Architecture et Services des Intergiciels Temps Réel

Jerome Hugues

▶ To cite this version:

Jerome Hugues. Architecture et Services des Intergiciels Temps Réel. domain_other. Télécom Paris-Tech, 2005. English. NNT : . pastel-00001458

HAL Id: pastel-00001458
https://pastel.hal.science/pastel-00001458

Submitted on 15 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Doctorale d'Informatique, Télécommunications et Électronique de Paris

Thèse de doctorat de l'École Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

présentée par

Jérôme HUGUES

pour obtenir le grade de
Docteur de l'École Nationale Supérieure des Télécommunications

Architecture et Services des Intergiciels Temps Réel

soutenue publiquement le 27 Septembre 2005
devant le jury composé de

Bertil FOLLIO	Professeur à l'Université Paris VI	*président*
Laurence DUCHIEN	Professeur à l'Université des Sciences et Technologies de Lille	*rapporteur*
Juan A. DE LA PUENTE	Professeur à l'Université Polytechnique de Madrid	*rapporteur*
Serge HADDAD	Professeur à l'Université Paris Dauphine	*examinateur*
Philippe CHEVALLEY	Agence Spatiale Européenne	*invité*
Fabrice KORDON	Professeur à l'Université Paris VI	*directeur de thèse*
Laurent PAUTET	Maître de conférences à l'École Nationale Supérieure des Télécommunications, habilité à diriger les recherches	*directeur de thèse*

Il n'y a pas dans les livres des Anciens quelque chose de valeur que nous ne l'ayons mis dans ce livre. S'il ne se trouve pas à l'endroit où on a l'habitude de le mettre, je l'ai mis dans un autre endroit que j'ai estimé lui convenir davantage.

Avicenne

à Alice

Remerciements

Ce mémoire est le résultat de trois années d'efforts, de deux tours du monde, de plusieurs milliers de lignes de codes, de quelques billions de cycles CPU et de centaines de cafés. Il est aussi et surtout le fruit de ma rencontre avec de nombreuses personnes d'horizons divers qui m'auront beaucoup donné et appris. Je tiens ainsi à remercier tout particulièrement :

Fabrice KORDON et Laurent PAUTET pour avoir accepté de m'encadrer pour cette thèse, et pour la liberté qu'ils m'ont accordée lorsque j'ai voulu mener mes travaux vers des sentiers par trop inexplorés.

Laurence DUCHIEN et Juan Antonio DE LA PUENTE pour avoir accepté la charge de rapporteur de ce mémoire lors de la période estivale.

Bertil FOLLIOT, Serge HADDAD et Philippe CHEVALLEY pour l'intérêt porté à mes travaux, et leur participation au jury de thèse.

Michel RIGUIDEL, chef du département INFRES, et Isabelle DEMEURE, adjointe, pour la confiance qu'ils m'ont accordée en accueillant cette thèse puis en m'autorisant à participer à plusieurs cours et conférences.

Les membres de l'équipe ASTRE du département INFRES de l'ENST pour leur accueil, leurs conseils et les nombreux cafés échangés tout au long de ces trois années.

Thomas VERGNAUD et Khaled BARBARIA, doctorants à l'ENST, pour les nombreuses discussions et opinions sur les intergiciels, et leurs contributions depuis qu'ils ont rejoint le projet PolyORB.

Les membres du thème SRC au LIP6, dont Yann THIERRY-MIEG, Soheib BAARIR et Alexandre DURET-LUTZ pour leurs réponses à mes questions parfois naïves sur la vérification formelle et les réseaux de Petri, et pour avoir eu la patience d'effectuer toutes les corrections que je leur demandais.

Les membres de la compagnie AdaCore et plus particulièrement Thomas QUINOT pour ses nombreux conseils, et pour avoir réalisé la première version de PolyORB.

Vadim GODUNKO pour les nombreuses améliorations au projet PolyORB auxquelles il a contribué, et pour ses nombreux retours d'expérience qui nous ont permis d'améliorer notre architecture. спасибо Вадим !

Les élèves de l'École Nationale Supérieure des Télécommunications et de l'Université Paris VI — Pierre-et-Marie-Curie qui ont participé au projet PolyORB, et ont permis de concrétiser certaines des idées folles qui nous ont traversé l'esprit.

Les relecteurs et participants des conférences et ateliers où certains de ces travaux ont été présentés pour leurs nombreux commentaires.

Les relecteurs de ce mémoire pour leurs précieuses remarques qui ont contribué à sa qualité : Alice DE BIGAULT DE CASANOVE, Bertrand DUPOUY, Serge HADDAD, Philippe CHEVALLEY et Thomas QUINOT.

Je tiens à remercier aussi toutes les personnes qui auront rendu ces années d'intenses efforts plus agréables, fussent-ils visibles seulement par l'entremise du réseau :

drogués d'IRC, membres de forums, créateur de PhDComics, volleyeurs, compagnons rôlistes, amis de toujours. La liste est longue, et je demande pardon par avance à tout ceux que j'aurais oubliés.

Enfin, toutes mes pensées vont vers Alice, qui a su m'aider à traverser les derniers mois parfois chaotiques de la genèse de ce mémoire.

Résumé

L'utilisation d'intergiciels pour la réalisation de systèmes temps réel répartis embarqués (TR^2E) nécessite la preuve des propriétés de bon fonctionnement de l'intergiciel et une mesure de son impact sur la sémantique du système. Parallèlement, l'intergiciel doit être adaptable et répondre à de nombreuses contraintes (plate-forme d'exécution, ressources, interfaces de communication, ...).

Cette thèse propose une architecture adaptable d'intergiciel qui permet *1)* l'adaptabilité de ses fonctions élémentaires aux besoins de l'utilisateur, *2)* la vérification formelle des propriétés de l'intergiciel.

Cette architecture étend celle de l'intergiciel schizophrène. Elle fournit une séparation claire entre la boucle de contrôle et les services fonctionnels de l'intergiciel, facilitant le support de plusieurs politiques de concurrence et de gestions des ressources.

Nous avons validé notre proposition en vérifiant formellement les propriétés causales de deux configurations de l'intergiciel, en étendant les mécanismes de l'intergiciel pour proposer une implantation complète des spécifications RT-CORBA et DDS, ainsi qu'un guide pour le support d'autres classes d'intergiciels.

Enfin, nous proposons une analyse complète des performances, du déterminisme et de l'empreinte mémoire de configurations significatives de l'intergiciel.

Mots-clés : Répartition, Temps Réel, Embarqué, Vérification.

Abstract

Middleware for Distributed Real-time Embedded systems (DRE) must come with a complete analysis of its properties, the proof of its correct execution and an evaluation of its impact on the application's semantics. Besides, middleware must be versatile enough to support multiple constraints and requirements (execution platform, resources, communication, ...).

This thesis defines a middleware architecture supporting *1)* adaptability of its functions to meet user's requirements, *2)* formal verification of the middleware configuration to deploy.

This architecture extends the schizophrenic middleware architecture. It enforces a strict decoupling between control and functional elements, and helps the deployment of precise resource management and concurrency policies.

We validated our proposal through the formal verification of the causal properties of two configurations of the middleware and the construction of new middleware functions to support the RT-CORBA and DDS specifications. We also propose a guide to support other classes of middleware.

Finally, we analyze the performance, the determinism and the memory footprint of significant configurations of our architecture.

Keywords : Distribution, Real Time, Embedded, Verification.

Sommaire

1 Introduction générale **15**
1.1 Cadre d'étude 16
1.1.1 Définitions générales 16
1.1.2 Interaction temps réel/répartition/embarqué 19
1.1.3 Synthèse 22
1.2 Objectifs et contributions 24
1.3 Plan du mémoire 25

I Étude théorique et conception **27**

2 Problématique - État de l'Art **29**
2.1 Vers une architecture d'intergiciel dédiée 29
2.1.1 Besoins en mécanismes de répartition 30
2.1.2 Définitions générales 31
2.2 Taxonomie des intergiciels pour systèmes temps réel 32
2.2.1 Ingénierie des systèmes temps réel 32
2.2.2 Génie logiciel 35
2.2.3 Vérification et Validation 37
2.2.4 Analyse 40
2.3 Analyse des intergiciels adaptables existants 41
2.3.1 Intergiciels configurables 42
2.3.2 Intergiciels génériques 43
2.3.3 Intergiciels schizophrènes 46
2.3.4 Synthèse 50
2.4 Synthèse et axes de travail 51
2.4.1 Critique des architectures existantes 51
2.4.2 "Crise de l'Intergiciel" 51
2.4.3 Éléments de solution et méthodologie 52

3 Définition d'un intergiciel canonique **55**
3.1 Leçons tirées sur les architectures d'intergiciels 55
3.1.1 Reformulation des besoins 56
3.1.2 Vers une architecture généralisée d'intergiciels 57
3.2 Architecture générale pour les intergiciels 57
3.2.1 Patron de conception *"Broker"* 57
3.2.2 Utilisation de l'architecture schizophrène 59
3.3 Réduction des services d'un intergiciel 61

3.3.1 Services fonctionnels de l'intergiciel 61
3.3.2 Réduction des mécanismes de contrôle d'un intergiciel 64
3.3.3 Synthèse 65
3.4 Architecture du μBroker 66
3.4.1 Définition fonctionnelle du μBroker 66
3.4.2 Définition comportementale du μBroker 69
3.4.3 Réalisation du μBroker 73
3.4.4 Mise en œuvre du patron pour la construction d'intergiciels . 74
3.5 Conclusion 76

4 Vérification d'instances d'intergiciel 79
4.1 Motivations 79
4.2 Définition d'un processus de vérification 80
4.2.1 Formalismes pour la modélisation 80
4.2.2 Techniques de modélisation et vérification 81
4.3 Réseaux de Petri 82
4.3.1 Définitions générales 82
4.3.2 Analyse par model checking de l'espace d'états d'un réseau . 83
4.4 Vérification d'instances d'intergiciels 85
4.4.1 Processus de modélisation 85
4.4.2 Un modèle particulier : lecture des sources 86
4.4.3 Configurations du μBroker et modèles 86
4.4.4 Méthodes d'analyse 89
4.4.5 Résultats 92
4.5 Conclusion 95

II Mise en œuvre et expérimentations 97

5 Briques logicielles pour intergiciel temps réel 99
5.1 Langages de programmation pour le temps réel 99
5.1.1 Choix d'Ada 95 100
5.1.2 Fonctionnalités du langage Ada 95 pour le temps réel 100
5.2 Structures de données pour intergiciels temps réel 103
5.2.1 Suppression des exceptions 103
5.2.2 Table de hachage dynamique parfaite 105
5.2.3 Gestion de la QdS 108
5.2.4 Gestion de la priorité 108
5.2.5 Files d'exécution 110
5.2.6 Ordonnanceur de requêtes 113
5.2.7 Conclusion 113
5.3 Services d'intergiciel TR^2E 113
5.3.1 Adaptateur d'objets déterministe 114
5.3.2 Protocole asynchrone 115
5.3.3 Couche transport temps réel 117
5.4 Conclusion 122

6 Construction d'intergiciels pour systèmes TR^2E 125
6.1 Mode opératoire . . . 125
6.1.1 Composants de PolyORB . . . 126
6.1.2 Assemblage d'une configuration . . . 129
6.2 Construction d'un intergiciel minimal . . . 131
6.2.1 Définition . . . 131
6.2.2 Analyse du nœud construit . . . 132
6.3 Construction d'un intergiciel RT-CORBA . . . 133
6.3.1 Définition . . . 133
6.3.2 Dépendances sur l'exécutif . . . 135
6.3.3 Critiques de la spécification RT-CORBA . . . 135
6.3.4 Conclusion . . . 138
6.4 Prototypes . . . 138
6.4.1 RT-DSA . . . 138
6.4.2 DDS . . . 140
6.5 Conclusion . . . 141

7 Mesures 143
7.1 Bancs de mesure . . . 144
7.1.1 ORK . . . 144
7.1.2 MaRTE OS . . . 144
7.1.3 Sun Solaris . . . 144
7.2 Analyse du comportement temporel . . . 145
7.2.1 Définition d'un protocole de mesure . . . 145
7.2.2 Réalisation de la plate-forme de mesure . . . 148
7.2.3 Mesures sur Solaris . . . 148
7.2.4 Mesures sur un noyau temps réel . . . 152
7.2.5 Analyse critique . . . 153
7.3 Empreinte mémoire . . . 154
7.3.1 Mesures . . . 154
7.3.2 Optimisations . . . 155
7.3.3 Conclusion . . . 157
7.4 Conclusion . . . 157

8 Conclusion et perspectives 159
8.1 Intergiciels pour systèmes TR^2E . . . 159
8.2 Réalisations . . . 160
8.3 Perspectives . . . 161

Index 164

Bibliographie 165

Table des figures

1.1 Oppositions entre temps réel, répartition et embarqué 19
1.2 Système Contrôle/Commande 22
1.3 Système aéronautique 22
1.4 Construction d'un système TR^2E 23

2.1 Interactions entre les personnalités d'un intergiciel schizophrène . . . 48
2.2 Services d'un intergiciel schizophrène 48
2.3 Solution apportée par l'intergiciel aux systèmes TR^2E 52

3.1 Patron de conception *"Broker"* 58
3.2 Place du module μBroker 66
3.3 Intergiciel canonique 67
3.4 Envoi d'une requête à une personnalité protocolaire 69
3.5 Envoi d'une requête à une personnalité applicative 70
3.6 Réception d'une requête par une personnalité protocolaire 70
3.7 Schéma de fonctionnement du patron Half Sync/Half Async 71
3.8 Schéma de fonctionnement du patron Leader/Followers 72
3.9 Séparation des aspects du μBroker 73
3.10 Architecture générale du μBroker 74

4.1 C/S : Réseau de Petri d'un système client/serveur 83
4.2 L'espace d'états concrets associé à C/S 84
4.3 L'espace d'états symboliques associé à C/S 84
4.4 Étapes de modélisation 87
4.5 Un modèle de la bibliothèque 88
4.6 Espace d'états et espace quotient du μBroker pour $S_{max}= 4$ et $B_{size}= 5$ 93
4.7 Nombre de nœuds examinés pour évaluer $P3$, $T_{max}= 2$ et $B_{size}= 4$ . . 94

5.1 Comportement temporel des tables de hachages dynamiques parfaites 108
5.2 Automate d'états d'une tâche associée à une file d'exécution 111
5.3 Vue de la pile protocolaire MIOP côté serveur 116
5.4 Cycle élémentaire du protocole TDMA 118
5.5 Fonctionnement du protocole TDMA 119
5.6 Mesures et dispersion pour `slot_duration` = 2 ms 121
5.7 Mesures et dispersions pour `slot_duration` = 400 μs 121

6.1 Construction de deux intergiciels distincts 126
6.2 Construction d'un nœud PolyORB 130
6.3 Construction d'un nœud PolyORB minimal 131

6.4 Configuration d’un nœud PolyORB RT-CORBA 134
6.5 Configuration d’un nœud PolyORB RT-DSA 139
6.6 Configuration d’un nœud PolyORB DDS 141

7.1 Distributed Hardstone Benchmark original 145
7.2 Distributed Hardstone Benchmark avec intergiciel 146
7.3 Mesures sur la plate-forme Solaris, cas d’un appel/réponse. (valeurs en *s*) 149
7.4 Mesures sur la plate-forme Solaris, cas d’un appel ONEWAY. (valeurs en *s*) 150
7.5 Mesures de performance sous Solaris 151
7.6 Mesures de la bande passante de l’intergiciel. 152
7.7 Mesures sur la plate-forme ORK. (valeurs en *ns*) 152

1

Introduction générale

Sommaire

1.1 Cadre d'étude **16**
1.1.1 Définitions générales 16
1.1.2 Interaction temps réel/répartition/embarqué 19
1.1.3 Synthèse 22
1.2 Objectifs et contributions **24**
1.3 Plan du mémoire **25**

"2) Latency is zero."
The Eight Fallacies of Distributed Computing,
— Peter DEUTSCH, SUN MICROSYSTEMS.

La généralisation des applications réparties, conjointement à la disponibilité d'infrastructures de répartition - *intergiciels* - a radicalement modifié la façon dont les informations sont maintenant échangées et traitées.

Un intergiciel fournit un cadre général pour la construction d'applications réparties ; il permet de se concentrer sur l'application et de s'affranchir de certains des problèmes liés à la répartition. Les intergiciels font maintenant partie intégrante des outils de l'"Ingénierie des Logiciels".

Les intergiciels fournissent une solution aux besoins variés d'applications réparties telles que les serveurs d'applications, les services Web ou les applications multimédia imposant des contraintes de sécurité ou de qualité de service. Dans ce contexte, ils proposent une solution aux problèmes classiques d'algorithmique et d'ingénierie des applications réparties : ils mettent en œuvre de manière transparente du point de vue des utilisateurs les mécanismes permettant à l'application de fonctionner.

Parallèlement, les applications temps réel embarquées s'approprient peu à peu les techniques de la répartition en vue de les appliquer à des systèmes fortement contraints en termes de ressources mais aussi de déterminisme.

Les applications *"Temps Réel Réparti Embarqué"* doivent pouvoir connaître précisément l'impact sur les performances du système induit par chacun des choix réalisés, mais aussi calculer l'utilisation des ressources faite par le système pour éviter sa saturation. Ce besoin est des plus critiques pour les applications réparties temps réel et embarquées que nous considérerons dans ce mémoire : une erreur de conception, ou une hypothèse abusive - "la latence est négligeable" - suffit à compromettre l'intégrité de l'application.

La classe des applications “Temps Réel Réparti Embarqué” (TR^2E) pose une variété de problèmes qui ne sont que partiellement résolus. Elle se caractérise par des contraintes fortes en terme de conception et de réalisation, et demande des garanties : déterminisme, respect des échéances temporelles, support d’exécutif restreint, etc. Ceci inclut en particulier l’intergiciel.

L’intergiciel doit s’intégrer à l’application sans remettre en cause son fonctionnement correct. Il a un rôle crucial vis à vis de l’application : il contrôle l’échange de données entre les différents nœuds. Il a donc un impact significatif sur l’application. La conception et la réalisation d’intergiciels pour systèmes TR^2E doit permettre à l’utilisateur de mieux comprendre et analyser chacun de ses choix lors de la configuration et le déploiement de l’intergiciel.

Nous notons ainsi que l’utilisation d’un intergiciel dans un contexte TR^2E induit une dimension supplémentaire à la construction d’intergiciels : la nécessaire compréhension des mécanismes internes de l’intergiciel. Celle-ci permet d’aboutir *1)* à la sélection et la configuration du sous-ensemble de mécanismes répondant aux besoins de l’application et *2)* à la garantie que les besoins sont correctement remplis. L’“ingénierie des intergiciels” constitue une nouvelle discipline nécessaire à l’analyse et à la réalisation des applications réparties.

Le présent mémoire vise à fournir un canevas pour la conception et la réalisation d’intergiciel pour applications TR^2E. Notre approche cherche non pas à définir un nouvel intergiciel, mais à proposer :

1. un ensemble de guides méthodologiques permettant la construction raisonnée d’intergiciels. Ce guide doit permettre d’exprimer une solution aux besoins des applications ;
2. un processus de vérification et de validation des éléments proposés vis à vis des contraintes définies par l’application et son environnement d’exécution.

Dans ce chapitre, nous définissons les éléments formant notre cadre d’étude. Nous posons ensuite les objectifs que nous souhaitons voir remplis, et les contributions que nous proposons pour ce faire. Enfin, nous introduisons le plan du mémoire.

1.1 Cadre d’étude

La répartition, l’embarqué et le temps réel définissent trois classes de systèmes distinctes. Elles fournissent chacune des éléments pour la construction de systèmes informatiques complexes, elles imposent aussi un ensemble de contraintes.

La composition de ces trois classes doit répondre aux besoins en systèmes simultanément temps réel, répartis, embarqués (TR^2E). Nous présentons dans un premier temps chacune de ces classes, nous présentons ensuite les contraintes et les limites lors de leurs combinaisons.

1.1.1 Définitions générales

Dans cette section, nous définissons chacune de ces classes : nous introduisons les supports d’exécution sur lesquels elles reposent, ainsi qu’une liste non exhaustive de contraintes qu’elles imposent lors de la conception d’applications.

Systèmes répartis

Définition 1.1.1. *Un système réparti est une fédération d'ordinateurs communiquant et coopérant pour réaliser une tâche. On peut citer les applications de réservation en ligne ou une constellation de satellites. Les systèmes répartis utilisent des mécanismes permettant à plusieurs processus applicatifs, se trouvant sur des calculateurs différents (les* "nœuds" *du système) d'échanger et de faire évoluer une information.*

Les intergiciels (*"middleware"*) ont été introduits progressivement en réponse aux besoins en répartition ([Vinoski, 2004] fournit à ce sujet un historique détaillé). Ils fournissent des mécanismes et services pour la mise en œuvre d'un système réparti. Ainsi défini, un intergiciel est une couche de médiation entre l'application et les primitives de base proposées par le système d'exploitation ou la machine virtuelle qui sert de support à l'exécution de l'application.

Un intergiciel propose une abstraction masquant les détails de la répartition : le modèle de répartition. Ce modèle définit les différents entités et services fournis à l'utilisateur : gestion des connections, nommage des entités, etc. On peut citer le modèle objets répartis mis en avant par CORBA ou le modèle basé sur les appels de procédure distante. Cette abstraction est concrétisée par une interface et une bibliothèque dédiées (cas de l'interface JMS), ou un générateur de code (cas de CORBA).

Les services fournis par un intergiciel ne sauraient être figés et dépendent pour beaucoup du niveau d'abstraction du modèle qu'il fournit et de la valeur ajoutée apportée par l'implantation retenue [Bernstein, 1993].

Nous retenons que les intergiciels doivent faire face à deux contraintes fortes, liées à l'hétérogénéité des applications réparties et de leur support d'exécution :

opacité : un intergiciel doit masquer les hétérogénéités entre les nœuds de l'application telles que les différences d'architecture (architecture matérielle, système d'exploitation) ; mais aussi assurer l'échange de données en proposant une représentation des données cohérente.

généralité : un intergiciel doit fournir les services utiles pour construire une application. Ces services vont des plus basiques (cycle de vie des connections, enregistrement de nouveaux nœuds) aux plus avancés (terminaison, tolérance aux fautes). Ces services doivent être suffisamment généraux pour être adaptés à une large famille d'applications. Cependant, ils ajoutent aussi à la complexité de conception et d'utilisation de l'intergiciel.

Par ailleurs, des contraintes économiques sur les fournisseurs d'intergiciels font que ces derniers tendent à fournir une solution générale, applicable à une large famille d'applications, plutôt qu'une solution dédiée aux besoins spécifiques d'une application.

Systèmes temps réel

Définition 1.1.2. *Un système temps réel est tel que son fonctionnement correct dépend autant du traitement réalisé, que de la date à laquelle ce traitement est terminé. Cette classe de systèmes regroupe les applications incluant la notion de temps, en relation avec les stimuli d'un système physique, par exemple la mesure d'un capteur, le traitement d'un flot continu d'informations.*

Un système temps réel doit être conçu et réalisé de façon à garantir ses échéances temporelles. Cela nécessite un processus de développement précis, intégrant la notion

de performance des actions effectuées, de déterminisme des fonctions de base utilisées ainsi qu'un processus de validation ou de vérification garantissant que la solution formulée répond aux attentes.

La concurrence entre différents traitements amène à intégrer la notion de priorité et de criticité d'une action. On parlera d'échéance *dure* si la réalisation d'une action dans les temps est vitale pour l'application, et *lâche* dans le cas contraire [Demeure & Bonnet, 1999].

Différents algorithmes d'ordonnancement peuvent être implantés de façon à traiter les actions suivant leur criticité et leur échéance temporelle. Ils sont le plus souvent implantés par l'exécutif. Ces algorithmes définissent un ensemble de conditions suffisantes ou nécessaires garantissant l'exécution correcte du système.

Notons qu'il existe une différence importante entre les résultats théoriques d'ordonnancement qui proposent de nombreuses politiques évoluées, et ce qui est effectivement mis en œuvre par les noyaux temps réel usuels. Ainsi, peu de systèmes proposent d'algorithme d'ordonnancement autre qu'un modèle basé sur les priorités statiques : cette politique autorise un ordonnancement statique des actions grâce à l'analyse harmonique du système (RMA) [Liu & Layland, 1973] et rend possible une analyse complète du système. Au contraire, les autres classes d'algorithmes rendent les systèmes dynamiques, et donc plus difficilement analysables.

De fait, les systèmes temps réels nécessitent :

un référentiel : le système doit pouvoir être décrit en référence à un modèle précis, permettant de l'analyser à des fins de validation/vérification. Ce modèle pose des conditions suffisantes ou nécessaires garantissant le déterminisme et l'exactitude temporelle du système ;

des restrictions : l'utilisation d'un référentiel nécessite la mise en place de restrictions sur les constructions utilisables par un système. Ces restrictions permettent de garantir que le système se comporte conformément au référentiel choisi. On peut citer comme restrictions classiques l'interdiction d'allocation dynamique d'objets ou des limitations sur les primitives de concurrence utilisées. Des spécifications telles que DO-178B [SC-167, 1992] ou le profil Ravenscar [Dobbing & Burns, 1998] motivent et fournissent une telle liste.

Systèmes embarqués

Définition 1.1.3. *Un système embarqué est un sous-système dédié à une tâche. Un système embarqué est spécifique, et ne peut réaliser qu'un ensemble bien défini de tâches.*

Les spécificités des systèmes embarqués sont telles que leur conception et leur réalisation nécessitent l'emploi d'outils et de chaînes de production dédiés permettant ainsi la prise en compte de faibles ressources, d'interfaces dédiées, etc. Un noyau réduit tient lieu de système d'exploitation minimal et sert de support d'exécution pour l'application. Bien souvent, ce noyau est configurable et adaptable de façon à n'inclure que les fonctions strictement nécessaires, et prendre en compte la diversité des plates-formes cibles (processeur, entrées/sorties, etc).

Notons qu'il n'existe pas de système embarqué canonique. Un système embarqué peut aussi bien être un ordinateur léger de type PDA qu'un calculateur de bord d'un avion, voire un système de contrôle radar comportant de multiples processeurs et faiblement limité en mémoire.

Les noyaux pour systèmes embarqués doivent répondre à des besoins particuliers :

adaptabilité : ces noyaux doivent être adaptables pour supporter différents types de matériels, y compris des interfaces et des processeurs dédiés ;

faibles exigences : du fait de leur cycle de vie et des matériels utilisés, ces noyaux doivent limiter fortement leur besoin en ressources : empreinte mémoire, utilisation intelligente du processeur et de l'énergie de façon à prolonger la durée de vie du système ou encore répondre à des contraintes économiques ;

fiabilité : ces systèmes doivent pouvoir fonctionner sans défaillance sur une période donnée, ou pouvoir fonctionner en mode dégradé en présence de composants défaillants. Le degré de tolérance aux pannes est définie par l'architecture du système.

1.1.2 Interaction temps réel/répartition/embarqué

Nous avons fourni une définition des systèmes temps réel, des systèmes répartis et des systèmes embarqués. Comme nous l'avons montré, chaque classe de système vient avec ses propres problèmes à résoudre. Ils ont donné lieu à de multiples solutions, chacune pertinente pour son domaine d'étude.

Dans cette section, nous nous intéressons dans un premier temps à la construction de systèmes simultanément temps réel, répartis et embarqués. En particulier, nous mettons en évidence les difficultés conceptuelles et techniques sous-jacentes : chaque classe de systèmes introduit des contraintes contradictoires qu'il faut réconcilier.

Nous introduisons les différents problèmes pouvant se poser. La figure 1.1 résume la question de la définition d'un couple intergiciel/noyau temps réel capable de résoudre ces antagonismes. Par la suite, nous dressons un panorama des besoins exprimés par quelques applications TR^2E.

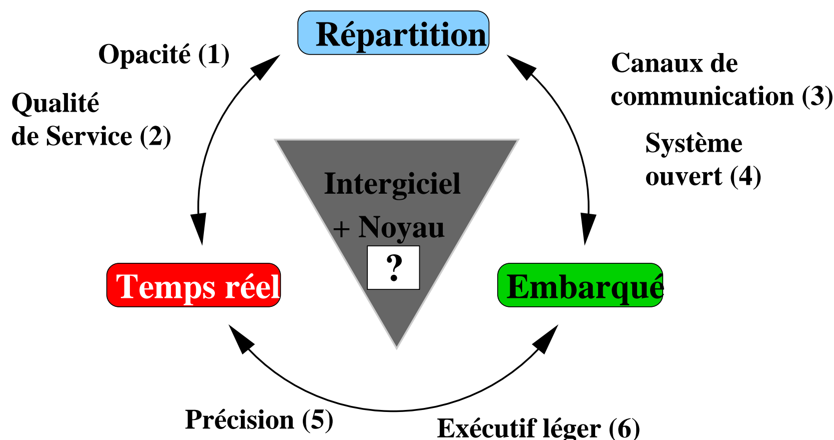


FIG. 1.1 – Oppositions entre temps réel, répartition et embarqué

Temps réel et répartition

Opacité (1) : La répartition offre à l'utilisateur un cadre descriptif général pour construire son application, en se basant par exemple sur le modèle objet de CORBA [OMG, 2004b] ou les interfaces de Java RMI [Sun Microsystems, Inc., 2002]. Ces modèles offrent une large variété de services qui permettent au développeur de

construire et déployer son application. Il se concentre sur l'application à écrire, les détails d'implantation ainsi que la plate-forme d'exécution sont masqués. Il est alors difficile d'extraire des propriétés de l'application.

Au contraire, un système temps réel doit faire la preuve qu'il respectera des propriétés précises (par exemple temporelles). Ceci nécessite une maîtrise complète du processus de fabrication de l'application.

En particulier, chaque composant doit détailler ses caractéristiques ou pouvoir être analysé afin de connaître sa durée d'exécution sur le système cible, son empreinte mémoire, etc. L'opacité introduite par les intergiciels est un frein à l'analyse du système.

Qualité de Service (2) : Généralement, un système temps réel associe une méta-information à chaque travail à effectuer. Cette information permet de contrôler le traitement en fonction de critères d'importance telles que la priorité ou la criticité. Cette information, conjointement à l'utilisation d'un algorithme d'ordonnancement, et sous réserve de respecter une ou plusieurs conditions de faisabilité, permet d'assurer l'échéance temporelle de chaque action.

Cela impose à l'intergiciel la prise en compte d'informations de contrôle via l'adjonction de paramètres de Qualité de Service (*"QoS"*[1]) et de politiques dédiées pour chaque paramètre retenu. Nous exhibons ainsi des problématiques de la répartition, en rendant visible des aspects de configuration du comportement de l'intergiciel.

Nous remarquons par ailleurs que la notion de QdS temporelle est fortement dépendante du point de vue considéré. Il est possible localement de garantir ces propriétés en se basant sur des techniques d'analyse classiques. Il est en revanche bien plus difficile d'aboutir aux mêmes garanties lors d'une communication entre entités réparties. La latence du canal de communication peut être très importante et supérieure à celle relevée sur le nœud. Par la suite, nous nous intéressons au cas du déterminisme à l'échelle d'un nœud ; la construction de systèmes temps réels répartis pour lequel le délai de communication de bout en bout est complètement déterministe sort du cadre de nos travaux.

Répartition et embarqué

Canaux de communication (3) : Les systèmes embarqués sont prévus pour fonctionner dans un contexte précis. En particulier, les interactions avec d'autres entités, y compris les entités extérieures au système, doivent être contrôlées : les ressources nécessaires au traitement des interactions doivent être correctement dimensionnées. Par ailleurs, les communications entre les différents nœuds du système, ou à l'intérieur d'un des nœuds, se fait à l'aide de canaux spécifiques (bus VME, bus militaire MIL-STD-1553, bus CAN (*"Controller Area Network"*) [Bosch, 1991], etc.).

La prise en compte de ces protocoles par l'intergiciel nécessite un travail d'intégration. Par exemple, le bus CAN impose des contraintes particulières sur le débit et la taille des messages échangés. Son intégration à un intergiciel basé sur CORBA [Kim *et al.*, 2000] aura posé de nombreux problèmes de conception et imposé l'adaptation du format des messages échangés.

Système ouvert (4) : Un intergiciel est le plus souvent vu comme un système ouvert, capable d'initier de nouvelles communications à tout instant pour le compte de l'application qu'il héberge. Ce caractère dynamique fait qu'il est alors délicat de dimensionner correctement les ressources nécessaires : descripteurs d'entrée/sortie, tampons

[1] traduit par QdS dans la suite de ce mémoire

mémoire. Il faut donc mettre en œuvre des mécanismes de contrôle pour éviter tout dépassement de capacité.

Temps réel et embarqué

Exécutif restreint et précis (5) : Les systèmes temps réel nécessitent que tous les éléments utilisés soient déterministes. En particulier, de nombreuses restrictions peuvent être portées sur l'exécutif utilisé (système d'exploitation, exécutif du langage, machine virtuelle, etc).

Ainsi, les profils Ravenscar [Dobbing & Burns, 1998] ou encore MISRA-C [MISRA, 2004] limitent les constructions du langage admissibles à celles permettant de mener une analyse poussée des propriétés du système. Ceci impose des contraintes fortes sur l'exécutif. En particulier, les propriétés de bon fonctionnement à l'exécution définies par ces profils doivent être présentes dans l'implantation réalisée.

Exécutif léger (6) : La prise en compte des ressources des systèmes embarqués amène à réduire les fonctionnalités disponibles, par exemple l'allocation dynamique de mémoire, le nombre de processus légers, la disponibilité des exceptions. Cela aura un impact sur l'application construite. En particulier, certains algorithmes de contrôle évolués tels que l'ordonnancement EDF (*"Earliest Deadline First"*) ne pourront être implantés.

Hétérogénéité des besoins

Nous avons répertorié quelques problèmes spécifiques à la construction d'applications TR^2E. Nous n'avons considéré que la définition d'une plate-forme à même de répondre à des besoins génériques et présenté quelques problèmes issus de la combinaison des contraintes des systèmes temps réel, répartis et embarqués. Dans cette section, nous nous intéressons aux besoins d'applications "réelles".

Nous remarquons qu'il n'existe pas d'application TR^2E "canonique" qui puisse servir de référence. Au contraire, chaque application définit un ensemble de besoins très spécifiques, augmentant d'autant le nombre de besoins à satisfaire. Nous présentons ici deux exemples d'applications TR^2E et indiquons les besoins non-fonctionnels s'ajoutant à ceux liés à la répartition, l'embarqué ou le temps réel.

- **Système "Contrôle/Commande"**
  Les systèmes contrôle/commande (figure 1.2) sont présents à différents niveaux dans l'industrie, notamment dans les domaines automobile et aéronautique.
  Un ensemble de capteurs envoie un flux de données à un calculateur. Ce calculateur utilise ces données et envoie une nouvelle consigne à un ou plusieurs actuateurs et afficheurs. Les systèmes actuels utilisent de nombreux calculateurs (de quelques dizaines pour une voiture à plusieurs centaines pour l'Airbus A380), mis en réseau par un bus industriel.
  Le support d'exécution de ces systèmes inclut des mécanismes de répartition. Ces systèmes se caractérisent par le lien fort entre les phénomènes physiques mesurés par les capteurs et l'exécution du système. Cela pose des contraintes temporelles fortes sur le système. De plus, les exécutifs doivent privilégier la sûreté de fonctionnement de façon à résister à une défaillance d'un ou plusieurs capteurs et actuateurs.
- **Flotte de Drônes et Satellites**
  Le projet européen ASSERT[2] s'intéresse aux technologies de conception de sys-

[2] ASSERT fait partie du sixième programme de l'union européenne, `http://www.assert-online.net`

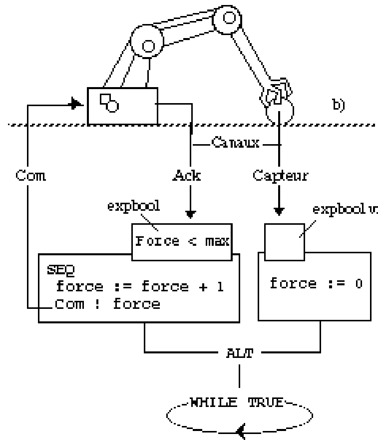


FIG. 1.2 – Système Contrôle/Commande

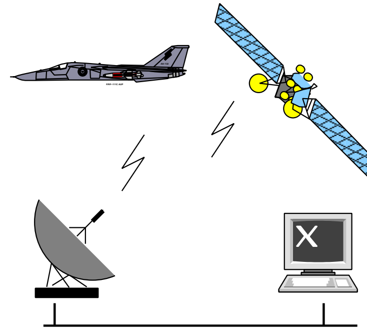


FIG. 1.3 – Système aéronautique

tèmes complexes à haut niveau telles que la définition de familles de systèmes, l'emploi de méthodes de conception assurant les propriétés du système à chaque étape de sa conception et la réutilisation de blocs logiciels ou matériels qui peuvent ensuite être composés.
Le projet ASSERT définit des besoins particuliers pour un intergiciel qui puisse être intégré dans un processus de conception d'applications qui assure et préserve les propriétés du système. L'Agence spatiale européenne (ESA) définit un scénario d'utilisation pour un tel intergiciel (figure 1.3) : des stations au sol et des flottes de satellites ou drônes collaborent pendant une mission d'observation. Cet intergiciel doit pouvoir fournir de multiples mécanismes de répartition, gérer les modifications des canaux de communication (perturbations et reconfiguration dynamique), être tolérant aux pannes, maximiser l'utilisation de ses ressources et être autonome pour les missions de longue durée.

1.1.3 Synthèse

Nous avons mentionné plusieurs aspects de la construction d'un système Temps Réel Réparti Embarqué (TR^2E). En première approche, on peut voir que ces aspects sont étroitement liés, et nécessitent donc une compréhension fine de chacun de ces domaines. Néanmoins, nous faisons la remarque que ces trois classes de systèmes posent des contraintes à des niveaux de conception différents.

Impact sur l'architecture de l'intergiciel

Si on se donne comme problème la construction d'un système TR^2E, alors nous pouvons faire les constatations suivantes :
– Les systèmes répartis requièrent un niveau d'abstraction suffisant pour représenter la sémantique de l'application désirée. Les intergiciels fournissent un ensemble de fonctions et services supportant un modèle de répartition.
– La garantie de propriétés non-fonctionnelles du système peut nécessiter l'adjonction de services additionnels tels que réplication pour la tolérance aux pannes, un service d'ordonnancement réparti, etc.

– Les systèmes temps réel et embarqués imposent des contraintes sur ce qui peut être implanté du fait des restrictions imposées par le référentiel utilisé ; mais aussi du fait des ressources disponibles pour la cible utilisée.
– Les systèmes temps réel et embarqués nécessitent que le système puisse garantir certaines de ces propriétés. Ceci ne peut se concevoir qu'en phase amont de conception, par construction ou vérification ; ou alors sur le système final, par validation.

Ces différentes contraintes vont peser à des niveaux différents sur l'architecture de l'intergiciel pour applications TR^2E.

Ainsi, les besoins en répartition vont être satisfaits par des fonctions de haut niveau, tandis que les contraintes temps réel et de l'embarqué vont être diffuses à l'échelle de l'intergiciel. Elles nécessitent que chaque composant puisse, isolément, fournir la preuve de ses propriétés avant d'être inclus. La combinaison des composants élémentaires doit à son tour répondre aux besoins initiaux en déterminisme et ressources.

Il faut donc disposer d'une architecture d'intergiciel facilitant l'analyse de ses fonctions élémentaires, de leurs compositions et de l'assemblage final.

Par ailleurs, la prise en compte des besoins de l'application amènera à une série de compromis, impliquant de relâcher certaines contraintes (par exemple sur l'empreinte mémoire ou la latence des opérations d'entrées/sorties) pour fournir des mécanismes de répartition évolués. Ce compromis doit dans la mesure du possible être issu d'un choix de l'utilisateur de l'intergiciel et non subi.

Conception de systèmes TR^2E

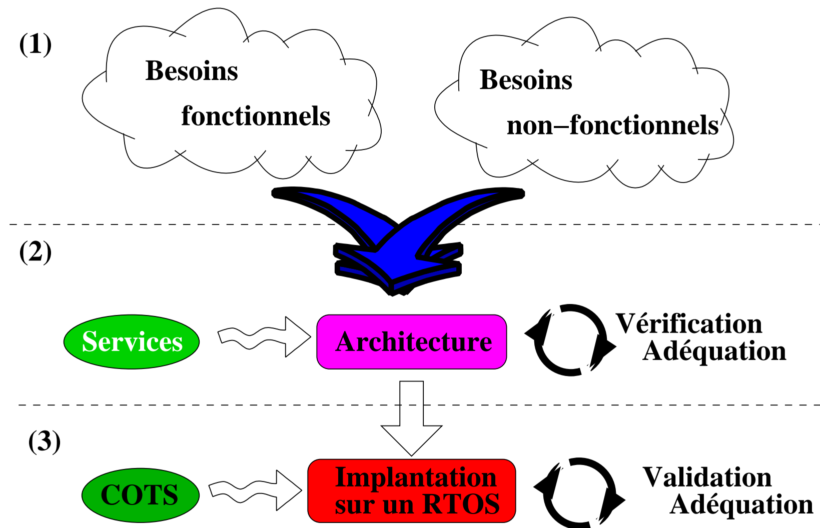


FIG. 1.4 – Construction d'un système TR^2E

Nous notons trois grandes étapes dans la conception d'un système TR^2E. Ce découpage permet de structurer sa conception et sa réalisation, et offre un premier découplage entre les services nécessaires et les propriétés à garantir (figure 1.4).

– Dans une première phase de *"définition" (1)*, les besoins fonctionnels et non-fonctionnels du système sont énoncés. Ils fournissent une description à haut niveau de l'application à construire.

– Dans une seconde phase de *"configuration" (2)*, on dérive de ces besoins une architecture candidate, exprimée à l'aide d'un ou plusieurs services de l'intergiciel et du noyau utilisés ; un travail de vérification permettra de s'assurer de son adéquation à une première famille de propriétés issues des besoins exprimés.
– Cette architecture pourra dans une dernière phase, 'de *'déploiement d'une solution" (3)*, être implantée sur un système d'exploitation temps réel et adapté à la cible embarquée prévue. Elle est ensuite déployée sur les nœuds de l'application. Des composants logiciels réutilisables (*Components-Off-The-Shelf, "COTS"*) seront sélectionnés pour l'implantation des services retenus. À nouveau, un travail de validation ou de vérification permettra d'assurer que la solution proposée est correcte.

Ce canevas générique fournit une vue des étapes clés du processus de construction d'un système TR^2E, utilisant un intergiciel adapté. Il s'agit d'une vue superficielle, qui doit être affinée suivant un processus méthodologique, par exemple imposé par le domaine industriel considéré.

1.2 Objectifs et contributions

L'objectif de cette thèse est de définir une architecture modulaire d'intergiciel permettant la construction d'applications temps réel réparties embarquées suivant le schéma décrit à la section précédente.

Notre approche consiste à reconnaître puis réutiliser les "bonnes pratiques" isolées et définies par différents projets existants. Ce travail nécessaire nous permet de caractériser un intergiciel pour les applications TR^2E.

Nous prolongeons cette étude en proposant une démarche incrémentale permettant de construire un tel intergiciel. Partant d'une base de "briques intergicielles" minimale, ce processus permet l'ajout des éléments requis par l'application, et la vérification ou la validation des propriétés de l'intergiciel.

Nous proposons plusieurs contributions à la construction d'intergiciels pour systèmes TR^2E :

1. Nous avons défini un noyau d'intergiciel minimal bâti autour du module μBroker et des services proposés par l'architecture d'intergiciel schizophrène. Ce module a une architecture modulaire supportant de nombreuses politiques de configuration, et implante les patrons relevés lors de notre état de l'art ;
2. Nous avons montré que ce noyau peut être utilisé dans une architecture plus large pour construire de nombreux intergiciels, supportant plusieurs modèles de répartition (objets répartis, RPC, passage de messages, etc) ;
3. Nous avons proposé des éléments de méthodologie permettant d'adapter notre architecture à différents besoins : configuration d'intergiciel minimal, construction d'intergiciels complets pour applications temps réel ;
4. Nous avons extrait un modèle complet de l'intergiciel, et proposé des structures de données déterministes permettent l'analyse de notre solution ;
5. Nous avons montré que cette architecture est vérifiable, et nous avons vérifié formellement plusieurs de ses propriétés pour des configurations significatives de l'intergiciel ;
6. Nous avons déduit de cette architecture une implantation qui est compatible avec les méthodes de développement de systèmes temps réel : utilisation du langage Ada 95, du profil Ravenscar, de structures de données déterministes ;

7. Notre architecture et son implantation ont été validées sur plusieurs exemples : RT-CORBA, protocole temps réel et intergiciel minimal pour les systèmes embarqués ;
8. Plusieurs tests ont démontré le déterminisme de la solution proposée. L'empreinte mémoire des nœuds peut être rendue faible. Nous avons par ailleurs discuté des optimisations pour réduire d'avantage la consommation de la mémoire.

1.3 Plan du mémoire

La suite de ce mémoire est organisée comme suit :

Dans la première partie, *"Étude théorique et conception"*, nous étudions les architectures des intergiciels pour systèmes TR^2E et indiquons quelques unes de leurs limites. Nous motivons et proposons une architecture d'intergiciel nouvelle à même de lever ces limites.

Au chapitre 2, nous analysons les solutions existantes. Celles-ci se répartissent en trois familles, chacune répondant à un besoin précis de ces systèmes : ingénierie des systèmes temps réel, adaptabilité, construction à partir de modèles. Cependant, ces intergiciels ne fournissent qu'une solution partielle pour la construction de systèmes temps réel, en privilégiant un unique axe de solution.

L'analyse d'une classe plus large d'intergiciels adaptables (intergiciels configurables, génériques et schizophrène) nous permet de mettre en évidence les éléments fondamentaux des intergiciels, et nous sert de base pour élaborer notre solution.

Nous réutilisons les services de l'intergiciel tels qu'ils sont définis par l'architecture d'intergiciel schizophrène. Nous affinons au chapitre 3 leur description en montrant qu'ils peuvent se ramener à des abstractions simples. Par ailleurs, ces services nous permettent d'isoler et définir le "μBroker", élément clé qui contrôle le comportement de l'intergiciel. Nous montrons comment ce module peut être utilisé pour prendre en charge plusieurs politiques de concurrence et de gestion des ressources.

Ces éléments de conception étant définis, nous nous intéressons au chapitre 4 à la vérification formelle de notre solution. Nous avons opté pour l'utilisation des réseaux de Petri pour la modélisation, et de la logique temporelle linéaire (LTL) pour l'écriture de propriétés de logique causale que notre système doit satisfaire. L'utilisation d'outils nouveaux nous ont permis de limiter l'explosion combinatoire inhérente à l'exploration de ce type de modèles. Ce faisant, nous proposons une architecture d'intergiciel dont le comportement causal est vérifié. Ceci permet de garantir certaines propriétés fortes comme l'absence d'interblocages (*deadlock*) dans notre système.

Dans la seconde partie, *"Mise en œuvre et expérimentations"*, nous nous intéressons à la mise en œuvre de notre architecture, et montrons qu'elle répond aux besoins que nous avons identifiés :

Au chapitre 5, nous présentons les briques logicielles de base qui nous ont servi à construire notre solution. Nous présentons les différents patrons de conception utilisés. Par ailleurs, nous illustrons comment ces patrons peuvent être utilisés pour étendre et construire des fonctions de l'intergiciel qui soient déterministes en proposant un adaptateur d'objets déterministe, des mécanismes de protocole asynchrones, et un mécanisme de transport temps réel basé sur le mécanisme TDMA.

Par la suite, nous montrons comment les différentes briques logicielles, services de l'intergiciel et le μBroker peuvent être employés pour la construction d'intergiciels adaptés. Nous décrivons le processus de construction d'une instance d'intergiciel à partir de ces éléments génériques. Nous montrons comment nous avons pu construire deux instances. La première, minimale, est dédiée aux systèmes embarqués. La seconde montre comment mettre en œuvre les spécifications RT-CORBA.

Une série de mesures complète notre étude au chapitre 7. Elle nous permet de valider sur plusieurs exemples significatifs le déterminisme temporel de notre solution.

Une dernière section est consacrée à l'analyse de nos travaux, ainsi qu'à leurs débouchés possibles.

Première partie

Étude théorique et conception

2

Problématique - État de l'Art

Sommaire

2.1 Vers une architecture d'intergiciel dédiée 29
- 2.1.1 Besoins en mécanismes de répartition 30
- 2.1.2 Définitions générales 31

2.2 Taxonomie des intergiciels pour systèmes temps réel 32
- 2.2.1 Ingénierie des systèmes temps réel 32
- 2.2.2 Génie logiciel 35
- 2.2.3 Vérification et Validation 37
- 2.2.4 Analyse 40

2.3 Analyse des intergiciels adaptables existants 41
- 2.3.1 Intergiciels configurables 42
- 2.3.2 Intergiciels génériques 43
- 2.3.3 Intergiciels schizophrènes 46
- 2.3.4 Synthèse 50

2.4 Synthèse et axes de travail 51
- 2.4.1 Critique des architectures existantes 51
- 2.4.2 "Crise de l'Intergiciel" 51
- 2.4.3 Éléments de solution et méthodologie 52

CEtte section définit et motive le problème que nous nous proposons de résoudre : la construction d'un intergiciel répondant aux besoins nombreux des applications temps réel réparties embarquées. Nous délimitons le domaine de notre étude, et proposons une première classification des éléments de solution partielle existants.

2.1 Vers une architecture d'intergiciel dédiée

La section précédente a fourni une première approche de ce que sont les systèmes Temps Réel Répartis Embarqués. Plusieurs travaux ont cherché à fournir une plateforme pour la construction de tels systèmes. Cette section présente les solutions les plus significatives, en les classant suivant la méthodologie de conception employée.

2.1.1 Besoins en mécanismes de répartition

Nous partons de l'hypothèse que la construction d'un système TR^2E nécessite un intergiciel reposant sur un environnement d'exécution temps réel.

En effet, de la même manière qu'un système d'exploitation fournit à l'utilisateur un cadre général pour l'utilisation des ressources de son système ; un intergiciel fournit un cadre général pour la construction d'un système réparti, tout en réduisant voire supprimant une partie de la complexité inhérente à la construction d'applications réparties.

Ainsi, nous retiendrons que la construction d'une application TR^2E nécessite la mise en place et la coopération des éléments suivants :

- *un noyau temps réel* : qui prend en charge l'interface avec les couches matérielles du système, et fournit des mécanismes de gestion de la concurrence et de la mémoire à bas niveau ;
- *un intergiciel* : qui fournit les fonctions pour la mise en œuvre d'un (ou plusieurs) modèle de répartition, ainsi qu'un ensemble de mécanismes permettant son adaptation. Cet intergiciel utilise les primitives du noyau temps réel choisi ;
- *un référentiel* qui permet, en fonction de l'application à construire et des mécanismes utilisés, d'analyser son architecture, et d'en déduire ses propriétés.

Ces éléments génériques ne suffisent pas à déterminer complètement des éléments de solutions. En effet, il existe une multiplicité de noyaux temps réel, d'intergiciels chacun adapté à un domaine d'application (par exemple avionique, spatial, automobile) ou à une famille d'applications (tolérance aux pannes, applications critiques, multimédia).

De ce fait, il est nécessaire de classer les différents besoins en mécanismes de répartition spécifiques et de proposer une architecture qui puisse répondre aux besoins d'une large classe d'applications.

Dans [Stefani, 1996], les auteurs définissent et motivent seize prérequis nécessaires à la construction d'intergiciels temps réel basés sur une approche orientée objets répartis. Ces prérequis sont définis en trois classes de besoins : *modularité*, *fonctionnel* et *non-fonctionnel*. Ils impliquent la définition d'une architecture d'intergiciel qui fournit les éléments suivants :

(R1) *stricte séparation* entre les mécanismes pour la répartition et les politiques de fonctionnement ;
(R2) *abstraction* des ressources utilisées par l'intergiciel ;
(R3) *contrôle complet* sur les paramètres de Qualité de Service (QdS) ;
(R4) *métriques* associées à l'exécution de ses fonctions.

Nous notons que ces prérequis couvrent avant tout la partie implantation de l'intergiciel. Ils définissent des mécanismes qui peuvent aider le développeur d'applications TR^2E dans sa tâche de conception, mais ils ne définissent pas de guide méthodologique complet permettant d'analyser et comprendre les choix de conception effectués. Ceci est un frein à la construction de systèmes déterministes.

Dans [Rajkumar *et al.*, 1995], les auteurs introduisent une famille de prérequis et d'objectifs pour un intergiciel temps réel, dont certains sont similaires à ceux déjà présentés. En particulier, les auteurs insistent sur la nécessité de définir à haut niveau les mécanismes de répartition, qui sont alors utilisables de manière totalement portable sur plusieurs architectures, et qui font abstraction des mécanismes sous-jacents (couches de protocoles, système d'exécution). Ainsi, les auteurs présentent un mécanismes basé sur l'envoi et la réception de messages pour la construction d'un intergiciel temps réel. Ceci nous amène à dégager un cinquième besoin :

(R5) *mécanismes haut niveau* facilitant la conception et l'analyse de l'intergiciel et de l'application.

Ces mécanismes de haut niveau fournissent le canevas permettant *1)* de définir son application à partir de constructions bien définies, *2)* d'analyser et éventuellement vérifier l'application.

Ces prérequis nous serviront par la suite de fil conducteur à la définition et la construction d'une architecture d'intergiciel adaptable pour systèmes TR^2E.

2.1.2 Définitions générales

La prise en compte de ces besoins pose de nombreux problèmes de conception et de réalisation. La spécificité des applications temps réel motive la définition d'une architecture dédiée, suffisamment riche pour prendre en compte la diversité des besoins qui découlent des problématiques de l'embarqué, du temps réel et de la répartition.

Nous introduisons plusieurs concepts dont les définitions sont issues de [Quinot, 2003] et de [Coulouris *et al.*, 1994]. Elles nous serviront tout au long de ce mémoire :

Définition 2.1.1. *Un mécanisme de répartition est une fonction atomique nécessaire au bon déroulement d'une application répartie. Il peut être transparent pour l'utilisateur, ou supporté par un composant logiciel visible (API, structure de données, etc).*

Parmi les mécanismes de répartition classiques, on peut citer le *nommage*, qui permet de retrouver une entité à partir d'une référence textuelle. Ce mécanisme peut être explicite, dans le cas des spécifications CORBA qui définissent un tel mécanisme dans le service `COS Naming` ; ou transparent, tel que définit dans l'annexe des systèmes répartis du langage Ada 95.

