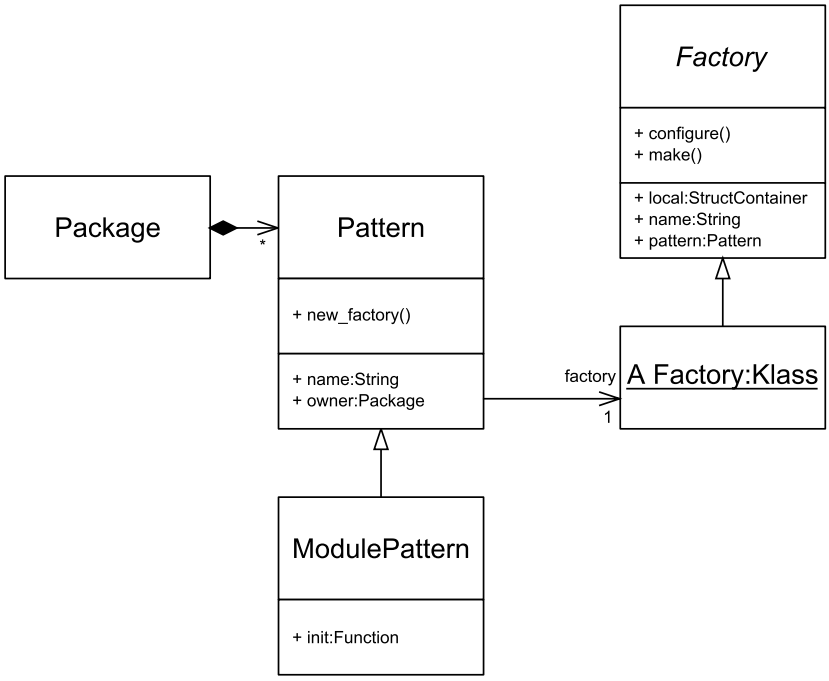


Figure 71. Les relations entre modèles et fabrique.

Il existe deux classes de modèles : les modèles correspondant à des objets de base (classe *Pattern*) tels que tâches ou variables, et les modèles correspondant à des objets conteneurs (classe *ModulePattern*), c'est-à-dire les composants et les canaux. Les objets composites ont en effet besoin d'une description explicite de leur contenu. Les objets de la classe *ModulePattern* contiennent à cet effet un champ qui pointe vers une fonction d'initialisation fournie par le développeur, qui sera appelée par le constructeur du module. Cette fonction est définie par la macro *NJN_DEF_COMPONENT()*.

La première étape de la construction consiste à créer une fabrique en appelant la méthode virtuelle *Pattern::new_factory()*. A chaque type d'objet de l'application correspond une classe de fabrique et au moins un objet modèle. Un type de fabrique ne peut en effet construire qu'un seul type d'objet. La hiérarchie de classes de fabrique et la hiérarchie de classes d'objets de l'application ont donc des structures semblables comme le montre la figure 72.

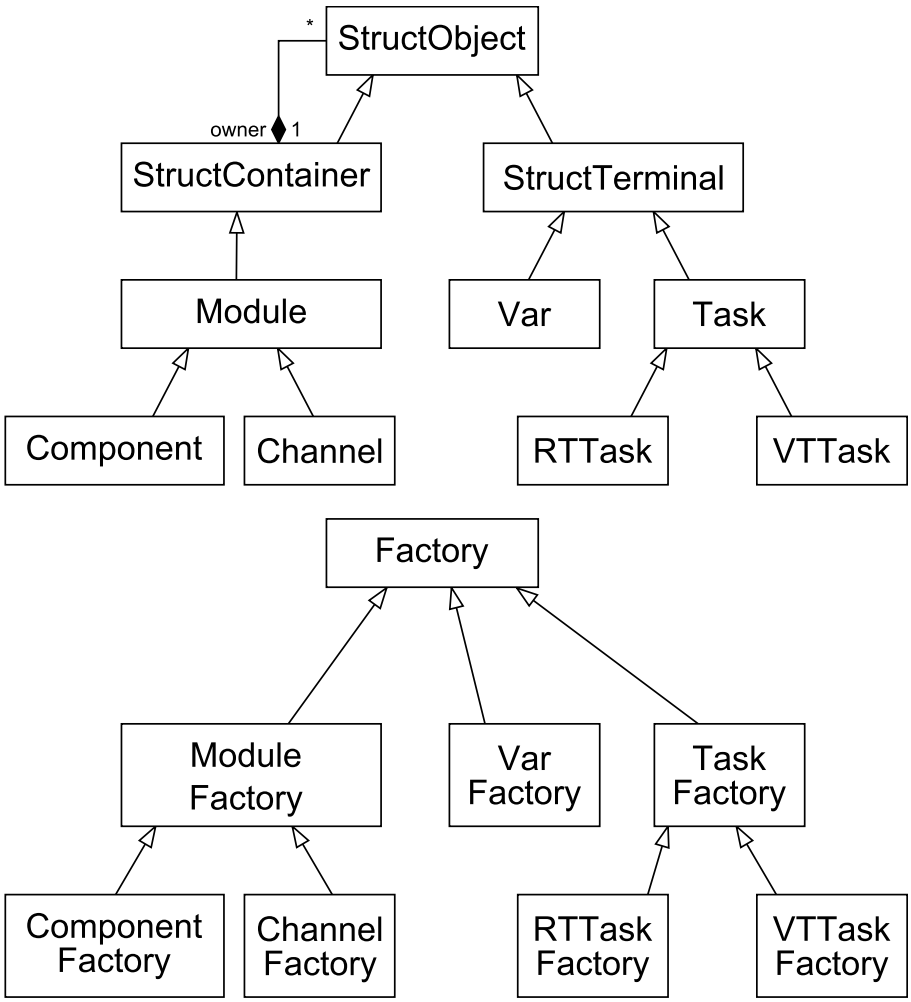


Figure 72. Diagramme de classe partiel de la hiérarchie de fabriques et d'objets structurels.

La fabrique ainsi construite doit être configurée par la méthode virtuelle *Factory::configure()* de manière à ce que toutes les informations nécessaires à la construction de l'objet soient disponibles lors de l'appel du constructeur de l'objet (figure 73). Cette méthode prend en argument une liste d'objets paramètres qui correspondent aux objets de la liste d'arguments variable fournie au constructeur générique *njn_new()*.

Chaque classe de paramètres implémente une fonction virtuelle *Parameter::configure()* qui, lorsqu'elle est appelée, réalise l'opération de configuration spécifique à la classe de fabrique qui lui est fournie en argument.

Une fois configurée, la fabrique contient toutes les informations utiles pour la construction de l'objet. Cette construction peut donc avoir lieu. NJN appelle pour ce faire la méthode virtuelle *Factory::make()*.

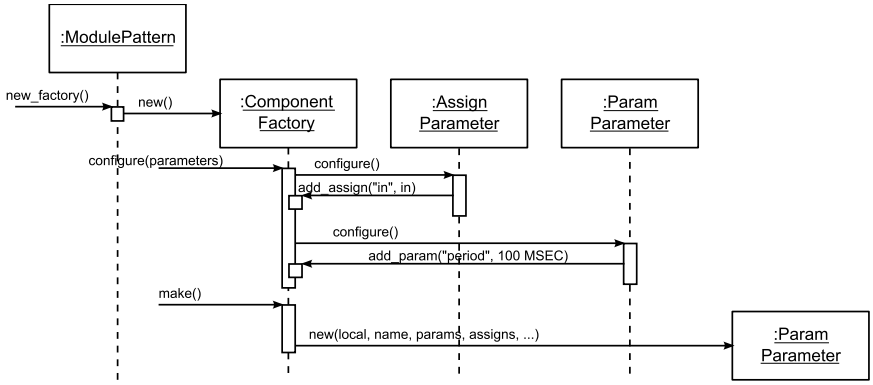


Figure 73. Configuration d'une fabrique par une liste de paramètres.

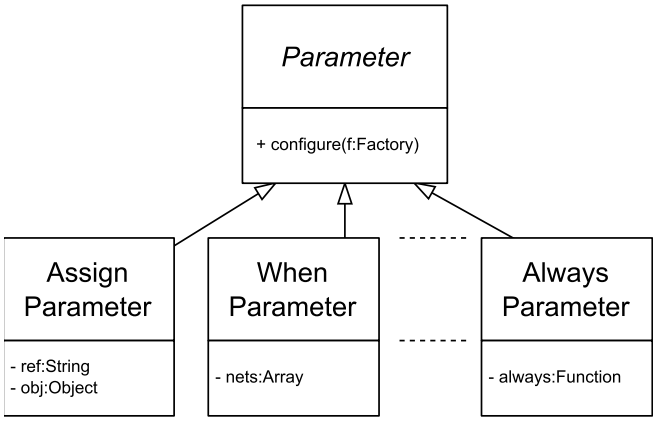


Figure 74. Différents types de paramètres.

Tout commence donc à partir d'un modèle décrit dans une bibliothèque. La section suivante se penche sur la gestion de ces modèles.

1.3 Chargement des bibliothèques

Les bibliothèques sont des unités de compilation qui prennent la forme de DLL sous le système d'exploitation à la fenêtre et de *shared object* lorsqu'on travaille avec le pingouin. NJN les chargent dynamiquement, à la demande.

Le point d'entrée de toute bibliothèque est défini par la macro *NJN_DEF_LIBRARY()* qui prend en argument le nom de la bibliothèque et son numéro de version. Cette macro place dans le code la fonction de chargement *__engine_library_loader()* qui servira de point d'entrée dans la bibliothèque.

```
NJN_DEF_LIBRARY(examples, 1.0){
    NJN_PACKAGE(self, DSP);
    return NJN_SUCCESS;
}
```

Lorsque la bibliothèque est chargée (figure 75), NJN crée une structure de données de type *Library* et fournit cette structure à la fonction de chargement de l'unité de compilation correspondant à la bibliothèque. La fonction de chargement *__engine_library_loader()* initialise alors l'objet *Library* qui lui est transmis en exécutant le bloc de code associé à la définition de la bibliothèque.

Une description de bibliothèque ne contient que des descripteurs de paquetages représentés par des appels à la macro *NJN_PACKAGE()*. Cette macro crée un objet de type *Package* dans la structure de données d'NJN et initialise celui-ci en exécutant le bloc de code associé à la définition du paquetage désigné en argument.

Dans notre exemple, la macro *NJN_PACKAGE()* crée un paquetage nommé *DSP* et appelle le bloc de code de même nom défini par la macro *NJN_DEF_PACKAGE()*.

```
NJN_DEF_PACKAGE(DSP){
    NJN_COMPONENT(self, Sample);
    /*...*/
    return NJN_SUCCESS;
}
```

L'initialisation du paquetage consiste à créer des modèles pour chaque composant (respectivement chaque canal) déclaré par un appel à la macro *NJN_COMPONENT()* (respectivement la macro *NJN_CHANNEL()*). Le modèle ainsi créé, un objet de type *ModulePattern,* est ajouté à une table de hachage interne au paquetage. De cette manière il sera possible de le retrouver à partir de son nom.