Définition 2.1.2. *Un modèle de répartition est un ensemble formé de la définition d'entités abstraites interagissantes, leurs interactions et d'une projection de ces entités sur les nœuds d'un système réparti. Il s'exprime à l'aide d'un ensemble de mécanismes de répartition.*

On peut ainsi citer le modèle des appels de procédures distantes (*"Remote Procedure Call", RPC*), des objets répartis (*"Distributed Object Computing", DOC*) mis en avant par CORBA, la mémoire partagée répartie, le passage de messages synchrone ou asynchrone.

Notons enfin que pour ces modèles, de nombreuses politiques peuvent exister, insistant sur certains aspects de la sémantique du modèle. Le passage de messages peut être réalisé sous le contrôle d'une transaction, supporter la communication de groupe, etc.

Définition 2.1.3. *Une politique est la réalisation concrète d'une fonction de l'intergiciel, correspondant à une spécialisation d'une fonction générale. Elle peut éventuellement être paramétrée.*

La notion de politique est nécessaire à l'architecture d'un intergiciel : elle permet d'exprimer plusieurs variations d'un même composant, par exemple la politique de traitement des évènements entrants.

Définition 2.1.4. *Un intergiciel implante un ou plusieurs modèles de répartition au dessus de blocs élémentaires fournis par l'environnement d'exécution, tels que les processus et le passage de messages.*

L'intergiciel contient donc l'ensemble des mécanismes et politiques permettant à un développeur de construire son application. Notons cependant qu'un intergiciel n'est

pas toujours suffisant. D'autres services annexes peuvent être nécessaires, comme le montrent les spécifications CORBA qui séparent le cœur des services annexes, ou encore CORBA CCM qui utilise CORBA comme support d'exécution.

Ces différentes définitions vont nous permettre d'analyser puis classifier plusieurs intergiciels existants conçus pour les applications temps réel. Nous détaillons notre analyse dans la section suivante.

2.2 Taxonomie des intergiciels pour systèmes temps réel

Du fait de la diversité des systèmes Temps Réel, Répartis Embarqués, nous constatons qu'il n'existe pas un modèle d'architecture canonique pour le support de la répartition, mais plusieurs familles d'architectures privilégiant chacune un axe de conception.

Nous présentons dans ce premier état de l'art un tour d'horizon des intergiciels conçus pour les applications temps réel, et les axes de conception qu'ils retiennent :

1. un axe privilégiant les méthodes issues de l'ingénierie des systèmes temps réel ;
2. un axe orienté sur les méthodes de génie logiciel au sens large ;
3. un axe basé sur les méthodes formelles.

Nous présentons chacun de ces axes, ainsi que les projets marquants pour chacun.

2.2.1 Ingénierie des systèmes temps réel

La conception et la réalisation de systèmes temps réel mono-processeur, non répartis est maintenant un processus maîtrisé grâce à de nombreuses méthodes de conception, et la mise à disposition d'algorithmes pour le calcul d'ordonnancement répondant aux contraintes temporelles de l'application.

Dans ces conditions, la description du système est suffisamment précise pour appliquer ces techniques de conception et de validation. Pour ce faire, on restreint le système de façon à être dans le domaine de validité de différents algorithmes. La complexité est réduite pour faciliter l'analyse du système.

Suivant la même démarche, différents projets se sont intéressés à la construction d'intergiciels qui répondent à ces problématiques. Nous présentons dans cette section ARMADA, OSA+, microORB et nDDS.

ARMADA

ARMADA [Abdelzaher *et al.*, 1997] est un projet collaboratif entre la Laboratoire d'Informatique Temps Réel (*Real-Time Computing Laboratory (RTCL)*) de l'université du Michigan, et le centre de technologie d'Honeywell. ARMADA propose un support pour la communication temps réel, un service pour la tolérance aux pannes et une suite d'outils pour l'évaluation et la validation de la fiabilité des applications. ARMADA est prévu pour la construction d'applications de type "commande/contrôle"

Le projet s'est avant tout intéressé au développement d'un service de communication temps réel pour un micro-noyau dédié. Une architecture générique indépendante de tout service de communication est introduite comme modèle permettant de garantir le déterminisme et la QdS de système de communication indépendemment du système de communication utilisé. Une architecture en couches permet l'insertion de nouveaux services. Le service de communication est construit autour de trois besoins [Abdelzaher *et al.*, 1999] :

- isolation des ressources : pour éviter la famine en cas de comportements malicieux d'un nœud, ou une surcharge sur un des canaux de communications ;
- différenciation de services : affectation de priorités différentes suivant le type de communication ;
- dégradation de service programmée en cas de surcharge du système.

ARMADA propose une bibliothèque de communication avec une sémantique basée sur les priorités (*"Communication Library for Implementing Priority Semantics", CLIPS*). CLIPS propose des mécanismes pour la gestion des ressources pour construire un canal de communication temps réel [Ferrari & Verma, 1990], [Kandlur *et al.*, 1991]. ARMADA inclut par ailleurs un protocole pour la communication de groupe, tolérante aux pannes : RTCAST.

Enfin, le projet fournit une suite d'outils pour l'évaluation et la validation du comportement temporel et la tolérance aux pannes du système construit. Ces outils incluent un injecteur de fautes au niveau système, protocolaire, et applicatif ; un générateur de charge ; et un outil de surveillance et visualisation de performances.

Cependant, ARMADA est fortement couplé au micro-noyau CORDS sur lequel il repose. De ce fait, ARMADA ne fournit pas une grande portabilité. Ceci est un frein à sa large utilisation dans des projets hétérogènes. De plus, ARMADA ne fournit pas un support pour un standard industriel tel que CORBA. Tous les développements autour de cet intergiciel sont donc spécifiques.

OSA+

OSA+ [Schneider *et al.*, 2002; Bechina *et al.*, 2001] est un intergiciel dédié aux systèmes embarqués temps réel. Il repose sur une architecture de type micro-noyau pour réduire le surcoût introduit par un intergiciel : le cœur de la plate-forme fournit les services de base, indépendants du matériel ou du système d'exploitation utilisé. Le cœur peut être ensuite étendu pour fournir d'avantage de fonctionnalités.

OSA+ propose un modèle de répartition basé sur les évènements traités par un ou plusieurs "services". Chaque service fournit un mécanisme de base de l'intergiciel : service de gestions des processus, des évènements, de communication, de résolution des noms, de gestion de la mémoire. De par sa conception, OSA+ a un encombrement mémoire faible, de l'ordre de $60Ko$, qui répond parfaitement aux contraintes de l'embarqué. Cependant, l'ajout de fonctions plus évoluées tel que le support de plusieurs protocoles, de nouveaux mécanismes d'ordonnancement n'est pas discuté par les auteurs, et reste problématique.

Comme pour ARMADA, le développement autour de OSA+ est spécifique à une machine virtuelle Java particulière, rendant toute réutilisation impossible et limitant l'interopérabilité avec d'autres applications.

microORB

microORB a été développé par SciSys Ltd dans le cadre du projet de rechercher RAMA (*"Remote Agent Management Architecture"*) pour l'Agence Spatiale Européenne (ESA)[1]. L'objectif du projet RAMA est de montrer qu'un système de contrôle embarqué peut être implanté en utilisant plusieurs entités logiciels faiblement couplées, qui répondent au cahier des charges strict de l'ESA.

[1] Il est prévu que microORB soit diffusé suivant une licence Open Source par l'ESA.

Ce système est conçu pour passer avec succès les processus de vérification et de validation en vue d'une qualification. Ainsi, microORB peut être utilisé pour la construction d'applications critiques et répondre aux exigences des domaines spatiaux. Enfin, le système construit doit permettre l'ajout, le retrait, ou la mise à jour de ses composants tout en restant en fonctionnement.

Un modèle a été mis au point pour représenter les interactions entre entités applicatives sous la forme d'évènements. Il permet ainsi d'effectuer quelques vérifications sur le comportement du système, et vérifier par exemple le respect des échéances temporelles.

Ce modèle a été implanté en utilisant un intergiciel basé sur une infrastructure logicielle orientée composant. microORB propose une bibliothèque écrite en C ANSI, utilisant les APIs POSIX, implantant RT-CORBA 1.x (ordonnancement statique), CORBA AMI ainsi qu'un mécanisme de transport extensible prévu pour la plate-forme spatiale en cours de normalisation CCSDS SOIS (*"Spacecraft On-board Interface Services"*) [Plummer & Plancke, 2002].

Ces composants ont été développés suivant une approche incrémentale (*"bottom-up"*). L'implantation a été réalisée en prenant garde à ne conserver que les services utiles à l'application finale, garantissant une faible empreinte mémoire ($60Ko$).

Cet intergiciel fournit une API normalisée, inspirée des spécifications CORBA dont il ne conserve qu'un sous-ensemble. La démarche de conception alliant modélisation et implantation est intéressante car elle fournit de solides garanties sur le fonctionnement correct du système. Cependant, cet intergiciel est taillé sur mesure pour un modèle de répartition spécifique. Ceci limite sa réutilisabilité pour différents projets.

nDDS

L'OMG a défini le "Data Distribution Service" (DDS) [OMG, 2004a]. Cette spécification complète les spécifications des services COS Notification et COS Event définies dans le cadre de CORBA et les étend à un cadre plus générique, dans la mouvance MDA (*"Model Driven Architecture"*).

Ainsi, DDS définit un modèle générique (PIM, *"Platform Independent Model*) d'un intergiciel orienté messages, dont les politiques de configuration et déploiement sont adaptées aux applications temps réel. DDS est formé de deux couches : DCPS (*"Data Centric Publish/Subscribe"*) qui se charge du transfert efficace des données entre nœuds, et DLRL (*"Data Local Reconstruction Layer"*) qui se charge de l'envoi et du filtrage des données à l'échelle du nœud et de l'application. DDS peut soit être implanté au dessus d'un intergiciel existant comme CORBA, soit être implanté directement au dessus des primitives de communication de l'exécutif.

nDDS est l'intergiciel qui a inspiré les spécifications DDS. Il s'agit d'un intergiciel multi-langages (C, $C++$, *Java*) et disponible sur plusieurs noyaux temps réel. nDDS est implanté comme une couche de communication bas niveau, offrant les primitives de la couche DCPS au développeur. Ce modèle est particulièrement bien adapté aux applications de type contrôle/commande, et a fait ses preuves dans les systèmes de pilotage de systèmes avioniques.

nDDS fournit des primitives de répartition ayant une sémantique simple, permettant ainsi la construction rapide de systèmes TR^2E. Ce modèle est orienté données, il est donc restreint à une famille d'applications.

Synthèse

Cette analyse des intergiciels dédiés aux temps réel indique que cette famille d'intergiciels est minimaliste, et ne fournit qu'un ensemble très restreint de fonctionnalités.

Ces fonctions sont intégrées à l'ordonnanceur du nœud local et proposent une surcouche aux primitives de communication de l'exécutif. Ils simplifient certaines tâches de configuration des canaux de communications. La réutilisabilité de ces intergiciels est de ce fait réduite par le manque de portabilité.

Néanmoins, ces intergiciels répondent parfaitement aux besoins de la classe d'applications temps réel embarquées pour laquelle ils ont été conçus. Ils apportent des mécanismes pour la construction d'applications réparties temps réel, et décrivent la liste des fonctionnalités utiles, et les moyens de les mettre en œuvre.

2.2.2 Génie logiciel

Les besoins en répartition évoluant, les intergiciels doivent s'étoffer et fournir de nombreux services, chacun répondant à des besoins fonctionnels précis (choix des couches de transport, réification des mécanismes d'accès aux entités réparties, contrôle de la concurrence, etc).

La prise en compte de ces besoins passe par une réflexion sur l'architecture de l'intergiciel, et l'utilisation des concepts du génie logiciel au service du "génie des intergiciels" de façon à pouvoir générer plusieurs configurations de l'intergiciel.

Dans [Schmidt & Buschmann, 2003], les auteurs montrent comment un ensemble bien défini de bibliothèques et schémas de conception permettent de construire des intergiciels adaptables. Nous présentons ici TAO, RT-Zen et nORB.

Ces différents projets sont issus des travaux autour de la bibliothèque ACE, l'implantation de patrons de conception pour applications réparties.

TAO

TAO [Schmidt & Cleeland, 1997] est une implantation Open Source de CORBA, dont il couvre la quasi totalité. Il est développé par le groupe DOC (*"Distributed Object Computing"*) rassemblant les universités de Washington à St Louis, d'Irvine en Californie et de Vanderbilt à Nashville.

TAO est implanté en $C++$, et vise les environnements temps réel, mais aussi les environnements Unix et Windows. Il est compatible avec les spécifications RT-CORBA [OMG, 2003b], dont il a grandement influencé la conception, ainsi que la plupart des spécifications annexes dédiées aux applications temps réel ou embarquées (COS Event, COS Notification, Minimum CORBA, etc).

Le support des contraintes temps réel par TAO a été obtenu grâce à une analyse précise des différentes fonctions mises en œuvre par l'intergiciel, et leur optimisation (pile protocolaire, demultiplexage des requêtes, génération de code, etc).

Le cœur de TAO est construit autour de ACE (*"The Adaptative Communication Environment"*) [Schmidt *et al.*, 1998]. ACE est une bibliothèque portable qui supporte des primitives de communications et de concurrence. TAO supporte ainsi de nombreux protocoles [O'Ryan *et al.*, 2000], politiques de gestion de la concurrence et du multiplexage des requêtes [Pyarali *et al.*, 2001].

Pour ce faire, TAO utilise de façon intensive des *patrons de conception* (*"design pattern"*) [Gamma *et al.*, 1994]. Un patron est un élément de conception de haut ni-

veau : « Une description d'objets communicants et de classes qui sont particularisés pour résoudre un problème général dans un contexte particulier ».

L'architecture de TAO utilise de nombreux patrons de conception, décrits en détail dans de nombreux articles et livres [Schmidt & Cleeland, 1998]. En particulier, plusieurs patrons contribuent à la configurabilité de l'intergiciel.

D'un point de vue fonctionnel, TAO fournit une solution élégante pour la construction d'intergiciels temps réel, en fournissant une extrême configurabilité. Ceci a néanmoins un prix : l'intergiciel se révèle lourd, son empreinte mémoire dépasse $2Mo$ pour des configurations simples. De plus, le recours aux patrons de conception induit une architecture complexe utilisant de nombreuses fonctionnalités des langages orientés objets, rendant l'analyse de toute application construite au dessus de TAO très difficile voire impossible.

RT-Zen

RT-Zen [Klefstad *et al.*, 2002] est une implantation Open Source des spécifications CORBA et RT-CORBA. RT-Zen est un intergiciel écrit en Java, suivant les spécifications Java temps réel (*Real-time Specification of Java (RTSJ)*) [Gosling & Bollella, 2000]. RT-Zen peut être déployé aussi bien sur des machines virtuelles Java classiques, que sur des plates-formes RTSJ. Ce projet a débuté en 2002, et ne fournit pas encore de version publique librement évaluable[2].

La conception et la réalisation d'un intergiciel temps réel tirant partie des spécificités de RTSJ nécessitent l'utilisation précise des certaines fonctionnalités de ce langage, comme la mémoire hiérarchisée, la gestion des entrées/sorties, etc. Ceci rend la conception de RT-Zen complexe [Krishna *et al.*, 2004].

RT-Zen a hérité de certaines techniques de conception utilisées par TAO, en particulier plusieurs de ses patrons de conception. L'architecture de RT-Zen repose sur un modèle en couches interchangeables. Ce modèle repose sur une identification précise des services de l'intergiciel dont le comportement est configurable, voire non nécessaire. Ces services sont alors rendus facultatifs à travers l'utilisation du patron de conception "composant virtuel"(*virtual component*) [Corsaro *et al.*, 2002]. Une telle architecture permet de n'inclure que les composants nécessaires, et la rend donc intéressante pour les systèmes embarqués. RT-Zen souffre cependant des mêmes limitations que TAO, son architecture est difficilement analysable du fait de sa forte configurabilité.

nORB

nORB [Subramonian *et al.*, 2003; Gill *et al.*, 2003a] est un intergiciel léger, conçu pour les systèmes embarqués. Il a été mis au point pour la plate-forme expérimentale de Boeing (*OEP - "Boeing Open Experimental Platform"*), rattaché au programme NEST de la DARPA [Sharp, 1998].

nORB a pour but la garantie de performances et de respect des échéances fermes pour une large variété de plates-formes, et doit pouvoir gérer une charge de travail variable, par exemple rafale de messages sporadiques. Par ailleurs, nORB doit pouvoir être reconfigurable, et pouvoir modifier les canaux de communication utilisés pour des configurations embarquées. Un nœud basé sur nORB nécessite, suivant les services retenus, environ $340Ko$.

[2]Une distribution de RT-Zen est annoncée sur le site de auteurs, mais non téléchargeable.

L'extension FCS/nORB fournit une infrastructure de répartition temps réel portable, auto-configurable. Ainsi, FCS/nORB intègre un mécanisme d'ordonnancement auto-stabilisant (*" Feedback Control real-time Scheduling (FCS)"*) [Chenyang *et al.*, 2003]. FCS/nORB implante une boucle de rétro-action basée sur un contrôleur PID (Proportionnel Intégral Dérivée) qui garantit des performances temps réel en ajustant le taux des invocations distantes de façon transparente pour l'application.

L'architecture de nORB suit une approche par composition de services "du bas vers le haut" (*"bottom-up"*). Ceci permet aux applications construites de n'embarquer que les fonctionnalités requises. Pour ce faire, nORB réutilise les services et patrons de conception proposés par ACE, TAO et l'ordonnanceur configurable Kokyu [Gill *et al.*, 2003b], en les adaptant aux systèmes de faible capacité. Cette approche montre les possibilités offertes par la réutilisation de COTS (*"Components-Off-The-Shelf"*), mais requiert une connaissance importante des différents modules.

Synthèse

Ces différents projets démontrent l'intérêt des techniques de génie logiciel (patrons de conception, modèle objet, généricité, etc) pour la conception d'intergiciels pour les systèmes temps réel. Ils fournissent une palette de fonctions répondant chacune à des besoins précis. Les performances obtenues répondent en partie aux exigences. Cependant, la notion de modèle comportemental de l'intergiciel, associée à un processus de développement rigoureux n'apparaît pas.

En particulier, la phase de validation est réduite à un jeu de tests et de mesure de performances. Les différents mécanismes mis en œuvre font appel à des optimisations locales des différentes fonctions utilisées, et reposent sur de nombreuses fonctionnalités avancées des langages objets (programmation générique, héritage multiple, etc) qui rendent difficile la vérification ou la validation des propriétés de l'application.

Cette classe d'intergiciels fournit un ensemble de services pertinents pour la construction de systèmes TR^2E. Néanmoins, il est nécessaire de procéder à un long travail de validation pour s'assurer que leur architecture ne remet pas en cause les propriétés de l'application. Notons par ailleurs que ces projets ne se sont intéressés qu'au modèle de répartition à objets répartis.

2.2.3 Vérification et Validation

Les sections précédentes ont montré que les intergiciels pouvaient fournir une réponse aux besoins des applications. Cependant, les projets identifiés sont soit trop contraignants en terme d'architecture, soit trop complexes à manipuler, et rendent donc difficile la validation ou la vérification de leurs propriétés. Ceci réduit donc l'utilisation des intergiciels pour ce type d'application.

En réponse, certains projets se sont intéressés à l'utilisation de méthodes formelles ou de techniques de modélisation pour la conception d'un système réparti temps réel. La mise en place de ces techniques permet de garantir les propriétés comportementales et temporelles du système. Nous présentons ici TURTLE-P, Cadena, QuO et CosMIC.

TURTLE-P

TURTLE-P [Apvrille *et al.*, 2003], développé au LAAS-CNRS, propose un profil UML pour la validation d'applications réparties. Il étend le profil TURTLE (*"Timed UML and RT-LOTOS Environment"* [Apvrille *et al.*, 2001], une extension des dia-

grammes de classes UML 1.5 fournissant une sémantique formelle au parallélisme et aux synchronisations entre classes. Il ajoute aux diagrammes d'activités des opérateurs de synchronisation et des opérateurs temporels (délai, etc). TURTLE-P étend les notations de TURTLE et permet la modélisation de l'architecture générale d'un système réparti (topologie des nœuds, canaux de communication, etc), ainsi que les paramètres de QdS du système.

TURTLE-P propose un processus itératif par raffinement pour la conception d'une application, alternant analyse, conception et validation grâce à l'outil RTL. Pour ce faire, les spécifications TURTLE-P sont traduites en RT-LOTOS, puis analysées [LAAS, 2004]. Cette analyse s'opère soit par simulation du système, soit par vérification formelle à l'aide d'automates temporisés. Ces derniers permettent une exploration des états accessibles, ou de la vérification de propriétés par l'outil Kronos. Cette démarche a ainsi été appliquée à la modélisation d'un système de communication par satellite.

Cette approche permet une analyse formelle d'un modèle de l'application en vue de déterminer ses propriétés, et vérifier son exactitude. Cependant, elle modélise un système, sans référence à un intergiciel ou une plate-forme d'exécution concrète. Ceci complique la phase d'implantation du système. Par ailleurs, seul le modèle est vérifié, le problème de l'implantation de l'application subsiste : il est nécessaire de garantir que les propriétés vérifiées sont toujours présentes. Enfin, l'explosion combinatoire peut limiter l'étendue des modèles (et donc des systèmes) vérifiables.

Cadena

Cadena [Deng *et al.*, 2003] est un environnement de modélisation pour les applications basées sur les spécifications composant de CORBA (CORBA CCM). Cadena s'appuie sur une série d'outils permettant de définir une application, couvrant les différentes étapes de son cycle de vie. En particulier, ces outils offrent différentes vues d'un même système, et permettent de spécifier un comportement de plusieurs manières différentes. Cette redondance de l'information permet la mise en évidence d'erreurs de conception lors de la fusion des différentes vues.

Une palette d'outils d'ingénierie inverse permettent de déduire d'un modèle d'une application les dépendances entre composants, ainsi que des informations sur le comportement temporel ou sur l'utilisation de mécanismes de répartition pour la communication entre composants.

Cadena peut se coupler avec le vérificateur de modèle (*"model checker"*) Bogor. Bogor est un model checker adaptable. Un langage de représentation intermédiaire permet de définir des types primitifs correspondant aux entités manipulées : composants, ports, évènements ainsi que leur sémantique. Ceci permet une projection des entités CORBA CCM vers les notations utilisées par Bogor en vue de vérifier les propriétés du modèle. Les auteurs mettent en avant le haut niveau d'abstraction de ces transformations, qui permet de décrire un système.

Cette approche intègre l'utilisation des spécifications CORBA CCM pour la modélisation et la vérification de systèmes. De fait, elle masque les nombreuses interprétations réalisées lors de l'implantation effective de ces spécifications. Ce problème est non négligeable, il peut masquer certains problèmes au déploiement ou à l'exécution du système. Ainsi, les auteurs de [Bastide *et al.*, 2000] ont mis en évidence de nombreuses différences de comportement entre des implantations différentes de services CORBA.

De plus, le haut niveau d'abstraction utilisé complique le travail du model checker. La génération automatique d'un modèle à partir d'une spécification de haut niveau tend

à générer un modèle complexe, ce qui augmente d'autant la taille de l'espace d'états à analyser. L'explosion combinatoire qui en résulte limite les capacités à analyser ces modèles, et nécessite de recourir à des configurations de type "grappe de calcul" pour tester des modèles simples [Hoosier *et al.*, 2004].

Bien que Cadena permette de définir explicitement le rôle de l'intergiciel au sein de l'application, cette approche ne permet pas de rendre compte de ces détails d'implantation et de ces choix de conception. Cette distance entre modèle et réalisation ne fournit donc au mieux qu'une indication du comportement de l'application et non un résultat définitif.

QuO

QuO (*"Quality Objects"*) [Vanegas *et al.*, 1998] propose un cadre général pour la prise en compte de la Qualité de Service (QdS) d'applications réparties basées sur une utilisation intensive des canaux de communication (diffusion de flux, systèmes d'alerte, etc). QuO cherche à fournir une solution pour une large classe de systèmes, allant des systèmes embarqués aux systèmes à large échelle.

QuO propose un mécanisme générique pour tirer avantage des mécanismes de QdS mis en place par les couches protocolaires basses, telles que les mécanismes de réservation `RSVP` [Braden *et al.*, 1997]. QuO introduit une passerelle entre les points où les informations de QdS sont construites et ceux où elles sont utilisées (couches protocolaires basses) et les points où elles sont utiles à l'application (objets répartis). Ceci permet d'adjoindre un mécanisme de QdS à un modèle de répartition, et a été notamment utilisé par des applications basées sur CORBA ou RMI.

Cette passerelle est réalisée à l'aide de points d'interception ("proxies" ou "delegate") placés sur le chemin d'invocation d'une requête distante. Ces points d'interception prennent en charge l'adéquation entre un contrat de QdS que deux entités applicatives tentent de maintenir, et les paramètres de configuration des couches protocolaires le réalisant. Les contrats de QdS sont exprimés à l'aide d'une famille de langages dédiés de description (QDLs, *"Quality Description Languages"*) ; une suite d'outils permet de valider et déployer ces contrats sur les différents nœuds de l'application.

Ce mécanisme d'interception peu intrusif permet ainsi de garantir la QdS d'une application répartie, par exemple les garanties temps réels souples des applications multimédia. Néanmoins, ce type d'approche a un impact négatif sur les performances. Par ailleurs, il agit en modifiant l'exécution du programme en ajoutant des mécanismes de contrôle. Il ajoute donc un niveau d'interaction au sein de l'application, ajoutant ainsi à sa complexité et réduisant les possibilités d'analyse.

CosMIC

CosMIC (*"Component Synthesis using Model Integrated Computing"*) [Gokhale *et al.*, 2002] fédère plusieurs projets pour fournir une suite d'outils permettant la construction d'applications réparties temps réel à base de composants. Il réutilise les modules de TAO (CORBA) and CIAO (implantation de CORBA CCM au dessus de TAO).

Les différents outils utilisés par CosMIC couvrent le cycle de vie d'une application, de la phase de conception initiale, jusqu'à la phase de configuration et déploiement. De manière analogue à QuO, ces outils utilisent une famille de langages de description pour la définition des différents paramètres de QdS, la configuration des nœuds, etc.

Cette approche est intégrée au cycle de vie de l'application. De ce fait les mécanismes de contrôle et gestion de la QdS au sein de l'application sont intégrés plus

finement. Néanmoins, elle repose sur l'utilisation extensive de ACE, TAO et CIAO, ceci conduit à embarquer de nombreuses fonctionnalités dans chaque application, conduisant à une empreinte mémoire très importante.

Par ailleurs, même si CosMIC prend en charge une partie de la validation du contrat de QdS passé entre plusieurs nœuds, aucune garantie stricte n'existe sur les modules utilisés. De fait, la chaîne de preuve entre le système conçu, et sa réalisation est cassée.

Synthèse

Ces différents outils fournissent un modèle pour la conception et la validation de certaines propriétés d'une application répartie temps réel. Cependant, cette démarche est incomplète : certains étapes de la démarche peuvent remettre en cause les propriétés obtenues sur une spécification haut niveau. Ainsi, l'intergiciel est absent de la démarche, ou incomplètement caractérisé. De fait, le résultat obtenu ne fournit aucune garantie stricte quant aux propriétés de l'application finale. Il permet seulement de valider le modèle retenu.

Plusieurs raisons techniques et architecturales expliquent cette limitation : difficultés à modéliser les différents blocs, explosion combinatoire lors de l'exploration du système, différences entre le modèle et l'intergiciel créé.

2.2.4 Analyse

Les sections précédentes ont présenté plusieurs solutions au problème de la construction d'applications TR^2E. Elles montrent comment un processus raisonné permet la construction d'intergiciels à même de supporter les contraintes de ces systèmes.

Cependant, ces solutions sont incomplètes et se concentrent sur un unique trait de l'application à construire, et négligent d'autres aspects nécessaires à l'ingénierie des intergiciels, et des systèmes temps réel, tels que l'adaptabilité, la garantie des propriétés. En se concentrant sur un aspect spécifique de l'application à construire, elles entrent dans un cercle vicieux :

– les approches basées sur l'ingénierie des systèmes temps réels se concentrent sur la phase de conception et d'implantation du système à construire.
  L'intergiciel devient une couche intégrée à l'application et non plus une infrastructure pour la répartition réutilisable. L'intergiciel perd son rôle de plateforme portable et adaptable.
– les approches basées sur le génie logiciel s'intéressent aux mécanismes de configuration de l'intergiciel et aux optimisations locales permettant d'assurer le déterminisme de l'application construite. L'intergiciel conserve son rôle de plateforme réutilisable et adaptable pour la répartition.
  Cependant, ce type d'intergiciels apparaît trop lourd et trop complexe, rendant leur analyse difficile. Bien qu'une phase de validation sur des exemples permette de s'assurer de certaines de leurs propriétés, il est difficile de mener à bien un processus de certification complet permettant de qualifier l'application pour des besoins industriels forts tels que ceux de l'aéronautique.
– les projets se basant sur une approche vérification, validation ou simulation permettent de garantir certaines propriétés de l'application construite. Ces projets remplacent l'intergiciel par une abstraction de haut niveau, masquant ainsi de nombreux points de configuration. Cette abstraction permet de déduire certaines propriétés sur le comportement du modèle de l'intergiciel.

Néanmoins, cette approche introduit une trop grande distance entre le modèle et l'implantation sous-jacente. Ceci limite la portée des résultats obtenus et les rend difficilement exploitables lors de la phase de déploiement de l'intergiciel.

Ainsi, aucune architecture ne parvient à couvrir l'ensemble des besoins. Le tableau 2.1 fait la synthèse des architectures que nous avons examinées.

Projet \ Propriété	Temps réel	Embarqué	Adaptation	Analyse	Modèle de rép.	Standard	Point fort
ARMADA	$++$	$++$	$--$	$+$	$+$	$\times$	Contrôle/Commande
OSA+	$++$	$++$	$--$	$+$	$+$	$\times$	Orienté Évènement
microORB	$++$	$++$	$--$	$++$	$+$	$\surd$	CORBA Minimal
nDDS	$+$	$+$	$+$	$-$	$-$	$\surd$	Pub/Sub
TAO	$+$	$+$	$++$	$-$	$-$	$\surd$	Configuration
RT-Zen	$+$	$++$	$++$	$-$	$-$	$\surd$	Virtualisation
nORB	$++$	$++$	$+$	$-$	$-$	$\times$	COTS
TURTLE-P	$\emptyset$	$\emptyset$	$\emptyset$	$++$	$++$	$\surd$	UML Temps réel
Cadena	$\emptyset$	$\emptyset$	$\emptyset$	$++$	$++$	$\times$	Model Checking
QuO	$+$	$-$	$++$	$-$	$-$	$\surd$	Interception
CosMIC	$+$	$--$	$++$	$+$	$+$	$\surd$	Cycle de Vie

$\{--,-,+,++,\emptyset\}$: niveau de support d'une fonctionnalité, ou absence
$\{\times,\surd\}$: absence, support d'une fonctionnalité

TAB. 2.1 – Propriétés remarquables des intergiciels analysés

Cette analyse montre qu'il existe une séparation nette entre les capacités d'un intergiciel à répondre aux besoins des systèmes temps réel embarqués ; ses capacités d'adaptation et la vérification et la validation de ses propriétés. Or, ces trois facettes nous paraissent complémentaires pour pouvoir fournir une solution unifiée aux problèmes posés par les systèmes TR^2E.

Nous constatons en effet que l'inadéquation entre les besoins des applications et les services fournis par l'intergiciel constitue une *"Crise de l'Ingénierie des Intergiciels"* : de multiples architectures spécifiques sont conçues et déployées, chacune avec une conception propre pour résoudre un ensemble restreint de problèmes. Cette multiplicité des structures augmente le coût financier du développement de ces plates-formes, qui de *"composants pris sur l'étagère"* (*"Components-Off-The-Shelf", COTS*) deviennent des systèmes dédiés.

2.3 Analyse des intergiciels adaptables existants

En parallèle aux intergiciels temps réels, de nombreuses architectures ont été définies en réponse à des besoins plus généraux en des intergiciels adaptables.

Bien que n'étant pas prévu pour répondre aux besoins précis des systèmes TR^2E, ces architectures sont intéressantes car elles fournissent des solutions pour l'adaptabilité de l'intergiciel.

Notre objectif est d'étudier ces architectures pour en déduire les "bonnes pratiques" de conception sur lesquelles elles reposent et discuter des possibilités d'utilisation de leurs concepts pour nos travaux. Nous étudions les architectures configurables, génériques et schizophrènes.

2.3.1 Intergiciels configurables

Les intergiciels configurables forment la classe la plus générale d'intergiciel. Les modules de l'intergiciel sont construits de telle manière que le comportement de certaines de leurs fonctions puisse être défini par l'utilisateur. Dans ce contexte, on peut définir un intergiciel configurable comme suit :

Définition 2.3.1. *Un intergiciel configurable permet à l'utilisateur de sélectionner les éléments de l'intergiciel et les politiques répondant au mieux à ses besoins, parmi une bibliothèque prédéfinie.*

Parmi les points de configuration élémentaires, on peut citer la sélection et la configuration des canaux de communication (ports d'entrées/sorties), le nombre de tâches dédiées au traitement des requêtes, ... Nous présentons ici deux intergiciels configurables, retenus pour les mécanismes qu'ils utilisent et le niveau de configuration qu'ils proposent : TAO et GLADE

TAO : configurabilité locale par assemblage

Nous avons déjà présenté quelques éléments de l'architecture de TAO. Celui-ci se distingue par une utilisation forte des caractéristiques de l'orienté objet et des patrons de conception. Nous nous intéressons ici à ses mécanismes de configuration et déploiement.

L'intergiciel est configuré en activant certaines politiques de configuration lors de la compilation ou de l'initialisation du nœud. L'utilisateur peut alors choisir parmi un ensemble de modules ceux correspondant le mieux à ces besoins.

Cette approche offre un niveau de configuration par sélection parmi une palette. Cette sélection est intégrée au code ou à l'exécution de l'application. Aucun mécanisme ne permet de configurer l'application répartie dans son ensemble, ni d'assurer que la configuration d'un nœud est compatible avec les autres nœuds. Ceci tient en partie au modèle CORBA implantée par TAO qui n'adresse pas les problèmes de déploiement de l'application.

Le projet dynamicTAO [Kon *et al.*, 2000] étend les mécanismes de base de TAO en lui ajoutant la capacité de s'adapter et se reconfigurer dynamiquement. Une telle capacité est intéressante car elle permet l'adaptation au contexte d'exécution de l'application. Elle pose en retour le problème de la durée et de la gestion de l'état transitoire de l'application lors de sa reconfiguration.

GLADE : configurabilité globale par description

Le langage Ada 95 définit une annexe normative pour la répartition (*"Distributed System Annex", Ada/DSA*). Elle ajoute des mécanismes de répartition à quelques constructions du langage : appels de procédure à distance, objets répartis, variables partagées (*shared memory*), etc. Les utilisateurs de Ada/DSA peuvent concevoir, mettre en

application et tester leur application dans un environnement non distribué, puis commuter vers un mode réparti en recompilant leur application suivant les directives définies par Ada/DSA.

GLADE, la première implantation validée de l'annexe, inclut l'outil de déploiement GNATDIST [Kermarrec *et al.*, 1996]. Il fournit un cadre global intégré au compilateur GNAT pour la configuration et la construction d'applications basées sur Ada/DSA.

Ceci permet à l'utilisateur de spécifier la configuration de l'application à déployer. Le langage de description de GNATDIST permet ainsi de placer les composants de l'application sur les nœuds logiques de l'application, le paramétrage du sous-ensemble de communication ou des politiques d'allocation de ressources.

2.3.2 Intergiciels génériques

Les intergiciels génériques étendent le concept d'intergiciels configurables. Une étude des implantations d'intergiciels configurables existantes montrent qu'elles partagent de nombreux éléments de conception, de sorte qu'il est possible de définir un modèle de répartition, et son implantation, à l'aide d'un ensemble de service génériques en suivant une approche guidée par les fonctions à fournir. Ces éléments peuvent ensuite être adaptés pour correspondre à une spécification précise durant la phase de *personnalisation* de l'intergiciel.

Définition 2.3.2. *Un intergiciel générique est un ensemble d'entités, indépendantes de tout modèle de répartition, qui proposent des mécanismes pour la répartition représentés sous forme d'interfaces et de services à étendre.*

Définition 2.3.3. *Une personnalité est la combinaison d'entités qui implantent les interfaces d'un intergiciel générique, et qui fournit un accès à ses services. La sémantique des entités est précisée lors de la phase de "personnalisation" de l'intergiciel générique.*

Les intergiciels génériques peuvent être classés suivant le degré de généricité offert par la définition des modules de l'intergiciel, et le taux de réutilisation de code ainsi permis. Nous présentons UIC, ACT, Quarterware et Jonathan.

UIC : interfaces abstraites

UIC (*"The Universal Interoperable Core"*) [Román *et al.*, 2001] définit un ensemble d'interfaces abstraites représentant un modèle d'un intergiciel. Leur concrétisation permet de préciser le comportement du modèle de répartition à construire.

UIC propose une implantation des spécifications CORBA pour la partie cliente, orientée pour les systèmes légers de type PDA répondant à des contraintes fortes en terme de ressources, mais aussi en adaptabilité grâce à son architecture réflexive. Dans ce contexte, UIC a une occupation mémoire inférieure à quelques dizaines de Ko.

UIC se concentre sur certaines fonctions de base d'un intergiciel : protocoles de transport, gestion des connexions, représentations des données échangées, gestion des références, etc. Ces fonctions sont séparées les unes des autres, permettant de s'intéresser à l'implantation de chacune séparément.

Ce niveau de granularité permet au développeur de se concentrer sur chacune des fonctions pour les optimiser en taille mémoire ou en temps ; voire les désactiver lorsqu'elles ne sont plus nécessaires via un mécanisme de chargement dynamique de code.

Enfin, les mécanismes de réflexivité de UIC permettent le support de plusieurs personnalités, par exemple en attachant une pile protocolaire SOAP à l'intergiciel.

L'approche retenue par UIC permet au développeur de se concentrer précisément sur les fonctions requises par l'intergiciel. Celui-ci doit être capable de remplacer chacune de ces fonctions dans le canevas fourni.

Ces propriétés sont intéressantes dans un contexte où les ressources sont fortement limitées ; mais il ne facilite pas l'ingénierie de l'intergiciel en ne fournissant pas un guide méthodologique pour la reconnaissance et la projection des fonctions d'un modèle de répartition vers les entités définies. Cette phase d'identification et de projection peut se révéler technique et coûteuse.

ACT : pile protocolaire pour intergiciel

ACT (*"Advanced Communication Toolkit"*) [Francu & Marsic, 1999] propose un ensemble d'interfaces pour faciliter la construction d'intergiciels. ACT fournit une architecture pour l'implantation efficace de modèles de répartition bâtis autour du patron de conception *"Broker"* [Buschmann *et al.*, 1996]. Pour se faire, il définit quatre interfaces fournissant les services des couches hautes d'une pile protocolaire OSI : transport, protocole, client/serveur, représentation.

En parallèle, ACT permet à l'utilisateur d'adapter certains éléments de son moteur d'exécution. Ainsi, les objets `ConnectionFactory` et `RequestHandlerFactory` permettent de contrôler la gestion des connexions et le traitement des requêtes, ce qui permet de mettre en place des politiques de multiplexage des connexions, ou de concurrence dans le traitement des requêtes pour mieux gérer le parallélisme de l'application.

L'architecture de ACT a été déployée pour construire deux personnalités : CORBA et cBus, un intergiciel orienté message. Ces personnalités démontrent l'intérêt du schéma de conception retenu, et sa faculté à prendre en compte deux modèles de répartition de natures différentes.

Les personnalités cBus et CORBA sont construites autour d'un socle commun de composants logiciels représentant entre 35% et 59% de l'intergiciel construit. Ainsi, l'approche retenue par ACT offre au développeur une généricité basée sur la projection entre le modèle ISO d'un protocole, représenté par quatre composants, et un modèle de répartition. Il fournit donc un guide simple pour écrire de nouvelles personnalités. Néanmoins, ACT fournit peu de mécanismes de contrôle sur les composants génériques, hormis deux politiques de contrôles des évènements.

Quarterware : composants 'd'intergiciel 'RISC"

Quarterware [Singhai *et al.*, 1998] propose un ensemble de composants permettant la conception d'intergiciels suivant une approche basée sur des composants réduits, empruntée à la conception des microprocesseurs "RISC" (*Reduced Instruction Set Computers*). Ces processeurs définissent un ensemble de fonctions élémentaires qui permettent ensuite par assemblage de construire des instructions plus évoluées.

Par analogie à ces microprocesseurs, les différentes fonctions de l'intergiciel forment autant de composants que le développeur peut ensuite étendre et combiner pour fabriquer les personnalités correspondant à ses besoins. À titre d'illustration, Quarterware propose les personnalités CORBA, RMI and MPI.

Pour ce faire, Quarterware propose une démarche pour la reconnaissance et la définition des fonctions utilisées par plusieurs classes d'intergiciels. Ces fonctions sont isolées sous la forme de composants, pour lesquels les interfaces définissent les actions

qu'ils permettent, et les services qu'ils offrent. Ces composants sont définis de façon à les rendre indépendants les uns des autres, à permettre une recombinaison de plusieurs composants et rendre explicite leur assemblage. Ces trois conditions garantissent que plusieurs assemblages seront possibles, et simples à réaliser.

Ainsi, cette approche offre une connaissance précise d'un ensemble de composants, de leur rôle et de leur sémantique de façon à permettre à l'utilisateur d'exprimer un modèle de répartition complet. L'assemblage est réalisé en référence au modèle de répartition à construire. En particulier, aucun guide méthodologique ne permet de guider le développeur dans le choix de ses composants et leur assemblage, si ce n'est la compatibilité des interfaces.

Néanmoins, l'utilisation de ce type d'architecture est relativement complexe à mettre en place et à maîtriser. Par ailleurs, il est difficile d'assurer la pérennité du découpage fonctionnel proposé et de l'évaluer car le code n'est pas disponible.

Jonathan : abstraction du mécanisme de liaison

Jonathan [Dumant *et al.*, 1998] repose sur une architecture orientée composants qui définit les différents traits architecturaux de l'intergiciel.

Il est formé d'un cœur qui fournit les interfaces des mécanismes de base de l'intergiciel. Par ailleurs, il introduit la notion de *personnalités*, une instance du cœur qui implante un certain modèle de répartition. Des instances pour CORBA (*David*) et RMI (*Jeremie*) ont été construites. Notons par ailleurs qu'une personnalité pour le transfert de flux multimédia existe [Seinturier *et al.*, 1999].

La notion de *liaison* (*binding*) est le pivot de l'architecture de Jonathan. Une liaison représente le processus par lequel deux entités sont associées, et les ressources affectées à cet effet. La liaison permet aux entités liées d'interagir. Cette notion est issue notamment du modèle de référence *ODP* (*Open Distributed Processing*) [ODP, 1995].

La liaison peut être locale à un nœud, ou au contraire concerner deux nœuds distants. Une fois réalisée, la fonction de liaison retourne au nœud initiateur de la liaison une référence sur l'objet lié, le *subrogé*. Ce subrogé met en œuvre les fonctions de répartition. En particulier, les interactions avec l'objet distant se font de manière transparente au travers du subrogé, qui reproduit l'interface de l'objet représenté.

Ainsi, une conséquence immédiate de cette architecture est le parti pris implicite pour un modèle de répartition basé sur les objets répartis. Ceci est confirmé par les expérimentations réalisées autour de Jonathan qui se sont intéressées à l'implantation de CORBA et RMI.

Notons enfin que Jonathan, bien que fournissant un mécanisme central pour la construction d'intergiciel, n'offre pas une forte réutilisation de code, de l'ordre de 10% tel qu'indiqué dans [Quinot, 2003].

SAFRAN : architecture à composants adaptables

SAFRAN [David & Ledoux, 2003] s'intéresse à la construction d'applications adaptables par l'utilisation de *composants logiciels*. SAFRAN repose sur la méthodologie orientée composants Fractal [Coupaye *et al.*, 2002]. Il permet d'associer dynamiquement à l'application des règles d'adaptation en réaction à des évènements détectés par les composants applicatifs.

Ces règles sont exprimées à l'aide d'un langage dédié permettant d'exprimer un ensemble de données à mesurer et les conditions déclenchant l'adaptation, par exemple

une dégradation de la bande passante disponible modifiera le flux de données à envoyer en débit ou en qualité.

L'intérêt de l'approche réside dans la facilité d'adapter l'application à de nouveaux besoins par extension des règles d'adaptation. Différentes expérimentations ont validé cette méthodologie en détaillant la construction d'applications adaptables, y compris réparties, telles que des applications multimédia ou des serveurs Web.

L'utilisation de composants permet ainsi d'exprimer clairement le rôle de chaque entité utilisée (interfaces et leurs instances) et ensuite les combiner. Ceci fournit une vue claire de l'application à mettre en œuvre, permettant de vérifier la cohérence de l'assemblage et des règles d'adaptation.

Néanmoins, cette approche nécessite de se conformer à un modèle de programmation dédié, et de se reposer sur une plate-forme d'exécution spécifique. Enfin, le coût de l'adaptation et son impact sur l'exécution restent à analyser.

Synthèse

Chacun de ces intergiciels génériques retient une unique abstraction pour proposer une définition générale d'un intergiciel, qui peut ensuite être déclinée pour plusieurs modèles de répartition. Ainsi, UIC propose des interfaces abstraites formant un squelette d'intergiciel, ACT identifie un intergiciel à une pile protocolaire tandis que Quarterware retient l'image de composants minimaux combinables. Enfin, Jonathan réifie le mécanisme de liaison et l'étend à l'ensemble de l'intergiciel.

Cette abstraction centrale sert de tronc à l'architecture de l'intergiciel, autour duquel viennent se greffer plusieurs entités annexes. La possibilité de déploiement d'un modèle de répartition se fait donc sous une contrainte unique : l'adéquation ou la souplesse d'adaptation de cette abstraction vis à vis de besoins particuliers. Cependant, représenter une large classe de besoins à l'aide d'une unique abstraction semble illusoire, de sorte que de nombreux autres choix de conception restent masqués, ou implicites. L'architecture proposée n'est donc que partiellement décrite.

Un second élément, plus pragmatique, vient limiter les résultats obtenus par ces projets : l'adaptation doit pouvoir être effectuée par le développeur d'applications réparties, et non pas le concepteur de l'intergiciel. La mise à disposition d'un guide méthodologique facilitant l'adaptation de l'intergiciel générique est donc nécessaire. Nous remarquons cependant qu'aucun de ces projets ne fournit d'éléments complets. L'approche composant fournit un tel guide, mais sa mise en œuvre reste coûteuse, et non encore testée dans un cadre temps réel réparti embarqué.

Une dernière limite forte est la faible réutilisation de code permise. Ainsi, même si l'abstraction retenue se révèle intéressante, l'utilisateur doit réécrire une forte portion de code. Ce travail reste donc relativement coûteux, et source d'erreurs.

2.3.3 Intergiciels schizophrènes

L'architecture d'intergiciel schizophrène a été introduite en réponse aux besoins d'interopérabilité entre intergiciels hétérogènes, par exemple entre CORBA et Ada/DSA.

La problématique du "M2M" (*Middleware to Middleware*) [Baker, 2001] est croissante du fait de la nécessité de faire interagir ou d'intégrer des éléments d'applications issus d'applications bâties sur des technologiques différentes (par exemple CORBA et services Web). Ce besoin en une nouvelle architecture d'intergiciel a été défini par les travaux de Laurent PAUTET [Pautet, 2001] et concrétisé par les travaux de Thomas QUINOT [Quinot, 2003] réalisés au sein de l'équipe ASTRE où se sont déroulés

une partie de nos travaux de thèse. Les travaux de ce présent mémoire utilisent ainsi certains de leurs résultats.

Nous présentons dans un premier temps l'architecture d'intergiciel schizophrène, puis PolyORB, son implantation.

Architecture d'un intergiciel schizophrène

Un intergiciel schizophrène étend l'architecture d'intergiciel générique. Il adjoint à la notion de personnalités définies par un intergiciel générique une couche neutre coordonnant plusieurs personnalités sur un seul et même nœud.

À la différence d'un intergiciel générique qui fournit un cœur neutre abstrait, donc non implanté, la couche neutre dispose d'une réalisation concrète qui favorise la mise en commun de code entre les différentes personnalités, et permet en particulier un échange de requêtes entre les différentes personnalités transparent du point de vue de l'application. On peut ainsi définir un intergiciel schizophrène comme suit :

Définition 2.3.4. *Un intergiciel schizophrène permet la cohabitation simultanée de plusieurs personnalités au sein d'une même instance, et met en place un mécanisme efficace d'interactions entre ses personnalités.*

Cette architecture a été définie pour la première fois dans [Quinot *et al.*, 2001], nous en rappelons ici les principaux éléments. Elle distingue les personnalités applicatives des personnalités protocolaires.

Les personnalités applicatives forment la couche d'adaptation entre les composants applicatifs et l'intergiciel grâce à une interface dédiée (par exemple l'interface normalisée de CORBA), ou un générateur de code (un compilateur Ada 95 compatible avec l'annexe des systèmes distribuées Ada/DSA). Les personnalités applicatives enregistrent les composants applicatifs auprès de la couche neutre et interagissent avec elles pour permettre l'échange de requêtes.

Les personnalités protocolaires prennent en charge la projection entre les requêtes neutres (représentant les interactions entre les entités de l'application) vers les messages échangés au travers d'un canal de communication grâce à un protocole défini. Les requêtes à transmettre peuvent provenir d'entités applicatives (et sont transmises par la couche neutre et une personnalité applicative) ou depuis un autre nœud de l'application répartie. Elles peuvent aussi être reçues depuis une autre personnalité protocolaire. Dans ce cas, le nœud agit comme une passerelle dynamique entre deux nœuds.

La couche Neutre (Neutral Core Middleware, NCM) est la couche de médiation entre les personnalités applicatives et protocolaires. Elle prend en charge la gestion des ressources pour l'exécution, et fournit les abstractions nécessaires au passage de requêtes transparent entre les personnalités. La couche neutre est indépendante et neutre vis à vis des personnalités. Ceci permet la sélection et l'interaction de toute combinaison de personnalités applicatives ou protocolaires.

Les personnalités mettent en œuvre une facette spécifique d'un modèle de répartition. La couche neutre permet la présence simultanée et l'interaction de plusieurs personnalités applicatives et protocolaires au sein de la même instance de l'intergiciel, d'où sa nature *"schizophrène"*.

La figure 2.1 présente les interactions possibles entre différentes personnalités au sein d'un intergiciel schizophrène.

Cette architecture sépare un intergiciel en trois modules principaux : une partie protocolaire, une applicative et un cœur neutre. Ceci fournit un premier découplage entre les modules d'un intergiciel. Les personnalités fournissent des mécanismes spécifiques

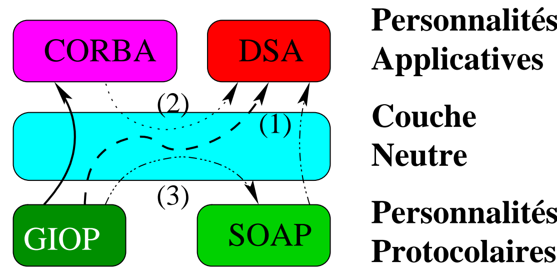


FIG. 2.1 – Interactions entre les personnalités d'un intergiciel schizophrène

à la sémantique d'un modèle de répartition, pour une spécification donnée. Cependant, ces fonctions sont conceptuellement proches, et peuvent être définies comme des instances de services génériques.

Dans ce contexte, l'architecture schizophrène définit des services génériques représentant les fonctions clés d'un intergiciel. Cette phase de définition fait suite à un travail d'identification issu de l'expérience acquise par les auteurs lors de l'implantation des intergiciels GLADE [Tardieu, 1999] et AdaBroker [Azavant *et al.*, 1999].

Les services génériques se concentrent sur la réalisation d'une interaction entre deux nœuds d'une application répartie. Les instances de ces services peuvent être combinées pour construire la couche neutre, ou un personnalité protocolaire ou applicative.

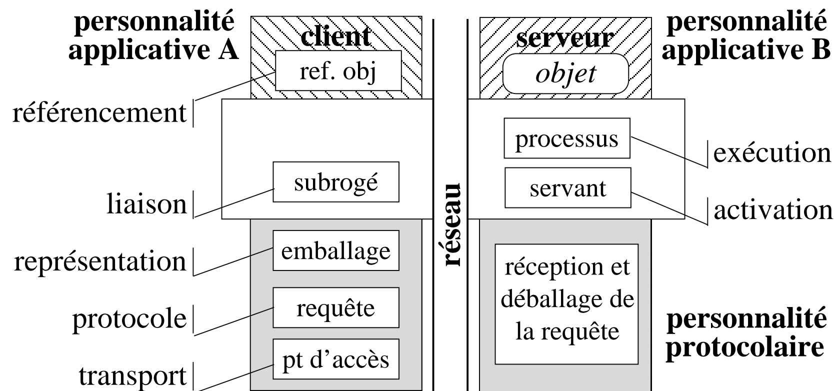


FIG. 2.2 – Services d'un intergiciel schizophrène

- Le service d'**Adressage** fournit à chaque entité applicative un identifiant unique la référençant à travers l'application répartie ;
- Le service de **Liaison** établit et maintient les associations entre les entités distantes, et les ressources permettant cette interaction (socket, pile protocolaire) ;
- La fonction de **Représentation** traduit en une donnée propre à être transmise à travers le réseaux, et interprétée par des nœuds applicatifs hétérogènes ;
- Le service de **Protocole** implante un protocole pour la transmission de requêtes ;
- Le mécanisme de **Transport** gère les canaux de communication permettant la transmission des messages ;
- Le service d'**Activation** assure qu'une entité applicative concrète est disponible pour traiter la requête ;
- Le service d'**Exécution** affecte des ressource pour l'exécution de la requête

(tâches, contexte d'exécution, etc).

La figure 2.2 illustre la coopération entre les services d'un intergiciel schizophrène lors de la transmission d'une requête depuis une personnalité protocolaire vers une autre, entre deux nœuds utilisant un protocole commun.

Le client (personnalité applicative $'A'$) obtient une référence sur l'entité distante grâce au service *d'adressage* ; la couche neutre crée un objet de liaison (service de *liaison*) : une passerelle dynamique vers l'entité distante à travers laquelle le client peut communiquer. Un message est construit à partir de la requête (respectivement par les services de *représentation* et de *protocole*). Ce message est ensuite envoyé au nœud distant par le service de *transport*. Lorsqu'il est reçu, l'intergiciel sur le nœud distant s'assure qu'une entité concrète est disponible pour exécuter la requête (service d'*activation*) et lui affecte des ressources pour l'exécution (*exécution*) conduisant à l'exécution du code de l'entité enregistrée par la personnalité applicative $'B'$. La réponse éventuelle suivra le chemin inverse.

PolyORB : un intergiciel schizophrène

PolyORB est l'implantation d'un intergiciel schizophrène réalisée dans le cadre des travaux de thèse de Thomas QUINOT. Nous rappelons ici l'avancement de PolyORB au moment où nos travaux de thèse ont débuté, en Septembre 2002.

Ces travaux ont permis de valider le concept d'intergiciel schizophrène en fournissant un intergiciel complet et conforme à des spécifications industrielles et validées sur différents exemples significatifs.

PolyORB propose les personnalités protocolaires CORBA, Ada/DSA (implantant l'annexe des systèmes répartis du langage Ada 95), MOMA (un intergiciel orienté message auquel nous avons participé) et les personnalités protocolaires GIOP/IIOP et SOAP.

PolyORB permet à l'utilisateur de sélectionner le profil de tâches à utiliser : mono-tâche, profil Ravenscar ou profil complet. Différents mécanismes permettent ponctuellement de sélectionner certains paramètres. Enfin, PolyORB propose la configuration de certains mécanismes de concurrence en implantant des patrons de conception usuels pour l'exécution de requêtes : "tâche par requête", "pool de tâches", "tâche par connexion", "mono-tâche".

Une analyse du code source de PolyORB a montré que la couche neutre autorise une forte réutilisation du code source. La couche neutre représente environ 66% pour une configuration incluant les personnalités CORBA et GIOP.

Enfin, les performances mesurées montrent un surcoût lié à l'architecture et à certaines fonctions coûteuses en temps. Ce surcoût peut cependant être réduit en optimisant certaines constructions très localisées et fortement pénalisantes. En particulier, PolyORB était avant tout défini comme une preuve de faisabilité ; la réalisation était orientée vers les fonctionnalités à fournir plutôt que leur efficacité.

PolyORB fournit une vue précise des différentes fonctions formant un intergiciel. Il en fournit ainsi une vue "canonique" qui peut ensuite être adaptée au modèle de répartition retenu. Le fort taux de réutilisation de code démontre la pertinence de l'approche choisie. Le manque de points d'adaptation de l'implantation pénalise néanmoins son utilisation dans des contextes hétérogènes.

Le but premier de l'implantation réalisée est de démontrer la faisabilité d'une interopérabilité dynamique entre modèles de répartition. Les performances et le déterminisme étaient donc des critères annexes.

Néanmoins, PolyORB dispose initialement de patrons de conception (tels que `Annotations`) et d'une bibliothèque de concurrence adaptée au temps réel basée sur le profil Ravenscar. Ces éléments fournissent une première base pour construire un intergiciel temps réel. Il faut cependant poursuivre cet effort pour assurer que l'ensemble des éléments d'intergiciel, et leur assemblage garantit de bonnes propriétés pour les systèmes TR^2E. En particulier, le déterminisme n'est pas garanti pour certaines constructions fréquemment utilisées (telles que les exceptions, l'allocation mémoire).

2.3.4 Synthèse

L'étude de différentes architectures mises en avant par les intergiciels configurables, génériques et schizophrène montrent qu'il existe plusieurs approches complémentaires pour définir l'architecture d'un intergiciel. Ces approches serviront de fondation à notre solution.

Les intergiciels configurables proposent le paramétrage de leurs mécanismes internes. Ils reposent sur une architecture monolithique autour de laquelle s'articulent quelques modules éventuellement interchangeables (cas de TAO). Ceci permet de contrôler précisément certains mécanismes de l'intergiciel. Ces architectures sont dédiées à un modèle de répartition et à une spécification. Il est impossible de les faire évoluer sans remettre en cause toute l'architecture. Néanmoins, ces architectures démontrent par l'exemple qu'elles peuvent supporter des contraintes temps réel et embarqué, et fournissent une solution directement utilisable dans l'industrie.

Les intergiciels génériques offrent au développeur la possibilité de construire et combiner les modules dont il a exactement besoin. Ainsi, les intergiciels génériques se concentrent sur la définition des mécanismes internes formant le cœur de l'architecture de l'intergiciel suivant deux axes :

Sémantique L'utilisateur peut instancier ou dériver de nouvelles fonctions, spécifiques au modèle de répartition qu'il souhaite utiliser (cas de Jonathan). Cette approche permet à l'utilisateur de définir précisément la sémantique de chacune des fonctions mises en œuvre, et permet aussi une meilleure compréhension des mécanismes internes de l'intergiciel.

Canonisation L'utilisateur peut réutiliser différents modules, les personnaliser puis les assembler pour former un nouvel intergiciel, comme le montre Quarterware.

Cependant, on constate que le coût de personnalisation de ces plates-formes est élevé. De fait, elles sont peu utilisées dans l'industrie et n'ont jamais été utilisées ni même expérimentées dans un contexte temps réel.

L'architecture schizophrène réalise la synthèse de ces deux architectures et propose une architecture configurable et générique. Les différents travaux réalisés au commencement de notre thèse démontrent une forte capacité à résoudre plusieurs des problèmes non résolus par les architectures configurables et génériques.

Nos travaux se placeront dans le cadre du projet PolyORB, et viseront à étendre son architecture et ses modules de manière à renforcer son déterminisme, contrôler son empreinte mémoire et garantir certaines de ses propriétés. Cette nouvelle version, *"PolyORBv2"*, vise à pousser plus loin la réflexion sur l'architecture des intergiciels pour intégrer les problématiques des systèmes TR^2E.

2.4 Synthèse et axes de travail

Les sections précédentes nous ont permis de positionner l'état de l'art des intergiciels et des méthodologies de conception pour les applications temps réel réparties embarquées. Dans cette section, nous tirons les leçons de cet état de l'art, et posons les éléments de réponse que nous développons dans le reste du mémoire.

2.4.1 Critique des architectures existantes

Les différents projets que nous avons étudiés démontrent qu'il est possible de construire un intergiciel répondant à un sous-ensemble particulier de besoins des applications TR^2E. Cependant, ils s'intéressent tous à un problème particulier : construire un intergiciel répondant à un besoin ou trop généraliste ou trop particulier.

Nous notons que ces intergiciels se sont développés suivant un cercle vicieux, cherchant à mettre en avant une facette particulière au détriment des autres :

1. Les intergiciels développés suivant une approche "ingénierie temps réel" s'intéressent à leur empreinte mémoire, leur déterminisme. Les mécanismes de répartition proposés sont pauvres.
2. Les intergiciels suivant une approche "génie logiciel" (dont font partie les intergiciels configurables et génériques) privilégient une architecture élégante, basée sur l'orienté objet, qui permet de définir des mécanismes pour l'adaptation des fonctions de l'intergiciel au détriment des possibilités d'analyse de l'architecture.
3. L'utilisation de méthodes de conception de haut niveau ou les méthodes formelle facilitent la conception d'applications réparties en facilitant l'analyse d'un modèle de l'application. Néanmoins, la distance entre le modèle et l'implantation limite considérablement la réutilisation des résultats.

Ces différents éléments nous amènent à remarquer que les projets étudiés ont avant tout considéré l'intergiciel comme étant un outil parmi d'autres. La répartition et ses spécificités sont devenues annexes devant des problèmes tels que l'empreinte mémoire, les politiques de QdS, etc. Par ailleurs, cette spécialisation de l'architecture de l'intergiciel réduit les possibilités de réutilisation d'un intergiciel dans un autre contexte : nouvelle cible, nouvelle application, etc.

2.4.2 "Crise de l'Intergiciel"

Dès lors, on peut s'interroger sur le statut de l'"Ingénierie des Intergiciels" en tant que discipline rattachée au Génie Logiciel.

L'analyse précédente nous amène à une constatation qui rejoint [Gibbs, 1991] où les auteurs relèvent l'existence d'une *"crise chronique du logiciel"*, appliquée au domaine de la répartition : les intergiciels rencontrent les même difficultés à répondre aux très nombreuses contraintes et demandes exercées par leurs utilisateurs que tout autre logiciel. Ceci rend leur utilisation difficile et coûteuse. De fait, il existe une multiplicité d'architectures, répondant chacune à un ensemble spécifique de ces revendications.

Nous remarquons que *la difficulté principale n'est pas de construire un intergiciel pour les systèmes TR^2E, mais de construire un intergiciel qui puisse répondre aux nombreuses contraintes hétérogènes et antagonistes d'une large classe d'applications.*

Cette crise de *"l'Ingénierie de l'Intergiciel"* est un frein à l'utilisation des intergiciels pour le développement de systèmes TR^2E, et limite de ce fait l'exploitation des capacités de la répartition pour ces applications.

Une première explication, comme le notent certains industriels [RIS, 2002], est le manque de poids (notamment économique) des acteurs de l'embarqué et des systèmes critiques dans le processus de définition et de construction d'un intergiciel pour des systèmes temps réel critiques. Ceci les contraint bien souvent à développer en interne des solutions spécifiques.

Une seconde explication, technique cette fois, ressort de notre analyse : la prise en compte de besoins spécifiques lors de la définition et de la construction de l'intergiciel détourne l'attention vers des problèmes d'implantation, alors qu'elle devrait plutôt se porter sur les fondements de l'intergiciel (modèle de répartition, services).

2.4.3 Éléments de solution et méthodologie

Nous nous proposons de replacer l'intergiciel et son architecture au cœur du problème à résoudre (figure 2.3). Notre revendication principale est que la construction d'un système TR^2E ne peut se concevoir qu'en fournissant des mécanismes d'adaptation de l'intergiciel et un processus d'analyse fine de ses propriétés.

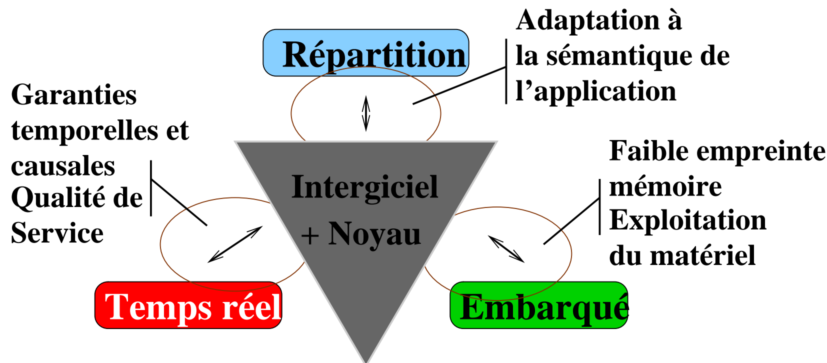


FIG. 2.3 – Solution apportée par l'intergiciel aux systèmes TR^2E

L'intergiciel, adossé à un noyau temps réel, est rendu compatible avec les besoins de l'application et ses méthodologies de conception.

Ainsi, notre objectif premier est de fournir une architecture d'intergiciel qui puisse répondre à des besoins hétérogènes en permettant l'adaptation de certains de ces modules.

Notons que cet objectif sous-tend un ensemble de problèmes distincts, comme nous l'avons montré à la section 2.2.4. Il est donc nécessaire de les ordonner pour ensuite les résoudre de manière systématique.

Nous remarquons qu'une application répartie requiert un modèle de répartition qui lui permette d'exprimer les relations entre ses entités. Ce modèle est concrétisé par un assemblage de modules logiciels formant une instance d'intergiciel. Ces modules peuvent être fonctionnels (implantant un mécanisme normalisé) ou non-fonctionnels (garantissant certaines propriétés à l'exécution). Enfin, ils doivent être construits indépendamment les uns des autres pour permettre une large réutilisation.

Nous en déduisons un graphe de dépendances entre différents problèmes à résoudre, et proposons de parcourir ce graphe en profondeur et de fournir une solution

à chaque étape. Considérant la diversité des besoins, nous proposons les éléments de solution et de méthodologie suivants :

1. Définir une architecture d'intergiciel adaptable, utilisable par une large classe de modèles de répartitions (chapitre 3) ;
2. Montrer que notre architecture est vérifiable formellement pour des configurations réalistes de l'intergiciel (chapitre 4) ;
3. Proposer de nouvelles fonctions et mécanismes architecturaux permettant le support des sémantiques classiques rencontrées dans les différents projets que nous avons étudiés (chapitre 5). Ces éléments fourniront les briques de base permettant de construire un intergiciel adapté à l'application ;
4. Valider notre architecture sur des exemples concrets, issus de cas d'études (chapitres 6 et 7).

Les chapitres suivants détaillent ces éléments de solution, et leur adéquation avec les objectifs que nous nous sommes fixés.

3

Définition d'un intergiciel canonique

Sommaire

3.1 Leçons tirées sur les architectures d'intergiciels 55
3.1.1 Reformulation des besoins 56
3.1.2 Vers une architecture généralisée d'intergiciels 57
3.2 Architecture générale pour les intergiciels 57
3.2.1 Patron de conception *"Broker"* 57
3.2.2 Utilisation de l'architecture schizophrène 59
3.3 Réduction des services d'un intergiciel 61
3.3.1 Services fonctionnels de l'intergiciel 61
3.3.2 Réduction des mécanismes de contrôle d'un intergiciel . . 64
3.3.3 Synthèse 65
3.4 Architecture du μBroker 66
3.4.1 Définition fonctionnelle du μBroker 66
3.4.2 Définition comportementale du μBroker 69
3.4.3 Réalisation du μBroker 73
3.4.4 Mise en œuvre du patron pour la construction d'intergiciels 74
3.5 Conclusion 76

Ce chapitre présente la démarche que nous proposons pour la construction d'un intergiciel temps réel adaptable. Ce *"guide méthodologique"* détaille les différentes étapes utiles à sa conception et sa réalisation.

Nous présentons d'abord la méthodologie retenue, puis les éléments d'architecture définissant un intergiciel canonique. Nous montrons ensuite comment cet intergiciel peut supporter plusieurs mécanismes de gestion du parallélisme.

3.1 Leçons tirées sur les architectures d'intergiciels

Le chapitre précédent a fourni plusieurs éléments sur l'architecture des intergiciels, temps réel ou non, et sur leurs propriétés d'adaptation aux besoins de l'utilisateur. Dans cette section, nous analysons les besoins en intergiciels adaptables et introduisons la solution que nous proposons. Nous avons présenté certains de ces éléments dans [Hugues *et al.*, 2005b].

3.1.1 Reformulation des besoins

Nous notons deux familles de besoins, complémentaires, auxquels un intergiciel adaptable doit répondre ·

– *sémantiques* : plusieurs modèles de répartition existent, avec ses spécificités et caractéristiques propres. Considérant que chacun a son rôle pour une classe d'application, un intergiciel doit pouvoir en supporter plusieurs, éventuellement simultanément.
– *comportementaux* : l'intergiciel doit fournit les interfaces et mécanismes permettant l'adaptation précise de son comportement, de manière à répondre à des besoins en performance, ou tout autre besoin non-fonctionnel tel que la tolérance aux pannes.

Remarquons dans un premier temps que des solutions à ces différents besoins existent. Ainsi, CORBA peut être étendu et supporter plusieurs sémantiques d'échanges de données grâce à différentes extensions (Asynchronous Message Interface, COS Event, . . .). De même, chaque intergiciel fournit un jeu de politiques et de paramètres permettant l'adaptation de son comportement.

Par ailleurs, ce découplage n'est pas orthogonal. Par exemple, RT-CORBA nécessite explicitement un environnement multi-tâches. Néanmoins, notre opinion est que séparer ces deux éléments permet d'exprimer simplement deux facettes complémentaires de l'intergiciel pour ensuite les combiner. Les différentes expériences décrites dans la suite de ce mémoire confortent notre approche.

Les solutions que nous avons présentées dans notre état de l'art sont en général peu adaptables. Ajouter de nombreux mécanismes est un travail complexe, induisant un surcoût non négligeable lors de la conception de l'intergiciel, son déploiement, mais aussi sa maintenance.

Comme le note l'auteur de [Tanenbaum, 1995] : *"le modèle optimal (de système d'exploitation) n'est pas obtenu quand il ne reste rien à rajouter, mais quand tout ce qui n'était pas strictement indispensable a été supprimé"*.

Ainsi, l'ajout de nouvelles fonctionnalités s'effectue alors non plus en adaptant l'architecture, mais en l'étendant en lui ajoutant de nouvelles entités, et en les composant à celles existantes par exemple par dérivation et spécialisation (modèle objet), tissage d'aspects (programmation orientées aspects), assemblage de composants, etc. Ce faisant, on rend l'architecture évolutive, et précisément adaptable.

Cependant, il y a un risque à mélanger choix d'implantation et architecture d'un intergiciel. Ainsi, TAO a fait de l'utilisation des patrons de conception, et de l'orienté objet la clé de voûte de son implantation. Ceci rend l'ensemble extrêmement complexe. L'architecture est vue non plus comme un ensemble de fonctions interagissantes, mais comme plusieurs patrons coopérants. Cette architecture remplace ainsi le *"pourquoi"* par le *"comment"*, rendant difficile toute analyse de l'architecture.

Une autre constatation est que la prise en compte de la sémantique d'un modèle de répartition peut se faire avec peu de primitives, quelques opérations suffisent pour construire un intergiciel orienté messages ou basé sur un modèle RPC [1]. Le coût d'un intergiciel et sa valeur ajoutée proviennent alors essentiellement des mécanismes de configuration fournis à l'utilisateur.

De fait, la notion d'intergiciel disparaît, pour laisser place à la notion *"d'intergiciel adaptable"*, dont une instance correspond aux besoins exprimés par l'application.

[1] `http://www.javaworld.com/javaworld/jw-05-1998/jw-05-step.html` décrit comment implanter son propre MOM en un millier de lignes de code Java

3.1.2 Vers une architecture généralisée d'intergiciels

Ces différentes constatations nous amènent à définir une démarche pour la construction d'intergiciel, et d'intergiciels dédiés aux systèmes TR^2E.

Plusieurs modèles architecturaux, dont ODP [ODP, 1995], définissent une application à travers différentes vues complémentaires. L'assemblage de ces vues permet de construire l'application complète.

Nous retenons un modèle plus simple qui définit une application suivant deux axes orthogonaux : 1) contrôle de l'exécution et 2) calcul ou logique applicative.

Par extension au cas des intergiciels, nous cherchons dans ce chapitre à définir et séparer dans notre architecture ces deux aspects, en suivant une démarche de réduction poussée à l'extrême : toute fonction de l'intergiciel facultative est déplacée dans un module qui sera éventuellement utilisé pour construire l'intergiciel.

La séparation entre la *sémantique* et le *comportement* d'un intergiciel fournit un premier élément de réponse pour isoler les différents traits d'un intergiciel. Par ailleurs, l'architecture schizophrène fournit une vue précise des différentes fonctions d'un intergiciel, et donc sur les modules permettant de mettre en œuvre sa sémantique. Différentes politiques issues des intergiciels configurables fournissent des éléments sur la manière d'infléchir son comportement.

Considérant les larges besoins d'adaptabilité des intergiciels, nous nous fixons deux étapes pour la construction d'intergiciels adaptables :

1. Définition d'une architecture générale pour les intergiciels, qui puisse rendre compte d'une large classe de besoins ;
2. Rendre cette architecture minimale, et proposer un mécanisme pour l'ajout de nouvelles fonctionnalités.

Cette architecture sera validée sur des exemples significatifs.

3.2 Architecture générale pour les intergiciels

L'ingénierie des intergiciels a pour credo *"qu'une taille ne suffit pas"* (*"One size does not fit all"*) pour justifier les besoins en intergiciels adaptables, et la multiplicité des architectures qui y en découlent, comme nous l'avons illustré précédemment.

Cette constatation est certes légitime à un haut niveau d'analyse. Cependant, elle apparaît trop restrictive et pessimiste. Nous pensons qu'une architecture commune peut être définie par analyse des patrons de conception utilisés, et par synthèse des notions et concepts communs à de nombreux intergiciels existants. Cette architecture "pivot" formera un socle sur lequel construire plusieurs intergiciels, chacun répondant à une famille de besoin.

Nous analysons dans cette section la mise en œuvre du patron de conception *"Broker"* et l'architecture schizophrène pour la construction d'intergiciels adaptables. Ces deux patrons proposent une vue canonique d'une architecture d'intergiciel.

3.2.1 Patron de conception *"Broker"*

Le patron de conception *"Broker"* est un patron architectural qui fournit une vue synthétique d'un intergiciel [Buschmann *et al.*, 1996]. Tel que défini, son rôle est de *"[..] structurer les applications réparties à l'aide de composants découplés qui interagissent par invocations de services distants. Un composant courtier (*"Broker"*) prend*

en charge la coordination et la communication entre composants, tels que l'aiguillage de requêtes ou la transmission des résultats et des exceptions" [2].

La figure 3.1 fournit une vue des différents éléments formant ce patron. il s'agit d'un patron de conception architectural, il fournit une vue exhaustive des différents éléments qui entrent dans la construction d'un système réparti.

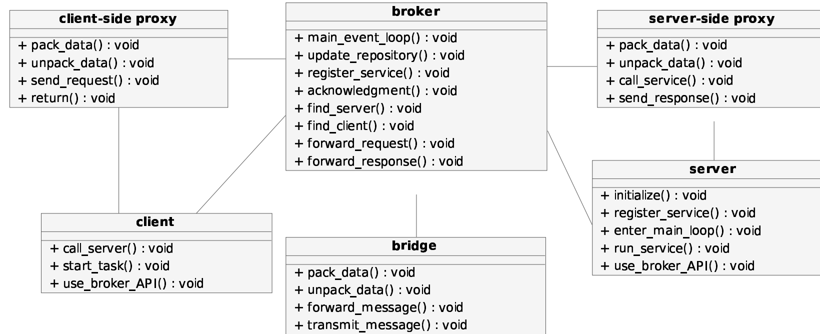


FIG. 3.1 – Patron de conception *"Broker"*

Il définit ainsi les modules *Broker*, *Proxy*, *Client*, *Serveur*, *Référentiel* et *Passerelle*. Ces objets interagissent de la façon suivante : les *Serveurs* s'enregistrent auprès du *Broker* grâce au *Référentiel*, et rendent des services accessibles par des *Clients* en publiant leurs interfaces. Les *Clients* accèdent aux *Serveurs* en échangeant des requêtes au travers du *Broker*, qui aiguille la requête jusqu'au *Serveur* destinataire, puis renvoie la réponse au *client*. Les modules *Proxy* et *Passerelle* gèrent les mécanismes de communication et assurent l'échange de données entre plates-formes hétérogènes.

Ces différents objets couvrent l'ensemble des mécanismes fondamentaux présents dans un intergiciel : pile protocolaire, représentation des données, allocation de ressources, échanges de messages, enregistrement de composants applicatifs, etc.

Dans [Hugues *et al.*, 2003b], nous analysons ce patron architectural, et constatons qu'il est présent dans de nombreux intergiciels. Ce patron sert ainsi de fil directeur pour les spécifications CORBA ou .Net. Par ailleurs, plusieurs intergiciels tels que ACT ou TAO font explicitement référence à ce patron pour décrire leur propre architecture.

Ainsi, ce patron est suffisamment général pour décrire plusieurs modèles de répartition, et rendre compte des aspects normatifs de spécifications industrielles.

Cependant, nous notons que chaque implantation spécialise et adapte ce patron à ses propres besoins. Ce patron ne permet donc pas de décrire à lui seul une *"famille"* d'intergiciels. Quelque soit l'intergiciel considéré, il faudra passer par une phase d'analyse pour définir et combiner les modules permettant la construction d'un intergiciel.

Inversement, toute implantation d'un intergiciel adaptera ces différents modules. L'architecture proposée par ce patron de conception n'est donc pas suffisamment générale pour servir de base solide pour la définition d'une architecture pivot. En particulier, peu de modules seront mutualisés par différentes instances.

Il apparaît donc plus intéressant de s'intéresser non pas à un modèle mais à une architecture d'intergiciel favorisant la mutualisation de modules. Nous discutons cette option dans la section qui suit.

[2] Le texte original est *" [..] (to) structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions."*

3.2.2 Utilisation de l'architecture schizophrène

Nous avons discuté au chapitre précédent différentes architectures d'intergiciel et retenu l'architecture schizophrène comme architecture candidate pour nos travaux. Dans cette section, nous montrons comment cette architecture peut être facilement adaptée pour supporter une nouvelle famille d'intergiciel.

Nous avons fourni précédemment une première description des services fournis par un intergiciel schizophrène. Nous avons montré dans [Hugues *et al.*, 2002] que cette architecture permettait le support des modèles de répartition objets répartis, RPC et intergiciel orienté messages. Ce support passe par l'utilisation et la spécialisation de plusieurs services fondamentaux proposés par l'architecture schizophrène.

Dans la suite, nous montrons comment l'architecture schizophrène peut servir de support pour le prototypage rapide et la construction d'intergiciels. Nous montrons en particulier comment l'architecture initiale peut servir de support d'intergiciels basé sur un modèle de répartition autre que le modèle à objets répartis proposé initialement. Nous rappelons les principaux éléments de cette construction. Cet exercice d'instanciation a été publié dans [Hugues *et al.*, 2003a].

Définition d'une personnalité "MOM" pour PolyORB

Les intergiciels orienté messages (*"Message Oriented Middleware", MOM*) fournissent des mécanismes pour le passage de messages asynchrone entre les différents nœuds de l'application, à la façon des courriers électroniques ou des forums de discussion. Ils fournissent une large classe de services allant des politiques de livraison des messages, la persistance, la tolérance aux pannes, etc. Les intergiciels orientés messages sont ainsi utilisés dans des systèmes d'information à large échelle [Banavar *et al.*, 1999], mais aussi les systèmes mobiles [Kaddour & Pautet, 2003].

Nous avons défini la personnalité applicative MOMA (*"Message Oriented Middleware for Ada"*), il s'agit d'une adaptation pour le langage Ada 95 des spécifications du service JMS (*"Java Message Service"*) [Sun Microsystems, Inc., 1999].

JMS propose une interface standardisée pour les modèles de passage de messages classiques *Point-à-Point* et *Publication/Abonnement*, ainsi qu'une modèle complet pour un MOM pour les plates-formes Java. Il définit précisément les différentes étapes du cycle de vie d'un message lors des phases de création, émission, réception, lecture et destruction. En revanche, JMS ne définit aucun mécanisme pour les couches basses de l'intergiciel telles que les protocoles, la représentation des données.

Ces mécanismes sont pris en charge par le *"Fournisseur JMS"*, une infrastructure pour la répartition au dessus de laquelle est implantée l'API. Ce découplage permet de séparer clairement deux modules, et permet la construction d'une large classe de fournisseurs, adaptés à chaque besoin. En contrepartie, cela limite grandement l'interopérabilité entre implantations différentes.

JMS est construit autour de plusieurs objets collaborant à l'échange de messages : producteurs, consommateurs, files et objets de configuration : connexions, sessions. Leurs projections vers Ada 95, supportant le paradigme objet est immédiat.

Nous nous intéressons à la projection de ces entités sur l'architecture de PolyORB, et la définition des personnalités requises.

Projection des entités JMS vers PolyORB

JMS fournit les primitives pour permettre aux clients d'interagir avec des files de messages : primitives pour poster et recevoir des messages par exemple. Un fournisseur

prend en charge les mécanismes de répartition, donc les communications et l'implantation des files de messages. Un fournisseur JMS est donc la combinaison de deux personnalités et de la couche neutre :

– *personnalité applicative* implante les objets MOMA (producteurs, consommateurs, ...) et l'interface des client ;
– *personnalité protocolaire* prend en charge la transmission des données.

Personnalités applicative

Les interactions entre un client et une file de messages sont similaires à des appels de méthodes sur des objets distants. Nous avons donc définis une personnalité applicative fournissant l'échange de message entre nœuds suivant un patron de conception *"servants"* : les clients invoquent les méthodes de servants spécifiques pour échanger leurs messages. Ces servants sont enregistrés auprès de l'adaptateur d'objets de PolyORB et accessible à distance.

Personnalité protocolaire

Nous nous sommes intéressés aux fonctionnalités d'un MOM d'un point de vue applicatif. Pour compléter sa réalisation, nous avons besoin d'une personnalité protocolaire qui permette l'échange de messages entre un client et une file de messages. Chacune des personnalités protocolaires permet de réaliser cette tâche. Nous avons ainsi pu réutiliser la personnalité GIOP pour tester notre implantation.

Nous avons pu construire un MOM au dessus de la couche neutre de PolyORB et fournir ainsi un sous-ensemble fonctionnel des primitives habituelles d'un MOM. La couche neutre et les personnalités existantes ont fourni un support direct des mécanismes pour la répartition allant du protocole à l'échange des requêtes jusqu'à l'activation des servants. Ainsi, nous avons pu nous concentrer sur la logique applicative du MOM et fournir une implantation des différents servants.

Cette analyse montre que l'architecture schizophrène se prête à la mise au point rapide de nouvelles personnalités, permettant ainsi le support de plusieurs modèles de répartition. Nous avons ainsi réalisé une personnalité "MOM", qui vient compléter les personnalités CORBA (qui supporte un modèle DOC) et Ada/DSA (RPC et DOC).

Parallèlement, les travaux de DEA de Thomas VERGNAUD [Vergnaud, 2003], que nous avons co-encadrés, ont permis de conforter cette démarche en proposant un prototype d'une personnalité AWS [Obry, 2000], permettant la construction d'applications pour le Web : serveurs applicatifs, applications SOAP, etc. Enfin, dans [Pautet & Kordon, 2004], les auteurs montrent comment déployer l'architecture schizophrène pour faciliter l'interopérabilité entre des composants applicatifs hétérogènes par construction de passerelles dynamiques.

Contrairement au patron *"Broker"*, le support d'un nouveau modèle de répartition par l'architecture schizophrène se fait par extension de certaines fonctions. Ceci facilite grandement le travail de conception. Les différentes mesures faites montrent que le taux de réutilisation de code est supérieur à 60%. Ceci démontre la facilité d'adaptation de cette architecture.

Nous retenons ainsi comme premier résultat que l'architecture schizophrène fournit une architecture à même d'être adaptée à plusieurs modèles de répartition.

3.3 Réduction des services d'un intergiciel

L'étude effectuée à la section précédente est à “gros grain”, et montre comment l'architecture schizophrène peut être utilisée pour construire à faible coût plusieurs types d'intergiciels. Notre objectif est de tirer partie de ces différents résultats pour définir précisément les éléments constitutifs d'une architecture générale d'intergiciel.

Une analyse des services et leur implantation dans PolyORB montre que l'on peut les réduire à des entités simples, représentables par des structures algorithmiques classiques, listées notamment dans [Cormen *et al.*, 2002] ou encore les patrons de conception définis dans [Gamma *et al.*, 1994]. Nous détaillons cette analyse dans la section qui suit.

3.3.1 Services fonctionnels de l'intergiciel

Dans cette première section, nous nous intéressons à la réduction des services fonctionnels de l'intergiciel, tels qu'ils sont définis par l'architecture schizophrène.

Adressage

La fonction d'adressage attribue à chaque entité de l'application une référence unique la désignant sans ambiguïté. Cette référence peut ensuite être transmise à d'autres nœuds pour qu'ils puissent contacter l'entité ainsi référencée.

La fonction d'adressage combine plusieurs informations pour construire une référence : identifiant local d'un objet ; un ensemble d'identifiants représentant les différents canaux de communication permettant de le joindre (un profil de liaison) ; des paramètres de qualité de service, etc.

Par ailleurs, cette référence une fois construite peut être transformée sous forme de chaîne de caractères (`IOR` CORBA, `corbaloc` `URI`, etc), puis échangée.

Notons que la mise en œuvre d'un tel service se fait en utilisant plusieurs fabriques d'objets, auprès desquelles les constructeurs de référence pour un profil donné s'enregistrent. De même, la transformation d'une référence se fait par projection d'une référence vers une certaine représentation textuelle. Cette fonction de projection est elle aussi enregistrée auprès d'une fabrique.

Bien que l'implantation puisse se révéler sophistiquée pour permettre l'agrégation de plusieurs sources d'informations, il s'agit conceptuellement d'un module simple. Ainsi, ce service utilise intensivement le patron de conception *“Fabrique d'objets”* pour l'enregistrement de fonction de rappel (*“call-backs”*).

La seule contrainte d'intégrité à prendre en compte est une exclusion mutuelle sur un ou plusieurs de ces modules pour éviter tout conflit dans les références attribuées.

Liaison

La fonction de liaison reprend et étend les mécanismes de liaison définit par Jonathan. Son rôle est de créer une structure locale, l'objet de liaison, qui représente l'objet désigné par une référence, et mettre en place les mécanismes permettant d'interagir avec lui. Ainsi, un objet de liaison est un *subrogé* de l'objet réel, qui lui prend en charge le transfert des invocations.

Lorsque la référence renvoie à un objet local au nœud, l'objet de liaison est l'entité encapsulant le code de l'utilisateur. Lorsque la référence pointe vers un objet distant,

l'objet de liaison regroupe l'ensemble des mécanismes permettant le dialogue avec l'objet distant : pile protocolaire, fonction de représentation, connexion, etc.

De manière analogue à la fonction d'adressage, la fonction de liaison nécessite plusieurs fabriques d'objets, et une fonction de choix qui va choisir parmi plusieurs fabriques celle qui sera utilisée pour concrétiser une liaison. Cette fonction de choix utilise des informations de configuration et de contexte pour choisir un mécanisme.

Ainsi, ce service utilise lui aussi intensivement le patron de conception *"Fabrique d'objets"*. Ses contraintes sont les mêmes que pour la fonction d'adressage.

Représentation

La fonction de représentation convertit une donnée D d'un type T du modèle de données locales en un message conforme à la représentation imposée par le protocole de communication utilisé. La fonction de représentation définit, pour tout type, une transformation injective projetant toute valeur du type sur une valeur d'un type du modèle de la personnalité protocolaire.

$$< D >_T \xmapsto{Representation} msg$$

Cette transformation peut s'effectuer avec une perte d'information : le message résultant msg doit contenir suffisamment d'informations pour reconstituer la valeur D connaissant T et msg, mais pas nécessairement pour reconstituer $< D >_T$ en connaissant seulement msg. Ainsi, dans la représentation CDR de CORBA, une suite de quatre octets peut correspondre à un entier long ou bien à deux entiers courts.

La transformation réciproque de la fonction de représentation doit donc prendre non seulement msg mais aussi le type original T en entrée pour reconstituer cette information perdue lors de la transmission. La personnalité applicative doit fournir *a priori* à la fonction de représentation le type des données qu'il souhaite recevoir. Cette information fait partie de l'interface de l'objet destinataire.

Soit $< \cdot >_T$ un conteneur vide possédant l'information de type T, nous définissons la transformation inverse par :

$$(msg, < \cdot >_T) \xmapsto{Representation^{-1}} < D >_T$$

Il s'agit d'un ensemble de fonctions (au sens mathématique), assimilables au patron *"Filtres"* : une donnée est transformée en une autre suivant un processus déterministe.

Protocole

La fonction de protocole coordonne le déroulement d'une invocation distante : à partir d'une demande d'invocation, un message est préparé puis émis. L'intergiciel client est mis en attente d'une réponse si nécessaire ; lorsque celle-ci est reçue, elle est retraduite sous forme neutre et signalée à la couche neutre, qui la transmet à la personnalité applicative.

Sur un serveur, à la réception d'une demande d'invocation, l'identité de l'objet et de la méthode concernés sont extraites, ainsi que les paramètres de l'appel. Une requête sous forme neutre est construite et confiée à la couche neutre, afin que celle-ci la traite grâce à un servant local ou la retransmette vers un objet de liaison lorsqu'une passerelle est établie. Une fois la requête traitée, la couche neutre signale à la couche protocolaire que la réponse éventuelle peut être retournée à l'intergiciel client.

La fonction de protocole se charge donc de la transformation d'une requête en message et de la transformation réciproque. Dans ce dernier cas, l'adaptateur d'objet lui fournit les informations nécessaires au déballage du message et notamment le conteneur typé $< \cdot >_T$ attendu par la fonction de représentation.

On peut représenter le protocole par un automate plus ou moins évolué. En première approche, le protocole ne peut pas créer d'interférence avec d'autres modules, les instances étant indépendantes.

Transport

La fonction de transport consiste à transférer une information d'un point à un autre et offre deux abstractions principales : les points d'accès du service de transport et les points de terminaison du service de transport. Les points d'accès du service de transport représentent les entités créées par le système d'exploitation à la demande de l'intergiciel afin de recevoir des connexions de la part de correspondants distants à travers un réseau de communication. Les points de terminaison du service de transport représentent les entités du système au moyen desquelles les données sont échangées. La connexion établie, les deux intergiciels peuvent dialoguer par échange de données au moyen des points de terminaison ainsi créés.

Les points d'accès et points de terminaison du service de transport représentent des entités permettant à un intergiciel d'interagir avec le monde extérieur. Les instants où des événements se produisent sur ces entités et requièrent l'attention de l'intergiciel ne sont pas connus *a priori*. Ces objets constituent donc des sources d'événements asynchrones qui doivent être régulièrement scrutées par l'intergiciel afin que ces événements externes soient pris en compte.

Ce service prend appui sur une bibliothèque existante, telles que les primitives d'entrées/sorties du système d'exécution ou un COTS. Elle fournit en général deux primitives `Send` et `Receive`, auxquelles s'ajoutent des mécanismes de configuration.

Ce service peut donc être ramené aux patrons *"Pipe"* pour la partie écriture/lecture de données, et *"Interrupt"* [Douglass, 2002] pour la gestion de la réception asynchrone de données. Notons par ailleurs que ce service impose une certaine sémantique aux autres services et modules à prendre en compte : orienté connexion, absence de réponse, etc.

Activation

Le service d'activation prend en charge l'association entre une entité et la requête demandant à interagir avec celle-ci. Cette entité va prendre en charge l'exécution de la requête. Le service a ainsi pour but de retrouver l'entité correspondant parmi celles enregistrées par l'utilisateur. Il peut éventuellement la créer suivant un mode défini par l'utilisateur (mise en place d'une session pour traiter la requête pour les services Web, utilisation des `ServantManagers` de CORBA, etc).

Ainsi, cette association peut être réalisée suivant plusieurs modes (statiques, dynamiques) et avec plusieurs niveaux de configuration (par exemple grâce aux multiples politiques du `POA` de CORBA).

Aussi, ce service peut être ramené à une *"Fabrique d'objets"* ou à un *"Dictionnaire"*. Sa caractérisation exacte dépend de l'implantation souhaitée. Notons que ces deux patrons partagent une interface proche : ils retournent une entité correspondant à certains critères.

Exécution

Un appel de méthode correspondant à une requête reçue doit être affecté à une tâche, et orienté vers le sous-programme applicatif approprié. La tâche utilisée peut être soit une tâche prêtée temporairement à l'intergiciel par l'application, soit une tâche propre de l'intergiciel. Dans ce dernier cas, la tâche peut être banalisée, ou bien dédiée à l'exécution des requêtes liées à une entité.

L'ordonnanceur de la fonction d'exécution se charge d'affecter une tâche à une requête. Les différentes variations de son comportement se font par délégation d'une partie de ses fonctionnalités à des objets déterminant la politique de parallélisme de l'intergiciel. La détermination du sous-programme applicatif correspondant est réalisée sous le contrôle de la personnalité applicative, en fonction de l'interface que lui présentent les objets applicatifs.

3.3.2 Réduction des mécanismes de contrôle d'un intergiciel

Nous cherchons dans cette section à caractériser la boucle de contrôle d'un intergiciel. Pour ce faire, nous proposons de reprendre les fonctions de base de l'intergiciel, et de les réduire à l'extrême.

Cycle de contrôle d'un processeur

Les processeurs modernes fonctionnent tous suivant un cycle de von Neumann, *"Fetch, Decode, Execute"* qui précise les différentes phases d'exécution d'une instruction par le processeur. Il s'agit d'une boucle infinie représentée par l'algorithme 1 :

Algorithm 1 Cycle Fetch/Execute

1: $PC \leftarrow PC_0$ {Initialisation du PC}
2: **loop**
3: FETCH *instruction [PC]*
4: DECODE *instruction [PC]*
5: EXECUTE *instruction [PC]*,
$PC \leftarrow f(instruction[PC])$ {Recalcul du PC}
6: **end loop**

PC représente le registre `Program Counter` qui contient la position de l'instruction courante. *instruction [PC]* est une instruction élémentaire de la machine. *PC* est mis à jour après chaque instruction exécutée. Il est recalculé après une instruction de saut, incrémenté après une instruction classique.

Cycle de contrôle d'un intergiciel

Nous remarquons qu'un intergiciel est un mandataire, chargé de transmettre une requête d'un service de l'intergiciel à un autre, éventuellement au travers du réseau, et de prendre en charge l'ensemble des mécanismes sous-jacents pour assurer ces fonctions. Nous avons listé à la section 2.3.3 l'enchaînement des différentes actions nécessaires.

De sorte que, de manière analogue à un processeur, nous pouvons définir un cycle général de fonctions exécutées par l'intergiciel.

L'algorithme 2 détaille les différents éléments à effectuer pour traiter une requête. Un puits de requêtes renvoie à une origine possible des requêtes : sources d'entrée/sorties,

interaction avec le code de l'utilisateur sur le nœud local. Une fois une donnée reçue sur un puits, celle-ci est traitée par l'intergiciel.

Algorithm 2 Traitement général d'une requête

1: **loop**
2: lire les données depuis le puits {"FETCH"}
3: extraire l'entête de la requête {"DECODE"}
4: aiguiller vers le composant cible de la requête
5: extraire les arguments de la requête
6: allouer les ressources pour le traitement de la requête
7: effectuer le traitement de la requête {"EXECUTE"}
8: **end loop**

Ce cycle fournit un ordre d'exécution admissible pour traiter une requête. Il masque en revanche les communications entre processus et entre nœuds. On peut alors décider des politiques d'ordonnancement de ces étapes en fonction des besoins de l'application, de la même manière qu'un processeur ordonnance le traitement des instructions suivant plusieurs politiques.

Par ailleurs, cette description masque les différentes fonctions nécessaires pour implanter effectivement chacune de ces étapes. Ceci ajoute un second niveau d'adaptabilité, en permettant de contrôler chacune de ces primitives.

Ainsi, cette analyse montre que l'on peut ramener un intergiciel à une boucle simple, pouvant être adaptée suivant les besoins de l'application suivant deux axes :

1. L'ordre des actions permet de contrôler le fonctionnement de l'intergiciel ;
2. La concrétisation des primitives permet d'affiner les fonctions fournies par l'intergiciel.

Cette boucle est définie de manière purement conceptuelle, nous nous proposons de définir plus en détail les différentes entités logicielles la formant à la prochaine section.

3.3.3 Synthèse

Dans cette section, nous avons passé en revue les différentes fonctions définies par l'architecture d'intergiciel schizophrène. Nous avons montré que ces fonctions pouvaient être ramenées à des abstractions simples. Nous avons montré par ailleurs que ces fonctions coordonnées étaient suffisantes pour représenter plusieurs classes d'intergiciels (MOM, DOC, RPC, Web). Elles représentent la "logique métier" de l'intergiciel.

Par ailleurs, nous avons présenté la boucle de contrôle d'un intergiciel. Ce module ainsi isolé permet de coordonner finement les différentes fonctions, et prend en charge la gestion des ressources nécessaires à chacune. Il apparaît naturellement du fait du découpage fonctionnel de cet architecture.

Cette étude montre ainsi que l'architecture schizophrène fournit les bases pour séparer les parties contrôle et commande d'un intergiciel. Nous disposons d'une définition de la partie commande de l'intergiciel grâce aux différents services définis. Nous définissons à la section suivante cette partie contrôle, incarnée par le module "μBroker".

3.4 Architecture du μBroker

Nous avons montré que l'architecture schizophrène permettait de construire autour d'une base de code commune des intergiciels supportant des modèles de répartition hétérogènes. Les fonctions constitutives de cette architecture peuvent être réduites à des abstractions simples. Ceci fournit une séparation entre la logique de l'intergiciel fournissant API et protocoles (mise en œuvre par les sept fonctions de base identifiées) et la partie contrôle de l'intergiciel (figure 3.2).

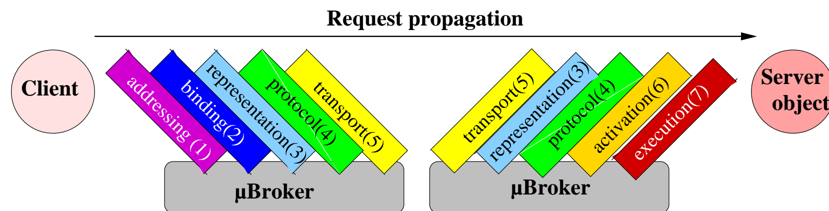


FIG. 3.2 – Place du module μBroker

Dans cette section, nous définissons le μBroker, et montrons son rôle d'intergiciel "canonique". Ce module définit un patron architectural pour la construction d'intergiciels. Ce patron et l'implantation qui lui est associée fournissent les modules adaptables permettant la construction d'intergiciels. Nous montrons que, compte tenu du découpage retenu, l'architecture dispose des modules minimaux nécessaires à cette construction.

Cette architecture nous servira de base aux chapitres suivants pour la construction d'intergiciels adaptés aux besoins des systèmes TR^2E en modèles de répartition et politiques de configuration. Nous en définissons d'abord l'architecture, puis la mise en œuvre. Nous présentons enfin les éléments de configuration de cette architecture, en particulier le support de plusieurs politiques de concurrence et de gestion des ressources..

3.4.1 Définition fonctionnelle du μBroker

Nous présentons dans cette section l'architecture du μBroker, en détaillant d'abord son diagramme de classe, puis les diagrammes de séquence présentant son rôle dans les différentes phases d'exécution de l'intergiciel.

Diagramme de classe

La figure 3.3 fournit une vue du diagramme des classes de la couche neutre de l'architecture schizophrène. Les différents services sont définis à l'aide des stéréotypes proposés à la section 3.3. Ce diagramme fournit une vue compacte des différents services fournis par la couche neutre, et les types de données utilisés. Nous présentons les différentes classes impliquées :

- `Servant` est un container pour les entités applicatives définies par l'utilisateur (servants CORBA, objets MOMA, etc) ;
- `Profile` fournit une vue particulière d'un `Servant` qu'il rend accessible (par exemple au travers d'une personnalité protocolaire) ;

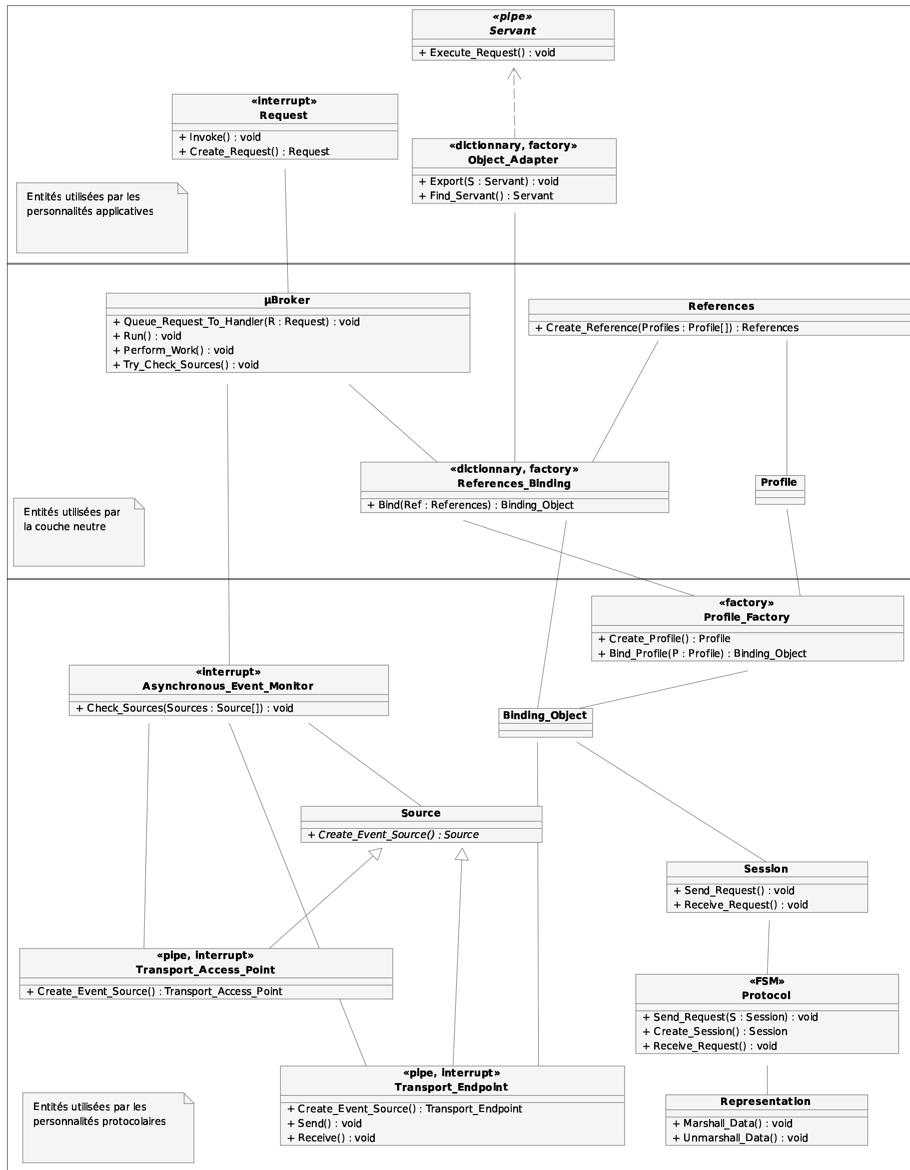


FIG. 3.3 – Intergiciel canonique

– References est utilisé pour le nommage des entités rendues accessibles par les différents nœuds de l'application, il s'agit d'une liste de Profile ;
– Binding_Object contient les informations relatives à une instance d'une pile protocolaire ;
– Request contient les informations échangées par l'intergiciel pour le compte des différentes personnalités. Une requête contient une référence de l'objet de liaison utilisé, un ensemble de paramètres de QdS et les données applicatives utiles ;
– μBroker est le cœur de l'intergiciel, il prend en charge la coordination des différents services de l'intergiciel, et le service *d'Adressage* ;
– References_Binding fournit le support pour le service de *Liaison* ;
– Object_Adapter prend en charge le service *Activation* ;
– Protocol définit les interfaces du service *Protocole* ;
– Representation définit le service *Représentation* ;
– Transport_Access_Point et Transport_Endpoint le service de *Transport*. Ces points de transport dérivent de l'objet source, qui représente une source d'évènements asynchrones pour l'intergiciel. Ces évènements représentent des interactions avec d'autres nœuds distants (ouverture de connexion, envoi de données, etc).

Diagrammes de séquence

La figure 2.1 présente les différents trajets que peut suivre une requête. Nous distinguons trois situations significatives :

1. envoi d'une requête à une personnalité protocolaire ;
2. envoi d'une requête à une personnalité applicative ;
3. réception d'une requête par une personnalité protocolaire.

Dans cette section, nous présentons les diagrammes de séquence associés à ces situations. Ces trois diagrammes couvrent les cas d'utilisation typiques, nous laissons de côté les phases d'initialisation qui seront abordées au chapitre suivant.

Envoi d'un requête à une personnalité protocolaire

La figure 3.4 présente cette configuration : une tâche représentant une tâche de l'utilisateur, ou une tâche réceptionnant une requête (figure 3.6) met dans la file du μBroker une requête.

Le μBroker prend en charge son traitement et lui alloue une tâche. Il peut s'agir éventuellement de la tâche de l'utilisateur. Ce paramètre est contrôlé par le μBroker.

Le service de *Liaison* construit l'objet de liaison associé à la référence stockée dans la requête (appel à Bind).

Dans le cas que nous traitons, cet objet de liaison représente un subrogé vers l'objet destinataire, la construction instancie alors une pile protocolaire et créée une *Session* au travers de laquelle le subrogé pourra transmettre la requête (appels successifs à Create_Profile et Create_Session).

La requête est alors transmise à l'objet de *Session* pour être emballée par le service de représentation puis envoyée.

Envoi d'une requête à une personnalité applicative

La figure 3.5 reprend la configuration initiale précédente et détaille la situation où l'objet de liaison retourné est un objet local (*servant*).

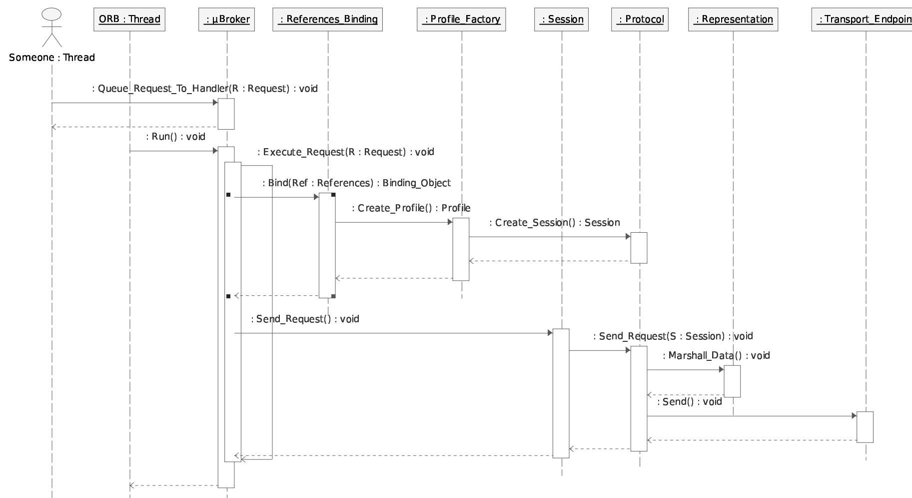


FIG. 3.4 – Envoi d'une requête à une personnalité protocolaire

Dans ce cas, le service de *Liaison* interroge l'*Object Adapter* pour déterminer le *servant* associé à la référence (appel à `Find_Servant`).

La requête est alors transmise au servant pour qu'il traite la requête.

Réception d'une requête par une personnalité protocolaire

Dans cette configuration (figure 3.6), une tâche est utilisée par le μBroker pour attendre sur les sources d'évènements de l'intergiciel.

Lorsqu'un évènement est détecté, celui-ci est traité par une des tâches confiées au μBroker. Le diagramme de séquence traite le cas où l'évènement correspond à une requête[3]. Celle-ci est alors lue par l'objet *Protocol*.

Une requête correspondant au message reçu est alors crées (appel à `Create_Request`), et est ensuite mise dans la file de travaux du μBroker.

La requête sera ensuite traitée suivant l'un ou l'autre des chemins d'exécution présentés précédemment.

Ces différents diagrammes de séquences complètent le diagramme de classe que nous avons présenté initialement. Ils précisent le déroulement des différentes phases du cycle de vie d'une requête et le rôle de chaque module. Ils fournissent ainsi une vue des interactions entre les classes entourant le μBroker.

3.4.2 Définition comportementale du μBroker

Nous avons présenté les fonctions impliquées dans l'architecture schizophrène, nous détaillons ici différentes politiques permettant d'ordonnancer ces actions.