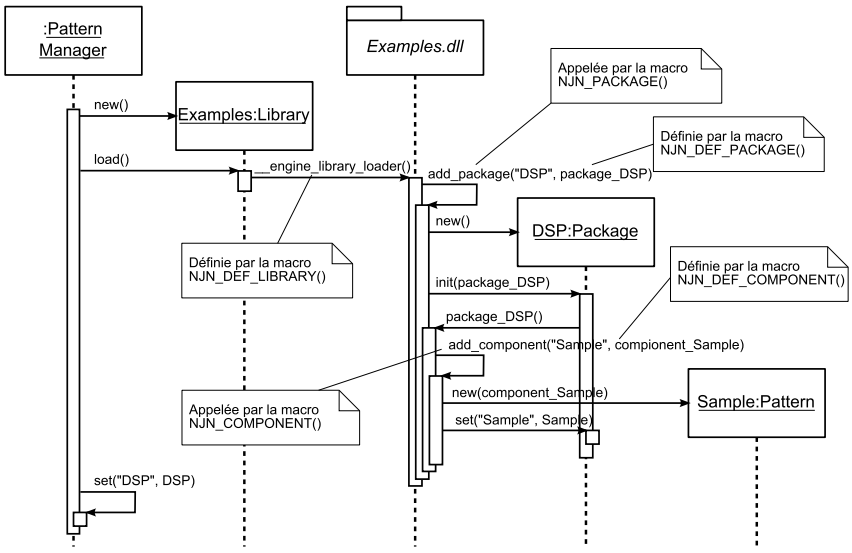


Figure 75. Séquence de chargement d'une bibliothèque.

Chaque modèle du paquetage fait référence à une fonction d'initialisation définie quelque part dans la bibliothèque par la macro *NJN_DEF_COMPONENT()*. La correspondance entre le modèle et la fonction est établie à partir du nom donné à

ces deux éléments. La fonction d'initialisation doit bien évidemment être placée dans la même unité de compilation. Elle sera appelée par la suite, lorsque le modèle qui lui est associé sera sollicité pour la création d'un composant.

```
NJN_DEF_COMPONENT(Sampler){
    /*...*/
    return NJN_SUCCESS;
}
```

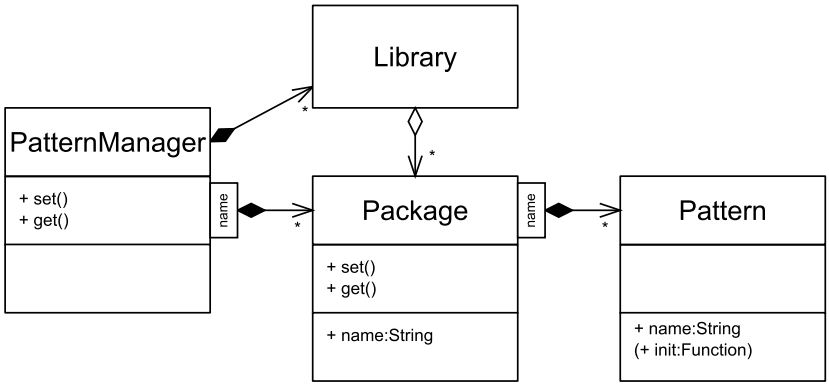


Figure 76. Relations entre bibliothèques, paquetages et modèles.

Une fois initialisé, l'objet *Package* est placé dans le gestionnaire de paquetages du noyau d'NJN.

L'ensemble des paquetages est placé dans une table de hachage. Ainsi, ils sont accessibles par leur nom. Pour solliciter un modèle, il suffit de fournir au gestionnaire de modèles d'NJN le chemin qui lui correspond : le nom de son paquetage d'abord, puis le nom du modèle. Dans notre exemple, on accède au modèle *Sample* à partir du chemin « *DSP::Sample* ».

Nous venons de voir les mécanismes qui sont mis en œuvre dans NJN pour créer les objets de la hiérarchie structurelle. La partie suivante montrera comment la vie de ces objets est gérée.

2 Structure hiérarchique

La structure hiérarchique d'NJN repose sur un modèle composite tout à fait classique (figure 72 page 174). Des conteneurs hébergent des objets pour constituer une hiérarchie d'objets. Chaque objet porte un nom, peut accéder à celui qui le contient, à ceux qu'il contient, etc.

L'originalité du modèle réside dans l’aspect temporel et dynamique des choses. Pendant l'exécution de l'application, un objet peut être créé ou détruit. Mais les tâches de l'application observent la structure en fonction de leur temps propre et non en fonction de la date donnée par l'horloge murale. Aussi les dates de création et de destruction sont positionnées sur un axe temporel absolu qu'il est possible de

consulter à n'importe quel instant. Au même instant de l'exécution, du point de vue de l'horloge murale, un objet peut être présent ou absent de la structure de l'application selon la tâche qui l'observe. Par ailleurs, à la suite d'une violation de causalité, la création ou la destruction d'un objet peut être remise en cause par un retour arrière. La représentation de la durée de vie des objets de l'application et leur existence en mémoire sont donc deux choses relativement différentes.

2.1 Les objets et leur ligne de vie

La durée de vie de chaque objet et représentée par un signal appelé ligne de vie. A un instant donné, pour savoir si l'objet en question existe dans la structure de l'application, il suffit de consulter l'état de cette ligne.

Une ligne de vie comporte au moins un évènement qui correspond à la création de l'objet. Lorsque l'objet est détruit, un second évènement est placé sur la ligne de vie pour représenter l'opération de destruction. S'il s'agit d'un conteneur, l'inscription de l'évènement de destruction est reproduite par récursion sur les objets contenus. Un objet ne peut évidemment par survivre à la destruction de son conteneur.

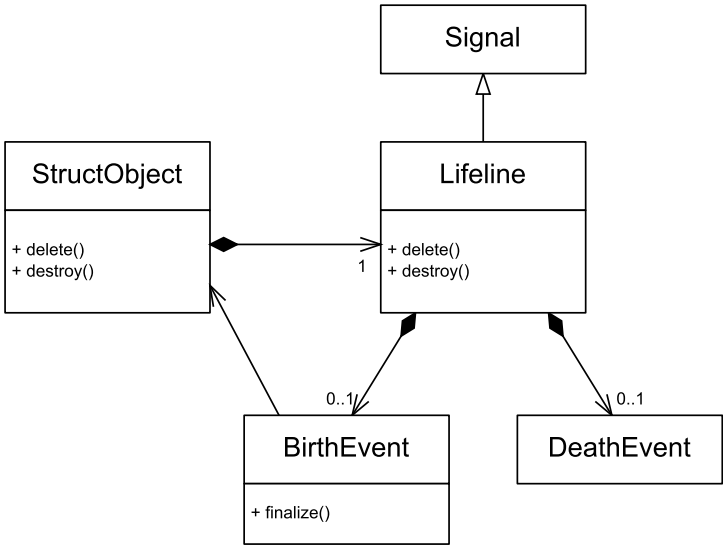


Figure 77. Relation entre un objet structurel et sa ligne de vie.

Pour savoir si un objet est présent dans la structure, il suffit de le lui demander. C'est la méthode *StructObject::is_alive()* qui permet de retrouver cette information. Elle se contente de lire l'état du signal de ligne de vie à la date qu'on lui fournit en argument. Si l'évènement consulté sur la ligne de vie à la date donnée est un évènement de création, l'objet existe du point de vue de l'application. Si l'évènement consulté est un évènement de destruction, l'objet n'a plus d'existence et ne doit faire l'objet d'aucune sollicitation ou action quelle qu'elle soit. Enfin si aucun évènement n'est consultable, c'est que l'objet considéré n'existe par encore dans l'application à la date d'observation (figure 77).

Comme tout évènement, la création ou la destruction d'un objet n'est pas une opération définitive. Des retours arrière peuvent se produire qui annulent les opérations sur la structure de l'application. L'infrastructure d'NJN gère la durée de vie des objets à travers leur ligne de vie, comme n'importe quel signal de l'application.

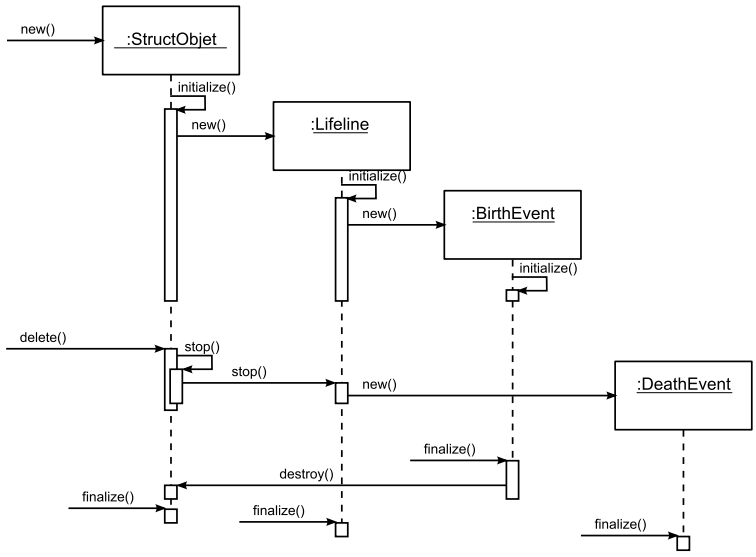


Figure 78. Séquence de création et de destruction d'un objet structurel.

La gestion mémoire des objets structurels n'est pas complètement triviale. Détruire la structure de données en mémoire lorsque l'évènement de destruction de l'objet est créé est tout simplement un non sens. Pour récupérer l'espace alloué, il faut attendre d'être certain que l'objet ne sera plus utilisé dans aucune circonstance. Cette situation se produit dans deux cas :

- L'objet a été créé mais un retour arrière a annulé l'évènement de création ;
- L'évènement de création a rejoint les objets fossiles, autrement dit, compte tenu du temps propre du moteur évènementiel et des latences des différentes tâches, l'évènement de création n'est plus accessible en lecture à aucune tâche du système.

Dans les deux cas, c'est la disparition de l'évènement de création qui témoigne du fait que l'objet considéré peut être supprimé de la mémoire (figure 78).

Pour réaliser cette opération au bon moment, l'évènement de création est associé à un objet *callback* DOHC. Ce type d'objet exécute une fonction ad-hoc lorsqu'il est nettoyé par le ramasse-miette de DOHC. Ainsi, lorsque l'évènement de création est finalisé pour être supprimé de la mémoire, la méthode *StructObject::destroy()* est appelée sur l'objet structurel associé permettant au ramasse-miette de nettoyer cet objet à son tour.

Bien que détruit par un appel à *StructObject::delete()*, un objet n'est effectivement supprimé de la structure de données que lorsque son évènement de création est

nettoyé de la mémoire. La façon dont son conteneur est impacté par cette réalité logicielle n'est pas anodine. La section suivante en dévoile la complexité.

2.2 Les composants et leur contenu

Le contenu d'un conteneur varie au cours de sa vie, au gré des créations et destructions des objets qu'il contient. Par définition, la ligne de vie des objets contenus doit être active. Par ailleurs, le nom des objets est représenté par un signal qui peut être modifié à tout moment.