On s'intéresse ici à deux des principaux patrons de conception permettant de décrire une architecture visant à traiter une requête, c'est à dire transformer un message en une exécution d'une portion de code. : *"Leader/Followers"* et *"Half Sync/Half Async"*.

[3] *Note : Les messages spécifiques au protocole ne sont pas représentés.*

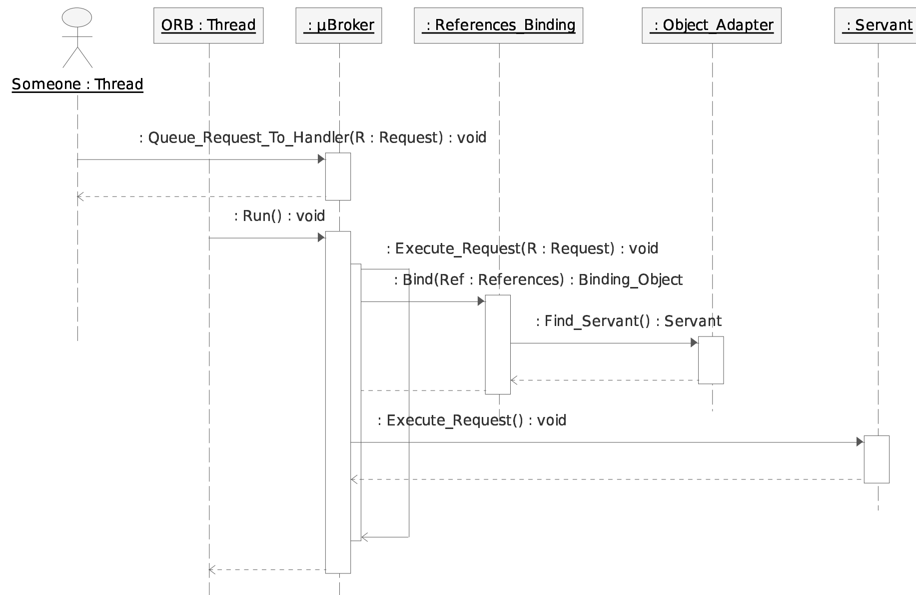


FIG. 3.5 – Envoi d'une requête à une personnalité applicative

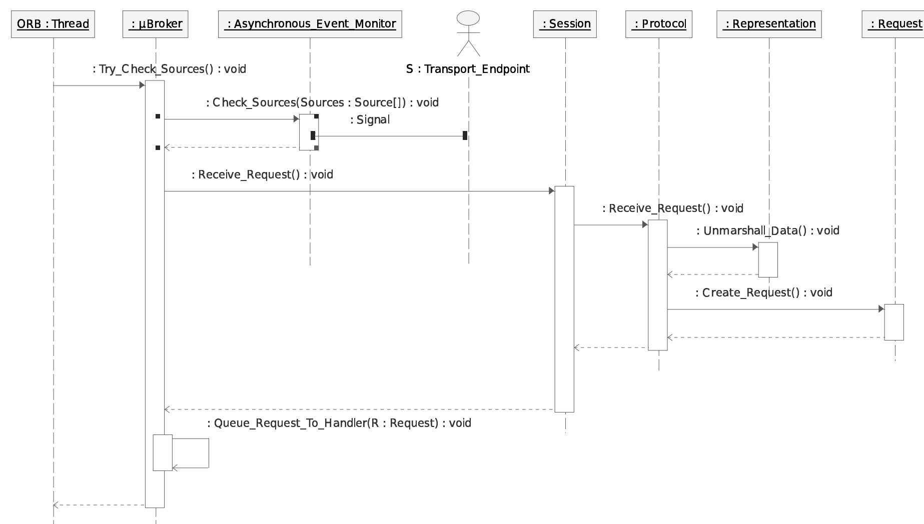


FIG. 3.6 – Réception d'une requête par une personnalité protocolaire

Cette étude nous permettra de mieux définir le couplage entre les modules définis dans la section précédente.

Ces deux patrons de conception sont définis dans [Schmidt *et al.*, 2000], et implantés dans de nombreux projets, dont TAO, Zen et nORB. On pourra se référer à [Subramonian *et al.*, 2002] pour les schémas de principe. On trouvera une analyse de ces patrons de conception dans le cas de RT-CORBA dans [Pyarali *et al.*, 2001].

Patron Half Sync/Half Async

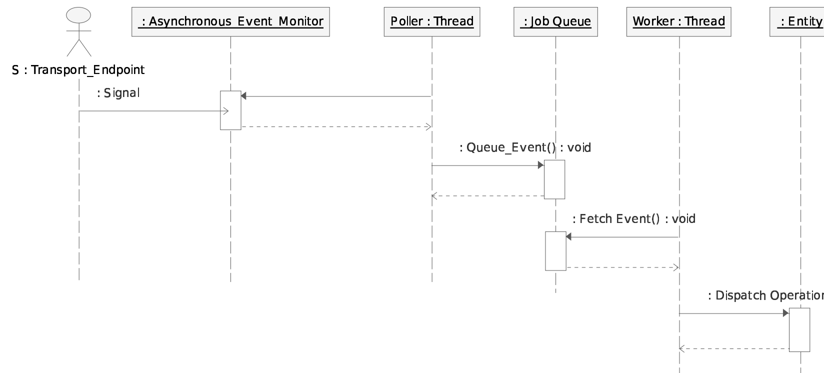


FIG. 3.7 – Schéma de fonctionnement du patron Half Sync/Half Async

Dans ce modèle, une tâche (*"Poller"*) prend en charge la lecture des données et stocke dans une file tous les évènements reçus. n tâches prennent en charge le traitement des évènements reçus (*"Workers"*). La synchronisation entre tâches est ainsi élevée.

La tâche chargée exclusivement de l'attente a une exécution que l'on peut analyser. On peut vérifier si on peut ou non perdre des données du fait de la saturation des tampons des entrées/sorties. On peut garantir que tout évènement détecté sera effectivement traité. Par ailleurs, on peut décider du niveau de priorité de chacune des tâches, de manière à rendre compte de la priorité de la requête.

Notons que ces différents paramètres peuvent être configurés statiquement avant la construction de l'application.

On vérifie ainsi le bon fonctionnement à l'exécution dans le cas de forte charge, mais aussi une efficacité moindre lorsque la charge est basse, due aux synchronisations.

Patron Leader/Followers

Un processus "maître" (*Leader*) écoute sur un ensemble de sources, un ensemble de processus "suiveurs" (*Followers*) sont en attente. Lors de la survenue d'un évènement sur une source, le maître cède son poste à un des suiveurs, puis traite l'évènement. Le patron de conception `Reactor` (cf [Schmidt *et al.*, 2000]) peut être utilisé pour traiter l'évènement de façon portable.

Il faut noter que, par définition du patron, même si plusieurs évènements sont détectés simultanément, un seul évènement sera traité. Les autres évènements restent dans l'ensemble des sources. Ils seront traités lorsque le nouveau maître écoutera sur les sources.

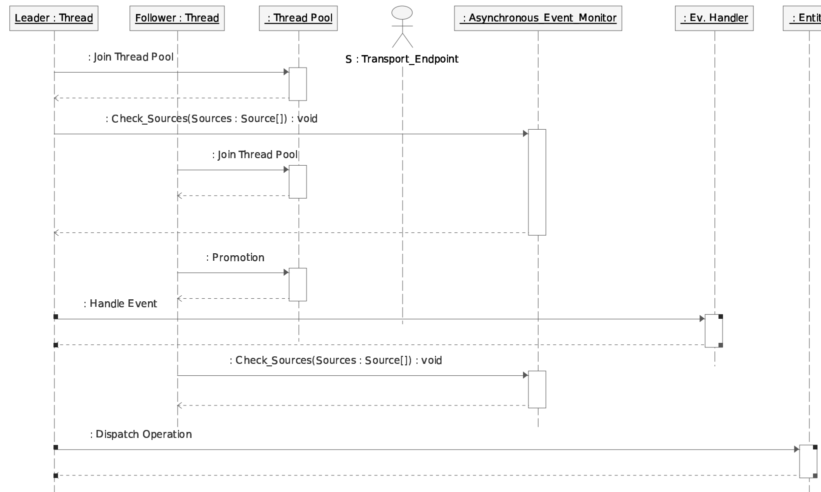


FIG. 3.8 – Schéma de fonctionnement du patron Leader/Followers

Ce patron est intéressant car il limite l'allocation mémoire et les synchronisations entre processus. Néanmoins, on peut critiquer le fait que l'on traite un seul évènement à la fois. Cela peut conduire aux comportements fautifs suivants :

– Famine :
  Considérons $n+1$ sources, n tâches et l'absence de notion d'ordre sur les sources et les évènements reçus (cas des `socket`). Alors, on peut se retrouver dans une situation où les

  1. $n+1$ sources reçoivent chacune un évènement,
  2. n sont en cours de traitement par les n tâches, 1 reste dans la file des évènements,
  3. si à nouveau les n autres sources reçoivent un évènement avant la fin du traitement précédent, alors on se ramène au (1)
  4. alors, par absence d'ordre sur les évènements, le même évènement peut à nouveau rester dans la file des évènements.

  Une solution, implantée dans le cas de PolyORB, et vérifiée formellement au chapitre 4, consiste à adopter une démarche hybride et lire tous les évènements puis les stocker dans la file des travaux de l'intergiciel. Une tâche sera élue *"Leader"*, tandis que d'autres seront réveillées pour traiter ces évènements.
– Pertes de message :
  Si tous les processus sont en train de traiter des requêtes, aucun n'écoute. Les données entrantes peuvent alors saturer le tampon des entrées/sorties et conduire à des pertes de données.

Un second problème provient de la gestion de la priorité de la tâche traitant une requête. La priorité de la requête ne peut être connue dans certains cas qu'une fois la requête entièrement lue, impliquant de pouvoir ensuite ajuster la priorité de la tâche. Ce changement dynamique de priorité va à l'encontre de certaines règles de conception de systèmes critiques.

Ainsi, ce patron de conception est peu intéressant lorsque le serveur est soumis à une forte charge, diverses erreurs peuvent se produire. En revanche, il peut se montrer

très efficace dans le cas où le serveur est peu sollicité. Le dimensionnement correct des ressources est donc critique.

Synthèse

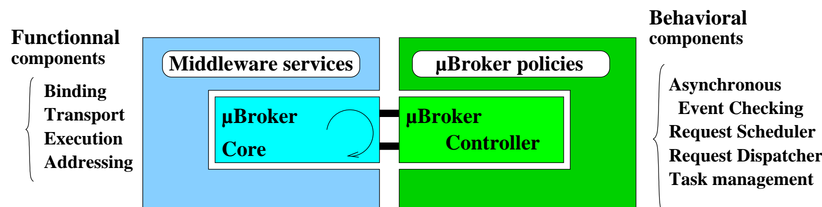


FIG. 3.9 – Séparation des aspects du μBroker

Les patrons que nous avons présentés fournissent les mécanismes nécessaires pour synchroniser et ordonnancer les différentes fonctions de l'intergiciel. Ils montrent aussi la relative indépendance entre les fonctions de l'intergiciel et leur composition et fournit une indication sur les évènements déclenchant une action de l'intergiciel et l'ordonnancement d'une tâche.

Sur la base de cette analyse, nous proposons (figure 3.9) de séparer le μBroker en deux sous-modules définissant chacun une facette : un cœur rassemblant les services de l'intergiciel, et un *"Contrôleur"* les ordonnançant. Ce découpage favorise la séparation des boucles de contrôle et de calcul de l'intergiciel.

La définition de ces deux facettes se fait en tenant compte *1)* des évènements à traiter, *2)* des interactions avec les autres fonctions et services de l'intergiciel.

3.4.3 Réalisation du μBroker

La partie précédente a motivé une nouvelle architecture, à plusieurs niveaux. Nous présentons ici sa réalisation.

Le μBroker coordonne les éléments de l'intergiciel. Plusieurs "stratégies" ont été définies pour contrôler et utiliser les ressources de l'intergiciel : [Pyarali *et al.*, 2001] détaille les politiques de traitement concurrent des requêtes de TAO ; le projet CA-RISM [Kaddour & Pautet, 2004] permet de contrôler l'instanciation et la reconfiguration dynamique des piles protocolaires.

De manière à prendre en compte cette variété de comportements, le μBroker doit être suffisamment adaptable tout en proposant une architecture simple. Nous avons retenu les éléments de conception suivants (figure 3.9) :

- Le cœur du μBroker prend en charge la partie fonctionnelle, et son interaction avec les autres fonctions et services de l'intergiciel. Il fournit une interface pour configurer le nœud et des routines auxiliaires pour certaines fonctions telles que la gestion des entrées/sorties et l'exécution d'un travail. Ce module interagit avec les services de *Liaison*, *Transport*, *Exécution* et *Adresage*.
- Le contrôleur du μBroker a la responsabilité de l'automate d'états associé au μBroker. Il gère l'accès aux ressources de l'intergiciel (tâches, entrées/sorties, files, etc) et ordonnance les tâches. Plusieurs politiques permettent de préciser son comportement (figure 3.10) :
  - La politique *"Asynchronous Event Checking"* contrôle la gestion de lecture et de contrôle des évènements sur les sources d'entrées/sorties ;

- Le module *"Request Scheduler"* ordonnance les requêtes (e.g. FIFO, etc) ;
- Le *"Dispatcher"* sélectionne une tâche pour traiter un évènement, ou effectuer un travail interne à l'intergiciel tel que l'écoute sur les entrées/sorties.

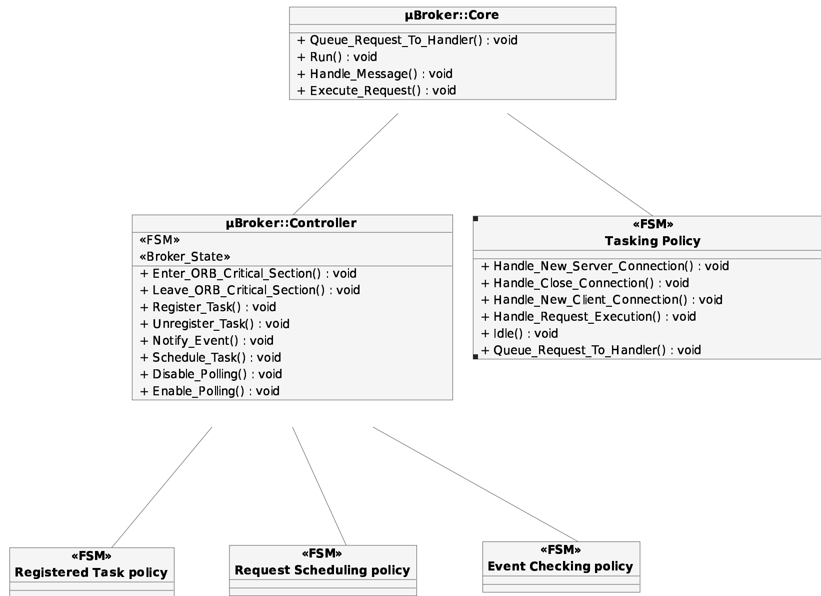


FIG. 3.10 – Architecture générale du μBroker

Ainsi, les différents modules du μBroker sont définis par leur interface et un modèle de comportement abstrait de haut niveau au travers des différents évènements qu'il peut traiter. Les spécialisations de ces entités raffinent ces comportements pour supporter une sémantique précise, telle que la gestion des tâches qui rentrent sous son contrôle (extrait 1).

Cette architecture a été implantée dans PolyORB, elle repose sur des entités simples. Nos tests (chapitre 6) montrent qu'elle peut être adaptée pour supporter les mécanismes classiques des intergiciels, mais aussi ceux plus sophistiqués tels que RT-CORBA.

Le μBroker propose une définition complète de la boucle de contrôle d'un intergiciel. Il complète la vue fonctionnelle de l'intergiciel que nous avons proposés de manière à fournir un canevas pour la construction d'intergiciels adaptés.

3.4.4 Mise en œuvre du patron pour la construction d'intergiciels

Nous avons détaillé à la section 3.2.2 l'utilisation de l'architecture schizophrène pour la construction d'intergiciels. Nous pouvons exprimer cette construction à l'aide des diagrammes UML précédents.

Sur les bases de notre analyse, construire un intergiciel dédié revient à :

1. étendre le comportement de certaines classes pour construire les personnalités désirées, et proposer une implantation adaptée des entités :
   - `Object_Adapter` et `Servants` pour une personnalité applicative ;
   - `Transport_Access_Point`, `Transport_Endpoint`, `Protocol` et `Representation` pour une personnalité protocolaire.
2. fournir les mécanismes pour l'enregistrement et l'instanciation de ces entités ;

Extrait de code 1 Spécification de `PolyORB.ORB_Controller`

```
package PolyORB.ORB_Controller is

   package PRS renames PolyORB.Request_Scheduler;
   package PTI renames PolyORB.Task_Info;

   type Event_Kind is
     (End_Of_Check_Sources,
      Event_Sources_Added,
      Event_Sources_Deleted,
      Job_Completed,
      ORB_Shutdown,
      Queue_Event_Job,
      Queue_Request_Job,
      Request_Result_Ready,
      Idle_Awake
      );

   type Event (Kind : Event_Kind) is record
      --  .. additional data for each Event_Kind

   end record;

   type ORB_Controller (RS : PRS.Request_Scheduler_Access) is
     abstract tagged limited private;

   type ORB_Controller_Access is access all ORB_Controller'Class;

   procedure Enter_ORB_Critical_Section (O : access ORB_Controller)
      is abstract;
   --  Enter ORB critical section

   procedure Leave_ORB_Critical_Section (O : access ORB_Controller)
      is abstract;
   --  Leave ORB critical section

   --  The following subprograms must be called from within the
   --  ORB critical section.

   procedure Register_Task
     (O  : access ORB_Controller;
      TI :        PTI.Task_Info_Access)
      is abstract;
   --  Register TI to scheduler S. TI may now be used by the ORB
   --  Controller to process ORB actions.

   procedure Unregister_Task
     (O  : access ORB_Controller;
      TI :        PTI.Task_Info_Access) is abstract;
   --  Unregister TI from Scheduler

   procedure Notify_Event (O : access ORB_Controller; E : Event) is abstract;
   --  Notify ORB Controller O of the occurence of event E.
   --  This procedure may change status of idle or blocked tasks.

   procedure Schedule_Task
     (O  : access ORB_Controller;
      TI :        PTI.Task_Info_Access)
      is abstract;
   --  Return the next action to be executed

end PolyORB.ORB_Controller;
```

3. assembler un ensemble cohérent de modules pour former *"l'instance d'intergiciel"* voulue.

Ces différents diagrammes complètent et enrichissent la description initiale de l'architecture schizophrène. Ils fournissent une description compacte des différentes entités impliquées, et détaillent les actions clés nécessaires à son bon fonctionnement.

Ainsi, ils forment une description d'un intergiciel minimal, bâti sur les concepts de l'architecture schizophrène, pour lequel nous disposons de nombreux résultats au niveau implantation. Nous détaillons les différents modules et politiques implantés dans les chapitres suivants.

La vision de l'architecture schizophrène que nous proposons fournit un modèle canonique, indépendant de tout modèle de répartition, d'un intergiciel. Ainsi, nous suivons une approche de conception *"à la MDA"* [OMG, 2003a]. Le modèle canonique définit un modèle indépendant (*"Platform Independent Model", PIM*) d'un intergiciel. Ce modèle est ensuite étendu pour supporter de nouveaux modèles de répartition, donnant forme à un modèle spécifique (*"Platform Specific Model", PSM*). Un PSM est alors une instance de l'intergiciel, formée d'une ou plusieurs personnalités interagissantes.

Nous montrons au chapitre 6 comment cette approche est mise en œuvre par combinaisons de briques intergicielles élémentaires qui étendent les services et blocs génériques définis par l'architecture schizophrène en fonction des besoins de l'utilisateur et du modèle de répartition désiré.

3.5 Conclusion

Ce chapitre a montré comment les architectures d'intergiciels configurable, générique et schizophrène fournissent une base intéressante pour la construction d'intergiciels temps réel adaptés aux besoins de l'application.

La définition précise des fonctions de l'intergiciel permet de construire un intergiciel qui est adaptable suivant un ou plusieurs axes. Ainsi, les intergiciels configurables permettent de paramétrer certaines fonctions de l'intergiciel, les intergiciels génériques fournissent un cadre général pour la construction d'intergiciel, ils définissent un abstraction simple mais difficile à utiliser. L'architecture schizophrène propose une vue originale d'un intergiciel en définissant précisément les fonctions mises en œuvre pour le traitement d'une requête.

Nos travaux démontrent l'intérêt de cette dernière architecture pour construire des intergiciels utilisant des modèles de répartition différents. Par ailleurs, le fort taux de réutilisation de code entre plusieurs instances d'intergiciel nous incite à conserver cette architecture pour le prototypage rapide de nouvelles fonctionnalités.

Une limitation commune à tous ces travaux est l'absence de guide méthodologique fiable, et l'absence d'une définition canonique d'intergiciel servant de base à la construction de nouvelles fonctionnalités. Nous avons effectué un travail sur l'architecture schizophrène visant à en extraire les principales caractéristiques, et en fournir une vue synthétique. Ainsi :

1. nous avons proposé une réduction des fonctions de base de l'architecture schizophrène pour les ramener à des patrons de conception ou des structures algorithmiques classiques ;
2. nous avons isolé les parties contrôle et fonction de l'intergiciel ; le *"μBroker"* a été isolé et définit comme étant le module central contrôlant l'intergiciel. Il

complète la palette de services de l'intergiciel proposée par l'architecture schizophrène ;

3. nous avons proposé les diagrammes de classes de séquence de ce module, fournissant ainsi une vue synthétique de l'architecture et de son fonctionnement ;
4. nous avons discuté comment ces modèles présentés sous forme de diagramme pouvaient être ensuite étendus pour la construction de nouveaux intergiciels. Nous avons noté le parallèle entre notre approche et le processus MDA.

Ces différents éléments fournissent donc une vue réduite d'un intergiciel, pouvant ensuite être étendue pour supporter de nouvelles fonctions. Les résultats nouveaux que nous avons apportés (modèle de répartition orienté messages, prototypage d'intergiciels) montrent qu'il est alors possible de construire de nouvelles fonctions qui vont enrichir cette vue et apporter les fonctions nécessaires à chaque application TR^2E.

Par ailleurs, notre analyse du patron de conception *"Leader/Followers"* montre que la définition incomplète d'un patron de conception peut conduire à des comportements fautifs. Son utilisation dans une application peut donc conduire à un échec du système. Nous proposons de vérifier formellement un assemblage de politiques du μBroker pour garantir le bon fonctionnement de la configuration.

Nous détaillons ces travaux dans les différents chapitres qui suivent.

4

Vérification d'instances d'intergiciel

Sommaire

4.1 Motivations **79**
4.2 Définition d'un processus de vérification **80**
4.2.1 Formalismes pour la modélisation 80
4.2.2 Techniques de modélisation et vérification 81
4.3 Réseaux de Petri **82**
4.3.1 Définitions générales 82
4.3.2 Analyse par model checking de l'espace d'états d'un réseau 83
4.4 Vérification d'instances d'intergiciels **85**
4.4.1 Processus de modélisation 85
4.4.2 Un modèle particulier : lecture des sources 86
4.4.3 Configurations du μBroker et modèles 86
4.4.4 Méthodes d'analyse 89
4.4.5 Résultats 92
4.5 Conclusion **95**

Ce chapitre s'intéresse à la vérification de la solution que nous proposons. Notre objectif est d'assurer des propriétés de déterminisme de notre architecture pour des instances (ou configurations) particulières d'intergiciel.

4.1 Motivations

Les architectures d'intergiciel pour systèmes TR^2E classiques ont montré qu'elles pouvaient répondre à des besoins précis. Cependant, nous remarquons qu'elles ne fournissent que peu d'informations sur leurs propriétés comportementales. Bien souvent, elles reposent sur un ensemble de tests et de scénarios permettant de tester des configurations bien spécifiques, par exemple dans le cas du projet Bold Stroke OFP [Sharp, 1998] mené par Boeing.

Nous notons plusieurs évènements qui conduisent à une explosion combinatoire du nombre d'états d'un système TR^2E : le nombre de chemins d'exécution admissibles pour une configuration d'intergiciel sur un nœud augmente avec le nombre de tâches et de requêtes à traiter. Le nombre de configurations de l'intergiciel est fonction de ses capacités d'adaptabilité et les plates-formes cibles qu'ils supportent.

Ainsi, nous affirmons que l'exécution de tests ne peut permettre à elle seule d'analyser les propriétés comportementales d'un intergiciel telles que *l'absence de verrouillage fatal (deadlock)*, *l'équité entre les requêtes*, *le dimensionnement correct des ressources*. Au contraire, les techniques formelles permettent de garantir, lorsqu'elles convergent, qu'une propriété est valide pour le modèle considéré, ou le cas échéant à un contre-exemple montrant en quoi elle est fausse.

Nous proposons donc d'utiliser des techniques de modélisation et de vérification formelle pour assurer certaines propriétés de notre architecture et de son implantation. Nous cherchons ainsi à réduire la *"distance"* entre le modèle et le système réel.

La vérification de systèmes est un domaine d'expertise distinct du domaine des intergiciels. Appliquer les techniques de modélisation et de vérification n'est pas immédiat, et requiert de s'intéresser aux mécanismes existants pour déterminer un processus de vérification permettant de déterminer précisément les propriétés du système.

Pour établir un lien entre la conception d'un intergiciel, et sa modélisation, nous avons travaillé en collaboration avec l'équipe de vérification formelle du thème SRC du LIP6 [Hugues *et al.*, 2004] pour mener notre étude à bien.

4.2 Définition d'un processus de vérification

Dans cette section, nous présentons le processus de modélisation et de vérification que nous avons retenu. Le comité ISO définit la vérification comme suit :

Définition 4.2.1. *La vérification est la confirmation par l'examen et la disposition de preuves objectives que les spécifications ont été remplies [ISO, 1994].*

La notion de preuves objectives nous incite à rechercher des mécanismes de vérification fiables et non ambigus, limitant les erreurs d'interprétation.

La diversité des formalismes et outils existants nous amènent à faire un choix. Compte tenu du type de système que nous modélisons, et la diversité des configurations à modéliser, nous privilégions une approche automatisable. D'autres paramètres tels que le pouvoir d'expression du formalisme et la disponibilité d'outils sont aussi à prendre en compte pour arrêter notre choix.

4.2.1 Formalismes pour la modélisation

Une analyse complète du patron de conception μBroker nécessite dans un premier temps une description précise de ses interfaces, de leur sémantique et des propriétés que l'on souhaite vérifier. Par la suite, cette description est exprimée grâce à une notation qui permet sa vérification formelle ; plusieurs transformations permettent de passer d'un modèle haut-niveau informel à la description précise d'un composant logiciel.

Ceci nous amène à nous interroger sur le choix de la notation (ou l'ensemble des notations) la plus adéquate. Nous avons considéré l'usage de plusieurs notations parmi lesquelles les automates, les diagrammes UML, les réseaux de Petri (colorés, stochastiques, temporisés, etc) ; ainsi que les langages de description d'architecture (*"ADL"*).

Nous notons qu'aucune de ces notations ne vient avec un cycle complet incluant modélisation et vérification. Chacune ne couvre qu'une part restreinte du cycle de vie du système : les diagrammes UML s'intéressent aux spécifications haut niveau ; les réseaux de Petri à la spécification formelle de systèmes de commandes ; les ADLs à la description d'architectures.

Par conséquent, il est nécessaire d'utiliser au moins deux notations pour couvrir à la fois la spécification et la vérification d'un système. Les travaux de recherche actuels s'intéressent à la combinaison de différentes modélisations d'un même système pour fournir sa description complète :

- Une possibilité est de dériver des réseaux de Petri des machines à états et des diagrammes UML [Merseguer *et al.*, 2002] à des fins de vérification. Cependant, aucun outil ne permet de réaliser cette tâche automatiquement. L'avènement d'UML2 et notamment les profils UML devraient lever ces limitations ;
- Une autre solution consiste à utiliser des notations spécifiques d'un domaine, tel que AADL (*"Architecture, Analysis and Description Language"*) [Feiler *et al.*, 2003], initialement conçu pour l'avionique, ou *LfP* (*"Language for Prototyping"*) [Regep & Kordon, 2001]. Ces approches montrent comment une notation permet de fédérer plusieurs techniques de modélisation et de vérification. Cependant, ces approches sont embryonnaires, et le manque d'outils pour tirer partie de ces formalismes réduit leurs champs d'utilisation.

Ces projets fournissent cependant un point d'entrée pour la spécification et la vérification formelle de composants logiciels. Ils suivent une approche descendante, de spécifications de haut niveau vers des spécifications formelles plus détaillées, qui permettent la vérification de propriétés du système.

Nous nous proposons de suivre une approche similaire, adaptée à un problème spécifique : la vérification de plusieurs configurations du μBroker. Les chapitres précédents ont fourni une vue précise des différents composants et interfaces mis en œuvre. Dans ce chapitre, nous nous intéressons à leur modélisation formelle.

4.2.2 Techniques de modélisation et vérification

Il existe deux grandes familles de méthodes formelles : la première basée sur une approche par preuves (tels que B [Abrial, 1995] ou Z [Diller, 1994]), la seconde basée sur la vérification de modèles par parcours exhaustif du graphe d'états (*"model checking"*), en utilisant des outils ou langages tels que SPIN [Holzmann, 2004] ou LUSTRE [Halbwachs, 1993].

Dans une approche par preuves, le modèle est décrit au moyen d'axiomes, les propriétés sont des théorèmes qui sont alors vérifiés par un prouveur.

Dans une approche par parcours du graphe d'états, le modèle est exprimé dans un langage formel dont on peut déduire un modèle d'exécution exhaustif. Ceci requiert généralement une définition mathématique rigoureuse. Un "moteur d'exécution" produit alors l'espace d'états complet associé au système sous forme d'un graphe où les actions sont associés aux états du système. Il est alors possible d'explorer le graphe et vérifier si une propriété est satisfaite. Ces deux approches sont complémentaires :

Les approches basées sur les preuves permettent l'analyse de systèmes dont l'espace d'états est infini. Cependant, l'utilisation d'un prouveur est une tâche technique qu'il est difficile d'automatiser.

Au contraire, le model checking est dédié aux système finis, cependant la plupart des étapes sont complètement automatisées [Clarke *et al.*, 2000], ce qui facilite leur utilisation par un utilisateur qui n'est pas expert dans le domaine. Par exemple, l'outil Qasar [Evangelista *et al.*, 2003] permet de construire un réseau de Petri à partir d'un code source et analyser certaines de ces propriétés élémentaires.

Nous avons retenu les réseaux de Petri bien formés comme langage pour la modélisation. Cette famille de réseaux de Petri permet la modélisation à haut niveau. Ce

formalisme permet de prendre en compte le typage des entités, et de définir des ensembles d'entités de cardinalité finie. Ceci permet une définition concise et paramétrée du système, tout en préservant sa sémantique.

Un des principaux bénéfices des réseaux de Petri bien formés est qu'ils permettent la construction automatique et efficace du *"graphe des marquages accessibles symboliques"*. Il s'agit d'un espace d'états quotient dans lequel les nœuds sont des classes d'équivalence entre états concrets, et les arcs des classes d'équivalence entre évènements [Chiola *et al.*, 1991]. Ce graphe est construit en exploitant les symétries du modèle, ce qui conduit à un graphe des marquages qui est potentiellement exponentiellement plus petit que le graphe des marquages classiques, et donc plus simple à manipuler.

Des travaux récents ont automatisé l'analyse des symétries admises par un système [Thierry-Mieg *et al.*, 2003], et des symétries d'une propriété [Baarir *et al.*, 2004]. Ces techniques, couplées à des outils de vérification automatique, nous autorisent une vérification du modèle par des propriétés de logique temporelle LTL, tout en combattant efficacement l'explosion combinatoire de l'espace d'états inhérente à ce type d'analyse et en limitant ses effets.

Comme nous le montrons dans les sections 4.4.4 et 4.4.5, l'utilisation de ces réductions basées sur les symétries nous permet de vérifier les propriétés de notre modèle, alors que les techniques classiques basées sur une exploration exhaustive de tous les états du système est impossible.

4.3 Réseaux de Petri

Nous présentons brièvement dans cette section le formalisme des réseaux de Petri, et indiquons comment ils peuvent être utilisés pour vérifier un système modélisé.

4.3.1 Définitions générales

Les réseaux de Petri proposent une notation formelle pour la modélisation et la vérification de systèmes concurrents, réagissant à des évènement extérieurs. Les outils existants permettent une vérification automatique des propriétés structurelles du modèle, ainsi que la vérification de modèle suivant des formules de logique temporelle. Ces formules nous permettent de vérifier l'absence d'un état interdit du système ou la causalité entre états.

Dans cette section, nous présentons rapidement les réseaux de Petri ; puis nous détaillons le processus de modélisation suivi.

Définition 4.3.1. *Un réseau de Petri coloré bien formé (*"Well-formed colored Petri net"*) [Girault & Valk, 2002] est un 5-uplet* $<P,T,Pre,Post,Types,M_0>$ *auquel on associe une représentation graphique. La figure 4.1 présente ainsi un modèle simplifié d'un système formé d'un client et d'un serveur, relié par un canal de communication sans perte.*

- *P est un ensemble de places (représentées par des cercles),*
- *T un ensemble de transitions (représentées par des rectangles),*
- *$Pre[t]$ la fonction de pré-condition associée à la transition t,*
- *$Post[t]$ la fonction de post-condition associée à la transition t,*
- *$Types$ l'ensemble des types de bases (un type de base étant un ensemble fini) et M_0 le marquage initial.*

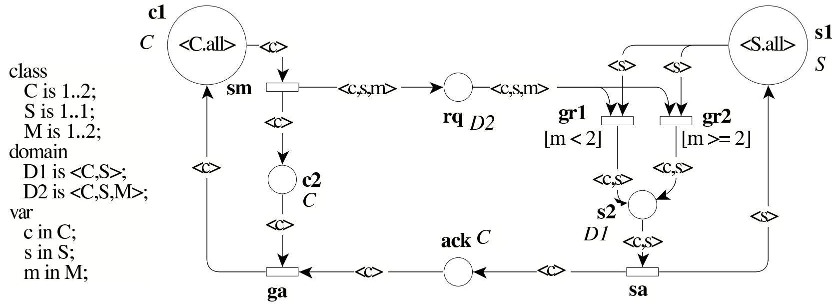


FIG. 4.1 – C/S : Réseau de Petri d'un système client/serveur

La définition des composants d'un réseau de Petri permet d'"encoder" la dynamique d'un système modélisé. Nous détaillons leur sémantique :

À chaque place p, on associe un domaine $Dom(p)$ où $Dom(p)$ est le produit cartésien de certains types de bases. Il correspond à l'ensemble des jeton colorés qu'une place p peut contenir. Dans la figure 4.1, la classe C est présentée. Le domaine de la place **c1** est C, l'ensemble des clients du système.

Parallèlement, un marquage $M(p)$ est associé à chaque place p, $M(p)$ est un multi-ensemble dans $Dom(p)$.

Mathématiquement, un multi-ensemble M d'éléments dans un ensemble E est une fonction de E vers l'ensemble des entiers naturels ($\mathbb{N}$), qui fait correspondre à chaque élément x de E, le nombre $M(x)$ d'occurrences de x dans M.

On définit ainsi un marquage M comme étant la fonction associant un marquage à chaque place p de P. Un élément du marquage d'un place est un *"jeton"*. Dans la figure 4.1, le marquage initial de la place $c1$ est $< C.all >$, c'est à dire qu'il contient un jeton par couleur de la classe C.

Les fonctions *Pre* et *Post* décrivent comment un marquage est modifié lorsqu'une action est réalisée. Les actions sont associées aux transitions du modèle, de sorte qu'on dit souvent qu'une "transition est tirée" pour signifier qu'une "action est réalisée".

On associe à chaque transition un ensemble de variables $Var(t)$, chaque variable est à valeur sur l'ensemble des types de base. Ainsi, $Var(\mathbf{sm})$ = $<c>$ et $Var(\mathbf{gr1})$ = $<c,s,m>$. La liaison d'une transition est l'association d'une valeur à chaque paramètre.

4.3.2 Analyse par model checking de l'espace d'états d'un réseau

Lorsqu'une transition est tirée, les jetons correspondants sont consommés des places en entrée, d'autres jetons sont générés dans les places de sortie. En se basant sur cette évolution, l'espace d'états associé au modèle peut être construit. La figure 4.2 présente l'espace d'états associé au modèle de la figure 4.1. Cet espace d'états est dit concret : il isole chacun des états du système.

Construire l'espace d'états du système permet d'analyser son comportement et vérifier la présence d'états particuliers : *"est-ce que telle situation est possible"* ou vérifier la relation causale entre deux états : *"si l'état e_1 est atteint, alors est-ce que l'état e_2 sera éventuellement atteint ?"*.

Les réseaux de Petri permettent aussi une analyse structurelle. Des propriétés telles que les invariants (*"le nombre de jetons présents sur un sous-ensemble de places reste constant "*) ou les symétries sont calculées sur la structure du modèle et ne nécessitent

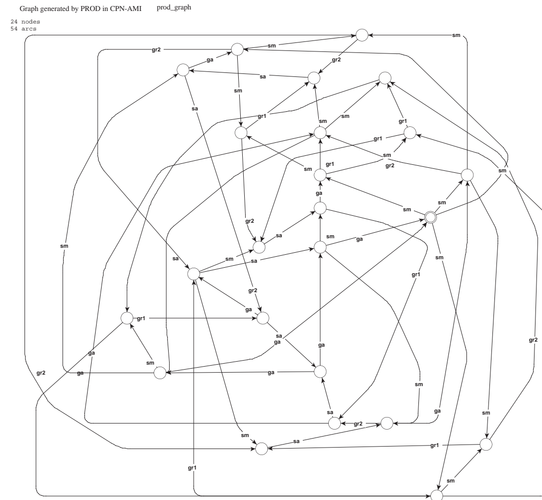


FIG. 4.2 – L'espace d'états concrets associé à C/S

pas la construction de l'espace d'états. Ceci permet de vérifier des systèmes dont l'espace d'états est infiniment grand [Girault & Valk, 2002].

Notons cependant que la construction de l'espace d'états concrets pose un problème pratique : isoler chacun de ces états a un coût mémoire important. Lorsque la combinatoire du système est élevée (cas de classes de couleurs de grande taille), il devient difficile voire impossible de construire l'espace d'états. Ce problème d' *"explosion combinatoire"* pose une limite forte sur l'utilisation de ces techniques.

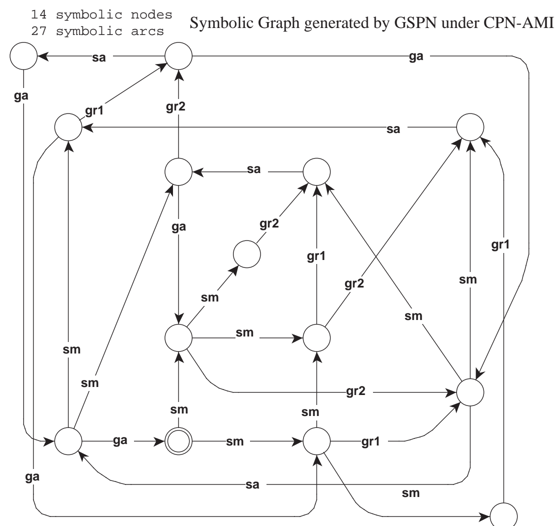


FIG. 4.3 – L'espace d'états symboliques associé à C/S

Pour contourner ce problème, les auteurs de [Thierry-Mieg *et al.*, 2004] proposent d'analyser non plus le graphe d'états concrets, mais le graphe quotient représentant les états symboliques du système. Ce graphe est construit à partir des informations de

structures du modèle initial, telles que les symétries. Ces informations permettent de construire des classes d'équivalences entre états.

Le systèmes comporte de nombreux états dénotant l'action d'un client sur une requête, par exemple états <tache_1_traite_requete_1>,... <tache_1_traite_requete_N>, <tache_C_traite_requete_1>, ... Ces différents états peuvent être réduits et représentés par l'état unique :
<une_tache_traite_une_requete>. Ceci permet de diminuer d'autant la taille du graphe d'états (figure 4.3).

Nous détaillons l'utilisation de ces différentes méthodes d'analyse par la suite, lors de l'exploitation des différents réseaux de Petri que nous avons construits.

4.4 Vérification d'instances d'intergiciels

Dans cette section, nous présentons les différentes étapes qui nous ont permis de vérifier certaines propriétés comportementales clés d'instances d'intergiciels, construites suivant le modèle exposé au chapitre précédent.

4.4.1 Processus de modélisation

Nous décrivons maintenant la phase de modélisation de notre architecture en utilisant le formalisme des réseaux de Petri comme langage de modélisation formelle. Les différentes étapes sont représentées figure 4.4.

Étape 1 : Les différentes entités de l'intergiciel sont isolés, et ramenés à un code source de petite taille, moins de deux cents lignes significatives pour chaque fonction. Cette phase utilise les résultats sur la réduction des fonctions de l'intergiciel, et la définition du composant μBroker présenté section 3.3.

Étape 2 : Nous construisons un réseau de Petri pour chaque implantation d'un élément de l'intergiciel. Les transitions du réseau de Petri représentent des actions atomiques ; les places sont soit des états de l'intergiciel, soit des ressources. Les interactions entre ces blocs sont représentées par des places communes à plusieurs réseaux, qui fonctionnent comme des *"places-canaux"* [Souissi & Memmi, 1989].

Étape 3 : Nous sélectionnons, pour une configuration du μBroker, plusieurs réseaux de Petri que nous assemblons afin de construire le modèle complet. Chacun des modèles sélectionnés correspond à une variation d'une des fonctions de l'intergiciel. Les places de communications (représentées en noir) représentent les liaisons vers d'autres fonctions du μBroker, ou d'autres fonctions de l'intergiciel.

Étape 4 : Les modules sélectionnés sont fusionnés pour produire un modèle global en reliant les interfaces des différents composants modélisés. Ce modèle et un marquage initial permettent la vérification des propriétés de la configuration du μBroker ainsi modélisée.

Cette construction par étapes du modèle permet de vérifier séparément chacune des fonctions avant de les intégrer dans le modèle global. Par ailleurs, plusieurs modèles globaux peuvent être construits à partir d'une même bibliothèque de modèles. Ainsi, nous pouvons tester et vérifier des conditions d'exécution spécifiques représentées par la mise en place de certaines politiques (représentées par des modèles élémentaires) ou des états précis (sélection du marquage initial).

Le marquage initial du réseau de Petri définit les ressources disponibles (nombre de tâches, de canaux de communications) ou l'état interne du système. Ceci fournit un paramétrage du modèle, et permet de tester plusieurs configurations facilement. Les classes du système fournissent un moyen simple de tester l'impact de ressources (par exemple des tâches) sur l'exécution du système.

Nous avons défini un marquage initial définissant le nombre de tâches et de requêtes pouvant circuler dans le modèle. Tâches et requêtes peuvent passer à tout moment d'un état libre, hors du contrôle de l'intergiciel, à un état où elles ont un rôle à jouer. Ceci nous permet de rendre compte du caractère dynamique de l'arrivée des requêtes sur un nœud de l'application.

L'espace d'états construit à partir du marquage initial couvre tous les entrelacements possibles d'actions atomiques : nous examinons ainsi tous les ordonnancements possibles. Cette couverture de l'ensemble des états atteignables du système nous permet de garantir que les propriétés vérifiées restent valides quelque soit la plate-forme d'exécution utilisée et quelque soit l'ordonnancement imposé. Ces résultats sont réutilisables.

4.4.2 Un modèle particulier : lecture des sources

Nous détaillons ici un des modèles construits (figure 4.5). Ce modèle prend en charge la lecture des données entrantes et leur transmission à la couche neutre.

Cette procédure est déclenchée lorsqu'une tâche est bloquée, en attente d'évènements, et est composée de deux phases : *1)* attente sur les sources d'évènements et *2)* traitement des évènements.

- **Attente sur les sources d'évènements :** La place *ThreadPool* contient toutes les tâches affectées à l'écoute des sources d'évènements. Par construction du système, au plus une tâche devrait occuper cette place. Lorsque la transition *ChckSrcB* est tirée, une tâche est sélectionnée pour entrer en phase d'attente. La transition *ChckSrcE* peut ensuite être tirée lorsqu'un évènement est détectée ((place *SigOut*). La place *Polling* joue le rôle de barrière et assure qu'au plus une tâche est en phase d'écoute.
- **Traitement des évènements :** Les évènements sont lus depuis la place *EvtHole*. Cette lecture doit se faire atomiquement (place *Lock*). La transition *FlushEvt* est tirée autant de fois qu'il y a d'évènements à traiter. La transition *FlushDone* est tirée lorsque tous les évènements sont consommés (arc inhibiteur entre *EvtHole* et *FlushDone*). Les évènements sont stockés dans une file, ils seront traités par la suite par une autre fonction, définie dans un autre modèle. Lorsque tous les évènements ont été traités, la transition *ProcEvtE* est tirée et libère le verrou. La tâche est alors libérée.

Les places représentées en noir ont un rôle spécial vis à vis du module : ils représentent les interactions avec d'autres modules. Leur marquage est généré par ces modules, et assurent une connexion entre les différents modules. Ils représentent soit des connections avec d'autres fonctions de l'intergiciel, soit la réception d'événements transmis par d'autres composants.

4.4.3 Configurations du μBroker et modèles

Dans cette section, nous passons en revue les paramètres clés qui caractérisent le μBroker, et les propriétés attendues de ce composant.

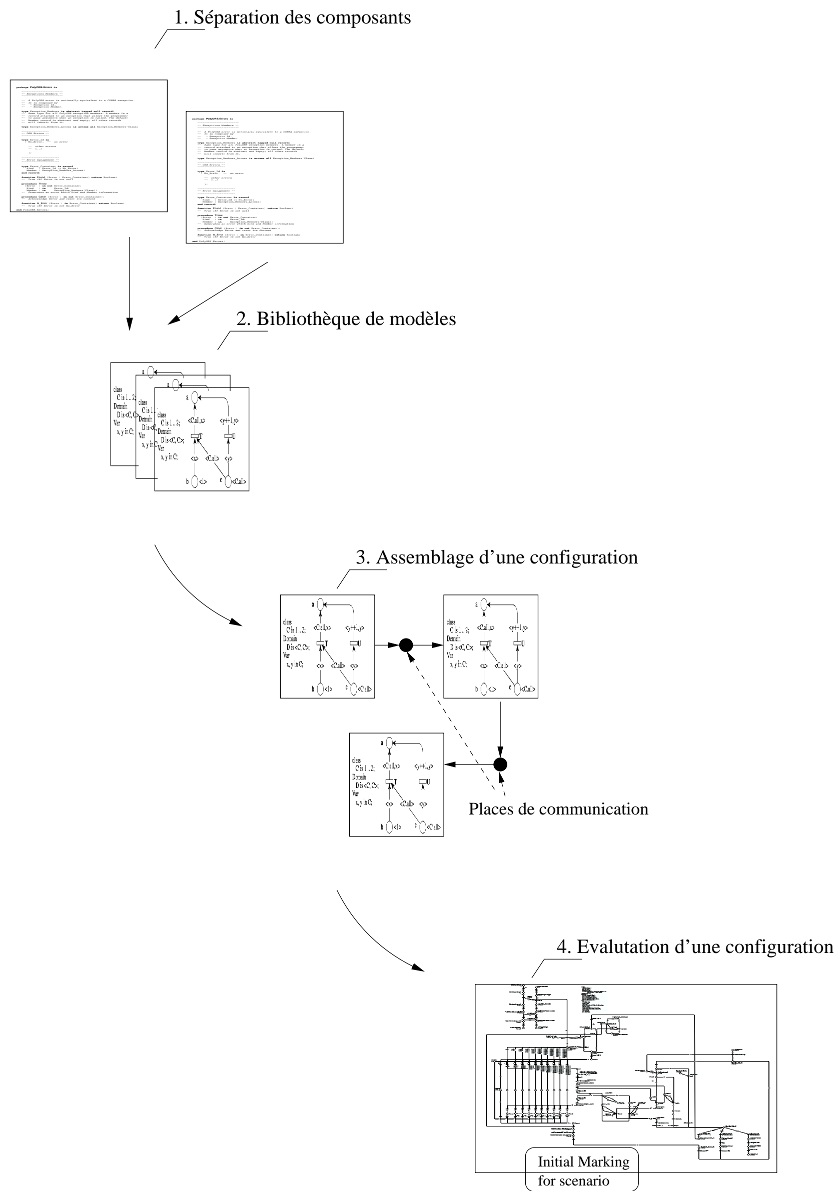


FIG. 4.4 – Étapes de modélisation

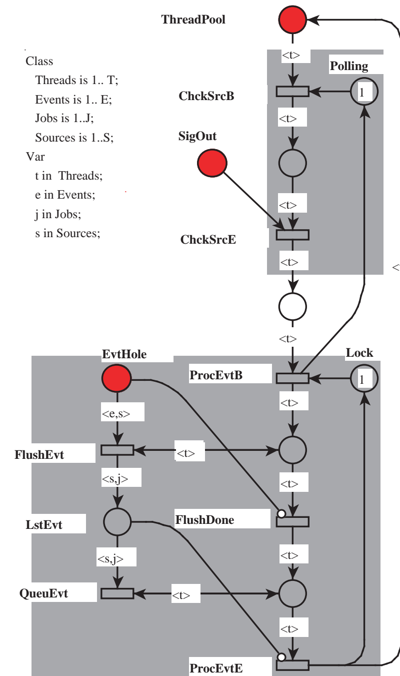


FIG. 4.5 – Un modèle de la bibliothèque

Une configuration du μBroker est définie par l’ensemble des politiques et des ressources qu’elle utilise. Ces paramètres sont communs pour une large classe d’applications. Nous considérons par la suite une instance d’un intergiciel, en mode serveur, qui traite toutes les requêtes entrantes. Nous étudions deux configurations du μBroker : *Mono-tâche*, disposant d’une tâche principale fournie par l’environnement d’exécution ; *Multi tâches*, plusieurs tâches s’exécutant en parallèle, et coordonnées suivant la politique *“Leader/Followers”* [Pyarali *et al.*, 2001].

Nous faisons l’hypothèse que les ressources utilisées par l’intergiciel sont allouées à l’initialisation du nœud. Nous considérons donc que l’intergiciel dispose d’un pool de tâches pré-allouées ; un nombre fixé de canaux de communications, représentées du point de vue du nœud modélisation par des sources d’évènements ; et une zone mémoire où stocker les requêtes en court de traitement. Cette hypothèse est raisonnable et correspond aux techniques d’ingénierie logiciel pour les systèmes critiques. Notre implantation et le modèle correspondant sont contrôlés par trois paramètres :

S_{max}	nombre de sources d’évènements
T_{max}	nombre de tâches disponibles
B_{size}	la taille du tampon mémoire alloué

S_{max} and T_{max} définissent un profil de charge pour le nœud (nombres de requêtes et d’évènements à traiter) ; B_{size} impose une contrainte sur la mémoire allouée par le μBroker pour traiter des requêtes. Ces paramètres contrôlent l’exécution correcte de l’intergiciel, et vont aussi définir sa “bande passante”.

Nous définissons quatre propriétés fondamentales pour le μBroker :

$P0$ (symétrie)	les tâches et sources d'évènements ne sont pas ordonnées
$P1$ (pas d'interblocage)	le système peut toujours traiter de nouvelles requêtes
$P2$ (cohérence)	il n'y a pas de corruption de la mémoire
$P3$ (équité)	tout évènement détecté sur une source sera traité

Notons que $P0$ n'est pas en soit une propriété déterminante du système. Il s'agit cependant d'une propriété structurelle forte. Elle fournit une indication de la cohérence du modèle construit. On s'attend en effet à ce que les tâches et les sources d'évènements soient interchangeables. Cette propriété nous permet d'utiliser les techniques d'analyse basées sur le graphe quotient, ceci nous permet de réduire grandement l'analyse du modèle.

$P0$, $P1$, $P3$ sont difficiles à valider même si l'on se limite à l'exécution de quelques scénarios : il faudrait être certain de couvrir tous les états du système. Cela peut ne pas être faisable techniquement en un temps limité, ni même possible à cause de l'entrelacement des tâches et des requêtes et du contrôle qu'exerce le système d'exécution sur l'ordonnancement du système.

Par ailleurs, le dimensionnement adéquat de ressources assurant le fonctionnement correct du système ($P2$) est un prérequis fort pour les systèmes TR^2E, mais aussi un problème difficile pour un système ouvert tel qu'un nœud d'une application répartie.

Nous présentons dans la section suivante la vérification de ces propriétés par model checking.

4.4.4 Méthodes d'analyse

Le système que nous modélisons est complexe, et coordonne plusieurs ressources. Nous nous attendons à ce que son espace d'états soit large. Nous détaillons ces chiffres à la section 4.4.5 : sa taille peut dépasser 10^{11} états pour des configurations raisonnables, ce qui rend ce système difficilement analysable par des outils classiques.

Ainsi, nous avons décidé de réduire la complexité algorithmique du problème à résoudre en utilisant des outils et méthodes développés dans le thème SRC du LIP6 où une partie de cette thèse a eu lieu.

Nous avons dans un premier temps effectué une analyse des symétries du modèle de façon à réduire le nombre d'exécutions du model checker. Elle nous permet aussi de construire le graphe des marquages symboliques. Cette analyse des symétries est complètement automatisée. Elle a été réalisée automatiquement pour chaque configuration du μBroker étudiée.

Enfin, nous prenons aussi en compte la propriété à vérifier, ce qui permet d'avantage de réductions de l'espace d'états par les outils utilisés. Ceux-ci ne considèrent alors que les comportements qui sont pertinents pour la propriété à vérifier, en se basant sur des algorithmes de parcours du graphe d'états adaptés à la propriété à vérifier.

Les outils utilisés ont joué un rôle crucial pour compléter ce travail de vérification et exploiter le travail de modélisation que nous avons effectué. Les risques d'explosion combinatoire sont élevés pour ce type de modèle. Ceci peut mettre en évidence les limites des outils utilisés, et rendre vain l'effort de modélisation, et impossible toute vérification.

- Nos modèles ont été construits en utilisant l'outil CPN-AMI [Kordon & Paviot-Adet, 1999], un environnement pour la construction et l'analyse de Réseaux de Petri, qui fédère de nombreux outils pour l'analyse structurelle et le model checking.