Au sein d'un conteneur, on accède aux objets à travers leur nom. A un instant donné, pour accéder au contenu d'un composant, il nous faut donc disposer d'une table de hachage associant les objets actifs à leur nom. La pérennité de cette table devra être remise en cause dès lors qu'un nouvel objet est créé, détruit ou qu'il change de nom.

Toutes les tâches n'observent pas la structure de l'application au même instant. Nous avons déjà évoqué ce point plusieurs fois. Il est donc impératif de disposer de l'état du conteneur aux différents instants d'observation. Un signal sera, là encore, le support de l'information. On distingue donc deux choses : l'état structurel du conteneur au moment de l'observation et la structure de données liée à son contenu potentiel.

Du point de vue de la structure de données, les objets sont associés au conteneur tant qu'ils ont une représentation en mémoire. Une simple liste stocke l'ensemble des objets contenus sans classement ou organisation particulière.

L'état structurel du conteneur est représenté par un signal, appelé cache structurel. Le comportement de ce signal est un peu particulier. La figure 79 tente d'en montrer les caractéristiques dans le cas de l'ajout d'une variable à un composant. L'état du signal est modifié à chaque fois qu'une modification structurelle est engagée à l'intérieur d'un conteneur – création ou destruction d'un objet, changement du nom d'un objet, etc. Lorsque cela se produit, un évènement à valeur nulle est inscrit sur le cache. En lecture, c'est-à-dire lorsqu'une tâche tente d'accéder au contenu d'un composant, deux cas peuvent se présenter :

- Si le cache contient pour la date de lecture une donnée nulle, une table de hachage est construite en consultant la ligne de vie et le nom des objets potentiellement contenus dans le composant. Une fois constituée, la table se substitue à la valeur nulle de l'évènement présent dans le cache. Elle est également renvoyée à la tâche à l'origine de la lecture.
- Si le cache contient déjà une table de hachage, celle-ci est renvoyée à la tâche active.

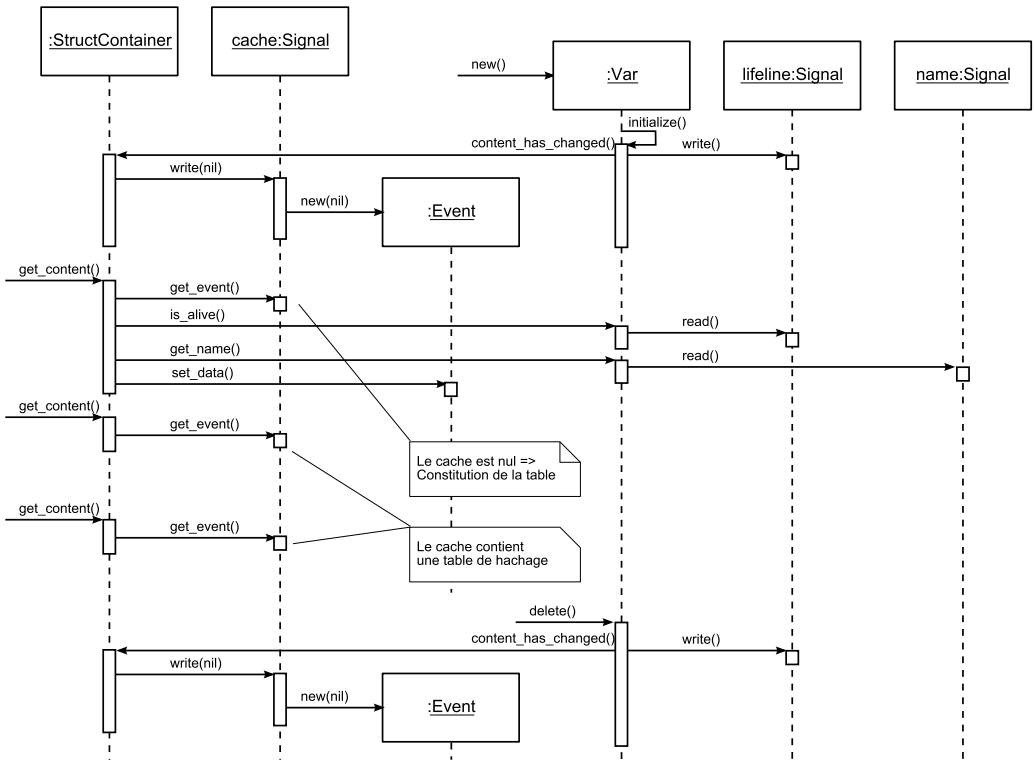


Figure 79. Gestion du cache structurel des conteneurs.

Ainsi, pour chaque section temporelle consultée, le cache contient une table de hachage représentative du contenu du composant. Chacune de ces tables n'est construite qu'une seule fois et uniquement lorsque cela est nécessaire. Les sections temporelles non consultées ne donnent en effet lieu à aucune construction de table.

2.3 Les variables, les ports et les connecteurs

La connexion des ports d'un composant est un autre problème non trivial de la vie structurelle de l'application. Tout est souvent construit en même temps : le composant, ses variables de port et les variables externes qui se connectent à ces dernières. Comme toute opération dans NJN, la création d'un composant, et de ses ports n'est visible qu'au bout d'un délai d'au moins un cycle Δ. Pour établir le plan de fabrique des interconnections, il est donc nécessaire de désigner des objets qui n'ont, à l'instant de l'opération, aucune existence.

Voyons cela sur l'exemple d'instanciation ci-dessous :

```
njn_ref_t a  = njn_new(self, "Var", "a");
njn_ref_t b = njn_new(self, "Var", "b");
njn_new(self, "DSP::Sample", "op",
    NJN_ASSIGN("in", a),
    NJN_ASSIGN("out", b),
    NJN_PARAM("period", dohc_int(100 MSEC)));

/*...*/
```

Les objets *a*, *b* et *op* sont créés simultanément. NJN étant incapable d'observer les opérations qu'il effectue dans le même cycle *Δ*, les objets en question ne seront visibles dans la structure de l'application qu'au cycle suivant. Néanmoins, le constructeur d'objet *njn_new()* renvoie une référence sur l'objet qu'il crée. Il est donc possible de manipuler les objets *a* et *b* bien qu'ils n'aient pas encore d'existence[1]. C'est cette référence qui est fournie au constructeur du composant dans les assignations de ports d'entrées/sorties. A ce stade, la correspondance entre le nom donné à la variable du langage C *a* et le nom de l'objet "*a*" est arbitraire.

L'assignation de port est désignée par le paramètre *NJN_ASSIGN()* qui associe le nom d'un port à la référence d'un objet. Dans notre exemple, le port "*in*" est associé à l'objet référencé par la variable *a*, le port "*out*" est associé à l'objet *b*. Au moment de l'appel du constructeur d'objet, le premier argument de *NJN_ASSIGN()* désigne un objet qui n'existe pas encore, ni du point de vue de l'application, ni dans la mémoire de l'exécutif. Pour cette raison, il est désigné par son nom sous la forme d'une chaine de caractères.

Lors de la construction du composant, les ports sont enfin créés.

```
NJN_DEF_COMPONENT(Sampler){
    njn_new(self, "var", "in",  NJN_INPUT);
    njn_new(self, "var", "out", NJN_OUTPUT);
    /*...*/
    return NJN_SUCCESS;
}
```

Mais encore une fois, ils n'auront d'existence au niveau de l'application qu'un cycle *Δ* plus tard. Il est donc impossible de les retrouver dans le contenu du composant pour faire l'association.

Pour résoudre ce problème, NJN crée au moment de l'instanciation du composant une cartographie temporaire de son interface future où sont référencés les variables de ports, les objets *Param*, etc. Cette table n'est accessible que pendant l'instanciation. Dès lors que l'on passe au cycle *Δ* suivant, il n'est plus possible d'y avoir accès. Lorsqu'une assignation de port est sollicitée, NJN cherche dans cette table l'objet concerné et réalise l'association avec la référence fournie à *NJN_ASSIGN()*. Ainsi, bien que ni le port, ni la variable qui s'y connecte n'aient d'existence à l'instant de l'opération, leur connexion est réalisée créant entre les deux un objet connecteur et son répéteur associé.

La structure de la connexion est représentée figure 80. Un connecteur, associé à la variable source, est créé pour représenter l'association. Il relie la variable source et la variable cible et permet à un objet répéteur de reproduire les évènements de la variable source sur la variable cible.

La variable cible n'a pas connaissance du connecteur. Elle est cependant en mesure d'identifier les variables qui lui fournissent des évènements à travers un lien vers ses sources.

1 sauf évidemment dans la structure de données d'NJN.

Comme tout objet structurel, le connecteur est associé à une ligne de vie qui peut admettre plusieurs cycles d'activité à la faveur des connexions et déconnexions successives qui peuvent être mises en jeu entre deux mêmes variables.

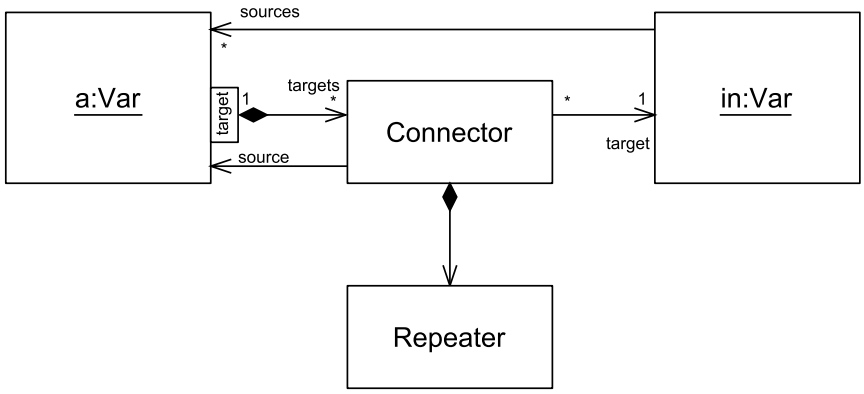


Figure 80. Relation de connexion entre deux variables.

La destruction du connecteur n'est effective que lorsque l'une des deux variables de l'association disparait de la structure de données d'NJN.

On peut noter une certaine asymétrie dans la connexion : d'une part, le connecteur est associé à la source, d'autre part, aucun lien direct ne relie la cible au connecteur. Comme l'a implicitement montré la section 2.3 du chapitre précédent (page 167), dans le cadre d'une association entre une variable et le port d'un composant distant, cette structure particulière prend tout son sens. La prochaine partie de ce chapitre reviendra sur ce point.

3 La distribution, les aller/retours et les ombres

La construction d'applications distribuées est réalisée en associant au composant instancié à distance une ombre locale qui permet au premier d'être manipulé à distance par son composant hôte. La première section établit l'ensemble des relations à mettre en place entre le composant et son ombre.

3.1 Tant de choses à faire (en un aller/retour)...

La figure 81 montre les relations qui existent entre un composant distant et son ombre.

Les connexions de port donnent lieu chacune à deux liens distants. Le connecteur est placé côté source, c'est-à-dire côté ombre pour une entrée et côté composant pour une sortie. Le connecteur est lié à sa cible distante. La cible, elle, ne voit que la source ; elle ne voit pas le connecteur.

La ligne de vie, le nom et le cache structurel du composant sont accessibles au programmeur via des propriétés. L'ombre fournit une copie locale de ces trois éléments. Les signaux qui sont le support de ces informations sont donc dupliqués

et appareillés. Toute modification effectuée sur l'une de ces propriétés côté composant est reproduite côte ombre, et réciproquement.

Les liens distants entre objets reposent sur les objets *Handle* de DOHC. Il s'agit de références qui se substituent aux objets non sérialisables lorsqu'on tente de les transmettre en argument d'un service ou valeur de retour. Ces références permettent de retrouver l'objet original lorsqu'elles sont retournées à l'hôte qui héberge le dit objet. Ainsi, il est possible, si l'on dispose d'une telle référence, de manipuler un objet distant.

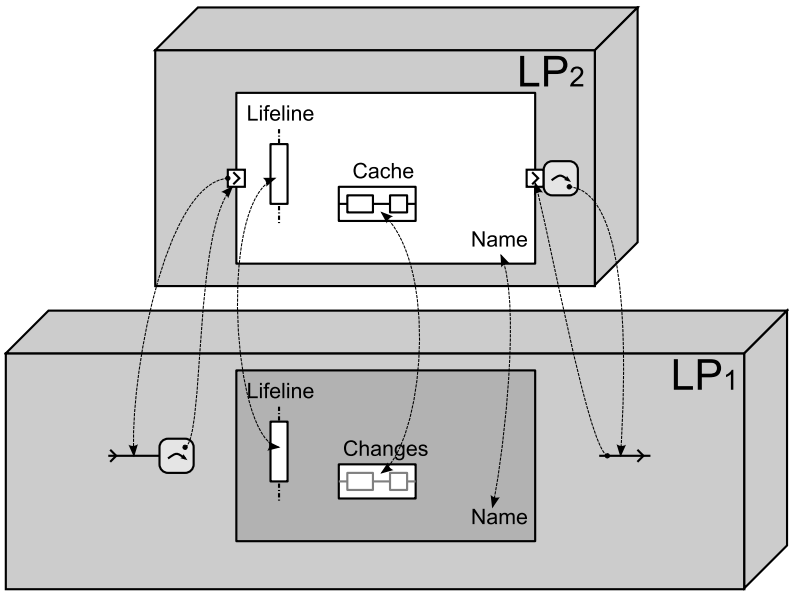


Figure 81. Relations entre un composant et son ombre.

L'élaboration à distance doit construire l'ensemble des relations entre objets en un seul appel de service ce qui pose quelques difficultés et qui influence fortement la structure des interconnexions.

3.2 La construction des composants et de leur ombre

L'instanciation à distance est initiée par la création de l'ombre locale (figure 82.a). Sont créés, l'ombre, sa ligne de vie, le signal support du nom d'instance et un signal *changes* qui a pour rôle de prévenir de tout changement dans la structure du composant distant. Une carte d'association est construite à l'aide de références à ces trois signaux. Elle sera envoyée au composant pour réaliser l'appareillage des propriétés de celui-ci avec celles de l'ombre.

La carte de connexion est également constituée et référence les variables à relier aux ports du composant.

Ces deux cartes, ainsi que la référence complète du modèle de composant à instancier – bibliothèque, paquetage et modèle – suffisent à construire l'objet

distant. Toutes ses informations sont envoyées au processus hôte via le service ad-hoc.

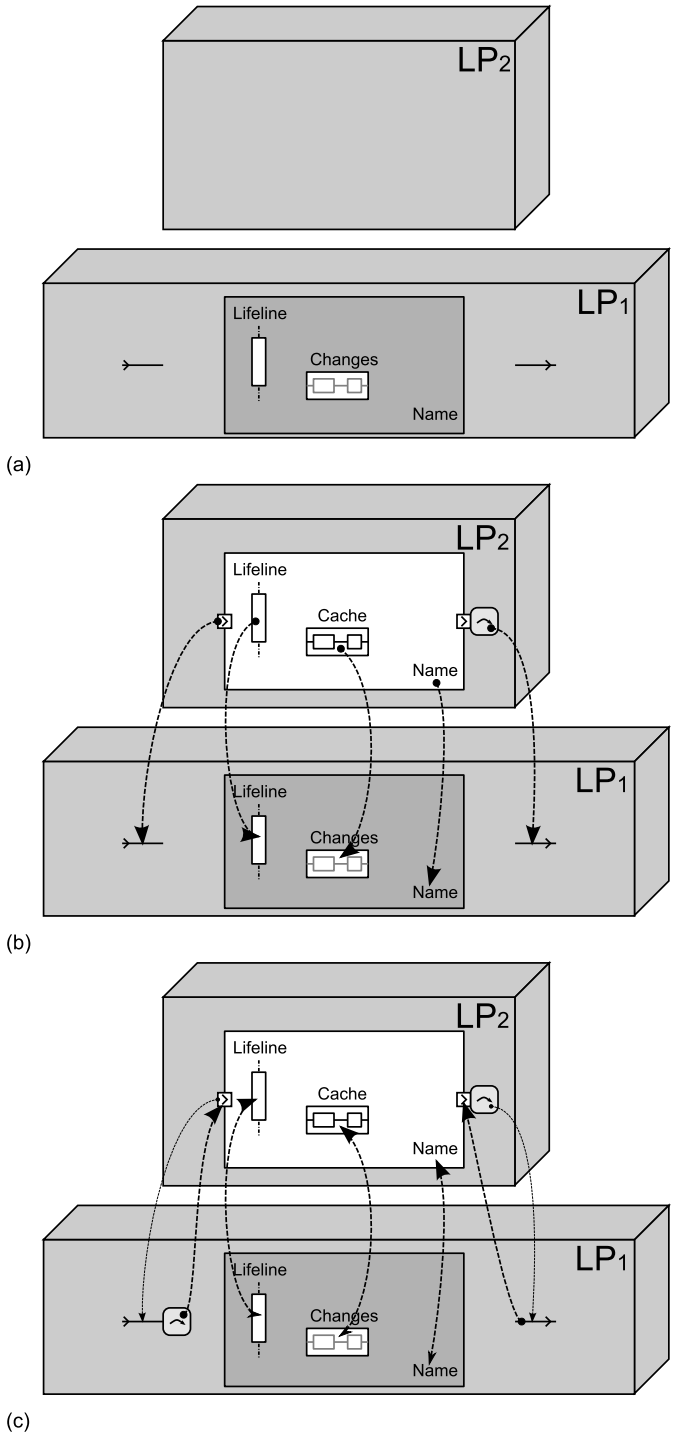


Figure 82. Séquence de construction d'un composant distant.

L'exécution du service se traduit par la construction du composant et l'appareillage de ses propriétés internes (figure 82.b). Une nouvelle carte est constituée, cette fois avec la ligne de vie, le nom et le cache du composant, à destination de l'ombre. Elle permettra à l'ombre de faire l'appareillage des propriétés correspondantes.

Les ports sont reliés partiellement : les entrées référencent leurs sources et les sorties sont chacunes associées à un connecteur. Les références nécessaires à la finalisation de la connexion côté ombre sont empilées dans la file de messages de retour pour permettre de finaliser les interconnexions.

Le retour de service permet de terminer le travail en finalisant les appareillages et les connexions (figure 82.c). Les connecteurs sont créés pour les entrées et les signaux de sorties reçoivent la référence sur leur source. La construction est alors terminée.

Entre la construction de l'ombre et le retour du service d'instanciation distante, des évènements peuvent être associés aux différents signaux mis en jeu. Il peut également se produire un retour arrière qui remet totalement en cause la construction de l'objet. Le risque, lorsque ces situations se présentent, est que les signaux appareillés perdent leur correspondance ce qui remet en cause le fonctionnement du système. Pour palier cette difficulté, NJN met en œuvre pour chaque signal concerné une file d'évènements non traités (*pending events*). Cette file stocke tous les évènements inscrits sur un signal en cours d'appareillage. Lorsque l'appareillage est enfin finalisé, c'est-à-dire que le signal concerné dispose d'une référence sur son signal jumeau, les évènements concernés sont envoyés au signal associé pour qu'il puisse être mis à jour convenablement.

Les appareillages et connexions étant réalisés, la vie du composant peut commencer.

3.3 La vie d'un composant distant vue de son ombre et vice-versa

Lorsqu'un composant distant est lié à son ombre, deux types de communications doivent être gérés par NJN. D'une part, il faut acheminer les évènements entre les signaux interconnectés et, d'autre part, il faut répercuter tout évènement se produisant sur un signal appareillé vers le signal auquel il est associé.

La connexion entre une variable et un port n'est pas d'une grande difficulté à mettre en œuvre. Le répéteur associé à un connecteur analyse la nature de l'objet cible de la connexion. S'il s'agit d'un signal ou d'une variable, tout évènement se produisant sur la source est répercuté sur la cible. S'il s'agit d'un *Handle* DOHC, c'est que la cible est distante. Dans ce cas, le répéteur fait appel à un service NJN spécifique pour mettre à jour la cible.

L'appareillage de propriétés a ceci de particulier qu'il s'agit d'une relation bidirectionnelle. Toute opération effectuée d'un côté, quelle qu'elle soit, doit être répercutée de l'autre. Pour ce faire, les signaux disposent de deux méthodes d'écriture. La première est la méthode *Signal::write()*. La seconde est la méthode

Signal::update(). Lorsqu'une tâche inscrit, par la méthode *write()*, un évènement sur un signal appareillé, un service réclame l'appel de la méthode *update()* sur le signal distant associé. Il est donc possible de modifier le nom d'un composant distant des deux côtés de la distribution.

*
* *

Ce chapitre clôt notre exposé sur l'implémentation d'NJN. Tout n'a pas été évoqué. Nous nous sommes concentrés sur quelques éléments critiques de l'implémentation concernant la construction des objets, leur fonctionnement et leur instanciation à distance. Dans un contexte distribué, toutes ses opérations ne peuvent respecter la causalité sans s'appuyer sur l'infrastructure d'NJN qui procure les mécanismes de synchronisation et de retour arrière.

L'implémentation actuelle d'NJN a permis de valider l'ensemble des concepts fondamentaux du projet AROS. Il reste cependant beaucoup de travail pour arriver à un produit efficace et commercialisable. Dans NJN, nous ne faisons aucun compromis sur la causalité et envisageons toujours le pire cas de figure : tout objet peut disparaitre à tout instant. En analysant les choses un peu plus finement et en se donnant, dans certaines situations critiques, la possibilité de rompre avec la règle de causalité, il y a assurément moyen de simplifier les choses et de rendre l'exécution plus efficace. La conclusion tentera de dégager certaines pistes d'amélioration en ce sens.

CONCLUSION ET PERSPECTIVES

L'objectif du projet AROS (*Automotive Robust Operating Services*) est de proposer une plateforme de prototypage rapide pour les applications temps-réel distribuées, principalement dans le domaine de l'automobile et celui de la robotique. Les caractéristiques attendues pour ce projet s'articulent autour de cinq points. Les applications doivent être : **temps-réel**, avec un degré de dureté modulable ; **distribuables** sur un réseau hétérogène ; **robustes et fiables** ; **reconfigurables** dynamiquement ; enfin, **faciles à développer**.