- Les symétries ont été calculées automatiquement en utilisant les outils de Yann THIERRY-MIEG [Thierry-Mieg *et al.*, 2003].
- L'analyse de l'espace d'états a été effectuée en utilisant une variation des bibliothèque du model checker GreatSPN [Departimento di Informatica, 2003]. Elles intègrent les algorithmes exploitant les propriétés de symétries des modèles, et les techniques basées sur le produit synchronisé.
- La bibliothèque SPOT [Duret-Lutz & Poitrenaud, 2004] a fourni une interface simple pour tous ces outils, masquant ainsi la complexité des algorithmes mis en œuvre, en particulier les formalismes utilisés lors des différentes phases de calculs intermédiaires.

P0 (symétrie) : Analyser les symétries du modèle

Nous souhaitons vérifier la propriété $P0$ (symétrie) ; la première étape est une analyse des symétries du système par une exploration de la structure du modèle. Ceci permet de déterminer les types de données présentant un comportement homogène.

Définition 4.4.1. *Deux éléments e_1 et e_2 d'un type de base sont* "symétriques" *si échanger e_1 et e_2 lors de l'exécution du système ne modifie pas son comportement du point de vue de l'*observateur.

L'algorithme présenté dans [Thierry-Mieg *et al.*, 2003] recherche les symétries d'un système en examinant chaque action (transition) du système et détermine quels sont les éléments distingués (si ils existent), et conclut que tout élément distingué par au moins une action doit être distingué par la relation d'équivalence $\mathscr{R}$ construite. $\mathscr{R}$ liste toute les valeurs symétriques pour le langage des actions du système.

Ce calcul automatique assure que tous les éléments équivalents pour la relation $\mathscr{R}$ ne seront pas distingués lors de l'exécution du système. $\mathscr{R}$ définit un ensemble de permutations et de rotations admissibles [Chiola *et al.*, 1991] qui peuvent être appliquées au système sans affecter les transitions de celui-ci.

Les symétries d'un modèle dépendent uniquement de sa structure. La complexité de ce calcul dépend de la taille du modèle (nombre de places et de transitions), ce qui rend ce calcul relativement rapide comparativement à un parcours de l'espace d'états associé au modèle. Le graphe quotient des marquages accessibles pour $\mathscr{R}$ est construit en utilisant les bibliothèques de GreatSPN [Departimento di Informatica, 2003]. Les nœuds de ce graphe sont des classes d'équivalence d'états pour la relation $\mathscr{R}$.

L'analyse des symétries du μBroker montre que les tâches et sources d'évènements forment deux classes d'équivalence. Ainsi, nous vérifions $P0$ (symétrie).

Notons que cette propriété structurelle dépend autant du système à modéliser que des abstractions utilisées. Une première version de nos modèles utilisaient une classe dédiée au comptage des requêtes stockées dans la file, conduisant à la construction d'une sous-classe statique ordonnée par ce compteur. Cette structure introduisait une asymétrie *"parasite"* induite par la modélisation et non le système à modéliser. Les outils nous ont permis de localiser l'asymétrie, que nous avons pu lever pour tirer partie de ces méthodes.

P1 (pas d'interblocage) : Analyse du graphes des marquages

La propriété $P1$ indique qu'il n'existe pas d'état d'interblocage.

Cette propriété peut-être vérifiée sur le graphe des marquages associé au modèle, en recherchant la présence d'éventuels nœuds terminaux, indiquant un état où le système cesse d'évoluer.

Cette analyse tire profit du graphe quotient pour rechercher les nœuds terminaux. Nous vérifions l'absence de tels nœuds pour les modèles considérés.

P2 (cohérence) : Vérification d'un propriété symétrique

Pour vérifier la propriété $P2$ (*cohérence*), nous devons vérifier que les accès à la zone mémoire sont corrects. Les places $(\mathcal{D}ata\mathcal{S}lots_i)_{i \in 1..M}$ de notre modèle représentent cette zone mémoire sous la forme de M "cases". L'opération d'écriture insère un jeton (une donnée) dans cet ensemble, une opération de lecture en supprime un.

Une erreur mémoire apparaît lorsque l'on tente d'écrire deux fois de suite dans une même zone mémoire.

Ceci peut être testé grâce à la propriété de sûreté exprimée par la formule de logique temporelle LTL (4.1), qui affirme qu'un tel état n'est pas atteignable.

$$\forall d \in \mathcal{D}ata\mathcal{S}lots, \mathsf{G}(card(d) \leq 1) \tag{4.1}$$

Nous remarquons que cette propriété est directement observable sur le graphe quotient produit pour la relation $\mathscr{R}$: les permutations d'éléments ne changeront pas le cardinal de la place testé par $P1$. Ainsi, si on considère un nœud de l'espace d'états quotient, alors tous ces éléments seront équivalents pour $\mathscr{R}$, et nous pouvons tester si les places vérifient $card(d) \leq 1$. La vérification de cette formule utilise automatiquement cette analyse des symétries pour réduire la complexité de l'analyse.

Ainsi, nous vérifions qu'il n'y a pas de corruption de données pour $S_{max} \leq B_{size}$ pour différentes valeurs de S_{max} et B_{size}. Nous notons qu'il peut y avoir corruption de données pour d'autres valeurs.

Par ailleurs, ce résultat peut aussi être interprété comme suit : le modèle ne permet pas plus de $card(\mathcal{D}ata\mathcal{S}lots)$ opérations d'écriture successives, et donc ne sature pas la zone mémoire. De sorte que la vérification de $P2$ implique aussi que la propriété $P2'$ (pas de saturation mémoire) est vraie.

Cette analyse nous permet de lier la taille du tampon mémoire utilisé au nombre de sources d'entrées/sorties du nœud. Ceci nous fournit une indication sur la configuration de l'intergiciel à utiliser pour l'application.

P3 (équité) : Utilisation des symétries du comportement observé

La relation d'équivalence $\mathscr{R}$ que nous avons introduite à la section précédente, bien que permissive, permet de vérifier des propriétés symétriques efficacement, en réduisant l'espace d'états à explorer.

La vérification de $P3$ nécessite que tout évènement détecté pour la source s sera finalement traité par le système. Pour ce faire, le jeton représentant s doit aller de la place indiquant la présence d'un évènement `ModifiedSrc` ("le μBroker a détecté une requête en attente") à la place initiale `DataOnSrc` ("nouvelle requête entrante"). Ceci amène donc à l'évaluation de la formule LTL suivante :

$$\forall s \in \mathcal{S}ources, \mathsf{G}(\{s\} \subseteq \texttt{ModifiedSrc} \Rightarrow \mathsf{F}(\{s\} \subseteq \texttt{DataOnSrc})) \tag{4.2}$$

La vérification de la formule $P3$ (*équité*) impose d'isoler la variable s lors du parcours du graphe des marquages. Ainsi, on associe à la variable s l'action "le μBroker a

détecté une requête en attente sur la source s" pour suivre le cheminement de s à travers l'intergiciel. Il nous faudrait ainsi vérifier $P3$ pour chacune des sources.

La propriété $P0$ (*symétrie*) nous permet d'affirmer, puisque les sources d'évènements sont équivalentes, le résultat suivant :

$$\exists s \in \mathcal{S}ources / P3(s) \Rightarrow \forall s \in \mathcal{S}ources, P3(s) \tag{4.3}$$

Ainsi, la proposition 4.3 nous indique qu'il suffit de vérifier la propriété pour une des sources pour garantir qu'elle sera vraie par extension pour toutes les autres sources. Ceci réduit le nombre d'invocation au model checker d'un facteur $card(\mathcal{S}ources)$.

Nous choisissons une source s que nous individualisons, en construisant un graphe quotient sous une relation d'équivalence $\mathscr{R}'$ plus restrictive, telle que s ne puisse être permuté avec une autre source. ainsi, l'observateur LTL peut isoler les évènements liés à s, et les distinguer des évènements liés aux autres sources.

Cependant, distinguer une source revient à considérer de nouveaux états pour chaque source isolée pour chaque nœud du graphe quotient initial, sous la relation $\mathscr{R}$.

Ainsi, même pour un modèle symétrique, l'analyse d'une formule non symétrique conduit à nouveau à une explosion combinatoire. Elle est seulement retardée par rapport à une analyse classique. Dans le pire des cas, le graphe d'états peut avoir la même taille que le graphe d'états initial, on perd donc le bénéfice de ces approches.

Les techniques d'analyse basées sur l'utilisation du graphe quotient d'accessibilité souffrent d'une limitation forte : l'incapacité à analyser des systèmes partiellement symétriques, ou de vérifier des propriétés partiellement symétriques.

Pour contourner ces problèmes, nous employons une autre technique d'analyse, elle aussi développée au LIP6/SRC : le produit synchronisé symbolique (*"Symbolic Synchronized Product"* (SSP)) [Baarir *et al.*, 2004].

L'idée de cette méthode est de modifier l'observateur LTL pour que celui-ci s'adapte à la volée à la propriété à vérifier. L'observateur ne distingue que les états pertinents pour vérifier une propriété, et agrège les autres états. Ainsi, l'observateur "zoome" sur les états devant être distingués, et ne conserve qu'une vision floue des autres états.

Par exemple, la propriété peut distinguer un objet seulement sur une partie de son exécution, comme pour la propriété $P3$. Les sources peuvent être permutées lorsqu'il n'y a pas de requêtes en attente. Nous n'avons besoin de distinguer la source s que lorsque la place `ModifiedSrc` contient une requête sur la source s.

Les algorithmes SSP fournissent un outil puissant, capable de tirer partie des propriétés des systèmes partiellement symétriques, dans lesquels les objets ont un comportement asymétriques seulement sur une partie restreinte du modèle. Ceci permet de réduire considérablement l'espace d'états du modèle.

Ces algorithmes nous ont permis de vérifier la validité de la propriété $P3$ pour les différents modèles considérés.

4.4.5 Résultats

Le modèle *Mono-Tasking* a 47 places, 38 transitions et 134 arcs. Le modèle *Multi-Tasking* est deux fois plus large. Le nombre de tâches et de sources n'a aucun effet sur la taille du modèle : ils sont codés comme des types de jetons dans le réseau de Petri. Le tampon est codé explicitement dans le modèle, sous la forme de N cases, sa taille a donc un impact sur le nombre de places et de transitions du modèle.

La figure 4.6 fournit la taille de l'espace d'états pour un tampon de taille $B_{size}=5$, et $S_{max}=4$ sources. Les modèles analysés ont une taille raisonnable. Cependant, l'espace

d'états est grand, jusqu'à 10^{11} états pour les configurations testées. Ceci rend compte de la complexité du système, et du fort entrelacement entre tâches et sources. Sa taille croit exponentiellement avec T_{max} et S_{max}. Remarquons aussi que la taille du graphe d'états symboliques est inférieure de plusieurs magnitudes à celle du graphe concret, et que cette différence croit avec T_{max}.

Ce gain provient du graphe quotient qui stocke dans chaque état jusqu'à $S_{max}! \cdot T_{max}!$ permutations de manière compacte. Il est donc exponentiellement plus petit que l'espace d'états initial, permettant une vérification accélérée de cette propriété.

Ces gains en espaces se répercutent sur l'utilisation mémoire et CPU. Les calculs ont été réalisés dans un temps raisonnable, sur des stations de travail classiques. Ainsi, les calculs les plus longs requièrent moins de 10 heures, sur un Pentium-IV cadencé à 2.6GHz, disposant de 512Mo de mémoire, sans recours à la mémoire tampon.

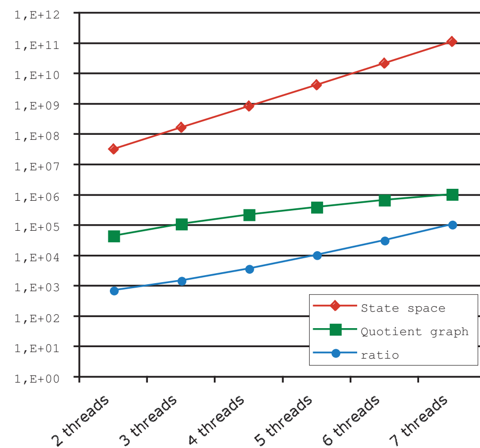


FIG. 4.6 – Espace d'états et espace quotient du μBroker pour S_{max}= 4 et B_{size}= 5

Importance des outils

Ces différentes mesures montrent que l'analyse de PolyORB n'aurait sans doute pas pu être menée sans l'utilisation de ces techniques avancées de model checking : l'espace d'états à parcourir est trop grand pour les stations de travail actuelles comme le montre la figure 4.6. On estime en effet qu'un model checker devient inefficace à partir de 10^8 états du fait de l'encombrement mémoire, et le recours à la mémoire tampon qui réduit considérablement les performances.

Nous notons un gain exponentiel entre l'espace d'états concret et l'espace quotient qui lui est associé. Comme illustration, remarquons que pour une configuration de l'intergiciel raisonnable (sept tâches et quatre sources), le système présente 10^{11} états, mais il a pu être analysé en parcourant un graphe quotient contenant 10^6 états.

Nous remarquons aussi que le graphe quotient croit moins vite que le graphe concret. Ainsi, il croit de 80% lors du passage de quatre à cinq tâches, de 57% pour cinq à six. Ceci est interessant lors de l'analyse de grands systèmes, et à mettre en perspective avec le graphe concret qui croit exponentiellement.

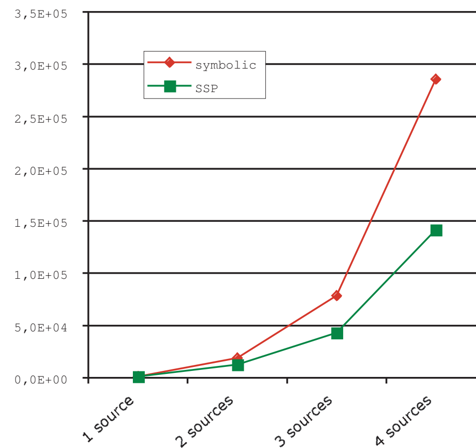


FIG. 4.7 – Nombre de nœuds examinés pour évaluer $P3$, T_{max}= 2 et B_{size}= 4

La technique basée sur le produit synchronisé (SSP) fournit d'avantage de réductions pour les systèmes partiellement symétriques comme le montre la figure 4.7. Elle détaille le gain obtenu lors de la vérification de la propriété $P3$, et en particulier l'augmentation de ce gain lorsque les paramètres varient. Nous ne fournissons aucune valeur pour $S_{max} \geq 5$; B_{size}= 4 car nous savons alors que $P1$ n'est plus vérifiée. Il est alors inutile de vérifier cette configuration.

Apports du travail de modélisation et de vérification

Des outils récents pour le model checking nous ont permis de réduire la complexité du temps de calcul nécessaire à la vérification. Cependant, l'utilisation d'outils combattant l'explosion combinatoire efficacement n'est qu'un aspect pratique de la vérification.

La réalisation de modèles conformes au système initial est une phase critique et difficile qui ne peut être automatisée. Comme nous l'avons indiqué lors de la vérification de la propriété $P0$, un modèle mal conçu peut rendre inopérantes certaines optimisations. Par ailleurs, il est nécessaire de mettre en œuvre une démarche rigoureuse de modélisation garantissant que le système modélisé et son modèle sont cohérents.

Nous avons effectué le travail de modélisation en parallèle au travail de définition et de conception des différents composants formant le μBroker : nous avons procédé à une définition précise de chacun, puis nous les avons réduits de façon à n'en conserver qu'une définition significative pour ensuite les modéliser à l'aide de réseaux de Petri. Ce travail de *"co-design"* entre le modèle et son implantation nous garantissent que le modèle et le composant modélisé sont proches sémantiquement.

Les propriétés que nous avons vérifiées suivant ce processus fournissent une preuve forte que l'architecture et l'implantation que nous en proposons sont correctes.

4.5 Conclusion

Dans ce chapitre, nous avons présenté les travaux de vérification formelles de notre architecture. Nous avons construit plusieurs modèles représentant le plus fidèlement possible la sémantique de plusieurs configurations du μBroker. Nous avons ensuite vérifié certaines de leur propriétés (absence d'interblocage, de famines, etc) à l'aide de formules de logique temporelle.

Ces travaux démontrent par l'exemple qu'une validation de ces propriétés à l'aide de tests auraient été difficilement possible, et certainement moins fiable. Des paramètres tels que l'ordonnanceur du système d'exécution utilisé, la fréquence du processeur ou le nombre de tâches ou d'entrées/sorties rendent certains états non atteignables par la seule expérience. La couverture des états du système serait donc incomplète.

Ainsi, nous avons modélisé plusieurs entités formant le cœur de l'intergiciel, dont nous avons isolé le code source pour ensuite le modéliser. Nous réduisons ainsi la distance entre le système et son modèle. Ce travail prolonge l'analyse présentée au chapitre précédent, et repose sur l'identification du μBroker et ses différentes politiques et leur réduction à des entités simples. Nous proposons ainsi une modélisation complète d'un intergiciel, exploitable. Le modèle est en liaison directe avec l'implantation.

Les modèles construits sont de taille réduite, et peuvent être vérifiés séparément pour s'assurer de leur cohérence avant de les assembler. Plusieurs modèles ont été assemblés et vérifiés. Chacun correspond à une instance d'intergiciel. Cette méthode a pour avantage de reproduire au niveau modélisation ce qui est réalisé à l'échelle du code source et lors de la phase de compilation.

Nous avons par ailleurs montré que certaines fonctions de l'intergiciel pouvaient être réduites à des abstractions classiques, réduisant ainsi la taille des modèles à considérer. De fait, il suffit de modéliser le μBroker et certaines de ses politiques pour pouvoir analyser une instance pour les propriétés que nous avons considérées.

Les propriétés démontrées pour une configuration sont aussi valides pour toute autre configuration compatible, par exemple utilisant une personnalité protocolaire proche (IIOP ou SOAP). Ce faisant, nous étendons la portée des propriétés prouvées à une large classe de configurations.

Compte tenu des résultats obtenus lors de ce travail de vérification, nous apportons les contributions suivantes :

1. Nous avons défini un processus de modélisation de l'intergiciel complet. Il isole les blocs nécessaires à l'évaluation de ses propriétés comportementales.
2. Nous avons modélisé plusieurs configurations d'intergiciels en associant un modèle élémentaire et une portion de code source. La distance entre modèle et implantation est donc réduite.
3. Nous avons utilisé des techniques nouvelles de vérification, qui permettent de réduire l'explosion combinatoire. Nos travaux ont en outre permis de valider sur des exemples concrets ces techniques.
4. Les propriétés vérifiées fournissent une base solide pour la construction d'intergiciels fiables et prouvés.

Nos travaux ont montré la pertinence de l'architecture proposée, et son apport bénéfique pour la modélisation d'un intergiciel à des fins de vérification. Nous avons pu opérer la vérification de propriétés fondamentales de l'intergiciel de façon automatique en faisant certains assemblages de modèles manuellement.

Ces travaux de modélisations ont fourni des résultats nouveaux aux deux communautés intergiciel et vérification. En effet, modéliser à l'aide de réseaux de Petri le cœur

d'un intergiciel est un processus complexe qui fournit une vue originale de l'intergiciel et permettant une analyse précise. Par ailleurs, nos travaux ont fourni un modèle original pour mettre à l'épreuve les outils de vérification et ainsi expérimenter leurs techniques sur des modèles moins classiques. Ce faisant, nous contribuons aussi à la mise au point des algorithmes et outils de vérification développés autour du projet CPN-AMI.

Une évolution naturelle serait d'enrichir PolyORB d'une description architecturale grâce à un ADL. Cette description et une bibliothèque de modèles permettraient de simplifier grandement l'évaluation de plusieurs configurations, et pousser plus en avant les travaux que nous avons réalisés. Nous discutons cette évolution dans la conclusion de ce mémoire.

Une seconde évolution serait d'étendre le processus de vérification à l'ensemble de l'application répartie. Comme le montre [Gill, 2003], la construction d'applications réparties et le choix de certaines politiques de configuration peut conduire à l'épuisement de ressources sur un nœud, conduisant au blocage d'autres nœuds. Ces situations résultent d'une erreur de conception du système qui doit être analysée à l'échelle de l'application. La validation de ce type de système devra mettre en œuvre un processus de vérification adapté, prenant en compte les échanges entre nœuds.

Deuxième partie

Mise en œuvre et expérimentations

5

Briques logicielles pour intergiciel temps réel

Sommaire

5.1 Langages de programmation pour le temps réel 99
5.1.1 Choix d'Ada 95 100
5.1.2 Fonctionnalités du langage Ada 95 pour le temps réel . . . 100
5.2 Structures de données pour intergiciels temps réel 103
5.2.1 Suppression des exceptions 103
5.2.2 Table de hachage dynamique parfaite 105
5.2.3 Gestion de la QdS 108
5.2.4 Gestion de la priorité 108
5.2.5 Files d'exécution 110
5.2.6 Ordonnanceur de requêtes 113
5.2.7 Conclusion 113
5.3 Services d'intergiciel TR^2E 113
5.3.1 Adaptateur d'objets déterministe 114
5.3.2 Protocole asynchrone 115
5.3.3 Couche transport temps réel 117
5.4 Conclusion 122

Un intergiciel utilise les services de base fournis par le système d'exploitation ou l'exécutif du langage ainsi que plusieurs briques logicielles élémentaires. Dans ce chapitre, nous présentons les différents modules logiciels que nous avons sélectionnés, du langage de programmation jusqu'aux services des personnalités, en passant par les structures de données. Nous montrons que chacun de ces niveaux répond aux besoins des systèmes TR^2E en matière de déterminisme ou de respect des contraintes de ressources.

5.1 Langages de programmation pour le temps réel

Dans cette section, nous motivons le choix du langage Ada 95 pour la réalisation de notre plate-forme, et détaillons ses fonctionnalités les plus pertinentes pour nos besoins.

5.1.1 Choix d'Ada 95

Le langage Ada 95 découle d'un besoin du département américain de la Défense de disposer d'un langage unique et généraliste pour la construction de ces systèmes. Le cahier des charges de ce langage [US. Department of Defense, 1978] couvre les besoins en gestion, les applications scientifiques, ou encore les applications temps réel et embarquées. Le langage Ada 95 découle de cet appel d'offre émis par le DoD. Ada 95 est maintenant une norme ISO [ISO, 2000a], ayant subi deux révisions en 1983 et 1995. Une révision du langage est en cours de rédaction et devrait donner naissance au langage Ada 05 [ISO, 2005]. Il devrait notamment ajouter la notion d'interfaces, empruntée au langage Java, modifier certaines règles de visibilité et ajouter plusieurs constructions pour les systèmes temps réel.

Le langage Ada 95 se caractérise par son typage fort, le haut niveau d'abstraction qu'il propose, et le support intégré pour la concurrence et l'ordonnancement des tâches. Par ailleurs, il inclut un support pour les interactions avec le matériel à bas niveau (clauses de représentation, adressage de la mémoire, etc). Enfin, l'utilisateur peut utiliser un sous ensemble restreint de constructions du langage, par le biais de restrictions. Elles permettent de garantir que le programme compilé n'utilise pas ses constructions, ce qui facilite sa validation et son déploiement sur des plates-formes restreintes.

PolyORB est écrit initialement en Ada 95, le département où cette thèse a été effectuée ayant une forte implication dans ce langage. Nous notons cependant que ce choix est aussi tout à fait raisonnable pour répondre à nos besoins.

Comme le note l'auteur de [Wheeler, 1996], d'autres langages plus traditionnels comme *C*, $C++$ ou Java ne supportent que très partiellement les besoins en fonctions avancées et en validation. Les spécifications Real-Time Java [Gosling & Bollella, 2000] sont encore en cours d'élaboration et de tests. Elles ne sont pas suffisamment matures pour être utilisées en production et donnent lieu à de nombreux ajustements en cours de finalisation [Dibble & Wellings, 2004]. L'utilisation des langages *C* ou $C++$ pose des problèmes de portabilité des bibliothèques d'accès aux couches basses et de concurrence. Ceci rend donc la tâche de réalisation plus longue. Au contraire, Ada 95 est maintenant un langage reconnu, fiable et extrêmement portable. Ainsi, ce langage se prête parfaitement à nos besoins d'implantation.

Par ailleurs, la disponibilité de PolyORB, produit maintenu par ADACORE dont le support des normes s'accroît [Vergnaud *et al.*, 2004] rend notre travail plus attrayant du fait de son impact rapide sur les communautés scientifiques d'une part, et industrielles d'autre part. Ainsi, nous pouvons noter que plusieurs des modules et fonctionnalités décrits par la suite sont maintenant utilisés dans le cadre de projets extérieurs, et donc validés “sur le terrain”.

5.1.2 Fonctionnalités du langage Ada 95 pour le temps réel

Nous listons ici certaines des fonctionnalités et guides utiles pour la construction de systèmes critiques en Ada 95.

Norme

La norme du langage fournit un ensemble de fonctions et d'annexes qui permettent de construire des applications temps réel critiques, citons :

- Annexe C : accès aux couches bas niveaux, support des interruptions, clauses de représentation des données, et pour l'accès aux interfaces matérielles ;

– Annexe D : support pour les systèmes temps réel, incluant l'ordonnancement préemptif, et les protocoles d'accès aux primitives de concurrence permettant de limiter et contrôler les inversions de priorité ;
– Annexe H : besoins pour les systèmes haute intégrité, permettant la validation et la certification de systèmes.

Extrait de code 2 Quelques fonctionnalités de Ada 95 pour le temps réel

```
with Ada.Real_Time; use Ada.Real_Time;
with System;

procedure Ada_Sample is

   type Data is range 0 .. 127;
   for Data'Size use 7;
   Invalid_Data : constant Data := 127;

   type Register is record
      Send : Data;
      Recv : Data;
   end record;

   for Register use record
      Send at 0 range 0 .. 6;
      Recv at 0 range 7 .. 13;
   end record;

   Device_Register : Register;
   for Device_Register'Address use System'To_Address (16#100#);

   protected Data_Driver is
      entry Get (M : out Data);
   private
      Current : Data := Invalid_Data;
      procedure Handle;
      pragma Attach_Handler (Handle, Device_IT_Id);
      pragma Interrupt_Priority;
   end Data_Driver ;

   protected body Data_Driver is
      entry Get (M : out Data) when Current /= Invalid_Data is
      begin
         M := Current;
         Current := Invalid_Data;
      end Get;

      procedure Handle is
      begin
         Current := Device_Register.Recv;
      end Handle;
   end Data_Driver ;

   task Real_Time_Sensor is
      pragma Priority (10);
   end Real_Time_Sensor;

   task body Real_Time_Sensor
   is
      D : Time := Clock;
      M : Data;
   begin
      loop
         Data_Driver.Get (M);
         delay until D;
         D := D + Milliseconds (40);
      end loop;
   end Real_Time_Sensor;

begin
   null;
end Ada_Sample;
```

L'exemple 2 fournit une illustration de ces fonctionnalités. Il montre comment interfacer un registre d'un micro-contrôleur (`Device_Register`) à une structure de données Ada 95. Les clauses de représentations permettent d'assurer une correspondance entre les données du registre et sa contrepartie Ada 95. Un gestionnaire d'interruption (`Data_Driver`) permet de rendre compte les modifications de ce registre. Enfin, une tâche périodique (`Real_Time_Sensor`) permet de lire ce registre.

Profil de restrictions Ravenscar

Le profil Ravenscar a été défini notamment dans [Dobbing & Burns, 1998] pour répondre à un besoin croissant d'utilisateurs en un sous-ensemble des constructions du langage Ada 95 facilitant la validation des propriétés temporelles et de déterminisme de l'application. Ce profil fait maintenant partie de la prochaine révision du langage, Ada 05 [ISO, 2005, (D.13.1)].

Le profil Ravenscar a été défini pour répondre aux besoins suivants :

– un modèle de concurrence déterministe qui supporte les derniers résultats de la théorie de l'ordonnancement ;
– développement d'un système exécutif Ada 95 efficace et de complexité réduite.

Le premier objectif permet de vérifier que chaque tâche remplira ses échéances temporelles, condition de bon fonctionnement intrinsèque des systèmes temps réel critiques. Le second objectif permet de limiter l'empreinte mémoire de l'exécutif Ada 95 ainsi que sa complexité. Ceci facilite la validation et la certification du système.

Pour ce faire, le profil définit un ensemble de restrictions sur le langage, renforcée par l'emploi de la directive de compilation `pragma Restrictions(<identifiant>)`. Chaque restriction définit un ensemble de fonctionnalités du langage qui sont retirées, et éventuellement de nouvelles vérifications effectuées lors de la compilation ou l'exécution de l'application.

Le profil Ravenscar impose les politiques d'ordonnancement `FIFO_Within_Priorities` et de verrouillage `Ceiling_Locking`. Il inclut les tâches allouées statiquement ne finissant pas, les objets protégés définis à l'échelle de la bibliothèque, les objets protégés avec au plus une entrée ayant une barrière booléenne, les délais absolus, une horloge temps réel monotone. D'autres constructions jugées peu sures ont été retirées, citons le changement dynamique de priorité, la destruction de tâches, la dépendance sur certains paquetages, dont `Ada.Time`, etc.

Les politiques retenues par le profil permettent d'analyser le temps de réponse des tâches, mais aussi effectuer d'autres analyses statiques (débordement de pile, analyse du flot de contrôle, d'exécution du programme, etc), qui sont définies dans des guides de conception d'applications critiques.

Guides de conception

Le profil Ravenscar, ou tout autre profil pouvant être mis en œuvre par combinaison de restrictions, ne fournit qu'un moyen de contraindre la sémantique de l'application.

Des guides de conception viennent compléter ces restrictions pour permettre de valider ou certifier l'application. Ainsi, [ISO, 2000b] détaille plusieurs méthodes de validation, et leur utilisation. En particulier, il définit l'impact de chaque fonctionnalité du langage vis à vis de plusieurs types d'analyse.

Suivant le degré de criticité du système, ou la nécessité d'être compatible avec des normes strictes telle que [SC-167, 1992], certaines vérifications devront pouvoir être menées. La compatibilité avec certaines restrictions permet de garantir ces propriétés, ou du moins facilite le travail de validation. Ces guides sont donc pertinents pour la construction de système TR^2E.

Enfin, la chaîne de production de code joue un rôle particulier pour renforcer ces différentes restrictions. Le frontal du compilateur réalise la vérification des restrictions du langage choisies par l'utilisateur. L'exécutif du langage peut être configuré et restreint suivant les restrictions choisies. Le compilateur peut ajouter de nombreuses vérifications de style, assurant que tout le code compilé est effectivement exécutable

(détection de code mort, vérification du caractère constant de certaines variables, etc) ou que le code répond à certaines règles d'écriture pour faciliter la relecture.

Pour la suite, nous chercherons à produire une implantation qui puisse, sous certaines conditions de configuration, être compatible avec le profil Ravenscar. Nous proposons une analyse précise des restrictions violées au chapitre 6.

5.2 Structures de données pour intergiciels temps réel

Le choix d'un langage de programmation n'est qu'une étape dans la construction d'un système TR^2E. Dans cette section, nous présentons les différentes structures de données que nous avons réalisées : suppression des exceptions, mise en œuvre de tables de hachages dynamiques parfaites, gestion de la qualité de service, des priorités, file d'exécution et ordonnanceur de requêtes.

Ces différentes structures répondent à des besoins ponctuels de l'intergiciel en fonctions déterministes. Elles permettent ainsi de construire un intergiciel adaptable répondant à un cahier des charges large.

5.2.1 Suppression des exceptions

Une exception représente une situation exceptionnelle, par laquelle l'exécution abandonne son cours normal. En tant que telle, une exception ne doit donc être employée que pour signaler l'occurrence d'un évènement rare.

Par ailleurs, l'utilisation des exceptions nécessite la mise en place de mécanismes particuliers au sein de l'exécutif pour l'arrêt d'une partie de l'exécution, jusqu'à entrer dans le gestionnaire d'exception prévu, ou terminer l'exécution du programme. Enfin, ceci pose le problème de la restauration du contexte d'exécution initial lorsque l'exception est traitée (libération de ressources, de verrous, etc) comme l'explique les auteurs de [Koopman & DeVale, 2000]. Ceci peut dans certains cas remettre en cause l'intégrité du système.

Ainsi, les systèmes critiques évitent d'utiliser ce type de mécanismes, sauf en cas d'erreur grave. Dans ce cas, un mécanisme de *"dernière chance"* est mis en œuvre et permet la prise en compte du système de cette erreur et son traitement éventuel. Les autres cas d'erreurs sont signalés par retour d'un code d'erreur.

Nous avons adopté une approche similaire pour la conception de la couche neutre et des personnalités protocolaires : les exceptions levées ne sont utilisées que pour signaler une faute grave dans le système. Il s'agit le plus souvent de tests de cohérence de certaines structures réalisés soit par l'exécutif du système, soit par le code écrit. Toute autre condition retourne un code d'erreur et une donnée précisant la nature de l'erreur. L'ensemble des codes d'erreurs utilisables est défini dans `PolyORB.Errors`.

Lorsqu'elles reçoivent ce code d'erreur, les personnalités applicatives peuvent transcoder cette erreur en une exception qui leur est spécifique. Les personnalités protocolaires retourneront un message spécifique à l'émetteur de la requête.

Cette approche nous garantit que le code écrit est compatible avec une partie des règles d'ingénierie des systèmes critiques. Par ailleurs, il nous permet un meilleur contrôle du flot d'exécution du programme : l'absence d'exceptions nous garantit que les points d'entrées et de sorties des différentes fonctions sont ceux que nous avons définis dans le code source. Nous détaillons ces différents points lors de l'étude critique de notre implantation, au chapitre 6.

Extrait de code 3 Spécification de `PolyORB.Errors`

```
package PolyORB.Errors is

   ------------------------
   -- Exceptions Members --
   ------------------------

   --  A PolyORB error is notionally equivalent to a CORBA exception.
   --  It is composed by
   --   - Exception Id,
   --   - Exception Member.

   type Exception_Members is abstract tagged null record;
   --  Base type for all PolyORB exception members. A member is a
   --  record attached to an exception that allows the programmer
   --  to pass arguments when an exception is raised. The default
   --  Member record is abstract and empty; all other records
   --  will inherit from it.

   type Exception_Members_Access is access all Exception_Members'Class;

   ----------------
   -- ORB Errors --
   ----------------

   type Error_Id is
     ( No_Error,  --  no error

       --  other errors
       --  <..>

       );

   ----------------------
   -- Error management --
   ----------------------

   type Error_Container is record
      Kind   : Error_Id := No_Error;
      Member : Exception_Members_Access;
   end record;

   function Found (Error : Error_Container) return Boolean;
   --  True iff Error is not null

   procedure Throw
     (Error  : in out Error_Container;
      Kind   : in     Error_Id;
      Member : in     Exception_Members'Class);
   --  Generates an error whith Kind and Member information

   procedure Catch (Error : in out Error_Container);
   --  Acknowledge Error and reset its content

   function Is_Error (Error : in Error_Container) return Boolean;
   --  True iff Error is not No_Error

end PolyORB.Errors;
```

5.2.2 Table de hachage dynamique parfaite

La construction de dictionnaires est résoluble d'un point de vue algorithmique en utilisant des tables de hachage, qui fournissent des garanties sur le temps nécessaire pour retrouver une objet, en général en $O(g(card(dictionnaire)))$, avec $g \neq 1$.

La construction d'applications temps réel motive la construction de dictionnaires en $O(1)$. Plusieurs algorithmes existent, dont celui présenté dans [Dietzfelbinger *et al.*, 1994] pour la construction de table de hachage dynamique parfaite. L'implantation originale de PolyORB proposait une première réalisation de cet algorithme, sans pour autant répondre totalement à ses hypothèses.

Nous notons par ailleurs qu'il n'existe aucun intergiciel utilisant ces mécanismes. Ceux-ci se limitent en général au cas de la construction de dictionnaires statiques, en utilisant des outils tels que `gperf` [Schmidt, 1990].

Une autre difficulté intrinsèque est la construction d'une fonction de hachage. Les algorithmes de la littérature ne considèrent que des entiers. Le cas des chaînes de caractères est rarement abordé. Il nous faut donc proposer une classe de fonctions optimisée pour les chaînes de caractères.

Compte tenu de la difficulté de se procurer une preuve des différentes constructions proposées, nous reproduisons les différentes étapes de construction d'une table de hachage dynamique parfaite à titre de référence. En particulier, nous avons repris les différentes étapes de cette construction pour assurer les hypothèses de faisabilité des théorèmes employés et lever les erreurs de réalisation initiales.

Classes de fonction de hachage universelle

Plusieurs définitions existent, nous ne retiendrons qu'une seule par la suite.

Définition 5.2.1. *[Carter & Wegman, 1979] Soit $\mathbb{U}$ un univers, I un ensemble, R un ensemble fini. Une classe de fonctions de hachage $\mathcal{H} = \{h_i : \mathbb{U} \to R, i \in I\}$ est dite (c,k)-universelle ssi*

$$\forall (x,y) \in \mathbb{U}, x \neq y, P(h \in \mathcal{H}/h(x) = h(y), x \neq y) \leq \frac{c}{|R|^k} \tag{5.1}$$

Cette définition indique qu'en moyenne au plus une fonction de hachage sur $|R|$ prise aléatoirement provoquera une collision pour deux valeurs différentes de l'alphabet. Cette propriété est utilisée pour assurer la construction d'une table de hachage parfaite.

Par la suite, on ne considérera que des classes de fonctions de cardinal fini. Ainsi, l'espace de probabilité associé à ce calcul de probabilité est l'espace canonique $(\mathcal{H}, \mathcal{F}, P)$ où $\mathcal{F}$ est l'ensemble des parties de $\mathcal{H}$.

Par convention, une classe de fonctions c-universelle indique une classe de fonctions $(c,1)$-universelle, une classe de fonctions universelle une classe $(1,1)$-universelle.

Table de hachage dynamique parfaite

Définition 5.2.2. *Une table de hachage est dite parfaite lorsqu'elle est sans collision.*

L'algorithme que PolyORB utilise pour la construction de ses tables de hachage dynamique parfaite est une version étendue de l'algorithme présenté dans [Fredman *et al.*, 1984] pour le cas dynamique, il a les propriétés suivantes :

Théorème 5.2.1. *[Dietzfelbinger* et al.*, 1994] Soit $\mathcal{H}$ une classe de fonctions de hachage universelle parfaite, alors la table de hachage construite suivant la méthode proposée est parfaite. Plus précisément, la recherche d'une clé est en $O(1)$, l'insertion d'une clé se fait en moyenne en $O(1)$ et dans le pire cas en $O(card(dictionnaire))$. L'espace mémoire utilisé est en $O(card(dictionnaire))$.*

Notons que dans le pire cas, l'insertion d'une clé nécessite la reconstruction complète de la table, la complexité de l'opération est alors $O(card(dictionnaire))$.

Construction d'une classe de fonctions de hachage universelle

Cette construction provient de [Cormen *et al.*, 2002]. Nous en fournissons ici une démonstration originale complète.

Définition 5.2.3. *Soit p un nombre premier. Soit $\mathbb{U}$ l'ensemble des p-uplets de $\mathbb{Z}/\mathbb{Z}_p$. Pour tout p-uplet $a = (a_0, a_1, .., a_{p-1})$ de $\mathbb{U}$ et pour tout $b \in \mathbb{Z}/Z_p$. On définit la fonction $h_{a,b}$ comme suit.*

$$h_{a,b}: \left\{ \begin{array}{ccc} \mathbb{U} & \rightarrow & \mathbb{Z}/\mathbb{Z}_p \\ x & \mapsto & (\sum_{j=0}^{p-1} a_j x_j + b) \bmod p \end{array} \right. \tag{5.2}$$

Théorème 5.2.2. *La classe de fonctions $\mathcal{H} = \{h_{a,b}, (a,b) \in \mathbb{U} \times \mathbb{Z}/\mathbb{Z}_p\}$ est universelle.*

Démonstration. Note : On utilise une caractérisation algébrique de l'équation à résoudre pour ne pas à avoir à compter les solutions.

p étant premier, $\mathbb{Z}/\mathbb{Z}_p$ est un corps commutatif, et $\mathbb{U} = (\mathbb{Z}/\mathbb{Z}_p)^p$ peut être assimilé à un espace vectoriel sur le corps $\mathbb{Z}/\mathbb{Z}_p$ de dimension p.

Soient $(a,b) \in \mathbb{U} \times \mathbb{Z}/\mathbb{Z}_p$, $(x,y) \in \mathbb{U}^2$.

$$h_{a,b}(x) = h_{a,b}(y) \Leftrightarrow \sum_{j=0}^{p-1} a_j(x_j - y_j) = 0$$

Dit autrement, a appartient au noyau de la forme linéaire f définie comme suit.

$$f: \left\{ \begin{array}{ccc} \mathbb{U} & \rightarrow & \mathbb{Z}/\mathbb{Z}_p \\ a & \mapsto & \sum_{j=0}^{p-1} a_j(x_j - y_j) \end{array} \right. \tag{5.3}$$

$Kerf$ est un hyperplan de $\mathbb{U}$, de dimension $p-1$.

Un $(\mathbb{Z}/\mathbb{Z}_p)$-espace vectoriel de dimension n contient exactement p^n vecteurs distincts. De sorte que,

$$P(h/h_{a,b}(x) = h_{a,b}(y)) = \frac{|Kerf|}{|\mathbb{U}|} = \frac{p^{dimKerf}}{p^{dim\mathbb{U}}} = \frac{p^{p-1}}{p^p} = \frac{1}{p}$$

On en déduit alors l'universalité de $\mathcal{H}$. □

$h_{a,b}$ est à valeur dans $[\![0, p-1]\!]$ pour p premier. Ceci peut ne pas convenir. Montrons comment se ramener à un intervalle image quelconque.

Définition 5.2.4. *Soit p un nombre premier, soit $\mathbb{U}$ l'ensemble des p-uplets de $\mathbb{Z}/\mathbb{Z}_p$. Soit m un entier, $m < p$. Soit le p-uplet $a = (a_0, a_1, .., a_{p-1})$ de $\mathbb{U}$ et soit $b \in \mathbb{Z}/\mathbb{Z}_p$. On définit la fonction $h^m_{a,b}$ comme suit.*

$$h^m_{a,b}: \left\{ \begin{array}{ccc} \mathbb{U} & \rightarrow & \mathbb{Z}/\mathbb{Z}_m \\ x & \mapsto & ((\sum_{j=0}^{p-1} a_j x_j + b) \bmod p) \bmod m \end{array} \right. \tag{5.4}$$

Théorème 5.2.3. *La classe $\mathcal{H}_m = \{h^m_{a,b}, (a,b) \in \mathbb{U} \times \mathbb{Z}/\mathbb{Z}_p\}$ est universelle.*

Démonstration. Soit $m \in \mathbb{N}$. Soient $(a,b) \in \mathbb{U} \times \mathbb{Z}_p$, $(x,y) \in \mathbb{U}^2$.

$$h^m_{a,b}(x) = h^m_{a,b}(y) \Leftrightarrow (\sum_{j=0}^{p-1} a_j(x_j - y_j) = 0 \bmod p) \bmod m$$

Or $p > m$ donc a appartient à l'un des $E(\frac{p}{m})$ espaces affines d'équation

$$\sum_{j=0}^{p-1} a_j(x_j - y_j) = i \bmod p, i \in \{0..E(\frac{p}{m})\}$$

Chaque espace affine est de dimension $p-1$, on en déduit

$$P(h/h_{a,b}(x) = h_{a,b}(y)) \leq \frac{|Kerf|}{|\mathbb{U}|} x \frac{p}{m} = \frac{p^{dimKerf}}{p^{dim\mathbb{U}}} x \frac{p}{m} = \frac{p^{p-1}}{p^p} x \frac{p}{m} = \frac{1}{m}$$

Donc $\mathcal{H}_m$ est universelle.

□

Réalisation

La construction des fonctions de hachage présentée à la section précédente nécessite la construction d'un vecteur a, pris aléatoirement dans $\mathbb{U}$.

Ceci pose un double problème pratique et théorique :

- `Ada.Numerics.Discrete_Random`, le paquetage par défaut proposé par la norme du langage Ada 95 ne dispose pas de la bonne catégorisation, PolyORB requiert un paquetage `Preelaborate`.
- L'implantation proposée par GNAT a une période de l'ordre de 2^{49}, ce qui peut poser problème pour la construction du vecteur a lorsque le dictionnaire devient de taille importante.

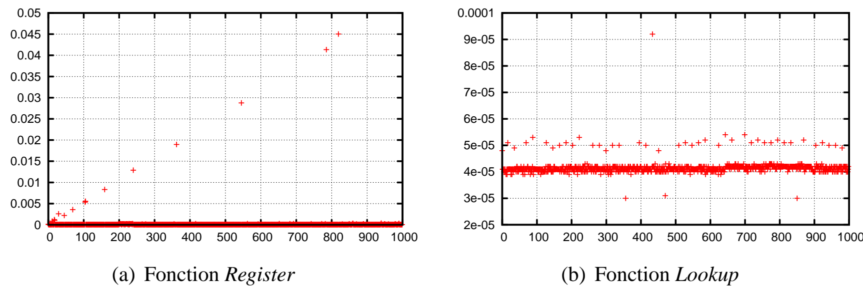
L'algorithme *"Mersenne Twister"* proposé par [Matsumoto & Nishimura, 1998] propose un algorithme original, de période élevée ($2^{19937} - 1$). Nous avons proposé une implantation de cet algorithme pour PolyORB, adaptée de l'implantation *C* proposée par les auteurs. Ce paquetage est utilisé pour construire la classe de fonctions de hachage que nous proposons.

Cette implantation a été testée sur plusieurs exemples, et sert en particulier au moteur d'initialisation de PolyORB. Ceci nous assure sa fiabilité.

Nous présentons ici une série de mesures effectuées sur un ordinateur de type PC, équipé d'un Pentium-IV cadencé à 2.4Ghz. Nous nous intéressons aux opérations d'enregistrement (*"Register"*) et de recherche (*"Lookup"*).

Les mesures présentées à la figure 5.1 confirment par l'expérience les propriétés attendues de l'algorithme implanté :

- La fonction d'enregistrement (figure 5.1(a)) a un comportement conforme à l'algorithme utilisé. Elle a un comportement globalement constant, sauf aux instants où la table doit être redimensionnée pour stocker d'avantage de clés. Ceci conduit à un traitement en $O(card(dictionnaire))$.
- La fonction de recherche est à un temps constant comme le montre la figure 5.1(b). L'écart-type mesuré est de l'ordre de $3.704 \cdot 10^{-5}$.



(a) Fonction *Register* (b) Fonction *Lookup*

FIG. 5.1 – Comportement temporel des tables de hachages dynamiques parfaites

5.2.3 Gestion de la QdS

La prise en compte de la Qualité de Service se fait au niveau de la requête. Le paquetage `PolyORB.Request_QoS` (extrait 4) fournit l'ensemble des primitives permettant d'attacher dynamiquement de nouveaux paramètres.

L'ajout de nouveaux paramètres se fait en enregistrant la fonction de rappel (*"call-back"*) . Cette fonction sera appelée par le gestion de QdS. Elle renseignera le paramètre pour lequel elle a été enregistrée. La décision d'inclure ces fonctions de call-back se fait au moment de la construction de l'application. Ainsi, seuls les paramètres utiles pourront effectivement être positionnés. L'état courant de l'intergiciel fera que seul un sous-ensemble des paramètres enregistrés sera utilisé.

Les informations de QdS sont stockées sous forme d'un objet `NotePad`, défini par le patron de conception *Annotations* dans [Quinot *et al.*, 2001]. Ce patron permet d'ajouter dynamiquement une information à un objet sans nécessiter d'étendre son type.

Notons que l'utilisation de ces paramètres est multiple, et dépend des paramètres considérés. Ainsi, la QdS est diffuse dans l'intergiciel. Elle est créée à un point du traitement d'une requête ou une session, consommée à un autre. Il est donc nécessaire de disposer d'un mécanisme générique pour la transmission de ces informations. L'utilisation du patron *Annotations* répond parfaitement à ces exigences.

Ceci nous autorise une grande souplesse d'utilisation : de nouveaux paramètres de QdS peuvent ainsi être ajoutés sans modification du code. Il suffit d'ajouter à l'objet `NotePad` une note correspondant aux paramètres à transporter là où cette information est connue (par exemple dans le code utilisateur). Ces informations seront utilisées par la suite par différents modules pour lesquels l'information est pertinente (par exemple au niveau du service de transport).

Nous avons ainsi utilisé ce mécanisme pour le passage de la priorité des requêtes et les informations de services de CORBA/GIOP (*"Services Contexts"*).

5.2.4 Gestion de la priorité

La notion de priorité au sein d'une application répartie, et son interprétation lors du traitement d'une requête sont multiples. L'échelle des priorités peut varier d'un nœud à l'autre, du fait de leur hétérogénéité. Il faut donc définir un mécanisme de normalisation permettant d'affecter une même valeur de priorité quelque soit le nœud considéré.

Le paquetage `PolyORB.Tasking.Priorities` enrichit la couche neutre en lui fournissant un mécanisme proche de ceux proposés par RT-CORBA.

Extrait de code 4 Spécification de `PolyORB.Request_QoS`

```
with PolyORB.References;
with PolyORB.Requests;

package PolyORB.Request_QoS is

   --  List of supported QoS policies

   type QoS_Kind is
     (Static_Priority,
      GIOP_Code_Sets,
      GIOP_Service_Contexts);

   --  Definition of QoS parameters

   type QoS_Parameter (Kind : QoS_Kind) is abstract tagged null record;

   type QoS_Parameter_Access is access all QoS_Parameter'Class;

   procedure Release (QoS : in out QoS_Parameter_Access);

   type QoS_Parameters is array (QoS_Kind) of QoS_Parameter_Access;

   function Fetch_QoS
     (Ref : in PolyORB.References.Ref)
     return QoS_Parameters;
   --  Return the list of the QoS parameters to be applied when
   --  sending a request to the target denoted by Ref. This functions
   --  iterated over the different call-backs.

   procedure Set_Request_QoS
     (Req : in PolyORB.Requests.Request_Access;
      QoS : in QoS_Parameters);

   function Extract_Request_Parameter
     (Kind : in QoS_Kind;
      Req  : in PolyORB.Requests.Request_Access)
     return QoS_Parameter_Access;
   --  Return QoS parameter of type Kind from request QoS, or a null
   --  if no parameter matches Kind.

   function Extract_Reply_Parameter
     (Kind : in QoS_Kind;
      Req  : in PolyORB.Requests.Request_Access)
     return QoS_Parameter_Access;
   --  Return QoS parameter of type Kind from reply QoS, or a null
   --  if no parameter matches Kind.

   type Fetch_QoS_CB is access function
     (Ref : in PolyORB.References.Ref)
     return QoS_Parameter_Access;
   --  This call-back function retrieves one QoS_Parameter to be
   --  applied when sending a request to the target denoted by Ref.
   --  Return null if QoS parameter is not applicable for Ref.

   procedure Register (Kind : in QoS_Kind; CB : in Fetch_QoS_CB);
   --  Register one call-back function attached to QoS_Kind Kind

   function Image (QoS : in QoS_Parameters) return String;
   --  For debugging purposes. Return an image of QoS

end PolyORB.Request_QoS;
```

Le type `External_Priority` définit les niveaux de priorités telles qu'elles sont manipulées par les différentes personnalités. Il s'agit d'un intervalle de valeurs quelconques, pris sur les entiers.

Le type `ORB_Priority` définit les niveaux de priorités telles qu'elles sont manipulées par les entités de la couche neutre. Ce type est un sous-ensemble des priorités disponibles sur la plate-forme cible.

Nous avons noté une similitude entre le comportement de l'intergiciel (représenté par le μBroker) et un noyau de système d'exploitation. Tous deux ont un mode de fonctionnement dirigé par les évènements à traiter (section 3.3.2).

Ceci nous amène à raffiner les niveaux de priorités internes, pour rejoindre la manière dont sont traitées les tâches par un noyau temps réel : la gestion des interruptions, et l'ordonnancement des tâches se font à une priorité interne élevée, et préempte toute autre action réalisée.

Le niveau `ORB_Priority` est lui même découpé en deux sous-classes :
– `ORB_Core_Priority` correspond au niveau de priorité interne le plus élevé ;
– `ORB_Component_Priority` correspond aux actions moins prioritaires. Ceci permet de partitionner les différentes actions effectuées par l'intergiciel en deux classes dédiées. Certaines actions de l'intergiciel telles que les entrées/sorties ou l'affectation des travaux aux processus légers de l'intergiciel doivent en effet pouvoir être effectuées à une priorité maximale.

Les fonctions `To_External_Priority` et `To_ORB_Priority` assurent une projection entre priorités des domaines `External_Priority` et `ORB_Priority`. Notons que ces transformations ne peuvent être bijectives. Il appartient donc à l'utilisateur de correctement définir les projections utilisées. Nous discutons certaines de ces contraintes dans le cas de RT-CORBA à la section 6.3.3.

5.2.5 Files d'exécution

Les files d'exécution (*"Thread Lanes"*) correspondent à des tâches spécialisées, dédiées au traitement d'une classe de travaux. Elles fournissent un mécanisme analogue aux objets `ThreadpoolLanes` de RT-CORBA. Cet objet nous servira de brique de base pour l'extension de la fonction d'exécution de PolyORB.

Les files d'exécution fonctionnent suivant un mode producteur/consommateur. Une tâche externe confie un travail au gestionnaire de files. Ce travail peut éventuellement comporter une information sur la manière de le traiter, telle que sa priorité d'exécution. Par la suite, une tâche de la file sera réveillée, puis prendra en charge ce travail, tel qu'exposé par le diagramme 5.2.5.

Le paquetage `PolyORB.Lanes` définit plusieurs types de files (extrait 6), suivant le graphe d'héritage suivant :
– `Root_Lane` est la classe racine abstraite. Elle dispose d'une unique primitive qui teste si une requête peut être traitée, et l'enregistre dans sa file le cas échéant.
– `Lane` correspond à une file d'exécution simple. Il s'agit d'un ensemble de tâches statiques, auxquelles peuvent venir s'ajouter plusieurs tâches créées dynamiquement. Toutes les tâches sont à la même priorité.
– `Lane_Set` est un ensemble statique d'objet `Lanes`.
– `Extensible_Lane` est une variante dynamique de `Lane_Set`.

Ces différentes classes nous permettent de construire les entités qui pourront traiter les requêtes soumises par les différentes personnalités.