La complexité des applications temps-réel visés par AROS est telle qu'il est difficile d'obtenir une preuve d'ordonnançabilité. Les algorithmes mis en œuvre par exemple sont difficiles à estimer temporellement même si leur mise en œuvre pratique montre de bonnes propriétés. Le temps-réel dur n'est donc pas la première attente du projet. Néanmoins, moyennant la détermination de l'ordonnancement, il doit être possible d'obtenir une application dure.

Les réseaux ciblés par AROS sont fortement hétérogènes, tant du point de vue des cibles matérielles que des réseaux empruntés : du micro-contrôleur embarqué au serveur de trafic en passant par le processeur multi-coeur ; du réseau CAN ou de la simple liaison série au réseau UMTS, en passant par le réseau Ethernet. En s'appuyant sur les services du système d'exploitation hôte, un système AROS doit pouvoir se distribuer de façon transparente, sans qu'il soit nécessaire d'inscrire de façon explicite dans l'architecture de l'application les points de connexion entre les différents processus.

Parce qu'elle est liée à la sécurité des personnes et des infrastructures, une application AROS doit être fiable et robuste. Typiquement, l'arrêt accidentel d'un processus ou la perte de connexion ne doit pas mettre en danger l'application dans

son ensemble. Des techniques de monitoring, de diagnostic et de relance des processus en défaut doivent être disponibles nativement dans AROS.

Le point précédent nécessite une grande plasticité des applications. Il doit être possible avec AROS de développer des applications reconfigurables dynamiquement. Par ailleurs, pour des problématiques d'équilibrage de charge par exemple, il doit également être possible de modifier la distribution de l'application pendant son exécution, sans que cela affecte son fonctionnement.

Enfin une application AROS doit être facile à développer. Des composants doivent être disponibles en bibliothèque. Pour construire un système, il doit suffir d'instancier les composants dans l'application et de les interconnecter. Il doit être également possible de développer des composants supplémentaires de façon simple. Pour faciliter le processus de développement, l'application doit pouvoir être modifiée alors qu'elle est en cours d'exécution. Des outils de diagnostic doivent aider le développeur dans ses choix d'architecture et de distribution.

Le modèle proposé s'organise en quatre volets : **un modèle structurel**, **un modèle d'exécution**, **un modèle de distribution** et **un modèle de reconfigurabilité**. La plateforme NJN implémente, en langage C, le support de ces quatre modèles.

Comme nous l'avons évoqué plus haut, l'architecture d'une application NJN est une construction hiérarchique de composants. Les composants sont munis de ports d'entrées et de sorties qui leur permettent d'être interconnectés au sein de la hiérarchie. Ils hébergent des tâches qui forment le code actif de l'application. L'exigence de facilité de développement est principalement satisfaite grâce à ce modèle. Cette base structurelle est enrichie d'objets plus originaux tels que les canaux qui offrent une vision structurée des interconnexions et les méthodes qui confèrent à NJN une orientation service. Chacune des classes d'objets du modèle structurel est associée à un ou plusieurs modèles (*pattern*) qui fournissent les paramètres généraux nécessaires à la construction des objets. Les modèles sont rangés dans des bibliothèques qu'il suffit de charger pour être exploitables. Un modèle de composant par exemple renvoie à la procédure d'initialisation qui a en charge la construction des éléments internes du composant. Lors de la construction des objets, le modèle est complété par une liste d'arguments qui définissent les caractéristiques spécifiques de l'instance créée. Par exemple pour un composant, on précisera les variables qui sont connectées à ses ports d'entrées/sorties. Une application NJN peut être intégralement développée en instanciant et interconnectant des composants disponibles en bibliothèque. En outre, lorsqu'une fonctionnalité manque, il suffit de décrire le composant correspondant, par exemple en langage C, et de le compiler sous la forme d'une bibliothèque dynamique[1] pour qu'NJN soit en mesure de l'utiliser.

Le modèle d'exécution repose sur la définition de deux types de tâches – les tâches temps-virtuel et les tâches temps-réel – et sur un ordonnancement causal. Les tâches temps-virtuel ont en charge de traiter les données issues de l'environnement

1 *DLL* ou *Shared object* selon la plateforme.

et de produire les résultats attendus. Leur temps propre n'est pas directement lié à l'horloge murale. Il dépend pour l'essentiel des dépendances de données entre tâches. Les tâches temps-réel doivent gérer les interactions avec l'environnement, d'une part, et assurer la synchronisation temps-réel du système, d'autre part. Leur temps propre est celui de l'horloge murale. Une application élémentaire comporte donc au minimum une tâche temps-réel d'entrée qui récolte des données en provenance de capteurs, une chaine de traitement constituée de plusieurs tâches temps-virtuel et une tâche temps-réel de sortie qui assure le pilotage des actionneurs. La tâche de sortie est caractérisée par une latence temporelle qui donne à toute la chaine de traitement la plage de temps nécessaire à son exécution. L'ordonnancement mis en œuvre s'inspire des techniques de simulation évènementielle distribuée et en particulier de l'algorithme de Jefferson [42]. L'idée maitresse de cet algorithme est de permettre à chaque processus logique de l'application d'évoluer sans attendre d'avoir la certitude que les données nécessaires à l'exécution des tâches sont à jour. Chaque exécution de tâche produit des évènements – couple (valeur, date) – qui sont organisés au sein de signaux. L'apparition d'un évènement sur un signal active à son tour une ou plusieurs tâches. Lorsque l'algorithme détecte une violation de causalité liée à la mise-à-jour tardive d'une donnée du système, le système applique une procédure de retour arrière : les évènements impactés par la violation de causalité sont annulés et les tâches qui les ont produits sont ré-exécutées afin de fournir de nouvelles données cohérentes. Ce type d'ordonnancement garantit que, quelque soient les conditions d'exécution, de distribution, etc., le résultat produit par le système est toujours identique. Le comportement du système est donc parfaitement déterministe. De cette manière, on améliore l'efficacité du débogue et la fiabilité de l'application. L'implémentation de cet algorithme de synchronisation repose sur une structure de données appelée graphe causal qui retrace l'ensemble des relations de cause à effet entre les différents évènements produits pendant l'exécution. Partant d'une violation de causalité identifiée, il suffit de parcourir le graphe pour retrouver toutes les opérations invalides.

Le modèle de distribution se décompose en deux voies selon ce qui est visé par la distribution. En premier lieu, il est possible de relier par un canal de communication deux applications distantes. La distribution est alors inscrite dans la structure de l'application. Cette possibilité est celle classiquement proposée dans la plupart des plateformes distribuées du marché. Elle est bien adaptée lorsqu'on cherche par exemple à relier des sous-systèmes relativement indépendants. Dans des contextes de distribution où l'objectif est de répartir une charge de calcul importante, NJN propose un mode de distribution qui repose sur le modèle structurel. Il est possible de déporter un composant sur une cible distante, et ce, de façon totalement transparente. En plus d'assurer la répartition de charge dynamiquement, cette technique permet d'isoler les parties critiques de l'application. Un composant isolé sur un processus logique peut être monitoré à distance. En cas de défaillance du processus hôte, il est possible de relancer un nouveau processus pour y exécuter une nouvelle instance du composant et assurer ainsi la continuité du service sans que l'application en soit impactée. On met ici en œuvre certaines facultés de structuration dynamique d'NJN. La distribution

s'appuie sur la gestion de services distants asynchrones. Un processus logique peut manipuler des objets distants par l'intermédiaire de références qui permettent d'identifier l'hôte de ses objets et les objets eux-mêmes. Il suffit pour cela de faire appel à un objet service auquel on fournit la référence de l'objet manipulé. NJN se charge ensuite d'acheminer l'objet service et de l'exécuter là où se situe l'objet à manipuler.

NJN permet non seulement de gérer dynamiquement la distribution mais plus généralement de retravailler la structure complète de l'application pendant qu'elle est en cours d'exécution. Il est possible de détruire un composant ou d'en créer une nouvelle instance sans interrompre le fonctionnement du système. Cette faculté permet d'adapter l'application, de mettre à jour ses composants, sans interruption de service. Une tâche de l'application peut également agir sur les objets logiciels qui l'entourent. De cette manière, le système peut s'adapter aux changements de son environnement. En outre, une classe d'objets particuliers, les vecteurs, permet de décrire des applications génériques capables d'adapter leur taille aux besoins, en temps réel. Pour assurer ces fonctionnalités dynamiques, NJN gère la structure de l'application de la même façon que les données manipulées par les tâches, c'est-à-dire à travers des signaux. Chaque objet structurel est associé à un signal particulier appelé ligne de vie qui identifie par des évènements la date de création de l'objet et sa date de destruction. L'objet n'est réellement supprimé de la mémoire que lorsqu'on est certain qu'aucune tâche ne pourra plus y accéder.

A l'exception des vecteurs, l'ensemble des constructions de ces quatre modèles a été implémenté dans la plateforme NJN. Cependant, la version actuelle n'a pas encore la maturité nécessaire pour supporter la complexité de véritables applications. Nous reviendrons sur ce point à la fin de cette conclusion. Elle a néanmoins permis de valider tous les concepts énoncés dans les lignes précédentes.

Parmi tous les concepts mis en œuvre dans NJN, ceux liés à **la synchronisation du moteur d'exécution** et à **la gestion dynamique de la structure de l'application** sont les principales contributions de ces travaux. Les paragraphes qui suivent en développent les éléments majeurs.

Baser l'ordonnancement et la synchronisation temps-réel d'une application distribuée sur un protocole inspiré de la simulation apparaît assez tôt dans la littérature puisque Jefferson lui-même évoque ce point dès 1985 [42]. C'est cependant dans un contexte de simulation interactive ou de simulation *hardware in the loop* que cette approche est le plus souvent exploitée. Travailler avec ce type d'ordonnancement, dans des contextes temps réel relativement durs n'apparait que très récemment, en 2006, avec Zhao [93]. L'objectif est avant tout de rompre avec la pratique de l'ordonnancement à priorités qui condamne les applications au non-déterminisme comportemental. L'ordonnancement proposé par Zhao utilise un protocole conservatif qui nécessite une analyse statique de la topologie de l'application. Il n'est donc pas possible, dans cette approche, de modifier l'application en cours de fonctionnement.

L'originalité de notre approche est de s'appuyer sur le protocole optimiste de Jefferson pour, d'une part, permettre de modifier la structure de l'application pendant son exécution sans remettre en cause durablement le respect des contraintes temps-réel et, d'autre part, relâcher les dépendances temporelles entre processus logiques en cas d'arrêt accidentel d'une partie de l'application. Plus difficile à prévoir temporellement que les protocoles conservatifs, l'algorithme de Jefferson offre en contrepartie des facultés d'adaptation remarquables et un comportement parfaitement déterministe.

Pour améliorer la maitrise temporelle du système, nous retardons l'exécution effective des tâches temps-virtuel [26] de telle sorte que les retours arrière deviennent statistiquement improbables. Chaque tâche de l'application est exécutée avec un retard d'exécution vis-à-vis de sa date d'activation que nous appelons le délai de garde. Il est déterminé statistiquement à partir de la datation des évènements du système. Lorsque l'application a atteint un point de stabilité, les délais de garde suffisent à éviter les retours arrière dans la majorité des cas. On tente ainsi de s'approcher du comportement d'un protocole conservatif. Lorsqu'un bouleversement structurel majeur se produit dans l'application, les temps de transit des données entre les tâches sont bouleversés ce qui remet en cause le point de stabilité du système. Le protocole de synchronisation garantit alors la causalité des données produites grâce au mécanisme de retour arrière. Dans le même temps NJN réévalue les délais de garde des différentes tâches de manière à retrouver un point de stabilité. Notons cependant que, même si les quelques expérimentations menées montrent de bonnes réactions du système sur de petites applications, la stabilité du contrôle par la garde n'a pas été totalement étudié. Nous reviendrons sur ce point.

La stratégie de synchronisation temps-réel que nous avons adoptée comporte également certaines originalités. Comme dans beaucoup de simulateurs interactifs [41], les tâches de notre système sont partagées en deux grandes familles : celles qui interagissent avec le monde réel – nous les appelons tâches temps-réel – et celles qui se contentent de consommer des données et d'en produire sans lien direct avec l'extérieur – nous les appelons tâches temps-virtuel. Dans tous ces systèmes, l'exécution des tâches temps-réel est synchronisée sur l'horloge murale alors que celle des tâches temps-virtuel ne l'est pas. En règle générale, chaque tâche, qu'elle soit réelle ou virtuelle, est associée à un délai correspondant grosso modo au temps nécessaire à son exécution, de telle sorte que les contraintes temporelles entre l'entrée et la sortie soient relâchées.

L'originalité de notre approche consiste à associer aux tâches temps-réel et à elles seules, un délai caractéristique appelé latence dont le rôle est d'absorber les temps de calcul entre les entrées et les sorties du système. La latence définit deux choses :

- la durée qui sépare la date d'activation de la tâche de sa date d'exécution ;
- le retard avec lequel la tâche voit les données de l'application pendant son exécution.

Pour une chaine de traitement donnée, un seul paramètre temporel est donc à déterminer : la latence de la tâche de sortie. Il est par ailleurs possible de l'adapter en temps-réel aux propriétés d'exécution de l'application puisque NJN autorise la modification de la latence en cours d'exécution.

Il y a une autre conséquence plus remarquable à ce choix : les données consommées par la chaine de traitement sont toutes en cohérence temporelle. En effet, les tâches de traitement – de type temps-virtuel – n'introduisent aucun délai[1] entre la date des données qu'elles consomment et la date des données qu'elles produisent. L'intégralité de la chaine de traitement manipule donc des données datées in fine par les tâches temps-réel d'entrée qui sont à l'origine de l'exécution. La date associée à chaque évènement devient indépendante de la durée effective d'exécution des tâches de traitement. L'application traite, tout au long de la chaine, des données synchrones. L'étude de cas du chapitre 6 a montré l'intérêt d'un tel principe de fonctionnement lorsqu'il s'agit de fusionner des données d'origines diverses. Alors que dans les systèmes temps-réel classiques la fusion des données est problématique, NJN prend naturellement en charge cet aspect.

La gestion dynamique de la structure de l'application pose de redoutables problèmes compte-tenu du protocole de synchronisation utilisé. Le premier concerne la création ou destruction d'un objet. Ces deux actions peuvent être remises en cause à tout moment par un retour arrière. Le second est lié à la présence conjointe dans l'application de dates d'observation différentes. Au même instant de l'exécution, les temps propres des tâches de l'application ne sont pas identiques. Selon la tâche en cours d'exécution, un objet peut donc être présent ou absent de la structure en fonction de la date à laquelle la tâche observe les données du système. Il n'y a donc pas de rapport immédiat entre l'existence d'un objet en mémoire et sa présence dans la structure de l'application. La vie de l'objet doit donc être représentée par autre chose. NJN utilise un signal particulier nommé ligne de vie qui est associé à chaque objet de la structure hiérarchique de l'application. Pour chaque opération impliquant un objet, la tâche active doit vérifier l'ensemble des objets présents dans sa portée. Concrètement, lorsque la méthode *njn_get()* est appelée, NJN balaie l'ensemble des objets de la portée locale et interroge leur ligne de vie pour ne retenir que les objets actifs. Si une modification structurelle vient remettre en cause la structure observée, un retour arrière des tâches concernées est opéré naturellement par l'infrastructure d'NJN. Cette manière de procéder garantit la causalité du système, y compris lorsque sa structure est remaniée.

Comme le montre cette dernière remarque, le maitre mot de ce projet est bien la causalité de l'exécution, en toute circonstance. Cette caractéristique permet à NJN d'avoir un comportement toujours déterministe, ce qui était bien le but recherché.

Cependant, les performances d'exécution de l'implémentation actuelle de notre plateforme montrent les limites de cette approche causale sans concession. L'ensemble des concepts développés forme un édifice parfaitement cohérent

1 Excepté un cycle Δ.

qu'une optimisation des structures de données et des algorithmes pourra sans doute préserver un peu. Mais l'épreuve sur le terrain montre à l'évidence que l'optimisation ne sera pas suffisante. Un compromis doit être trouvé pour atteindre des performances acceptables. Les paragraphes suivants tentent de faire la part des choses entre ce qui peut être amélioré dans le cadre strictement causal fixé et ce qui peut faire l'objet de compromis.

Quelque soit l'objet considéré, NJN considère que le pire est toujours probable, c'est-à-dire que l'objet peut apparaitre ou disparaitre à chaque instant de l'exécution. Cette hypothèse de départ fait émerger nombre de cas plus ou moins tordus que l'implémentation résout en effectuant certaines opérations « au cas où » ou en consultant frénétiquement la ligne de vie des objets manipulés.

Par exemple, les tâches sont activées indépendamment de l'état de leur ligne de vie. Une tâche, même morte, est systématiquement activée dans les conditions de sa liste de sensibilité tant que la structure de données qui la représente n'est pas supprimée de la mémoire. Cette précaution est prise au cas où l'évènement signalant la destruction de la tâche soit remis en cause. La ligne de vie n'est consultée qu'au moment de l'exécution. Si la ligne de vie est inactive, le corps de la tâche n'est pas exécuté. La lecture systématique de la ligne de vie garantit qu'en cas de retour arrière, la tâche sera bien réactivée. De façon générale, la ligne de vie des objets manipulés est consultée à chaque opération pour s'assurer de l'existence des objets.

Or, une analyse un peu rapide des flux de données dans NJN montre qu'il est peu probable qu'une tâche soit réveillée par un évènement si le composant qui l'héberge n'est pas actif. En effet, sauf dans les cas de génération périodique autonome, les données consommées transitent à travers des répéteurs des ports d'entrées/sorties qui stoppent toute activité dès lors que leur composant hôte disparait. Il y a donc bien souvent redondance dans la vérification des lignes de vie des simples tâches. Une analyse plus fine permettrait sans doute d'identifier les configurations pour lesquelles la consultation de la ligne de vie est indispensable. Un tel travail ne remettrait nullement en cause la reproductibilité comportementale de l'application et permettrait, nous le pensons, de faire des économies de temps substantielles.

Dans le même ordre d'idée, chaque connexion entre deux variables cache un répéteur, une tâche dont le rôle est de reproduire sur sa cible les évènements constatés sur sa source. La présence de ces répéteurs est justifiée par le possible remaniement des interconnexions pendant l'exécution. Dans les parties statiques de l'application, les répéteurs sont donc inutiles. Deux variables connectées de façon définitive peuvent partager le même signal support. Cela permettrait, d'une part, de faire l'économie mémoire de nombreux événements et, d'autre part, d'économiser le temps nécessaire à l'activation et l'exécution des répéteurs ainsi supprimés.

Pour aller plus loin, nous pensons que les applications nécessitant des modifications structurelles importantes et pour lesquelles la rupture de service ou de causalité est intolérable sont certainement très limitées. Il pourrait être envisagé

de définir des domaines d'exécution pour lesquels la causalité est garantie quelque soit l'opération engagée et d'autres domaines ou les modifications structurelles, plus rares, pourraient entrainer quelques incohérences transitoires. Un compromis entre rapidité d'exécution et reproductibilité serait donc possible en cas de restructuration de l'application.

Le choix d'un moteur d'exécution systématiquement optimiste est finalement assez radical. Ce choix est tout à fait cohérent lorsqu'on travaille sur des cibles de taille importante capables d'exécuter plusieurs processus logiques indépendants et lorsqu'on cherche à isoler des portions de code qui manquent de fiabilité. L'algorithme de Jefferson permet en effet de rendre indépendant chaque processus logique. Typiquement, si un processus s'arrête inopinément, le reste de l'application est simplement privée de données en provenance de ce processus. Elle n'est pas bloquée en attente de données qui n'arriveront jamais.

Pour les cibles matérielles de petite taille, l'algorithme est trop lourd à gérer pour un gain finalement assez réduit puisque ces cibles n'admettent bien souvent qu'un processus unique. On pourrait envisager par exemple de simplifier la synchronisation en raisonnant uniquement sur la garde ou en mettant en œuvre un protocole conservatif sur certaines zones localisées de l'application. Parallèlement, il est possible d'entrelacer l'ordonnancement des tâches temps-réel et des tâches temps-virtuel de manière à ne constituer qu'une liste d'exécution pour laquelle le temps évolue de façon continument croissante. L'avantage immédiat serait de réduire la plage temporelle sur laquelle travaillent les différentes tâches. Ces deux mesures permettraient de réduire le volume de données conservées par l'application et donc de faire des économies de mémoire.

Bien évidemment, la stratégie précédente est incompatible avec les retours arrière ce qui, dans les cas limites, compromet la causalité de l'application dans son ensemble. Mais on peut noter que les cibles de petite taille sont naturellement destinées à gérer le pilotage de capteurs et d'actionneurs. Elles hébergent donc en général des tâches temps-réel pour lesquelles les retours arrière sont impossibles.

De façon plus générale, la séparation entre l'infrastructure et la superstructure pourrait être beaucoup plus nette. Selon les applications et les contextes d'exécution, il pourrait être intéressant de choisir un type de moteur plutôt qu'un autre. En définissant une API d'infrastructure, il devient possible de développer plusieurs moteurs que le développeur pourra choisir pour optimiser certains aspects de l'exécution plutôt que d'autres.

D'autres possibilités d'optimisation sont sans doute possibles. Les structures de données que nous avons construites ne sont pas toujours optimales et résultent d'intuitions pas toujours évidentes à vérifier. Une analyse de performance devra être effectuée de manière à identifier les goulots d'étranglement et les opérations à optimiser. Par exemple, les listes d'évènements sont identiques dans l'implémentation actuelle, qu'il s'agisse de listes d'évènements d'écriture, de lecture ou d'exécution. Les opérations effectuées sur ces trois types de listes ne sont pas identiques. Il y a donc certainement matière à optimisation selon la nature des évènements les constituant.

En marge de ces optimisations, notons que la question de la stabilité de la garde n'a pas été analysée. Le problème est relativement complexe à analyser compte tenu de l'explosion combinatoire des situations possibles. Pourtant, cette analyse est indispensable. Un gros travail de modélisation et d'analyse devra être fait pour déterminer la meilleure façon de fixer le délai de garde des tâches temps-virtuel.