Extrait de code 5 Spécification de `PolyORB.Tasking.Priorities`

```
with System;
with PolyORB.Types;

package PolyORB.Tasking.Priorities is

   --  ORB priorities are derived from Ada native priorities. We
   --  define ORB_Core and ORB_Component priority levels, so we make a
   --  distinction between ORB Core entities that require high
   --  priority to process some information and other components.

   subtype ORB_Priority is System.Priority range
     System.Priority'First .. System.Priority'Last;
   --  ORB priority range

   ORB_Core_Levels : constant Natural := 1;
   --  Number of priority levels affected to the ORB Core

   subtype ORB_Component_Priority is ORB_Priority range
     ORB_Priority'First .. ORB_Priority'Last - ORB_Core_Levels;
   --  ORB_Component_Priority defines the priority an ORB component
   --  may have. This range usually applies to most components,
   --  including user components.

   Default_Component_Priority : constant ORB_Component_Priority;
   --  Default priority for ORB Components

   subtype ORB_Core_Priority is System.Priority range
     ORB_Priority'Last - ORB_Core_Levels + 1 .. ORB_Priority'Last;
   --  ORB_Core_Priority defines the priority of some ORB key
   --  components. It is reserved to high priority loops, such as
   --  PolyORB.ORB main loop.

   Default_Core_Priority : constant ORB_Core_Priority;
   --  Default priority for ORB Core

   --  External priorities are derived from integer. They represent
   --  priority levels as defined by PolyORB's personalities.

   type External_Priority is new Integer;

   Invalid_Priority : constant External_Priority;

   --  These funcitons define mapping between ORB_Priority and
   --  External_Priority. When False, Returns indicate the mapping was
   --  not possible.

   type To_External_Priority_T is access procedure
     (Value    : in  ORB_Priority;
      Result   : out External_Priority;
      Returns  : out PolyORB.Types.Boolean);

   type To_ORB_Priority_T is access procedure
     (Value    : in  External_Priority;
      Result   : out ORB_Priority;
      Returns  : out PolyORB.Types.Boolean);

   To_External_Priority : To_External_Priority_T;

   To_ORB_Priority : To_ORB_Priority_T;

end PolyORB.Tasking.Priorities;
```

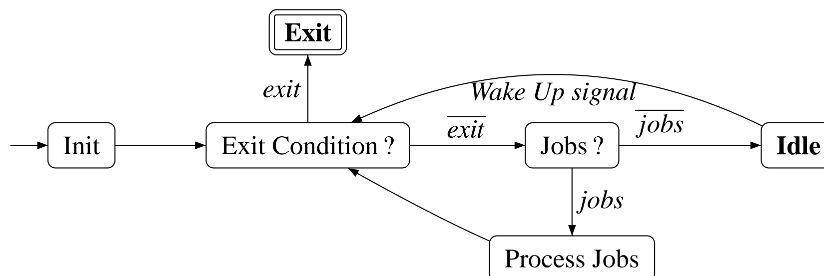


FIG. 5.2 – Automate d'états d'une tâche associée à une file d'exécution

Extrait de code 6 Spécification de `PolyORB.Lanes`

```
package PolyORB.Lanes is

   type Lane_Root is abstract tagged limited private;
   type Lane_Root_Access is access all Lane_Root'Class;
   --  Lane_Root is the root type for all lanes

   procedure Queue_Job
     (L             : access Lane_Root;
      J             :        Job_Access;
      Hint_Priority :        External_Priority := Invalid_Priority)
      is abstract;
   --  Queue job J in lane L, Hint_Priority defines a base priority to
   --  be used by the lane to queue J.

   function Is_Valid_Priority
     (L        : access Lane_Root;
      Priority :        External_Priority)
     return Boolean
     is abstract;
   --  Return True if a request at priority Priority can be handled by L.

   type Lane
     (ORB_Priority              : ORB_Component_Priority;
      Ext_Priority              : External_Priority;
      Base_Number_Of_Threads    : Natural;
      Dynamic_Number_Of_Threads : Natural;
      Stack_Size                : Natural;
      Buffer_Request            : Boolean;
      Max_Buffered_Requests     : PolyORB.Types.Unsigned_Long;
      Max_Buffer_Size           : PolyORB.Types.Unsigned_Long)
   is new Lane_Root with private;
   type Lane_Access is access all Lane'Class;

   --  [Lane's primitives]

   type Extensible_Lane is new Lane with private;
   --  An Extensible_Lane is a lane to which no thread are attached at
   --  startup. Thread may be attached to this lane, they will be used
   --  to process queued jobs.

   procedure Attach_Thread (EL : in out Extensible_Lane; T : Thread_Access);

   --  [Extensible_Lane's primitives]

   type Lanes_Set (Length : Positive) is new Lane_Root with private;
   --  A Lane_Set is a set of Lanes

   --  [Lane_Set's primitives]

   procedure Add_Lane
     (Set   : in out Lanes_Set;
      L     :        Lane_Access;
      Index :        Positive);
   --  Add lane L at position Index in Set

end PolyORB.Lanes;
```

5.2.6 Ordonnanceur de requêtes

Le module ORB_Controller du μBroker prend en charge l'affectation des travaux de l'intergiciel, dont les requêtes, à une tâche. Le module Request_Scheduler permet de contrôler de manière précise cette affectation. Il s'agit d'une politique étendant le comportement du μBroker.

L'acceptation d'un travail par l'intergiciel doit se faire en conformité avec l'ordonnanceur de l'exécutif et les paramètres de QdS de l'application.

Extrait de code 7 Spécification de PolyORB.Request_Scheduler

```
package PolyORB.Request_Scheduler is

   type Request_Scheduler is abstract tagged limited null record;
   type Request_Scheduler_Access is access all Request_Scheduler'Class;

   function Try_Queue_Request_Job
     (Self   : access Request_Scheduler;
      Job    :        PolyORB.Jobs.Job_Access;
      Target :        PolyORB.References.Ref)
     return Boolean
      is abstract;
   --  Try to have Job scheduled by Self, return False if the request
   --  scheduler refuses Job.

end PolyORB.Request_Scheduler;
```

La primitive Try_Queue_Request_Job permet ainsi de vérifier sur un nouveau travail peut être accepté, et prend en charge l'affectation de ressources particulières pour son traitement, par exemple le job peut être délégué à un objet Lane particulier dans le cas d'un intergiciel RT-CORBA.

Le test d'acceptation d'un job peut découler notamment de la mise en œuvre de tests issus de la théorie de l'ordonnancement, tels que *RMS* ou *EDF*, ou encore de l'acceptation de la requête par une file d'exécution.

Ce module est intégré au μBroker, et permet ainsi d'aiguiller une requête vers la tâche la mieux adaptée pour son traitement.

5.2.7 Conclusion

Ces différents éléments forment les briques élémentaires pour la construction d'un intergiciel temps réel. Leur implantation au sein de PolyORB permet d'étendre de manière naturelle certaines fonctions de base de l'intergiciel. En particulier, ils nous permettent d'étendre les services de base de l'intergiciel schizophrène et permettent la mise au point de nouveaux services plus adaptés aux applications TR^2E.

Nous montrons dans les sections suivantes comment ces éléments nous permettent de construire des services d'intergiciel déterministes, puis des personnalités adaptées pour PolyORB.

5.3 Services d'intergiciel TR^2E

Dans cette section, nous nous intéressons à la mise en œuvre de services pour intergiciels TR^2E. Nous présentons la construction d'un adaptateur d'objets déterministe, un protocole asynchrone et un mécanisme de transport temps réel.

Ces services tirent parti des briques élémentaires présentées à la section précédente et nous permettent de rendre déterministes les fonctions de l'intergiciel.

5.3.1 Adaptateur d'objets déterministe

L'adaptateur d'objets (*"OA"*) réalise la fonction d'activation. Nous avons montré à la section 3.3 comment ce service pouvait se réduire à la combinaison d'un dictionnaire et une fabrique d'objets.

Le dictionnaire permet de retrouver l'entité associée à un identifiant d'objets, la fabrique d'objets construira si nécessaire une entité à partir d'un identifiant si aucune entité n'est présente dans le dictionnaire. Enfin, l'adaptateur d'objets enregistre dans le dictionnaire les entités de l'application dont il a la charge.

Plusieurs politiques existent pour contrôler précisément son rôle. Ainsi, PolyORB propose un adaptateur d'objets minimal, le `Simple Object Adapter`. CORBA définit un adaptateur plus évolué, le POA (`Portable Object Adapter`), qui propose une hiérarchie d'adaptateurs ainsi que sept politiques de configuration avec leur sémantique. Enfin, RT-CORBA apporte de nouvelles politiques pour le contrôle des ressources d'exécution associées à chaque POA.

Les politiques définies autorisent plusieurs variétés de comportement. Nous n'en retenons que deux pour la suite : l'affectation d'identifiants, et la hiérarchie des adaptateurs d'objets. Les autres politiques contrôlent la création et la gestion des identifiants, elles sont peu pertinentes pour le déterminisme de l'adaptateur d'objets.

Le chapitre 3 a défini le fonctionnement de l'intergiciel. En particulier, la figure 3.5, page 70 détaille le cheminement d'une requête vers le servant.

La requête comporte une référence vers l'objet destinataire. Lorsque cette référence dénote un objet local (cas de la remontée d'une requête vers le servant destinataire), cette référence est réduite à un identifiant d'objet (`Oid`). Un `Oid` représente de manière non ambiguë le servant sur le nœud. Un `Oid` est une suite d'octets encodant l'identifiant de l'OA et l'identifiant de l'objet au sein de cet OA. Il comporte éventuellement plusieurs informations de QdS.

L'algorithme de recherche d'un servant se découpe en deux phases : *1)* recherche de l'OA, *2)* recherche du servant. La recherche du servant peut être grandement facilitée par la structure de données utilisées par l'OA.

Dans le cas du `Simple Object Adapter`, l'OA est un tableau enregistrant les différents servants. L'Oid est alors égal à la position du servant dans ce tableau. La recherche est alors triviale, et de complexité $O(1)$.

Dans le cas du `Portable Object Adapter`, l'OA a une structure arborescente, l'identifiant de chaque OA est une chaîne de caractères. Par ailleurs, l'Oid du servant peut être spécifié par l'utilisateur (cas de la politique `USER_ID` de CORBA). Ceci nous a amené à repenser la structure de l'OA.

Pour ce faire, nous avons utilisé les tables de hachages dynamiques parfaites pour stocker les différentes Oid. Ceci nous permet de retrouver les différentes entités en temps constant. Ainsi, nous utilisons trois tables de hachages différentes :

- Chaque OA dispose d'une table des servants. Si l'Oid du servant est généré par l'utilisateur on utilise une table de hachage, sinon un tableau suffit ;
- Chaque OA stocke la liste de ses OA fils dans une table de hachage parfaite ;
- Une table de hachage globale stocke tous les POA enregistrés, avec leur chemin absolu. Ceci nous permet de retrouver un OA sans parcours du graphe.

Ainsi, nous garantissons que la recherche d'un servant ou d'un POA se fait à temps constant. Notre implantation, basée sur l'utilisation des tables de hachage dynamiques parfaites nous permet de proposer un POA déterministe quelque soit la méthode utilisée pour la construction de l'Oid associée à un servant.

Contrairement à des intergiciels tels que TAO ou RT-Zen, l'utilisation de tables de

hachage dynamiques parfaites garantit un temps de recherche constant quelque soit la politique de nommage utilisée pour les POAs et les servants. Ceci offre d'avantage de souplesse lors du déploiement de l'application.

Ces différents éléments nous permettent de construire un POA dont le comportement lors de la phase de résolution du servant associé à une requête est déterministe.

5.3.2 Protocole asynchrone

L'utilisation d'un protocole synchrone introduit une interaction forte entre les nœuds impliqués lors d'un envoi de requête. Le synchronisme, du point de vue de l'application, peut amener un temps de blocage élevé. Par ailleurs, ce temps de blocage peut être non borné et dépend de l'état de chacun des nœuds impliqués. Il ajoute donc au non-déterminisme de l'application. Passer à un mode de transmission asynchrone permet de réduire ce temps et de contrôler le temps d'émission de chaque requête et le borner.

Dans cette section, nous fournissons quelques éléments sur l'implantation de services de protocole asynchrone. Elles ont été réalisées dans le cadre des travaux de DEA de Bertrand PAQUET [Paquet, 2003] que nous avons supervisés. Nous présentons MIOP et DIOP, deux personnalités protocolaires de PolyORB qui étendent les mécanismes du protocole générique GIOP défini par CORBA.

MIOP : unreliable Multicast Inter-ORB Protocol

CORBA définit un modèle de communication entre des objets répartis, basé sur le protocole GIOP (*General Inter-ORB Protocol*) [OMG, 2004b, chap. 15]. GIOP définit le format des messages ainsi que le format de représentation des données (CDR, *"Common Data Representation"*). Il suppose un canal de communication qui soit bi-directionnel, fiable et qui préserver l'ordre des messages.

IIOP (*Internet Inter-ORB Protocol*) est l'instance normalisée de GIOP. Il s'agit d'une instance de GIOP utilisant le protocole TCP/IP. Il impose une communication point-à-point entre les différents nœuds.

MIOP [OMG, 2002b] lève certaines des restrictions imposées par GIOP pour permettre une communication de groupe. Tout comme GIOP, MIOP est une extension de GIOP pour la communication de groupe. Il s'agit d'un protocole générique, qu'il faut instancier en utilisant un mécanisme de transport dédié.

MIOP propose ainsi UIPMC, *"Unreliable IP Multicast"*, une instance basée sur les concepts de Multicast/IP [Armstrong *et al.*, 1992], Le manque de fiabilité de ce protocole provient de l'absence de couche de gestion des erreurs par Multicast/IP. La perte de paquets n'est pas détectée au niveau protocole. Cependant, sous réserve de ne pas saturer le tampon de réception de la pile protocolaire, la perte de paquets est suffisamment faible pour être négligeable.

La personnalité MIOP/UIPMC implantée par PolyORB a une structure plus simple que celle des personnalités existantes. Le modèle de communication proposé par MIOP est unidirectionnel. La propagation de résultats est interdite. Ceci supprime tout besoin en un automate d'états pour gérer les messages de retour.

MIOP réutilise de nombreux éléments de GIOP qu'il étend au besoin. Ainsi, MIOP ne repose que sur une série de filtres pour envoyer une requête, de la phase d'emballage à la phase d'envoi. Des services connexes assurent la construction de références de groupes (`IOGR`) et de l'adaptateur d'objets référençant les groupes (`GOA`). Enfin, le

service de transport de PolyORB a été étendu pour supporter l'envoi et la réception de datagrammes au dessus de Multicast/IP.

Nous fournissons ici un aperçu des modules utilisés (figure 5.3) :

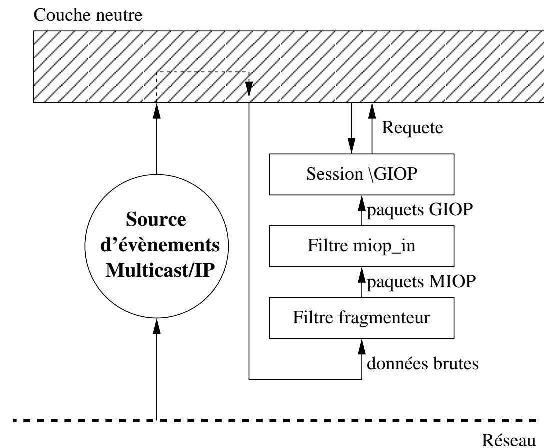


FIG. 5.3 – Vue de la pile protocolaire MIOP côté serveur

1. Les servants formant un groupe d'objets s'enregistrent auprès de l'adaptateur d'objets du nœud et du groupe multicast associé ;
2. Une référence de groupe associée à ce groupe d'objets est construite ;
3. Lorsqu'un nœud souhaite contacter le groupe d'objets, il prépare sa requête : le protocole GIOP prépare une requête générique, le filtre `miop_out` la fragmente pour que chaque fragment corresponde à un datagramme, le service transport l'envoie au groupe multicast concerné ;
4. les opérations duales sont effectuées côté serveur par le filtre `miop_in`.

L'implantation réalisée est conforme aux dernières spécifications de MIOP et interopérable avec d'autres réalisations, telle que celle de TAO.

Une série de mesures a évalué le taux de perte de messages lors de la saturation des tampons de réception. Lors de l'envoi en rafale de 1000 messages, le taux de perte de messages est de l'ordre de 45%, il redevient nul lorsqu'un délai raisonnable de quelques millièmes de secondes entre chaque message est maintenu.

DIOP : Datagram Inter-ORB Protocol

DIOP est une évolution de MIOP : DIOP fournit un mécanisme d'envoi de requêtes point-à-point asynchrone, basé sur UDP/IP.

DIOP réutilise de nombreux mécanismes déjà mis en place :
- gestion de références de GIOP pour identifier un nœud ;
- protocole simple pour l'envoi de messages unidirectionnels issus de MIOP ;
- extension du service de transport pour supporter UDP/IP ;

L'implantation de la personnalité DIOP montre comment adapter à faible coût la personnalité GIOP et les différents services existants pour supporter de nouveaux mécanismes de transport. Ainsi, DIOP requiert moins de 1000 lignes de codes dédiées. Nous utiliserons cette capacité à la section suivante pour proposer une personnalité protocolaire temps réel.

Ces deux instances ont démontré comment implanter un protocole asynchrone au sein de PolyORB. Ces travaux confortent notre vue de l'architecture schizophrène que nous avons proposée au chapitre 3 : les modules de personnalités protocolaires ont été définis à l'aide de filtres et automates simples.

5.3.3 Couche transport temps réel

Dans cette section, nous présentons une extension des mécanismes de transport introduits dans PolyORB visant le déterminisme. Nous avons retenu une variante dynamique de TDMA, offrant un bon compromis entre flexibilité et utilisation des ressources.

Définition d'une couche de transport pour le temps réel

Le modèle OSI [ISO, 1995], définit un modèle canonique d'un protocole de communication. Il définit un protocole de communication comme étant formé de sept couches, chacun ayant un rôle précis. Ainsi, ce modèle affecte à la couche de transport la responsabilité du transfert transparent de données entre deux nœuds du système, la gestion des erreurs et le contrôle du flux échangé.

Les mécanismes de transport ne sont en général pas adaptés au temps réel. La survenue aléatoire des erreurs et leur gestion ainsi que les émissions sporadiques de messages et les mécanismes d'accès au medium introduisent un coût difficilement quantifiable sur le transfert des données.

Ainsi, il est nécessaire de concevoir un mécanisme qui préserve la bande passante disponible dans le canal, tout en fournissant des garanties suffisantes sur la QdS offerte.

Protocole TDMA dynamique

Dans [Pedreiras *et al.*, 2002], les auteurs proposent une classification de différents mécanismes permettant d'étendre ethernet et le rendre temps réel. Les auteurs introduisent FTT Ethernet (*"Flexible Time Triggered Ethernet"*). Ce protocole repose sur l'émission d'une super-trame de synchronisation déclenchant l'émission de messages synchrones ou asynchrones. L'émission synchrone est ordonnancée, et plus prioritaire que les émissions asynchrones. Ces dernières n'ont lieu que si des ressources sont disponibles. Ce protocole n'est implanté qu'au dessus du micro-noyau S.Ha.R.K [Gai *et al.*, 2001]. Nous souhaitons proposer une architecture portable, utilisable dans le cadre de nos travaux sur de multiples plates-formes.

Nous ne retenons de ce protocole que l'utilisation d'une super-trame permettant la synchronisation et l'émission de messages synchrones. Ceci nous permet de supporter un mécanisme proche de TDMA, relativement simple. Par ailleurs, ce modèle offre un bon compromis entre les performances atteignables et l'utilisation des ressources. Dans un contexte où les ressources sont allouées statiquement, ce modèle facilite l'analyse d'ordonnancement de l'application construite.

Nous avons modifié légèrement le protocole décrit dans FTT-Ethernet pour permettre l'enregistrement dynamique des nœuds, et ne pas se limiter à l'affectation d'un unique slot d'émission pour chacun des nœuds.

Ces différentes modifications ne remettent pas en cause le déterminisme de ce protocole, démontré dans le cas du protocole FTT-Ethernet. Notre proposition ne fait que modifier le mécanisme d'initialisation de ce protocole sans modifier son fonctionnement en régime normal.

Hypothèses

La variante du protocole TDMA que nous décrivons repose sur plusieurs hypothèses simplificatrices, que nous avons retenues avant de disposer d'un prototype intégrable à PolyORB :

– *panne d'une station* : ce cas n'est pas traité, il découle d'une erreur grave du système ;
– *station "folle"* : le cas où une station emmétrait à un intervalle de temps qui ne lui est pas attribué n'est pas géré : il résulte d'une erreur de configuration du système, ou une erreur de fonctionnement grave d'un des nœuds du système (problème d'horloge temps réel interne par exemple) ;
– *perte d'un paquet* : ce cas n'est pas géré, et est délégué à la couche de transport ;
– *fragmentation* : les messages forment des unités indépendantes, leur taille est bornée par $Payload_{max}$.

Par ailleurs, nous nous plaçons dans le cas où les différents nœuds de l'application sont sur un sous-réseau dédié et isolé, sans perturbation extérieure. En particulier, les seules stations émettant sur le réseau sont celles participant au protocole TDMA.

Messages échangés

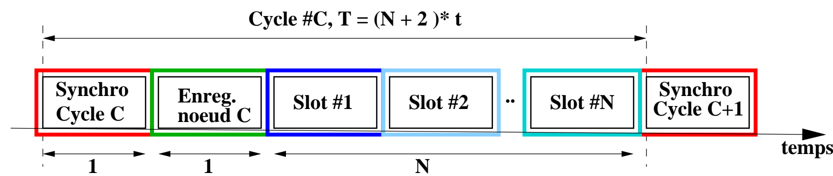


FIG. 5.4 – Cycle élémentaire du protocole TDMA

La figure 5.4 détaille le déroulement d'un cycle élémentaire du protocole TDMA.

Chaque cycle est indexé par un entier $C \in \mathbb{Z}/\mathbb{Z}_N$, où N est le nombre maximal de stations pouvant prendre part au système.

Chaque station se voit affecter un identifiant unique $i \in \{1 \ldots N\}$.

1. Un message de synchronisation est émis à l'instant T_C par le maître à destination de toutes les stations ; ce signal de synchronisation contient la valeur du cycle courant et les résultats d'affectation de slots d'émission ;
2. La station dont l'identifiant correspond à la valeur du cycle courant peut s'enregistrer auprès du maître et demander l'affectation d'un slot d'émission ;
3. Chacune des stations envoie ses messages aux instants $T_C + t * (2 + N_{slot})$, où N_{slot} correspond à un numéro de slot affecté par le maître. t est la durée d'un slot.

Notons que la valeur de t et la taille maximale des messages ($Payload_{max}$) dépendent du mécanisme de transport utilisé, et de l'implantation qui sera réalisée.

Chaque nœud voit son timer remis à zéro à chaque cycle, l'attente est au plus de $(N+2) * t$, la dérive d'horloge doit donc être non perceptible sur cet intervalle. Ceci impose une contrainte sur la configuration du protocole.

Les performances obtenues sont donc comparables au protocole TDMA de base, tout en permettant d'avantage de souplesse dans l'affectation des slots d'émission (k slots pour un nœud par exemple).

Fonctionnement du maître

Ce nœud réalise deux fonctions effectuées à des instants différents :
- entre les instants $T_C + t$ et T_{C+1} les demandes d'enregistrement sont traitées, le message de synchronisation préparé ;
- à l'instant T_{C+1} le message de synchronisation est émis.

Ces deux actions sont réalisées par deux tâches cycliques, la première traitant les demandes d'enregistrement, la seconde traitant l'émission du message. Ceci permet notamment de contrôler la priorité à laquelle elles s'exécutent, et simplifie l'automate associé à chaque action, facilitant son analyse temporelle.

Le message de synchronisation est de taille fixe, aucune allocation dynamique de mémoire n'est nécessaire. De même aucune primitive de synchronisation n'est nécessaire, les deux tâches étant actives à des instants différents par construction. Notons que dans le cas contraire le système n'aurait pas le temps de traiter les demandes, et raterait donc des échéances. Il y aurait donc une erreur de conception.

Par construction, le maître interagit de manière transparente avec l'utilisateur. Celui-ci ne dispose donc que d'une interface minimale pour sa configuration et son initialisation. Une fois initialisé, le maître s'exécute jusqu'à extinction du système.

Fonctionnement du client

Le client réalise trois actions en parallèle :
- attendre le message de synchronisation ; et armer le timer interne lorsque ce message est reçu ;
- attendre tout message d'autres nœuds ;
- émettre un message aux instants autorisés.

De même que pour le maître, chacune de ces actions est réalisée par une tâche dédiée, simplifiant la mise au point du code et son analyse.

La figure 5.5 présente l'architecture du client. Il doit fournir une interface suffisante pour l'ouverture de canaux de communication, et l'émission et la réception de données.

Sans perte de généralité, nous avons opté pour une interface proche de l'interface de programmation des sockets, telle qu'elle est proposée par le paquetage `GNAT.Sockets`. Ceci nous garantit une compatibilité au niveau source avec des programmes basées sur ce paquetage, et facilite l'inclusion du protocole dans d'autres logiciels.

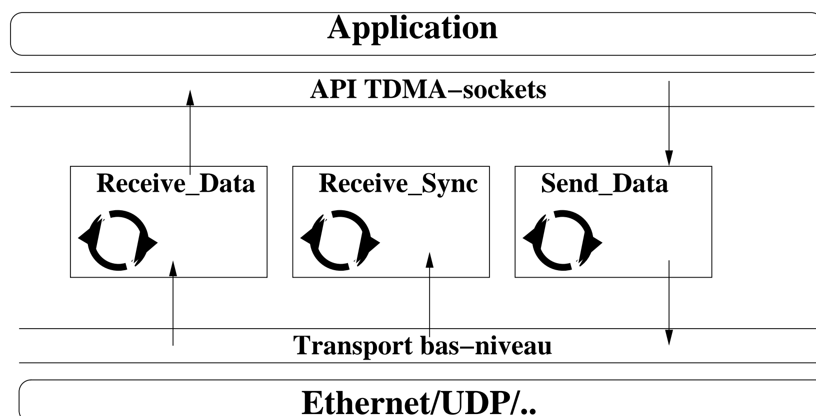


FIG. 5.5 – Fonctionnement du protocole TDMA

L'implantation sépare deux facettes du protocole :

1. les mécanismes de configuration des différents canaux de communication, les primitives d'émission et de réception des messages ;
2. la mise en œuvre des tâches assurant le bon fonctionnement du protocole. Ces trois automates s'exécutent en arrière plan, et sont sollicités par l'utilisateur pour l'envoi ou la réception de données.

Ce découpage est intéressant car il nous offre la possibilité de modifier facilement le code des différents automates et ainsi supporter différentes sémantiques pour le traitement des messages : délivrance différée ou immédiate, priorité des files de messages.

Mise en œuvre et tests

L'architecture retenue est la suivante :

– `ip_transport` : ce paquetage fournit l'accès aux primitives de communication du système basé sur UDP/IP. Les constructions de ce paquetage sont utilisées au travers d'une façade par les autres paquetages. Ceci garantit l'indépendance de l'implantation vis à vis du protocole UDP/IP, et permet l'utilisation d'autres mécanismes de transport, dont ethernet "nu".
– `TDMA.Master` : il fournit les points d'entrée pour déployer un nœud maître ;
– `TDMA.Client` : il fournit une API proche de celle des `sockets` et de son interface pour le langage Ada 95 `GNAT.Sockets`. Ce choix facilite l'adaptation pour ce protocole d'applications existantes.

Une mise en œuvre de ce protocole a été réalisée, elle utilise les bibliothèques de PolyORB comme support pour la gestion des tampons mémoire et de la concurrence. Il représente un total de $1,300$ SLOCs. L'empreinte mémoire de la bibliothèque d'un client est de $16Ko$, et pour le maître de $13Ko$, auxquels s'ajoutent $8Ko$ pour la couche de transport basée sur les sockets BSD.

Le débit utile maximum est donné par :

$$\frac{8*N*Payload_{max}}{(N+2)*t} \tag{5.5}$$

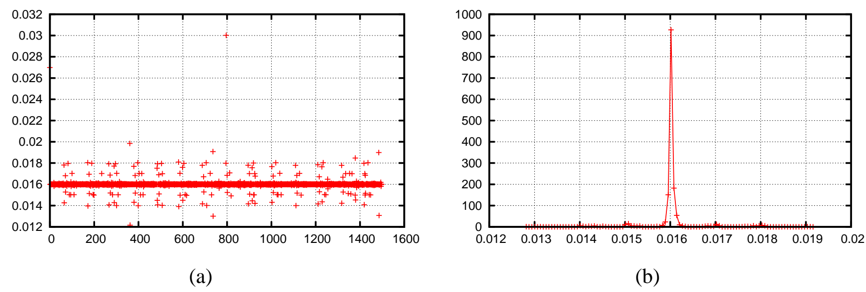
Dans le cadre d'un transport basé sur ethernet 100BaseT, $Payload_{max} = 1492$ octets, t vaut au minimum $8*Payload_{max}/10^8 = 119,36\mu s$. Cette valeur est cependant peu réaliste : elle ne prend pas en compte le traitement effectué sur chacun des nœuds, ni la latence éventuelle du lien (fonction de la distance, des chipsets, etc). Nous retenons pour t une valeur de l'ordre de $200\mu s$, obtenue empiriquement.

Ainsi, le débit utile sera pour $N = 1$ est $19.89Mbit/s$. Il s'agit d'un cas où une unique station émet un message. Notons que le débit utile croit avec le nombre de stations et vaut $59.68Mbit/s$ lorsque N tend vers l'infini.

Ce protocole a été testé sur plusieurs exemples : diffusion d'un flux vidéo *"ASCII Art"*[1], échange de messages entre N nœuds suivant une topologie en anneau.

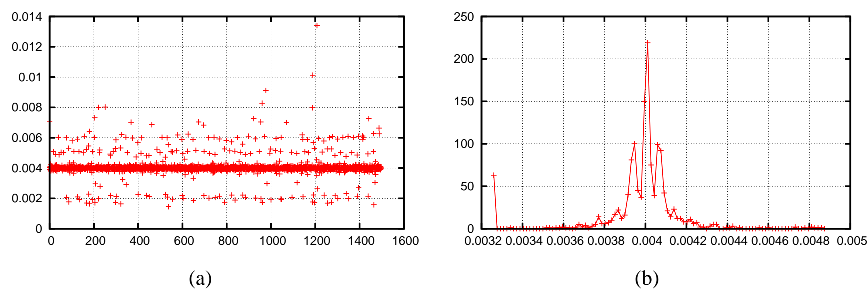
Nous fournissons ici les performances obtenues lors de l'échange d'un message entre deux nœuds, suivant un ping pong. La plate-forme de test est composée de quatre nœuds : le maître, deux nœuds qui effectuent mille fois la boucle émettre un message, le lire et le réemmetre et un nœud "initiateur" qui se charge d'envoyer un premier message à un des noeuds. Les mesures ont été effectuées sur un ensemble de stations Linux (figures 5.6 et 5.7).

[1] flux vidéo dont les blocs de pixels sont remplacés par des caractères ASCII



(a) (b)

FIG. 5.6 – Mesures et dispersion pour `slot_duration` = 2 ms



(a) (b)

FIG. 5.7 – Mesures et dispersions pour `slot_duration` = 400 μs

Ces tests démontrent l'exactitude de l'implantation, et l'importance de la valeur du paramètre t. Ainsi, pour des plates-formes PC de type Pentium-IV, une valeur de $200\mu s$ est une limite acceptable en dessous de laquelle il est difficile de descendre sans compromettre l'exactitude du fonctionnement des différents nœuds.

Une valeur plus importante pénalise les performances locales du fait du traitement des messages de synchronisation qui s'effectue à chaque cycle. Enfin, il pose une contrainte plus forte sur la précision de l'horloge des différents nœuds et sur la durée autorisée pour le traitement des fonctions d'émission. Il y a donc une phase de dimensionnement de la couche de transport en fonction des besoins de l'application et des ressources disponibles pour sa réalisation.

Une seconde phase de tests nous a amené à construire une personnalité TDMAIOP, sur le modèle de la personnalité DIOP. Elle permet à des nœuds PolyORB d'interagir suivant le protocole TDMA. Ceci permet de renforcer le déterminisme de l'échange de messages GIOP. Nous avons ainsi pu faire interagir des nœuds CORBA et vérifier comme pour le ping pong une valeur moyenne conforme aux paramètres du protocole, et une faible dispersion.

Du fait de la similitude entre l'API du protocole, et celle du paquetage `GNAT.Sockets`, la transition a été simplifiée, et peu de code spécifique a été écrit.

Conclusion

Ce travail nous a permis d'intégrer un mécanisme de transport déterministe à PolyORB. Les résultats obtenus nous ont permis de valider notre prototype sur des exemples simples, jusqu'à l'intégration de ce protocole pour une application CORBA.

L'implantation réalisée permet de substituer la couche de transport qui sert de support aux mécanismes, permettant ainsi d'utiliser directement un chipset ethernet. Ce travail fait l'objet de travaux en cours au moment de la rédaction dont un port de ce protocole pour le noyau temps réel ORK.

Par ailleurs, ce protocole impose certaines contraintes sur l'exécution temporelle de l'application. L'émission d'une requête est bornée par un pire temps de $(N+2)*t$. Remarquons par ailleurs qu'un échange requête/réponse va nécessiter dans le pire cas $2*(N+2)*t+T_{req}$.

Remarquons cependant que ce temps peut être réduit suivant l'entrelacement des messages retenus. Ainsi, pour N suffisamment grand, le nœud 1 peut émettre à l'instant dédié au slot 1, le nœud 2 répondre à l'instant du slot $k \leq N$, si $T_{req} < (k-1)*t$. La détermination correcte de k est un problème complexe qui dépend (entres autres éléments) de la charge du nœud 2.

Une extension possible à cette implantation serait d'intégrer les mécanismes du protocole d'anneau à jeton à priorité [Martínez *et al.*, 2003]. Ceci se ferait simplement par substitution des automates présents sur chaque nœud par ceux définis par ce protocole. Ceci permettrait de supporter plusieurs mécanismes de transport dans une infrastructure unique, offrant ainsi d'avantage de configurabilité à l'application.

5.4 Conclusion

Ce chapitre a détaillé les différents travaux et extensions des mécanismes de PolyORB visant à le rendre déterministe. La mise à disposition de modules élémentaires permet de composer des personnalités déterministes.

Nous avons dans un premier temps présenté les différentes entités logicielles de base utilisées : langage Ada 95, les règles d'écriture de code que nous avons utilisées. Nous avons ensuite présenté les mécanismes et structures de données que nous utilisons : mécanismes de configuration, tables de hachages dynamiques parfaites, gestion de la qualité de service, des priorités, files d'exécution et ordonnanceur de requêtes.

Par la suite, nous avons présenté la construction d'un adaptateur d'objets temps réel, un protocole asynchrone et un mécanisme de transport temps réel.

L'assemblage de ces éléments nous permet de définir au chapitre suivant des personnalités protocolaires et applicatives à même d'exhiber des propriétés temps réel, nous permettant ainsi la construction d'instances d'intergiciel pour applications TR^2E.

Notons cependant que l'analyse d'ordonnancement d'un système TR^2E est un problème complexe et qui sort du cadre du présent mémoire. Nous avons cherché ici à mettre en place les éléments rendant notre implantation déterministe et analysable. L'analyse est laissée à la charge de l'utilisateur qui dispose dès lors de tous les éléments utiles.

6

Construction d'intergiciels pour systèmes TR^2E

Sommaire

6.1 Mode opératoire 125
6.1.1 Composants de PolyORB 126
6.1.2 Assemblage d'une configuration 129
6.2 Construction d'un intergiciel minimal 131
6.2.1 Définition 131
6.2.2 Analyse du nœud construit 132
6.3 Construction d'un intergiciel RT-CORBA 133
6.3.1 Définition 133
6.3.2 Dépendances sur l'exécutif 135
6.3.3 Critiques de la spécification RT-CORBA 135
6.3.4 Conclusion 138
6.4 Prototypes 138
6.4.1 RT-DSA 138
6.4.2 DDS 140
6.5 Conclusion 141

Ce chapitre explore différents aspects de l'implantation réalisée. En particulier, il s'intéresse au déploiement de l'intergiciel pour deux configurations distinctes : configuration minimale, pouvant être embarquée sur des noeuds disposant de peu de ressources ; configuration RT-CORBA, pour des applications complètes.

6.1 Mode opératoire

Nous avons montré précédemment comment l'architecture que nous proposons sépare les différentes fonctions de l'intergiciel, et présente plusieurs services ou "briques intergicielles" pouvant être combinés pour former une instance de l'intergiciel.

Dans ce chapitre, nous montrons comment plusieurs intergiciels peuvent être construits à partir d'un ensemble de modules par dérivation/extension d'un modèle générique simple que nous avons présenté au chapitre 3. Cette évolution est dirigée par le besoin d'une fonctionnalité qui découle des besoins de l'utilisateur, permettant de passer d'un modèle "indépendent de toute application" de l'intergiciel, à un modèle "spécifique".

La figure 6.1 fournit une vue d'ensemble du processus. Il reprend un déroulement analogue aux travaux de modélisation et vérification que nous avons présentée au chapitre 3.

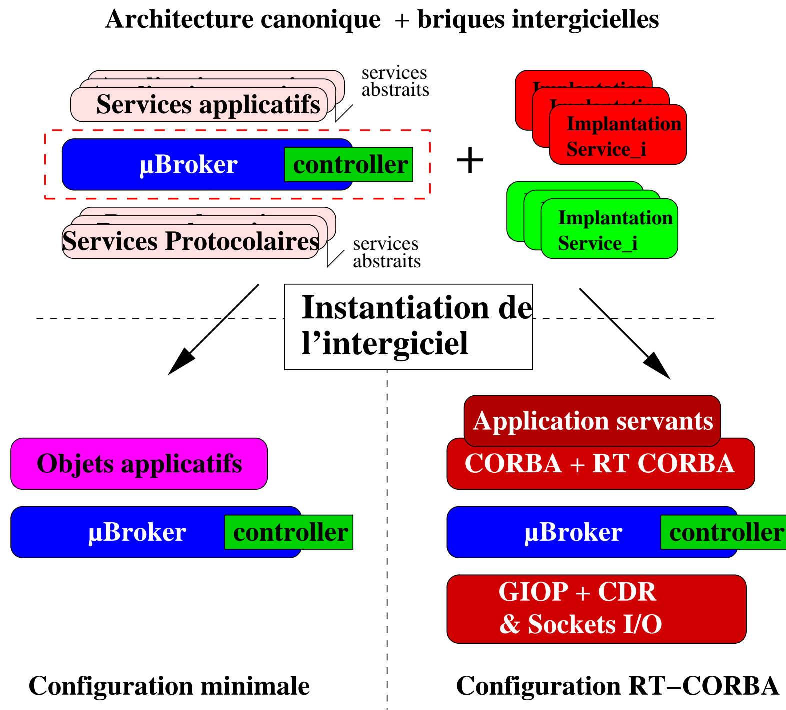


FIG. 6.1 – Construction de deux intergiciels distincts

Dans un premier temps, nous déterminons les éléments nécessaires à l'instance à construire : personnalités applicatives ou protocolaires, primitives de concurrence, politiques de QdS, etc. Ces éléments sont ensuite assemblés lors des phases de compilation et d'édition de liens réalisées par la chaîne de production. Enfin, le système d'initialisation de PolyORB prend en charge la mise en route du nœud.

6.1.1 Composants de PolyORB

Cette section fournit une classification des nombreux modules d'intergiciel que PolyORB propose. Nous fournissons une liste exhaustive de ces éléments dans le guide d'utilisation de PolyORB [Hugues, 2004]. Ces modules sont regroupés suivant le type de fonctionnalités qu'ils apportent. On peut les classer en deux familles : généraux et servant à toutes les configurations de personnalités (politiques du μBroker), ou spécifiques et nécessaires à une ou plusieurs personnalités (fonctions de représentation CDR pour GIOP).

Configuration

Ces entités fournissent des fonctions annexes pour la configuration du nœud,
– `PolyORB.Parameters.File` lit le fichier de configuration de PolyORB ;

– `PolyORB.Parameters.Environment` recherche les variables d'environnement utilisable pour configurer le nœud ;
– `PolyORB.Log.*` configure la sortie utilisée pour la mise au point d'un nœud.

Primitives de concurrence

Ces modules fournissent les primitives de base pour la concurrence, issus des primitives POSIX : verrous, variables conditionnelles, threads.

Ces primitives sont proposées pour trois exécutifs différents : mono-tâche, profil de concurrence complet du langage Ada 95, profil restreint compatible avec Ravenscar.

Politiques du μBroker

Ces politiques implantent les éléments présentés à la section 3.4.2. L'utilisateur a le choix entre les politiques `No Tasking`, `Half Sync/Half Async`, `Leader/Followers` et `Workers`.

Les dépendances entre briques de l'intergiciel assurent la compatibilité entre politiques du μBroker et la disposition de primitives de concurrence.

Paramètres des requêtes

La structures des requêtes manipulées par PolyORB est suffisamment générique pour permettre l'interopérabilité entre les différentes personnalités. Ceci impose que le type requêtes comporte suffisamment d'information pour se conformer à la sémantique des requêtes de chaque personnalité : éléments de QdS, notion d'exception, de valeur de retour associée à une requête bi-directionnelle, etc.

Ceci conduit à une inflation de cette structure, qui contient alors des données trop spécifiques ou inutiles à certaines configurations. Pour répondre à ce problème, plusieurs éléments d'une requête peuvent être rendus optionnels :
– *exceptions* : la notion d'exception est étrangère à la couche neutre, il s'agit d'une information retournée lorsque le traitement d'une requête n'a pu être complétée. Sous ces conditions, il peut être supprimé lorsqu'aucune personnalité n'utilise ce mécanisme, par exemple pour Minimum CORBA, configuration minimale.
– *Qualité de service* : les paramètres de qualité de service permettent de contrôler le traitement de requêtes. Ils sont le fait de l'application, ou de la personnalité applicative qui les définit, l'intergiciel les met en place. Ils peuvent donc être rendus optionnels aussi.

Types de données

Les fonctions de représentations et le cœur de l'intergiciel incluent le support pour de nombreux types : types simples (booléens, entiers, flottants, etc) ou complexes (tableaux, structures, unions, etc). Le support de ces différents types impose de disposer des mécanismes de représentation associés et éventuellement d'un support par l'exécutif voire le processeur, par exemple dans le cas de nombres flottants ou entiers de grande taille.

Néanmoins, nous notons qu'une application n'a le plus souvent besoin que d'un sous-ensemble de ces types. Ici encore, il apparaît intéressant de rendre optionnels certains de types de données de l'utilisateur. Ceci permet de contrôler et supprimer certains éléments de code inutiles.

Une solution consiste alors à utiliser certaines informations extraites de l'application et les utiliser pour générer des versions des paquetages dédiés à la gestion des types spécifiques à l'application.

Cette information est dans certains cas fournie par l'utilisateur, par exemple lors de la génération de code à partir d'un fichier IDL, ou à la compilation d'un programme écrit pour l'annexe des systèmes répartis du langage Ada 95.

Plusieurs expériences ont été réalisées pour étudier l'impact de la suppression de ces types sur la taille de l'exécutable, nous les présentons à la section 7.3.2.

Composants de personnalités

Les personnalités de PolyORB ont chacune des dépendances strictes sur les modules de l'intergiciel. Nous listons ici certaines de ces dépendances :

- *Transport* : PolyORB dispose de trois mécanismes de transport.
  - `Datagram`, dont la sémantique est celle de UDP/IP, qui est nécessaire aux personnalités DIOP et MIOP.
  - `Connected` (TCP/IP) qui est utilisé par IIOP et SOAP.
  - `TDMA` dont dépend la personnalité TDMAIOP.
- *Adaptateur d'Objets* : PolyORB propose trois implantations. Ceux-ci fournissent différents niveaux de fonctionnalités.
  - Le `SOA` ("Simple Object Adapter") permet d'enregistrer un objet applicatif (un `servant`) dans un tableau. La personnalité MOMA peut utiliser cette implantation, ou au contraire utiliser le `POA`.
  - Le `POA` reprend les concepts du "Portable Object Adapter" introduit par CORBA pour fournir d'avantage d'options de configuration, et une hiérarchie d'adaptateurs d'objets. Il a été optimisé pour que la recherche d'un adaptateur s'effectue en temps constant. Il est utilisé par les personnalités AWS, CORBA, Ada/DSA et peut être utilisé par MOMA.
  - Le `RT-POA` étend le `POA` en lui ajoutant des politiques de configuration propre au temps réel (notion de priorité, de file, etc). Cette extension fournit les constructions nécessaires pour RT-CORBA.
- *Exportation de références* : PolyORB permet d'exporter des références sous forme de chaînes de caractères suivant trois formats : `IOR`, `corbaloc` qui sont définis par CORBA, et `URI`. Ils peuvent être utilisés par toutes les personnalités. L'interopérabilité avec des nœuds basés sur d'autres intergiciels (par exemple `URI` pour les applications Web), où lorsque la référence doit comporter des informations de QdS précises (cas des `IOR` utilisées par RT-CORBA).

Choix des personnalités

La sélection des personnalités s'opère différemment suivant le type de personnalités considérées :

- *applicative* : son utilisation sous-entend une dépendance explicite vers des modules de cette personnalité.. Sa sélection se fera donc en tirant par transitivité les bibliothèques nécessaires à son fonctionnement (cas de CORBA ou MOMA) ; ou en utilisant le générateur de code associé (cas de Ada/DSA et du générateur *IDL-vers-Ada* de CORBA).
- *protocolaire* : la mise à disposition d'une personnalité protocolaire aux composants applicatifs locaux, et aux autres nœuds, est une action volontaire et explicite. Le nœud doit en effet initialiser le point d'accès et les fabriques de proto-

coles associées. Cette action se fait en enregistrant auprès du service d'initialisation de PolyORB les paquetages associés à ces fonctions.

Notons que le choix d'une ou plusieurs personnalités tirera par transitivité certains des modules de personnalités que nous avons présentés à la section précédente.

Discussion

Il est important de noter que ces différentes briques ajoutent de nouvelles dépendances sur l'exécutif Ada 95 ou sur des fonctions de l'exécutif sous-jacent (existence d'un système de fichiers, support de la concurrence, etc). Ceci fait que l'ensemble des fonctionnalités requises pour un nœud dépendra aussi grandement des modules optionnels qui seront choisis.

Les dépendances entre modules peuvent par ailleurs poser un frein au déploiement d'un nœud : résoudre ces dépendances suppose de connaître précisément les contraintes et besoins de chacun. L'utilisation d'un langage de description d'architecture (ADL) permettrait de s'affranchir de cette complexité en fournissant un outil d'aide à la sélection des modules à partir de besoins exprimés par l'utilisateur. Ces travaux font l'objet d'une thèse en cours à l'ENST. Nous discutons ce point en conclusion de notre mémoire.

Chaque brique intergicielle ajoute éventuellement une dépendance sur des fonctionnalités de l'exécutif Ada 95. Ceci conduit naturellement à la définition d'un ensemble minimal de fonctionnalités nécessaires à notre intergiciel. Nous caractérisons cet ensemble à la section suivante, en analysant une configuration minimale.

6.1.2 Assemblage d'une configuration

Dans cette section, nous présentons les différentes étapes de construction d'un nœud basé sur PolyORB.

Nous avons détaillé dans les sections précédentes les briques intergicielles pouvant être utilisé par un nœud. La figure 6.2 fournit les différentes étapes nécessaires. Le processus se découpe en deux étapes : *"compilation"* et *"initialisation"*. Chacune met en place une partie des modules d'intergiciel utilisés.

Compilation

Cette première phase vise à construire un exécutable. Pour ce faire, l'utilisateur sélectionne les différents modules qu'il souhaite utiliser. La sélection se fait en se reposant sur le mécanisme de dépendance mis en place par le modèle de compilation du langage Ada 95. Il suffit de rajouter la dépendance sur une unité Ada 95 pour que celle-ci soit rajoutée à la liste des modules utilisés par le nœud.

Par transitivité des dépendances entre paquetages définie par Ada 95, d'autres paquetages seront alors éventuellement rajoutés : les éléments ainsi sélectionnés permettent de construire une application cohérente du point de vue du compilateur : tous les symboles doivent être résolus.

L'extrait de code 8 montre comment l'unité `PolyORB.Setup.Base` ajoute une dépendance sur différents modules de PolyORB. L'exécutable ainsi construit comporte les entités sélectionnées par l'utilisateur, et qui répondent aux besoins de l'application. Il s'agit d'un *"intergiciel paramétrique"* : certains modules peuvent être configurés et adaptés lors de l'initialisation du nœud.

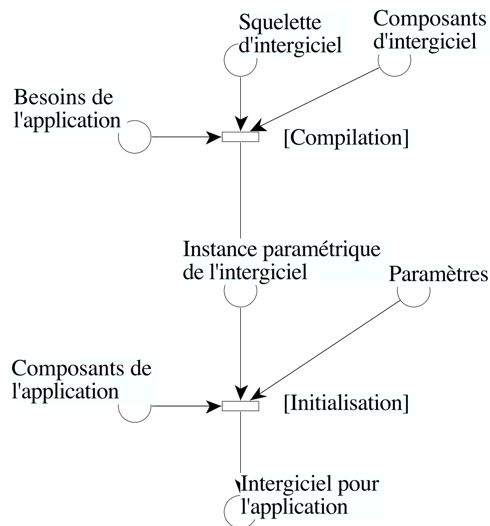


FIG. 6.2 – Construction d’un nœud PolyORB

Extrait de code 8 Implantation de `PolyORB.Setup.Base`

```
with PolyORB.Log.Stderr;
pragma Warnings (Off, PolyORB.Log.Stderr);
pragma Elaborate_All (PolyORB.Log.Stderr);

with PolyORB.Parameters.File;
pragma Warnings (Off, PolyORB.Parameters.File);
pragma Elaborate_All (PolyORB.Parameters.File);

with PolyORB.Parameters.Environment;
pragma Warnings (Off, PolyORB.Parameters.Environment);
pragma Elaborate_All (PolyORB.Parameters.Environment);

with PolyORB.Parameters.Registry;
pragma Warnings (Off, PolyORB.Parameters.Registry);
pragma Elaborate_All (PolyORB.Parameters.Registry);

package body PolyORB.Setup.Base is

end PolyORB.Setup.Base;
```

Initialisation

La seconde phase vise à réaliser l'initialisation du nœud, et être prêt à exécuter le code de l'utilisateur. Nous utilisons pour ce faire les mécanismes mis en œuvre par T. QUINOT pour l'initialisation de PolyORB.

Ainsi, à chaque unité est associée une portion de code exécutée lors de l'élaboration de la partition. Ce code se charge d'enregistrer auprès du gestionnaire d'initialisation une routine qui configure cette unité, ainsi qu'une liste de dépendances vers d'autres unités. Ce graphe de dépendance est ensuite parcouru et chacune des fonctions d'initialisation est exécutée.

Les dépendances entre unités assurent que les unités `PolyORB.Parameters.*` sont initialisées en premier. Ceci permet aux autres unités de lire et utiliser les tuples positionnés par le fichier de configuration, les variables d'environnement, ou tout autre mécanisme de stockage de paramètres, tels que LDAP.

Une fois les différentes routines d'initialisation exécutées, le nœud est complètement configuré, et est apte à exécuter le code de l'utilisateur, ou traiter les requêtes.

6.2 Construction d'un intergiciel minimal

L'approche par composition nous permet de construire des nœuds PolyORB adaptés à une famille de besoin. Nous nous intéressons dans cette section à la construction d'un intergiciel qui soit minimal.

6.2.1 Définition

Pour cette configuration, nous retenons le strict minimum des fonctionnalités permettant d'instancier un nœud basé sur PolyORB (c-à-d pouvoir obtenir un exécutable), et disposer d'un nœud capable de traiter des requêtes.

Nous retenons la configuration suivante : *un servant, disposant d'une unique méthode `echoString`, utilisant un `Simple Object Adapter`, mono-tâche, sans personnalité applicative ni protocolaire ni modules de configuration*. Ces seules entités suffisent à construire un nœud complet, qui peut être initialisé.

Par construction, ce nœud est minimal : il ne comporte aucune personnalité, chacun des modules sélectionnés est par construction la version la plus réduite possible de la fonction qu'il remplit.

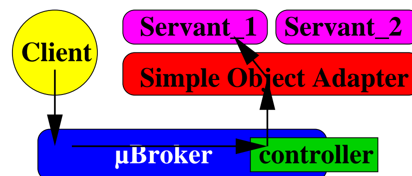


FIG. 6.3 – Construction d'un nœud PolyORB minimal

Le nœud ne peut être utilisé que localement (figure 6.3). Nous retenons qu'un intergiciel permet d'enregistrer des entités applicatives (*"servants"*). Dans cette configuration, PolyORB joue alors le rôle d'une infrastructure orientée composants minimale, tel que Fractal [Coupaye *et al.*, 2002]. Le code utilisateur que nous avons placé effectue

une requête sur ce composant local. Ceci permet de tester que le chemin d'invocation peut être complété.

6.2.2 Analyse du nœud construit

Cette configuration minimale nous permet de fournir quelques éléments sur la couche neutre, cœur de PolyORB.

Notons d'abord que le code associé à ces modules intergiciels représente 61 paquetages Ada 95, pour un total $11,039$ lignes de code significatives une fois les commentaires et lignes vides supprimées[1].

Un second aspect de qualité du code est le type de restrictions du langage qui sont violées ou validées.

Le langage Ada 95 fournit une liste de restrictions du langage qui peuvent être vérifiées par le compilateur ou le système exécutif utilisé. Ces restrictions permettent de s'assurer de la qualité du code écrit, en particulier que certaines constructions du langage jugées dangereuses ne sont pas présentés dans une application jugée critique.

`gnatls`, outil de la chaîne de compilation de GNAT, permet de déterminer l'ensemble des restrictions violées par les différents paquetages formant un exécutable. Notons que certaines des restrictions violés par PolyORB sont définies par GNAT, et non par le manuel de référence du langage. Il s'agit de restrictions supplémentaires définies pour répondre à des besoins en qualité de code qui ont été formulées par les utilisateurs de ces outils.

Nous avons déterminé que notre configuration minimale viole plusieurs restrictions du langage et de GNAT, que nous classons de la façon suivante :

- *Manipulation de types* : `No_Enumeration_Maps`, `No_Fixed_Point`, `No_Floating_Point`,
- *Orienté Objet* : `No_Dispatch`, `No_Finalization`, `No_Nested_Finalization`
- *Manipulation de la mémoire* : `No_Access_Subprograms`, `No_Allocators`, `No_Local_Allocators`, `No_Secondary_Stack`, `No_Standard_Storage_Pools`, `No_Unchecked_Access`, `No_Unchecked_Conversion`, `No_Unchecked_Deallocation`
- *Exceptions* : `No_Exception_Handlers`, `No_Exceptions`
- *Génération implicite de code* : `No_Elaboration_Code`, `No_Implicit_Conditionals`, `No_Implicit_Loops`
- *Règles d'écriture de code* : `No_Direct_Boolean_Operators`, `No_Recursion`
- *Dépendance à la chaîne de production* : `No_Implementation_Attributes`, `No_Implementation_Pragmas`

Cette liste de vingt-trois restrictions fournit une première vue de la qualité de l'implantation réalisée. En particulier, il démontre l'indépendance vis à vis des constructions liées aux entrées/sorties, à la concurrence, et à l'ensemble des constructions du langage interdites par le profil Ravenscar. Ces résultats étaient attendus vu le peu de fonctionnalités présentes dans cette configuration.

Notons cependant que certaines constructions jugées "dangereuses" restent présentes, et sont pleinement justifiées par des impératifs de conception. Ceci exclut donc PolyORB du cadre d'une certification stricte. Cependant, une analyse précise du code démontre que :

- les fonctions de manipulations de types sont présentes pour l'affichage d'information de mise au point (`No_Enumeration_Maps` interdit l'utilisation des attri-

[1] Valeurs mesurées au moyen de SLOCCount de David A. WHEELER [Wheeler, 2004]

buts `'Image` par exemple), ou nécessaires pour l'emballage de certains types (`No_Fixed_Point`, `No_Floating_Point`).
- les gestionnaires d'exception présents le sont à titre de mise au point, et permettent le cas échéant de diagnostiquer tout erreur dans le code ; ou pour gérer les erreurs remontées par la bibliothèque de transport basée sur `GNAT.Sockets`.

En revanche, certaines constructions restent présentes :
- l'orienté objet est la base de l'architecture, il sera donc difficile d'en faire l'économie à moins d'un long travail de conception, qui sort du cadre de nos travaux. Notons cependant que les appels dynamiques sont implantés par l'exécutif de sorte qu'ils s'effectuent à temps constant. Ainsi, ces constructions ne remettent pas en cause le déterminisme de l'application.
- la gestion de la mémoire reste dépendante des mécanismes standards du langage. Notons cependant qu'il est possible dans GNAT de surcharger les mécanismes d'allocation mémoire au niveau global, en proposant une nouvelle implantation du paquetage `System.Memory`. Ceci permettrait de remplacer l'appel à la fonction `malloc` par un allocateur mémoire déterministe, par exemple TLSF [Masmano *et al.*, 2003].

Ainsi, nous montrons que le code produit pour cette configuration répond aux contraintes fortes imposées lors de la construction de systèmes critiques. Le travail de vérification formelle effectué au chapitre précédent et cette étude démontrent ainsi la qualité de l'architecture proposée et de l'implantation des éléments centraux de PolyORB.

6.3 Construction d'un intergiciel RT-CORBA

Dans cette section, nous nous intéressons à la construction d'un nœud basé sur RT-CORBA. Nous fournissons une analyse des modules nécessaires, puis différentes études faites pour analyser l'application construite.

6.3.1 Définition

RT-CORBA [Schmidt & Kuhns, 2000] est une extension des spécifications CORBA, utilisant les mécanismes protocolaires mis en place par GIOP pour échanger des informations de qualité de service entre les différents nœuds (priorités des appels, etc). RT-CORBA utilise les mécanismes de concurrence pour modifier les priorités des tâches de chaque nœud, et associer un pool de tâches à chaque `POA`. Ainsi, RT-CORBA définit des mécanismes permettant la construction d'applications réparties temps réel, fonctionnant suivant le modèle des objets répartis.

La mise en place d'une application RT-CORBA va nécessiter :
- la personnalité GIOP et son instance pour IIOP ;
- un adaptateur d'objets pour le temps réel,
- les personnalités CORBA et RT-CORBA,
- une bibliothèque de concurrence compatible avec le changement dynamique de priorité et l'utilisation de plusieurs tâches,
- un exécutif temps réel.

Nous avons présenté au chapitre précédent des briques intergicielles remplissant certaines de ces fonctions. Ainsi, l'implantation de RT-CORBA nécessite :

1. La sélection des briques intergicielles répondant le mieux aux spécifications ;

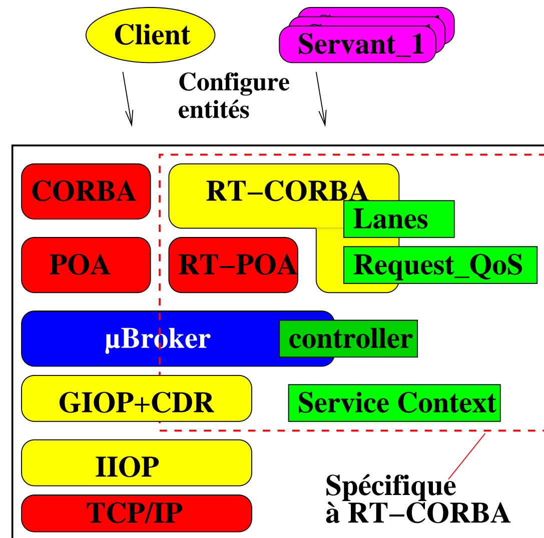


FIG. 6.4 – Configuration d'un nœud PolyORB RT-CORBA

2. La configuration de ces briques ;
3. La construction de la façade correspondant à l'API de RT-CORBA et quelques fonctions utilitaires.

Seule la dernière étape est spécifique à la personnalité à construire, les deux premières étapes peuvent être réalisées dans d'autres contextes. Cette séparation entre cœur fonctionnel et la façade RT-CORBA joue un rôle important dans la forte réutilisation de code permis par l'architecture de PolyORB.

Nous remarquons ainsi qu'aucun élément fonctionnel vital pour le modèle objet répartis temps réel n'est spécifique à RT-CORBA. Seule la façade sera de fait spécifique. Ceci nous permettra de réutiliser ces modules dans d'autres configurations.

Notons que l'utilisation de l'API de RT-CORBA configurera de façon transparente les modules de la personnalité CORBA. La sélection des autres briques se fait par la mise en place du fichier de configuration adéquat.

L'implantation que nous avons réalisée couvre la spécification 1.x de RT-CORBA, dédiée à l'ordonnancement statique. Elle définit notamment `RTCosScheduling`, un ordonnanceur de tâches statiques, les politiques de transmission de priorités `Client_Propagated` et `Server_Declared`, ainsi que les pools de processus légers (*"ThreadPools"*).

Une analyse du code source confirme qu'une majeure partie du code source responsable du comportement déterministe de notre implantation de RT-CORBA se trouve en dehors de cette personnalité. Les mesures effectuées ci-après, montrent que RT-CORBA représente environs 5% de l'instance, essentiellement des interfaces de configuration. Les autres blocs responsables (adaptateurs d'objets, pools de tâches, etc) représentent environ 10% du code, présent à différents niveaux de l'architecture.

Couche neutre	38083 SLOCS
CORBA	11716 SLOCS
RT-CORBA	3365 SLOCS
GIOP	9871 SLOCS

Nous avons testé sur différents exemples présents dans la distribution le fonctionnement correct de ses différentes fonctions, et le respect de la sémantique imposée par RT-CORBA. Ces tests démontrent la faisabilité de l'implantation d'une spécification industrielle complète sur l'architecture que nous proposons.

6.3.2 Dépendances sur l'exécutif

Les spécifications RT-CORBA répondent à des besoins très larges. Par ailleurs, RT-CORBA est une extension des mécanismes de base de CORBA. Dès lors une application basée sur ces spécifications aura à sa disposition de nombreux mécanismes.

En contrepartie, l'exécutable imposera de fortes contraintes sur l'exécutif, celui-ci devra en particulier fournir d'avantage de fonctionnalités. En particulier, l'utilisation faite de la concurrence par RT-CORBA requiert un exécutif supportant des primitives de concurrence évoluées notamment permettant le changement dynamique de priorités, politique de verrouillage limitant les inversions de priorité, etc. Enfin, RT-CORBA requiert aussi un mécanisme de transport orienté connexion FIFO.

RT-CORBA nécessite un exécutif complet, limitant de fait son utilisation dans un contexte contraint. Néanmoins, la forte configurabilité de PolyORB permet d'utiliser un sous-ensemble de ces spécifications dans de tels contextes (par exemple utilisation du profil Ravenscar, protocoles d'invocation de requêtes réduits). Pour ce faire, l'utilisateur doit effectuer une analyse précise des constructions utilisées, et accepter de n'utiliser qu'une partie d'entre elles. Ceci ne peut être effectué qu'après une analyse précise des spécifications RT-CORBA.

Par exemple, sous réserve de ne pas changer dynamiquement la priorité d'une tâche (utilisation de `RTCORBA::Current`) et en configurant soigneusement les objets `ThreadPool`, alors l'application n'effectuera aucun changement dynamique de priorité, répondant ainsi à une des exigences de Ravenscar. Nous fournissons d'avantage d'éléments à la prochaine section.

6.3.3 Critiques de la spécification RT-CORBA

Les spécifications RT-CORBA [OMG, 2002a] définissent un des mécanismes pour la construction d'applications TR^2E. Nous notons cependant que l'utilisation de ces spécifications n'est pas sans risque, et peut dans certains cas être contradictoire avec les guides de "bonne ingénierie" des systèmes temps réels, et des systèmes critiques.

Notons que contrairement à un compilateur Ada 95, il est difficile de vérifier l'emploi correct des constructions d'un intergiciel sans recourir à une analyse précise du code source. Par ailleurs, des différences d'interprétation peuvent amener l'application à agir différemment de ce qui est attendu [Bastide *et al.*, 2000]. De fait, il convient donc d'être prudent quant aux constructions utilisées.

Nous proposons une lecture des spécifications RT-CORBA et montrons quelques cas limites d'utilisation de ces spécifications pouvant entrer en conflit avec mes règles d'ingénierie pour les systèmes critiques.

L'analyse que nous faisons est loin d'être exhaustive et vise avant tout à montrer quelques unes des limites inhérentes à l'utilisation de standards pour la construction d'applications, et l'attention particulière qui doit être portée à leur utilisation.

Concurrence et comportement dynamique

– Il est possible de changer dynamiquement les priorités des tâches utilisateur grâce à l'interface `RTCORBA.Current`. Ceci va à l'encontre notamment des règles du profil Ravenscar. Ce changement de priorité a un impact sur l'ordonnancement de l'application qui peut être difficile à analyser.
– Des tâches peuvent être créées dynamiquement lors de la construction d'instances de `ThreadpoolLane`. Ceci est aussi interdit par le profil Ravenscar.

Ces deux points peuvent être levés en s'interdisant les changements dynamiques de priorité, et en préallouant les tâches. Ces deux actions sont permises par PolyORB et l'utilisation du profil Ravenscar.

Compatibilité entre politiques

– Il n'y a pas de tests de compatibilité entre les politiques `THREADPOOL_POLICY` et `PRIORITY_MODEL_POLICY`, ceci conduit aux problèmes suivants :
  – On peut spécifier qu'un servant exécutera toutes les requêtes à une priorité P_1 grâce au choix de la politique `SERVER_DECLARED` de `PRIORITY_MODEL_POLICY`, mais l'affecter à un pool de tâches à la priorité P_2.
  – On peut définir un pool de tâches à une priorité P_1, et utiliser une invocation via un appel sous la politique `CLIENT_PROPAGATED` à la priorité P_2.

  Dans ces deux cas, l'invocation s'effectuera en fait à une priorité définie par défaut par le pool, et non à le modèle de priorité désiré. Ainsi, la sémantique de l'application n'est pas préservée par une erreur de configuration non détectée.
– Il n'y a pas de tests de compatibilité entre la politique `THREADPOOL_POLICY` et les appels à la méthode `RTPortableServer : :POA : :activate_object_with_id_and_priority`. Un servant peut être enregistré à une priorité qui n'est pas couverte par le pool de tâches. Il est alors rattaché à un groupe de tâches à une priorité par défaut. Ce comportement "par défaut" compromet la sémantique de l'application.
– L'utilisation effective des tâches d'un pool pour scruter les sources n'est pas spécifiée. On peut ainsi occuper tous les tâches d'un pool à lire des données ou on contraire n'avoir qu'un seul tâche dédié à la lecture. Ce paramètre est dépendant de l'implantation de l'ORB et peut ne pas être configurable.
– L'extension RT-CORBA Dynamic Scheduling [OMG, 2003b] réalise l'ordonnancement de la requête une fois la requête lue et remontée jusqu'à un point d'interception. Rien n'est prévu pour ordonnancer les tâches dédiées à la lecture des données. La lecture de requêtes peut ainsi perturber l'exécution de requêtes à haute priorité.

Compatibilité avec d'autres spécifications de l'OMG

– RT-CORBA, Dynamic Scheduling utilise des constructions interdites par Minimum CORBA, parmi lesquelles l'interface `PortableInterceptors`. Ceci compromet l'utilisation de ces spécifications pour la construction de systèmes embarqués. Il y a ici un fort conflit entre deux spécifications qui auraient pu être complémentaires.

Spécifications incomplètes

– La sémantique de la méthode `RTCORBA : :RTORB : :Destroy_Threadpool` est incomplètement spécifiée. L'utilisateur peut détruire un pool de tâches asso-

cié à un POA qui est en train de traiter des requêtes. Il peut ainsi perturber le comportement de l'application, par exemple en arrêtant certains traitements.

– L'utilisateur peut créer des pools de tâches de 0 tâche statique et dynamique ou avec une pile de 0 octet, donc un pool inutilisable. Ceci tient au typage faible utilisé pour définir les interfaces de RT-CORBA, qui ne dispose pas d'un type `Positive` comme en Ada 95. De plus, aucun test à l'exécution n'est prévu pour détecter ces anomalies, ni pour assurer la compatibilité entre une taille de pile nécessaire aux besoins de l'intergiciel et la taille de pile spécifiée pour ce pool.
– Lors de la création d'un tampon pour stocker les requêtes, il n'y a aucun lien entre la taille du tampon et la structure interne utilisée par l'ORB pour stocker les requêtes. Cette valeur a donc un rôle purement indicatif, difficile à évaluer.
– Les spécifications RT-CORBA ne définissent pas les éléments de configuration de l'exécutif ayant un impact sur le comportement des nœuds, telle que la politique d'ordonnancement des processus légers. Ceci crée un problème lors du port de l'application sur d'autres plates-formes.

Comportements dangereux

– Les informations de QdS sont stockées dans le contexte de la requête, qui est un dictionnaire. Ceci pose un problème d'efficacité dans la recherche de ses paramètres non trivial : allocation mémoire, recherche de données, etc.
– On peut perturber l'ordonnancement de façon potentiellement incompatible en utilisant simultanément des objets de type `PriorityTransforms` et `PRIORITY_MODEL_POLICY`. En particulier, le client peut voir son exécution passer à une autre priorité malgré la mise en place de la politique `CLIENT_PROPAGATED`. En outre, le client ignore si un objet de type `PriorityTransform` a été déployé. Ce faisant, on masque une information importante sur la sémantique de l'application.
– Des inversions de priorité sont possibles entre deux tâches à des priorité RT-CORBA différentes, mais à des priorités natives identiques. La solution proposée est de faire l'ordonnancement plus précis au niveau d'un des objets `PortableInterceptors` utilisés par RT-CORBA Dynamic Scheduling, ou d'éviter les collisions entre priorités natives et priorités RT-CORBA.
– Quel est le statut d'une tâche traitant une requête qui doit, dans ce contexte, exécuter une nouvelle requête distante ? est-elle bloquée ? ou est-elle libre pour traiter une nouvelle requête ?
  Le premier cas peut conduire à un épuisement des tâches disponibles et donc un blocage du nœud. Le second cas peut conduire à un délai aléatoire ajouté au traitement de la première requête.

Ces éléments, loin d'être exhaustifs, fournissent une vue des problèmes possibles lors de l'utilisation de RT-CORBA. De fait, ils montrent une des limites fortes liées à l'utilisation d'un intergiciel : la nécessaire compréhension de ses mécanismes internes pour éviter tout comportement non désiré pouvant compromettre le bon fonctionnement de l'application.

Compte tenu de la forte modularité de l'architecture proposée, il est possible de raffiner les personnalités existantes pour implanter un profil restreint de RT-CORBA plus à même de répondre aux contraintes des systèmes critiques. Ce profil répondrait alors plus précisément aux contraintes de l'application.

Dans ce contexte, l'OMG définit à l'heure actuelle CORBA High Assurance [Haverkamp & Richards, 2002], il s'agit d'un profil restreint de CORBA et RT-CORBA qui vise la construction de systèmes répartis temps réel embarqués fiables, par exemple pour le domaine avio-

nique. Notre analyse fournit un ensemble de résultats permettant la définition de ce profil. Nous discutons ce point en conclusion de ce mémoire.

6.3.4 Conclusion

Dans cette section, nous avons détaillé la construction d'une personnalité RT-CORBA pour PolyORB. Nous avons montré comment les différentes briques d'intergiciel proposées au chapitre 5 peuvent être combinées pour proposer une implantation compatible avec ces spécifications.

Notons que la construction de cet intergiciel a été relativement rapide. Nous avons pu compléter l'implantation en quelques semaines. Ceci montre que les abstractions retenues sont naturelles et bien adaptées au problème à résoudre.

Le processus de conception a consisté en l'identification des paramètres de configuration pertinents pour RT-CORBA, et l'enrichissement de certaines fonctions clés de l'intergiciel pour les supporter. Ainsi, GIOP a été enrichi pour supporter le transport des informations de QdS, l'adaptateur d'objets a évolué pour supporter les `lanes`, files de tâches, imposées par les spécifications.

Enfin, nous avons listé plusieurs cas d'utilisation limites de ces spécifications, afin d'illustrer la difficulté de construire une application TR^2E autour de RT-CORBA. Loin d'être une critique de ces spécifications, nous tirons de cette analyse plusieurs enseignements qui permettront de mieux comprendre la sémantique des modules construits, et leurs limites d'utilisation en particulier pour l'ordonnancement de l'application. Ces travaux trouvent écho aux travaux de normalisation du profil CORBA High Assurance auxquels nous comptons participer.

Nous détaillons les mesures de performances effectuées pour différentes applications construites autour de notre implantation de RT-CORBA au prochain chapitre.

6.4 Prototypes

La section précédente a montré comment l'architecture et les briques intergicielles que nous proposons pouvaient être adaptées pour supporter une spécification complète.

Dans cette section, nous nous intéressons à la mise en place d'autres personnalités dédiées au temps réel. Nous fournissons les grandes lignes de l'implantation à réaliser. Nous montrons comment construire à faible coût de nouvelles personnalités dédiées aux applications réparties temps réel en réutilisant les modules déjà définis.

6.4.1 RT-DSA

L'annexe des systèmes répartis de Ada 95 (Ada/DSA) définit les mécanismes permettant un support transparent de la répartition pour les applications temps réels.

GLADE [Pautet & Tardieu, 2000], l'implantation normalisée de l'annexe, ajoute la notion de priorité aux messages, définit un mécanisme d'échange de priorité basé sur les politiques `Client_Propagated` et `Server_Declared` qui sont proches des mécanismes proposés par RT-CORBA. Néanmoins, contrairement à RT-CORBA, ces politiques sont définies au niveau d'une partition, et non au niveau des `POA`.

Dans [Campos *et al.*, 2004], les auteurs fournissent plusieurs extensions pour rendre l'annexe et son implantation GLADE temps réel, et construire ainsi RT-GLADE. Pour ce faire, ils proposent de réorganiser le code source d'une ancienne version de GLADE

(3.14p) pour supprimer l'allocation dynamique de tâches, permettent d'affecter la priorité des différentes tâches impliquées dans le traitement d'une requête. Enfin, ils utilisent RT-EP, un protocole dédié pour le transport de données temps réel sur ethernet. Notons que RT-GLADE n'est disponible que sur la plate-forme MaRTE OS.

Ces options sont mises en œuvre en utilisant des options de compilation et des fichiers de configurations spécifiques, basés sur le langage de configuration de `GNATDIST`.

Ces deux extensions sont des évolutions des mécanismes originaux et de ceux proposés par GLADE. Nous remarquons qu'il s'agit uniquement des modifications des mécanismes internes de l'intergiciel et de ses mécanismes de configuration.

La version initiale de PolyORB dispose d'une implantation de Ada/DSA au dessus de PolyORB, décrite dans [Quinot, 2003, chap 7.3]. Cette implantation utilise les mécanismes de PolyORB pour la construction et la gestion des différentes entités à répartir. Nous en présentons ici une vue rapide.

Ada/DSA définit les types d'entités pouvant être rendus visibles au travers de l'intergiciel : sous-programmes distants, pointeur sur objet distant et pointeur sur sous-programmes distants. La personnalité Ada/DSA de PolyORB associe à chacune de ces entités une projection particulière qui sera réalisée par le compilateur lors de la phase d'expansion de code. Cette projection définit les entités intermédiaires responsables de la mise en œuvre de la sémantique de ces constructions.

Nous remarquons que ces différentes projections ont en commun d'associer à chaque entité pouvant être répartie un servant enregistré auprès d'un `Portable Object Adapter`, fourni par la couche neutre de PolyORB. Le servant aura pour rôle d'aiguiller la requête vers la procédure correspondante. Ce faisant, ce modèle ramène les différentes constructions de Ada/DSA à l'exécution d'une certaine méthode sur un servant.

Nous constatons que les différents mécanismes proposés pour GLADE peuvent être mis en œuvre à moindre coût par la personnalité Ada/DSA de PolyORB en réutilisant certains des mécanismes déjà mis en œuvre pour RT-CORBA (figure 6.5).

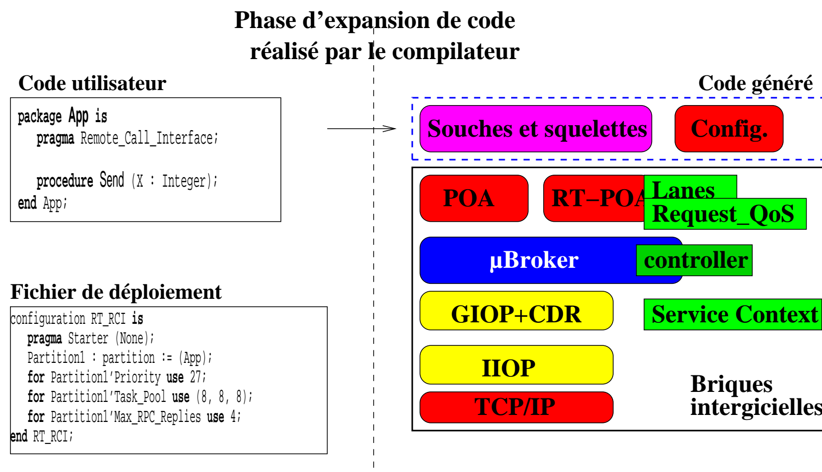


FIG. 6.5 – Configuration d'un nœud PolyORB RT-DSA

Ainsi, les politiques du μBroker peuvent gérer un pool de tâches statiques, les mécanismes de propagation de priorités sont mis en place au niveau du `RT-POA` de la couche neutre et de la personnalité protocolaire GIOP, un mécanisme de transport temps réel est lui aussi disponible et utilisable avec GIOP.

Une adaptation de la phase d'expansion de code permettrait de modifier le code généré pour configurer ces différents éléments et ainsi intégrer les notions de priorités aux requêtes et objets applicatifs et utiliser des structures internes déterministes.

Il faudrait ainsi modifier le code généré pour :

1. modifier le code généré pour qu'il utilise des structures déterministes pour le démultiplexage des requêtes ;
2. modifier la configuration des nœuds pour inclure les briques intergicielles gérant la qualité de service aux niveaux applicatifs et protocolaires et permettant la construction de références d'objets portant ces informations ;
3. utiliser des `RT-POA` et configurer ces derniers avec des politiques compatibles avec les consignes de compilation associées aux entités réparties.

Nous montrons ainsi qu'il est possible, à une reconfiguration près de l'intergiciel et du générateur de code utilisé, d'adapter la personnalité Ada/DSA existant pour construire une personnalité *"RT-DSA"*.

En particulier, cette démarche illustre comment les mécanismes mis à disposition par la couche neutre peuvent être réutilisés pour construire de nouvelles personnalités, ou enrichir les personnalités existantes.

6.4.2 DDS

Nous avons déjà présenté DDS au chapitre 2. DDS définit un modèle générique (PIM, *"Platform Independent Model"*) d'un intergiciel orienté messages, dont les politiques de configuration et déploiement sont adaptées aux applications temps réel.

DDS est formé de deux couches : DCPS (*"Data Centric Publish/Subscribe"*) qui se charge du transfert efficace des données entre nœuds, DLRL (*"Data Local Reconstruction Layer"*) qui se charge de l'envoi et du filtrage des données à l'échelle du nœud et de l'application. Les données sont propagées par le biais de canaux de communication fédérateurs (*"Topic"*).

Ainsi définies, les spécifications DDS peuvent être utilisées pour construire une personnalité applicative pour PolyORB. Cette personnalité prend en charge la gestion des Topics, le contrôle des paramètres de QdS associées aux données, des objets chargés de transférer les données, la découverte des nœuds participant à l'application (`DomainParticipant`). Enfin, elle réagit aux changements des données d'un des participants, et propage les nouvelles valeurs aux autres participants.

Nous avons implanté MOMA, une personnalité supportant le passage de message de PolyORB que nous avons présenté à la section 3.2.2. Cette personnalité supporte les mécanismes de publication/souscription qui forment le corps de DDS.

Une étude préliminaire des spécifications DDS a montré que les objets applicatifs peuvent être construits en utilisant les mécanismes de MOMA. L'extension de l'API de MOMA et une refonte du mécanisme de configuration des objets formant le fournisseur MOMA permet de supporter la sémantique de DDS (figure 6.6).

Cette analyse démontre l'adaptabilité de l'architecture que nous avons définie. Les différentes entités définies par DDS ont pu être implantées au dessus des entités fournies par MOMA. Les abstractions fournies par PolyORB permettent le support de cette spécification en se concentrant uniquement sur les spécificités de DDS, et non sur la réalisation de fonctions utilitaires déjà fournies par PolyORB.

Une phase de conception et d'intégration a été menée lors d'un stage de Master Recherche que nous avons co-encadré [Can, 2005]. Elle a fourni un premier prototype

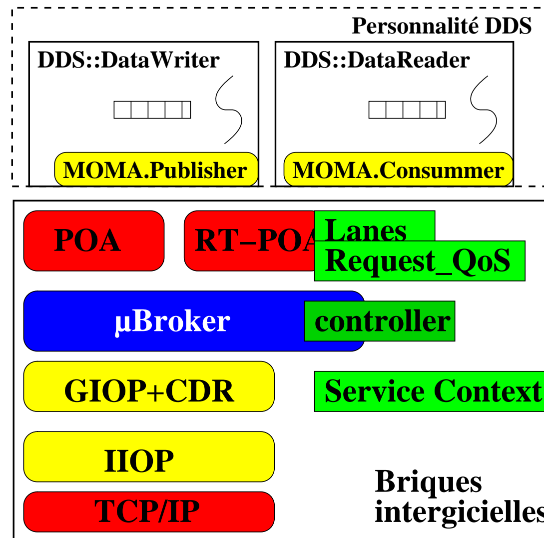


FIG. 6.6 – Configuration d'un nœud PolyORB DDS

concluant d'une personnalité DDS, couvrant l'ensemble des politiques de QdS définies par la norme à l'exception des politiques dédiées à la tolérance aux pannes.

Ce prototype réutilise les constructions de MOMA et de la couche neutre, dont les fonctions d'intergiciels mises en œuvre pour RT-CORBA. Il fournit une API conforme aux spécifications de la partie "DCPS" *Data Centric Publish and Subscribe*.

Nous notons sur cet exemple que de nombreux éléments de ce prototype sont issus de la bibliothèque de briques intergicielles que nous avons définie précédement. Ainsi, le taux de réutilisation de code atteint près de 90% ; le développement a pu se concentrer uniquement sur les éléments significatifs des spécifications DDS et accélérer d'autant le développement de cette personnalité.

Ce prototype démontre l'apport de l'architecture schizophrène que nous avons raffinée et les multiples services que nous lui avons adjoints pour la construction rapide de nouveaux intergiciels.

De prototype, cette personnalité devrait bientôt rejoindre les autres personnalités déjà existantes au sein du projet PolyORB.

6.5 Conclusion

Ce chapitre a présenté plusieurs expérimentations réalisées autour de l'architecture que nous avons définie. Ces expérimentations ont visé la construction d'intergiciels adaptés aux systèmes temps réel répartis embarqués en utilisant les modules d'intergiciels que nous avons proposés.

Nous avons dans un premier temps détaillé et classé les différents modules à notre disposition. La sélection et l'initialisation de ces modules ont été détaillées. Dans un premier temps, les modules à utiliser sont sélectionnés en se fondant sur le mécanisme de dépendances entre paquetages de Ada 95. Par transitivité, d'autres modules seront sélectionnés. Cette ensemble permet d'assurer que le nœud pourra être compilé. Un mécanisme d'initialisation contrôle les dépendances entre les différents modules, et les

initialise.

Nous avons ensuite montré comment construire plusieurs intergiciels dédiés aux systèmes TR^2E, en discutant la construction d'intergiciels supportant plusieurs niveaux de fonctionnalités.

Un intergiciel minimal a été proposé. Son analyse nous a permis de mettre en évidence le nombre limité de modules nécessaire à son fonctionnement. Il illustre ainsi la forte configurabilité de la solution. Par ailleurs, l'utilisation des outils de la chaîne de production nous a fourni plusieurs informations sur la qualité du code écrit, et les restrictions du langage qui sont violées.

Nous avons ensuite détaillé la construction d'un intergiciel basé sur RT-CORBA. RT-CORBA réutilise plusieurs éléments de CORBA dont il étend certains mécanismes. Par extension, notre implantation de RT-CORBA réutilise de nombreux modules de la couche neutre de PolyORB mais aussi de la personnalité CORBA. Ainsi, nous notons que la réalisation de cet intergiciel a été grandement facilitée par l'architecture utilisée qui permet une forte réutilisation de modules.

La rapidité d'implantation de RT-CORBA nous amène à considérer l'implantation d'autres intergiciels pour systèmes TR^2E. Nous avons, à titre d'illustration, tracé les grandes lignes de la conception de personnalités RT-DSA et DDS, et montré comment l'architecture existante permet de construire de tels intergiciels rapidement.

Ainsi, ces différents éléments nous permettent de fournir une utilisation nouvelle d'un intergiciel : plutôt que fournir un framework bas niveau de communication ou un intergiciel figé et dédié à un modèle, nous proposons un ensemble de mécanismes et les moyens de les adapter permettant la construction d'une famille d'intergiciels.

Ainsi, nous fournissons une solution permettant de prendre en compte la forte hétérogénéité des besoins des systèmes TR^2E. Comme nous l'indiquons dans [Hugues *et al.*, 2005a], notre approche permet de fournir une infrastructure pour la répartition utilisable comme un "COTS" mais facilement adaptable à de nouveaux besoins.

7

Mesures

Sommaire

7.1 Bancs de mesure **144**
7.1.1 ORK 144
7.1.2 MaRTE OS 144
7.1.3 Sun Solaris 144
7.2 Analyse du comportement temporel **145**
7.2.1 Définition d'un protocole de mesure 145
7.2.2 Réalisation de la plate-forme de mesure 148
7.2.3 Mesures sur Solaris 148
7.2.4 Mesures sur un noyau temps réel 152
7.2.5 Analyse critique 153
7.3 Empreinte mémoire **154**
7.3.1 Mesures 154
7.3.2 Optimisations 155
7.3.3 Conclusion 157
7.4 Conclusion **157**

Ce chapitre explore différents aspects de l'implantation réalisée. Nous nous intéressons aux performances que notre solution peut fournir. Nous avons proposé un intergiciel pour la construction d'applications temps réel embarquées, deux paramètres vont rentrer en ligne de compte pour son évaluation :

1. *le comportement temporel* du système construit, en particulier son comportement lorsque la charge varie, le nombre de clients varie, etc
2. *l'empreinte mémoire*, en particulier en fonction des fonctionnalités embarquées.

Un point important à considérer est qu'aucune mesure ne saurait être absolue. Les performances mesurées dépendent de nombreux paramètres : configuration des différents nœuds, topologie de l'application. Un autre facteur concerne l'architecture de l'intergiciel en lui même, et le nombre de points de configuration et d'adaptation qu'il offre. Il y a un compromis entre performances d'une architecture, et le degré d'adaptation qu'elle fournit, comme le montre une comparaison entre implantations de CORBA [Buble *et al.*, 2003].

7.1 Bancs de mesure

Dans cette section, nous définissons les différentes plates-formes utilisées.

PolyORB tire profit des propriétés de portabilité du langage Ada 95. Ainsi, PolyORB est disponible sur une large palette de plates-formes pour lesquelles un compilateur Ada 95 et une interface de communication sont disponibles. Le portage sur de multiples cibles est donc grandement facilité, l'effort requis est limité voire nul. Il se réduit bien souvent à adapter certains paquetages d'entrées/sorties aux spécificités de la plate-forme.

Ceci nous permet de comparer facilement les performances de PolyORB sur plusieurs noyaux et systèmes d'exploitation disposant d'un support des primitives de communication et de concurrence de niveau différent. Nous avons sélectionné ORK, MaRTE OS et Solaris. Il s'agit de confronter ces mesures effectuées et fournir une vue du comportement de notre solution pour des configurations hétérogènes.

7.1.1 ORK

ORK [de la Puente *et al.*, 2000] est un noyau temps réel qui supporte le profil Ravenscar défini par le langage Ada 95. Ce noyau, écrit en Ada 95, a une empreinte mémoire et une complexité réduites. Il est conçu pour être utilisé dans des applications temps réel critiques, éventuellement certifiées. ORK est diffusé sous la forme d'un logiciel libre [DIT/UPM, 2004].

ORK est intégré aux outils de la suite d'outils de GNAT sous la forme d'un système d'exécution Ada 95 utilisable par l'application de façon transparente. ORK est disponible pour les cibles ERC-32 (une version renforcée du processeur SPARC-V7, supportant les radiations électromagnétiques) et pour architecture x86.

La version utilisée pour nos mesures est la version $2.1w$, basée sur le compilateur GNAT Pro 5.02a. Les mesures effectuées pour ce noyau utilisent quatre PCs disposant chacun d'un processeur Pentium-II cadencé à 350Mhz, 64Mo de mémoire, et une carte réseau Intel EEPro 100.

7.1.2 MaRTE OS

MaRTE OS [Rivas & Harbour, 2001] est un noyau temps réel qui suit les spécifications Minimal Real-Time POSIX.13. Il fournit une implantation des différentes primitives POSIX classiques, ainsi que différentes extensions pour le support des horloges temps réel, un allocateur mémoire déterministe et le protocole ethernet temps réel RT-EP. MaRTE OS fournit un ensemble d'outils utilisant GNAT et un système d'exécution spécifique pour les architectures x86.

Nous avons utilisé la version 1.56 de MaRTE OS, intégrée au compilateur GNAT *"GNAT Academic Program"* 1.0.0, et la même plate-forme matérielle que pour ORK.

7.1.3 Sun Solaris

Sun Solaris est un système Unix classique. Nous avons sélectionné Solaris car il fournit un bon compromis entre une plate-forme de développement générique, et une plate-forme temps réel. En effet, Solaris est connu pour avoir une pile de protocole ayant une latence limitée et de bonnes propriétés temporelles.

Nous avons utilisé plusieurs stations Ultra-5, sous Solaris 9, disposant de 128Mo de RAM et d'un processeur Ultra-Sparc IIi à 333Mhz. Cette plate-forme est suffisamment

contrainte pour montrer le poids de l'intergiciel sur des systèmes disposant de faibles ressources, tout en offrant un support aisé pour la mise au point. Les différents tests ont été compilés en utilisant le compilateur GNAT Pro 5.03a.

7.2 Analyse du comportement temporel

Nous nous intéressons au comportement temporel de notre solution. Deux paramètres sont à prendre en compte : *1)* la valeur moyenne de certaines actions, *2)* la variance sur les mesures. Ces deux paramètres nous permettront d'évaluer le déterminisme de différentes configurations.

Dans un premier temps, nous définissons un protocole de mesure, basé sur le protocole *"Distributed Hardstone benchmark"*, nous présentons ensuite la configuration testée, basée sur la personnalité RT-CORBA. Enfin, nous détaillons différentes mesures effectués sur notre banc de mesure.

7.2.1 Définition d'un protocole de mesure

[Kameno & Weiderman, 1991] et [Mercer *et al.*, 1990] proposent tout deux une extension du protocole de mesure Harstone, défini initialement pour les systèmes monoprocesseurs [Weiderman, 1989] : le *"Distributed Hardstone benchmark"*

Il s'agit d'un jeu de tests de type "boîte noire", pour lesquels aucune connaissance a priori sur le système n'est demandée. L'intergiciel est utilisé comme une boîte noire. On s'intéresse ici exclusivement à son comportement du point de vue de l'application.

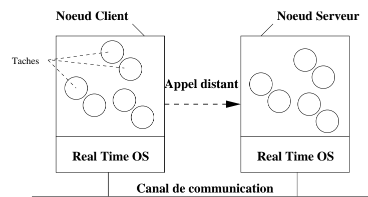


FIG. 7.1 – Distributed Hardstone Benchmark original

Ces différents protocoles font l'hypothèse que des tâches communiquent par un réseau (figure 7.1). Ils mesurent pour plusieurs ensembles de tâches communiquant la limite à partir de laquelle le système ne peut plus traiter de travaux de façon correcte, et commence à manquer des échéances temporelles.

Ces protocoles de tests font des hypothèses quant au modèle de répartition utilisé (modèle tâche communiquante), et quelques hypothèses plus générales sur la couche de communication utilisée (FIFO sans perte). Nous avons adapté ces protocoles à notre intergiciel. Nous présentons successivement chacun des tests réalisés, ainsi que le résultat mesuré.

La figure 7.2 présente l'architecture que nous testons : *S* entités serveurs et *C* entités clientes interagissent (1), PolyORB (2) et un système d'exploitation temps réel (3) servent de support d'exécution à ce système.

La configuration de l'intergiciel PolyORB varie suivant les différentes phases de test. Nous la présentons par la suite. Par rapport au banc de test initial, l'ajout d'une brique intergicielle offre d'avantage de souplesse en termes de configuration et de déploiement. En particulier, il est possible de choisir plus finement la priorité à laquelle

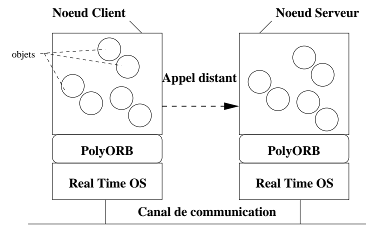


FIG. 7.2 – Distributed Hardstone Benchmark avec intergiciel

chacune des entités sera exécutée, de façon indépendante du code exécuté (modèle objet réparti). En contrepartie, il faut prendre en compte dans le protocole de test le nécessaire aiguillage des requêtes réalisé par l'intergiciel. Par la suite, tous les tests utiliseront le modèle proposé par RT-CORBA.

Pour chaque test, nous utilisons une fonction de charge issue de la suite de tests `Whetstone` [Curnow & Wichmann, 1976]. Cette fonction simule un temps de calcul par le processeur, dont la durée peut être paramétrée.

Performances brutes

Le premier test s'intéresse au coût minimal du traitement d'une requête par l'intergiciel. Ce temps mesure le temps nécessaire à l'envoi d'une requête à travers l'intergiciel. Ce temps varie en fonction de la charge de chaque nœud, mais aussi en fonction des politiques de synchronisation et de gestion de la concurrence utilisées. De façon à mesurer l'impact de l'intergiciel, nous nous basons sur une configuration sans tâche du nœud serveur, de façon à supprimer l'usage de la concurrence. Ceci nous garantit que l'on mesurera seulement le temps passé dans les fonctions de l'intergiciel, noté T_{MW}.

Quatre paramètres sont intéressants à mesurer : la valeur minimale et maximale mesurée ; la valeur moyenne et la dispersion des mesures autour de cette valeur, mesurée par son écart-type. Une analyse des ces estimateurs statistiques nous renseignera sur la dispersion des mesures, et donc leur déviation par rapport à un comportement déterministe reproductible.

Algorithm 3 Test de performances : Pseudo-code du client

1: **for** $j = 1$ to N **do**
2: $begin_j \leftarrow CLOCK$
3: `<request> (payload)`
4: $end_j \leftarrow CLOCK$
5: **end for**
6: $Err_T1 \leftarrow CLOCK$
7: $Err_T2 \leftarrow CLOCK$
8: $\varepsilon \leftarrow (Err_T2 - Err_T1)$ {Erreur sur la mesure}

Le pseudo-code 3 présente le code exécuté par le client : l'envoi d'une requête au serveur. Nous distinguons deux types de requêtes :

1. `round_trip` envoie une donnée (payload), représentée par un entier codé sur 32bits. Le serveur reçoit ses données et retourne immédiatement, sans traitement.

2. `ping` envoie une donnée (payload), représentée par un entier codé sur 32bits. La requête est envoyée par la couche de transport et le client retourne immédiatement.

Les vecteurs $(begin_j)_{j\in 1..N}$ et $(end_j)_{j\in 1..N}$ mesurent les instants marquant respectivement le début et la fin de l'envoi d'une requête. Ils serviront au calcul des estimateurs. Par ailleurs, nous évaluons l'erreur commise sur la mesure de ses vecteurs, en évaluant le temps nécessaire à deux appels consécutifs à l'horloge.

Nous notons que la valeur mesurée pour l'envoi des données est fonction des données à transporter et du protocole utilisé. Nous pouvons ainsi distinguer les paramètres influant sur cette mesure en tenant compte du cheminement de la requête à travers l'intergiciel présenté au chapitre 3 :

1. la clé identifiant l'objet destinataire, qui dépend de la manière dont l'objet a été enregistré auprès de l'adaptateur d'objets de l'intergiciel ;
2. les paramètres associés à la méthode (nombre et taille) ;
3. la configuration du nœud : nombre d'objets, configuration utilisée, etc.

Il nous faut donc réaliser différents tests, mesurant l'impact de chacun de ses paramètres sur le temps de traitement d'une requête.

Nous avons déjà montré par ailleurs que l'utilisation des tables de hachages dynamiques parfaites rendait la recherche d'un objet déterministe, à temps constant. Ce paramètre n'est donc pas significatif. Ainsi, nous ne retenons comme paramètres pour nos tests que la taille des paramètres et la configuration du nœud.

Bande passante disponible

La bande passante représente le débit d'informations que propose l'intergiciel sur les canaux de communication à sa disposition. On peut le qualifier suivant deux modes :

1. *requêtes/s* : nombre de requêtes maximal que l'intergiciel pourra traiter.

   Cette donnée se déduit de la mesure de performance de l'intergiciel. L'intergiciel pourra traiter d'autant plus de requêtes que le temps de traitement de chacune sera faible. Par conséquent, on retient comme estimateur du nombre maximal de requêtes le nombre de requêtes émises pour le cas le plus favorable, à savoir celui ayant le minimum de données spécifiques de l'application.

   La détermination de ce cas est discuté par la suite, lors de la mise en place du banc de mesure.

2. *bit/s* : mesure la taille maximale des requêtes que l'intergiciel pourra traiter en 1s. Elle représente la charge utile disponible pour l'application. Elle est fonction de la couche de communication utilisée, ainsi que du protocole et du format de représentation des données choisis.

   La mesure de performance précédente fournit une mesure du temps de latence pour transférer une requête minimale. Cette nouvelle mesure vise à évaluer l'impact qu'a la taille des données à transmettre (arguments variables) sur le temps de traitement d'une requête.

   Ainsi, si on note RTT la latence mesurée lors du transfert d'une requête 'vide', et en faisant l'hypothèse simplificatrice que les nœuds sont homogènes, alors on

déduit la bande passante, notée MW_{bw}, du temps de traitement d'une requête par l'intergiciel (T_{MW}) comme suit

$$T_{MW} = RTT + 2\frac{sizeof(Data)}{MW_{bw}} \Rightarrow MW_{bw} = \frac{2 * sizeof(Data)}{T_{MW} - RTT} \quad (7.1)$$

Algorithm 4 Test de bande passante : Pseudo-code du client

1: **for** $j = 1$ to N **do**
2: $begin_j \leftarrow CLOCK$
3: result ← `round_trip (payload <1 .. N>)`
4: $end_j \leftarrow CLOCK$
5: **end for**
6: $Err_T1 \leftarrow CLOCK$
7: $Err_T2 \leftarrow CLOCK$
8: $\varepsilon \leftarrow (Err_T2 - Err_T1)$ {Erreur sur la mesure}

7.2.2 Réalisation de la plate-forme de mesure

La spécification IDL (extrait de code 9) représente l'interface des différents tests mis en œuvre. Nous avons opté pour la personnalité CORBA, en conjonction avec les interfaces de RT-CORBA. Ceci fournit une interface simple d'emploi pour la mise au point des différents tests que nous avons définis.

Nous avons utilisé la version de développement courante de PolyORB, compilée avec les options de compilation par défaut, sans les options de mise au point. Chaque nœud utilise le profil de concurrence complet, un pool de tâches, la politique `Half_Sync_Half_Async` du μBroker, ainsi que les personnalités RT-CORBA et GIOP/IIOP. Il s'agit d'un choix arbitraire visant à étudier une configuration à même d'être déployée par l'utilisateur.

Les différentes mesures de temps ont été effectuées en utilisant les primitives du paquetage `Ada.Real_Time`. L'horloge fournie est de type monotone, dont les propriétés sont clairement définies par la norme du langage, réduisant ainsi les erreurs d'interprétation. Un test préliminaire a vérifié la précision de la fonction `Clock` retournant la valeur courante de l'horloge.

7.2.3 Mesures sur Solaris

Nous avons effectué deux campagnes de mesures sur Solaris.

Dans un premier temps, nous avons mesuré la latence d'un RPC. Nous avons ensuite analysé l'impact de la configuration sur le déterminisme de l'application.

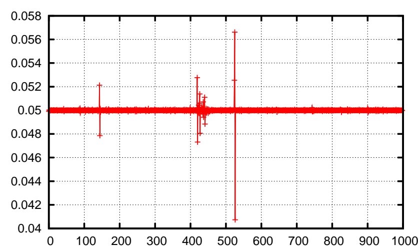
Latence d'un RPC

Nous avons mesuré le temps d'exécution d'un RPC dans le cas d'une requête avec valeur de retour (figure 7.3), puis dans le cas d'une requête ONEWAY (figure 7.6). Ces deux nœuds utilisent les méthodes `Round_Trip` et `Ping` que nous avons définies précédemment.

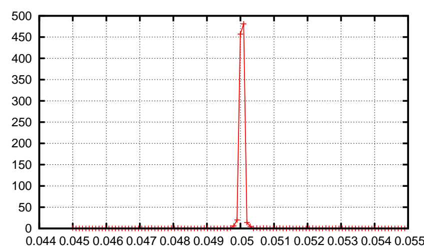
Dans cette configuration, nous utilisons une connexion TCP/IP pour relier les deux nœuds. Nous mesurons le temps complet pour effectuer un cycle appel/réponse.

Extrait de code 9 Spécification IDL de Distributed Hardstone Benchmark

```
import ::RTCORBA;

module DHB {
  // This module is derived from the Distributed Hartstone
  // benchmarks. It is used to bench ORB internals.

  typedef unsigned long KWIPS;
  // Represent the number of Kilo Whetstone Instruction Per Second

  interface Worker {
    // A Worker is a remote entity that performs some work on behalf
    // of a client entity. This interface defines functions that serve
    // to measure some metrics of the underlying ORB.

    typedef sequence<unsigned long> U_sequence;

    void Do_Some_Work (in KWIPS Kilo_Whetstone);
    // Worker performs Kilo_Whestone operations

    void Do_Some_Work_With_Payload (in KWIPS Kilo_Whetstone,
                                    in U_sequence Payload);
    // Worker performs Kilo_Whestone operations + transmit some Payload

    KWIPS Get_KWIPS ();
    // Return the number of Kilo Whetstone Instruction Per Seconds
    // that the worker can provide

    RTCORBA::Priority Running_Priority ();
    // Return the running priority of the servant

    unsigned long Round_Trip (in unsigned long data);
    // Round trip with data as payload, simply return data

    U_sequence Round_Trip_With_Payload (in U_sequence Payload);
    // Round trip with some more payload, simply return Payload

    oneway void Ping (in unsigned long data);
    // Ping the remote node
  };

  interface Background_Worker {
    // A Background_Worker is a remote entity that performs some work
    // in background.

    KWIPS Get_KWIPS ();
    // Return the number of Kilo Whetstone Instruction Per Seconds
    // that the worker can provide

    oneway void Do_Background_Work (in KWIPS Kilo_Whetstone,
                                    in RTCORBA::Priority Priority);

    boolean Is_Working ();
    // Return true iff Background_Worker is acutally performing some work
  };
};
```



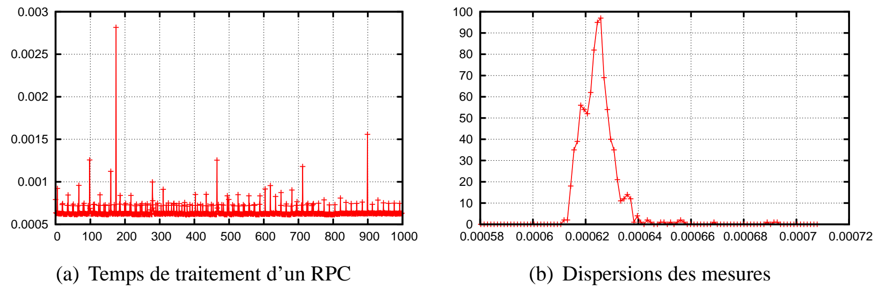
(a) Temps de traitement d'un RPC



(b) Dispersions des mesures

FIG. 7.3 – Mesures sur la plate-forme Solaris, cas d'un appel/réponse. (valeurs en *s*)

Nous notons une valeur moyenne de $50ms$, et une dispersion faible : l'écart-type est de $4.14758E-04$, les valeurs sont contenues dans une fenêtre de $15.88ms$ de large. Néanmoins, nous notons que les valeurs extrêmes sont des artefacts qui peuvent être imputés à certains traitements du système perturbant notre mesure, ce que confirme le faible écart-type.



(a) Temps de traitement d'un RPC

(b) Dispersions des mesures

FIG. 7.4 – Mesures sur la plate-forme Solaris, cas d'un appel ONEWAY. (valeurs en s)

Cette seconde configuration utilise aussi une connexion TCP/IP. Nous mesurons le temps requis pour émettre une requête. Nous ne mesurons ainsi que le temps de traversée des différentes couches de l'intergiciel, de la personnalité CORBA jusqu'aux sockets. La valeur moyenne est de $645\mu s$, l'écart-type de $9.54898E-05$, les valeurs sont contenues dans une fenêtre de $2ms$ de large. Ici encore, plusieurs artefacts dénotent une perturbation du système externe.

Ces valeurs nous permettent de comparer respectivement le temps d'un aller/retour et le temps d'une traversée de l'intergiciel. Ceci nous montre que la synchronisation entre deux nœuds et le temps de traversée des couche de transport est important.

En effet, le temps de communication d'une trame ethernet est de $1492 * 8/10^8 = 119.36\mu s$, soit près de 13% du temps de traversée de l'intergiciel. Si on ajoute à cela le temps de nécessaire au traitement des couches protocolaires (contrôle de flux TCP/IP, envoi des tampons mémoire, etc).

On aboutit alors à la conclusion qu'une part importante du traitement d'une requête peut être occupée par les couches protocolaires. Ainsi, une source d'une latence importante peut être la plate-forme d'exécution utilisée et non à l'intergiciel.

Ainsi, comme nous le supposions à la section 1.1.2, l'impact de la latence du canal de communication est fort.

Impact de la configuration sur le déterminisme

Nous présentons figure 7.5 les mesures effectuées sous Solaris pour trois configurations différentes d'une application CORBA :
- *local* : client et serveur sont dans un même processus,
- *distribué, mono-tâche* : client et serveur sont deux processus distants, le profil mono-tâche est sélectionné,
- *distribué, multi-tâche* : client et serveur sont deux processus distants, le profil de concurrence complet est utilisé.

Ces trois configurations nous permettent de comparer l'impact de chaque fonctionnalité sur le déterminisme du système. Pour chaque configuration, nous mesurons le temps nécessaire pour un appel à la fonction `Round_Trip`.

Pour fournir un point de comparaison entre les différentes mesures effectuées, nous utilisons la définition suivante du vecteur statistique normalisé :

Définition 7.2.1. *Soit* $X = (X_i)_{i \in 1...N}$ *un vecteur statistique, alors le vecteur normalisé* $Y = (Y_i)_{i \in 1...N}$ *associé est tel que*

$$\forall i \in \{1 \ldots N\}, Y_i = \frac{(X_i - E[X])}{\sigma_x} \tag{7.2}$$

Le vecteur normalisé ramène les différentes mesures effectuées à un même intervalle. Ceci facilite la comparaison de la dispersion des mesures effectuées

Nous notons que les résultats obtenus sont proches, quelque soit la configuration utilisée. Une étude de la dispersion montre que tous les échantillons sont dans un intervalle large de $100\mu s$, représentant moins de 10% de la durée totale du temps de traitement de la requête. Elle s'explique en grande partie par l'exécutif utilisé : latence de la pile TCP/IP, non-déterminisme de l'allocateur mémoire et des primitives de concurrence.

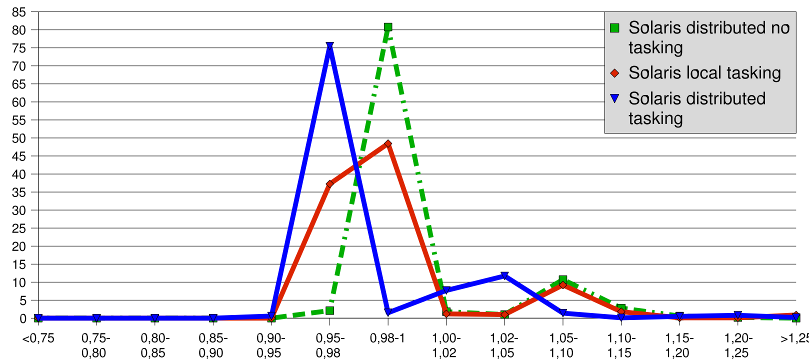


FIG. 7.5 – Mesures de performance sous Solaris

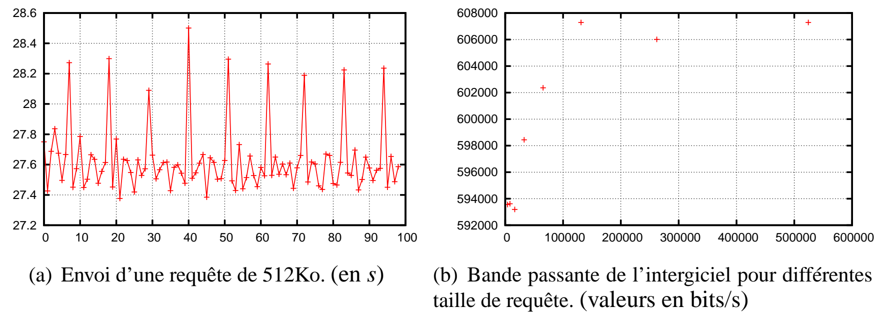
Bande passante de l'intergiciel

Dans ce test, nous nous intéressons à l'impact de la taille d'une requête sur le comportement de l'intergiciel. Nous mesurons pour différentes tailles du paramètre `Payload` le temps requis pour un appel à `Round_Trip_With_Payload`.

La figure 7.6(a) le temps de traitement pour une requête contenant $512Ko$ de données. Nous notons une forte variabilité de la valeur mesurée : $1.2s$, représentant 4% de la valeur moyenne. Ce temps peut s'expliquer par la latence de l'allocation dynamique de mémoire de Solaris.

Nous notons sur les mesures présentées à la figure 7.6(b) que cette variation est distribuée pour les différentes tailles de requêtes que nous avons considérées. La valeur moyenne du temps de parcours de l'intergiciel garantit une bande passante comprise entre $593Kbit/s$ et $608Kbit/s$. Ainsi, la bande passante varie peu avec la taille de la requête à construire, ceci garantit l'envoi d'une requête en un temps proportionnel à la taille des données à envoyer.

Une requête GIOP minimale est de l'ordre de 60 octets, nous déduisons que l'intergiciel peut traiter près de 1200 requêtes RT-CORBA/s.



(a) Envoi d'une requête de 512Ko. (en s)

(b) Bande passante de l'intergiciel pour différentes taille de requête. (valeurs en bits/s)

FIG. 7.6 – Mesures de la bande passante de l'intergiciel.

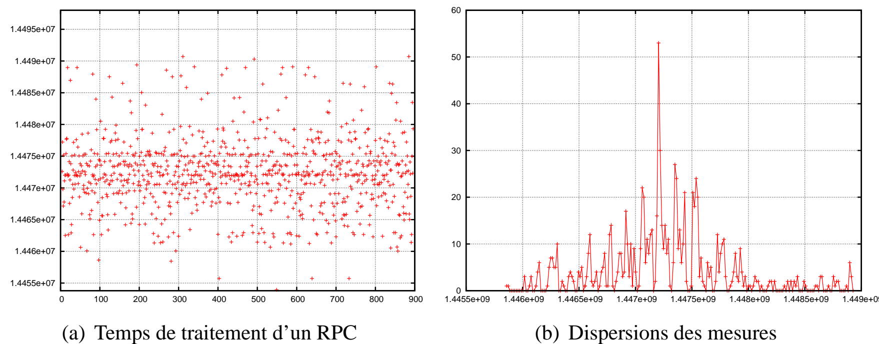
7.2.4 Mesures sur un noyau temps réel

Dans cette section, nous nous intéressons au comportement de l'intergiciel déployé avec un noyau temps réel. Nous évaluons sa latence en "boucle locale", et la comparons à celle d'un protocole classique.

Mesures locales

Nous avons mesuré le dispersion du temps de traitement d'un appel distant en utilisant un protocole de type *"boucle locale"* qui simule une communication synchrone FIFO entre deux processus. Ceci nous permet de mesurer le temps de traitement complet d'une requête sans perturbation de l'environnement extérieur. Les résultats sont résumés figure 7.7.

Nous mesurons une valeur moyenne de $14.472ms$. Une étude de la dispersion des valeurs montrent que tous les échantillons sont dans un intervalle de $340\mu s$ de large, représentant 2.35% du temps de traitement total d'une requête.



(a) Temps de traitement d'un RPC

(b) Dispersions des mesures

FIG. 7.7 – Mesures sur la plate-forme ORK. (valeurs en ns)

Nous avons ensuite porté ce test vers le noyau MaRTE OS. Nous mesurons alors une valeur moyenne de $19.39ms$. Tous les échantillons sont dans un intervalle de $274\mu s$ de large, représentant moins de 1.44% du temps de traitement total d'une requête. L'amélioration de la dispersion peut s'expliquer par l'utilisation d'un allocateur mémoire déterministe mis en œuvre par MaRTE OS.

Une dernière série de mesures sur Solaris a montré l'impact de l'exécutif sur la latence du système, les valeurs moyennes et l'écart mesurés sont équivalents, $2ms$. Ceci montre l'impact de l'ordonnanceur du système pour éviter une dérive de la latence en cas de forte charge sur le nœudtraitant les requêtes.

Impact d'un protocole

Indépendamment de l'intergiciel, nous nous intéressons ici au coût d'un protocole de communication de manière à pouvoir comparer la latence à l'intérieur d'un nœud et entres nœuds d'une application.

Nous avons dans un premier temps considéré l'implantation du protocole TCP/IP fournie par RTEMS. RTEMS (*"Real-Time Operating System for Multiprocessor Systems"*) est un noyau temps réel complet qui supporte une large variété de systèmes et interfaces de programmation, donc POSIX 1003.1b. Il fournit par ailleurs une pile TCP/IP complète, issue de FreeBSD. Il utilise les outils de GCC, dont GNAT pour sa chaîne de production. Nous avons utilisé RTEMS 4.6.2, et la même plate-forme matérielle que pour ORK.

Nous avons mesuré le temps pris par un cycle émission/réception de paquets TCP/IP de 500 octets. Nous avons mesuré une valeur moyenne de $300\mu s$, et une dispersion importante, égale à $420\mu s$.

En comparaison, le protocole TDMA que nous avons présenté au chapitre 6 permet de borner le délai d'émission de manière déterministe. Nous avons évalué ce délai en tenant compte de la latence introduite par les différentes couches matérielles et logicielles à quelques centaines de μs.

Ces deux mesures montrent que le protocole de communication entre nœuds induit une latence équivalente voire supérieur à celle constatée sur un nœud local.

7.2.5 Analyse critique

Cette série de mesures sur différentes plates-formes d'exécution fournit une indication quantitative du comportement temporel de notre intergiciel.

La faible dispersion des valeurs mesurées nous indique que l'intergiciel a un comportement globalement déterministe. Nos tests ont mis en évidence l'influence de certains paramètres externes à l'intergiciel qui sont hérités de la plate-forme de test : allocateur mémoire, primitive de concurrence, couche de transport.

Nous notons cependant que même une plate-forme disposant d'un processeur à 333 ou 350Mhz, la dispersion mesurée est de l'ordre de quelques pourcents de la valeur moyenne. Cette valeur est intéressante et démontre le fort déterminisme de notre solution. A titre de comparaison, cette écart atteint 10% pour RT-Zen [Krishna *et al.*, 2004]. Notre solution fournit des performances analogues à TAO pour cette première phase d'analyse.

Les mesures réalisées sur un noyau temps réel montrent que ces derniers permettent de renforcer les propriétés de déterminisme de l'intergiciel. Ainsi, la variance relevée pour ORK et MaRTE OS est suffisamment faible pour garantir de bonnes propriétés à l'application et effectuer une analyse d'ordonnancement fine. La mesure séparée sur le

protocole de communication permet d'évaluer l'impact de chacun des composants sur le comportement temporel de l'application globale.

7.3 Empreinte mémoire

Nous nous intéressons dans cette section à l'empreinte mémoire d'un nœud PolyORB. En particulier, nous cherchons à évaluer le coût des différentes fonctions et services offerts par l'intergiciel.

Nous avons défini à la section 6.2 la configuration d'un intergiciel minimal bâti au dessus de PolyORB. Partant de ce nœud minimal, nous analysons l'empreinte mémoire de différentes configurations, et présentons le coût associé à l'ajout de nouveaux mécanismes dans l'intergiciel.

Ces travaux ont été mené dans le cadre d'un stage de Master Recherche que nous avons co-encadré [Alves, 2005].

7.3.1 Mesures

Les différentes mesures présentées ici ont été obtenues en analysant la taille de l'exécutable construit pour la configuration considérée à l'aide de l'utilitaire `size`.

Pour chacun, les options de compilation retenues sont celles utilisées par PolyORB en configuration "de production", c'est à dire optimisation en temps et en espace et désactivation de nombreux mécanismes de contrôle opérant à l'exécution (vérification de la taille de la pile, tests associés aux assertions, calcul et affichage d'information de mise au point, ...). Le compilateur utilisé est GNAT Pro 5.02a1, pour plate-forme GNU/Linux x86.

Nous avons mesuré les exemples disponibles dans le dépôt de PolyORB. Ceux-ci incluent du code *"utilisateur"* pour l'affichage d'informations, l'initialisation, etc. Ils sont donc proches d'applications finales.

Nous avons considéré trois configurations :

1. *minimale* : l'exécutable basé sur la configuration minimale définie à la section 6.2, et différentes variantes incluant les mécanismes de concurrence (`no_tasking` et `thread_pool`), et remplaçant le `Simple Object Adapter` par un `Portable Object Adapter` (`*_poa`).

text	data	bss	dec	filename
553301	143473	30628	727402	polyorb-test-no_tasking
674245	174729	38404	887378	polyorb-test-no_tasking_poa
637663	160361	45988	844012	polyorb-test-thread_pool
758607	191617	53764	1003988	polyorb-test-thread_pool_poa

   Une analyse des différents fichiers objets nous indique que l'exécutif Ada 95 représente environs $300Ko$. Nous notons par ailleurs que le coût de la concurrence est d'environs $114Ko$, celui du `POA` de $156Ko$.

2. *CORBA/All Functions* : nœuds client et serveur utilisant les personnalités CORBA et IIOP. Cette configuration n'utilise qu'un type de données, mais teste les différentes politiques de passage de paramètres définis par CORBA. Elle utilise les mécanismes de concurrence complets de PolyORB.

text	data	bss	dec	filename
1398295	340097	67012	1805404	all_functions/client
2093232	492901	99940	2686073	all_functions/server

Ces deux exécutables montrent que le poids des personnalités CORBA et GIOP sur un exécutable est de l'ordre de $1Mo$. La taille d'un client CORBA de PolyORB est de l'ordre de $1,72Mo$. Cette valeur est comparable à d'autres intergiciels tels que TAO[1].

3. *CORBA/All Types* : nœuds client et serveur utilisant les personnalités CORBA et IIOP. Cette configuration utilise tous les types définis par le langage IDL. Elle utilise le profil mono-tâche de PolyORB, aucune construction de concurrence n'est présente.

text	data	bss	dec	filename
1770047	415261	70980	2256288	all_types/client
2434045	562921	90916	3087882	all_types/server

Ces deux exemples tirent partie des nombreuses constructions de types fournies par CORBA, ceci a un impact attendu et mesuré sur la taille de l'exécutable qui dépasse $2,15Mo$, voire $3Mo$ pour le serveur.

7.3.2 Optimisations

La réduction de la taille de l'exécutable peut être effectuée en sélectionnant de manière précise les modules utilisés, par exemple les personnalités protocolaires. Néanmoins, cette approche est limitée par la granularité de l'architecture déployée.

Un second jeu d'optimisations peut être défini afin de *"tailler sur mesure"* l'intergiciel en supprimant les portions de codes inutiles, ou en n'ajoutant que le code nécessaire au bon fonctionnement de l'application.

Une approche visant à n'ajouter que les portions de code nécessaires à l'application nécessite une modélisation précise de l'intergiciel et des outils sophistiqués pour construire le code associé à un assemblage, par exemple en utilisant une méthodologie basée sur les aspects [Pawlak *et al.*, 2004] ou l'ingénierie des modèles. Cette approche sort du cadre de nos travaux.

Nous avons opté pour une approche visant à supprimer les entités non utilisées suivant deux axes : *1)* suppression des types non utilisés, *2)* suppression des symboles non référencés. Pour ce faire, nous avons utilisé une solution naïve basée sur les outils mis à notre disposition par la chaîne de production.

Ce travail est exploratoire et servira à terme à définir un processus d'optimisation de l'intergiciel. Il permet d'ores et déjà de tester la modularité de notre architecture.

Suppression de types non-utilisés

Les mesures effectuées sur le test all_types montrent que l'utilisation de types évolués augmente la taille de l'exécutable. Nous faisons aussi la remarque que, par construction, même si un nœud ne manipule qu'un nombre limité de types, l'intergiciel embarquera les fonctions permettant de gérer tous les types, d'où une surcharge importante et inutile en espace mémoire.

Les types sont manipulés à deux niveaux dans PolyORB. `PolyORB.Any` fournit des fonctions de manipulations indépendantes de toute personnalité. La fonction de *Representation* fournit une fonction de conversion spécifique à une personnalité protocolaire.

Connaissant les types utilisés par une application, nous proposons d'adapter ces deux paquetages en modifiant les portions de code associées à des types non utilisés et en les remplaçant par des levées d'exception.

[1] `http://www.dre.vanderbilt.edu/Stats/` rassemble des mesures sur différents aspects de TAO.

Pour ce faire, une version annotée des fichiers sources de ces paquetages a été créé. Les annotations définissent le rôle de chaque portion de code vis à vis des types de données. Il s'agit en fait de directives compatibles avec le pré-processeur gnatprep.

Un exécutable prends en charge de construire à partir de ces annotations et de la liste des types nécessaires à l'application une version réduite des paquetages `PolyORB.Any` et de la fonction de *Representation* CDR.

La détermination des types utilisés par une application découle de la connaissance des personnalités utilisées et des messages échangés. Dans le cas d'une application CORBA, cette information peut être déduite par analyse des fichiers IDL et des paquetages de la personnalité CORBA utilisés. Des outils ont été mis au point pour extraire cette information automatiquement.

Ce faisant, nous réduisons la taille du code objet généré et garantissons la compatibilité des interfaces de ces deux paquetages. Nous présentons ici le gain obtenu :

text	data	bss	dec	filename	gain
435345	119413	29188	583946	polyorb-test-no_tasking	16%
557105	150757	36964	744826	polyorb-test-no_tasking_poa	16%
520675	136441	44548	701664	polyorb-test-thread_pool	17%
641651	167697	52324	861672	polyorb-test-thread_pool_poa	14%
1142243	271929	65860	1480032	all_functions/client	6%
2006572	475293	98788	2580653	all_functions/server	4%
1742775	409025	70884	2222684	all_types/client	1%
2406773	556685	90820	3054278	all_types/server	1%

Les gains relevés sont significatifs : l'exemple all_functions qui utilise peu de types a un gain de l'ordre de 6%, soit environ $100Ko$. Il en va de même pour les tests basés sur la configuration minimale. Ils démontrent l'intérêt de cette approche pour des applications n'utilisant qu'un sous-ensemble des types disponibles.

Suppression de symboles non référencés

Supprimer du code connu comme non utilisé ne couvre qu'une partie du problème de l'élimination du code inutile à une application.

L'utilisation du compilateur GNAT nous assure au niveau source que seuls les paquetages utiles à la compilation d'une unité sont effectivement incorporés. Ce premier niveau nous permet d'assurer qu'aucun paquetage "mort" n'est inclus. Cependant, ces techniques ne fournissent des résultats qu'à une forte granularité. Certaines fonctions d'une interface peuvent n'être jamais utilisées pour une application particulière. Cependant, il est difficile de supprimer ces fonctions manuellement.

Une seconde approche d'optimisation consiste à utiliser des outils de réduction du code compilé. Nous étudions l'impact de l'outil `gnatelim` sur l'empreinte mémoire.

`gnatelim` fait partie de la suite d'outils GNAT Pro. Il s'agit d'un outil permettant de réduire la taille d'un exécutable. Il procède en déterminant la liste des fonctions et procédures qui sont effectivement utilisées par une application. Il en déduit la liste des symboles non utilisés. Connaissant cette liste, on peut alors recompiler l'application en fournissant en paramètre la liste des fonctions à supprimer. L'exécutable ainsi construit est de taille inférieure à l'exécutable initial.

Nous avons testé l'impact de `gnatelim` sur les exécutables précédents, une fois les types inutiles supprimés. Le gain que nous présentons ici est par rapport à l'application test initiale.

text	data	bss	dec	filename	gain
380601	104093	29188	513882	polyorb-test-no_tasking	29%
490057	130809	36964	657830	polyorb-test-no_tasking_poa	26%
465139	120889	44548	630576	polyorb-test-thread_pool	25%
572515	147317	52324	772156	polyorb-test-thread_pool_poa	23%
1142243	271929	65860	1480032	all_functions/client	18%
1817844	415709	98788	2332341	all_functions/server	13%
1525511	348321	70884	1944716	all_types/client	14%
2164949	485013	90824	2740786	all_types/server	11%

Nous notons que la combinaison des deux optimisations proposées permet de réduire significativement la taille de l'application construite. Le gain oscille entre 215*Ko* et 305*Ko*, soit entre 11% et 29% de l'application. Un tel gain est appréciable, en particulier pour les plates-formes embarquées.

Cette phase de réduction de code est exploratoire. Elle repose sur une analyse des exécutables générés, et ne s'intéresse qu'à une facette de l'intergiciel. Elle fournit cependant des résultats encourageants.

7.3.3 Conclusion

Dans cette section, nous avons analysé l'empreinte mémoire de plusieurs applications construites autour de PolyORB. Nous avons montré que la taille des exécutables CORBA générés était similaire à d'autres projets tels que TAO ou omniORB.

Il est aussi possible de construire des exécutables de faible taille pour les cibles embarqués. La taille de l'exécutable complet est ici aussi compatible avec les ordres de grandeurs de ce type de projets.

Nous avons ensuite proposé deux optimisations de l'empreinte mémoire possibles. Elles reposent sur la suppression des entités non utilisées : réécriture de certains paquetages de PolyORB, utilisation de `gnatelim`. Nos tests montrent que le gain atteint est significatif, représentant plus de 200*Ko*.

Cette approche de réduction répond ainsi à nos attentes. Les valeurs atteintes démontrent que PolyORB peut être intégré à un nœud léger (empreinte inférieure à 500Ko), mais aussi fournir une sémantique plus riche supportant des spécifications étendues comme RT-CORBA. Ainsi, l'architecture proposée démontre qu'elle peut répondre à une large classe de besoins.

7.4 Conclusion

Nous avons procédé dans ce chapitre à une analyse de l'implantation de notre solution. Cette validation a posteriori nous permet de confirmer par l'expérience certaines des propriétés de PolyORB.

Dans un premier temps, nous avons défini les différentes plates-formes de test utilisées. La portabilité du langage Ada 95 et la disponibilité de plusieurs noyaux temps réel nous permet de définir une palette de plate-forme de test. Chaque noyau définit un profil particulier pour le temps réel. Cette variété nous permet de valider l'adéquation de PolyORB aux contraintes de l'hôte.

Nous avons ensuite défini un jeu de test capable d'évaluer le déterminisme d'applications construites autour de RT-CORBA. Nous avons repris les différents tests proposés par le "Distributed Hartstone Benchmark" et nous les avons adaptés au cas de

RT-CORBA. L'exécution de ce jeu de tests sur différentes plates-formes nous a permis de valider le déterminisme de notre implantation.

Enfin, nous nous sommes intéressés à l'empreinte mémoire d'un nœud PolyORB. Nous montrons que la taille des exécutables est proche de celle observée pour d'autres intergiciels. Différentes méthodes permettant de réduire cette empreinte sont envisageables et mises en œuvre dans différents projets. Nous avons proposé une solution basée sur l'élimination sélective de code source non utilisé. Les mesures ont montré que cette approche est pertinente et permet de réduire significativement la taille de l'exécutable ainsi généré.

Ainsi, nous complétons la construction d'intergiciels temps réel adaptés aux besoins de l'application : l'architecture adaptable que nous proposons nous permet de garantir le déterminisme et la faible consommation de ressources de l'intergiciel.

8

Conclusion et perspectives

Sommaire

8.1 Intergiciels pour systèmes TR^2E 159
8.2 Réalisations 160
8.3 Perspectives 161

8.1 Intergiciels pour systèmes TR^2E

Nous avons fait la constatation initiale que l'utilisation d'un intergiciel dans le cadre de la construction d'une application *"Temps Réel Répartie Embarquée"* (TR^2E) levait de nombreux problèmes, dont certains restaient non résolus.

En particulier, l'intergiciel voit son rôle inversé : de couche d'abstraction masquant les problématiques de répartition et de qualité de service, l'intergiciel devient un élément intégré à l'application qui doit définir précisément son comportement et ses propriétés. L'intergiciel doit favoriser l'analyse de l'application pour laquelle il assure le support de la répartition. Parallèlement, l'intergiciel devient un *"COTS"*, un composant pris sur l'étagère qui doit pouvoir être facilement adapté aux besoins de l'application.

Nous avons effectué une analyse des différentes solutions existantes et avons montré que chacune ne résolvait qu'une partie des problèmes posés. Trois axes de développement ont ainsi été mis en évidence : *1)* une approche "ingénierie temps réel" qui s'intéresse à l'empreinte mémoire et au déterminisme de l'intergiciel ; *2)* une approche "génie logiciel" qui privilégie la mise en œuvre de mécanismes de répartition configurables ; *3)* l'utilisation des méthodes de conception haut-niveau, ou les méthodes formelles, qui permettent de valider les propriétés d'un modèle d'une application.

Les solutions évoquées dans l'état de l'art ne fournissent qu'une solution partielle. Elles échouent à fournir une solution ou suffisamment adaptable ou suffisamment analysable. Il est alors difficile d'intégrer un intergiciel à la construction de systèmes TR^2E.

Cependant, nous notons que ces trois axes sont nécessaires à la construction d'intergiciels pour systèmes TR^2E, qui soient adaptables aux nombreux besoins posés par les concepteurs d'application.

8.2 Réalisations

Une analyse des architectures d'intergiciel configurables, génériques et schizophrènes nous a permis d'isoler certains patrons de conception et fonctions communes à plusieurs intergiciels, ainsi que les mécanismes d'adaptation qu'ils proposent.

Nous avons analysé l'intergiciel schizophrène PolyORB et montré qu'il se prête bien à une adaptation de ces mécanismes. Il permet le support de plusieurs modèles de répartition, protocoles, mécanismes de transport, etc. Nous avons retenu son architecture modulaire pour proposer des améliorations à même de supporter les besoins et contraintes des systèmes TR^2E. Nous avons ainsi proposé plusieurs contributions à la construction d'intergiciels pour systèmes TR^2E :

- Nous avons noté une inadéquation entre la variété des besoins énoncés au chapitre 1 et les solutions existantes (chapitre 2). Cette inadéquation forme ce que nous appelons la "crise de l'intergiciel".
  Nous avons proposé au chapitre 3 un noyau d'intergiciel minimal bâti autour d'un noyau d'intergiciel, le μBroker, et des services proposés par l'architecture d'intergiciel schizophrène. Nous montrons que ce noyau favorise la construction d'une famille d'intergiciels répondant aux besoins hétérogènes des systèmes TR^2E.
- Le μBroker a une architecture modulaire supportant de nombreuses politiques de configuration et implante les patrons de conception que nous avons relevés lors de notre état de l'art. Ce noyau d'intergiciel est suffisamment versatile pour supporter plusieurs modèles de répartition ;
  Nous avons validé la propriété d'adaptabilité de l'architecture proposée et montré que ce noyau peut être utilisé dans une architecture plus large pour construire de nombreux intergiciels, supportant plusieurs modèles de répartition (objets répartis, RPC, passage de messages, etc). Nous vérifions la forte réutilisation de code permise par l'architecture, ainsi que les nombreuses configurations qui peuvent être construites ;
- Les besoins hétérogènes des systèmes TR^2E imposent de pouvoir adapter l'intergiciel. Nous avons ainsi proposé des éléments de méthodologie permettant d'adapter notre architecture à différents besoins : configuration d'intergiciel minimale, construction d'intergiciels pour applications temps réel complets supportant des normes industrielles. Nous démontrons ainsi l'apport de notre solution pour la construction d'intergiciels ;
- Nous avons mis en évidence la difficulté de garantir les propriétés de bon fonctionnement de l'intergiciel (absence de famine, d'interblocage, etc). Nous avons extrait un modèle complet de l'intergiciel et proposé des structures de données déterministes permettent l'analyse de notre solution ;
- Nous avons montré au chapitre 4 que cette architecture est vérifiable et nous avons vérifié formellement que notre solution est exempte d'interblocage, de famine et nous avons proposé une condition assurant le dimensionnement correct des ressources. La modélisation et la vérification ont été effectuées pour des configurations significatives de l'intergiciel ;
- Conjointement à notre travail de modélisation, nous avons déduit de cette architecture une implantation qui est compatible avec les méthodes de développement des systèmes temps réel : utilisation du langage Ada 95, du profil Ravenscar, de structures de données déterministes (chapitre 5) ;
- Nous avons validé au chapitre 6 notre architecture et son implantation sur plusieurs exemples : RT-CORBA, protocole temps réel et intergiciel minimal pour

les systèmes embarqués ;
– Plusieurs tests (chapitre 7) ont confirmé le déterminisme de la solution proposée. Par ailleurs, nous avons montré que l'empreinte mémoire d'un nœud peut être rendue faible. Nous avons par ailleurs discuté des optimisations possibles pour réduire d'avantage la consommation de la mémoire.

Ces différentes contributions ont été intégrées à PolyORB, validées et diffusées.

Ainsi, le projet PolyORB a acquis une dimension nouvelle et est maintenant largement diffusé. Nous avons participé à la diffusion de plusieurs versions stables de PolyORB ; PolyORB a rejoint le consortium ObjectWeb[1] et AdaCore poursuit le support de PolyORB auprès de ses clients.

8.3 Perspectives

Nos différents travaux permettent d'ores et déjà de construire des applications réparties temps réel. Les différents mécanismes proposés et leurs compositions au sein d'une architecture vérifiée fournissent des garanties fortes quant au fonctionnement correct de l'application.

Ces résultats nous permettent d'anticiper des développements futurs, dont certains sont déjà en cours de réalisation.

Les différentes contributions que nous avons apportées ouvrent la voie à de nouveaux développements de l' "ingénierie des intergiciels" et permettent notamment de fournir de nouveaux outils et méthodes de haut niveau pour la construction d'applications réparties pour systèmes TR^2E : fabrique d'intergiciels, aide au déploiement et à la configuration, analyse fine des propriétés de l'intergiciel, définition d'une plate-forme de répartition pour les systèmes critiques.

"Fabrique" d'intergiciels

Nous avons défini un ensemble de briques intergicielles adaptables, ainsi qu'un guide méthodologique permettant de les combiner pour construire des instances d'intergiciel adaptées.

Nous avons ainsi proposé un processus et les éléments de base permettant de construire un intergiciel : briques intergicielles, méthodologie de composition, optimisation de la taille de l'exécutable généré.

Ceci ouvre naturellement la voie à une phase "d'industrialisation" et à la définition d'une fabrique d'intergiciel. Cette fabrique a pour but la mise en œuvre guidée et précise des différents éléments de méthodologie que nous avons définis pour aider à la construction d'intergiciels supportant de nouveaux modèles de répartition ou de nouvelles politiques de qualité de service.

Ainsi définie, cette fabrique fournirait une réponse adaptée aux besoins récurrents en des intergiciels adaptables répondant à des contraintes évoluant vite et pour un coût de développement faible. En particulier, une telle fabrique résorberait la "crise des intergiciels" que nous avons constatée lors de notre étude initiale.

Déploiement et configuration automatisés d'applications TR^2E

La phase de configuration d'un intergiciel pour une application et le déploiement de cette application sont deux étapes complexes. L'utilisation d'un langage de descrip-

[1] `http://polyorb.objectweb.org`

tion d'architectures (*ADL, "Architecture Description Language"*) permet de décrire les différents éléments constitutifs d'un intergiciel (les entités formant son modèle de répartition) et fournit un cadre descriptif complet pour décrire l'application.

Le concepteur d'application peut alors prendre appui sur ce canevas et définir plus précisément ses propres briques. Des outils d'analyse assurent ensuite la cohérence de la description complète et aident au déploiement de l'application ainsi définie.

Des travaux sont actuellement menés au laboratoire INFRES de l'École Nationale Supérieure des Télécommunications pour fournir une suite d'outils aidant à la configuration et au déploiement d'applications réparties. Le langage AADL (*"Architecture, Analysis and Description Language"*) [Feiler *et al.*, 2003] est utilisé et sert de fil conducteur pour la définition de l'application au dessus des composants de l'intergiciel, sa projection sur les différents nœuds et la configuration locale de l'intergiciel. Ce travail vise à faciliter la mise en œuvre des technologies intergicielles pour la construction d'applications critiques [Vergnaud *et al.*, 2005].

Analyse temporelle et ordonnancement

Les différents paramètres de configuration que nous avons introduits ont un impact mesurable sur le comportement de l'intergiciel : chacun a été défini par un stéréotype analysable (filtres, automates, etc). Cependant, la multiplicité des configurations induit une multiplicité des instances d'intergiciel possibles.

Nous avons montré au chapitre 4 comment vérifier formellement les propriétés causales de l'architecture que nous proposons. En particulier, nous avons défini un processus de modélisation permettant de décrire fidèlement l'implantation réalisée.

Suivant un processus analogue, nous pouvons étendre cette analyse aux propriétés temporelles de l'intergiciel et offrir de nouvelles garanties nécessaires à la construction de systèmes TR^2E à même de respecter leurs échéances temporelles de bout en bout.

Construction d'intergiciels à base de preuves

Parallèlement à la construction guidée d'intergiciel, il est nécessaire de garantir les propriétés de la solution construite tout au long de son cycle de vie : de la phase amont de conception jusqu'à la phase de déploiement et d'exécution.

Le projet européen ASSERT a pour objectif la définition d'une méthodologie pour la construction d'applications critiques où chaque étape de conception, implantation, déploiement, etc. garantirait les propriétés attendues à l'étape précédente, mais aussi de nouvelles propriétés compatibles. Cette garantie doit être atteinte par des preuves précises et détaillées des propriétés du système.

Dans ce contexte, nos travaux autour de PolyORB fournissent une base de travail qui est actuellement analysée et utilisée pour la conception d'un premier prototype. En particulier, la notion de tolérance aux pannes et de consensus ont été récemment ajoutées à PolyORB.

Intergiciels pour systèmes critiques

Comme nous l'avons montré, de nombreux résultats existent pour permettre la construction de systèmes TR^2E. Un sous-ensemble de ces applications, dites *"critiques"*, pose des contraintes fortes en terme d'ingénierie.

En plus des contraintes que nous avons déjà discutées, elles nécessitent une analyse précise du système construit à des fins de certification. Cet aspect est pour le moment peu abordé par les intergiciels, un développement ad hoc est préféré le plus souvent.

L'OMG définit à l'heure actuelle CORBA High Assurance [Haverkamp & Richards, 2002], il s'agit d'un profil restreint de CORBA qui vise la construction de systèmes répartis temps réel embarqués fiables, par exemple pour le domaine avionique.

Cette spécification est à l'état de *"Request for Proposal"*. Nous notons que les thèmes retenus par cette RFP rejoignent certaines de nos problématiques : environnement critiques, garanties fortes de bon fonctionnement, configurabilité de l'intergiciel, etc. Une perspective intéressante serait d'étendre l'architecture proposée pour prendre en compte les spécifications définies, ou participer à leur définition.

En particulier, notre analyse des spécifications RT-CORBA (section 6.3.3) et les résultats que nous avons obtenus concernant la vérification de propriétés cruciales pour le bon fonctionnement de l'intergiciel (chapitre 4) peuvent servir de base à la définition de ce profil. Nous entretenons des contacts avec les auteurs de cette RFP en ce sens.

Nos différents travaux ont étendu l'utilisation des intergiciels aux systèmes TR^2E. Ils proposent une solution souple pour l'adaptation de l'intergiciel à de nombreux besoins.

Nous avons fourni un moyen de contrôler deux forces antagonistes s'appliquant sur les intergiciels pour systèmes TR^2E : la première, *"centripète"* se concentre sur les besoins minimaux à remplir et réduit les fonctionnalités offertes ; la seconde, *"centrifuge"* vise au contraire à étendre les mécanismes fournis.

En conciliant ces deux aspects, nous contribuons à l'utilisation toujours croissante des mécanismes de la répartition à de nombreuses familles de systèmes temps réel.

Index

couche neutre, 47

DDS, 34, 140

exemple de code
- `PolyORB.ORB_Controller`, 75
- `PolyORB.Setup.Base`, 130
- Distributed Hardstone Benchmark, 149
- Exemple Ada 95, 101
- `PolyORB.Lanes`, 112
- `PolyORB.Errors`, 104
- `PolyORB.Tasking.Priorities`, 111
- `PolyORB.Request_QoS`, 109
- `PolyORB.Request_Scheduler`, 113

Fetch/Decode/Execute, 64

hartstone, 145

intergiciel, 15
- configurable, 42
- générique, 43
- schizophrène, 47

MaRTE OS, 144
MDA, 76
mesures
- plates-formes, 144
- protocole, 145

MOMA, 59

nœud minimal, 131

ORK, 144

performances
- Bande Passante, 148
- Round Trip Time, 146

personnalité, 43
- applicative, 47
- protocolaire, 47

PolyORB, 49

réseaux de Petri, 82
requête
- composants, 127
- traitement, 65

RT-CORBA, 133
RT-DSA, 138

services d'un intergiciel, 48
Solaris, 144
système
- embarqué, 18
- réparti, 17
- temps réel, 17

μBroker, 66

Bibliographie

Abdelzaher, T., Dawson, S., Feng, W., Jahanian, F., Johnson, S., Mehra, A., Mitton, T., Shaikh, A., Shin, K., Wang, Z. & Zou, H. (1997). ARMADA Middleware Suite. Tech. rep., Dept. of Electrical Engineering and Computer Science, University of Michigan.

Abdelzaher, T., Dawson, S., Feng, W.C., Jahanian, F., Johnson, S., Mehra, A., Mitton, T., Shaikh, A., Shin, K., Wang, Z., Zou, H., Bjorkland, M. & Marron, P. (1999). ARMADA Middleware and Communication Services. *Real-Time Syst.*, **16**, 127–153.

Abrial, J. (1995). *Z: an introduction to formal methods*. Cambridge University Press.

Alves, R. (2005). *Études des optimisations de l'intergiciel PolyORB*. Mémoire de Master SAR, ENST.

Apvrille, L., de Saqui-Sannes, P., Lohr, C., Sénac, P. & Courtiat, J.P. (2001). A New UML Profile for Real-time System : Formal Design and Validation. In *Proceeding of the Fourth International Conference on the Unified Modeling Language (UML'2001)*, Toronto, Canada.

Apvrille, L., de Saqui-Sannes, P. & Khendek, F. (2003). TURTLE-P: Un profil UML pour la validation d'architectures distribuees. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP)*, Hermes, Pparis, France.

Armstrong, S. *et al.* (1992). Multicast transport protocol. Tech. Rep. 1301.

Azavant, F., Cottin, J.M., Niebel, V., Pautet, L., Ponce, S., Quinot, T. & Tardieu, S. (1999). CORBA and CORBA Services for DSA. In *Proceedings of SIGAda'99*, acmpress.

Baarir, S., Haddad, S. & Ilié, J.M. (2004). Exploiting Partial Symmetries in Wellformed nets for the Reachability and the Linear Time Model Checking Problems. In *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France.

Baker, S. (2001). Middleware To Middleware. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*.

Banavar, G., Chandra, T., Strom, R. & Sturman, D. (1999). A case for message oriented middleware. In *International Symposium on Distributed Computing*, 1–18.

Bastide, R., Sy, O., Palanque, P. & Navarre, D. (2000). Formal Specifications of CORBA Services: Experience and Lessons Learned. In *Proceedings of the ACM Conference on Object-Oriented Programmng, Systems, Langu ages and Applications (00PSLA'2000)*, Minneapolis, Minnesota, USA.

Bechina, A., Brinkshulte, U., Picioroaga, F. & Schneider, E. (2001). Real time middleware for industrial embedded measurement and control. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications , PDPTA'2001, Las Vegas, USA*.

Bernstein, P.A. (1993). Middleware: An archictecture: for distributed system services. Tech. Rep. CRL 93/6, Cambridge MA (USA).

Bosch (1991). CAN specification, version 2.0.

Braden, R., Zhang, L., Berson, S., Herzog, S. & Jamin, S. (1997). RFC 2205 - Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification.

Buble, A., Bulej, L. & Tuma, P. (2003). CORBA Benchmarking: A Course With Hidden Obstacles. In *Proceedings of the IPDPS Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEOPDS 2003)*, 279 – 285, Nice, France.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal., M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley and Sons Ltd.

Campos, J.L., Gutiérrez, J.J. & Harbour, M.G. (2004). The Chance for Ada to Support Distribution and Real-Time in Embedded Systems. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, vol. LNCS 3063, 91–105, Springer Verlag, Palma de Mallorca, Spain.

Can, S.D. (2005). *Conception et réalisation de la personnalité DDS pour PolyORB*. Mémoire de Master SAR, ENST.

Carter, J. & Wegman, M.N. (1979). Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, **18**, 143–154.

Chenyang, L., Xiaorui, W. & Gill, C. (2003). Feedback controlreal-time scheduling in orb middleware. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Chiola, G., Dutheillet, C., Franceschini, G. & Haddad, S. (1991). On Well-Formed Coloured Nets and their Symbolic Reachability Graph. *High-Level Petri Nets. Theory and Application, LNCS*.

Clarke, E., Grumberg, O. & Peled, D. (2000). *Model Checking*. MIT Press.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. & Stein, C. (2002). *Introduction à l'algorithmique*. Dunod, 2nd edn.

Corsaro, A., Schmidt, D., Klefstad, R. & O'Ryan, C. (2002). Virtual component: a design pattern for memory-constrained embedded applications.

Coulouris, G., Dollimore, J. & Kindberg, T. (1994). *Distributed Systems, Concepts and Design*. Addison-Wesley, 2nd edn.

Coupaye, T., Bruneton, E. & Stefani, J. (2002). The fractal composition framework. Tech. rep., The ObjectWeb Group.

Curnow, H. & Wichmann, B. (1976). A Synthetic Benchmark. *The Computer Journal*, **19**, 43–49.

David, P.C. & Ledoux, T. (2003). Towards a Framework for Self-Adaptive Component-Based Applications. In *Proceedings of Distributed Applications and Interoperable Systems 2003, the 4th IFIP WG6.1 International Conference, DAIS 2003*, vol. 2893, 1–14, Lecture Notes in Computer Science.

de la Puente, J.A., Ruiz, J.F. & Zamorano, J. (2000). An Open Ravenscar Real-Time Kernel for GNAT. In *Proceedings of the Reliable Software Technologies. Ada-Europe 2000*, Lecture Notes in Computer Science, 1845.

Demeure, I. & Bonnet, C. (1999). *Introduction aux systèmes temps réels*. Hermes.

Deng, W., Dwyer, M.B., Hatcliff, J., Jung, G., Robby & Singh, G. (2003). Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*.

Departimento di Informatica, U.T. (2003). GreatSPN home page. `http://www.di.unito.it/~greatspn/index.html`.

Dibble, P. & Wellings, A. (2004). The real-time specification for Java: current status and future work. In *Proceedings of the IEEE Seventh International Symposium on Object-Oriented Real-Time Distributed Computing*, 71–77.

Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H. & Tarjan, R.E. (1994). Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, **23**, 738–761.

Diller, A. (1994). *The B-book*. John Willey & SONS.

DIT/UPM (2004). Open ORK Home Page. `http://polaris.dit.upm.es/~jpuente/rts/proyectos/ork/`.

Dobbing, B. & Burns, A. (1998). The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA.

Douglass, B.P. (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley.

Dumant, B., Horn, F., Tran, F.D. & Stefani, J.B. (1998). Jonathan: an Open Distributed Processing environment in Java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 175–190, Springer Verlag, Londres.

Duret-Lutz, A. & Poitrenaud, D. (2004). SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, 76–83, IEEE Computer Society Press, Volendam, The Netherlands.

Evangelista, S., Kaiser, C., Pradat-Peyre, J. & Rousseau, P. (2003). Quasar : a new tool for analyzing concurrent programs. In *Proceddings of the International Conference on Reliable Software Technologies, Ada-Europe (Ada-Europe'03)*, vol. LNCS 2655, 166–181, Springer Verlag.

Feiler, H., Lewis, B. & Vestal, S. (2003). The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*.

Ferrari, D. & Verma, D.C. (1990). A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, **8**, 368–379.

Francu, C. & Marsic, I. (1999). An Advanced Communication Toolkit for Implementing the Broker Pattern. In *Proceedings of ICDCS'99*, IEEE.

Fredman, M.L., Komlós, J. & Szemerédi, E. (1984). Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, **31**, 538 – 544.

Gai, P., Abeni, L., Giorgi, M. & Buttazzo, G. (2001). A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts.

Gibbs, W.W. (1991). Software"s Chronic crisis. 72 – 81.

Gill, C., Subramonian, V., Parsons, J., Huang, H.M., Torri, S., Niehaus, D. & Stuart., D. (2003a). Orb middleware evolution for networked embedded systems. In *Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS'03), Guadalajara, Mexico*.

Gill, C.D. (2003). A Vision for Integration of Embedded System Properties - via a Model-Component-Aspect System Architecture. In *Proceedings of the Monterey Workshop*, Chicago, IL.

Gill, C.D., Schmidt, D.C. & Cytron, R.K. (2003b). Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. **91**, 183 – 197.

Girault, C. & Valk, R. (2002). *Petri Nets for System Engineering*. Springer Verlag.

Gokhale, A., Natarjan, B., Schmidt, D.C., Nechypurenko, A., Wang, N., Gray, J., Neema, S., Bapty, T. & Parsons, J. (2002). CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA.

Gosling, J. & Bollella, G. (2000). *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co. Inc.

Halbwachs, N. (1993). A tutorial of Lustre.

Haverkamp, D.A. & Richards, R.J. (2002). Towards safety critical middleware for avionics applications. In *LCN*, 716–724, IEEE Computer Society.

Holzmann, G. (2004). *SPIN Model Checker, The: Primer and Reference Manual*. Addison Wesley Professional.

Hoosier, M., Hatcliff, J., Robby & Dwyer, M.B. (2004). A Case Study in Domain-customized Model Checking for Real-time Component Software.

Hugues, J. (2004). *PolyORB User's Guide*.

Hugues, J., Kordon, F., Pautet, L. & Quinot, T. (2002). A case study of Middleware to Middleware: MOM and ORB interoperability. In *Addendum to the proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 29–32, University of California at Irvine, Irvine, CA, USA.

Hugues, J., Pautet, L. & Kordon, F. (2003a). Contributions to middleware architectures to prototype distribution infrastructures. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, 124–131, IEEE, San Diego, CA, USA.

Hugues, J., Pautet, L. & Kordon, F. (2003b). Refining middleware functions for verification purpose. In *Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (MONTEREY'03)*, 79–87, Chicago, IL, USA.

Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S. & Vergnaud, T. (2004). On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, vol. ENTCS 133, 139 – 157, Elsevier, Linz, Austria.

Hugues, J., Kordon, F. & Pautet, L. (2005a). Revisiting COTS middleware for DRE systems. In *Proceedings of the 8th International Symposium on Object-oriented Real-time distributed Computing*, 72–79, Seattle, WA, USA.

Hugues, J., Kordon, F. & Pautet, L. (2005b). Towards Proof-Based Real-Time Distribution Middleware. In *Proceedings of the 13th International Conference On Real-Time Systems*, 51–70, BIRP, Paris, France.

ISO (1994). *Quality management and quality assurance - vocabulary*. ISO, ISO 8402:1994.

ISO (1995). *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO, ISO/IEC 7498-1:1994.

ISO (2000a). *Information Technology – Programming Languages – Ada*. ISO, iSO/IEC/ANSI 8652:1995(E) with Technical Corrigendum 1.

ISO (2000b). *Information Technology – Programming Languages – Guide for the use of the Ada programming language in high integrity systems*. ISO, ISO/IEC/TR 15942::2000(E).

ISO (2005). *Information Technology – Programming Languages – Ada*. ISO, iSO/IEC/ANSI 8652:1995(E) with Technical Corrigendum 1 and Amendment 1 (Draft 11).

Kaddour, M. & Pautet, L. (2003). Towards an adaptable message oriented middleware for mobile environments. In *Proceedings of the IEEE 3rd workshop on Applications and Services in Wireless Networks*, Berne, Switzerland.

Kaddour, M. & Pautet, L. (2004). A middleware for supporting disconnections and multi-network access in mobile environments. In *Proceedings of the Perware workshop at the 2nd Conference on Pervasive Computing (Percom)*, Orlando, Florida, USA.

Kameno, N.I. & Weiderman, N.H. (1991). Hartstone distributed benchmark: requirements and definitions. In I.C.S. Press, ed., *Proceedings of the 12th IEEE Real-Time Systems Symposium*, 199–208.

Kandlur, D.D., Shin, K.G. & Ferrari, D. (1991). Real-time communication in multihop networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, 300–307, IEEE Computer Society, Washington, DC.

Kermarrec, Y., Nana, L. & Pautet, L. (1996). GNATDIST: A configuration language for distributed ada 95 applications. In *Proceedings of TRI-Ada'96*, 63–72, acm-press.

Kim, K., Geon, G., Hong, S., Kim, S. & Kim, T. (2000). Resource-conscious customization of CORBA for CAN-based distributed embedded systems. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, 34–41.

Klefstad, R., Schmidt, D.C. & Ryan., C.O. (2002). The Design of a Real-Time CORBA ORB using Real-Time Java. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time distributed Computing*.

Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhes, L. & Campbell, R. (2000). Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).*, New York, New York, USA.

Koopman, P. & DeVale, J. (2000). The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Trans. Software Eng.*, **26**, 837–848.

Kordon, F. & Paviot-Adet, E. (1999). Using CPN-AMI to validate a safe channel protocol. In *Proceedings of the International Conference on Theory and Applications of Petri Nets - Tool presentation part*, Williamsburg, USA.

Krishna, A., Schmidt, D.C. & Klefstad, R. (2004). Enhancing Real-Time CORBA via Real-Time Java. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Tokyo, Japan.

LAAS (2004). The RT-LOTOS Project. `http://www.laas.fr/RT-LOTOS`.

Liu, C. & Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, **20**, 46–61.

Martínez, J.M., Harbour, M.G. & Gutiérrez, J.J. (2003). RT-EP: Real-Time Ethermet Protocol for Analyzable Distributed Applications on a Minimum Real-Time POSIX Kernel. In *2nd International Workshop on Real-Time LANs in the Internet Age (RT-LIA 2003)*, Porto, Portugal.

Masmano, M., Ripoll, I. & Crespo, A. (2003). Dynamic storage allocation for real-time embedded systems. In *Proccedings of the Real-Time Systems Symposium, Work-in-Progress Session*, Cancun, Mexico.

Matsumoto, M. & Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, **8**, 3–30.

Mercer, C., Ishikaw, Y. & Tokuda, H. (1990). Distributed Hartstone: a distributed real-time benchmark suite. In I.C.S. Press, ed., *Proceedings of the 10th International Conference on Distributed Computing Systems*, 70–77.

Merseguer, J., Campos, J., Bernardi, S. & Donatelli, S. (2002). A compositional semantics for UML state machines aimed at performance evaluation. In *Proceedings of the Sixth International Workshop on Discrete Event Systems*.

MISRA (2004). *"MISRA-C:2004 - Guidelines for the use of the C language in critical systems"*.

Obry, P. (2000). AWS: Ada Web Server. In *Actes d'Ada-France, session spéciale Ada et Internet*, Ada France.

ODP (1995). ODP Reference Model: overview. ITU-T -- ISO/IEC Recommendation X.901 -- International Standard 10746-1.

OMG (2002a). *Real-Time CORBA Specification, static scheduling, v1.1*. OMG, oMG Technical Document formal/2002-08-02.

OMG (2002b). *unreliable Multicast InterORB Protocol specification*. OMG, oMG Technical Document ptc/03-01-11.

OMG (2003a). *Overview and guide to OMG's architecture*. OMG, oMG Technical Document formal/03-06-01.

OMG (2003b). *Real-Time CORBA Specification, dynamic scheduling, v2.0*. OMG, oMG Technical Document formal/2003-11-01.

OMG (2004a). *Data Distribution Service: Architecture and Specification, revision 1.0*. OMG, OMG Technical Document.

OMG (2004b). *The Common Object Request Broker: Architecture and Specification, revision 3.0.3*. OMG, oMG Technical Document formal/2004-03-12.

O'Ryan, C., Kuhns, F., Schmidt, D.C., Othman, O. & Parsons, J. (2000). The design and performance of a pluggable protocols framework for real-time distributed object computing middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'00)*.

Paquet, B. (2003). *Implémentation de la personnalité MIOP dans PolyORB*. Mémoire de DEA, ENST.

Pautet, L. (2001). *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Habilitation à diriger des recherches, Université Pierre et Marie Curie – Paris VI.

Pautet, L. & Kordon, F. (2004). Des vertus de la schizophrénie pour le prototypage d'applications à composants interopérables. *TSI*, **23**.

Pautet, L. & Tardieu, S. (2000). *GLADE User Guide*. Ada Core Technologies.

Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F. & Florin, G. (2004). Aspect-Oriented Software Development with Java Aspect Components. In S. Clarke, B. Filman, T. Elrad & M. Aksit, eds., *Aspect-Oriented Software Development (AOSD)*.

Pedreiras, P., Almeida, L. & Gai, P. (2002). The FTT-Ethernet Protocol: Merging Flexibility,Timeliness and Efficiency. In *14 th Euromicro Conference on Real-Time Systems (ECRTS'02)*, Vienna, Austria.

Plummer, C. & Plancke, P. (2002). The Spacecraft Onboard Interfaces, SOIF, Standardisation Activity. In *Proceedings of DASIA 2002*, Dublic, Ireland.

Pyarali, I., Spivak, M., Cytron, R. & Schmidt, D.C. (2001). Evaluating and Optimizing Thread Pool Strategies for RT-CORBA. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, ACM.

Quinot, T. (2003). *Conception et réalisation d'un intergiciel schizophrène pour la mise en œuvre de systèmes répartis interopérables*. Thèse de doctorat, Université Pierre-et-Marie-Curie.

Quinot, T., Kordon, F. & Pautet, L. (2001). From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, IEEE Computer Society Press.

Rajkumar, R., Gagliardi, M. & Sha, L. (1995). The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings of the IEEE Real Time Technology and Applications Symposium 1995*, 66–75.

Regep, D. & Kordon, F. (2001). **L*f*P**: a specification language for rapid prototyping of concurrent systems. In *12th IEEE International Workshop on Rapid System Prototyping*.

RIS (2002). Intergiciel et sûrete de fonctionnement. Tech. rep., Réseau d'Ingénierie de la Sûreté de fonctionnement.

Rivas, M.A. & Harbour, M.G. (2001). MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In *Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001*, LNCS, Leuven, Belgium.

Román, M., Kon, F. & Campbell, R.H. (2001). Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, **2**.

SC-167 (1992). Software considerations in airborne systems and equipment certification. Advisory circular DO-178B, Radio Technical Commission for Aeronautics.

Schmidt, D. & Buschmann, F. (2003). Patterns frameworks and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineer ing*.

Schmidt, D. & Cleeland, C. (1997). Applying patterns to develop extensible and maintainable ORB middleware. *Communications of the ACM, CACM*, **40**.

Schmidt, D., Levine, D. & Mungee, S. (1998). The design and performance of real-time object request brokers.

Schmidt, D.C. (1990). Gperf: A perfect hash function generator. In *Proceedings of the 2 nd C++ Conference*, 87–102, San Francisco, California.

Schmidt, D.C. & Cleeland, C. (1998). Applying patterns to develop extensible orb middleware. *IEEE Communications Magazine Special Issue on Design Patterns*.

Schmidt, D.C. & Kuhns, F. (2000). An Overview of the Real-time CORBA Specification.

Schmidt, D.C., Stal, M., Rohnert, H. & Buschmann, F. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurren t and Networked Objects*, vol. 2. Wiley and Sons.

Schneider, E., Picioroaga, F. & Bï¿½hina, A. (2002). Distributed real-time computing for microcontrollers-the osa+ approach. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 169, IEEE Computer Society.

Seinturier, L., Laurent, A., Dumant, B., Gressier-Soudan, E. & Horn, F. (1999). A framework for a real-time communication based object oriented indutrial messaging services. Tech. rep., CNET, France Télécom-CNET.

Sharp, D.C. (1998). Reducing avionics software cost through component based product line. In *Proceedings of the 10th Annual Software Technology Conference*.

Singhai, A., Sane, A. & Campbell, R. (1998). Quarterware for Middleware. In *Proceedings of ICDCS'98*, IEEE.

Souissi, Y. & Memmi, G. (1989). Compositions of Nets via a communication medium. In *10th International Conference on Application and theory of Petri Nets*, Bonn, Germany.

Stefani, J.B. (1996). Requirements for a real-time ORB. Tech. Rep. RT/TR-96-8, CNET.

Subramonian, V., Gill, C. & Stuard, D. (2002). Design and implementation of norb.

Subramonian, V., Xing, G., Gill, C.D. & Cytron, R. (2003). The design and performance of special purpose middleware: A sensor networks case study. Technical report of the University of Washington Saint Louis.

Sun Microsystems, Inc. (1999). *Java Message Service (JMS) Specification*. Version 1.1.

Sun Microsystems, Inc. (2002). *Java Remote Method Invocation (RMI) Specification*. Version 1.4.

Tanenbaum, A.S. (1995). *Distributed Operating Systems*. Prentice Hall.

Tardieu, S. (1999). *GLADE, une implémentation de l'annexe des systèmes répartis d'Ada 95*. Thèse de doctorat, École nationale supérieure des télécommunications.

Thierry-Mieg, Y., Dutheillet, C. & Mounier, I. (2003). Automatic Symmetry Detection in Well-Formed Nets. In *Proc. of ICATPN 2003*, vol. 2679 of *Lecture Notes in Computer Science*, 82–101, Springer Verlag.

Thierry-Mieg, Y., Baarir, S., Duret-Lutz, A. & Kordon, F. (2004). Nouvelles techniques de model-checking pour la vérification de systèmes complexes. *Génie Logiciel*, 17–23.

US. Department of Defense (1978). *Requirements for High Order Computer Programming Languages, "Steelman"*.

Vanegas, R., Zinky, J., Loyall, J., Karr, D., Schantz, R. & Bakken, D. (1998). QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, England.

Vergnaud, T. (2003). *Conception et réalisation de la personnalité AWS pour PolyORB*. Mémoire de DEA, ENST.

Vergnaud, T., Hugues, J., Pautet, L. & Kordon, F. (2004). PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, vol. LNCS 3063, 106 – 119, Springer Verlag, Palma de Mallorca, Spain.

Vergnaud, T., Hugues, J., Pautet, L. & Kordon, F. (2005). Rapid Development Methodology for Customized Middleware. In *Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, 111–117, IEEE, Montreal, Canada.

Vinoski, S. (2004). An Overview of Middleware. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, vol. LNCS 3063, 35 – 51, Springer Verlag, Palma de Mallorca, Spain.

Weiderman, N. (1989). Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications. Tech. Rep. CMU/SEI-89-TR-023.

Wheeler, D.A. (1996). Ada, C, C++, and Java vs. The Steelman.

Wheeler, D.A. (2004). SLOCCount, a set of tools for counting physical source lines of code (sloc).