Pour conclure ce mémoire, autorisons-nous à quelques conjectures sur l'évolution future des pratiques dans le domaine des applications temps-réel distribuées. La vision bipolaire de Kopetz [43] qui opposait l'approche orthodoxe et sûre de l'ordonnancement temporel (*time triggered*) et l'approche plus souple mais imprévisible de l'ordonnancement évènementiel (*event triggered*) par priorité devient obsolète car une troisième approche émerge qui tente de rendre l'évènementiel à la fois déterministe et plus sûr. La littérature montre en effet un intérêt très récent pour les ordonnancements évènementiels causaux. Des travaux tels que ceux de Ghosal [33], Liu [53] et Zhao [93] ou les réflexions très critiques de Lee [49] sur les pratiques traditionnelles montrent que ce type d'approche est dans l'air du temps. Par ces travaux, nous avons modestement contribué à faire avancer la communauté dans cette direction.

BIBLIOGRAPHIE

[1] Abrams, Marc (1988), The object library for parallel simulation (OLPS). In *Proceedings of the 20th conference on Winter simulation* (WSC '88, New York, NY, USA). ACM, p. 210--219.

[2] Bagrodia, Rajive L. ; Shen, Chien-Chung. MIDAS: Integrated Design and Simulation of Distributed Systems. *IEEE Trans. Softw. Eng.*, , October 1991, vol. 17, no. , p. 1042--1058.

[3] Bauer, H. ; Sporrer, C.. Reducing Rollback Overhead In Time-warp Based Distributed Simulation With Optimized Incremental State Saving. *Simulation Symposium, 1993. Proceedings., 26th Annual*, , Mar-1 Apr 1993, vol. 0, no. , p. 12-20.

[4] Bauer, David ; Yaun, Garrett ; Carothers, Christopher D. ; Yuksel, Murat ; Kalyanaraman, Shivkumar (2005), Seven-O'Clock: A New Distributed GVT Algorithm Using Network Atomic Operations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (PADS '05, Washington, DC, USA). IEEE Computer Society, p. 39--48.

[5] Bernat, Guillem ; Colin, Antoine ; Petters, Stefan M. (2002), WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium* (RTSS '02, Washington, DC, USA). IEEE Computer Society, p. 279--.

[6] Botelho, S.C. ; Alami, R. (2000), A multi-robot cooperative task achievement system. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on* (,). , p. 2716 -2721 vol.3.

[7] Christopher D. Carothers ; Kalyan S. Perumalla ; Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, , 1999, vol. 9, no. 3, p. 224--253.

[8] *CArtographie par RObоT d'un TErritoire* (http://www.defi-carotte.fr/).

[9] Damien Chabrol ; Guy Vidal-Naquet ; Vincent David ; Christophe Aussaguès ; Stéphane Louise (2004), OASIS. A Chain of Development for Safety-Critical Real-Time Systems. In *Embedded Real-Time Systems* (, Toulouse, France). , p. .

[10] A Chandler ; J Finney (2005), Rendezvous: Supporting Real-time Collaborative Gaming in High Latency Environments. In *Proceedings of the Second ACM International Conference on Advances in Computer Entertainment Technology (ACE 2005) in co-operation with SIGCHI,15-17* (,). , p. .

[11] Chandy, K.M. ; Misra, J.. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, , sept. 1979, vol. SE-5, no. 5, p. 440 - 452.

[12] K. M. Chandy ; J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, , 1981, vol. 24, no. 4, p. 198--206.

[13] Gilbert Chen ; Boleslaw K. Szymanski (2002), Lookback: a new way of exploiting parallelism in discrete event simulation. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation* (, Washington, DC, USA). IEEE Computer Society, p. 153--162.

[14] Chetlur, Malolan ; Wilsey, Philip A. (2006), Causality information and fossil collection in timewarp simulations. In *Proceedings of the 38th conference on Winter simulation* (WSC '06,). Winter Simulation Conference, p. 987--994.

[15] Cornhill, Dennis ; Sha, Lui. Priority inversion in Ada. *Ada Lett.*, , November 1987, vol. VII, no. , p. 30--32.

[16] Dahl, Ole-Johan ; Nygaard, Kristen. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, , September 1966, vol. 9, no. , p. 671--678.

[17] Damani, Om. P. ; Garg, Vijay K.. Fault-tolerant distributed simulation. *SIGSIM Simul. Dig.*, , July 1998, vol. 28, no. , p. 38--45.

[18] Das, Samir ; Fujimoto, Richard ; Panesar, Kiran ; Allison, Don ; Hybinette, Maria (1994), GTW: a time warp system for shared memory multiprocessors. In *Proceedings of the 26th conference on Winter simulation* (WSC '94, San Diego, CA, USA). Society for Computer Simulation International, p. 1332--1339.

[19] Deutsch, L. Peter ; Bobrow, Daniel G. An efficient, incremental, automatic garbage collector. *ACM*, , 1976, vol. 19, no. 9, p. 552--526.

[20] Dias, A.F. ; Akil, M. ; Lavarenne, C. ; Sorel, Y. (2000), Vers la synthèse automatique de circuits à partir de graphes algorithmiques factorisés. In *Actes du 5ème Workshop AAA sur l'Adéquation Algorithme Architecture* (,). , p. .

[21] Dickens, P. M. ; Reynolds,Jr., P. F. (1990). SRADS with Local Rollback. , , .

[22] E. Dijkstra. Cooperating sequential processes. *Programming Languages*, , 1968, vol. , no. , p. .

[23] Ekl\"{o}f, Martin ; Moradi, Farshad ; Ayani, Rassul (2005), A framework for fault-tolerance in HLA-based distributed simulations. In *Proceedings of the 37th conference on Winter simulation* (WSC '05,). Winter Simulation Conference, p. 1182--1189.

[24] Fabbri, Alessandro (1999), GVT and scheduling in space time memory based techniques. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation* (PADS '99, Washington, DC, USA). IEEE Computer Society, p. 54--61.

[25] Ferscha, A. ; Luthi, J.. Estimating rollback overhead for optimism control in Time Warp. *Simulation Symposium, 1995., Proceedings of the 28th Annual*, , Apr 1995, vol. 0, no. , p. 2-12.

[26] Ferscha, Alois (1995), Probabilistic adaptive direct optimism control in Time Warp. In *PADS '95: Proceedings of the ninth workshop on Parallel and distributed simulation* (, Washington, DC, USA). IEEE Computer Society, p. 120--129.

[27] Fleischmann, J. ; Wilsey, P.A.. Comparative analysis of periodic state saving techniques in time warp simulators. *Parallel and Distributed Simulation, 1995. (PADS'95), Proceedings., Ninth Workshop on (Cat. No.95TB8096)*, , Jun 1995, vol. 0, no. , p. 50-58.

[28] Steve Franks ; Fabian Gomes ; Brian Unger ; John Cleary. State saving for interactive optimistic simulation. *SIGSIM Simul. Dig.*, , 1997, vol. 27, no. 1, p. 72--79.

[29] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, , 1990, vol. 33, no. 10, p. 30--53.

[30] Fujimoto, Richard M.. Time warp on a shared memory multiprocessor. *Trans. Soc. Comput. Simul. Int.*, , January 1990, vol. 6, no. , p. 211--239.

[31] Fujimoto, Richard M. ; Hybinette, Maria. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul.*, , October 1997, vol. 7, no. , p. 425--446.

[32] Gafni, A. (1988), Rollback mechanisms for optimistic distributed simulation systems. In *SCS Multiconference on Distributed Simulation* (,). , p. pp. 61-67.

[33] A. Ghosal ; T. A. Henzinger ; C. M. Kirsch ; M. A. Sanvido. (2004), Event-driven programming with logical execution times.. In *In Seventh International Workshop on Hybrid Systems: Computation and Control (HSCC),* (,). Springer-Verlag, p. 357-371.

[34] Kaushik Ghosh ; Richard M. Fujimoto ; Karsten Schwan. Time Warp simulation in time constrained systems. *SIGSIM Simul. Dig.*, , 1993, vol. 23, no. 1, p. 163--166.

[35] Kaushik Ghosh ; Richard M. Fujimoto ; Karsten Schwan (1994), PORTS: a parallel, optimistic, real-time simulator. In *PADS '94: Proceedings of the eighth workshop on Parallel and distributed simulation* (, New York, NY, USA). ACM, p. 24--31.

[36] Gomes, Z. Xiao F. ; Unger, B. ; Cleary, J. (1995), A fast asynchronous GVT algorithm for shared memory multiprocessor architectures. In *Proceedings of the ninth workshop on Parallel and distributed simulation* (PADS '95, Washington, DC, USA). IEEE Computer Society, p. 203--208.

[37] Gordon, Geoffrey (1961), A general purpose systems simulation program. In *Proceedings of the December 12-14, 1961, eastern joint computer conference: computers - key to total systems control* (AFIPS '61 (Eastern), New York, NY, USA). ACM, p. 87--104.

[38] Mathieu Hillion (2009). *Contrôle de combustion en transitoires des moteurs à combustion interne*. Mémoire, Mines Paristech.

[39] Maria Hybinette ; Richard Fujimoto (1999), Optimistic Computations In Virtual Environments. In *In Proceedings of the Virtual Worlds and Simulation Conference (VWSIM'99* (,). , p. 39--44.

[40] Hybinette, Maria ; Fujimoto, Richard M.. Cloning parallel simulations. *ACM Trans. Model. Comput. Simul.*, , October 2001, vol. 11, no. , p. 378--407.

[41] Hybinette, Maria ; Fujimoto, Richard M.. Latency hiding with optimistic computations. *J. Parallel Distrib. Comput.*, , March 2002, vol. 62, no. , p. 427--445.

[42] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, , 1985, vol. 7, no. 3, p. 404--425.

[43] Kopetz, Hermann (1991), Event-Triggered Versus Time-Triggered Real-Time Systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond* (, London, UK). Springer-Verlag, p. 87--101.

[44] Kopetz, H. ; Bauer, G.. The time-triggered architecture. *Proceedings of the IEEE*, , January 2003, vol. 91, no. 1, p. 112 - 126.

[45] *LabVIEW* (http://www.ni.com/labview/).

[46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, , 1978, vol. 21, no. 7, p. 558--565.

[47] Claude Laurgeau (2009). *Le siècle de la voiture intelligente*. Mines ParisTech.

[48] Lavarenne, C. ; Sorel, Y.. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, , 1997, vol. Volume 14 - Numéro 6, no. , p. 569-578.

[49] Lee, Edward A. ; Zhao, Yang (2005), Reinventing computing for real time. In *Proceedings of the 12th Monterey conference on Reliable systems on unreliable networked platforms* (, Berlin, Heidelberg). Springer-Verlag, p. 1--25.

[50] Lee, Edward A.. The Problem with Threads. *Computer*, , May 2006, vol. 39, no. , p. 33--42.

[51] Yi-Bing Lin ; Bruno R. Preiss ; Wayne M. Loucks ; Edward D. Lazowska. Selecting the checkpoint interval in time warp simulation. *SIGSIM Simul. Dig.*, , 1993, vol. 23, no. 1, p. 3--10.

[52] Liu, C. L. ; Layland, James W.. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, , 1973, vol. 20, no. 1, p. 46--61.

[53] J. Liu ; E. A. Lee. Timed multitasking for real-time embedded software.. *IEEE Control Systems Magazine*, , 2003, vol. , no. , p. 65-75.

[54] Lubachevsky, Boris ; Schwartz, Adam ; Weiss, Alan. An analysis of rollback-based simulation. *ACM Trans. Model. Comput. Simul.*, , April 1991, vol. 1, no. , p. 154--193.

[55] Markowitz ; Harry Max ; J. Hausner ; H. W. Karr (1962). SIMSCRIPT: A Simulation Programming Language. , RM-3310-PR, .

[56] McLean, Thom ; Fujimoto, Richard (2003), Predictable Time Management for Real-Time Distributed Simulation. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation* (PADS '03, Washington, DC, USA). IEEE Computer Society, p. 89--.

[57] Sébastien Moutault ; Jacques Weber (2009). *Le langage SystemVerilog, Synthèse et vérification des circuits numériques complexes*. Dunod.

[58] Sébastien Moutault ; Jacques Weber ; Maurice Meaudre (2011). *Le langage VHDL, du langage au circuit, du circuit au langage*. Dunod.

[59] Nance, Richard E.. The time and state relationships in simulation modeling. *Commun. ACM*, , April 1981, vol. 24, no. , p. 173--179.

[60] Nance, Richard E. (1996). A history of discrete event simulation programming languages. , , 369--427.

[61] *Nao* (http://www.youtube.com/watch?v=QnQQG6mqrvU).

[62] *OpenModelica* (http://www.openmodelica.org/).

[63] Palaniswamy, Avinash C. ; Wilsey, Philip A.. An analytical comparison of periodic checkpointing and incremental state saving. *SIGSIM Simul. Dig.*, , July 1993, vol. 23, no. , p. 127--134.

[64] Perumalla, Kalyan S. (2006), Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 38th conference on Winter simulation* (WSC '06,). Winter Simulation Conference, p. 84--95.

[65] Pritsker, A. Alan B. (1971), The basics of GASP II: A tutorial. In *Proceedings of the 5th conference on Winter simulation* (WSC '71, New York, NY, USA). ACM, p. 405--408.

[66] *Ptolemy II* (http://ptolemy.eecs.berkeley.edu/ptolemyII/).

[67] Puschner,, Peter ; Burns,, Alan. Guest Editorial: A Review of Worst-Case Execution-TimeAnalysis. *Real-Time Syst.*, , 2000, vol. 18, no. 2-3, p. 115--128.

[68] Paul F. Reynolds, Jr. (1988), A spectrum of options for parallel simulation. In *WSC '88: Proceedings of the 20th conference on Winter simulation* (, New York, NY, USA). ACM, p. 325--332.

[69] R\"{o}nngren, Robert ; Ayani, Rassul. Adaptive checkpointing in Time Warp. *SIGSIM Simul. Dig.*, , July 1994, vol. 24, no. , p. 110--117.

[70] *Robot-Aspirateur Roomba* (http://www.robot-aspirateur.com/).

[71] *ROS nodelets* (http://www.ros.org/wiki/nodelet).

[72] *Le logiciel RTMaps* (http://www.intempora.com/ENG/maps/).

[73] Sha, Lui ; Abdelzaher, Tarek ; AArzén, Karl-Erik ; Cervin, Anton ; Baker, Theodore ; Burns, Alan ; Buttazzo, Giorgio ; Caccamo, Marco ; Lehoczky, John ; Mok, Aloysius K.. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Syst.*, , 2004, vol. 28, no. 2-3, p. 101--155.

[74] Simmonds, Rob ; Bradford, Russell ; Unger, Brian (2000), Applying parallel discrete event simulation to network emulation. In *Proceedings of the fourteenth workshop on Parallel and distributed simulation* (PADS '00, Washington, DC, USA). IEEE Computer Society, p. 15--22.

[75] *Simulink - Simulation et Model-Based Design* (http://www.mathworks.com/).

[76] Sokol, L.M. ; Weissman, J.B. ; Mutchler, P.A. (1991), MTW: an empirical performance study. In *Simulation Conference, 1991. Proceedings., Winter* (,). , p. 557 -563.

[77] Tapas K. Som ; Robert G. Sargent. A probabilistic event scheduling policy for optimistic parallel discrete event simulation. *SIGSIM Simul. Dig.*, , 1998, vol. 28, no. 1, p. 56--63.

[78] J. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. *Advances in Parallel and Distributed Simulation, SCS Simulation Series*, , January 1991, vol. 23, no. , p. 95-103.

[79] Steinman, Jeff S.. Interactive SPEEDES. *SIGSIM Simul. Dig.*, , April 1991, vol. 21, no. , p. 149--158.

[80] Steinman, Jeff S. (1994), Discrete-event simulation and the event horizon. In *PADS '94: Proceedings of the eighth workshop on Parallel and distributed simulation* (, New York, NY, USA). ACM, p. 39--49.

[81] Steinman, Jeffrey S. ; Lee, Craig A. ; Wilson, Linda F. ; Nicol, David M.. Global Virtual Time and distributed synchronization. *SIGSIM Simul. Dig.*, , July 1995, vol. 25, no. , p. 139--148.

[82] Steux, B. (2001). *RTMaps, un environnement logiciel dédié à la conception d'applications embarquées temps-réel. Utilisation pour la détection automatique de véhicules par fusion radar / vision.*. Mémoire, Ecole des Mines de Paris.

[83] Steux, B., El Hamzaoui, O. (2010), tinySLAM : a SLAM Algorithm in less than 200 lines of C code. In *International Conference on Control, Automation, Robotics and Vision (ICARCV)* (,). , p. .

[84] *SynDEx* (http://www-rocq.inria.fr/syndex/).

[85] Tang, Yarong ; Perumalla, Kalyan S. ; Fujimoto, Richard M. ; Karimabadi, Homa ; Driscoll, Jonathan ; Omelchenko, Yuri (2005), Optimistic Parallel Discrete Event Simulations of Physical Systems Using Reverse Computation. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation* (PADS '05, Washington, DC, USA). IEEE Computer Society, p. 26--35.

[86] Vaucher, Jean G. ; Duval, Pierre. A comparison of simulation event list algorithms. *Commun. ACM*, , April 1975, vol. 18, no. , p. 223--230.

[87] Villasenor, J., Mangione-Smith, W. H.. Configurable Computing. *Scientific American,*, , 1997, vol. June, no. , p. pp 66-71.

[88] *Watchdog* (http://fr.wikipedia.org/wiki/Chien_de_garde_%28informatique%29).

[89] West, D. (1988). *Optimizing Time Warp: Lazy rollback and lazy re-evaluation.*. Mémoire, University of Calgary.

[90] Wilhelm, Reinhard ; Engblom, Jakob ; Ermedahl, ;reas ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem ; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenstr\"{o}m, Per. The worst-case execution-time problem\—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, , May 2008, vol. 7, no. , p. 36:1--36:53.

[91] Xiao, Z. ; Unger, B. ; Simmonds, R. ; Cleary, J. (1999), Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation* (PADS '99, Washington, DC, USA). IEEE Computer Society, p. 20--28.

[92] Zhang, Jing Lei ; Tropper, Carl (2001), The dependence list in time warp. In *Proceedings of the fifteenth workshop on Parallel and distributed simulation* (PADS '01, Washington, DC, USA). IEEE Computer Society, p. 35--45.

[93] Y. Zhao ; E. A. Lee ; J. Liu. (2006). Programming temporally integrated distributed embedded systems.. , *UCB/EECS-2006-82*, .

[94] Zhao, Yang ; Liu, Jie ; Lee, Edward A. (2007), A Programming Model for Time-Synchronized Distributed Real-Time Systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium* (, Washington, DC, USA). IEEE Computer Society, p. 259--268.

Définition et implémentation d'un modèle causal d'exécution temps-réel distribuée

Résumé :

Ces travaux s'inscrivent dans le cadre du projet AROS (*Automotive Robust Operating Services*). Il a pour objectif de proposer un outil de prototypage rapide d'applications dynamiques distribuées temps-réel, principalement dans le domaine de l'automobile et de la robotique. Les applications distribuées temps-réel sont traditionnellement développées selon deux approches. La première, l'ordonnancement temporel, est basée sur l'analyse du pire temps d'exécution (*worst execution time*). Un partage du temps entre les différentes tâches de l'application est établi de façon statique. Cette technique offre une grande sureté de fonctionnement au prix d'une analyse temporelle parfois difficile à mener. La seconde, l'ordonnancement par priorité, est basée sur l'attribution à chaque tâche d'un niveau de priorité qui permet d'établir l'ordre d'exécution en fonction des évènements reçus par le système. Cette seconde technique, plus souple à mettre en œuvre, offre moins de garanties et conduit à un comportement non déterministe de l'application. La structure des applications AROS étant dynamique, l'approche temporelle est exclue car elle demande une analyse statique qu'il est impossible de produire. L'approche basée sur les priorités d'exécution est également exclue à cause de son non déterminisme comportemental. Nous proposons une approche basée sur un ordonnancement évènementiel causal inspirée des techniques d'ordonnancement des simulateurs évènementiels distribués. Tout en étant relativement simple à utiliser pour le concepteur d'application, cette technique produit des applications dont le comportement est parfaitement déterministe. Deux principales difficultés sont à surmonter : la synchronisation en temps-réel du moteur d'exécution et le respect des contraintes temps-réel.

Mots clés :

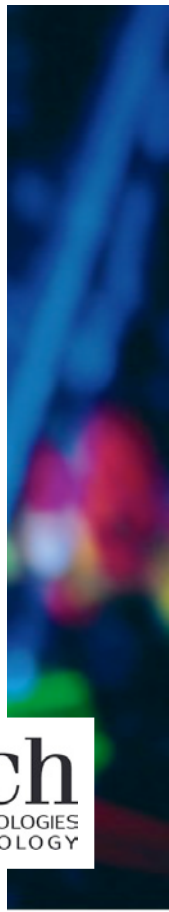
Application temps-réel, application distribution, causalité, architecture dynamique.

Abstract :

This work is part of the AROS project. Its goal is to define a fast prototyping tool for dynamic and distributed real-time applications, mostly for automotive industry and robotic. Two distinct methods are normally used to develop distributed real-time applications. The first one –the time triggered approach– is based on worst execution time analysis, whereby time sharing for the various tasks of an application is statically defined. This approach offers considerable safety but the time analysis is sometimes difficult to process. The second one –the priority scheduling approach– is based on ascribing a priority level to each task, which will then allow the system to define an execution order, based on the events is has received. This second approach is more flexible and easier to implement but is less safe and cannot ensure that the application behaves predictably. The structure of the AROS applications being dynamic, the time-triggered approach is irrelevant as it requires a static analysis that cannot be conducted. The priority scheduling approach is also irrelevant because of the non predictable behaviour. We propose an approach based on causal events scheduling inspired by distributed event simulators scheduling techniques. While comparatively easy to use for application designers, this new approach produces applications with a perfectly predictable behaviour. Two main obstacles must be overcome: the real time synchronisation of the execution engine and compliance with real-time constraints.

Keywords :

Real-time application, distributed application, causality, dynamic design.



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY